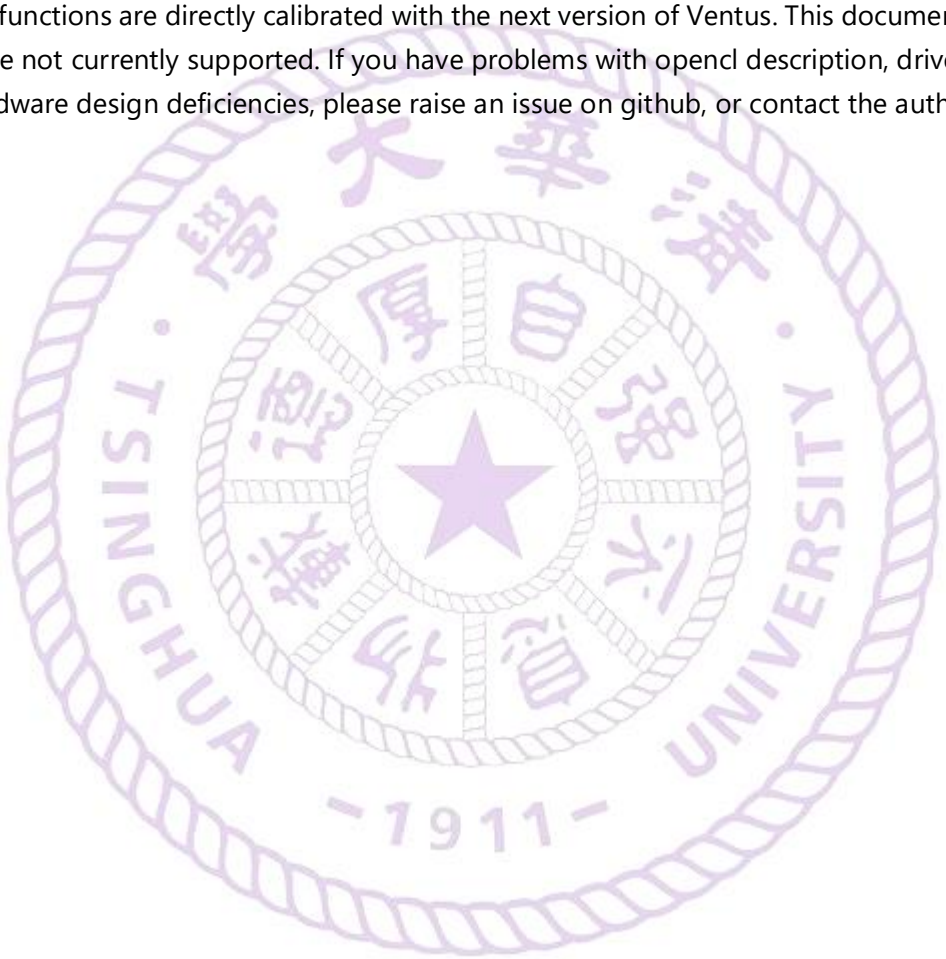
Ventus GPGPU Architecture Whitepaper v1.95

Written by：Kexiang Yang (yangkx20@mails.tsinghua.edu.cn)，Fangfei Yu (yff22@mails.tsinghua.edu.cn)

School of Integrated Circuits, Tsinghua University dsp-lab

Ventus is an open source GPGPU based on RISC-V vector extension, project home page: THU-DSP-LAB/ventus-gpgpu: GPGPU processor supporting RISCV-V extension, developed with Chisel HDL (github.com)

This document mainly describes the function of Ventus GPGPU from the perspective of software and hardware. Some functions are directly calibrated with the next version of Ventus. This document will note the situations that are not currently supported. If you have problems with opencl description, drivers, compiler functions, or hardware design deficiencies, please raise an issue on github, or contact the author by email.



Introduction

This document describes the design content of Ventus GPGPU, including OpenCL programming perspective and microarchitecture perspective. Ventus GPGPU instruction set uses RISC-V vector extension (hereinafter referred to as RVV) as the core to design GPGPU. Compared with RISC-V scalar instructions, RVV has richer expressive meanings, and can realize memory access characterization, distinguish between workgroup and thread operations, and other functions. The main idea is to use the v instruction as the behavior description of the thread at the compiler level, and merge the common data of thread->warp/workgroup into scalar instructions. A warp from the hardware is an RVV program. Usually, the vector element length is num_thread, and at the same time, the uniform instructions in the work group like common address calculation and jump instruction are executed as scalar instructions, that is, the Vector-Thread architecture. The hardware maps the warp to the lane of the RVV processor in time-sharing for execution. Compared with other SIMT architectures, the hardware compromise is that it cannot achieve complete per thread per pc, and it still needs to be executed in workgroup (or warp_split in branch state). The RVV instruction set has three aspects in variable length: hardware vlen changes; SEW element width changes; LMUL grouping changes. The characteristic of this architecture is that these three parameters have been fixed during compilation, and the number of elements is also fixed at num_thread in most cases, **this architecture is essentially SIMT**.

Terminology

- SM：streaming multiprocessor sGPR：
- scalar general purpose register vGPR：
- vector general purpose register

Memory division and programming model definition: (in this document, local memory and shared memory may be referred to by localmem and sharedmem at the same time; thread warp and thread will use the term warp and thread in cuda; other words use opencl terminology)

cuda	***opencl***	***explanation***
globalmem	globalmem	Global memory, described by __global, can be accessed by all threads of the kernel
constantmem	constantmem	Constant memory, described by __constant, is part of the global address space
localmem	privatemem	Private memory, each thread's own variables and kernel parameters, are part of global memory
sharedmem	localmem	Local memory, described by __local, for data exchange between threads in the same work-group
grid	NDRange	A kernel consists of multiple NDRanges, and one NDRange consists of multiple workgroups
block/CTA	workgroup	Workgroup, the basic unit of execution on the SM
warp	wavefront(AMD)	32 threads form a warp, visible only to hardware

cuda	*opencl*	*explanation*
thread	work-item	Thread/work-item, is the minimum unit described in OpenCL C programming

Parameters

variable	*explanation*
num_thread	Number of threads in a warp, the default value is 32
num_warp	The maximum number of warps allowed on the hardware per SM(can come from different workgroups)
num_block	The maximum number of workgroups allowed on the hardware per SM
num_lane	The number of threads that can be processed at one time in one SM on the hardware
localmem_max	The maximum space of localmem provided in an SM on the hardware

Therefore, num_thread*num_warp represents the maximum number of threads per SM.

Programming Model and Driver Functionality

Look at the device from an OpenCL perspective.

Correspondence with hardware

The entire GPU is used as a compute device, the SM corresponds to the Compute Unit (CU), and multiple execution units inside the SM correspond to multiple PEs.

Task execution model

Consistent with OpenCL, the workgroup is mapped onto the CU for execution, and each thread is mapped onto the PE for execution. The hardware will package the threads in units of warp (a group of 32 adjacent threads), showing the effect of SIMD execution. Currently NDRange is split into workgroups on the driver, and workgroups are split into warps on the hardware.

The functions provided by the driver

Use the opencl driver (pocl) to manage the command queue, create and allocate buffers. pocl assigns tasks in unit of kernels, and creates metadata buffers for each task. The shared memory space, the sequence between tasks and the event synchronization mechanism are also managed by pocl. After the pocl is passed to the hardware driver, the hardware driver will send the task to the CTA-scheduler in units of workgroups for processing. Add verilator-based ventus device and ISS spike ventus device to the pocl backend to complete physical address allocation and task startup. The buffers currently created by pocl include:

- Metadata buffer and kernel program of NDRange
- Argument buffer of kernel
- buffer explicitly referenced in kernel argument
- Space allocated for private mem, print buffer

When the task starts, the signal directly transmitted by the hardware driver is:

- PTBR // page table base addr
- CSR_KNL // metadata buffer base addr
- CSR_WGID // The id of the current workgroup in SM, only for hardware identification
- CSR_WID // warp id，the current warp's position in the workgroup
- LDS_SIZE // localmem_size，compiler provided, localmem space that the workgroup needs to occupy. privatemem_size is allocated according to 1kB per thread by default.
- VGPR_SIZE // vGPR_usage，compiler provided, number of vGPRs actually used by the workgroup (alignment 4)
- SGPR_SIZE // sGPR_usage，compiler provided, number of sGPRs actually used by the workgroup (alignment 4)
- CSR_GIDX/Y/Z // workgroup idx in NDRange
- host_wf_size // number of thread per warp
- host_num_wf // number of workgroup per wavefront

Runtime behavior

When the kernel is started, the parameters of NDRange are passed through the buffer of metadata, and the contents of the buffer are:

```
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
*/
```

The parameters of the kernel are passed by another piece of kernel_arg_buffer, which will prepare the arguments of the kernel in order, including specific parameter values or addresses of other buffers. Only the address knl_arg_base of kernel_arg_buffer is provided in the metadata of NDRange. Before the kernel function is executed, start.S will be executed first:

```
# 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
```

It is agreed that the printing information of the kernel is transmitted to the host through the print buffer. The address and size of the print buffer are provided in the metadata_buffer. After the running thread finishes printing, it sets the CSR_PRINT of the warp it belongs to. When the host polls that there is unprocessed information, it will take out the print buffer from the device side and reset CSR_PRINT.

Architecture description

The ABI on the hardware, the instruction set, the part of the register interface, and the description of the memory system to meet the programming needs of the OpenCL kernel.

Instruction set scope

RV32V

The actual selected instruction set range is: RV32 I M A zfinx zve32f V mainly supports independent data path instructions, and the original shuffle widen narrow gather reduction of RVV are not supported. The following table lists the scope of currently supported standard commands, and the changed commands have been declared.

	Support Status	***Command Changes***
RV32I	does not support ecall ebreak, supports SV39 virtual address	
RV32M F	Support RV32M zfinx zve32f	
RV32A	Support	
RV32V-Register State	Only supports LMUL=1 and 2	
RV32V-ConfigureSetting	Support computing vl, you can configure elements with different widths through this option	
RV32V-LoadsAndStores	Support vle32.v vlse32.v vluxei32.v memory access mode	The semantics of instructions such as vle8 are changed to "each thread writes to the vector register element position" instead of continuous writing
RV32V-IntergerArithmetic	Supports most int32 computing instructions	The semantics of vmv.x.s is changed to "each thread writes to the scalar register", instead of always writing to the vector register idx_0, and simultaneous writing by multiple threads is undefined behavior. Correctness is guaranteed by the programmer; vmv.s.x semantics changed to be consistent with vmv.v.x

	Support Status	***Command Changes***
RV32V-FixedPointArithmetic	Add int8 support, and add other types according to application requirements	
RV32V-FloatingPointArithmetic	Support most fp32 instructions, add fp64 fp16 support	
RV32V-ReductionOperations	Add it according to the application and compiler requirements, for example, when it needs to support work_group_reduce in OpenCL2.0	
RV32V-Mask	Support independent calculation and mask setting for each lane	The semantics of instructions such as vmsle are changed to "each thread writes to the vector register element position" instead of continuous writing
RV32V-Permutation	Not supported, consider adding it depending on application and compiler requirements	
RV32V-ExceptionHandling	Not supported, consider adding it depending on application and compiler requirements	

Custom Instructions

barrier: thread synchronization instruction

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

The barrier corresponds to the barrier(cl_mem_fence_flags flags) and work_group_barrier(cl_mem_fence_flags flags, [memory_scope scope]) functions of opencl to realize data synchronization between threads in the same workgroup. The default value of memory_scope is memory_scope_work_group. imm is 5bit, and the specific encoding is as follows:

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

After the _opencl_c_subgroups feature is turned on, it is changed to the barriersub command, corresponding to the case of memory_scope=subgroup. At this time, imm[4:3] is fixed at 0, and cl_mem_fence_flags is imm[2:0], which is consistent with the barrier command.

endprg: end of task instruction

```
endprg x0,x0,x0 # meet the end of the kernel
```

It needs to be explicitly inserted at the end of the kernel to indicate the end of the current warp execution. Can only be used without branches.

vbeq/join: thread divergence control instruction

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

Implicit SIMT-stack implements branch control. The end of each branch block needs to use join to indicate the address of the meeting point. The source operand of join defaults to 0. vbeq provides the vbne vblt vbge vbltu vbgeu version with reference to beq, and the instruction code is modified in the func3 section.

regext{i}: register expand instruction

```
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
```

Expand the number of available registers with a macro instruction that indicates that the register number (and immediate value) of the next instruction will be expanded. regexti only takes effect on the V-extension VI class OP-imm5 instruction, which is treated as imm11. Immediate value expansion is not supported for other immediate value instructions, and regext can be used for register expansion. The current version of the compiler supports register extension instructions as needed. For immediate data instructions other than user-defined instructions, 11-bit immediate data is used by default, that is, 64-bit long instructions are always composed of regexti + vi instructions. From the compiler's point of view, immediate instructions will skip the regext stage.

vlw.v/vsw.v: privatemem access instruction

Only used to access privatemem, instructions provided in the form of scalar memory access instructions, but will access vector addresses and write vector registers. For the convenience of compilers and programming, the address is a continuous address of 0-1k from the perspective of per-thread, and the offset is completed by the hardware according to thread_id and CSR_PDS.

```
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: vector memory access instruction with 12-bit immediate address

```
vlw12.v vd, imm12(vs1) # vd <- mem[vs1+imm12]
```

```
vsw12.v vs2, imm12(vs1) # mem[vs1+imm12] <- vs2
```

Vlw12 provides vlh12 vlb12 vlhu12 vlbu12 version with reference to lw, and vsw12 also has vsh12 version.

vadd12.vi: vector integer addition and subtraction instruction with 12-bit immediate

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

vftta.vv: floating point convolution instruction

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

vfexp.v: floating point exponent instructions

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

The complete command code and description are as follows：

31 25	24 20	19 15	14 12	11 7	6 0	assemble	description	type
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(in_stack) += sext(offset)	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	
off[12|10:5]	vs2	vs1	0 1 1	off[4:1|11]	1 0 1 1 0 1 1	join vs2, vs1, offset	分支汇合,vs1和vs2默认为零	
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			regexti x0,x0,imm	扩展vop.vi指令的寄存器和立即数。rs1和rd默认为零	
0 0 0 0 0 0 0		rs1	1 0 0	rd		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]				barrier x0,x0,imm	对应barrier(),imm提供memory_scope和	
0 0 0 0 0 1 1		imm[4:0]				barriersub x0,x0,imm	cl_mem_fence_flags	
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]		rs1	0 1 0	vd	0 1 0 1 0 1 1	vlw.v vd,offset(rs1)	vd<-mem[addr] addr=(rs1+imm)*num_thread_in_wg+thread_idx 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(rs1)		
			0 0 0			vlb.v vd,offset(rs1)		
			1 0 1			vlhu.v vd,offset(rs1)		
			1 0 0			vlbu.v vd,offset(rs1)		
1 imm[10:5]	vs2	rs1	0 1 0	imm[4:0]	0 1 0 1 0 1 1	vsw.v vs2,offset(rs1)	mem[addr]<-vs2 addr=(rs1+imm)*num_thread_in_wg+thread_idx funct3字段编码与sw sh sb等访存指令一致	
			0 0 1			vsh.v vs2,offset(rs1)		
			0 0 0			vsb.v vs2,offset(rs1)		

Register and ABI

Register settings

Number of architectural registers: 64 sGPRs, 256 vGPRs, and the element width is 32bit. 64bit data uses register pair. Number of physical registers: 256 sGPRs, 1024 vGPRs, the mapping from architectural registers to physical registers is realized by hardware. The compiler provides the usage of GPR (the actual usage number of vGPR and sGPR is a multiple of 4), and the hardware allocates more workgroups for simultaneous scheduling according to the actual usage.

> From the perspective of RVV, the vector register width vlen is fixed at num_thread*32bit, which is equivalent to SEW=32bit, ma, ta, LMUL=1 of the vsetvli instruction in hardware. From the perspective of SIMT programming, each thread has at most 256 vGPRs with a width of 32 bits, and the workgroup has 64 sGPRs. The entire workgroup only needs to do one operation, such as address calculation in kernel and non-kernel functions, use sGPR; if there is a branch, use vGPR, such as parameter passing of

non-kernel functions.

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							

Grouped registers are unrolled by the compiler to provide support for OpenCL vector types.

> Grouped registers require multi-cycle emission on hardware, and register dependencies are not easy to detect. It is much easier to complete this step on the compiler. In the future, we will consider providing grouped register operations for scalar load/store. Scalar memory access instructions have their own particularities: the access to continuous addresses is greatly improved, such as S_LOAD_DWORDX8 in GCN3.

Special registers

- x0：0 register
- x1：ra return PC register
- x2：sp stack pointer - localmem baseaddr
- x4：tp privatemem baseaddr

Stack space description

Since OpenCL does not allow the use of dynamic memory functions such as malloc in the Kernel, and there is no heap, the stack space can be increased upwards. tp is used to push the stack when the private registers of each thread are insufficient (that is, vGPR spill stack slots), sp is used to push public data on the stack, and the data that explicitly declares the __local tag in programming (that is, sGPR spill stack slots and localmem access , in fact both sGPR spill stack slots and local data will be part of localmem). The compiler provides the overall usage of localmem data (according to sGPR spill 1kB, combined with the size of local data, collectively as localmem_size), for the hardware to complete the workgroup allocation.

Parameters passing ABI

For the kernel function, a0 is the base address pointer of the parameter list, and the starting address of the memory set by the first clSetKernelArg is stored in a0 register, and the kernel loads parameters from this position by default. For non-kernel functions, use v0-v31 and stack pointer to pass parameters, and v0-v15 as

the return value.

customized CSR

Note: You can use a lowercase suffix in the assembler to indicate the corresponding CSR, for example, use tid instead of CSR_TID.

description	***name***	***addr***
Thread id with the smallest id in the warp has a value of CSR_WID*CSR_NUMT, and other thread ids can be calculated with vid.v.	CSR_TID	0x800
Total number of warps in the workgroup	CSR_NUMW	0x801
Total number of threads in a warp	CSR_NUMT	0x802
baseadd of the metadata buffer of the workgroup	CSR_KNL	0x803
Workgroup id corresponding to this warp in the SM	CSR_WGID	0x804
Warp id corresponding to this warp in the workgroup	CSR_WID	0x805
baseaddr of the local memory allocated by the workgroup, also is the base address of the xgpr spill stack of the warp	CSR_LDS	0x806
baseaddr of the private memory allocated by the workgroup, also is the base address of the thread's vgpr spill stack	CSR_PDS	0x807
x id of the workgroup in NDRange	CSR_GIDX	0x808
y id of the workgroup in NDRange	CSR_GIDY	0x809
z id of the workgroup in NDRange	CSR_GIDZ	0x80a
CSR used to interact with the host when printing to the print buffer	CSR_PRINT	0x80b

Memory system

Each SM of Ventus GPGPU has a separate L1 memory subsystem, including instruction cache, shared memory (local memory/scratchpad memory), data cache, and constant cache. Among them, the instruction cache is directly accessed by the front-end fetching stage, and other memories are accessed by instructions through the LSU. All SMs access the same L2 memory subsystem.

Address space

Currently designed according to the 32-bit address space. Access to privatemem requires the use of a special vlw.v instruction, which automatically calculates its address offset for each thread. From the perspective of the compiler, the privatemem space available to each thread is the continuous 0-1kB space starting from CSR_PDS, and the hardware will automatically convert it to facilitate the continuous memory access of warp. Localmem and globalmem are accessed using vle32.v vloexi32.v vlw32.v instructions. The two use the address field to distinguish, and the address smaller than local_mem_max is considered to access localmem. Localmem uses real addresses to access on-chip SRAM; addresses used by globalmem, privatemem, and print buffer are allocated and managed by the driver (physically, all three will be mapped to ddr).

Consistency and Coherency Features

The consistency of the data cache between SMs is maintained by the programmer explicitly through synchronization instructions (Consistency-Directed Coherence), and the hardware does not maintain consistency. Synchronous instructions correspond to specific instructions in the RISC-V A extension, and the hardware cache provides flush and invalidate functions. Consistency can be understood in Ventus as whether the order in which the execution results of the current SM memory access instructions are visible to other SMs is the same as the order in which the current SM executes these memory access instructions. In RVWMO (RISC-V Weak Memory Ordering), the coherence requirements of ordinary memory access operations are defined by PPO (Preserved Program Order) rules 1 and 2. For regular read and write operations, the microarchitecture of Ventus L1 data cache is shown in the following table:

	<	**same**	**address**	**>**	**<**	**different**	**address**	**>**
Order of operations type	R-R	W-W	W-R	R-W	R-R	W-W	W-R	R-W
Whether to keep the order (to other SM)	Y	Y	Y	Y	Y	N	N	Y
Order Preservation Requirements for RVWMO PPO	require	require	not require	require	not require	not require	not require	not require

*In this table, A-B means that the program sequence of the memory access operation A is earlier than that of the memory access operation B.

L1 intruction cache

The main features are as follows:

- (Not yet fixed) 2-way set associative, cache line size is blocksize (unit Byte), total capacity 64*blocksize;
- Maximum return width of each LSU access is 4B;
- Maximum width of each data read and write between L1-L2 is 128B (one cache line);
- Replacement strategy supports LRU and FIFO replacement strategies;
- Support invalidation of ongoing requests;
- (Not yet fixed) There are up to 128 outstanding L2 access requests at the same time; L2 access requests for the same cache line are supported to be merged, and the maximum number of merges is 8.

L1 data cache

The main features are as follows:

- (Not yet fixed) 2-way set associative, cache line size is blocksize (unit Byte), total capacity 64*blocksize;
- Generally, blocksize = num_thread;
- Virtual Address Index, Virtual Address Tag (VIVT);
- Maximum return width of each LSU access is blocksize;
- Maximum width of each data read and write between L1-L2 is blocksize (one cache line);
- Write strategy is write back - write without allocation;
- Replacement strategy supports LRU and FIFO replacement strategies;
- Supports invalidation and clearing operations on the entire data cache, and supports invalidation and clearing operations on a single cache line;
- (Not yet fixed) There are up to 128 outstanding L2 access requests at the same time; L2 access requests for the same cache line are supported to be merged, and the maximum number of merges is 8.

The instruction types supported by the data cache include:

- Basic instruction set I:
  - Memory access instructions: LOAD and STORE
  - Memory access sorting instruction: FENCE
- Atomic Instructions Extension A:
  - Atomic operation instructions: AMO
  - Reserved read/conditional write instructions: LR/SC
- Vector instruction extension V:
  - Vector read instructions: VL, VLS, VLX
  - Vector write instructions: VS, VSS, VSX
  - Custom vector memory access instructions

Share memory

The main features are as follows:

- (Not yet fixed) 2-way set associative, cache line size is blocksize (unit Byte), total capacity 64*blocksize;
- general, blocksize = num_thread;
- Maximum return width of each LSU access is blocksize;

Command types supported by share memory include:

- Basic instruction set I:
  - Memory access instructions: LOAD and STORE
- Atomic Instructions Extension A:
  - Atomic operation instructions: AMO
- Vector instruction extension V:
  - Vector read instructions: VL, VLS, VLX
  - Vector write instructions: VS, VSS, VSX
  - Custom vector memory access instructions

L2 cache

The main features are as follows:

- Multi-way groups are connected, the number of groups and the number of ways can be configured through CacheParameters, the cache line size and the smallest writable unit can be configured through InclusiveCacheMicroParameters, currently 128Bytes and 4Bytes respectively
- Physical Address Index, Physical Address Tag (PIPT)
- Write strategy is write-back-write-allocate
- Replacement strategy supports random, pseudo-LRU
- Supports invalidation and flushing of the entire data cache, supports invalidation and flushing of a single cache line
- With MSHR, support non-blocking memory access
- In addition to normal load and store operations, AMO atomic operations are supported, and LR and SC operations are supported through the Tilelink protocol
- Support stride prefetch
- L2cache sends data conforming to the AXI 4 protocol to the main memory through an AXI Adapter

Processor mode

Only machine mode is supported, and the CSR will be configured before the program starts. Memory management is done by the driver. Exception and interrupt handling are not currently supported.

Bus interface

Ventus contains an AXI4 master interface and an AXI4-Lite slave interface. L2Cache accesses off-chip memory through the AXI4 master interface. The host configures the workgroup through the AXI4-Lite interface.

Network on Chip

Several SMs form an SM_clusters, and the SM_clusters are connected to L2Cache banks through crossbars.

Microarchitecture (hardware perspective)

Task assignment and compilation related

CTA task assignment

At the hardware level, 32 threads form a warp, which is scheduled on the SM hardware as a whole. The warp of the same block can only run on the same SM, but the same SM can accommodate several warps from different blocks or even different grids. The task sent by the CPU to the GPU takes the workgroup as the basic unit and is received by the CTA scheduler. The CTA scheduler will allocate the block to the idle (that is, the remaining resource enough) SM. The CTA scheduler sends warp to SM one by one. The warp of the same workgroup will be allocated to the same SM. The low bit of warp_slot_id indicates the id of the warp in the current workgroup, and the high bit indicates the id of the workgroup itself. Correspondingly, SM will use this id to calculate the position of the current warp in the workgroup to which it belongs, and place the value in the CSR register for software use. The allocated localmem baseaddr needs to be read through the CSR, while the privatemem baseaddr and register base are implicitly mapped by the hardware. Since a warp has only one set of CSR, thread_id needs to be calculated with vid.v+CSR_TID.

Assembly Programming

1. get_global_id() is realized by vid.v + csrr tid three instructions.
2. The input parameters and memory access address need to be read from the CSR according to the preset parameter transfer method
3. Use of custom intructions:
    1. Predicate: While supporting the software control mask defined by rvv, we also support the use of custom instructions to start implicit hardware predicates. See the section on custom instructions for details.
    2. The endprg instruction needs to be explicitly used at the end of the warp (that is, the kernel).
    3. For thread synchronization in the same block, use the barrier instruction.

The rest of the behavior is consistent with rvv programming. For vector data that exceeds the length of a single set of hardware processing capabilities, the stripmining method defined in rvv is supported. By default, num_thread data is processed at a time. Different from the vector processor, it can be realized by software mask or SIMT-stack, or it can be split into more warps for scheduling when the block size allows.

> SIMT-stack Supplement: At present, from the perspective of software, single warp execution has exactly the same function as rvv mask. The advantage is that 1) when all threads have the same direction, the if branch or else branch can be skipped; 2) the number of accesses to the register file (v0) is reduced, and the bankconflict when obtaining operands is reduced; 3) fast nested branches are realized, and the current hardware supports `num_thread-1` layer nesting in the worst case; 4) Provide convenience for subsequent hardware implementation of multi-path IPDOM, independent thread scheduling, warp merging, etc.

Instruction Set Architecture

Combination of RVV and GPGPU

"Computer Architecture: A Quantitative Approach" mentioned that vector processing units and multi-threaded GPUs work very similarly at the SIMD level, and the lanes of vector processors are similar to the threads of multi-threaded SIMD. The difference is that the GPU usually has more hardware units, and the chime is shorter. The vector processor hides the delay through deeply pipelined access, and the GPU hides the delay through simultaneous multi-warp switching. Therefore, in terms of vector-level operations, RVV is sufficient to cover the operations in GPGPU. In addition, AMD and NV (after turing) provide a scalar ALU, which is also borrowed from the vector processor. Therefore, on the basis of RVV, adding custom branch divergence control instructions (in fact, it can also be implemented by using RVV's own mask), thread synchronization instructions, and thread control instructions can realize the function of GPGPU. In order to maximize the compatibility with the RVV open source toolchain, we support most of the commands in RVV. A few unsupported instructions include: 1. Instructions such as shuffle that involve data exchange between threads. In GPGPU, threads are usually operated independently, and data dependencies need to be explicitly operated with atomic or barrier. 2. Instructions that change the length and width of vector registers, It will hardly be touched in GPGPU (a few vector or quantization related functions will require similar functions) 3. 64bit related instructions will be supported in later versions.

Adding warp-level parallelism on the basis of RVV's stripmining may be able to cut vector/SIMT instructions, explore space partition and scheduling on a better scale.

Register design

The maximum number of warps that can be carried on a single SM at the same time is num_warp, and each warp consists of num_thread threads. Each warp has its own set of registers. The width of each scalar register is 32bit, and the width of each vector register is 32*num_thread, which are private to each thread. The physical register file adopts a unified method and is dynamically allocated according to the actual usage of each warp. Although the form and meaning of the instructions used are the same, but different from rvv, the GPGPU we currently implement does not support the change of the length and number of vector registers (the length is fixed at 32bit, and the number is fixed at num_thread), so for the vsetl series instructions, only the returning remaining number of elements is valid.

Address mapping

Use the address range to distinguish between localmem and globalmem, where localmem is managed by CTA-scheduler, and globalmem, privatemem, and constantmem are allocated and managed by the driver.

> The physical address space in the GPU includes loaclmem and globalmem. In early versions of cuda and OpenCL programming, address attributes need to be explicitly declared. In PTX, each address has an attribute to declare its type. At present, Ventus has not implemented MMU yet, all SMs share a 4GB global address space, and the fields with an address within the range of 0-128kB will be mapped to their respective sharedmem inside the SM, and the space from global_baseaddr to 4G will be mapped to the same ddr on the same address (that is, this part uses a physical address). Coherence is explicitly managed by software in the GPU, implemented through flush and invalidate, corresponding to the fence command.

Instruction set scope

Currently supported instructions in RVV include the following types:

1. Calculations, including integer operations (addition and subtraction, comparison, shift, bit operations, multiplication, multiplication and addition, division), single-precision floating-point operations (addition and subtraction, multiplication, multiplication and addition, division, integer conversion, comparison) , support execution with mask
2. Memory access class, including three memory access modes, and byte-level read and write, support execution with mask
3. Mask class, includes comparison and logic operations, but does not include intructions involving inter-thread communication such as vmsbf, and does not support operations such as gather The instructions that change the vector bit width are not supported yet, but the vsetl series instructions can return vl for stripmining. RV32I supports instructions except ecall ebreak, and M F supports 32bit related instructions.

Currently supported custom instructions include:

1. Predicate: While supporting the software control mask defined by rvv, it also supports the use of custom instructions to start the implicit hardware predicate. The implicit hardware predicate started by vbeq will take effect for subsequent instructions by default until a new vbeq or join is triggered. The way to start it is to use the custom vbeq series thread branch instructions, which will start a split, calculate the mask situation of the current branch, and push the mask corresponding to the else into the SIMT-stack, and then execute the if section with the mask, and wait for when the join is encountered at the end of the if section, the else section and its corresponding mask are popped out of the stack. After the execution of the if section is completed, the mask is merged and restored, and the branch ends. This process can be nested, and the worst-case depth of stack is equal to the number of threads in a single warp 32 (if you always choose the most direction to push the stack instead of the default if, then the stack depth only needs to be 5). In addition, if the branch calculates that all branches are taken, the stack will not be pushed and another branch will be skipped. The for loop with branch divergence can also be implemented with this mechanism.
2. barrier: This instruction can be used to achieve synchronization between warps. Each warp that encounters this barrier command will wait until all unfinished warps in the workgroup encounter this instruction before continuing again.
3. endprg: The endprg instruction needs to be explicitly used at the end of the warp operation.
4. regext{i}: Expand the number of registers and immediate data width used for the next instruction.
5. For other custom instructions to improve code efficiency, please refer to the section "Architecture description - Instruction Set Scope - Custom Instructions"

Drive interface

Follow the interface information support provided by the CTA scheduler, and configure the number of registers used, localmem baseaddr, warp id, etc.

Microarchitecture design

It is generally divided into front-end and back-end. The front-end includes instruction fetching, decoding, instruction buffer, register file, issue, scoreboard, warp scheduling, and the back-end includes ALU, vALU, vFPU, LSU, SFU, CSR, SIMT-stack , warp control, etc. The handshake mechanism is used to transmit signals between the modules involved in the register connection.

Warp scheduling

The main functions of the warp scheduler include:

1. Receive the warp information provided by the CTA scheduler, allocate the hardware unit to which the warp belongs and preset the CSR register value, activate the warp and mark the block information it belongs to. After the warp is executed, the information is returned to the CTA scheduler and the corresponding hardware is released.
2. Receive the barrier instruction information sent by the pipeline, and lock the specified warp until all active warps of the block to which it belongs reach the barrier.
3. Select the warp sent to the icache, usually using a greedy strategy, but when the icache misses or the ibuffer is full, the PC of the warp will fall back and switch to the next warp launch.
4. Select the warp to be sent to the execution unit. Usually, a polling strategy is used to select the instructions whose current instruction buffer is valid, the scoreboard does not show conflicts, and the execution unit is idle. Switching warps takes only one cycle.

Fetch

Each warp stores its own pc, and the pc selected to be sent to the icache will be +4, and the rest will retain the original value, and will be replaced with the target address when a jump is encountered.

Decode

The instruction hit by the icache is decoded, and the decoder converts the corresponding control signal according to the instruction content, and sends it into the corresponding instruction buffer.

Instruction buffer

The instruction buffer is a series of FIFOs, each warp has its own ibuffer, which receives decoded input and waits to be selected for issue.

Oprand collector

The operand collector will receive the request from instruction buffer, and access to the register file to obtain data through the crossbar according to the required data type. After obtaining the required data, enter the state to be issued. The register file is designed with the unified scheme: on the hardware, there are 1024 vector registers and 512 scalar registers in a single SM, and a single warp can use up to 256 vector registers and 64 scalar registers, and the allocation of physical registers is completed by the hardware. The compiler specifies the number of registers used (a multiple of 4). The fewer registers a single warp uses, the more warps can be allocated on the hardware to run simultaneously. The register file hardware is divided into 4-banks, and each bank has a read port and a write port. The current version of scalar and vector registers does not support simultaneous access, which will be modified later.

Issue

The issue arbitration is performed by the warp scheduler, and the selected control signal, together with the source operand, is sent to the corresponding operation unit for execution according to the type of operation unit it requires.

Scoreboard

Each warp has its own scoreboard. When an instruction is successfully issued, the written register will be marked by the scoreboard. If the next instruction reads and writes the marked register, it is not allowed to issue until the instruction execution is completed and the board releases the corresponding register. Branches, thread divergences, jumps, and barrier instructions will also be locked, and will only be released when these instructions complete the judgment and can continue to execute sequentially. During the execution of the barrier, all pipelines (parts belonging to the warp) except ibuffer will be cleared at the same time; if a jump occurs, all pipelines (parts belonging to the warp) will be cleared at the same time, and then the scoreboard will be unlocked.

Writeback

Each arithmetic unit has a FIFO at the output stage, and the results waiting to be written back to the register are temporarily stored in it, and are selected by the arbiter and written back to the register. Writing back to scalar registers is a completely separate data path from vector registers.

ALU

Scalar operations are performed in the ALU, including data shared between warps, and jump control.

vALU

Copies of a single ALU, and it is the core integer computing unit for warp's multi-threaded lanes to perform operations. Typical operation consumes 1cycle. Support folding to num_lane.

vFPU

The core floating-point operation unit, which is used for multi-threaded lanes of warp to perform operations, and scalar floating-point operations are also performed here. Fully pipelined design, multiplication and multiplication-add have multiplexed data paths. Typical multiply-add consumes 5 cycles, multiplication consumes 3 cycles, and supports folding into num_lanes.

MUL

Multiplication operation unit is used for warp multi-threaded lanes to perform operations. Using wallace tree structure, full pipeline design, typical multiplication consumes 2cycles, and supports folding into num_lanes.

LSU

The core of memory access, it will send read and write requests to sharedmem or dcache according to the address range (then L1 dcache accesses L2 cache, and then accesses ddr, which is globalmemory). There is a structure in the form of MSHR in the LSU, which can store and record multiple LSU requests at one time, and also collect the data returned by dcache and sharedmem, and return it to the pipeline after collecting all the data. The calculation of vector addresses in strided and arbitrary modes will be completed in LSU, and combined access will be performed in units of cachelines according to the address range. In the most ideal situation (meaning that the addresses are continuous and the cacheline is aligned), the result required for a single warp can be fetched in one memory access. Bank conflict is handled by sharedmem and dcache. The existing memory access request information is also recorded in the LSU to implement the fence instruction.

When this command is encountered, all memory access requests of the warp to which it belongs will be processed (read request returns data, write request returns response) and then send a new memory access request.

CSR

When the warp starts, the corresponding CSR will set some values required by the application, including thread id. vsetl is also calculated here. The rest are consistent with the functions of the riscv cpu CSR.

SFU

Different from functions such as sin cos provided in PTX, the current SFU only supports integer division remainder, floating-point division, and floating-point square root functions defined by rv. The calculation itself requires multiple cycles to complete, and the number of calculation units in the SFU is less than the number of lanes, so if there are many unmasked threads, the chime will be longer.

TC

tensorcore， fully pipelined and supports tensor calculations in specific formats.

warp control

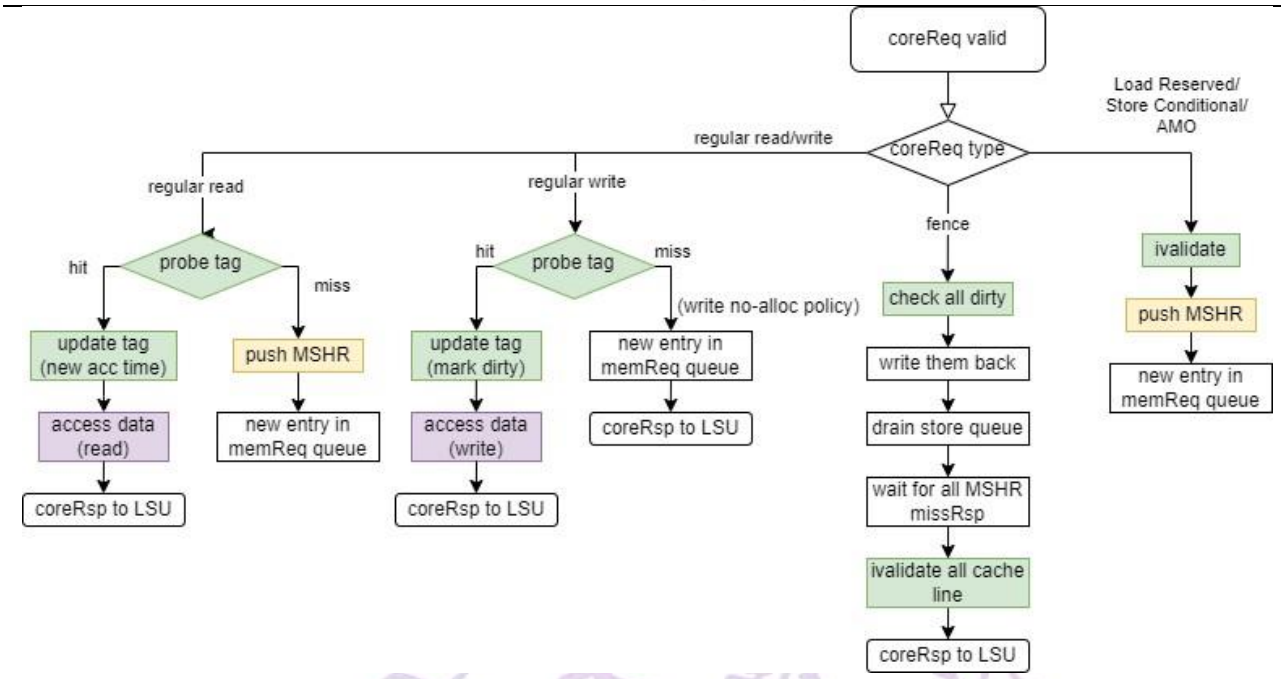
barrier and endprg will be sent to the warp scheduler to process. Support for dynamic parallelism will be considered in the future, and warpgen instruction will be added.

SIMT-stack

The main functions of the SIMT stack are as follows: maintain the control flow of branch nesting to ensure the correctness of program operation; when no branch divergence actually occurs, skip the execution of unnecessary program segments. The implicit mask set by SIMT-stack will remain in effect during the execution of the warp until it is modified by other branch management operation. This mask and the mask in the form of rvv software can be superimposed to take effect. There are 7 custom extended instruction sets related to branch management as shown in the above table, of which 1-6 are branch instructions. Taking the vbeq instruction as an example, the function to be completed is: take the source operand vs2 and vs1, and the valu compares the elements in the two vector registers one by one. For the i-th element, if vs2[i]= vs1[i], the calculation result out[i] is 1, and the final value output result is the else path mask corresponding to the branch instruction, and the decode module will send the branch occurrence flag, else path starting PC (PC branch) to the branch management module.

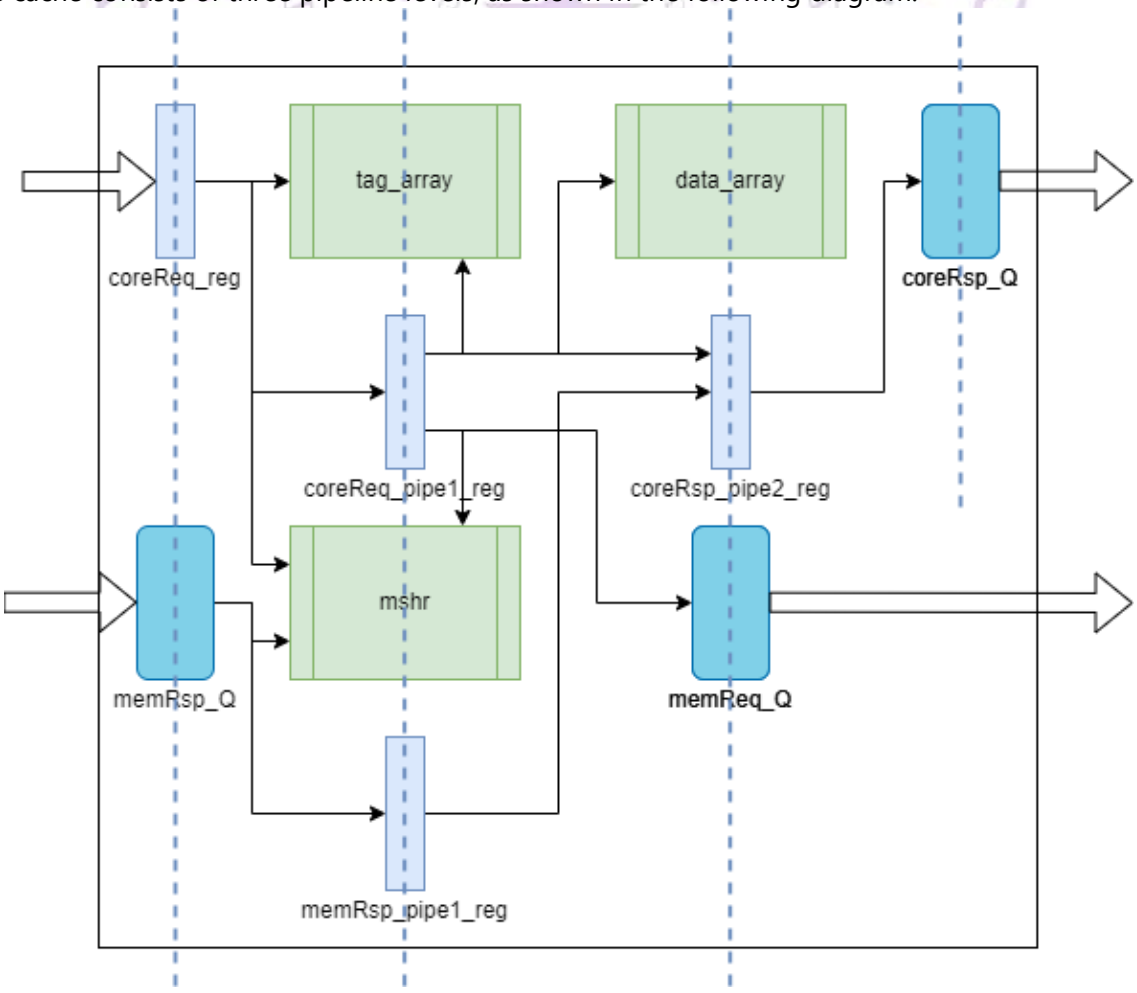
DCache

The entire cache is divided into two sets of interfaces coreReq and coreRsp for LSU access, and two sets of interfaces memReq and memRsp for access to higher-level cache (L2). Among them, coreRsp is directly connected to the write-back stage of the SM core pipeline. The request types supported by coreReq include: ordinary read requests (scalar/vector), ordinary write requests (scalar/vector), fence requests, reserved read requests, conditional write requests, and atomic operation requests. The processing operations of these requests in the cache are as follows:



Most of these requests correspond to instructions one-to-one. The AMO directive is an exception. AMO instructions allow .aq and/or .rl identifiers (acquire and release), and the future RV standard instruction set/extended instruction set may also add support for ordinary memory access instructions to carry this identifier. For a memory access instruction carrying such an identifier, the LSU will split it into a memory access request without an identifier and a corresponding fence request.

The entire cache consists of three pipeline levels, as shown in the following diagram:



For the coreReq path:

S0: Send tag and mshr SRAM access request.

S1: Determine the follow-up operation according to the request type and tag return value: push memReq_Q or/and issue a data access request.

S2: Push coreRsp_Q on the stack.

For the memRsp path:

S0: Issue mshr SRAM access request.

S1: Update tag and organize coreRsp according to the request type and mshr return value.

S2: Redundant pipeline stages introduced to avoid pipeline conflicts.

The overall microarchitecture design is shown in the figure.