



OpenGPU

乘影

“乘影” GPGPU 指令集架构  
文档手册

v2.02 2024.1.15

## 目录

1. 简介 .....	2
1.1. 术语表 .....	2
2. 编程模型和驱动程序功能 .....	3
2.1. 任务执行模型 .....	3
2.2. 驱动提供的功能 .....	3
2.3. Kernel 启动时的行为 .....	4
2.4. 栈空间说明 .....	5
2.5. 参数传递 ABI .....	5
3. 寄存器 .....	7
3.1. 寄存器设置 .....	7
3.2. 自定义 CSR .....	7
4. 指令集架构 .....	9
4.1. 指令集范围 .....	9
4.2. 自定义指令 .....	9
4.2.1. 分支控制指令 .....	9
4.2.2. 寄存器扩展指令 .....	10
4.2.3. 寄存器对拼接指令 .....	11
4.2.4. 同步和任务控制指令 .....	11
4.2.5. 自定义计算指令 .....	12
4.2.6. 自定义立即数访存指令 .....	12
4.2.7. 自定义私有内存访存指令 .....	13
4.3. 总结 .....	13

## 1. 简介

本文档从指令集架构及软硬件接口角度描述了乘影 GPGPU 的设计内容。

乘影 GPGPU 指令集以 RISC-V 向量扩展（后文简称为 RVV）为核心设计 GPGPU，相比 RISC-V 标量指令，具有更丰富的表达含义，可以实现访存特性表征、区分 workgroup 和 thread 操作等功能。核心思想是在编译器层面以 v 指令作为 thread 的行为描述，并将 thread->warp/workgroup 的公共数据合并为标量指令。硬件上一个 warp 就是一个 RVV 程序，通常向量元素长度为 num\_thread，同时又将 workgroup 中统一执行的公共地址计算、跳转等作为标量指令执行，即 Vector-Thread 架构。硬件将 warp 分时映射到 RVV 处理器的 lane 上去执行。

相比其它 SIMT 架构，在硬件上的折中是无法实现完全的 per thread per pc，仍然需要以 workgroup（或分支状态下的 warp\_split）执行。RVV 指令集在变长上有三个方面的体现：硬件 vlen 改变；SEW 元素宽度改变；LMUL 分组改变。本架构特点在于这三个参数在编译期都已固定，元素数目大部分情况也固定为 num\_thread。

### 1.1. 术语表

- SM: Streaming Multiprocessor, 流多处理器单元。
- sGPR: Scalar General Purpose Register, 标量寄存器
- vGPR: Vector General Purpose Register, 向量寄存器

GPGPU 中常用概念及释义如下表。

CUDA	OpenCL	释义
globalmem	globalmem	全局内存，用 __global 描述，可以被 kernel 的所有线程访问到
constantmem	constantmem	常量内存，用 __constant 描述，是全局地址空间的一部分
localmem	privatemem	私有内存，各 thread 自己的变量，和内核参数，是全局内存的一部分
sharedmem	localmem	局部内存，用 __local 描述，供同一 work-group 间的线程进行数据交换
grid	NDRange	一个 kernel 由多个 NDRange 组成，一个 NDRange 由多个 workgroup 组成
block/CTA	workgroup	工作组，在 SM 上执行的基本单位
warp	wavefront	32 个 thread 组成一个 warp，仅对硬件可见
thread	work-item	线程/工作项，是 OpenCL C 编程时描述的最小单位。



## 2. 编程模型和驱动程序功能

这部分从乘影作为 device 和 OpenCL 编程框架的交互来介绍 OpenCL 程序在执行前后的行为。乘影的编程模型兼容 OpenCL，即乘影的硬件层次与 OpenCL 的执行模型具有一一对应的关系。此外，OpenCL 编程框架需要提供一系列运行时实现，完成包括任务分配，设备内存空间管理，命令队列管理等功能。

### 2.1. 任务执行模型

在 OpenCL 中，计算平台分为主机端 (host) 和设备端 (device)，其中 device 端执行内核 (kernel)，host 端负责交互，资源分配和设备管理等。Device 上的计算资源可以划分为计算单元 (computing units, CU)，CU 可以进一步划分为处理单元 (processing elements, PE)，设备上的计算都在 PE 中完成。

乘影的一个 SM 对应 OpenCL 的一个计算单元 CU，每个向量车道 (lane) 对应 OpenCL 的一个处理单元 PE。

Kernel 在设备上执行时，会存在多个实例（可以理解为多线程程序中的每个线程），每个实例称为一个工作项 (work-item)，work-item 会组织为工作组 (work-group)，work-group 中的 work-item 可以并行执行，要说明的是，OpenCL 只保证了单个 work-group 中的 work-item 可以并发执行，并没有保证 work-group 之间可以并发执行，尽管实际情况中通常 work-group 是并行的。

在乘影的实现中，一个 work-item 对应一个线程，在执行时将会占用一个向量车道，由于硬件线程会被组织成线程组 (warp) 锁步执行，而 warp 中的线程数量是固定的，因此一个 work-group 在映射到硬件时可能对应多个 warp，但这些组成一个 work-group 的 warp 将会保证在同一个 SM 上执行。

### 2.2. 驱动提供的功能

乘影的驱动分为两层，一层是 OpenCL 的运行环境，一层是硬件驱动，运行环境基于 OpenCL 的一个开源实现 pocl 实现，主要管理命令队列 (command queue)，创建和管理 buffer，管理 OpenCL 事件 (events) 和同步。硬件驱动是对物理设备的一层封装，主要是为运行环境提供一些底层接口，并将软件数据结构转换为硬件的端口信号，这些底层接口包括分配、释放、读写设备内存，在 host 和 device 端进行 buffer 的搬移，控制 device 开始执行等。运行环境利用硬件驱动的接口实现和物理设备的交互。

每个 kernel 都有一些需要硬件获取的信息，这些信息由运行时创建，以 buffer 的形式拷贝到 device 端的内存空间。目前运行环境创建的 buffer 包括：

- NDRange 的 metadata buffer 和 kernel 机器码

- kernel 的 argument data buffer，即每个 kernel 参数的具体输入
- kernel\_arg\_buffer, 保存的内容为 device 端的指针，其值为 kernel 的 argument data buffer 在 device 端的地址。
- 为 private mem、print buffer 分配的空间

其中，metadata buffer 保存 kernel 的一些属性，具体内容为：

```

1. cl_int clEnqueueNDRangeKernel(cl_command_queue command_queue,
2.                                cl_kernel kernel,          //kernel_entry_ptr & kernel_arg_ptr
3.                                cl_uint work_dim,          //work_dim
4.                                const size_t *global_work_offset, //global_work_offset_x/y/z
5.                                const size_t *global_work_size, //global_work_size_x/y/z
6.                                const size_t *local_work_size, //local_work_size_x/y/z
7.                                cl_uint num_events_in_wait_list,
8.                                const cl_event *event_wait_list,
9.                                cl_event *event)
10. /*
11. #define KNL_ENTRY 0
12. #define KNL_ARG_BASE 4
13. #define KNL_WORK_DIM 8
14. #define KNL_GL_SIZE_X 12
15. #define KNL_GL_SIZE_Y 16
16. #define KNL_GL_SIZE_Z 20
17. #define KNL_LC_SIZE_X 24
18. #define KNL_LC_SIZE_Y 28
19. #define KNL_LC_SIZE_Z 32
20. #define KNL_GL_OFFSET_X 36
21. #define KNL_GL_OFFSET_Y 40
22. #define KNL_GL_OFFSET_Z 44
23. #define KNL_PRINT_ADDR 48
24. #define KNL_PRINT_SIZE 52
25. */

```

## 2.3. Kernel 启动时的行为

任务启动时，硬件驱动需要传递给物理 device 一些信号，这些信号有：

PTBR	page table base addr
CSR_KNL	metadata buffer base addr
CSR_WGID	当前 workgroup 在 SM 中的 id，仅供硬件辨识
CSR_WID	warp id，当前 warp 属于 workgroup 中的位置
LDS_SIZE	localmem_size，编译器提供 workgroup 需要占用的 localmem 空间。privatemem_size 默认按照每个线程 1kB 来分配
VGPR_SIZE	vGPR_usage，编译器提供 workgroup 实际使用的 vGPR 数目（对齐 4）

SGPR_SIZE	sGPR_usage, 编译器提供 workgroup 实际使用的 sGPR 数目（对齐 4）
CSR_GIDX/Y/Z	workgroup idx in NDRange
host_wf_size	一个 warp 中 thread 数目
host_num_wf	一个 workgroup 中 warp 数目

kernel 的参数由前述的 kernel\_arg\_buffer 传递，该 buffer 中会按顺序准备好 kernel 的 argument，包括具体参数值或其它 buffer 的地址。在 NDRange 的 metadata 中仅提供 kernel\_arg\_buffer 的地址 knl\_arg\_base。kernel 函数执行前会先执行 start.S：

```

1. # start.S
2. start:
3. csrr sp, CSR_LDS # set localmemory pointer
4. addi tp, x0, 0 # set privatememory pointer
5.
6. # clear BSS segment
7. #
8. # clear BSS complete
9.
10. csrr t0, CSR_KNL
11. lw t1, KNL_ENTRY(t0)
12. lw a0, KNL_ARG_BASE(t0)
13. jalr t1
14. # end.S
15. end:
16. endprg

```

约定 kernel 的打印信息通过 print buffer 向 host 传递。print buffer 的地址和大小在 metadata\_buffer 中提供，运行中的 thread 完成打印后，将所属 warp 的 CSR\_PRINT 置位。host 轮询到有未处理信息时，将 print buffer 从设备侧取出处理，并将 CSR\_PRINT 复位。

## 2.4. 栈空间说明

由于 OpenCL 不允许在 Kernel 中使用 malloc 等动态内存函数，也不存在堆，因此可以让栈空间向上增长。tp 用于各 thread 私有寄存器不足时压栈（即 vGPR spill stack slots），sp 用于公共数据压栈，（即 sGPR spill stack slots，实际上 sGPR spill stack slots 将作为 localmem 的一部分），在编程中显式声明了\_\_local 标签的数据也会存在 localmem 中。编译器提供 localmem 的数据整体使用量（按照 sGPR spill 1kB，结合 local 数据的大小，共同作为 localmem\_size），供硬件完成 workgroup 的分配。

## 2.5. 参数传递 ABI

对于 kernel 函数，a0 是参数列表的基址指针，第一个 clSetKernelArg 设置的显存起始地址存入 a0

register, kernel 默认从该位置开始加载参数。对于非 kernel 函数, 使用 v0-v31 和 stack pointer 传递参数, v0-v31 作为返回值。





### 3. 寄存器

#### 3.1. 寄存器设置

架构寄存器数目为 sGPR（标量）64 个，vGPR（向量）256 个，元素宽度均为 32 位。

当需要 64 位数据时，数据以偶对齐的寄存器对（Register Pair）的形式进行存储，数据低 32 位存储在 GPR[n] 中，高 32 位存储在 GPR[n+1] 中。

目前硬件中物理寄存器数目为 sGPR（标量）256 个，vGPR（向量）1024 个，GPU 硬件负责实现架构寄存器到硬件寄存器的映射。

编译器提供 vGPR、sGPR 的实际使用数目（4 的倍数），硬件会根据实际使用情况分配更多的 workgroup 同时调度。

从 RISC-V Vector 的视角来看寄存器堆，vGPR 共有 256 个，宽度 vlen 固定为线程数目 num\_thread 乘以 32 位，即相当于通过 vsetvli 指令设置 SEW=32bit，ma，ta，LMUL 为 1。而从 SIMD 的视角来看寄存器堆，每个 thread 至多拥有 256 个宽度 32 位的 vGPR。一个简单的理解是，将向量寄存器堆视作一个 256 行、num\_thread 列的二维数组，数组的每行是一个 vGPR，而每列是一个 thread 最多可用的寄存器。OpenCL 中定义了一些向量类型，这些向量类型需要使用分组寄存器的形式表达，即 float16 在寄存器堆中以列存储，占用 16 个 vGPR 的各 32 位。这部分工作由编译器进行展开。

workgroup 拥有 64 个 sGPR，整个 workgroup 只需做一次的操作，如 kernel 中的地址计算，会使用 sGPR；如果有发生分支的情况，则使用 vGPR，例如非 kernel 函数的参数传递。

一些特殊寄存器：

- x0: 0 寄存器；
- x1/ra: 返回 PC 寄存器；
- x2/sp: 栈指针 / local mem 基址；
- x4/tp: private mem 基址。

参数传递：

对于 kernel 函数，a0 是参数列表的基址指针，第一个 clSetKernelArg 设置的显存起始地址存入 a0 register，kernel

默认从该位置开始加载参数。

#### 3.2. 自定义 CSR

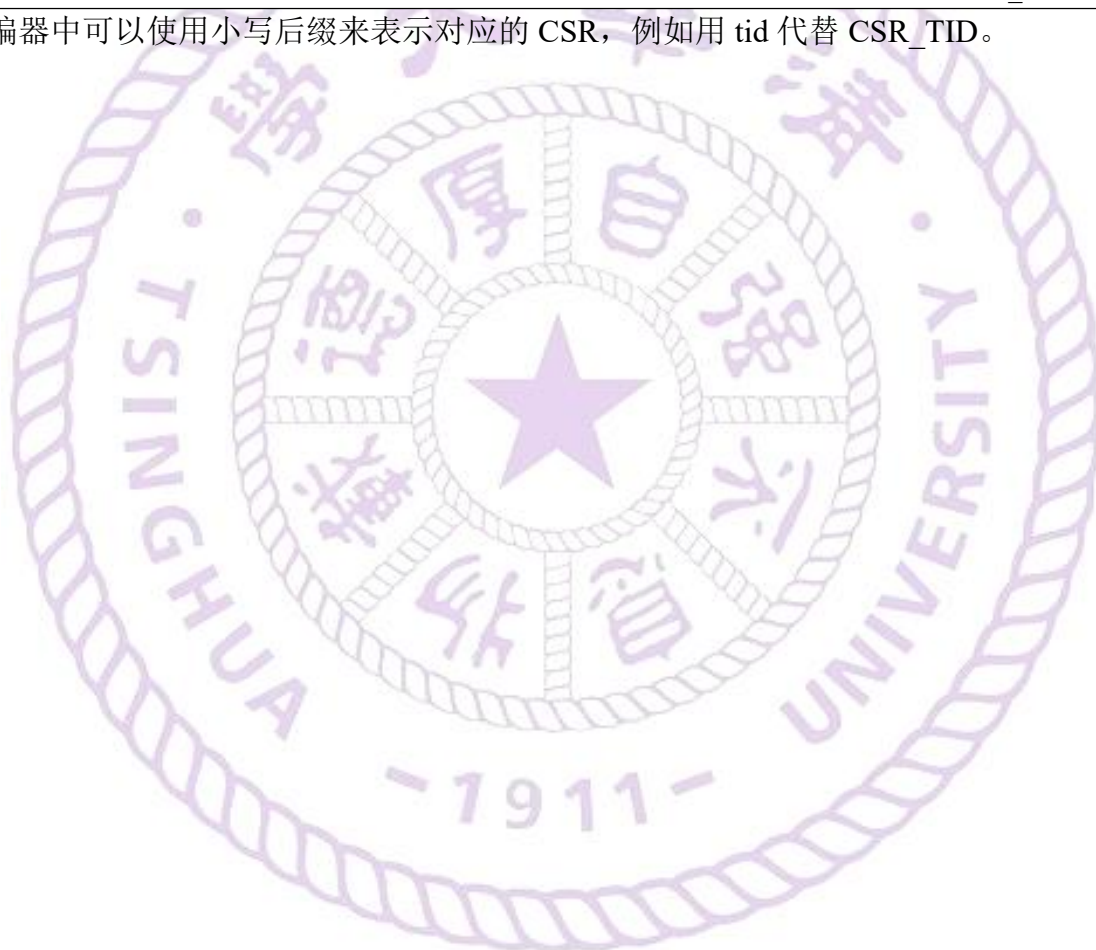
描述	名称	地址
该 warp 中 id 最小的 thread id，其值为 CSR_WID*CSR_NUMT，配合 vid.v 可计算其它	CSR_TID	0x800



## thread id

该 workgroup 中的 warp 总数	CSR_NUMW	0x801
一个 warp 中的 thread 总数	CSR_NUMT	0x802
该 workgroup 的 metadata buffer 的 baseaddr	CSR_KNL	0x803
该 SM 中本 warp 对应的 workgroup id	CSR_WGID	0x804
该 workgroup 中本 warp 对应的 warp id	CSR_WID	0x805
该 workgroup 分配的 local memory 的 baseaddr，同时也是该 warp 的 xgpr spill stack 基址	CSR_LDS	0x806
该 workgroup 分配的 private memory 的 baseaddr，同时是该 thread 的 vgpr spill stack 基址	CSR_PDS	0x807
该 workgroup 在 NDRange 中的 x id	CSR_GDX	0x808
该 workgroup 在 NDRange 中的 y id	CSR_GDY	0x809
该 workgroup 在 NDRange 中的 z id	CSR_GDZ	0x80a
向 print buffer 打印时用于与 host 交互的 CSR	CSR_PRINT	0x80b
重汇聚 pc 寄存器	CSR_RPC	0x80c

注：在汇编器中可以使用小写后缀来表示对应的 CSR，例如用 tid 代替 CSR\_TID。



## 4. 指令集架构

### 4.1. 指令集范围

选用 RV32V 扩展作为基本指令集，支持指令集范围为：RV32I, M, A, zfinx, zve32f.

V 扩展指令集主要支持独立数据通路的指令，不支持 RVV 原有的 shuffle, widen, narrow, gather, reduction 指令。

下表列出目前支持的标准指令范围，有变化指令已声明。

指令集	乘影支持状况	指令变化
RV32I	不支持 ecall, ebreak, 支持 SV39 虚拟地址	
RV32M F	支持 RV32M zfinx zve32f	
RV32A	支持	
RV32V-Register State	仅支持 LMUL=1 和 2	
RV32V-ConfigureSetting	支持计算 vl, 可通过该选项配置支持不同宽度元素	
RV32V-LoadAndStores	支持 vle32.v vlse32.v vluxei32.v 访存模式	vle8 等指令语义改为“各 thread 向向量寄存器元素位置写入”，而非连续写入
RV32V-IntergerArithmetiv	支持绝大多数 int32 计算指令	vmv.x.s 语义改为“各 thread 均向标量寄存器写入”，而非总由向量寄存器 idx_0 写入，多线程同时写入是未定义行为，正确性由程序员保证；vmv.s.x 语义改为与 vmv.v.x 一致
RV32V-FixedPointArithmetic	添加 int8 支持，视应用需求再添加其它类型	
RV32V-FloatingPointArithmetic	支持绝大多数 fp32 指令，添加 fp64 fp16 支持	
RV32V-ReductionOperations	视应用和编译器需求再考虑添加，例如需要支持 OpenCL2.0 中的 work_group_reduce 时	
RV32V-Mask	支持各 lane 独立计算和设置 mask 的指令	vmsle 等指令语义改为“各 thread 向向量寄存器元素位置写入”，而非连续写入
RV32V-Permutation	不支持，视应用和编译器需求再考虑添加	
RV32V-ExceptionHandling	不支持，视应用和编译器需求再考虑添加	

### 4.2. 自定义指令

#### 4.2.1. 分支控制指令

VBRANCH 分支指令使用 B 型指令格式。12 位 B 型有符号立即数进行符号位扩展后加上当前的 PC 值给出 else 路径的起始位置 PC, 读取 CSR\_RPC 的值 rpc, 根据向量寄存器计算比较结果操作 SIMT 线程分支管理栈。

31	25	24	20	19	15	14	12	11	7	6	0
imm[12]	imm[10:5]	vs2	vs1	funct3	imm[4:1]	imm[11]	opcode				
off[12 10:5]		src2	src1	VBEQ/VBNE	off[4:1 11]		VBRANCH				
off[12 10:5]		src2	src1	VBLT[U]	off[4:1 11]		VBRANCH				
off[12 10:5]		src2	src1	VBGE[U]	off[4:1 11]		VBRANCH				

线程分支指令将比较两个向量寄存器,如果线程对应的元素相等,VBEQ 的这些线程将跳转到 else PC 处执行,其余元素不等的线程将继续执行 PC+4,分支两条路径的跳转、聚合和掩码控制由 SIMT 栈控制,将 rpc, else PC 和元素比较结果压入栈中,当全部活跃线程对应元素均相等或不等时,不触发栈操作,全部跳转到 else PC 或继续执行 PC+4。当 vs1 和 vs2 中的操作数不等时,VBNE 将判断对应线程跳转执行 PC else 还是 PC+4。VBLT 和 VBLTU 将分别使用有符号数和无符号数比较 vs1 和 vs2,当对应的向量元素有 vs1 小于 vs2 时,对应线程跳转至 PC else,剩余活跃线程继续执行 PC+4。VBGE 和 VBGEU 将分别使用有符号数和无符号数比较 vs1 和 vs2,当对应的向量元素有 vs1 大于等于 vs2 时,对应线程跳转至 PC else,剩余活跃线程继续执行 PC+4。

线程分支汇聚 join 指令采用 S 型指令格式,默认源操作数、立即数均为 0。

31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]	vs2	vs1	funct3	imm[4:0]	opcode						
0000000	00000	00000	JOIN	00000	JOIN						

JOIN 指令将对比当前指令 PC 与 SIMT 栈顶重汇聚 PC 是否相等,若相等,则设置活跃线程掩码为当前栈顶掩码项,跳转至栈顶 else PC 处执行指令,若不等,则不做任何操作。

重汇聚 PC 设置指令 SETRPC 将 12 位立即数进行符号位扩展并与源操作数相加后,将结果写入 CSR\_RPC 的 csr 寄存器和目的寄存器。

31	20	19	15	14	12	11	7	6	0	
imm[11:0]			rs1		funct3		rd		opcode	
imm[11:0]			rs1		SETRPC		rd		SETRPC	

SETRPC 指令将设置对应线程束此后最近一次 VBRANCH 指令的重汇聚 PC 值,可以理解为,SETRPC 和 VBRANCH 系列指令共同完成了一次完成的线程分支操作。

#### 4.2.2. 寄存器扩展指令

REGEXT 和 REGEXTI 指令用于扩展它之后一条指令的寄存器编码及立即数编码。在 REGEXT 指令中,12 位的立即数被分拆为四段 3 位数,分别拼接在下一条指令中寄存器 rs3/vs3、rs2/vs2、rs1/vs1、rd/vd 编码的高位上。REGEXTI 指令则面向使用了 5 位立即数的指令,前缀包含 6 位的立即数高位,



以及 rs2/vs2、rd/vd 编码的高位。11 立即数由前缀指令中的立即数高位与 5 位立即数直接拼接得到。

此外，在不使用扩展的情况下，向量浮点运算的 vs3 就是 vd 的[11:7]段，但进行向量扩展时，vs3 和 vd 的高 3 位分离存储。

31	20	19	15	14	12	11	7	6	0	
imm[11:0]			rs1		funct3		rd		opcode	
x3, x2, x1, xd			00000		REGEXT		00000		REGEXT	
imm[10:5], x2, xd			00000		REGEXTI		00000		REGEXT	

### 4.2.3. 寄存器对拼接指令

REGPAIR 和 REGPAIRI 指令将用于指示其之后指令将使用相邻寄存器对拼接的方式实现 64 位访存操作，同时，寄存器对拼接指令也兼有寄存器扩展功能。

31	20	19	15	14	12	11	7	6	0	
imm[11:0]			rs1		funct3		rd		opcode	
x3, x2, x1, xd			00000		REGPAIR		00000		REGPAIR	
imm[10:5], x2, xd			00000		REGPAIRI		00000		REGPAIR	

REGPAIR 指令的 12 位立即数被分拆为四段 3 位数，分别拼接在下一条指令中寄存器 rs3/vs3、rs2/vs2、rs1/vs1、rd/vd 编码的高位上。REGPAIRI 指令则面向使用了 5 位立即数的指令，前缀包含 6 位的立即数高位，以及 rs2/vs2、rd/vd 编码的高位。11 立即数由前缀指令中的立即数高位与 5 位立即数直接拼接得到。REGPAIR/REGPAIRI 指令之后一条指令将指明源操作数、目的操作数低 32 位寄存器编码，寄存器对采用偶对齐，也即当后一条指令指明的源寄存器、目的寄存器编码为偶数时，意味着指明的寄存器中的 32 位数据作为低位、编码+1 的相邻寄存器中的数据作为高 32 位，拼接成为 64 位数据；当后一条指令指明的寄存器编码为奇数时，表明对应寄存器不扩展，仍作为 32 位数据计算。

### 4.2.4. 同步和任务控制指令

ENDPRG 指令需要显式插入到 Kernel 末尾，以指示当前 warp 执行结束。只能在无分支的条件下使用。

31	25	24	20	19	15	14	12	11	7	6	0
0000000			rs2		rs1		funct3		rd		opcode
0000000			00000		00000		ENDPRG		00000		ENDPRG

BARRIER 对应 OpenCL 的 `barrier(cl_mem_fence_flags flags)` 和 `work_group_barrier(cl_mem_fence_flags flags, [memory_scope scope])` 函数，实现同一 workgroup 内的 thread 间数据同步。memory\_scope 缺省值为 `memory_scope_work_group`。



31	25	24	20	19	15	14	12	11	7	6	0
funct7			rs2		imm[4:0]		funct3		rd		opcode
0000010			00000		imm[4:0]		BARRIER		00000		BARRIER
0000011			00000		imm[4:0]		BARRIERSUB		00000		BARRIER

五位立即数字段编码如下：

<b>imm[4:3]</b>	<b>00</b>	<b>01</b>	<b>10</b>	<b>11</b>
<b>memory_scope</b>	work_group (default)	work_item	device	all_svm_devices
<b>imm[2:0]</b>	<b>imm[2]=1</b>	<b>imm[1]=1</b>	<b>imm[0]=1</b>	<b>imm[2:0]=000</b>
<b>CLK_X_MEM_FENCE</b>	IMAGE	GLOBAL	LOCAL	USE_CNT

开启 `opencl_c_subgroups` 特性后，则改为 `barriersub` 指令，对应 `memory_scope=subgroup` 的情况，此时 `imm[4:3]` 固定为 0，`cl_mem_fence_flags` 为 `imm[2:0]`，与 `barrier` 指令一致。

#### 4.2.5. 自定义计算指令

31	20	19	15	14	12	11	7	6	0
imm[11:0]			vs1		funct3		vd		opcode
imm[11:0]			vs1		VADDVI		vd		VADDVI

VADD12.VI 将无符号数 `imm` 加到向量寄存器 `vs1` 赋予向量寄存器 `vd`。

31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]		vs2		vs1		funct3		vd		opcode	
000001	m	vs2		00000		VFEXP		vd		VFEXP	
000011	m	vs2		vs1		VFTTA.VV		vd		VFTTAVV	

VFEXP 计算 `exp(v2)` 赋予向量寄存器 `vd`。VFTTAVV 计算向量 `vs1` 和 `vs2` 的卷积，加上向量寄存器 `vd` 后赋予 `vd`。

#### 4.2.6. 自定义立即数访存指令

31	20	19	15	14	12	11	7	6	0
imm[11:0]			vs1		funct3		vd		opcode
offset[11:0]			base		width		dest		VLOAD12
offset[11:0]			base		010		dest		VLW12
offset[11:0]			base		001		dest		VLH12
offset[11:0]			base		000		dest		VLB12
offset[11:0]			base		101		dest		VLHU12
offset[11:0]			base		100		dest		VLBU12

与标准 RISC-V 指令一致，自定义访存指令的地址空间按字节寻址且为小端格式。有效字节地址  $Addr=vs1+offset$ ，将内存中以 `Addr` 为起始的连续向量长度个元素复制到向量寄存器 `vd`。VLW12 指令从内存中加载 32 位值的向量到 `rd` 中。VLH12 从内存中加载向量长度个 16 位值，然后将其符号扩展

到 32 位，再存储到 rd 中。VLHU12 从内存中加载向量长度个 16 位值，然后将其无符号扩展到 32 位，再存储到 rd 中。VLB12 和 VLBU12 对 8 位值有类似定义。

31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]			vs2		vs1		funct3		imm[4:0]		opcode
offset[11:5]			src		base		width		offset[4:0]		VSTORE12
offset[11:5]			src		base		110		offset[4:0]		VSW12
offset[11:5]			src		base		011		offset[4:0]		VSH12
offset[11:5]			src		base		111		offset[4:0]		VSB12

Addr=vs1+offset, VSW12、VSH12 和 VSU12 指令分别将向量寄存器 vs2 的低位中的 32 位、16 位和 8 位值的向量长度个数据连续存入以 Addr 起始的内存空间。

#### 4.2.7. 自定义私有内存访存指令

与 VLOAD12/VSTORE12 指令类似，单独定义了专用于访问私有内存的指令，地址计算有所差异。

31	30	25	24	20	19	15	14	12	11	7	6	0
0	imm[10:0]				vs1		funct3		vd		opcode	
0	offset[10:0]				base		width		dest		VLOAD	
0	offset[10:0]				base		010		dest		VLW	
0	offset[10:0]				base		001		dest		VLH	
0	offset[10:0]				base		000		dest		VLB	
0	offset[10:0]				base		101		dest		VLHU	
0	offset[10:0]				base		100		dest		VLBU	

Addr=(vs1+imm)\*num\_thread\_in\_workgroup+thread\_idx+csr\_pds, 将 Addr 地址的以 width 所指示的连续向量长度个元素存入向量寄存器 vd, 并做相应的有符号/无符号扩展。

31	30	25	24	20	19	15	14	12	11	7	6	0
1	imm[10:5]			vs2		vs1		funct3		imm[4:0]		opcode
1	offset[10:5]			src		base		width		offset[4:0]		VSTORE
1	offset[10:5]			src		base		010		offset[4:0]		VSW
1	offset[10:5]			src		base		001		offset[4:0]		VSH
1	offset[10:5]			src		base		000		offset[4:0]		VSB

Addr=(vs1+imm)\*num\_thread\_in\_workgroup+thread\_idx+csr\_pds, VSW、VSH 和 VSU 指令分别将向量寄存器 vs2 的低位中的 32 位、16 位和 8 位值的向量长度个数据连续存入以 Addr 起始的内存空间。

### 4.3. 总结

自定义扩展指令总结如下：

imm[12 10:5]	vs2	vs1	000	imm[4:1 11]	1011011	VBEQ
imm[12 10:5]	vs2	vs1	001	imm[4:1 11]	1011011	VBNE
imm[12 10:5]	vs2	vs1	100	imm[4:1 11]	1011011	VBLT
imm[12 10:5]	vs2	vs1	101	imm[4:1 11]	1011011	VBGE

imm[12 10:5]		vs2	vs1	110	imm[4:1 11]	1011011	VBLTU
imm[12 10:5]		vs2	vs1	111	imm[4:1 11]	1011011	VBGEU
0000000		00000	00000	010	00000	1011011	JOIN
imm[11:0]			rs1	011	rd	1011011	SETRPC
imm[11:0](x3,x2,x1,xd)			00000	010	00000	0001011	REGEXT
imm[11:0](imm[10:5],x2,xd)			00000	011	00000	0001011	REGEXTI
imm[11:0](x3,x2,x1,xd)			00000	101	00000	0001011	REGPAIR
imm[11:0](imm[10:5],x2,xd)			00000	111	00000	0001011	REGPAIRI
0000000		rs2	rs1	100	rd	0001011	ENDPRG
0000010		rs2	imm[4:0]	100	rd	0001011	BARRIER
0000011		rs2	imm[4:0]	100	rd	0001011	BARRIERSUB
imm[11:0]			vs1	000	vd	1011011	VADD12.VI
000010m		vs2	00000	110	vd	0001011	VFEXP
000010m		vs2	vs1	100	vd	0001011	VFTTA.VV
imm[11:0]			vs1	010	vd	1111011	VLW12.V
imm[11:0]			vs1	001	vd	1111011	VLH12.V
imm[11:0]			vs1	000	vd	1111011	VLB12.V
imm[11:0]			vs1	101	vd	1111011	VLHU12.V
imm[11:0]			vs1	100	vd	1111011	VLBU12.V
imm[11:5]		vs2	vs1	110	imm[4:0]	1111011	VSW12.V
imm[11:5]		vs2	vs1	011	imm[4:0]	1111011	VSH12.V
imm[11:5]		vs2	vs1	111	imm[4:0]	1111011	VSBI2.V
0	imm[10:0]		vs1	010	vd	0101011	VLW.V
0	imm[10:0]		vs1	001	vd	0101011	VLH.V
0	imm[10:0]		vs1	000	vd	0101011	VLB.V
0	imm[10:0]		vs1	101	vd	0101011	VLHU.V
0	imm[10:0]		vs1	100	vd	0101011	VLBU.V
1	imm[10:5]	vs2	vs1	110	imm[4:0]	0101011	VSW.V
1	imm[10:5]	vs2	vs1	011	imm[4:0]	0101011	VSH.V
1	imm[10:5]	vs2	vs1	111	imm[4:0]	0101011	VSBI.V