乘影GPGPU架构文档手册v2.02

编写人：杨轲翔 (yangkx20@mails.tsinghua.edu.cn)
于芳菲 (yff22@mails.tsinghua.edu.cn)
联系人：何虎 (hehu@tsinghua.edu.cn)
杨泽夏 (yang-zx21@mails.tsinghua.edu.cn)
清华大学集成电路学院dsp-lab

乘影是基于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)

本文档主要描述了乘影GPGPU在软硬件视角下的功能描述，部分功能直接以乘影下一版本标定，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，或邮件联系联系人。

简介

本文档描述了乘影GPGPU的设计内容，包括OpenCL编程视角和微架构视角。

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，**本架构本质上是SIMT**。

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：（在本文档中局部内存、共享内存可能同时使用localmem和sharedmem来指代；线程束和线程采用cuda的说法warp和thread；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

变量名	解释
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SM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，pocl以kernel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。pocl传递给硬件驱动后，硬件驱动将以workgroup为单位，把任务发送到CTA-scheduler处理。
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pocl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、print buffer分配的空间

任务启动时，由硬件驱动直接传递的信号为：

- 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

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
#define KNL_PRINT_ADDR 48
#define KNL_PRINT_SIZE 52
*/
```

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 localmemory pointer
    addi tp, x0, 0 # set privatememory pointer

    # 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
```

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。

架构说明

硬件上的ABI、指令集、寄存器接口的部分，以及对内存系统的说明，以应对OpenCL kernel的编程需求。

指令集范围

RV32V

实际选择的指令集范围为：RV32 I M A zfinx zve32f
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。
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。

	乘影支持情况	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-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；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	

	乘影支持情况	指令变化
RV32V-ExceptionHandling	不支持，视应用和编译器需求再考虑添加	

自定义指令

barrier 线程同步指令

```
barrier x0,x0,imm # meet barrier
barriersub x0,x0,imm # barrier for subgroup
```

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。

imm为5bit，具体编码如下：

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

开启_opencl_c_subgroups feature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。

endprg 任务结束指令

```
endprg x0,x0,x0 # 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 线程分支控制指令

```
vbeq vs2, vs1, offset # set predicate vs2==vs1, and set branch address pc+4+offset
join v0, v0, 0 # pop stack and jump
```

隐式SIMT-stack实现分支控制。join的源操作数默认为0。

vbeq参照beq提供了vbne vblt vbge vbltu vbgeu版本，指令编码修改了func3段。

setrpc 分支汇合地址写入指令

```
setrpc rd, rs1, offset # set reconvergence pc and rd = rs1 + sext(offset)
```

将分支汇合地址写入rpc的csr寄存器。

regext{i} 寄存器扩展指令

```
regext x0,x0,imm12 # (x3,x2,x1,xd)=imm12, register index extend
regexti x0,x0,imm12 # (imm[10:5],x2,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。

regexti只对V扩展的VI类OP-imm5指令生效，将其视为imm11。对其它立即数指令不支持立即数扩展，可用regext进行寄存器扩展。

当前版本编译器对寄存器扩展指令，支持按需要进行扩展。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regext这一阶段。

regpair{i} 寄存器对拼接指令

```
regpair x0,x0,imm12 # (x3,x2,x1,xd)=imm12, register pair with register index extend
regpairi x0,x0,imm12 # (imm[10:5],x2,xd)=imm12, register pair with imm and register index extend
```

用于扩展64位操作，将目标源寄存器与相邻寄存器拼接作为64位源操作数计算，同时包含寄存器扩展语义，支持寄存器索引及立即数扩展。配合其接续指令，将声明偶数索引的寄存器作为低32位，其相邻奇数索引寄存器作为高32位，即寄存器0与1成对、2与3成对，若其接续指令声明的源寄存器或目的寄存器索引为奇数，则不进行对拼接操作，按照32位操作数操作。

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0-1k的连续地址，由硬件根据thread_id和CSR_PDS完成偏移。

```
vlw.v vd, imm11(rs1) # vd <- mem[(rs1+imm11)*num_thread*num_warp+thread_idx]
vsw.v vs2, imm11(rs1) # mem[(rs1+imm11)*num_thread*num_warp+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(vs1) # mem[vs1+imm12] <- vs2
```

vlw12参照lw提供了vlh12 vlb12 vlhu12 vlbu12版本，vsw12也有vsh12 vsb12版本。

vadd12.vi 带12位立即数向量整数加减指令

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

vftta.vv 浮点卷积指令

```
vftta.vv vd, vs2, vs1, v0.mask # vd <- vs2 conv vs1 + vd
```

vfexp.v 浮点指数指令

```
vfexp.v vd, vs2, v0.mask # vd <- exp(vs2)
```

完整指令编码及描述如下：

31 25	24 20	19 15	14 12	11 7	6 0	assemble	description	
off[12|10:5]	vs2	vs1	0 0 0	off[4:1|11]	1 0 1 1 0 1 1	vbeq vs2, vs1, offset	if(vs2 == vs1) PC += sext(offset) else PC += 4	分支控制指令
			0 0 1			vbne vs2, vs1, offset	if(vs2 != vs1) PC += sext(offset)	
			1 0 0			vblt vs2, vs1, offset	if(vs2 <s vs1) PC += sext(offset) //signed	
			1 0 1			vbge vs2, vs1, offset	if(vs2 >=s vs1) PC += sext(offset) //signed	
			1 1 0			vbltu vs2, vs1, offset	if(vs2 <u vs1) PC += sext(offset) //unsigned	
			1 1 1			vbgeu vs2, vs1, offset	if(vs2 >=u vs1) PC += sext(offset) //unsigned	
0 0 0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 1 0	0 0 0 0 0	1 0 1 1 0 1 1	join v0,v0,0	pop stack and jump	
imm[11:0]		rs1	0 1 1	rd	1 0 1 1 0 1 1	setrpc rd,rs1,offset	set reconvergence pc and rd = rs1 + sext(offset)	
imm[11:0](x3,x2,x1,xd)		rs1	0 1 0	rd	0 0 0 1 0 1 1	regext x0,x0,imm	扩展寄存器编号。rs1和rd默认为零	寄存器扩展指令
imm[11:0](imm[10:5],x2,xd)			0 1 1			regextі x0,x0,imm	扩展vop.vi指令的寄存器和立即数，rs1和rd默认为零	
imm[11:0](x3,x2,x1,xd)		rs1	1 1 0	rd		regpair x0,x0,imm	寄存器拼接，包含扩展寄存器编号的语义	寄存器对拼接指令
imm[11:0](imm[10:5],x2,xd)			1 1 1			regpairi x0,x0,imm	寄存器拼接，包含扩展寄存器编号及立即数的语义	
0 0 0 0 0 0 0		rs1				endprg x0,x0,x0	kernel运行结束	同步和任务控制指令
0 0 0 0 0 1 0	rs2	imm[4:0]	1 0 0	rd		barrier x0,x0,imm	对应barrier(),imm提供memory_scope和cl_mem_fence_flags	
0 0 0 0 0 1 1		imm[4:0]				barriersub x0,x0,imm		
imm[11:0]		vs1	0 0 0	vd		vadd12.vi vd, vs1, imm	vd = vs1 + imm	自定义计算指令
0 0 0 0 1 0 m	vs2	0	1 1 0	vd	0 0 0 1 0 1 1	vfexp vd,v2,v0.mask	vd = exp(v2)	
0 0 0 0 1 1 m	vs2	vs1	1 0 0		0 0 0 1 0 1 1	vftta.vv vd,v2,v1,v0.mask	vd = v2 conv v1 + vd	
imm[11:0]		vs1	0 1 0	vd	1 1 1 1 0 1 1	vlw12.v vd, offset(vs1)	vd<-mem[addr], addr=vs1+offset	自定义访存指令，对标标量访存的长立即数版本
			0 0 1			vlh12.v vd, offset(vs1)		
			0 0 0			vlb12.v vd, offset(vs1)		
			1 0 1			vlhu12.v vd, offset(vs1)		
			1 0 0			vlbu12.v vd, offset(vs1)		
imm[11:5]	vs2	vs1	1 1 0	imm[4:0]	1 1 1 1 0 1 1	vsw12.v vs2,offset(vs1)	mem[addr]<-vs2, addr=vs1+offset	
			0 1 1			vsh12.v vs2,offset(vs1)		
			1 1 1			vsb12.v vs2,offset(vs1)		
0 imm[10:0]		vs1	0 1 0	vd	0 1 0 1 0 1 1	vlw.v vd,offset(vs1)	vd<-mem[addr] addr=(vs1+imm)*num_thread_in_wg+thread_idx+csr_pds funct3字段编码与lw lh lb等访存指令一致	自定义访存指令，专用于访问private memory
			0 0 1			vlh.v vd,offset(vs1)		
			0 0 0			vlb.v vd,offset(vs1)		
			1 0 1			vlhu.v vd,offset(vs1)		
			1 0 0			vlbu.v vd,offset(vs1)		
1 imm[10:5]	vs2	vs1	0 1 0	imm[4:0]	0 1 0 1 0 1 1	vsw.v vs2,offset(vs1)	mem[addr]<-vs2 addr=(vs1+imm)*num_thread_in_wg+thread_idx_csr_pds funct3字段编码与sw sh sb等访存指令一致	
			0 0 1			vsh.v vs2,offset(vs1)		
			0 0 0			vsb.v vs2,offset(vs1)		

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

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。

description	name	addr
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_TID	0x800
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_GIDX	0x808
该workgroup在NDRange中的y id	CSR_GIDY	0x809
该workgroup在NDRange中的z id	CSR_GIDZ	0x80a
向print buffer打印时用于与host交互的CSR	CSR_PRINT	0x80b

内存系统

乘影GPGPU每个SM有单独的L1内存子系统，包括指令缓存、共享内存（local memory/scratchpad memory）、数据缓存、常量缓存。其中指令Cache由前端取指级直接访问，其他存储器由指令通过LSU访问。所有SM访问同一块L2内存子系统。

地址空间

目前按照32位地址空间设计。 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空间是从CSR_PDS开始连续的0-1kB空间，硬件会自动转换以便于warp的连续访存。
localmem和globalmem的访问使用vle32.v vloexi32.v vlw32.v指令访问。二者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，访问片上SRAM；globalmem、privatemem、print buffer使用的地址由驱动实现分配和管理（物理上这三者都将映射到ddr）。

一致性和连贯性特

SM间数据缓存的一致性（memory coherence）由程序员显式通过同步指令维护（Consistency-Directed Coherence），硬件不维护一致性。同步指令对应到RISC-V A扩展中的特定指令，硬件上cache提供flush和 invalidate功能。
连贯性（memory consistency）在乘影中可以理解为当前SM访存指令执行结果对其他SM可见的顺序是否和当前SM执行这些访存指令的顺序相同。在RVWMO（RISC-V Weak Memory Ordering）中，普通访存操作的连贯性要求由PPO（Preserved Program Order）规则1和规则2定义。对于常规读写操作，乘影L1数据缓存的微架构表现如下表：

	【相	同	地	址】	【不	同	地	址】
操作顺序类型	R-R	W-W	W-R	R-W	R-R	W-W	W-R	R-W
(对其他SM)是否保序	是	是	是	是	是	否	否	是
RVWMO PPO的保序要求	要求	要求	不要求	要求	不要求	不要求	不要求	不要求

*本表格中，A-B表示访存操作A的程序顺序早于访存操作B。

L1指令缓存

主要特征如下：

- （尚未固定）2路组相联，缓存行大小为blocksize（单位Byte），总容量64*blocksize；
- 每次LSU访问的最大返回宽度为4B；
- 每次L1-L2之间数据读写的最大宽度均为128B（一个缓存行）；
- 替换策略支持LRU、FIFO替换策略；
- 支持对正在进行的请求进行无效化；
- （尚未固定）最多同时存在128个未完成L2访问请求；支持相同缓存行的L2访问请求合并，最大合并数8。

L1数据缓存

主要特征如下：

- （尚未固定）2路组相联，缓存行大小为blocksize（单位Byte），总容量64*blocksize；
- 一般的，blocksize = num_thread；
- 虚拟地址索引、虚拟地址标记（VIVT）；
- 每次LSU访问的最大返回宽度为blocksize；
- 每次L1-L2之间数据读写的最大宽度均为blocksize（一个缓存行）；
- 写策略为写回-写不分配；
- 替换策略支持LRU、FIFO替换策略；
- 支持对整个数据高速缓存的无效和清除操作，支持对单条缓存行的无效和清除操作；
- （尚未固定）最多同时存在128个未完成L2访问请求；支持相同缓存行的L2访问请求合并，最大合并数8。

数据缓存支持的指令类型包括：

- 基础指令集I：
  - 访存指令：LOAD和STORE
  - 访存排序指令：FENCE
- 原子指令扩展A：
  - 原子操作指令：AMO
  - 预留性读/条件写指令：LR/SC
- 向量指令扩展V：
  - 向量读指令：VL，VLS，VLX
  - 向量写指令：VS，VSS，VSX
  - 自定义向量访存指令

share memory

主要特征如下：

- （尚未固定）2路组相联，缓存行大小为blocksize（单位Byte），总容量64*blocksize；
- 一般的，blocksize = num_thread；
- 每次LSU访问的最大返回宽度为blocksize；

share memory支持的指令类型包括：

- 基础指令集I：
  - 访存指令：LOAD和STORE
- 原子指令扩展A：
  - 原子操作指令：AMO
- 向量指令扩展V：
  - 向量读指令：VL，VLS，VLX
  - 向量写指令：VS，VSS，VSX
  - 自定义向量访存指令

L2缓存

主要特征如下：

- 多路组相连，组数和路数可通过CacheParameters配置，缓存行大小以及最小可写单位可通过

InclusiveCacheMicroParameters配置，目前分别为128Bytes和4Bytes

- 物理地址索引，物理地址标记（PIPT）
- 写策略为写回-写分配
- 替换策略支持随机、伪LRU
- 支持对整个数据高速缓存的无效和清除操作，支持对单条缓存行的无效和清除操作
- 具备MSHR，支持非阻塞访存
- 除普通load、store操作外，支持AMO原子操作，通过Tilelink协议支持LR、SC操作
- 支持stride预取
- L2cache通过一个AXI Adapter发送符合AXI 4协议的数据给主存

处理器模式

仅支持机器模式，程序启动前即会配置好CSR。内存管理由驱动完成。
暂不支持异常和中断处理。

总线接口

乘影包含一个AXI4主设备接口和一个AXI4-Lite从设备接口。 L2Cache通过AXI4主设备接口访问片外存储器。
host通过AXI4-Lite接口，对workgroup进行配置。

互联网络

若干个SM组成一组SM_cluster， SM_cluster与L2Cache bank间通过crossbar连接。

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，thread_id需要用vid.v+CSR_TID计算出。

汇编编程说明

1. get_global_id()通过vid.v + csrr tid三条指令实现。
2. 输入参数和访存地址需要按照预设参数传递方式，从CSR读取使用
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 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由CTA-scheduler管理，globalmem、privatemem、constantmem由驱动分配和管理。

> 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。 目前乘影尚未实现MMU，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，对应fence指令。

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。

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。

warp控制

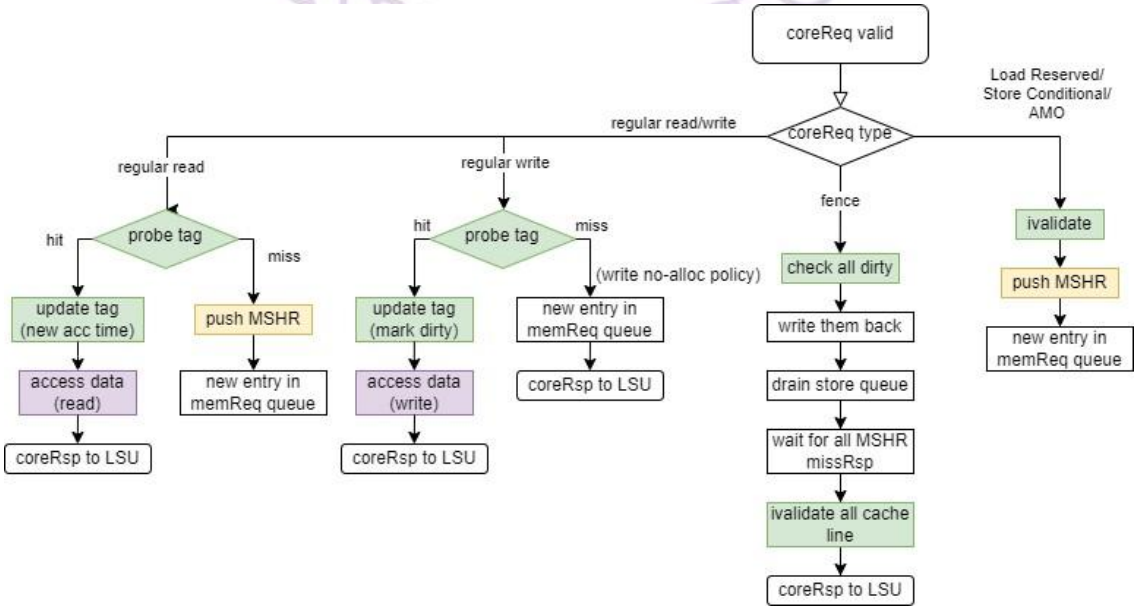
barrier、endprg会发送给warp调度器处理。
后续会考虑支持动态并行，添加warpgen指令。

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8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。

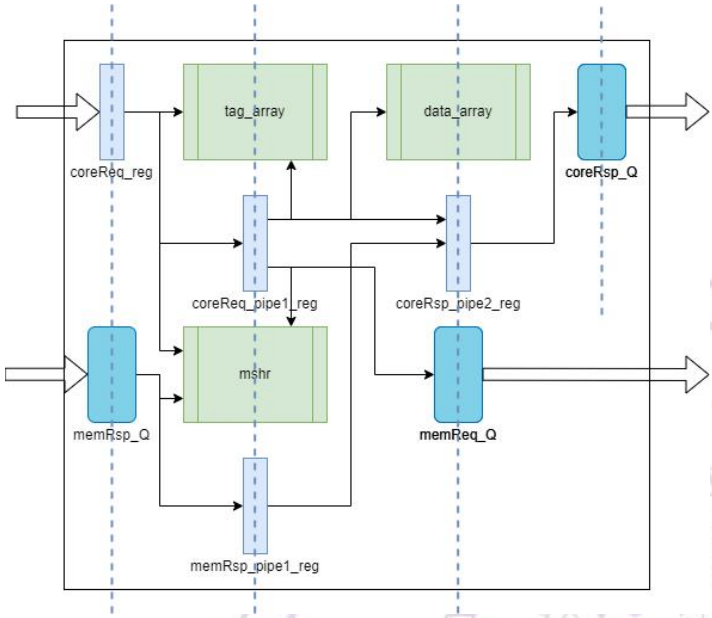
DCache

整个cache分为对LSU接入的两组接口coreReq和coreRsp，以及对更高层cache（L2）接入的两组接口memReq和memRsp。其中coreRsp直接接入SM核心流水线写回级。coreReq支持的请求类型包括：普通读请求（标量/向量）、普通写请求（标量/向量）、fence请求、预留性读请求、条件写请求、原子操作请求。这些请求在cache中经历的处理操作如下图：



这些请求操作大多数与指令一一对应。AMO指令是一个例外。AMO指令允许带有.aq和/或.rl标识符（acquire和release），未来RV标准指令集/扩展指令集也可能会添加普通访存指令携带该标识符的支持。对于携带此类标识符的访存指令，LSU会将其分割为一个不携带标识符的访存请求和一条对应的fence请求。

整个cache由三个流水级构成，示意图如下：



对coreReq路径来说：

S0：发出tag和mshr SRAM访问请求。

S1：根据请求类型和tag返回值决定后续操作方式：压栈memReq_Q 或/与 发出data访问请求。

S2：压栈coreRsp_Q。

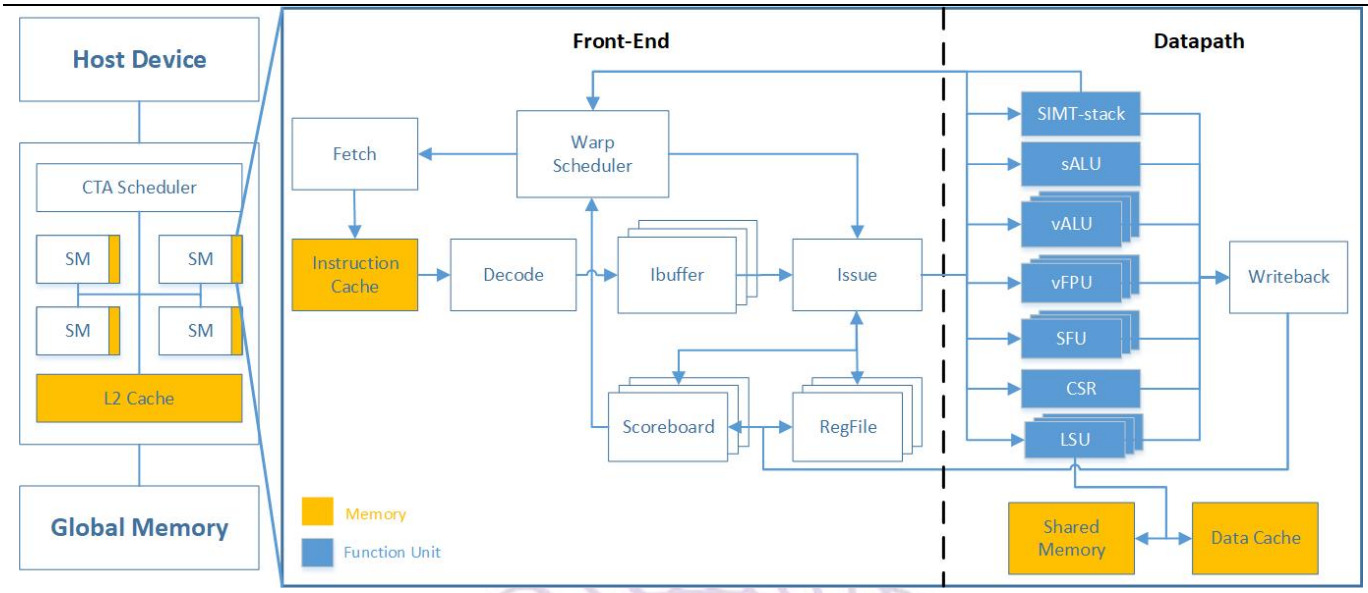
对memRsp路径来说：

S0：发出mshr SRAM访问请求。

S1：根据请求类型和mshr返回值，更新tag、组织coreRsp。

S2：为了避免流水线冲突引入的冗余流水级。

整体微架构方案如图所示。


Host Device
CTA Scheduler
SM
SM
SM
SM
L2 Cache
Global Memory
Front-End
Datapath
Fetch
Warp Scheduler
Instruction Cache
Decode
Ibuffer
Issue
Scoreboard
RegFile
SIMT-stack
sALU
vALU
vFPU
SFU
CSR
LSU
Writeback
Shared Memory
Data Cache
Memory
Function Unit