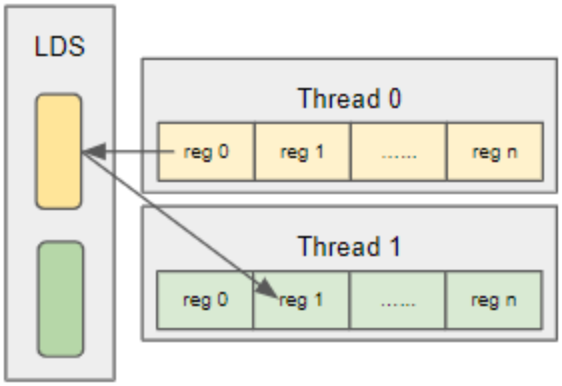
在LLVM后端实现跨通道数据搬移

AMD GPU的每个CU有一个64kB的存储空间，称为本地数据共享（Local Data Share，LDS），用于同一计算单元中的work group内各个work item之间的低延迟通信和数据共享。LDS配置为32个bank，每个bank有512个4字节的条目（entry）。尽管有LDS，大多数实际的计算指令仍对寄存器中的数据进行操作。从峰值性能的角度看，global memory带宽比LDS带宽小一个数量级，LDS带宽又比寄存器带宽小一个数量级。从延迟角度看，global memory的延迟约为500 cycles，LDS的延迟约为5 cycles，寄存器的延迟约为1 cycle。因此，寄存器是访问速度最快的存储类型。如果能通过寄存器实现数据共享，相较通过LDS实现数据共享，寄存器无疑是一种更高效的方式。如下图所示，使用LDS需要一次读，一次写，并且需要一个额外的寄存器来保存这些地址。寄存器为线程私有，需要软硬件提供额外的支持，才能实现在寄存器中进行跨通道数据搬移，实现比共享内存更高的效率。CUDA的__shfl_*()和AMD的__amdgcn_ds_*() intrinsic函数族为上述功能提供了warp/wavefront级原语软件支持。



1. CUDA中的Warp Shuffle函数定义

为了支持跨通道数据搬移，CUDA在Warp级原语（Warp-Level Primitives）中定义Shuffle函数，函数声明如下：

```
T __shfl_sync(unsigned mask, T var, int srcLane, int width=warpSize);
T __shfl_up_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=warpSize);
```

函数声明中的模板参数T可以是int、unsigned int、long、unsigned long、long long、unsigned long long、float或double等。

__shfl_*() intrinsic函数族允许在不使用共享内存的情况下，在warp中的线程之间通过直接读其它线程的寄存器值交换数据。一个warp中包含32通道（lane），每一个线程占用一个通道。交换会同时在warp内的所有活跃线程进行。由参数mask规定哪些线程为活跃线程，mask有32 bit，每个bit对应warp中的一个通道，1表示线程在活跃子集中。如果warp中所有线程都是活跃线程，则mask为0xffffffff。根据类型，每个线程移动4或8字节的数据。这种数据共享方式引入的线程间数据延迟极低，比通过共享内存进行线程间通讯的效果更好、延迟更低，同时也不消耗额外的内存资源来执行数据交换。

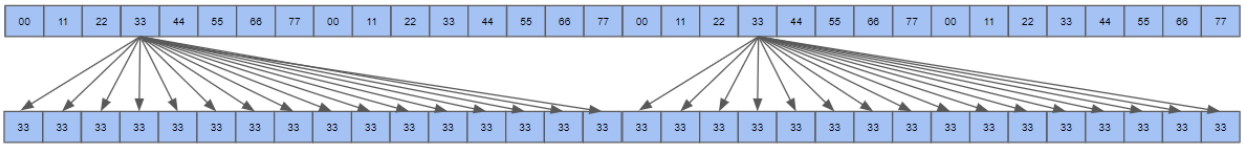
线程只能从正在活跃参与__shfl_*()命令的另一个线程中读取数据。如果目标线程处于非活跃状态，则检索到的值不确定。

所有__shfl_*()函数都有一个可选的width参数，该参数会更改内部函数的行为。width必须是2的幂的值。如果width不是2的幂或大于warpSize的数字，则结果不确定。

__shfl_sync()返回同一个warp中ID为srcLane对应的通道（线程）中的var值。如果width小于warpSize，则将warp分割为大小等于width的若干个线程子集，每个子集作为一个单独的实体，其起始逻辑通道ID为0。如果srcLane不在[0:width-1]范围内，则将srcLane对width取模的结果作为lane id，返回的值是对应于该id的通道持有的var值。

下例所示是将通道3中的值33广播到warp中的所有通道，width设为16。

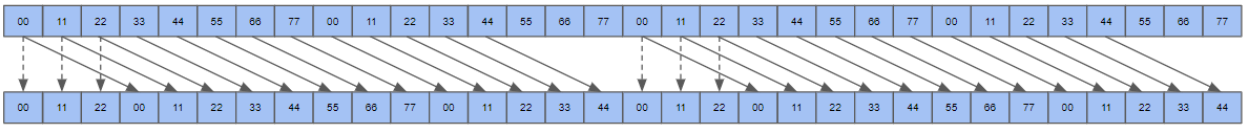
```
T __shfl_sync(unsigned mask, T var, int srcLane, int width=warpSize);

value = __shfl_sync(0xffffffff, value, 3, 16);
```



__shfl_up_sync()通过将调用者的通道ID值中减去delta值来计算源通道ID，即，对于处于 x 通道的调用者线程，__shfl_up_sync() 返回在同一个warp中的第 x - delta通道的var值。如果width小于warpSize，则将warp分割为大小等于width的若干个线程子集，每个子集作为一个单独的实体，其起始逻辑通道ID为0。如果源通道ID不在[0:width-1]范围内，源通道ID将不会对width取模。因此，warp中的低端delta个通道中的值保持不变。

下例所示是将通道x-3中的值返回给通道x，width设为16。调用函数后，低端3个通道中的值保持不变。

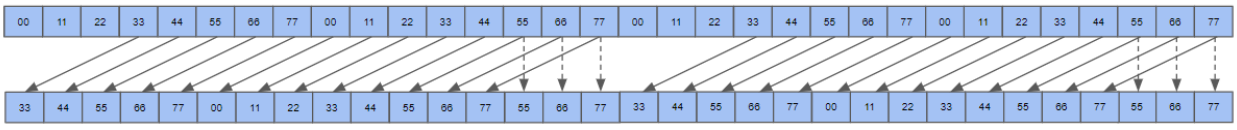
```
T __shfl_up_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);

int n = __shfl_up_sync(0xffffffff, value, 3, 16);
```



__shfl_down_sync()通过将调用者的通道ID值中加上delta值来计算源通道ID，即，对于处于 x 通道的调用者线程，__shfl_down_sync() 返回在同一个warp中的第 x + delta通道的var值。如果width小于warpSize，则warp的每个子部分都将作为一个单独的实体，其起始逻辑通道ID为0。如果源通道ID不在[0:width-1]范围内，源通道ID将不会对width取模。因此，warp中的高端delta个通道中的值保持不变。

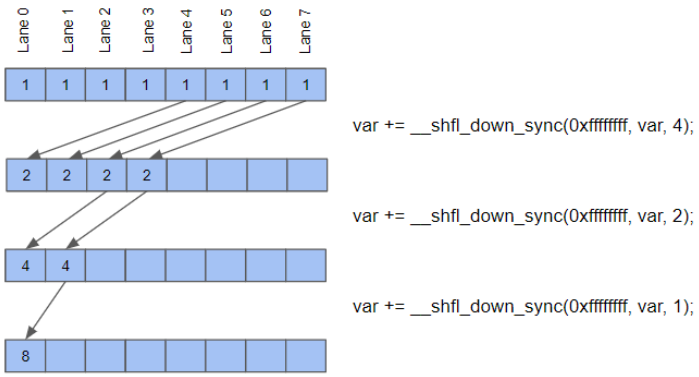
下例所示是将通道x+3中的值返回给通道x，width设为16。调用函数后，高端3个通道中的值保持不变。

```
T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);

int n = __shfl_down_sync(0xffffffff, value, 3, 16);
```



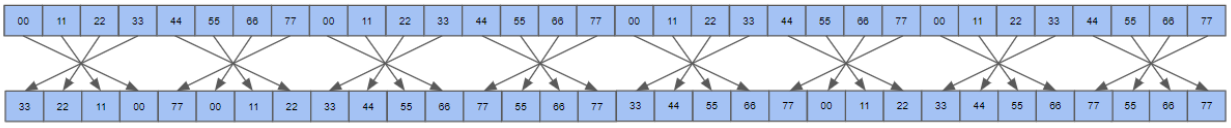
下面的例子使用__shfl_down_sync()，以tree-reduction方式计算warp中线程val值的和。代码执行后，warp中的第一个线程中的val值等于最终的和。

```
for (int delta = 4; delta > 0; delta /= 2)
  var += __shfl_down_sync(0xffffffff, var, delta);
```



__shfl_xor_sync()通过对调用者的通道ID与laneMask进行按位异或（XOR）运算来计算源通道ID。返回值为计算所得源通道中的var值。如果*width*小于*warpSize*，则每个*group*中的*width*个连续线程可以访问前一线程组中的元素，但是，如果试图访问后一线程组中的元素，则将返回其自身的*var*值。此模式实现了蝶形寻址模式。
下例所示是以laneMask=3实现蝶形寻址。例如，对于通道0，0与3的异或结果仍为3。因此，调用函数后，通道0的返回值为通道3中的33；对于通道1，1与3的异或结果为2。因此，调用函数后，通道1的返回值为通道2中的22。

```
T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=warpSize);

int n = __shfl_xor_sync(0xffffffff, value, 3, 16);
```



2. __shfl_*()在llvm中的nvvm IR intrinsic定义
NVVM IR是基于LLVM IR的编译器IR(internal representation)。NVVM IR用于表示GPU计算kernel（例如CUDA kernel）。高级语言前端，如CUDA C编译器前端或Clang，都可以生成NVVM IR。相应的，CUDA C中的__shfl_*() intrinsic函数族也被编译器前端翻译为nvvm IR intrinsic函数族：

declare {i32, i1} @llvm.nvvm.shfl.sync.i32(i32 %membermask, i32 %mode, i32 %a, i32 %b, i32 %c)

llvm.nvvm.shfl.sync.i32是和__shfl_*()对应的nvvm IR intrinsic函数族。其中，%membermask对应__shfl_sync的参数mask。当前正在执行的warp中的每个线程会基于输入参数%b、%c和%mode计算源通道索引j。如果计算出的源通道索引j在范围内，则llvm.nvvm.shfl.sync.i32返回的i32值将是通道j的%a值（即var）；否则，返回当前线程的%a值。如果对应通道j的线程处于非活跃状态，则返回的i32值是不确定的。如果源通道j在范围内，则llvm.nvvm.shfl.sync.i32返回的i1值为1，否则为0。
参数%mode必须为常量，不同常量值对应不同shuffle方式：0是IDX，1是UP，2是DOWN，3是BFLY。参数%b根据%mode的值不同指定源通道或源通道偏移。如果%mode为0，对应__shfl_sync()，%b指定源通道；如果%mode为1或2，对应__shfl_up_sync()或__shfl_down_sync()，%b指定源通道偏移（即delta）；如果%mode为3，对应__shfl_xor_sync()，%b指定laneMask。

参数%c包含拼凑在一起的两个值。其中一个值是掩码（mask），该值将warp逻辑上分为子段；另一个是钳位值（clamp），用于限制源通道索引的上限。
以下伪代码说明了llvm.nvvm.shfl.sync.i32的语义，从中可以理解上述各个参数的作用和意义：

```
%lane[4:0] = current_lane_id; // position of thread in warp
%bval[4:0] = %b[4:0]; // source lane or lane offset (0..31)
%cval[4:0] = %c[4:0]; // clamp value
%mask[4:0] = %c[12:8];

%maxLane = (%lane[4:0] & %mask[4:0]) | (%cval[4:0] & ~%mask[4:0]);
%minLane = (%lane[4:0] & %mask[4:0]);
switch (%mode) {
  case UP: %j = %lane - %bval; %pval = (%j >= %maxLane); break;
  case DOWN: %j = %lane + %bval; %pval = (%j <= %maxLane); break;
  case BFLY: %j = %lane ^ %bval; %pval = (%j <= %maxLane); break;
  case IDX: %j = %minLane | (%bval[4:0] & ~%mask[4:0]); %pval = (%j <= %maxLane); break;
}
if (!%pval) %j = %lane; // copy from own lane
if (thread at lane %j is active)
   %d = %a from lane %j
else
   %d = undef
return {%d, %pval}
```

3. DS-Permute指令

上述CUDA shuffle intrinsic需要翻译为底层机器指令才能在GPGPU上完成预期功能。AMD Vega开源ISA中包含两条DS-Permute指令：ds_permute_b32和ds_bpermute_b32。下一节会介绍将CUDA shuffle intrinsic翻译为底层（非Nvidia GP）permute指令的方法。这两条指令使用LDS硬件在wavefront的64条通道（lane）之间交换数据，提供了一种不同的方式来表达通道寻址，但并不实际写入LDS位置。ds_permute_b32指令实现前向permute（forward permute），即，将数据放入通道i；而ds_bpermute_b32（在permute之前注意字母“ b”）实现反向permute（backward permute），即，从通道i读取数据。其用法如下：

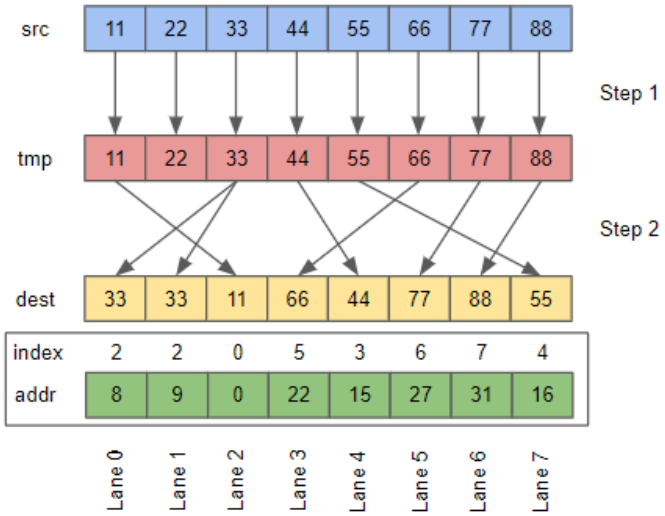
```
ds_permute_b32 dest, addr, src [offset:addr_offset] // push to dest
ds_bpermute_b32 dest, addr, src [offset:addr_offset] // pull from src

// Examples:
ds_permute_b32 v0, v1, v2
ds_bpermute_b32 v0, v0, v1 offset:0x10
```

其中dest、addr和src是VGPR，addr_offset是可选的立即数偏移量。两条指令均从src中获取数据，并根据提供的地址（addr + addr_offset）对其进行shuffle，并将结果保存到dest寄存器中。整个过程分为两步：第一，所有活跃通道将数据写入临时缓冲区；第二，所有活跃通道从临时缓冲区读取数据，未初始化的位置视为零值。

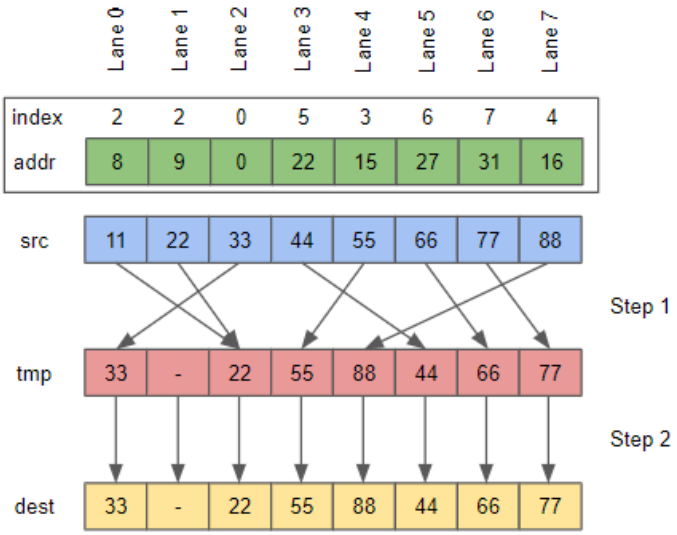
permute指令在通道之间移动数据，但仍然像其他LDS指令一样使用字节寻址的概念。由于VGPR值的宽度为4个字节，因此addr VGPR中的值应为required_lane_id * 4。
指令在访问临时缓冲区之前将addr_offset立即数添加到addr值 该立即数可用于旋转（***rotate***）src值。请注意，指令需要一个字节地址，但是只能移动完全对齐的双字。换句话说，仅使用最终地址的位[7:2]。
在许多情况下，permute地址是基于work item ID或lane ID。在kernel执行之前，将work item ID加载到v0（对于多维组，可能会加载到v1和v2）。以下代码将lane ID写入VGPR v6：

```
v_mbcnt_lo_u32_b32 v6, -1, 0
v_mbcnt_hi_u32_b32 v6, -1, v6
```

以下是简化的8通道（lane）wavefront的ds_bpermute_b32（反向permute）示例。第一步，所有通道将数据从src写入tmp中的相应位置。第二步，基于addr中的地址从tmp缓冲区读取数据放入dest。图中index的值是取addr[7:2]的结果，表示第二步中tmp元素的实际索引。例如，通道0和1中的addr值分别为8(0b00001000)和9(0b00001001)，取[7:2]的结果都为2(0b000010)。可见，虽然通道0和1中的addr值不同，但是，index都指向同一tmp元素。dest中的数据从index标识的通道中读取。例如，dest[0]和dest[1]对应的index都是2，因此，dest[0]和dest[1]的值从通道2读取，读取的值为33。dest[2]对应的index是0。因此，dest[2]的值从通道 0读取，读取的值为11，依此类推。整个过程可理解为dest从lane[index]读数据，即，index控制dest从哪个通道读数据。



在前向permute中，第一步，所有通道均根据addr寄存器中的index将src数据写入tmp缓冲区对应位置。例如，src[0]和src[1]对应的index都是2。两个通道可以写入同一tmp元素，这是ds_bpermute_b32不会出现的情况。前向permute解决这种冲突的方法与写入同一LDS地址的方法相同，即，ID较大的通道获胜。因此，在这个例子中，tmp[2]最后保留的值为10。在第二步中，dest直接读取tmp对应位置的数据。整个过程可以理解为src向lane[index]写数据，即，index控制src向哪个通道写数据。



与CUDA的__shfl_*() intrinsic函数族类似，AMD的HCC编译器为ds_permute和ds_bpermute提供intrinsic支持。这些是device function（标记为[[hc]]），因此可以从kernel（在hcc上运行）调用：

```
extern "C" int __amdgcn_ds_bpermute(int index, int src) [[hc]];
extern "C" int __amdgcn_ds_permute(int index, int src) [[hc]];
```

3.将nvvm IR intrinsic映射为自定义intrinsic或AMDGPU intrinsic

nvvm IR intrinsic针对nvdia GPU独家设计。虽然llvm中公开了nvvm IR intrinsic定义形式，但机器指令实现细节不可能公开。为了将__shfl_*()的nvvm IR intrinsic翻译为机器指令，首先要了解nvvm IR intrinsic的定义形式。__shfl_*()的nvvm IR intrinsic定义在llvm\include\llvm\IR\intrinsicNVVM.td文件中，此处仅以__shfl_sync()为例说明。

```
def int_nvvm_shfl_sync_idx_i32 :
  Intrinsic<[llvm_i32_ty], [llvm_i32_ty, llvm_i32_ty, llvm_i32_ty, llvm_i32_ty],
        [IntrInaccessibleMemOnly, IntrConvergent], "llvm.nvvm.shfl.sync.idx.i32">,
  GCCBuiltin<"__nvvm_shfl_sync_idx_i32">;
```

为了在非Nvdia GPU（AMD GPU或专有GPU）上兼容nvvm IR intrinsic功能，需要在AMDGPU后端或专有GPU后端，将nvvm IR intrinsic翻译为AMDGPU intrinsic或自定义intrinsic。本节以自定义intrinsic为例介绍相关实现流程，AMDGPU intrinsic实现流程与此类似。

A. 定义自定义intrinsic。

b/llvm/include/llvm/IR/Intrinsics<target>.td

```
def int_<target>_shfl_idx_i32 :
  Intrinsic<[llvm_anyint_ty], [llvm_i32_ty, llvm_i32_ty, llvm_i32_ty],
        [IntrNoMem, IntrConvergent, IntrSpeculatable], "">,
  GCCBuiltin<"__<target>_shfl_idx_i32">;
```

B. 定义instrinsic对应的SDNode。

```
def <target>shflidx_i32 : SDNode<"<target>ISD::SHFLIDX_I32",
            SDTypeProfile<1, 3, [SDTCisSameAs<0, 1>, SDTCisSameAs<0, 2>,
```

```
                SDTCisInt<0>, SDTCisInt<3>]>, []>;

b/llvm/lib/Target/<target>/<target>ISelLowering.cpp
  case Intrinsic::<target>_shfl_idx_i32:
    return DAG.getNode(<target>ISD::SHFLIDX_I32, DL, VT, Op.getOperand(1),
                       Op.getOperand(2), Op.getOperand(3));
```

C. 将nvvm intrinsic换成自定义intrinsic或AMDGPU intrinsic。

```
bool <target>LowerIntrinsics::lowerNVVMIntrinsics(CallInst *CI) {
……..
  case Intrinsic::nvvm_shfl_idx_i32:
      IRBuilder<> Builder(CI->getParent(), CI->getIterator());
      SmallVector<Value *, 8> Args(CS.arg_begin(), CS.arg_end());
      CallInst *NewCI = = Builder.CreateIntrinsic(Intrinsic::<target>_shfl_idx_i32, Args);
      NewCI->setName(CI->getName());
      if (!CI->use_empty())
        CI->replaceAllUsesWith(NewCI);
      CI->eraseFromParent();
    return true;
```

D. 定义intrinsic到Machine code之间的match patten。

```
b/llvm/lib/Target/<target>/<target>InstructionsDSlike.td
class DSlikep_SHFL_RET<string opName, RegisterClass rc = VGPR_32,
                       string NoRetOp = "">
    : DSlikep<opName,
              (outs rc:$vdst),
              (ins  rc:$src0, rc:$ml_offidx, rc:$src1),
              "$vdst, $src0, $ml_offidx, $src1",
              [(set i32:$vdst, (<target>shflidx_i32 i32:$src0, i32:$ml_offidx, i32:$src1))]> {
  let isRTN          = 1;
  let has_vdst       = 1;
  let has_src0       = 1;
  let has_src1       = 0;
  let mayLoad        = 0;
  let mayStore       = 0;
  let isConvergent   = 1;
  let PseudoInstr    = opName;
}

def DSlike_SHFL_B32       : DSlikep_SHFL_RET<"DSlike_shfl_b32",  VGPR_32, "">;
defm DSlike_SHFL_B32      : <target>_DSlikee_PERMUTE_MC<FUNC_DSlike_PERMUTE_B32,
"DSlike_SHFL_B32">;
```

参考文献:

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/#warp-shuffle-functions