learnopengl

wiki Compute Shader

cs 的OpenGL core 4.5 规范

Compute Shaders

除了图形相关的着色操作，如顶点着色器、细分着色器、几何着色器和片段着色器，GL还可以通过使用计算着色器执行通用计算。计算管线是一种单级机器，运行通用着色器。计算着色器使用类型参数COMPUTE_SHADER，如第7.1节所述进行创建。它们通过第7.3节描述的方式附加到程序对象并在其中使用。

计算工作负载是由称为工作组的工作项组成的，并由计算程序的可执行代码进行处理。工作组是执行相同代码的着色器调用的集合，可能是并行的。工作组内的调用可以通过共享变量（参见OpenGL着色语言规范的第4.3.8节“共享变量”）与同一工作组的其他成员共享数据，并发出内存和控制屏障以与同一工作组的其他成员同步。通过调用以下方式启动一个或多个工作组：

```
void DispatchCompute(uint num_groups_x, uint num_groups_y, uint num_groups_z);
```

每个工作组由计算着色器阶段的活动程序对象处理。计算着色器阶段的活动程序将以与其他管线阶段的活动程序相同的方式确定，如第7.3节所述。虽然工作组内的个别着色器调用作为一个单元执行，但工作组是完全独立且以未指定的顺序执行的。

`num_groups_x`、`num_groups_y`和`num_groups_z`分别指定在X、Y和Z维度上将要调度的本地工作组的数量。内建矢量变量`gl_NumWorkGroups`将使用`num_groups_x`、`num_groups_y`和`num_groups_z`参数的内容进行初始化。可以通过调用`GetIntegeri v`，将目标设置为`MAX_COMPUTE_WORK_GROUP_COUNT`，将索引设置为零、一或两，分别表示X、Y和Z维度，来确定一次可以调度的最大工作组数量。如果在任何维度上的工作组计数为零，则不会调度任何工作组。

在编译时，可以使用计算着色器附加到程序的一个或多个中的输入布局限定符来指定每个维度的本地工作大小（参见OpenGL着色语言规范的第4.4.1.4节“计算着色器输入”）。程序链接后，可以通过调用`GetProgramiv`，将`pname`设置为`COMPUTE_WORK_GROUP_SIZE`，如第7.13节所述，来查询程序的本地工作组大小。

可以通过调用`GetIntegeri v`，将目标设置为`MAX_COMPUTE_WORK_GROUP_SIZE`，将索引设置为0、1或2，来确定本地工作组的最大大小，分别检索X、Y和Z维度的最大工作大小。此外，可以通过调用`GetIntegerv`，将`pname`设置为`MAX_COMPUTE_WORK_GROUP_INVOCATIONS`，来确定单个本地工作组中调用的最大数量（即三个维度的乘积）。

命令

```
void DispatchComputeIndirect(intptr indirect);
```

等效于使用当前绑定到DISPATCH_INDIRECT_BUFFER绑定的缓冲区中的偏移处指定的三个uint值初始化num groups x、num groups y和num groups z，单位为基本机器单元。如果num groups x、num groups y或num groups z中的任何一个大于相应维度的MAX_COMPUTE_WORK_GROUP_COUNT的值，则结果是未定义的。

cs 的glsl 4.5 规范

4 Variables and Types

4.3 Storage Qualifier

shared:

compute shader only; variable storage is shared across all work items in a local work group

4.3.4 Input Variables

计算着色器不允许用户定义输入变量，并且与任何其他着色器阶段没有形成正式接口。请参阅第7.1节“内置变量”以了解内置计算着色器输入变量的描述。计算着色器的所有其他输入都是通过显式调用图像加载、纹理获取、从uniform或uniform缓冲区加载，或其他用户提供的代码来检索的。在计算着色器中不允许重新声明内置输入变量。

4.3.6 Output Variables

计算着色器没有内置的输出变量，不支持用户定义的输出变量，也与任何其他着色器阶段没有形成正式接口。计算着色器的所有输出都以诸如图像存储和对原子计数器的操作等副作用的形式呈现。

4.3.8 Shaderd Variables

`shared` 限定符用于声明在计算着色器本地工作组中所有工作项之间共享存储的全局变量。声明为共享的变量只能在计算着色器中使用（参见第2.6节“计算处理器”）。任何对共享变量的其他声明都将导致编译时错误。共享变量隐式具有一致性。也就是说，从一个着色器调用对共享变量的写操作最终将被同一本地工作组内的其他调用所看到。

声明为共享的变量不得具有初始化程序，它们的内容在着色器执行开始时是未定义的。对共享变量的任何写操作将对同一本地工作组内执行相同着色器的其他着色器处理器可见。关于通过不同调用的着色器对同一共享变量的读写的执行顺序没有定义。为了实现对共享变量的读写的顺序，必须使用 `barrier()` 函数使用内存屏障（参见第8.16节“着色器调用控制函数”）。

在单个程序中声明为 `shared` 的所有变量的总大小存在限制。此限制以基本机器单元的单位表示，可以使用OpenGL API查询 `MAX_COMPUTE_SHARED_MEMORY_SIZE` 的值来确定。

4.4.1 Input Layout Qualifiers

除了计算着色器之外的所有着色器，允许在输入变量声明、输入块声明和输入块成员声明上使用位置布局修饰符。其中，变量和块成员（但不包括块本身）还允许使用分量布局修饰符。

4.4.1.4 Compute Shaders Inputs

在计算着色器中，没有布局位置修饰符用于输入。

计算着色器输入的布局修饰符标识符是工作组大小修饰符：

```
layout-qualifier-id:
    local_size_x = integer-constant-expression
    local_size_y = integer-constant-expression
    local_size_z = integer-constant-expression
```

`local_size_x`、`local_size_y` 和 `local_size_z` 修饰符用于在计算着色器中分别声明第一、第二和第三维的固定本地组大小。每个维度的默认大小为1。如果着色器没有为某个维度指定大小，该维度的大小将为1。

例如，在计算着色器中的以下声明：

```
layout(local_size_x = 32, local_size_y = 32) in;
```

用于声明一个二维计算着色器，其本地大小为32 x 32元素，相当于一个三维计算着色器，其中第三个维度的大小为1。

另一个例子是以下声明：

```
layout(local_size_x = 8) in;
```

实际上指定正在编译一个一维计算着色器，其大小为8个元素。

如果着色器在任何维度上的固定本地组大小大于实现支持的该维度的最大大小，则会导致编译时错误。此外，如果在同一个着色器中多次声明这样的布局修饰符，则所有这些声明必须设置相同的本地工作组大小，并将它们设置为相同的值；否则将导致编译时错误。如果附加到单个程序对象的多个计算着色器声明了固定本地组大小，则这些声明必须是相同的；否则将导致链接时错误。声明本地组大小小于或等于零是编译时错误。

此外，如果程序对象包含任何计算着色器，至少其中一个必须包含一个输入布局修饰符，为程序指定一个固定的本地组大小，否则将发生链接时错误。

4.4.2 Output Layout Qualifiers

一些输出布局修饰符适用于所有着色器语言，而一些仅适用于特定语言。后者在下面的各个小节中讨论。

与输入布局修饰符一样，除了计算着色器之外的所有着色器都允许在输出变量声明、输出块声明和输出块成员声明上使用位置布局修饰符。其中，变量和块成员（但不包括块本身）还允许使用分量布局修饰符。

7 Build-in Variabes

某些OpenGL操作在固定功能中发生，并且需要向着色器可执行文件提供值或从中接收值。着色器通过使用内置输入和输出变量与固定功能的OpenGL管线阶段以及可选的其他着色器可执行文件进行通信。

在计算语言中，内置变量声明如下：

```
// 工作组维度
in uvec3 gl_NumWorkGroups;
const uvec3 gl_WorkGroupSize;

// 工作组和调用ID
in uvec3 gl_WorkGroupID;
in uvec3 gl_LocalInvocationID;

// 派生变量
in uvec3 gl_GlobalInvocationID;
in uint gl_LocalInvocationIndex;
```

内置变量 `gl_NumWorkGroups` 是一个计算着色器的输入变量，包含将执行计算着色器的工作组每个维度中的全局工作项的总数。其内容等于传递给 `DispatchCompute` API 入口点的 `num_groups_x`、`num_groups_y` 和 `num_groups_z` 参数中指定的值。

内置常量 `gl_WorkGroupSize` 是一个计算着色器的常数，包含着色器的本地工作组大小。工作组在 X、Y 和 Z 维度的大小存储在 x、y 和 z 分量中。`gl_WorkGroupSize` 中的常量值将与当前着色器的必需 `local_size_x`、`local_size_y` 和 `local_size_z` 布局限定符中指定的值匹配。这是一个常量，因此它可以用于确定本地工作组内可共享的内存数组的大小。在不声明固定本地组大小的着色器中使用 `gl_WorkGroupSize`，或在该着色器声明固定本地组大小之前使用它，将导致编译时错误。当为这些标识符中的一些指定了大小但不是全部时，相应的 `gl_WorkGroupSize` 将具有大小为 1。

内置变量 `gl_WorkGroupID` 是一个计算着色器的输入变量，包含当前调用正在执行的全局工作组的三维索引。可能的值范围从传递给 `DispatchCompute` 的参数，即 (0, 0, 0) 到 `(gl_NumWorkGroups.x - 1, gl_NumWorkGroups.y - 1, gl_NumWorkGroups.z -1)`。

内置变量 `gl_LocalInvocationID` 是一个计算着色器的输入变量，包含当前调用正在执行的全局工作组内的本地工作组的 t 维度索引。该变量的可能值范围在本地工作组大小内，即 (0,0,0) 到 `(gl_WorkGroupSize.x - 1, gl_WorkGroupSize.y - 1, gl_WorkGroupSize.z - 1)`。

内置变量 `gl_GlobalInvocationID` 是一个计算着色器的输入变量，包含当前工作项的全局索引。这个值唯一地标识了当前 `DispatchCompute` 调用启动的所有本地和全局工作组中的这个调用。这是通过以下计算得到的：

```
gl_GlobalInvocationID = gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID;
```

内置变量 `gl_LocalInvocationIndex` 是一个计算着色器的输入变量，包含 `gl_LocalInvocationID` 的一维表示。这对于唯一标识本地工作组内此调用用于使用的共享内存的唯一区域很有用。这是通过以下计算得到的：

```
gl_LocalInvocationIndex = gl_LocalInvocationID.z * gl_WorkGroupSize.x *
gl_WorkGroupSize.y + gl_LocalInvocationID.y * gl_WorkGroupSize.x +
gl_LocalInvocationID.x;
```

7.4.1 Compatibility Profile State

这些变量仅存在于兼容性配置文件中。它们对于计算着色器不可用，但对于所有其他着色器都可用。

8 Build-in Functions

8.16 Shader Invocation Control Functions

着色器调用控制函数仅在曲面细分控制着色器和计算着色器中可用。它用于控制用于处理补丁（在曲面细分控制着色器的情况下）或本地工作组（在计算着色器的情况下）的多个着色器调用的相对执行顺序，否则这些调用将以未定义的相对顺序执行。

语法描述：

```
void barrier();
```

对于任何给定的静态 `barrier()` 实例，必须在单个输入补丁的所有镶嵌控制着色器调用之前进入它，或者在单个工作组的所有调用之前进入它，然后才允许任何调用继续超越它。

函数 `barrier()` 提供了着色器调用之间执行顺序的部分定义。这确保在调用相同静态实例的 `barrier()` 之后，由一个调用写入的值可以安全地被其他调用读取。由于在这些 barrier 调用之间可能以未定义的顺序执行调用，因此在段落 4.3.6 “输出变量”（镶嵌控制着色器）和段落 4.3.8 “共享变量”（计算着色器）中枚举的情况下，计算着色器的每个顶点或每个补丁输出变量或共享变量的值将是未定义的。

对于镶嵌控制着色器，`barrier()` 函数只能放置在镶嵌控制着色器的 `main()` 函数内，不能在任何控制流中调用。在 `main()` 函数中的 return 语句之后也禁止使用 barrier()。任何这样错误放置的 barrier 将导致编译时错误。

对于计算着色器，`barrier()` 函数可以放置在流程控制内，但该流程控制必须是统一流程控制。也就是说，导致执行 barrier 的所有控制表达式必须是动态统一表达式。这确保如果任何着色器调用进入条件语句，则所有调用都将进入该语句。尽管鼓励编译器在可以检测到这种情况时发出警告，但编译器无法完全确定这一点。因此，确保 `barrier()` 仅存在于统一流程控制内是作者的责任。否则，一些着色器调用将无限期地停滞，等待其他调用永远无法到达的 barrier。

8.17 Shader Memory Control Functions

各种类型的着色器可以使用图像变量读取和写入纹理和缓冲对象的内容。
虽然单个着色器调用内读取和写入的顺序是明确定义的，但从多个单独的着色器调用到单个共享内存地址的读取和写入的相对顺序在很大程度上是未定义的。由一个着色器调用执行的内存访问顺序，如其他着色器调用所观察的，也在很大程度上是未定义的，但可以通过内存控制函数来控制。

语法描述	函数	可用性
`void memoryBarrier()`	控制单个着色器调用发出的内存事务的顺序。	所有着色器类型
`void memoryBarrierAtomicCounter()`	控制单个着色器调用发出的对原子计数器变量的访问的顺序。	所有着色器类型
`void memoryBarrierBuffer()`	控制在单个着色器调用内发出的对缓冲变量的内存事务的顺序。	所有着色器类型
`void memoryBarrierShared()`	控制在单个着色器调用内发出的对共享变量的内存事务的顺序。	**仅在计算着色器中可用**
`void memoryBarrierImage()`	控制在单个着色器调用内发出的对图像的内存事务的顺序。	所有着色器类型
`void groupMemoryBarrier()`	控制单个着色器调用内发出的所有内存事务的顺序，如同被同一工作组中的其他调用所看到的。	**仅在计算着色器中可用**

内存屏障内置函数可用于排序对其他着色器调用可访问的内存中存储的变量的读取和写入。调用这些函数时，它们将等待调用者先前执行的所有读取和写入完成，这些读取和写入访问了所选变量类型，然后返回，没有其他效果。内置函数 `memoryBarrierAtomicCounter()`、`memoryBarrierBuffer()`、`memoryBarrierImage()` 和 `memoryBarrierShared()` 分别等待对原子计数器、缓冲区、图像和共享变量的访问完成。内置函数 `memoryBarrier()` 和 `groupMemoryBarrier()` 等待对所有上述变量类型的

访问完成。函数 `memoryBarrierShared()` 和 `groupMemoryBarrier()` 仅在计算着色器中可用；其他函数在所有着色器类型中都可用。

当这些函数返回时，通过一致变量执行的任何内存存储的结果，该存储是在调用之前执行的，将对同一内存由其他着色器调用执行的任何未来一致访问可见。特别是，在一个着色器阶段中以这种方式编写的值将确保在由原始着色器调用执行触发的后续阶段的着色器调用进行一致内存访问时可见（例如，对于由特定几何着色器调用生成的基元的片段着色器调用）。

此外，内存屏障函数按其他着色器调用观察的顺序对由调用调用执行的存储进行排序。没有内存屏障时，如果一个着色器调用对一致变量执行两个存储，那么第二个存储的值可能在看到第一个存储的值之前就被第二个着色器调用看到。然而，如果第一个着色器调用在两个存储之间调用了内存屏障函数，那么在看到第一个存储的值之前，选定的其他着色器调用将永远不会看到第二个存储的结果。当使用 `groupMemoryBarrier()` 或 `memoryBarrierShared()` 函数时，此排序保证仅适用于同一计算着色器工作组中的其他着色器调用；所有其他内存屏障函数都向所有其他着色器调用提供此保证。不需要内存屏障来保证由执行存储的调用观察到的内存存储的顺序；从先前写入同一内存的变量读取的调用将始终看到最近写入的值，除非另一个着色器调用还写入了相同的内存。

mesa-18实现

OpenGL

核心接口

```
void GLAPIENTRY
_mesa_DispatchCompute_no_error(GLuint num_groups_x, GLuint num_groups_y,
                               GLuint num_groups_z)
{
   dispatch_compute(num_groups_x, num_groups_y, num_groups_z, true);
}

static ALWAYS_INLINE void
dispatch_compute(GLuint num_groups_x, GLuint num_groups_y,
                 GLuint num_groups_z, bool no_error)
{
   GET_CURRENT_CONTEXT(ctx);
   const GLuint num_groups[3] = { num_groups_x, num_groups_y, num_groups_z };

   ctx->Driver.DispatchCompute(ctx, num_groups);
        [jump state_tracker st_dispatch_compute]
}
```

```
extern void GLAPIENTRY
_mesa_DispatchComputeIndirect_no_error(GLintptr indirect)
{
   dispatch_compute_indirect(indirect, true);
}

static ALWAYS_INLINE void
dispatch_compute_indirect(GLintptr indirect, bool no_error)
{
   ctx->Driver.DispatchComputeIndirect(ctx, indirect);
```

```
        [jump  state_tracker st_dispatch_compute_indirect]
}
```

扩展接口

```
void GLAPIENTRY
_mesa_DispatchComputeGroupSizeARB_no_error(GLuint num_groups_x,
                                           GLuint num_groups_y,
                                           GLuint num_groups_z,
                                           GLuint group_size_x,
                                           GLuint group_size_y,
                                           GLuint group_size_z)
{
   dispatch_compute_group_size(num_groups_x, num_groups_y, num_groups_z,
                               group_size_x, group_size_y, group_size_z,
                               true);
}
```

glsl

hir生成

cs 特殊内建变量的添加

```
/**
 * Generate variables which only exist in compute shaders.
 */
void
builtin_variable_generator::generate_cs_special_vars()
{
   add_system_value(SYSTEM_VALUE_LOCAL_INVOCATION_ID, uvec3_t,
                    "gl_LocalInvocationID");
   add_system_value(SYSTEM_VALUE_WORK_GROUP_ID, uvec3_t, "gl_WorkGroupID");
   add_system_value(SYSTEM_VALUE_NUM_WORK_GROUPS, uvec3_t, "gl_NumWorkGroups");

   if (state->ARB_compute_variable_group_size_enable) {
      add_system_value(SYSTEM_VALUE_LOCAL_GROUP_SIZE,
                       uvec3_t, "gl_LocalGroupSizeARB");
   }

   add_system_value(SYSTEM_VALUE_GLOBAL_INVOCATION_ID,
                    uvec3_t, "gl_GlobalInvocationID");
   add_system_value(SYSTEM_VALUE_LOCAL_INVOCATION_INDEX,
                    uint_t, "gl_LocalInvocationIndex");
}
```

处理layout 中的local_size标识

```
static void
set_shader_inout_layout(struct gl_shader *shader,
             struct _mesa_glsl_parse_state *state)
{
   ...
   ...

   switch (shader->Stage) {
   ...
   case MESA_SHADER_COMPUTE:
      // 将数据存放在gl_shader_info 的Comp中
      if (state->cs_input_local_size_specified) {
         for (int i = 0; i < 3; i++)
            shader->info.Comp.LocalSize[i] = state->cs_input_local_size[i];
      } else {
         for (int i = 0; i < 3; i++)
            shader->info.Comp.LocalSize[i] = 0;
      }

      shader->info.Comp.LocalSizeVariable =
         state->cs_input_local_size_variable_specified;
      break;

   default:
      /* Nothing to do. */
      break;
   }

    ...
}
```

对barrier内建函数的添加

```
void
builtin_builder::create_builtins()
{
   ...

   add_function("barrier", _barrier(), NULL);

   add_function("memoryBarrier",
                _memory_barrier("__intrinsic_memory_barrier",
                                shader_image_load_store),
                NULL);
   add_function("groupMemoryBarrier",
                _memory_barrier("__intrinsic_group_memory_barrier",
                                compute_shader),
                NULL);
   add_function("memoryBarrierAtomicCounter",
                _memory_barrier("__intrinsic_memory_barrier_atomic_counter",
                                compute_shader_supported),
```

```
                NULL);
   add_function("memoryBarrierBuffer",
                _memory_barrier("__intrinsic_memory_barrier_buffer",
                                compute_shader_supported),
                NULL);
   add_function("memoryBarrierImage",
                _memory_barrier("__intrinsic_memory_barrier_image",
                                compute_shader_supported),
                NULL);
   add_function("memoryBarrierShared",
                _memory_barrier("__intrinsic_memory_barrier_shared",
                                compute_shader),
                NULL);
   ...
}

ir_function_signature *
builtin_builder::_memory_barrier(const char *intrinsic_name,
                                 builtin_available_predicate avail)
{
   MAKE_SIG(glsl_type::void_type, avail, 0);
   body.emit(call(shader->symbols->get_function(intrinsic_name),
                  NULL, sig->parameters));
   return sig;
}
```

链接cs

链接uniform变量对shared引用的处理

```
static bool
link_varyings_and_uniforms(unsigned first, unsigned last,
                           struct gl_context *ctx,
                           struct gl_shader_program *prog, void *mem_ctx)
{

   ...

   for (unsigned i = 0; i < MESA_SHADER_STAGES; i++) {
      ...

      if (i == MESA_SHADER_COMPUTE)
         lower_shared_reference(ctx, prog, prog->_LinkedShaders[i]);
      ...
   }

   return true;
}
```

```
void lower_shared_reference(struct gl_context *ctx,
                            struct gl_shader_program *prog,
```

```
                              struct gl_linked_shader *shader)
{
   // 如果着色器阶段不是计算着色器，直接返回
   if (shader->Stage != MESA_SHADER_COMPUTE)
      return;

   // 创建一个用于处理共享变量引用的访问者对象
   lower_shared_reference_visitor v(shader);

   /* 循环遍历指令，降低引用，因为使用共享变量数组的共享变量引用作为索引会产生一系列指令，
    * 所有这些指令都有为该数组索引克隆的共享变量引用。
    */
   do {
      v.progress = false;
      visit_list_elements(&v, shader->ir);
   } while (v.progress);

   // 更新程序对象中的计算着色器的共享大小信息
   prog->Comp.SharedSize = v.shared_size;

   /* OpenGL 4.5（核心配置文件）规范中的第19.1节（计算着色器变量）规定：
    *
    *   "单个程序对象中声明为shared的所有变量的总大小存在限制。
    *    这个限制以基本机器单元为单位，可以通过查询MAX_COMPUTE_SHARED_MEMORY_SIZE的值获
得。"
    */
   // 如果共享大小超过了最大限制，输出链接错误信息
   if (prog->Comp.SharedSize > ctx->Const.MaxComputeSharedMemorySize) {
      linker_error(prog, "使用的共享内存太多 (%u/%u)\n",
                   prog->Comp.SharedSize,
                   ctx->Const.MaxComputeSharedMemorySize);
   }
}
```

- 这里涉及到lower_shared_reference_visitor 对shared buffer的权限处理,共享处理
- 这段代码的目的是处理计算着色器中的共享变量引用，通过降低这些引用来确保不超过OpenGL规范中对共享内存大小的限制。

state_tracker

cs相关接口

```
void st_init_compute_functions(struct dd_function_table *functions)
{
   functions->DispatchCompute = st_dispatch_compute;
   functions->DispatchComputeIndirect = st_dispatch_compute_indirect;
   functions->DispatchComputeGroupSize = st_dispatch_compute_group_size;
}
```

pipe_grid_info

```
/**
 * 用于描述 launch_grid 调用的信息。
 */
struct pipe_grid_info
{
   /**
    * 对于将 PIPE_SHADER_IR_NATIVE 作为其首选 IR 的驱动程序，此值将是 opencl.kernels 元数据列表中内核的索引。
    */
   uint32_t pc;

   /**
    * 将用于初始化 INPUT 资源，应指向至少 pipe_compute_state::req_input_mem 字节的缓冲区。
    */
   void *input;

   /**
    * 网格的维度，1-3，例如传递给 clEnqueueNDRangeKernel 的 work_dim 参数。请注意，对于未使用的维度，block[] 和 grid[] 必须填充为 1。
    */
   uint work_dim;

   /**
    * 确定要使用的工作块的布局（以线程单元为单位）。
    */
   uint block[3];

   /**
    * 确定要使用的网格的布局（以块单元为单位）。
    */
   uint grid[3];

   /* 间接计算参数资源：如果不是 NULL，则块大小将从此缓冲区中获取，其布局如下：
    *
    *  struct {
    *     uint32_t num_blocks_x;
    *     uint32_t num_blocks_y;
    *     uint32_t num_blocks_z;
    *  };
    */
   struct pipe_resource *indirect;
   unsigned indirect_offset; /**< 必须为 4 字节对齐 */
};
```

st_dispatch_compute

```
static void st_dispatch_compute(struct gl_context *ctx,
                                const GLuint *num_groups)
{
   st_dispatch_compute_common(ctx, num_groups, NULL, NULL, 0);
```

```
}

static void st_dispatch_compute_common(struct gl_context *ctx,
                                       const GLuint *num_groups,
                                       const GLuint *group_size = NULL,
                                       struct pipe_resource *indirect = NULL,
                                       GLintptr indirect_offset = 0)
{
   struct gl_program *prog =
      ctx->_Shader->CurrentProgram[MESA_SHADER_COMPUTE];
   struct st_context *st = st_context(ctx);
   struct pipe_context *pipe = st->pipe;
   struct pipe_grid_info info = { 0 };

   st_flush_bitmap_cache(st);
   st_invalidate_readpix_cache(st);


   if (ctx->NewState)
      _mesa_update_state(ctx);

    // 触发计算管线状态更新
   if ((st->dirty | ctx->NewDriverState) & ST_PIPELINE_COMPUTE_STATE_MASK ||
       st->compute_shader_may_be_dirty)
      st_validate_state(st, ST_PIPELINE_COMPUTE);

   for (unsigned i = 0; i < 3; i++) {
      info.block[i] = group_size ? group_size[i] : prog->info.cs.local_size[i];
      info.grid[i]  = num_groups ? num_groups[i] : 0;
   }

   if (indirect) {
      info.indirect = indirect;
      info.indirect_offset = indirect_offset;
   }

   pipe->launch_grid(pipe, &info);
    [jump dri (*launch_grid)]
}
```

st_dispatch_compute_indirect

```
static void st_dispatch_compute_indirect(struct gl_context *ctx,
                                         GLintptr indirect_offset)
{
   struct gl_buffer_object *indirect_buffer = ctx->DispatchIndirectBuffer;
   struct pipe_resource *indirect = st_buffer_object(indirect_buffer)->buffer;

   st_dispatch_compute_common(ctx, NULL, NULL, indirect, indirect_offset);
}
```

- 同上

生成st_compute_program 的特殊处理

cs使用st_compute_program 来保存shader信息
在st_translate_program_common 内部有以下几点注意:

1. 将shader中的local_size转换为 TGSI_PROPERTY_CS_FIXED_BLOCK_*

```
st_translate_program( ...) {

   ...

   if (procType == PIPE_SHADER_COMPUTE) {
      emit_compute_block_size(proginfo, ureg);
   }

}

static void
emit_compute_block_size(const struct gl_program *prog,
                        struct ureg_program *ureg) {
   ureg_property(ureg, TGSI_PROPERTY_CS_FIXED_BLOCK_WIDTH,
                 prog->info.cs.local_size[0]);
   ureg_property(ureg, TGSI_PROPERTY_CS_FIXED_BLOCK_HEIGHT,
                 prog->info.cs.local_size[1]);
   ureg_property(ureg, TGSI_PROPERTY_CS_FIXED_BLOCK_DEPTH,
                 prog->info.cs.local_size[2]);
}
```

2. tgsi_tokens保存在 st_compute_program.tgsi.prog里面, 这个tgsi变量为pipe_compute_state,不同于其他pipe_shader_state

```
st_translate_program_common(struct st_context *st,..) {
    ...
   if (tgsi_processor == PIPE_SHADER_COMPUTE) {
      struct st_compute_program *stcp = (struct st_compute_program *) prog;
      out_state->tokens = ureg_get_tokens(ureg, &stcp->num_tgsi_tokens);
      stcp->tgsi.prog = out_state->tokens;
   } else {
      struct st_common_program *stcp = (struct st_common_program *) prog;
      out_state->tokens = ureg_get_tokens(ureg, &stcp->num_tgsi_tokens);
   }
   ...

}
```

cs 的状态转发处理

ST_NEW_CS_STATE

```
void
st_update_cp( struct st_context *st )
{
   struct st_compute_program *stcp;

   ...

   void *shader;

   if (st->shader_has_one_variant[MESA_SHADER_COMPUTE] && stcp->variants) {
      shader = stcp->variants->driver_shader;
   } else {
        // 编译shader
      shader = st_get_cp_variant(st, &stcp->tgsi,
                                 &stcp->variants)->driver_shader;
   }

   st_reference_compprog(st, &st->cp, stcp);

    //绑定cs 状态
   cso_set_compute_shader_handle(st->cso_context, shader);
   [jump dri (*bind_compute_state)]
}
```

ST_NEW_CS_CONSTANTS 常量状态更新

```
/* Compute shader:
 */
void
st_update_cs_constants(struct st_context *st)
{
   struct st_compute_program *cp = st->cp;

   if (cp)
      st_upload_constants(st, &cp->Base);
}
```

写入常量描述符

- 调用st_upload_constants 略

ST_NEW_CS_SAMPLER_VIEWS,

```
void
st_update_compute_textures(struct st_context *st)
{
   const struct gl_context *ctx = st->ctx;

   if (ctx->ComputeProgram._Current) {
      update_textures_local(st, PIPE_SHADER_COMPUTE,
                            ctx->ComputeProgram._Current);
   }
}
```

写入纹理描述符

- see texture*.pdf

ST_NEW_CS_SAMPLERS,

```
void
st_update_compute_samplers(struct st_context *st)
{
   const struct gl_context *ctx = st->ctx;

   if (ctx->ComputeProgram._Current) {
      update_shader_samplers(st,
                             PIPE_SHADER_COMPUTE,
                             ctx->ComputeProgram._Current, NULL, NULL);
   }
}
```

写入采样状态

- see 纹理

ST_NEW_CS_IMAGES,

```
void st_bind_cs_images(struct st_context *st)
{
   struct gl_program *prog =
      st->ctx->_Shader->CurrentProgram[MESA_SHADER_COMPUTE];

   st_bind_images(st, prog, PIPE_SHADER_COMPUTE);
}

static void
st_bind_images(struct st_context *st, struct gl_program *prog,
               enum pipe_shader_type shader_type)
{
   unsigned i;
   struct pipe_image_view images[MAX_IMAGE_UNIFORMS];
   struct gl_program_constants *c;

   if (!prog || !st->pipe->set_shader_images)
      return;
```

```
   c = &st->ctx->Const.Program[prog->info.stage];

   for (i = 0; i < prog->info.num_images; i++) {
      struct pipe_image_view *img = &images[i];

      st_convert_image_from_unit(st, img, prog->sh.ImageUnits[i],
                                 prog->sh.ImageAccess[i]);
   }
   cso_set_shader_images(st->cso_context, shader_type, 0,
                         prog->info.num_images, images);
   /* clear out any stale shader images */
   if (prog->info.num_images < c->MaxImageUniforms)
      cso_set_shader_images(
            st->cso_context, shader_type, prog->info.num_images,
            c->MaxImageUniforms - prog->info.num_images, NULL);
}
```

写入图想描述符

- see 纹理

ST_NEW_CS_UBOS,

```
void
st_bind_cs_ubos(struct st_context *st)
{
   struct gl_program *prog =
      st->ctx->_Shader->CurrentProgram[MESA_SHADER_COMPUTE];

   st_bind_ubos(st, prog, PIPE_SHADER_COMPUTE);
}
```

写入常量描述符

- see ubo.pdf

ST_NEW_CS_SSBOS,

```
void st_bind_cs_ssbos(struct st_context *st)
{
   struct gl_program *prog =
      st->ctx->_Shader->CurrentProgram[MESA_SHADER_COMPUTE];

   st_bind_ssbos(st, prog, PIPE_SHADER_COMPUTE);
}
```

- see ssbo.pdf

ST_NEW_CS_ATOMICS);

```
void
st_bind_cs_atomics(struct st_context *st)
{
   if (st->has_hw_atomics) {
      st_bind_hw_atomic_buffers(st);
      return;
   }
   struct gl_program *prog =
      st->ctx->_Shader->CurrentProgram[MESA_SHADER_COMPUTE];

   st_bind_atomics(st, prog, PIPE_SHADER_COMPUTE);

}
```

写入常量描述符

- see atom.pdf

dri

radeonsi

radeonsi cs api

```
void si_init_compute_functions(struct si_context *sctx)
{
    sctx->b.create_compute_state = si_create_compute_state;
    sctx->b.delete_compute_state = si_delete_compute_state;
    sctx->b.bind_compute_state = si_bind_compute_state;
    sctx->b.set_compute_resources = si_set_compute_resources;
    sctx->b.set_global_binding = si_set_global_binding;
    sctx->b.launch_grid = si_launch_grid;
}
```

si cs create

启动编译线程

```
static void *si_create_compute_state(
    struct pipe_context *ctx,
    const struct pipe_compute_state *cso)
{
    struct si_context *sctx = (struct si_context *)ctx;
    struct si_screen *sscreen = (struct si_screen *)ctx->screen;
    struct si_compute *program = CALLOC_STRUCT(si_compute);

    //异步编译开始
    si_schedule_initial_compile(sctx, PIPE_SHADER_COMPUTE,
                        &program->ready,
                        &program->compiler_ctx_state,
                        program, si_create_compute_state_async);
```

```
    return program;
}
```

产生selector info

```
/* Asynchronous compute shader compilation. */
static void si_create_compute_state_async(void *job, int thread_index)
{
    struct si_compute *program = (struct si_compute *)job;
    struct si_shader *shader = &program->shader;
    struct si_shader_selector sel;
    struct ac_llvm_compiler *compiler;
    struct pipe_debug_callback *debug = &program->compiler_ctx_state.debug;
    struct si_screen *sscreen = program->screen;

    compiler = &sscreen->compiler[thread_index];

    memset(&sel, 0, sizeof(sel));

    sel.screen = sscreen;

    if (program->ir_type == PIPE_SHADER_IR_TGSI) {
        tgsi_scan_shader(program->ir.tgsi, &sel.info);
        sel.tokens = program->ir.tgsi;
    } else {
    }

    /* Store the declared LDS size into tgsi_shader_info for the shader
     * cache to include it.
     */
    sel.info.properties[TGSI_PROPERTY_CS_LOCAL_SIZE] = program->local_size;

    sel.type = PIPE_SHADER_COMPUTE;
    si_get_active_slot_masks(&sel.info,
                &program->active_const_and_shader_buffers,
                &program->active_samplers_and_images);

    program->shader.selector = &sel;
    program->shader.is_monolithic = true;
    program->uses_grid_size = sel.info.uses_grid_size;
    program->uses_bindless_samplers = sel.info.uses_bindless_samplers;
    program->uses_bindless_images = sel.info.uses_bindless_images;
    program->reads_variable_block_size =
        sel.info.uses_block_size &&
        sel.info.properties[TGSI_PROPERTY_CS_FIXED_BLOCK_WIDTH] == 0;
    program->num_cs_user_data_dwords =
        sel.info.properties[TGSI_PROPERTY_CS_USER_DATA_DWORDS];

    void *ir_binary = si_get_ir_binary(&sel);

    /* Try to load the shader from the shader cache. */
    mtx_lock(&sscreen->shader_cache_mutex);
```

```
    if (ir_binary &&
        si_shader_cache_load_shader(sscreen, ir_binary, shader)) {
        ...
    } else {
        if (si_shader_create(sscreen, compiler, &program->shader, debug)) {
            ...
        }

        bool scratch_enabled = shader->config.scratch_bytes_per_wave > 0;
        unsigned user_sgprs = SI_NUM_RESOURCE_SGPRS +
                      (sel.info.uses_grid_size ? 3 : 0) +
                      (program->reads_variable_block_size ? 3 : 0) +
                      program->num_cs_user_data_dwords;

        shader->config.rsrc1 =
            S_00B848_VGPRS((shader->config.num_vgprs - 1) / 4) |
            S_00B848_SGPRS((shader->config.num_sgprs - 1) / 8) |
            S_00B848_DX10_CLAMP(1) |
            S_00B848_FLOAT_MODE(shader->config.float_mode);

        shader->config.rsrc2 =
            S_00B84C_USER_SGPR(user_sgprs) |
            S_00B84C_SCRATCH_EN(scratch_enabled) |
            S_00B84C_TGID_X_EN(sel.info.uses_block_id[0]) |
            S_00B84C_TGID_Y_EN(sel.info.uses_block_id[1]) |
            S_00B84C_TGID_Z_EN(sel.info.uses_block_id[2]) |
            S_00B84C_TIDIG_COMP_CNT(sel.info.uses_thread_id[2] ? 2 :
                        sel.info.uses_thread_id[1] ? 1 : 0) |
            S_00B84C_LDS_SIZE(shader->config.lds_size);

        }
    }
}
```

产生LLVM IR bin

```
int si_shader_create(struct si_screen *sscreen, struct ac_llvm_compiler
*compiler,
             struct si_shader *shader,
             struct pipe_debug_callback *debug)
{
    struct si_shader_selector *sel = shader->selector;
    struct si_shader *mainp = *si_get_main_shader_part(sel, &shader->key);
    int r;

    /* LS, ES, VS are compiled on demand if the main part hasn't been
     * compiled for that stage.
     *
     * Vertex shaders are compiled on demand when a vertex fetch
     * workaround must be applied.
     */
    if (shader->is_monolithic) {
        /* Monolithic shader (compiled as a whole, has many variants,
```

```
         * may take a long time to compile).
         */
        r = si_compile_tgsi_shader(sscreen, compiler, shader, debug);
        if (r)
            return r;
    } else {
        ...
    }

    si_fix_resource_usage(sscreen, shader);
    si_shader_dump(sscreen, shader, debug, sel->info.processor,
               stderr, true);

    /* Upload. */
    r = si_shader_binary_upload(sscreen, shader);
    if (r) {
        fprintf(stderr, "LLVM failed to upload shader\n");
        return r;
    }

    return 0;
}
```

初始化编译上下文, tgsi指令emit接口

```
int si_compile_tgsi_shader(struct si_screen *sscreen,
               struct ac_llvm_compiler *compiler,
               struct si_shader *shader,
               struct pipe_debug_callback *debug)
{
    struct si_shader_selector *sel = shader->selector;
    struct si_shader_context ctx;
    int r = -1;

    si_init_shader_ctx(&ctx, sscreen, compiler);
    si_llvm_context_set_tgsi(&ctx, shader);

    memset(shader->info.vs_output_param_offset, AC_EXP_PARAM_UNDEFINED,
           sizeof(shader->info.vs_output_param_offset));

    shader->info.uses_instanceid = sel->info.uses_instanceid;

    if (!si_compile_tgsi_main(&ctx)) {
        si_llvm_dispose(&ctx);
        return -1;
    }

//  si_llvm_optimize_module(&ctx);

    /* Post-optimization transformations and analysis. */
//  si_optimize_vs_outputs(&ctx);

    /* Compile to bytecode. */
    r = si_compile_llvm(sscreen, &shader->binary, &shader->config, compiler,
                ctx.ac.module, debug, ctx.type,
```

```
                si_get_shader_name(shader, ctx.type),
                si_should_optimize_less(compiler, shader->selector));
    si_llvm_dispose(&ctx);
    if (r) {
        fprintf(stderr, "LLVM failed to compile shader\n");
        return r;
    }

    /* Validate SGPR and VGPR usage for compute to detect compiler bugs.
     * LLVM 3.9svn has this bug.
     */
    if (sel->type == PIPE_SHADER_COMPUTE) {
        unsigned wave_size = 64;
        unsigned max_vgprs = 256;
        unsigned max_sgprs = sscreen->info.chip_class >= VI ? 800 : 512;
        unsigned max_sgprs_per_wave = 128;
        unsigned max_block_threads = si_get_max_workgroup_size(shader);
        unsigned min_waves_per_cu = DIV_ROUND_UP(max_block_threads, wave_size);
        unsigned min_waves_per_simd = DIV_ROUND_UP(min_waves_per_cu, 4);

        max_vgprs = max_vgprs / min_waves_per_simd;
        max_sgprs = MIN2(max_sgprs / min_waves_per_simd, max_sgprs_per_wave);

        if (shader->config.num_sgprs > max_sgprs ||
            shader->config.num_vgprs > max_vgprs) {
            fprintf(stderr, "LLVM failed to compile a shader correctly: "
                "SGPR:VGPR usage is %u:%u, but the hw limit is %u:%u\n",
                shader->config.num_sgprs, shader->config.num_vgprs,
                max_sgprs, max_vgprs);

            /* Just terminate the process, because dependent
             * shaders can hang due to bad input data, but use
             * the env var to allow shader-db to work.
             */
            if (!debug_get_bool_option("SI_PASS_BAD_SHADERS", false))
                abort();
        }
    }

    si_calculate_max_simd_waves(shader);
    si_shader_dump_stats_for_shader_db(shader, debug);
    return 0;
}
```

编译main

```
static bool si_compile_tgsi_main(struct si_shader_context *ctx)
{
    struct si_shader *shader = ctx->shader;
    struct si_shader_selector *sel = shader->selector;
    struct lp_build_tgsi_context *bld_base = &ctx->bld_base;

    // TODO clean all this up!
    switch (ctx->type) {
```

```
    ...
    case PIPE_SHADER_COMPUTE:
        ctx->abi.load_local_group_size = get_block_size;
        break;
    default:
        assert(!"Unsupported shader type");
        return false;
    }

    ctx->abi.load_ubo = load_ubo;
    ctx->abi.load_ssbo = load_ssbo;

    create_function(ctx);
        struct si_shader *shader = ctx->shader;
        struct si_function_info fninfo;
        LLVMTypeRef returns[16+32*4];
        unsigned i, num_return_sgprs;
        unsigned num_returns = 0;
        unsigned num_prolog_vgprs = 0;
        unsigned type = ctx->type;
        unsigned vs_blit_property =
            shader->selector->info.properties[TGSI_PROPERTY_VS_BLIT_SGPRS];

        si_init_function_info(&fninfo);

        LLVMTypeRef v3i32 = LLVMVectorType(ctx->i32, 3);

        switch (type) {
            ...
        case PIPE_SHADER_COMPUTE:
            declare_global_desc_pointers(ctx, &fninfo);
            declare_per_stage_desc_pointers(ctx, &fninfo, true);
            if (shader->selector->info.uses_grid_size)
                add_arg_assign(&fninfo, ARG_SGPR, v3i32, &ctx-
>abi.num_work_groups);
            if (shader->selector->info.uses_block_size &&
                shader->selector-
>info.properties[TGSI_PROPERTY_CS_FIXED_BLOCK_WIDTH] == 0)
                ctx->param_block_size = add_arg(&fninfo, ARG_SGPR, v3i32);

            unsigned cs_user_data_dwords =
                shader->selector-
>info.properties[TGSI_PROPERTY_CS_USER_DATA_DWORDS];
            if (cs_user_data_dwords) {
                ctx->param_cs_user_data = add_arg(&fninfo, ARG_SGPR,
                                  LLVMVectorType(ctx->i32, cs_user_data_dwords));
            }

            for (i = 0; i < 3; i++) {
                ctx->abi.workgroup_ids[i] = NULL;
                if (shader->selector->info.uses_block_id[i])
                    add_arg_assign(&fninfo, ARG_SGPR, ctx->i32, &ctx-
>abi.workgroup_ids[i]);
            }
```

```
            add_arg_assign(&fninfo, ARG_VGPR, v3i32, &ctx-
>abi.local_invocation_ids);
            break;
        default:
            assert(0 && "unimplemented shader");
            return;
        }

        si_create_function(ctx, "main", returns, num_returns, &fninfo,
                   si_get_max_workgroup_size(shader));

        shader->info.num_input_sgprs = 0;
        shader->info.num_input_vgprs = 0;

        for (i = 0; i < fninfo.num_sgpr_params; ++i)
            shader->info.num_input_sgprs += ac_get_type_size(fninfo.types[i]) /
4;

        for (; i < fninfo.num_params; ++i)
            shader->info.num_input_vgprs += ac_get_type_size(fninfo.types[i]) /
4;

        assert(shader->info.num_input_vgprs >= num_prolog_vgprs);
        shader->info.num_input_vgprs -= num_prolog_vgprs;

    if (sel->tokens) {
        if (!lp_build_tgsi_llvm(bld_base, sel->tokens)) {
            fprintf(stderr, "Failed to translate shader from TGSI to LLVM\n");
            return false;
        }
    } else {
    }

    si_llvm_build_ret(ctx, ctx->return_value);
    return true;
}
```

- 首先通过create_function 构建了LLVM IR
- 最后通过lp_build_tgsi_llvm 构造body LLVM IR

si cs 状态绑定

```
static void si_bind_compute_state(struct pipe_context *ctx, void *state)
{
    struct si_context *sctx = (struct si_context*)ctx;
    struct si_compute *program = (struct si_compute*)state;

    sctx->cs_shader_state.program = program;
    if (!program)
        return;

    /* Wait because we need active slot usage masks. */
    if (program->ir_type != PIPE_SHADER_IR_NATIVE)
        util_queue_fence_wait(&program->ready);
```

```
    // 激活const
    // active_const_and_shader_buffers通过si_get_active_slot_masks 获取， 而这又是通过tgsi_shader_info获取
    si_set_active_descriptors(sctx,
                  SI_DESCS_FIRST_COMPUTE +
                  SI_SHADER_DESCS_CONST_AND_SHADER_BUFFERS,
                  program->active_const_and_shader_buffers);
        struct si_descriptors *desc = &sctx->descriptors[desc_idx];

        int first, count;
        u_bit_scan_consecutive_range64(&new_active_mask, &first, &count);
        assert(new_active_mask == 0);

        /* Upload/dump descriptors if slots are being enabled. */
        if (first < desc->first_active_slot ||
            first + count > desc->first_active_slot + desc->num_active_slots)
            sctx->descriptors_dirty |= 1u << desc_idx;

        desc->first_active_slot = first;
        desc->num_active_slots = count;


    si_set_active_descriptors(sctx,
                  SI_DESCS_FIRST_COMPUTE +
                  SI_SHADER_DESCS_SAMPLERS_AND_IMAGES,
                  program->active_samplers_and_images);
}
```

cs 启动 si_launch_grid

```
tatic void si_launch_grid(
        struct pipe_context *ctx, const struct pipe_grid_info *info)
{
    struct si_context *sctx = (struct si_context*)ctx;
    struct si_compute *program = sctx->cs_shader_state.program;
    const amd_kernel_code_t *code_object =
        si_compute_get_code_object(program, info->pc);
    int i;


    if (sctx->last_num_draw_calls != sctx->num_draw_calls) {
        si_update_fb_dirtiness_after_rendering(sctx);
        sctx->last_num_draw_calls = sctx->num_draw_calls;
    }

    si_decompress_textures(sctx, 1 << PIPE_SHADER_COMPUTE);

    if (info->indirect) {
        si_context_add_resource_size(sctx, info->indirect);

        /* Indirect buffers use TC L2 on GFX9, but not older hw. */
        if (sctx->chip_class <= VI &&
            r600_resource(info->indirect)->TC_L2_dirty) {
            sctx->flags |= SI_CONTEXT_WRITEBACK_GLOBAL_L2;
```

```
            r600_resource(info->indirect)->TC_L2_dirty = false;
        }
    }

    if (!sctx->cs_shader_state.initialized)
        si_initialize_compute(sctx);  // 计算状态初始化

    if (sctx->flags)
        si_emit_cache_flush(sctx);

    if (!si_switch_compute_shader(sctx, program, &program->shader,
                  code_object, info->pc))
        return;

    // 上传cs所用的const,shader, sampler, image描述符资源
    si_upload_compute_shader_descriptors(sctx);

    //设定上述资源的寄存器地址
    si_emit_compute_shader_pointers(sctx);

    /* Global buffers */
    // global buffers用于tc, 暂时不考虑
    for (i = 0; i < MAX_GLOBAL_BUFFERS; i++) {
        ...
    }

    // 这一步是设置参数grid_size的寄存器地址
    if (program->ir_type != PIPE_SHADER_IR_NATIVE)
        si_setup_tgsi_user_data(sctx, info);

    // 发射grid设定pm4
    si_emit_dispatch_packets(sctx, info);


    sctx->compute_is_busy = true;
    sctx->num_compute_calls++;
    if (sctx->cs_shader_state.uses_scratch)
        sctx->num_spill_compute_calls++;
}
```

cs 下发参数数据及地址

```
static void si_setup_tgsi_user_data(struct si_context *sctx,
                                    const struct pipe_grid_info *info)
{
    struct si_compute *program = sctx->cs_shader_state.program;
    struct radeon_cmdbuf *cs = sctx->gfx_cs;

    // grid_size_reg 现在指向user_data_4
    unsigned grid_size_reg = R_00B900_COMPUTE_USER_DATA_0 +
                 4 * SI_NUM_RESOURCE_SGPRS;
    unsigned block_size_reg = grid_size_reg +
                  /* 12 bytes = 3 dwords. */
                  12 * program->uses_grid_size;
```

```
    unsigned cs_user_data_reg = block_size_reg +
                    12 * program->reads_variable_block_size;

    if (info->indirect) {
        if (program->uses_grid_size) {
            uint64_t base_va = r600_resource(info->indirect)->gpu_address;
            uint64_t va = base_va + info->indirect_offset;
            int i;

            radeon_add_to_buffer_list(sctx, sctx->gfx_cs,
                     r600_resource(info->indirect),
                     RADEON_USAGE_READ, RADEON_PRIO_DRAW_INDIRECT);

            for (i = 0; i < 3; ++i) {
                radeon_emit(cs, PKT3(PKT3_COPY_DATA, 4, 0));
                radeon_emit(cs, COPY_DATA_SRC_SEL(COPY_DATA_SRC_MEM) |
                            COPY_DATA_DST_SEL(COPY_DATA_REG));
                radeon_emit(cs, (va + 4 * i));
                radeon_emit(cs, (va + 4 * i) >> 32);
                radeon_emit(cs, (grid_size_reg >> 2) + i);
                radeon_emit(cs, 0);
            }
        }
    } else {
    //  设置grid参数, 标识使用了gl_NumWorkGroups
        if (program->uses_grid_size) {
            radeon_set_sh_reg_seq(cs, grid_size_reg, 3);
            radeon_emit(cs, info->grid[0]);
            radeon_emit(cs, info->grid[1]);
            radeon_emit(cs, info->grid[2]);
        }
        //设置block 寄存器地址 gl_LocalGroupSizeARB
        if (program->reads_variable_block_size) {
            radeon_set_sh_reg_seq(cs, block_size_reg, 3);
            radeon_emit(cs, info->block[0]);
            radeon_emit(cs, info->block[1]);
            radeon_emit(cs, info->block[2]);
        }
    }
    // 设置了user_data数据地址
    if (program->num_cs_user_data_dwords) {
        radeon_set_sh_reg_seq(cs, cs_user_data_reg, program-
>num_cs_user_data_dwords);
        radeon_emit_array(cs, sctx->cs_user_data, program-
>num_cs_user_data_dwords);
    }
}
```

- 根据是否采用indirect 接口决定通过寄存器还是缓冲派发数据
- 根据内建变量的使用情况,下发grid_size,block_size, param_use_data等参数数据
- grid_size, block_size, 都是连续的三个寄存器地址, 其中开始从COMPUTE_USER_DATA_4 开始类推

cs shader binary 程序地址下发

```
static bool si_switch_compute_shader(struct si_context *sctx,
                                     struct si_compute *program,
                        struct si_shader *shader,
                        const amd_kernel_code_t *code_object,
                        unsigned offset)
{
    struct radeon_cmdbuf *cs = sctx->gfx_cs;
    struct si_shader_config inline_config = {0};
    struct si_shader_config *config;
    uint64_t shader_va;

    if (sctx->cs_shader_state.emitted_program == program &&
        sctx->cs_shader_state.offset == offset)
        return true;

    if (program->ir_type != PIPE_SHADER_IR_NATIVE) {
        config = &shader->config;
    } else {
    }

    if (!si_setup_compute_scratch_buffer(sctx, shader, config))
        return false;

    if (shader->scratch_bo) {
        COMPUTE_DBG(sctx->screen, "Waves: %u; Scratch per wave: %u bytes; "
                    "Total Scratch: %u bytes\n", sctx->scratch_waves,
                config->scratch_bytes_per_wave,
                config->scratch_bytes_per_wave *
                sctx->scratch_waves);

        radeon_add_to_buffer_list(sctx, sctx->gfx_cs,
                  shader->scratch_bo, RADEON_USAGE_READWRITE,
                  RADEON_PRIO_SCRATCH_BUFFER);
    }

    /* Prefetch the compute shader to TC L2.
     *
     * We should also prefetch graphics shaders if a compute dispatch was
     * the last command, and the compute shader if a draw call was the last
     * command. However, that would add more complexity and we're likely
     * to get a shader state change in that case anyway.
     */
    if (sctx->chip_class >= CIK) {
        cik_prefetch_TC_L2_async(sctx, &program->shader.bo->b.b,
                     0, program->shader.bo->b.b.width0);
    }

    shader_va = shader->bo->gpu_address + offset;
    if (program->use_code_object_v2) {
        /* Shader code is placed after the amd_kernel_code_t
         * struct. */
        shader_va += sizeof(amd_kernel_code_t);
    }
```

```
    radeon_add_to_buffer_list(sctx, sctx->gfx_cs, shader->bo,
                              RADEON_USAGE_READ, RADEON_PRIO_SHADER_BINARY);

    radeon_set_sh_reg_seq(cs, R_00B830_COMPUTE_PGM_LO, 2);
    radeon_emit(cs, shader_va >> 8);
    radeon_emit(cs, S_00B834_DATA(shader_va >> 40));

    radeon_set_sh_reg_seq(cs, R_00B848_COMPUTE_PGM_RSRC1, 2);
    radeon_emit(cs, config->rsrc1);
    radeon_emit(cs, config->rsrc2);

    COMPUTE_DBG(sctx->screen, "COMPUTE_PGM_RSRC1: 0x%08x "
        "COMPUTE_PGM_RSRC2: 0x%08x\n", config->rsrc1, config->rsrc2);

    radeon_set_sh_reg(cs, R_00B860_COMPUTE_TMPRING_SIZE,
              S_00B860_WAVES(sctx->scratch_waves)
                 | S_00B860_WAVESIZE(config->scratch_bytes_per_wave >> 10));

    sctx->cs_shader_state.emitted_program = program;
    sctx->cs_shader_state.offset = offset;
    sctx->cs_shader_state.uses_scratch =
        config->scratch_bytes_per_wave != 0;

    return true;
}
```

- 通过R_00B830_COMPUTE_PGM_LO将地址下发

cs 描述符资源上传

```
bool si_upload_compute_shader_descriptors(struct si_context *sctx)
{
    /* Does not update rw_buffers as that is not needed for compute shaders
     * and the input buffer is using the same SGPR's anyway.
     */
     // SI_DESCS_FIRST_COMPUTE = 11
     // SI_NUM_DESCS = 13
     // 设置11 12
    const unsigned mask = u_bit_consecutive(SI_DESCS_FIRST_COMPUTE,
                        SI_NUM_DESCS - SI_DESCS_FIRST_COMPUTE);
    return si_upload_shader_descriptors(sctx, mask);
        // 取const_and_shader_buffer,sample and image 交集
        unsigned dirty = sctx->descriptors_dirty & mask;

        /* Assume nothing will go wrong: */
        sctx->shader_pointers_dirty |= dirty;

        while (dirty) {
            unsigned i = u_bit_scan(&dirty);

            if (!si_upload_descriptors(sctx, &sctx->descriptors[i]))
                return false;
        }
```

```
        sctx->descriptors_dirty &= ~mask;

        si_upload_bindless_descriptors(sctx);

        return true;

}
```

cs 描述符资源地址寄存器下发

cs 的描述符资源地址和其他shader配置不同，通过si_emit_compute_shader_pointers指定地址位R_00B900_COMPUTE_USER_DATA_0:

```
void si_emit_compute_shader_pointers(struct si_context *sctx)
{
    unsigned base = R_00B900_COMPUTE_USER_DATA_0;

    si_emit_consecutive_shader_pointers(sctx, SI_DESCS_SHADER_MASK(COMPUTE),
                        R_00B900_COMPUTE_USER_DATA_0);
    sctx->shader_pointers_dirty &= ~SI_DESCS_SHADER_MASK(COMPUTE);

    ///
    if (sctx->compute_bindless_pointer_dirty) {
        si_emit_shader_pointer(sctx, &sctx->bindless_descriptors, base);
        sctx->compute_bindless_pointer_dirty = false;
    }
}
```

cs grid_info pm4 下发

```
static void si_emit_dispatch_packets(struct si_context *sctx,
                                     const struct pipe_grid_info *info)
{
    struct si_screen *sscreen = sctx->screen;
    struct radeon_cmdbuf *cs = sctx->gfx_cs;
    bool render_cond_bit = sctx->render_cond && !sctx->render_cond_force_off;
    unsigned waves_per_threadgroup = ...;
    unsigned compute_resource_limits = ...;

    if (sctx->chip_class >= CIK) {
        unsigned num_cu_per_se = sscreen->info.num_good_compute_units /
        compute_resource_limits |= S_00B854_WAVES_PER_SH(sctx-
>cs_max_waves_per_sh);
    } else {
    }

    radeon_set_sh_reg(cs, R_00B854_COMPUTE_RESOURCE_LIMITS,
              compute_resource_limits);

    radeon_set_sh_reg_seq(cs, R_00B81C_COMPUTE_NUM_THREAD_X, 3);
    radeon_emit(cs, S_00B81C_NUM_THREAD_FULL(info->block[0]));
    radeon_emit(cs, S_00B820_NUM_THREAD_FULL(info->block[1]));
```

```
    radeon_emit(cs, S_00B824_NUM_THREAD_FULL(info->block[2]));

    unsigned dispatch_initiator =
        S_00B800_COMPUTE_SHADER_EN(1) |
        S_00B800_FORCE_START_AT_000(1) |
        /* If the KMD allows it (there is a KMD hw register for it),
         * allow launching waves out-of-order. (same as Vulkan) */
        S_00B800_ORDER_MODE(sctx->chip_class >= CIK);

    if (info->indirect) {
        uint64_t base_va = r600_resource(info->indirect)->gpu_address;

        radeon_add_to_buffer_list(sctx, sctx->gfx_cs,
                        r600_resource(info->indirect),
                        RADEON_USAGE_READ, RADEON_PRIO_DRAW_INDIRECT);

        radeon_emit(cs, PKT3(PKT3_SET_BASE, 2, 0) |
                        PKT3_SHADER_TYPE_S(1));
        radeon_emit(cs, 1);
        radeon_emit(cs, base_va);
        radeon_emit(cs, base_va >> 32);

        radeon_emit(cs, PKT3(PKT3_DISPATCH_INDIRECT, 1, render_cond_bit) |
                        PKT3_SHADER_TYPE_S(1));
        radeon_emit(cs, info->indirect_offset);
        radeon_emit(cs, dispatch_initiator);
    } else {
        radeon_emit(cs, PKT3(PKT3_DISPATCH_DIRECT, 3, render_cond_bit) |
                        PKT3_SHADER_TYPE_S(1));
        radeon_emit(cs, info->grid[0]);
        radeon_emit(cs, info->grid[1]);
        radeon_emit(cs, info->grid[2]);
        radeon_emit(cs, dispatch_initiator);
    }
}
```

- 函数首先下发了 COMPUTE_RESOURCE_LIMITS ,3 个 COMPUTE_NUM_THREAD_X 计算寄存器
- 如果grid通过buffer 下发，则首先下发PKT3_SET_BASE 设定buffer bo 地址后再通过 PKT3_DISPATCH_INDIRECT 给出偏移量
- 否则直接通过PKT3_DISPATCH_DIRECT 下发参数grid , block
- 从上面可以知道， 当shader里面使用了对应的内建变量时会直接从参数SGPR寄存器里面获取数据

生成cs LLVM IR 时对barrier等的处理

barrier

```
static void si_llvm_emit_barrier(const struct lp_build_tgsi_action *action,
                 struct lp_build_tgsi_context *bld_base,
                 struct lp_build_emit_data *emit_data)
{
    struct si_shader_context *ctx = si_shader_context(bld_base);

    /* SI only (thanks to a hw bug workaround):
     * The real barrier instruction isn’t needed, because an entire patch
```

```
     * always fits into a single wave.
     */
    if (ctx->screen->info.chip_class == SI &&
        ctx->type == PIPE_SHADER_TESS_CTRL) {
        ac_build_waitcnt(&ctx->ac, LGKM_CNT & VM_CNT);
        return;
    }

    ac_build_s_barrier(&ctx->ac);
        ac_build_intrinsic(ctx, "llvm.amdgcn.s.barrier", ctx->voidt, NULL,
                   0, AC_FUNC_ATTR_CONVERGENT);
}
```

radeonsi 中使用cs进行blit操作

blit 分为两种一种是clear， 一种是copy, copy的情况是在上层调用resource_copy_region 接口时， 当目的为缓冲时，会采用cs copy或者cp dma copy
在调用glClear过程中如果使用dccclear则会调用cs进行clear操作
两者的最终都是通过si_compute_do_clear_or_copy 实现的。，其中对于拷贝操作没有src buffer

```
// si_compute_blit.c
// 由于作的是clear操作，所以src=null,src_offset=0
static void si_compute_do_clear_or_copy(struct si_context *sctx,
                    struct pipe_resource *dst,
                    unsigned dst_offset,
                    struct pipe_resource *src ,
                    unsigned src_offset,
                    unsigned size,
                    const uint32_t *clear_value,
                    unsigned clear_value_size,
                    enum si_coherency coher)
{
    struct pipe_context *ctx = &sctx->b;

    assert(src_offset % 4 == 0);
    assert(dst_offset % 4 == 0);
    assert(size % 4 == 0);

    assert(dst->target != PIPE_BUFFER || dst_offset + size <= dst->width0);
    assert(!src || src_offset + size <= src->width0);

    //
    sctx->flags |= SI_CONTEXT_PS_PARTIAL_FLUSH |
               SI_CONTEXT_CS_PARTIAL_FLUSH |
               si_get_flush_flags(sctx, coher, SI_COMPUTE_DST_CACHE_POLICY);


    /* Save states. */
    void *saved_cs = sctx->cs_shader_state.program;
    struct pipe_shader_buffer saved_sb[2] = {};
    // 提取ssbo
    si_get_shader_buffers(sctx, PIPE_SHADER_COMPUTE, 0, src ? 2 : 1, saved_sb);
```

```
     */
    /* 内存访问是协同的(coalsced)，这意味着第一条指令写入整个波浪的第一个连续数据块，
     * 第二条指令写入第二个连续的数据块，依此类推。
     */
     // 根据操作不同设定不同线程数
    unsigned dwords_per_thread = src ? SI_COMPUTE_COPY_DW_PER_THREAD :
                       SI_COMPUTE_CLEAR_DW_PER_THREAD;
    unsigned instructions_per_thread = MAX2(1, dwords_per_thread / 4);
    unsigned dwords_per_instruction = dwords_per_thread /
instructions_per_thread;
    unsigned dwords_per_wave = dwords_per_thread * 64;

    unsigned num_dwords = size / 4;
    unsigned num_instructions = DIV_ROUND_UP(num_dwords, dwords_per_instruction);

    struct pipe_grid_info info = {};
    info.block[0] = MIN2(64, num_instructions);
    info.block[1] = 1;
    info.block[2] = 1;
    info.grid[0] = DIV_ROUND_UP(num_dwords, dwords_per_wave);
    info.grid[1] = 1;
    info.grid[2] = 1;

    struct pipe_shader_buffer sb[2] = {};
    sb[0].buffer = dst;
    sb[0].buffer_offset = dst_offset;
    sb[0].buffer_size = size;

    bool shader_dst_stream_policy = SI_COMPUTE_DST_CACHE_POLICY != L2_LRU;

    if (src) {
        sb[1].buffer = src;
        sb[1].buffer_offset = src_offset;
        sb[1].buffer_size = size;

        //
        ctx->set_shader_buffers(ctx, PIPE_SHADER_COMPUTE, 0, 2, sb);

        if (!sctx->cs_copy_buffer) {
            sctx->cs_copy_buffer = si_create_dma_compute_shader(&sctx->b,
                                SI_COMPUTE_COPY_DW_PER_THREAD,
                                shader_dst_stream_policy, true);
        }
        ctx->bind_compute_state(ctx, sctx->cs_copy_buffer);
    } else {
        // clear_value_size 可以为 4, 8, 16
        assert(clear_value_size >= 4 &&
               clear_value_size <= 16 &&
               util_is_power_of_two_or_zero(clear_value_size));


        //设置清理填充的用户数据
        for (unsigned i = 0; i < 4; i++)
            sctx->cs_user_data[i] = clear_value[i % (clear_value_size / 4)];
```

```
        //设置写入的缓冲
        ctx->set_shader_buffers(ctx, PIPE_SHADER_COMPUTE, 0, 1, sb);

        ///* Tunables for compute-based clear_buffer and copy_buffer: */
        //#define SI_COMPUTE_CLEAR_DW_PER_THREAD    4
        //#define SI_COMPUTE_COPY_DW_PER_THREAD 4
        if (!sctx->cs_clear_buffer) {
            sctx->cs_clear_buffer = si_create_dma_compute_shader(&sctx->b,
                                SI_COMPUTE_CLEAR_DW_PER_THREAD,
                                shader_dst_stream_policy, false);
        }
        ctx->bind_compute_state(ctx, sctx->cs_clear_buffer);
    }

    // 开始执行cs
    ctx->launch_grid(ctx, &info);

    enum si_cache_policy cache_policy = get_cache_policy(sctx, coher, size);
    sctx->flags |= SI_CONTEXT_CS_PARTIAL_FLUSH |
               (cache_policy == L2_BYPASS ? SI_CONTEXT_WRITEBACK_GLOBAL_L2 : 0);

    if (cache_policy != L2_BYPASS)
        r600_resource(dst)->TC_L2_dirty = true;

    /* Restore states. */
    ctx->bind_compute_state(ctx, saved_cs);
    ctx->set_shader_buffers(ctx, PIPE_SHADER_COMPUTE, 0, src ? 2 : 1, saved_sb);
}
```

- cs_clear_buffer保存了创建的cs 程序状态

创建dma cs

```
/* Create a compute shader implementing clear_buffer or copy_buffer. */
void *si_create_dma_compute_shader(struct pipe_context *ctx,
                   unsigned num_dwords_per_thread= 4,
                   bool dst_stream_cache_policy, bool is_copy)
{
    assert(util_is_power_of_two_nonzero(num_dwords_per_thread));

    // 设定coherent restrict 限定符
    unsigned store_qualifier = TGSI_MEMORY_COHERENT | TGSI_MEMORY_RESTRICT;
    if (dst_stream_cache_policy)
        store_qualifier |= TGSI_MEMORY_STREAM_CACHE_POLICY;

    /* Don't cache loads, because there is no reuse. */
    unsigned load_qualifier = store_qualifier | TGSI_MEMORY_STREAM_CACHE_POLICY;

    unsigned num_mem_ops = MAX2(1, num_dwords_per_thread / 4);
    unsigned *inst_dwords = alloca(num_mem_ops * sizeof(unsigned));

    for (unsigned i = 0; i < num_mem_ops; i++) {
        if (i*4 < num_dwords_per_thread)
            inst_dwords[i] = MIN2(4, num_dwords_per_thread - i*4);
```

```
    }

    // 创建tgsi所需要的ureg_create cs
    struct ureg_program *ureg = ureg_create(PIPE_SHADER_COMPUTE);
    if (!ureg)
        return NULL;

    // 声明shader layout(local_size x);
    ureg_property(ureg, TGSI_PROPERTY_CS_FIXED_BLOCK_WIDTH, 64);
    ureg_property(ureg, TGSI_PROPERTY_CS_FIXED_BLOCK_HEIGHT, 1);
    ureg_property(ureg, TGSI_PROPERTY_CS_FIXED_BLOCK_DEPTH, 1);

    struct ureg_src value;
    if (!is_copy) {
        // 清除时直接使用cs_user_data参数传入grid
        ureg_property(ureg, TGSI_PROPERTY_CS_USER_DATA_DWORDS, inst_dwords[0]);
        value = ureg_DECL_system_value(ureg, TGSI_SEMANTIC_CS_USER_DATA, 0);
    }

    // 获取参数 local_invocation_ids
    struct ureg_src tid = ureg_DECL_system_value(ureg, TGSI_SEMANTIC_THREAD_ID,
0);
    // 获取(workgroup_ids0, workgroup_ids1, workgroup_ids2)
    struct ureg_src blk = ureg_DECL_system_value(ureg, TGSI_SEMANTIC_BLOCK_ID,
0);

    //构造两个临时变量temp.x
    struct ureg_dst store_addr = ureg_writemask(ureg_DECL_temporary(ureg),
TGSI_WRITEMASK_X);

    // load_adder用于拷贝操作
    struct ureg_dst load_addr = ureg_writemask(ureg_DECL_temporary(ureg),
TGSI_WRITEMASK_X);

    // 声明一个ssbo buffer变量，用于保存数据
    struct ureg_dst dstbuf = ureg_dst(ureg_DECL_buffer(ureg, 0, false));
    struct ureg_src srcbuf;
    struct ureg_src *values = NULL;

    if (is_copy) {
        srcbuf = ureg_DECL_buffer(ureg, 1, false);
        values = malloc(num_mem_ops * sizeof(struct ureg_src));
    }

    /* If there are multiple stores, the first store writes into 0+tid,
     * the 2nd store writes into 64+tid, the 3rd store writes into 128+tid, etc.
     */
     // umad操作存在在store_addr
     ///store_addr + blk + 64 * num_mem_ops + tid
    ureg_UMAD(ureg, store_addr, blk, ureg_imm1u(ureg, 64 * num_mem_ops), tid);
    /* Convert from a "store size unit" into bytes. */

    // umul 操作
    / 将存储大小单元转换为字节<进行缩放
    ureg_UMUL(ureg, store_addr, ureg_src(store_addr),
          ureg_imm1u(ureg, 4 * inst_dwords[0]));
```

```
    // mov 操作,获取最终地址
    ureg_MOV(ureg, load_addr, ureg_src(store_addr));


    // 非拷贝操作为0
    /* Distance between a load and a store for latency hiding. */
    unsigned load_store_distance = is_copy ? 8 : 0;

    for (unsigned i = 0; i < num_mem_ops + load_store_distance; i++) {
        int d = i - load_store_distance;

        if (is_copy && i < num_mem_ops) {
            if (i) {
                ureg_UADD(ureg, load_addr, ureg_src(load_addr),
                      ureg_imm1u(ureg, 4 * inst_dwords[i] * 64));
            }

            values[i] = ureg_src(ureg_DECL_temporary(ureg));
            struct ureg_dst dst =
                ureg_writemask(ureg_dst(values[i]),
                               u_bit_consecutive(0, inst_dwords[i]));
            struct ureg_src srcs[] = {srcbuf, ureg_src(load_addr)};
            ureg_memory_insn(ureg, TGSI_OPCODE_LOAD, &dst, 1, srcs, 2,
                     load_qualifier, TGSI_TEXTURE_BUFFER, 0);
        }

        if (d >= 0) {
            // 距离如果大于0,则将他加入到存储地址偏移量,用于拷贝操作
            if (d) {
                ureg_UADD(ureg, store_addr, ureg_src(store_addr),
                      ureg_imm1u(ureg, 4 * inst_dwords[d] * 64));
            }

            struct ureg_dst dst =
                ureg_writemask(dstbuf, u_bit_consecutive(0, inst_dwords[d]));
            struct ureg_src srcs[] =
                {ureg_src(store_addr), is_copy ? values[d] : value};
            ureg_memory_insn(ureg, TGSI_OPCODE_STORE, &dst, 1, srcs, 2,
                     store_qualifier, TGSI_TEXTURE_BUFFER, 0);
        }
    }
    ureg_END(ureg);

    struct pipe_compute_state state = {};
    state.ir_type = PIPE_SHADER_IR_TGSI;
    state.prog = ureg_get_tokens(ureg, NULL);

    void *cs = ctx->create_compute_state(ctx, &state);
    ureg_destroy(ureg);
    free(values);
    return cs;
}
```

- 该函数首先声明三个内建变量，用于计算加载地址
- 声明ssbo 绑定到clear dst buffer
- 加入了两个操作指令umad, umul 获取imagestore的坐标加载地址, 然后通过imagstore将user_data参数数据填入这个位置的buffer里面,达到清除目的
- 最终创建的cs tgsi如下

```
COMP
PROPERTY FS_COORD_PIXEL_CENTER 4
PROPERTY CS_FIXED_BLOCK_HEIGHT 1
PROPERTY CS_FIXED_BLOCK_DEPTH 1
DCL SV[0], 47
DCL SV[1], THREAD_ID
DCL SV[2], BLOCK_ID
DCL BUFFER[0]
DCL TEMP[0..1]
IMM[0] UINT32 {64, 16, 0, 0}
  0: UMAD TEMP[0].x, SV[2], IMM[0].xxxx, SV[1]
  1: UMUL TEMP[0].x, TEMP[0], IMM[0].yyyy
  2: MOV TEMP[1].x, TEMP[0]
  3: STORE BUFFER[0], TEMP[0], SV[0], COHERENT, RESTRICT, STREAM_CACHE_POLICY
  4: END
$6 = void
```

llvmpipe/softpipe

softpipe cs 接口

```
   pipe->create_compute_state = softpipe_create_compute_state;
   pipe->bind_compute_state = softpipe_bind_compute_state;
   pipe->delete_compute_state = softpipe_delete_compute_state;
}
   softpipe->pipe.launch_grid = softpipe_launch_grid;
```

cs的创建

```
/** Subclass of pipe_compute_state */
struct sp_compute_shader {
   struct pipe_compute_state shader;
   struct tgsi_token *tokens;
   struct tgsi_shader_info info;
   int max_sampler;             /* -1 if no samplers */
};


static void *
softpipe_create_compute_state(struct pipe_context *pipe,
                              const struct pipe_compute_state *templ)
{
   struct softpipe_context *softpipe = softpipe_context(pipe);
```

```
   const struct tgsi_token *tokens;
   struct sp_compute_shader *state;
   if (templ->ir_type != PIPE_SHADER_IR_TGSI)
      return NULL;

   tokens = templ->prog;
   /* debug */
   if (softpipe->dump_cs)
      tgsi_dump(tokens, 0);

   state = CALLOC_STRUCT(sp_compute_shader);

   state->shader = *templ;
   state->tokens = tgsi_dup_tokens(tokens);
   tgsi_scan_shader(state->tokens, &state->info);

   state->max_sampler = state->info.file_max[TGSI_FILE_SAMPLER];

   return state;
}
```

- 接口极为简单，生成tgsi_shader_info

cs 状态绑定

```
static void
softpipe_bind_compute_state(struct pipe_context *pipe,
                            void *cs)
{
   struct softpipe_context *softpipe = softpipe_context(pipe);
   struct sp_compute_shader *state = (struct sp_compute_shader *)cs;
   if (softpipe->cs == state)
      return;

   softpipe->cs = state;
}
```

softpipe_launch_grid

```
void
softpipe_launch_grid(struct pipe_context *context,
                     const struct pipe_grid_info *info)
{
   struct softpipe_context *softpipe = softpipe_context(context);
   struct sp_compute_shader *cs = softpipe->cs;
   int num_threads_in_group;
   struct tgsi_exec_machine **machines;
   int bwidth, bheight, bdepth;
   int w, h, d, i;
   int g_w, g_h, g_d;
   uint32_t grid_size[3] = {0};
   void *local_mem = NULL;
```

```
   softpipe_update_compute_samplers(softpipe);
        set_shader_sampler(softpipe, PIPE_SHADER_COMPUTE, softpipe->cs->max_sampler);
           int i;
           for (i = 0; i <= max_sampler; i++) {
              softpipe->tgsi.sampler[shader]->sp_sampler[i] =
                 (struct sp_sampler *)(softpipe->samplers[shader][i]);
           }


   bwidth = cs->info.properties[TGSI_PROPERTY_CS_FIXED_BLOCK_WIDTH];
   bheight = cs->info.properties[TGSI_PROPERTY_CS_FIXED_BLOCK_HEIGHT];
   bdepth = cs->info.properties[TGSI_PROPERTY_CS_FIXED_BLOCK_DEPTH];
   num_threads_in_group = bwidth * bheight * bdepth;

   fill_grid_size(context, info, grid_size);
   ----------------------------------------- fill_grid_size
       struct pipe_transfer *transfer;
       uint32_t *params;
       if (!info->indirect) {
          grid_size[0] = info->grid[0];
          grid_size[1] = info->grid[1];
          grid_size[2] = info->grid[2];
          return;
       }
       params = pipe_buffer_map_range(context, info->indirect,
                                      info->indirect_offset,
                                      3 * sizeof(uint32_t),
                                      PIPE_TRANSFER_READ,
                                      &transfer);

       if (!transfer)
          return;

       grid_size[0] = params[0];
       grid_size[1] = params[1];
       grid_size[2] = params[2];
       pipe_buffer_unmap(context, transfer);


   if (cs->shader.req_local_mem) {
      local_mem = CALLOC(1, cs->shader.req_local_mem);
   }

   machines = CALLOC(sizeof(struct tgsi_exec_machine *), num_threads_in_group);
   if (!machines) {
      FREE(local_mem);
      return;
   }

   /* initialise machines + GRID_SIZE + THREAD_ID  + BLOCK_SIZE */
   for (d = 0; d < bdepth; d++) {
      for (h = 0; h < bheight; h++) {
         for (w = 0; w < bwidth; w++) {
            int idx = w + (h * bwidth) + (d * bheight * bwidth);
            machines[idx] = tgsi_exec_machine_create(PIPE_SHADER_COMPUTE);
```

```
            machines[idx]->LocalMem = local_mem;
            machines[idx]->LocalMemSize = cs->shader.req_local_mem;
            cs_prepare(cs, machines[idx],
                       w, h, d,
                       grid_size[0], grid_size[1], grid_size[2],
                       bwidth, bheight, bdepth,
                       (struct tgsi_sampler *)softpipe-
>tgsi.sampler[PIPE_SHADER_COMPUTE],
                       (struct tgsi_image *)softpipe-
>tgsi.image[PIPE_SHADER_COMPUTE],
                       (struct tgsi_buffer *)softpipe-
>tgsi.buffer[PIPE_SHADER_COMPUTE]);
            tgsi_exec_set_constant_buffers(machines[idx],
PIPE_MAX_CONSTANT_BUFFERS,
                                           softpipe-
>mapped_constants[PIPE_SHADER_COMPUTE],
                                           softpipe-
>const_buffer_size[PIPE_SHADER_COMPUTE]);
         }
      }
   }

   for (g_d = 0; g_d < grid_size[2]; g_d++) {
      for (g_h = 0; g_h < grid_size[1]; g_h++) {
         for (g_w = 0; g_w < grid_size[0]; g_w++) {
            run_workgroup(cs, g_w, g_h, g_d, num_threads_in_group, machines);
         }
      }
   }

}
```

- 接口内部首先通过创建一个tgsi执行引擎来运行tgsi
- 之后通过run_workgroup 执行cs

创建tgsi执行引擎并初始化

```
static void
cs_prepare(const struct sp_compute_shader *cs,
           struct tgsi_exec_machine *machine,
           int w, int h, int d,
           int g_w, int g_h, int g_d,
           int b_w, int b_h, int b_d,
           struct tgsi_sampler *sampler,
           struct tgsi_image *image,
           struct tgsi_buffer *buffer )
{
   int j;
   /*
    * Bind tokens/shader to the interpreter's machine state.
    */
   tgsi_exec_machine_bind_shader(machine,
                                 cs->tokens,
                                 sampler, image, buffer);
```

```
   if (machine->SysSemanticToIndex[TGSI_SEMANTIC_THREAD_ID] != -1) {
      unsigned i = machine->SysSemanticToIndex[TGSI_SEMANTIC_THREAD_ID];
      for (j = 0; j < TGSI_QUAD_SIZE; j++) {
         machine->SystemValue[i].xyzw[0].i[j] = w;
         machine->SystemValue[i].xyzw[1].i[j] = h;
         machine->SystemValue[i].xyzw[2].i[j] = d;
      }
   }

   if (machine->SysSemanticToIndex[TGSI_SEMANTIC_GRID_SIZE] != -1) {
      unsigned i = machine->SysSemanticToIndex[TGSI_SEMANTIC_GRID_SIZE];
      for (j = 0; j < TGSI_QUAD_SIZE; j++) {
         machine->SystemValue[i].xyzw[0].i[j] = g_w;
         machine->SystemValue[i].xyzw[1].i[j] = g_h;
         machine->SystemValue[i].xyzw[2].i[j] = g_d;
      }
   }

   if (machine->SysSemanticToIndex[TGSI_SEMANTIC_BLOCK_SIZE] != -1) {
      unsigned i = machine->SysSemanticToIndex[TGSI_SEMANTIC_BLOCK_SIZE];
      for (j = 0; j < TGSI_QUAD_SIZE; j++) {
         machine->SystemValue[i].xyzw[0].i[j] = b_w;
         machine->SystemValue[i].xyzw[1].i[j] = b_h;
         machine->SystemValue[i].xyzw[2].i[j] = b_d;
      }
   }
}
```

将tgsi执行引擎绑定到cs

```
static void
cs_prepare(const struct sp_compute_shader *cs,
           struct tgsi_exec_machine *machine,
           int w, int h, int d,
           int g_w, int g_h, int g_d,
           int b_w, int b_h, int b_d,
           struct tgsi_sampler *sampler,
           struct tgsi_image *image,
           struct tgsi_buffer *buffer )
{
   int j;
   /*
    * Bind tokens/shader to the interpreter's machine state.
    */
   tgsi_exec_machine_bind_shader(machine,
                                 cs->tokens,
                                 sampler, image, buffer);

   if (machine->SysSemanticToIndex[TGSI_SEMANTIC_THREAD_ID] != -1) {
      unsigned i = machine->SysSemanticToIndex[TGSI_SEMANTIC_THREAD_ID];
```

```
      for (j = 0; j < TGSI_QUAD_SIZE; j++) {
         machine->SystemValue[i].xyzw[0].i[j] = w;
         machine->SystemValue[i].xyzw[1].i[j] = h;
         machine->SystemValue[i].xyzw[2].i[j] = d;
      }
   }

   if (machine->SysSemanticToIndex[TGSI_SEMANTIC_GRID_SIZE] != -1) {
      unsigned i = machine->SysSemanticToIndex[TGSI_SEMANTIC_GRID_SIZE];
      for (j = 0; j < TGSI_QUAD_SIZE; j++) {
         machine->SystemValue[i].xyzw[0].i[j] = g_w;
         machine->SystemValue[i].xyzw[1].i[j] = g_h;
         machine->SystemValue[i].xyzw[2].i[j] = g_d;
      }
   }

   if (machine->SysSemanticToIndex[TGSI_SEMANTIC_BLOCK_SIZE] != -1) {
      unsigned i = machine->SysSemanticToIndex[TGSI_SEMANTIC_BLOCK_SIZE];
      for (j = 0; j < TGSI_QUAD_SIZE; j++) {
         machine->SystemValue[i].xyzw[0].i[j] = b_w;
         machine->SystemValue[i].xyzw[1].i[j] = b_h;
         machine->SystemValue[i].xyzw[2].i[j] = b_d;
      }
   }
}
```

- 这一步主要是将他cs的sampler_image，buffer等资源设定到tgsi_执行引擎
- tgsi执行引擎中使用的tgsi_sampler, tgsi_image, tgsi_buffer 等变量来保存这些资源
- 根据外部设置的tgsi_sampler , tgsi_image , tgsi_buffer中的接口来实现加载，存储, op资源操作
- tgsi引擎将内建变量的值保存在SystemValue

运行cs工作组及tgsi执行引擎

```
static void
run_workgroup(const struct sp_compute_shader *cs,
              int g_w, int g_h, int g_d, int num_threads,
              struct tgsi_exec_machine **machines)
{
   int i;
   bool grp_hit_barrier, restart_threads = false;

   do {
      grp_hit_barrier = false;
      for (i = 0; i < num_threads; i++) {
         grp_hit_barrier |= cs_run(cs, g_w, g_h, g_d, machines[i],
restart_threads);
      }
      restart_threads = false;
      if (grp_hit_barrier) {
         grp_hit_barrier = false;
         restart_threads = true;
```

```
      }
   } while (restart_threads);
}
```

```
static bool
cs_run(const struct sp_compute_shader *cs,
       int g_w, int g_h, int g_d,
       struct tgsi_exec_machine *machine, bool restart)
{
   if (!restart) {
      if (machine->SysSemanticToIndex[TGSI_SEMANTIC_BLOCK_ID] != -1) {
         unsigned i = machine->SysSemanticToIndex[TGSI_SEMANTIC_BLOCK_ID];
         int j;
         for (j = 0; j < TGSI_QUAD_SIZE; j++) {
            machine->SystemValue[i].xyzw[0].i[j] = g_w;
            machine->SystemValue[i].xyzw[1].i[j] = g_h;
            machine->SystemValue[i].xyzw[2].i[j] = g_d;
         }
      }
      machine->NonHelperMask = (1 << 1) - 1;
   }

   tgsi_exec_machine_run(machine, restart ? machine->pc : 0);

   if (machine->pc != -1)
      return true;
   return false;
}
```

环境变量设置

llvmpipe dump cs

boo : SOFTPIPE_DUMP_CS

调试

1. 使用piglit ext_framebuffer_multisample-clear 测试cs blit clear程序
2. 使用simple-barrier.shader_test快速测试 cs program

amd cs register

COMP:COMPUTE_DIM_X · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb804

COMP:COMPUTE_DIM_X 寄存器是一个可写的 32 位寄存器，用于指定 X 维中的线程组数量。该寄存器的地址为 0xb804。

字段名称	位范围	默认值	描述
SIZE	31:0	无	线程组数量的 X 维度，如果设置为 0 或小于等于 START_X，则不会分派任何工作。

COMP:COMPUTE_DIM_Y · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb808

COMP:COMPUTE_DIM_Y 寄存器是一个可写的 32 位寄存器，用于指定 Y 维中的线程组数量。该寄存器的地址为 0xb808。

以下是字段的定义：

字段名称	位范围	默认值	描述
SIZE	31:0	无	线程组数量的 Y 维度，如果设置为 0 或小于等于 START_Y，则不会分派任何工作。

COMP:COMPUTE_DIM_Z · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb80c

COMP:COMPUTE_DIM_Z 寄存器是一个可写的 32 位寄存器，用于指定 Z 维中的线程组数量。该寄存器的地址为 0xb80c。

以下是字段的定义：

字段名称	位范围	默认值	描述
SIZE	31:0	无	线程组数量的 Z 维度，如果设置为 0 或小于等于 START_Z，则不会分派任何工作。

COMP:COMPUTE_DISPATCH_INITIATOR · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb800

COMP:COMPUTE_DISPATCH_INITIATOR 寄存器是一个可写的 32 位寄存器，用于基于当前计算状态处理一个 Dispatch Command。

以下是字段的定义：

字段名称	位范围	默认值	描述
COMPUTE_SHADER_EN	0	无	如果为 1，则处理此 dispatch 启动器。如果为 0，则丢弃它。
PARTIAL_TG_EN	1	无	如果为 1，则遵循部分线程组设置，如果为 0，则忽略它们。

字段名称	位范围	默认值	描述
FORCE_START_AT_000	2	无	如果为 1，则覆盖每个 COMPUTE_START_X/Y/Z 为 0。
ORDERED_APPEND_ENBL	3	无	如果为 1，则支持有序附加（IA 将为每个线程组生成一个 wave_id 基值，SPI 随后将使用此值为线程组生成的每个 wave 生成唯一值。此值加载到 SGPR）。

COMP:COMPUTE_MAX_WAVE_ID · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb82c

COMP:COMPUTE_MAX_WAVE_ID 寄存器是一个可写的 32 位寄存器，用于生成 CS 波的 max wave_id（有序附加项）值，作为 SGPR 输入项。

以下是字段的定义：

字段名称	位范围	默认值	描述
MAX_WAVE_ID	11:0	0x320	通常应设置为（NUM_SE * NUM_SH_PER_SE * NUM_CU_PER_SH * 4 * NUM_WAVES_PER_SIMD）-1。写入此寄存器将内部 cs-wave-id 计数器重置为 0。

COMP:COMPUTE_NUM_THREAD_X · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb81c

COMP:COMPUTE_NUM_THREAD_X 寄存器是一个可写的 32 位寄存器，用于指定计算着色器线程组的 X 维度。1 表示 1 个线程，0 是无效设置。最大为 2k，*XYZ* 最大为 2k。

以下是字段的定义：

字段名称	位范围	默认值	描述
NUM_THREAD_FULL	15:0	无	在 X 维度中线程组为满时使用的维度（PARTIAL_TG_EN == 0 或 tgid.X < COMPUTE_DIM_X）。
NUM_THREAD_PARTIAL	31:16	无	在 X 维度中线程组为部分时使用的维度（PARTIAL_TG_EN == 1 并且 tgid.X == COMPUTE_DIM_X）。

COMP:COMPUTE_NUM_THREAD_Y · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb820

COMP:COMPUTE_NUM_THREAD_Y 寄存器是一个可写的 32 位寄存器，用于指定计算着色器线程组的 Y 维度。1 表示 1 个线程，0 是无效设置。最大为 2k，*XYZ* 最大为 2k。

以下是字段的定义：

字段名称	位范围	默认值	描述
NUM_THREAD_FULL	15:0	无	在 Y 维度中线程组为满时使用的维度（PARTIAL_TG_EN == 0 或 tgid.Y < COMPUTE_DIM_Y）。
NUM_THREAD_PARTIAL	31:16	无	在 Y 维度中线程组为部分时使用的维度（PARTIAL_TG_EN == 1 并且 tgid.Y == COMPUTE_DIM_Y）。

COMP:COMPUTE_NUM_THREAD_Z · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb824

COMP:COMPUTE_NUM_THREAD_Z 寄存器是一个可写的 32 位寄存器，用于指定计算着色器线程组的 Z 维度。1 表示 1 个线程，0 是无效设置。最大为 2k，*XYZ* 最大为 2k。

以下是字段的定义：

字段名称	位范围	默认值	描述
NUM_THREAD_FULL	15:0	无	在 Z 维度中线程组为满时使用的维度（PARTIAL_TG_EN == 0 或 tgid.Z < COMPUTE_DIM_Z）。
NUM_THREAD_PARTIAL	31:16	无	在 Z 维度中线程组为部分时使用的维度（PARTIAL_TG_EN == 1 并且 tgid.Z == COMPUTE_DIM_Z）。

COMP:COMPUTE_PGM_HI · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb834

COMP:COMPUTE_PGM_HI 寄存器是一个可写的 32 位寄存器，用于指定计算着色器程序的高位数据。

以下是字段的定义：

字段名称	位范围	默认值	描述
DATA	7:0	无	无

COMP:COMPUTE_PGM_LO · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb830

COMP:COMPUTE_PGM_LO 寄存器是一个可写的 32 位寄存器，用于指定计算着色器程序的低位数据。

以下是字段的定义：

字段名称	位范围	默认值	描述
DATA	31:0	无	无

COMP:COMPUTE_PGM_RSRC1 · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb848

COMP:COMPUTE_PGM_RSRC1 寄存器是一个可写的 32 位寄存器，用于指定计算着色器的程序设置。

以下是字段的定义：

字段名称	位范围	默认值	描述
VGPRS	5:0	无	VGPRS 数量，以 4 的粒度分配。范围是从 0 到 63，分配 4、8、12 等。
SGPRS	9:6	无	SGPRS 数量，以 8 的粒度分配。范围是从 0 到 15，分配 8、16、24 等。
PRIORITY	11:10	无	驱动 spi_priority 在 spi_sq newWave 命令中的优先级。
FLOAT_MODE	19:12	无	驱动 spi_sq newWave 命令中的 float_mode。
PRIV	20	无	驱动 spi_sq newWave 命令中的 priv。
DX10_CLAMP	21	无	驱动 spi_sq newWave 命令中的 dx10_clamp。
DEBUG_MODE	22	无	驱动 spi_sq newWave 命令中的 debug。
IEEE_MODE	23	无	驱动 spi_sq newWave 命令中的 ieee。

COMP:COMPUTE_PGM_RSRC2 · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb84c

COMP:COMPUTE_PGM_RSRC2 寄存器是一个可写的 32 位寄存器，用于指定计算着色器的程序设置。

以下是字段的定义：

字段名称	位范围	默认值	描述
SCRATCH_EN	0	无	此波使用 scratch 空间进行寄存器溢出。
USER_SGPR	5:1	无	SPI 应初始化的 USER_DATA 项的数量。范围是 0 到 16。

字段名称	位范围	默认值	描述
TRAP_PRESENT	6	无	启用陷阱处理。将 trap_en 位设置为 SQ，并导致 SPI 分配 16 个额外的 SGPR，并将 TBA/TMA 值写入 SGPR。
TGID_X_EN	7	无	启用将 TGID.X 加载到 SGPR。
TGID_Y_EN	8	无	启用将 TGID.Y 加载到 SGPR。
TGID_Z_EN	9	无	启用将 TGID.Z 加载到 SGPR。
TG_SIZE_EN	10	无	启用将与线程组相关的信息加载到 SGPR。详情请参阅着色器程序指南。
TIDIG_COMP_CNT	12:11	无	指定要写入 VGPR 的 thread_id_in_group 项的数量。0=X，1=XY，2=XYZ，3=未定义。
LDS_SIZE	23:15	无	为每个线程组分配的 LDS 空间量。以 64 为粒度，范围是从 0 到 128，分配 0 到 8K 个字。
EXCP_EN	30:24	无	驱动 spi_sq newWave 命令中的 excp 位。

COMP:COMPUTE_RESOURCE_LIMITS · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb854

COMP:COMPUTE_RESOURCE_LIMITS 寄存器是一个可写的 32 位寄存器，用于设置计算着色器的资源限制和锁定阈值。

以下是字段的定义：

字段名称	位范围	默认值	描述
WAVES_PER_SH	5:0	无	每个 SH 的 CS 波数限制，格式为 [9:4]。设置为 1 表示 16 波，设置为 63 表示 1008，设置为 0 表示禁用限制。
TG_PER_CU	15:12	无	每个 CU 的 CS 线程组限制。范围是 1 到 15，设置为 0 表示禁用限制。
LOCK_THRESHOLD	21:16	无	设置每个 SH 的低锁定阈值。以 4 为粒度，设置为 0 表示禁用锁定。如果 CS 的活动波数少于其设置，并且其分配不适合，则它可以锁定一个 CU，并阻止其他阶段分配给该 CU。
SIMD_DEST_CNTL	22	无	0 = 如果与目标 CU 的先前启动存在冲突，则调整首选的 SIMD；1 = 不要调整，始终优先 DEST SIMD。

COMP:COMPUTE_START_X · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb810

COMP:COMPUTE_START_X 寄存器是一个可写的 32 位寄存器，用于指定计算线程组在 X 维度上的起始位置。

以下是字段的定义：

字段名称	位范围	默认值	描述
START	31:0	无	X 维度上线程组的起始位置；通常设置为零。用作线程组创建的 X 维度的起始索引。

COMP:COMPUTE_START_Y · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb814

COMP:COMPUTE_START_Y 寄存器是一个可写的 32 位寄存器，用于指定计算线程组在 Y 维度上的起始位置。

以下是字段的定义：

字段名称	位范围	默认值	描述
START	31:0	无	Y 维度上线程组的起始位置；通常设置为零。

COMP:COMPUTE_START_Z · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb818

COMP:COMPUTE_START_Z 寄存器是一个可写的 32 位寄存器，用于指定计算线程组在 Z 维度上的起始位置。

以下是字段的定义：

字段名称	位范围	默认值	描述
START	31:0	无	Z 维度上线程组的起始位置；通常设置为零。

COMP:COMPUTE_STATIC_THREAD_MGMT_SE0 · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb858

COMP:COMPUTE_STATIC_THREAD_MGMT_SE0 寄存器是一个可写的 32 位寄存器，用于配置计算着色器（CS）在 SE0（Shader Engine 0）上的每个 CU 的启用情况。

以下是字段的定义：

字段名称	位范围	默认值	描述
SH0_CU_EN	15:0	0xFFFF	SH0 上每个 CU 的启用掩码。
SH1_CU_EN	31:16	0xFFFF	当存在时，SH1 上每个 CU 的启用掩码。

COMP:COMPUTE_STATIC_THREAD_MGMT_SE1 · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb85c

COMP:COMPUTE_STATIC_THREAD_MGMT_SE1 寄存器是一个可写的 32 位寄存器，用于配置计算着色器（CS）在 SE1（Shader Engine 1）上的每个 CU 的启用情况。

以下是字段的定义：

字段名称	位范围	默认值	描述
SH0_CU_EN	15:0	0xFFFF	SH0 上每个 CU 的启用掩码。
SH1_CU_EN	31:16	0xFFFF	当存在时，SH1 上每个 CU 的启用掩码。

COMP:COMPUTE_TBA_HI · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb83c

COMP:COMPUTE_TBA_HI 寄存器是一个可写的 32 位寄存器，用于配置计算着色器（CS）的 TBA（Trap Buffer Address）寄存器的高 8 位。

以下是字段的定义：

字段名称	位范围	默认值	描述
DATA	7:0	无	TBA 寄存器的高 8 位。

COMP:COMPUTE_TBA_LO · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb838

COMP:COMPUTE_TBA_LO 寄存器是一个可写的 32 位寄存器，用于配置计算着色器（CS）的 TBA（Trap Buffer Address）寄存器的低 32 位。

以下是字段的定义：

字段名称	位范围	默认值	描述
DATA	31:0	无	TBA 寄存器的低 32 位。

COMP:COMPUTE_TMA_HI · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb844

COMP:COMPUTE_TMA_HI 寄存器是一个可写的 32 位寄存器，用于配置计算着色器（CS）的 TMA（Trap Minimum Address）寄存器的高 8 位。

以下是字段的定义：

字段名称	位范围	默认值	描述
DATA	7:0	无	TMA 寄存器的高 8 位。

COMP:COMPUTE_TMA_LO · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb840

COMP:COMPUTE_TMA_LO 寄存器是一个可写的 32 位寄存器，用于配置计算着色器（CS）的 TMA（Trap Minimum Address）寄存器的低 32 位。

以下是字段的定义：

字段名称	位范围	默认值	描述
DATA	31:0	无	TMA 寄存器的低 32 位。

COMP:COMPUTE_TMPRING_SIZE · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb860

COMP:COMPUTE_TMPRING_SIZE 寄存器是一个可写的 32 位寄存器，用于配置计算着色器（CS）的临时环大小。

以下是字段的定义：

字段名称	位范围	默认值	描述
WAVES	11:0	无	分配区域的总大小，以波的数量表示。最大为 1024，适用于 Tahiti（Tahiti 有 32 个 CU，每个 CU 最多可以为 32 个波分配临时缓冲区）。
WAVESIZE	24:12	无	每个波使用的空间大小，以 dwords 为单位。以 256 个字为单位。该字段大小支持范围为 0 -> (2M-256) 字，每个波。

COMP:COMPUTE_USER_DATA_[0-15] · [W] · 32 bits · Access: 32 · GpuF0MMReg:0xb900-0xb93c

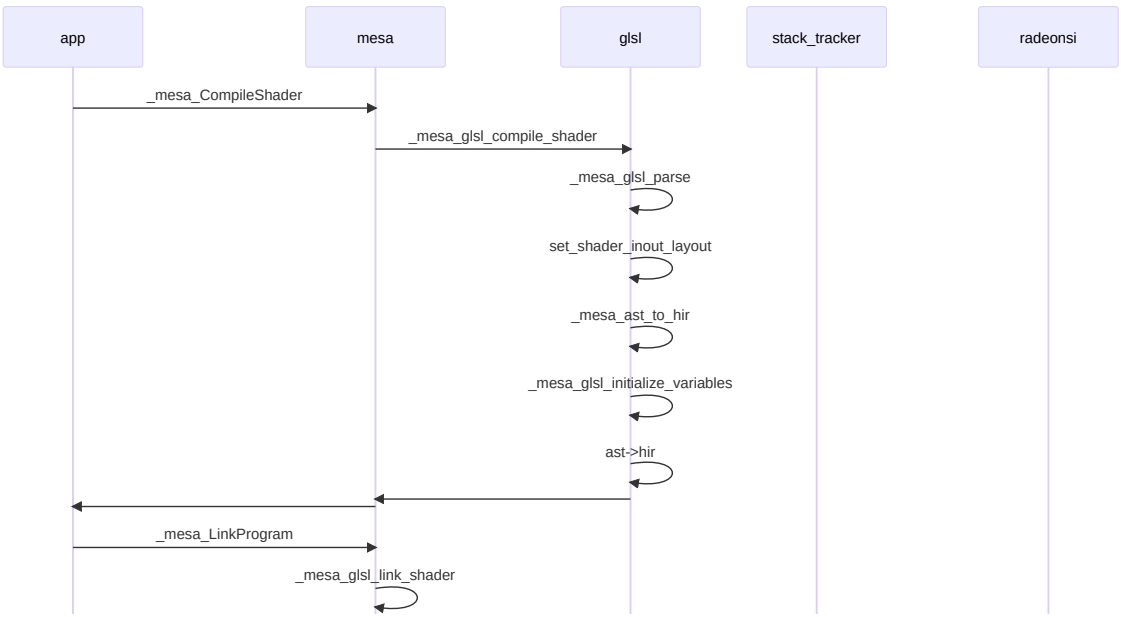
COMP:COMPUTE_USER_DATA_[0-15] 寄存器是可写的 32 位寄存器，用于设置计算着色器的用户数据。这组寄存器允许配置 16 个用户数据寄存器，每个寄存器包含 32 位数据。

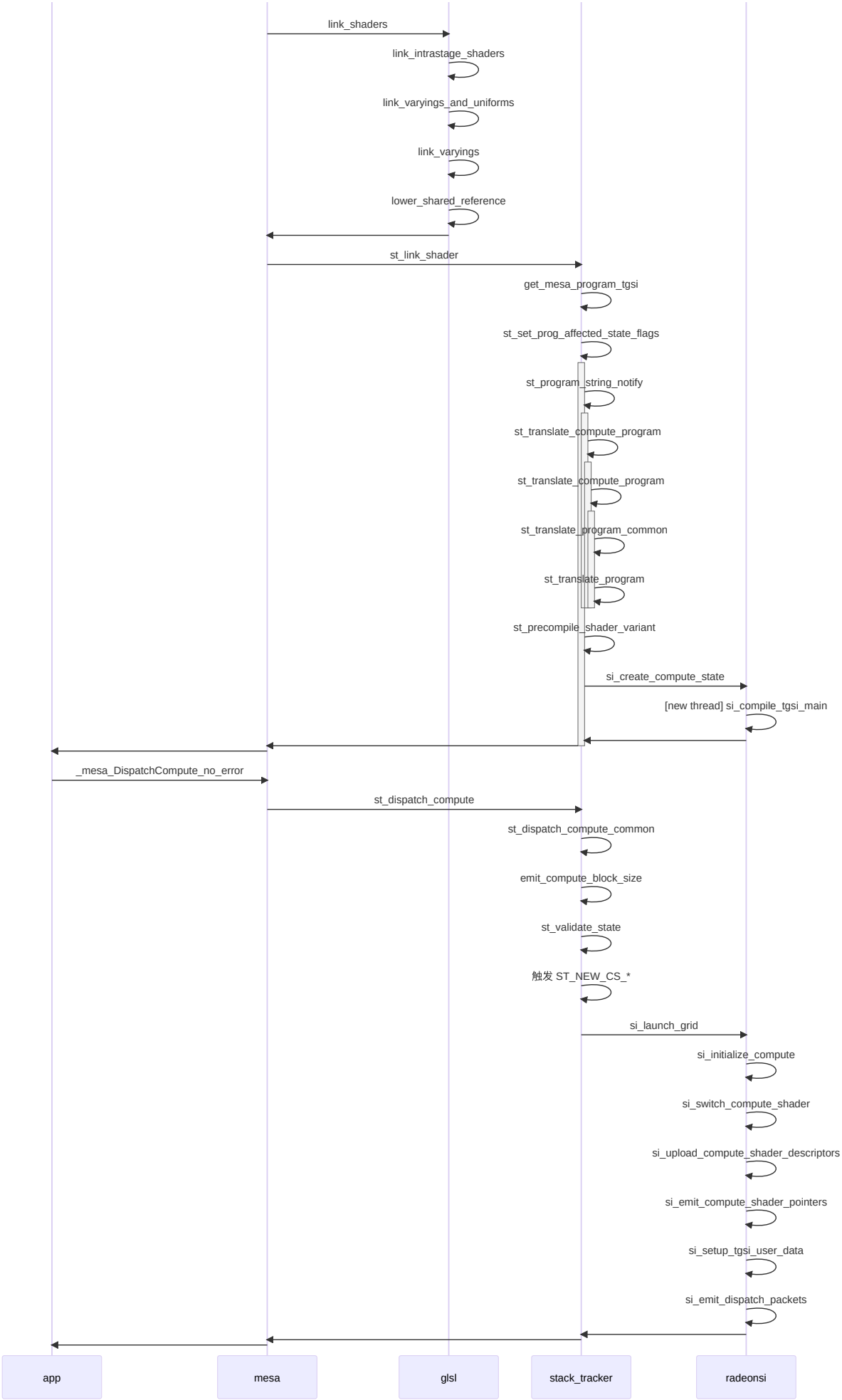
以下是字段的定义：

字段名称	位范围	默认值	描述
DATA	31:0	无	用户数据。

每个 **COMP:COMPUTE_USER_DATA_[0-15]** 寄存器都对应一个特定的用户数据寄存器。你可以根据需要设置这些寄存器，以便在计算着色器中使用相应的用户数据。

cs使用流程图


app
mesa
glsl
stack_tracker
radeonsi
_mesa_CompileShader
_mesa_glsl_compile_shader
_mesa_glsl_parse
set_shader_inout_layout
_mesa_ast_to_hir
_mesa_glsl_initialize_variables
ast->hir
_mesa_LinkProgram
_mesa_glsl_link_shader


link_shaders
link_intrastage_shaders
link_varyings_and_uniforms
link_varyings
lower_shared_reference
st_link_shader
get_mesa_program_tgsi
st_set_prog_affected_state_flags
st_program_string_notify
st_translate_compute_program
st_translate_compute_program
st_translate_program_common
st_translate_program
st_precompile_shader_variant
si_create_compute_state
[new thread] si_compile_tgsi_main
_mesa_DispatchCompute_no_error
st_dispatch_compute
st_dispatch_compute_common
emit_compute_block_size
st_validate_state
触发 ST_NEW_CS_*
si_launch_grid
si_initialize_compute
si_switch_compute_shader
si_upload_compute_shader_descriptors
si_emit_compute_shader_pointers
si_setup_tgsi_user_data
si_emit_dispatch_packets
app
mesa
glsl
stack_tracker
radeonsi