承影 GPGPU 架构文档手册 v1.8

编写人：杨轲翔 (yangkx20@mails.tsinghua.edu.cn)
清华大学集成电路学院 dsp-lab

承影是基于 RISC-V 向量扩展实现的开源 GPGPU，开源项目主页面见 THU-DSP-LAB/ventus-gpgpu: GPGPU processor supporting RISCV-V extension, developed with Chisel HDL (github.com)

1.8 版本更新（23.1.11）：

- 更新了访存指令，提供 private mem 专用访存指令并规定访存形式
- 整理了目前为止改变和添加的所有指令
- 更新了地址空间和栈的使用方式
- 更新了 CSR 列表
- 添加了 kernel 启动相关的驱动和 start.S 流程说明

待确定的问题：

- 不同数据类型(fp64 fp16)的支持方式
- 后续将地址空间扩展到 64 位的方式

1.7 版本更新（22.12.15）：

- 基于实际实现的需求，确定改用 RV32IMA zfinx zdinx V
- 向量和标量寄存器数目需要扩展，扩展方式是使用“宏指令”，编译器视角下为 64 个 sGPR 和 256 个 vGPR
- 分组寄存器的行为和使用方式，确定由编译器展开，以提供对原生向量类型的支持
- 在指令中显式指定 register pair、sub register 和 packed register，不再使用 RVV 的 LMUL 和 SEW 等修改

1.6 版本更新（22.11.29）：

- 取消了对 RISC-V F D 的支持，没有 zfinx zdinx 的需求了。
- 更新了对 SIMT-stack 和 vbeq/join 指令的补充说明。
- 建议硬件支持 context switch。其余 opencl3.0 特性如 generic address space、SVM、device-side enqueue 等，编译器改动不大，等当前版本硬件完成后再考虑添加。

1.5 版本更新：

- 明确了指令集支持范围。
- 更新了地址划分，确认改用 RV64 和 MMU
- 添加了对驱动程序的功能描述。

本文档主要描述了承影 GPGPU 在软硬件视角下的功能描述，部分功能直接以承影下一版本标定，本文档中会备注说明目前未支持到的情况。
如果在 opencl 描述、驱动、编译器功能上有问题，或者硬件设计上不足之处，欢迎在 github 上提 issue，或邮件联系作者。

简介

本文档描述了承影 GPGPU 的设计内容，包括 OpenCL 编程视角和微架构视角。
承影 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完成 thread->warp 的合并，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，通过额外指令实现不同数据类型，**本架构本质上是 SIMT**，RVV 动态变长对 calling convention 并不友好，编译时难以确定函数调用，因此该特性在编译器中不再使用。

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

memory 划分和编程模型定义：（在本文档中，除了 warp 和 thread 外，其它词均采用 opencl 中的说法）

cuda	opencl	解释
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(AMD)	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。

参数表

变量名	解释

num_thread	一个 warp 里的 thread 数，默认值 32
num_warp	硬件上一个 SM 里允许的最大 warp 数（可以来自不同 workgroup）
num_block	硬件上一个 SM 里允许的最大 workgroup 数
num_lane	硬件上一个 SM 的运算单元里一次能同时处理的 thread 数
localmem_max	硬件上一个 SM 里提供的 localmem 的最大空间

因此，num_thread*num_warp 就代表了一个 block 里的最大 thread 数目。

编程模型和驱动程序功能

从 OpenCL 视角来看这个 device。

硬件上的对应关系

整个 GPU 作为一个 compute device，SM 对应 Compute Unit(CU)，SM 内部多个执行单元对应多个 PE。

任务执行模型

与 OpenCL 一致，将 workgroup 映射到 CU 上执行，各个 thread 映射到 PE 上执行，硬件上会将 thread 以 warp(相邻 32 个 thread 一组)为单位打包，呈现出 SIMD 的执行效果。 目前 NDRange 拆分为 workgroup 在驱动上进行，workgroup 拆分为 warp 在硬件上进行。

驱动提供的功能

由 opencl 驱动(pocl)来管理 command queue，创建和分配 buffer，以 workgroup 为单位分配任务，并为每个任务创建 metadata buffer。共享内存空间、任务间的顺序和事件同步机制也由 pocl 管理。
在 pocl 后端添加基于 verilator 的 ventus device 和 ISS spike ventus device，以完成物理地址分配和任务启动。
目前驱动创建的 buffer 包括：

- NDRange 的 metadata buffer 和 kernel 程序
- kernel 的 argument buffer
- kernel argument 中显式引用的 buffer
- 为 private mem、constant mem 分配的空间

任务启动时，由驱动直接传递的信号为：

- PTBR // page table base addr
- CSR_KNL // metadata buffer base addr
- CSR_WID // workgroup id，低 log(num_SM) bit 为推荐的 SM
- LDS_SIZE // localmem_size，编译器提供 workgroup 需要占用的 localmem 空间
- 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

runtime 行为

约定 kernel 启动时，NDRange 的参数通过 metadata 的 buffer 传递，该 buffer 的内容为：

```
cl_int clEnqueueNDRangeKernel(cl_command_queue command_queue,
                              cl_kernel kernel,          //kernel_entry_ptr & kernel_arg_ptr
                              cl_uint work_dim,          //work_dim
                              const size_t *global_work_offset, //global_work_offset_x/y/z
                              const size_t *global_work_size,   //global_work_size_x/y/z
                              const size_t *local_work_size,    //local_work_size_x/y/z
                              cl_uint num_events_in_wait_list,
                              const cl_event *event_wait_list,
```

```
                    cl_event *event)
/*
#define KNL_ENTRY 0
#define KNL_ARG_BASE 4
#define KNL_WORK_DIM 8
#define KNL_GL_SIZE_X 12
#define KNL_GL_SIZE_Y 16
#define KNL_GL_SIZE_Z 20
#define KNL_LC_SIZE_X 24
#define KNL_LC_SIZE_Y 28
#define KNL_LC_SIZE_Z 32
#define KNL_GL_OFFSET_X 36
#define KNL_GL_OFFSET_Y 40
#define KNL_GL_OFFSET_Z 44
*/
```

kernel 的参数由另一块 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：

```
# start.S
start:
   csrr sp, CSR_LDS # set stack point
   addi tp, x0, 0

   # clear BSS segment
   #
   # clear BSS complete

   csrr t0, CSR_KNL
   lw t1, KNL_ENTRY(t0)
   lw a0, KNL_ARG_BASE(t0)
   jalr t1
```

```
# end.S
end:
   endprg
```

多任务机制

硬件上添加 MMU 支持虚拟地址，驱动完成页表创建。硬件支持在同一 SM 上运行多个 kernel 的任务，但目前限制是必须在同一上下文（即同一套地址空间），否则由于 VIVT cache 可能导致不可预期的问题。
context 更新时会刷新 TLB 和 PTBR。

kernel 内汇编指令说明

其实是 OpenCL C programming 的部分需求：
OpenCL kernel 的编程部分，和硬件上的 ABI、指令集、寄存器接口的部分。

指令集范围

RV32 I M A zfinx zdinx V

总体来看 V 里面支持的主要是独立数据通路的指令，当前 RVV 原有的 shuffle widen narrow gather reduction 都不支持，具体指令范围见附录。

下表列举出了目前支持的标准指令的范围，有变化的指令已经声明。

	承影支持情况	指令变化
RV32I	不支持 ecall ebreak，支持 SV39 虚拟地址	
RV32M F D	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-LoadsAndStores	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-IntergerArithmetic	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
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	视应用和编译器需求再考虑添加	
RV32V-ExceptionHandling	视应用和编译器需求再考虑添加	

自定义指令

barrier

barrier x0,x0,0 # meet barrier 对应 opencl 的 barrier(cl_mem_fence)函数，实现同一 block 内的 thread 间数据同步。

endprg

endprg x0,x0,0 # meet the end of the kernel 需要显式插入到 kernel 末尾，表明当前 warp 执行结束。只能在无分支的情况下使用。

vbeq/join 系列指令

vbeq vs2, vs1, offset # set predicate vs2==vs1, and set branch address pc+4+offset
join x0, x0, offset # set join address pc+4+offset 隐式 SIMT-stack 实现分支控制。每个分支块的结尾需要用 join 指示汇合点地址。
vbeq 参照 beq 提供了 vbne vblt vbge vbltu vbgeu 版本，指令编码修改了 func3 段。

regext{i}

regext x0,x0,imm12 # (x3,x2,x1,xd)=imm12，register index extend
regexti x0,x0,imm12 # (imm[10:5],x1,xd)=imm12, imm and register index extend
用宏指令扩展可用寄存器数目，该指令表明下一条指令的寄存器编号（和立即数）会扩展。
当前版本编译器对寄存器扩展指令，支持按需要进行扩展。
当前版本编译器对于除自定义指令外的立即数指令，默认使用 11 位立即数，即认为由 regexti + vi 指令总是组成 64bit 长指令。在编译器视角下，立即数指令将跳过 reg ext 这一阶段。

> 原因是部分立即数的值是在链接时才计算得到，此阶段已经无法消除无需扩展的 regexti 前缀。考虑到立即数指令使用频率相对较低（使用长立即数的场景只有 vlw 和 vadd.vv 指令，因此针对这两条指令专门定义 imm12 版本），故为其它立即数指令默认提供 64bit 版本。

vlw.v/vsw.v privatemem 访存指令

仅用于访问 privatemem，以标量访存指令形式提供的指令，但访问向量地址、写入向量寄存器。为便于编译器和编程等使用，在 per-thread 视角下该地址为连续地址，由硬件根据 thread_id 完成偏移。 vlw.v vd, imm11(rs1) # vd <- mem[(rs1+imm11)*num_thread+thread_idx]
vsw.v vs2, imm11(rs1) # mem[(rs1+imm11)*num_thread+thread_idx] <- vs2

vlw12.v/vsw12.v 带 12 位立即数地址的向量访存指令

vlw12.v vd, imm12(vs1) # vd <- mem[vs1+imm12]
vsw12.v vs2, imm12(vs2) # mem[vs1+imm12] <- vs2 vlw12 参照 lw 提供了 vlh12 vlb12 vlhu12 vlbu12 版本，vsw12 也有 vsh12 vsb12 版本，指令编码修改了 func3 段。

vadd12.vi/vsub12.vi imm12 版本立即数指令

vadd12.vi vd, vs1, imm12 # vd <- vs1 + imm12
vsub12.vi vd, vs1, imm12 # vd <- vs1 - imm12

vftta.vv 卷积指令

vftta.vv vd, vs2, vs1 # vd <- vs2 conv vs1

vfexp.v 指数运算指令

vfexp.v vd, vs2 # vd <- exp(vs2)
vfexp 以 intrinsic 方式提供。

31 25	24 20	19 15	14 12	11 7	6 0	assemble
imm[11:0](x3,x2,x1,xd)		0	0 1 0	0	0 0 0 1 0 1 1	regext x0,x0,imm
imm[11:0](imm[10:5],x2,xd)		0	0 1 1	0	0 0 0 1 0 1 1	regexti x0,x0,imm
imm[11:0]		vs1	0 0 0	vd	0 0 0 1 0 1 1	vadd12.vi vd, vs1, imm
imm[11:0]		vs1	0 0 1	vd	0 0 0 1 0 1 1	vsub12.vi vd, vs1, imm
0 0 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 0 0 1 0 1 1	endprg x0,x0,x0
0 0 0 0 0 0 1	rs2	imm[4:0]	1 0 0	rd	0 0 0 1 0 1 1	barrier x0,x0,0
imm[11:0]		vs1		vd	0 0 0 0 1 1 1	vlw12.v vd, offset(vs1)
imm[11:5]	vs2	vs1		imm[4:0]	0 1 0 0 1 1 1	vsw12.v vs2,offset(vs1)
0 imm[10:0]		rs1		vd	1 0 1 0 1 1 1	vlw.v vd,offset(rs1)
1 imm[10:5]	vs2	rs1		imm[4:0]	1 0 1 0 1 1 1	vsw.v vs2,offset(rs1)
off[12|10:5]	vs2	vs1	0 0 0	off[4:1|11]	1 0 1 1 1 1 1	vbeq vs2, vs1, offset
			0 0 1			vbne vs2, vs1, offset
			1 0 0			vblt vs2, vs1, offset
			1 0 1			vbge vs2, vs1, offset
			1 1 0			vbltu vs2, vs1, offset
			1 1 1			vbgeu vs2, vs1, offset
off[12|10:5]	vs2	0	0 1 1	off[4:1|11]	1 0 1 1 1 1 1	vjoin vs2, offset
0 0 0 0 0 0 0	rs2	rs1	0 1 0	rd	0 0 0 1 0 1 1	endprg
0 0 0 0 0 0 1	vs2	imm[4:0]	0 1 0	rd	0 0 0 1 0 1 1	barrier rd, rs1, imm
0 0 0 0 1 0 m	vs2	0	0 1 0	vd	0 0 0 1 0 1 1	vfexp vd,v2,v0.mask
0 0 0 0 0 1 m	vs2	vs1	0 0 0	vd	0 0 0 1 0 1 1	vftta.vv vd,v2,v1,v0.mask

寄存器和 ABI

寄存器设置

架构寄存器数目：sGPR 64 个， vGPR 256 个，元素宽度均为 32bit。64bit 数据使用 register pair。

物理寄存器数目：sGPR 256 个，vGPR 1024 个，由硬件实现架构寄存器到物理寄存器的映射。

编译器提供 GPR 的使用量（vGPR 和 sGPR 的实际使用数目，是 4 的倍数），硬件根据实际使用情况分配更多的 workgroup 同时调度。

> 从 RVV 视角下看，向量寄存器宽度 vlen 固定为 num_thread*32bit，硬件上相当于 vsetvli 指令的 SEW=32bit，ma，ta，LMUL=1。 从 SIMT 编程视角看，每个 thread 拥有至多 256 个宽度为 32bit 的 vGPR，而 workgroup 拥有 64 个 sGPR。整个 workgroup 只需要做一次的操作，如 kernel 和非 kernel 函数中的地址计算，就使用 sGPR；如果有分支的

情况，则使用 vGPR，例如非 kernel 函数的参数传递。

一个warp(32thread)拥有的寄存器资源：vgpr0-255, sgpr0-127，每个格子代表32bit								
	每个thread私有的一个float16 x变量						thread间有一个公共的float4 y变量	
	thread31	thread30	…	thread2	thread1	thread0		所有thread共有
v0	x.0	x.0		x.0	x.0	x.0	s0	
v1	x.1	x.1		x.1	x.1	x.1	s1	
v2	x.2	x.2		x.2	x.2	x.2	s2	
v3	x.3	x.3		x.3	x.3	x.3	s3	
v4	x.4	x.4		x.4	x.4	x.4	s4	
v5	x.5	x.5		x.5	x.5	x.5	s5	
v6	x.6	x.6		x.6	x.6	x.6	s6	
v7	x.7	x.7		x.7	x.7	x.7	s7	
v8	x.8	x.8		x.8	x.8	x.8	s8	y.0
v9	x.9	x.9		x.9	x.9	x.9	s9	y.1
v10	x.A	x.A		x.A	x.A	x.A	s10	y.2
v11	x.B	x.B		x.B	x.B	x.B	s11	y.3
v12	x.C	x.C		x.C	x.C	x.C	s12	
v13	x.D	x.D		x.D	x.D	x.D	s13	
v14	x.E	x.E		x.E	x.E	x.E	s14	
v15	x.F	x.F		x.F	x.F	x.F	s15	
v16							s16	
	v0的[31:0]是thread0私有的，[63:32]是thread1私有的，…							
	所以OpenCL的向量类型只能使用分组寄存器表达							

分组寄存器由编译器展开，以提供对 OpenCL 向量类型的支持。

分组寄存器在硬件上需要多周期发射，且寄存器依赖不便于检测，在编译器上完成这一步要容易很多。 后续可以考虑针对标量 load/store 提供分组寄存器操作，标量访存指令有其特殊性：对连续地址访问的提升很大，如 GCN3 中的 S_LOAD_DWORDX8。

特殊寄存器

- x0：0 寄存器
- x1：ra 返回 pc 寄存器
- x2：sp stack pointer - localmem baseaddr
- x4：tp privatemem baseaddr

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，以及在编程中显式声明了__local 标签的数据（即 sGPR spill stack slots 和 localmem 的访问，实际上 sGPR spill

stack slots 和 local data 都将作为 localmem 的一部分）。
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。

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-v15 作为返回值。

自定义 CSR

name	addr	description
CSR_TID	0x800	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 thread id。
CSR_NUMW	0x801	该 workgroup 中的 warp 总数
CSR_NUMT	0x802	一个 warp 中的 thread 总数
CSR_KNL	0x803	该 workgroup 的 metadata buffer 的 baseaddr
CSR_WID	0x805	该 workgroup 中本 warp 对应的 warp id，其前 n-bit 为 workgroup id
CSR_LDS	0x806	该 workgroup 分配的 local memory 的 baseaddr，同时也是该 warp 的 stack 基址
CSR_GIDX	0x808	该 workgroup 在 NDRange 中的 x id
CSR_GIDY	0x809	该 workgroup 在 NDRange 中的 y id
CSR_GIDZ	0x80a	该 workgroup 在 NDRange 中的 z id

处理器模式

仅支持机器模式，程序启动前即会配置好 CSR 并打开 TLB。
页表管理和内存管理由驱动完成，当前不支持 Unified Virtual Memory，不会发

生 page fault。
暂不支持异常和中断处理。

地址空间

目前按照 32 位地址空间设计。MMU 和页表按 sv39 格式、64 位地址设计，输入和输出的地址默认补零以兼容 32 位地址。
privatemem 的访问需要使用专门的 vlw.v 指令，该指令会为每个 thread 自动计算其地址偏移量。在编译器视角，每个线程可用的 privatemem 空间是连续的 0-1kB 空间，硬件会自动转换以便于 warp 的连续访存。
localmem 和 globalmem 使用地址字段进行区分，小于 local_mem_max 的地址均认为是访问 localmem 的。
localmem 使用实地址，globalmem、privatemem、constantmem 使用虚地址，由驱动和 MMU 共同实现分配和管理（物理上这三者都将映射到 ddr）。

微架构（硬件视角）

任务分配和汇编相关

CTA 任务分配

在硬件层面，将按照 32 个 thread 组成一个 warp 的形式，作为整体在 SM 硬件上进行调度。同一个 block 的 warp 只能在同一个 SM 上运行，但是同一 SM 可以可以容纳来自不同 block 甚至不同 grid 的若干个 warp。
CPU 发送给 GPU 的任务以 workgroup 为基本单位，由 CTA scheduler 接收，CTA scheduler 会按 block 中包含的总 warp 数信息，以及需要占用的 local memory、sharedmemory 大小，将 block 对应分配到空闲（即剩余资源足够）的 SM 上。
CTA scheduler 以 warp 为单位逐个发送给 SM，同一 workgroup 的 warp 会分配到同一 SM 中，warp_slot_id 的低位即表明了该 warp 在当前 workgroup 中的 id，高位表明了 workgroup 本身所属的 id。相应的，SM 会通过此 id，计算出当前 warp 在所属 workgroup 中的位置，并将该值置于 CSR 寄存器中，供软件使

用。分配的 localmem baseaddr 需要通过 CSR 读取，而 privatemem baseaddr 和 register base 则由硬件隐式映射。 **由于一个 warp 只有一套 CSR**，因此 thread_id 需要用 vid.v+CSR_TID 计算出。

汇编编程说明

1. ~~需要通过 vid.v 与 csrrs CSR_TID 相加来获得 thread 各自的 id~~get_global_id()通过 vid.v + csrr CSR_TID 三条指令实现。
2. ~~访存时预先读取 CSR_GDS 和 CSR_LDS 作为数据基址链接器分配好地址~~，驱动程序分页管理
3. 自定义指令的使用：
    1. predicate：我们在支持 rvv 定义的软件控制 mask 的同时，也支持用自定义指令来启动隐式的硬件 predicate，详见自定义指令一节。
    2. warp（即 kernel）运行结束时需要显式使用 endprg 指令。
    3. 同一 block 内 thread 同步，使用 barrier 指令。

其余行为与 rvv 编程一致。

对于超过单组硬件处理能力长度的向量数据，支持使用 rvv 中定义的 stripmining 方式执行，默认单次处理 num_thread 个数据。与向量处理器不同的是，可以用软件 mask 实现，也可以用 SIMT-stack 实现，也可以在 block 大小允许时拆分为更多 warp 去调度。

> SIMT-stack 补充：目前从软件视角看单 warp 执行，功能与 rvv mask 完全一致。优势在于 1)所有 thread 方向一致时，可跳过 if 分支或 else 分支；2)减少对寄存器堆(v0)访问次数，减少获取操作数时的 bankconflict；3)实现快速嵌套分支，目前硬件支持最坏情况下的 num_thread-1 层嵌套；4)为后续硬件实现 multi-path IPDOM、独立线程调度、warp 合并等提供便利。

指令集架构

RVV 与 GPGPU 的结合

《量化研究方法》中提到了向量处理单元与多线程 GPU 在 SIMD 层面上的工作形式十分相似，向量处理器的车道与多线程 SIMD 的线程是相似的。区别在于通常 GPU 的硬件单元更多，chime（钟鸣）更短，向量处理器通过深度流水线化的访问来隐藏延迟，GPU 则是通过同时多 warp 切换来隐藏延迟。因此在向量层面的操作上，RVV 足以覆盖住 GPGPU 中的操作。此外，形如 AMD 和 turing 后的 NV，提供了标量 ALU，也是借鉴了向量处理器的方式。

因此，在 RVV 的基础上添加自定义的分支控制指令（实际上沿用 RVV 本身的 mask 也能实现）、线程同步指令、线程控制指令，就能实现 GPGPU 的功能。

为了最大限度的保留对 RVV 开源工具链的兼容性，我们对 RVV 中的大部分指令都进行了支持。少数不支持的指令包括：1. 涉及线程间数据交换的 shuffle 等指令，在 GPGPU 中线程间通常是独立操作，数据依赖需要用 atomic 或 barrier 显式操作 2. 向量寄存器长度和宽度变化的指令，GPGPU 中几乎不会触及（少有的几条向量或者量化相关的功能会需要类似的功能） 3. 64bit 相关的指令，在后续版本将支持。

在 RVV 的 stripmining 基础上添加 warp 级别并行，或许能在更优尺度上裁剪向量/SIMT 指令，探索划分和调度空间。

寄存器设计

单个 SM 上能同时承载的最大 warp 数为 num_warp，每个 warp 由 num_thread 个线程组成。每个 warp 都有一套自己的寄存器，每个标量寄存器的宽度为 32bit，每个向量寄存器的宽度为 32*num_thread，并归属于各个 thread 私有。

物理寄存器堆采用统一方式，根据各 warp 的实际使用情况动态分配。

虽然所用指令形式和意义相同，但区别于 rvv，我们目前实现的 GPGPU 中并不支持向量寄存器的长度和数量变化（长度固定为 32bit，数量固定为 num_thread），因此对于 vsetl 系列指令，只有返回的剩余元素数量是有效的。

地址映射

用地址范围来区分 localmem 和 globalmem，其中 localmem 使用实地址，globalmem、privatemem、constantmem 由驱动和 MMU 共同实现分配和管理。

> GPU 中的物理地址空间包括 loaclmem 和 globalmem，早期版本 cuda 和 OpenCL 编程中需要显式声明地址属性，在 PTX 中每个地址都带有属性声明了其类型。 目前承影这部分代码尚未修改，所有 SM 共用一个 4GB 的全局地址空间，其中地址为 0-128kB 的字段将被映射到 SM 内部各自的 sharedmem 上，从 global_baseaddr 到 4G 的空间则映射到同一块 ddr 的相同地址上（即该部分使用物理地址）。 coherence 在 GPU 中是由软件显式管理的，通过 flush 和 invalidate 实现。

指令集范围

目前支持 RVV 中的指令包括以下类型：
1. 计算类，包括整数运算（加减、比较、移位、位运算、乘、乘加、除）、单精度浮点运算（加减、乘、乘加、除、整型转换、比较），支持带 mask 执行
2. 访存类，包括三种访存模式，以及 byte 级读写，支持带 mask 执行
3. mask 类，包括比较及逻辑运算，但不包括 vmsbf 等涉及 thread 间通信的指令，也不支持 gather 等操作
对于改变向量位宽的指令暂不支持，但 vsetl 系列指令可以返回 vl 供 stripmining 使用。 RV32I 中支持除 ecall ebreak 外的指令，M F 中支持 32bit 相关指令。

目前支持的自定义指令包括：
1. predicate：在支持 rvv 定义的软件控制 mask 的同时，也支持用自定义指令来启动隐式的硬件 predicate，由 vbeq 等启动的隐式硬件 predicate 将默认对后续指令生效，直到触发新的 vbeq 或 join 时才会改写。启动的方式是使用自定义的 vbeq 系列线程分支指令，该指令会启动一个 split，计算出当前分支的 mask 情形，并将 else 对应的 mask 压入 SIMT-stack，然后带有 mask 执行 if 段，待 if 段末尾遇到 join 后再将 else 段及其对应 mask 出栈，待 if 段执行完成后合并恢复 mask，分支结束。该过程可以嵌套，压栈的最坏情况深度等同于单 warp 中的线

程数 32（如果每次总选择最多的方向去压栈而非默认压 if，那么栈深度只需要 5 即可）。此外，如果分支计算出全走其中一条，将不会压栈并跳过另一条分支，带有 branch divergence 的 for 循环也可以用此机制实现。

2. barrier：用该指令可以实现 warp 间的同步，每个遇到该 barrier 指令的 warp 都会等待，直到该 workgroup 中所有未结束的 warp 都遇到此指令才会再次继续。

3. endprg：warp 运行结束时需要显式使用 endprg 指令。

4. regext{i}：为后一条指令扩展使用的寄存器数目和立即数宽度。

5. 其它为提高代码效率的自定义指令，可参见“kernel 内汇编指令说明 - 指令集范围 - 自定义指令”一节

驱动接口

遵循 CTA 调度器提供的接口信息支持，配置好使用的寄存器数目、localmem baseaddr、warp id 等。

微架构设计

总体分为前端和后端，前端包括了取指、译码、指令缓冲、寄存器堆、发射、记分板、warp 调度，后端包括了 ALU、vALU、vFPU、LSU、SFU、CSR、SIMT-stack、warp 控制等。涉及寄存器连接的模块间均采用握手机制传递信号。

warp 调度

warp 调度器主要的功能包括：

1. 接收 CTA 调度器提供的 warp 的信息，分配 warp 所属的硬件单元并预设 CSR 寄存器值，激活该 warp 并标记所属的 block 信息。在 warp 执行完成后，将该信息返回给 CTA 调度器并释放对应硬件。

2. 接收流水线发送的 barrier 指令信息，将指定的 warp 锁住直到其所属的 block 的所有活跃 warp 到达此 barrier。

3. 选择发给 icache 的 warp，通常采用贪婪的策略，但当 icache 发生 miss，或 ibuffer 已满时会将该 warp 的 pc 回退并切换到下一个 warp 发射。

4. 选择发给执行单元的 warp，通常采用轮询的策略，选择出当前指令缓冲有效、记分板未显示冲突、执行单元空闲的指令。切换 warp 仅需要一个周期。

取指

每个 warp 存储各自的 pc，被选中送入 icache 的 pc 会+4，其余保留原值，遇到跳转时则替换为目标地址。

译码

icache 命中的指令进行译码，译码器根据指令内容转换出对应的控制信号，并送入对应的指令缓冲中。

指令缓冲

指令缓冲是一系列的 FIFO，每个 warp 有各自的 ibuffer，接收译码后的输入并等待选中发射。

操作数收集器

操作数收集器会接收来自指令缓冲中的请求，依据所需数据类型，经由 crossbar 向寄存器堆访问获取数据，获取后即进入待发射状态。
寄存器堆采用 unified 方案设计：硬件上单个 SM 里共有 1024 个向量寄存器和 512 个标量寄存器，单个 warp 可以使用多达 256 个向量寄存器和 64 个标量寄存器，由硬件完成物理寄存器的分配。编译器指明使用的寄存器数目（4 的倍数），单个 warp 使用寄存器数目越少，就能在硬件上分配更多的 warp 同时运行。
寄存器堆硬件上分了 4-bank，每个 bank 一读一写 port。
目前版本标量和向量寄存器不支持同时访问，后续会修改。

发射

发射仲裁由 warp 调度器进行，被选中的控制信号与源操作数一起，依据识别其所需运算单元的类型，发送到对应运算单元执行。
每个周期可以发射两条指令。

记分板

每个 warp 有各自的记分板，当一条指令发射成功后，所写入的寄存器将被记分板标记，下一条指令若会读写已被标记的寄存器，则不允许发射，直到指令执行完成

记分板释放对应寄存器。
分支、线程分歧、跳转、barrier 指令也会被锁住，只有这些指令完成判断且能继续顺序执行时才会释放。执行 barrier 期间，会同时清空除 ibuffer 外的流水线（属于该 warp 的部分）；若发生跳转，会同时清空所有流水线（属于该 warp 的部分），此后记分板解除锁定。

写回

各个运算单元在输出级均有 FIFO，等待写回寄存器的结果暂存在其中，由 Arbiter 选中后写回寄存器。写回标量寄存器与向量寄存器是完全独立的数据通路。

ALU

ALU 中进行标量运算，包括 warp 间共用的数据，以及跳转控制等。

vALU

是单个 ALU 的复制，是核心的整型运算单元，供 warp 的多线程车道进行运算。典型运算消耗 1cycle。支持折叠为 num_lane 个。

vFPU

是核心的浮点运算单元，供 warp 的多线程车道进行运算，标量浮点运算也在此进行。全流水设计，乘法和乘加有复用数据通路。典型乘加消耗 5cycle，乘法消耗 3cycle，并支持折叠为 num_lane 个。

MUL

乘法运算单元，供 warp 多线程车道进行运算。使用 wallace tree 结构，全流水设计，典型乘法消耗 2cycle，并支持折叠为 num_lane 个。

LSU

是核心的访存单元，会根据地址范围判断将读写请求发送给 sharedmem 或 dcache（再由 L1 dcache 访问 L2cache，再访问 ddr 即 globalmemory）。LSU 中有 MSHR 形式的结构，可以一次存储和记录多个 LSU 请求，也会收集 dcache 和 sharedmem 返回的数据，集齐后再返回给流水线。

LSU 中会完成 strided 及任意模式下向量地址的计算，并根据地址范围以 cacheline 为单位进行合并访问。最理想的情况（指地址连续且对齐 cacheline）一次访存即可取出单个 warp 所需结果。
bank conflict 由 sharedmem 和 dcache 自行处理。
在 LSU 中还会记录现有的访存请求信息，以实现 fence 指令。当遇到该指令后，会让所属 warp 的所有访存请求处理完成（读数据返回数据，写数据返回写响应）后再发送新的访存请求。

CSR

在 warp 启动时，对应的 CSR 会设置好应用程序所需的一些值，包括 thread id 等。 vsetl 也在此计算。其余与 riscv cpu CSR 的功能一致。

SFU

区别于 PTX 中提供的 sin cos 等函数，目前的 SFU 只支持了 rv 定义的整数除法取余、浮点除法、浮点平方根功能。
本身运算就需要多周期完成，加上 SFU 中的运算单元数量少于 lane 数，因此如果未 mask 的线程较多时，chime 会更长。

TC

tensorcore，全流水，支持特定格式下的张量计算（示意图待补充）。

warp 控制

barrier、endprg 会发送给 warp 调度器处理。
后续会考虑支持动态并行，添加 warpgen 指令。

SIMT-stack

SIMT stack 的主要功能如下：维护分支嵌套控制流，保障程序运行的正确性；在实际没有分支分歧发生时，跳过不必要程序段的执行。
由 SIMT-stack 设置的隐式 mask 会在该 warp 执行过程中一直生效，直到有其它分支管理支持对其进行修改。该 mask 与 rvv 软件形式的 mask 可以叠加生效。
与分支管理相关的自定义扩展指令集有上表所示 7 条，其中 1-6 条为分支指令。

以 vbeq 指令为例，需要完成的功能为：取源操作数 vs2 与 vs1，valu 模块对这两个向量寄存器中的元素一一进行比较，对于第 i 个元素，若 vs2[i]=vs1[i]，则计算结果 out[i]为 1，最终 valu 的输出结果 out 为分支指令对应的 else 路径掩码，同时译码模块将向分支管理模块发送分支发生标记以及 else 路径 PC 起始值 PC branch。

整体微架构方案如图所示。