CnuasIR, Datasheet

Virtual instruction set architecture for the CnuasGPU accelerator

Document DS-CNU-016 • **Revision** D • **Issued** 06 August 2026 • **Status** Preview, executes on the interpreting backend

Item	Value
Part	CnuasIR
Type	Virtual instruction set architecture
Models	NVIDIA PTX
Status	Preview, object format, build tools and an interpreting backend implemented
Repo	PacketFive/CnuasGPU

Preview datasheet

The object format, the build side tools and an interpreting compute backend are implemented. A CnuasIR object can be packed from assembled machine code, validated against the v0.1 subset, disassembled, loaded with `cnuasModuleLoad` and run with `cnuasLaunchKernel`. What is not implemented is a compiler that emits CnuasIR, and the tensor, synchronisation and cross lane instruction classes in section 2, which are specified but not encoded. The rest of this datasheet records the specified design so that interfaces can be reviewed early.

0. Implementation status

Part	Status	Where
Object header format	Frozen and implemented	`lib/cnuasir/cnuasir.h`
Header parser and validator	Implemented, 26 checks	`lib/cnuasir/cnuasir_object.c`
Packer, validator, disassembler	Implemented, 230 checks	`tools/cnuasir-*.py`
Interpreting compute backend	Implemented, 242 checks	`lib/compute/cnuasir/`
Kernel launch API	Implemented, 17 checks	`lib/libcnuasrt/cnuasrt.c`
Compiler that emits it	Not implemented	CnuasCC

0.1 Build tools

Command	Does
`tools/cnuasir-pack.py`	Wraps a raw `.text` image in a CnuasIR header, deriving the extension flags from the code it finds
`tools/cnuasir-validate.py`	Rejects a header or an instruction outside the v0.1 subset
`tools/cnuasir-dis.py`	Prints the header, then the instruction stream

The three share one decoder, `tools/cnuasir/isa.py`, so a program cannot disassemble cleanly and then fail validation. The v0.1 subset is scalar RV64IMF plus the SEW=32, LMUL=1 slice of RVV 1.0; the tensor,

synchronisation and cross lane classes in section 2 are specified but not yet encoded, and the tools refuse them.

Roadmap 8.3a completed the scalar half. The subset originally carried only the instructions the hand written kernels used, which left it unable to express ordinary code: there was no `fsub.s`, so a subtraction between floats had no lowering, and no logical, shift or compare instructions, so neither a bitmask nor a predicate could be computed. The gaps were filled before starting the compiler rather than during it. The additions are the RV64IMF logical, shift, compare, unsigned branch, unsigned and remainder division, and floating point subtract, divide, square root, sign injection, minimum, maximum and comparison forms. Each is checked against `riscv64-unknown-elf-as`, so none is an invention.

Still deliberately absent: the RV64 word forms, doubleword memory, pc relative addressing, the fused scalar forms, and any move between the integer and floating point register files. That last one is what makes a float reach a kernel through memory, and it is why the launch API has no floating point argument kind.

Validation happens at build time because the planned interpreter decodes only the subset. Checking early turns a run time `cnuasErrorNotSupported` into a build failure, which is the difference between a kernel that fails on the accelerator and one that never ships.

Correctness is anchored to an external oracle rather than to the author's reading of the specification: `tools/test/test_cnuasir_tools.py` assembles a corpus with `riscv64-unknown-elf-as` and requires the decoder to agree with `objdump` instruction for instruction. It also parses `lib/cnuasir/cnuasir.h` and compares every shared constant, so the Python restatement of the object format cannot drift from the normative C header. The `objdump` group skips when no RISC-V toolchain is installed; the rest always runs.

The magic number is `'C','N','I','R'`. It read `'H','I','I','R'` while the project was named HiCAIN; no object using it was ever produced, and the parser rejects it.

0.2 Execution

`lib/compute/cnuasir/` is a compute backend that decodes and executes the v0.1 subset in plain C. It sets `CNUAS_CCAP_CNUASIR_EXEC` and implements `execute_cnuasir`, and it also provides the four numerical primitives, written as CnuasIR programs rather than as C.

Item	Value
`VLEN`	512 bits, so `VLMAX` is 16 at SEW=32, LMUL=1
Step budget	100000000 instructions, then `CNUASIR_EXEC_ERR_STEP_LIMIT`
Termination	`x1` starts at `text_size`, so a plain `ret` ends the program
Selection	Never wins the automatic race; set `CNUAS_COMPUTE_BACKEND=cnuasir`
Throughput	Several hundred times slower than the AVX2 backend

The backend stays out of the automatic backend race deliberately. It exists to be correct and readable, not fast, and a machine that picked it by accident would look broken rather than slow.

The shipped kernels are built at run time by the encoders in `lib/compute/cnuasir/encode.h`, not assembled from `.S` at build time, so what the backend can do does not depend on whether a RISC-V toolchain is installed on the build machine. The encoders are held to the assembler regardless: `lib/test/cnuasir_interp_smoke` checks all 44 of them against `riscv64-unknown-elf-as` when one is present.

Registers a kernel has not defined are poisoned before it runs, with `0xDEADBEEFDEADBEEF` in the integer file and a quiet NaN in the float and vector files, so a kernel that reads an undefined register gives an obviously wrong answer instead of one that only holds on a zeroed machine.

Memory access resolves to host addresses without a bounds check. The check that matters is in the runtime: `cnuasLaunchKernel` translates each device pointer against its declared extent, and that is the only place a wrong length is caught. `execute_cnuasir` is neither safer nor less safe than calling a function pointer the caller supplied.

0.3 Launch API

Call	Does
`cnuasModuleLoad`	Validates the header and copies the image, so the module does not alias the caller's buffer
`cnuasModuleUnload`	Frees it
`cnuasLaunchKernel`	Marshals arguments and calls the active backend's `execute_cnuasir`

An argument is either a device pointer, which is translated and bounds checked against the extent the caller declares, or a scalar, which is passed through unchanged. There is no floating point kind: the v0.1 subset has no move between the integer and floating point register files, so a kernel cannot get a float out of an argument register and has to load one from memory. Passing a float means putting it in a device buffer, which is what the shipped kernels do with alpha and beta.

The argument count in the module header must equal the count the caller passes. A mismatch is refused rather than padded, because the two sides disagreeing about a signature is how a kernel ends up writing outside the buffer it was given. Both the runtime and the backend check this independently.

A launch through a backend with no `execute_cnuasir` fails with `cnuasErrorNotSupported`.

1. Overview

CnuasIR is the intended device instruction set of CnuasGPU. It occupies the position PTX occupies for NVIDIA parts: the stable, versioned form that CnuasCC emits and that the runtime loads and executes.

The design deliberately builds on the RISC-V Vector extension rather than inventing a vector instruction set. That choice means the existing LLVM RISC-V Vector backend does most of the work, and only the tensor, synchronisation and cross lane operations are Cnuas specific.

2. Instruction classes

Class	Source	Examples
Scalar integer	RV64I	`add`, `sub`, `xor`, `lw`, `sw`
Floating point	RISC-V F, D and Q extensions	`fadd`, `fmul`, `fsqrt`
Vector	RISC-V Vector 1.0	`vadd.vv`, `vmul.vv`, `vfmacc.vv`, `vle32.v`
Tensor	Cnuas specific	`tmma`, `tload`, `tstore`, `tquant`, `tdequant`
Synchronisation	Cnuas specific	`bar.sync`, `bar.warp`, `membar.gl`
Cross lane	Cnuas specific	`shfl`, `vote.any`, `vote.all`

3. Execution model

Parameter	Specified value
Model	Single instruction, multiple threads over a host vector unit
Warp size	32 threads, matching the CUDA model
Divergence	Per lane active mask with a reconvergence stack
Tensor operands	Tile registers, for example 16 by 16 half precision
Tensor operation	One `tmma` performs a complete tile matrix multiply and accumulate
Host mapping	AVX2, AVX-512, ARM NEON and SVE, or RISC-V Vector, depending on host

4. Dependencies

Requirement	Position
Instruction set freeze at version 1	Not done, and the blocking item
LLVM backend emitting CnuasIR	Not started, see CnuasCC
Backend capability bit `CNUAS_CCAP_CNUASIR_EXEC`	Defined in the compute backend ABI, set by no backend
Object and container format	Not specified

The capability bit already exists in `cnuas_compute_ops.h`, so a backend that gains an interpreter can advertise it without an ABI change. See CnuasDev section 5.

5. Open design questions

These are genuinely undecided. The questions in section 11 of `src/cnuasgpu/docs/CnuasIR.md` are not; that section records decisions already taken and their reasoning.

- Whether the instruction set is frozen at version 1 or versioned per device, which decides whether an old binary must keep running on a new accelerator.
- How tile registers are named and saved across a context switch.
- Whether the container is a distinct object format or an ELF section embedded in the host binary.

6. Revision history

Revision	Notes
A	First publication as a preview. Content taken from the CnuasGPU design document, section 4.3.
B	Records roadmap 6.4b. New section 0.1 covering the packer, validator and disassembler, the shared subset decoder, and the cross check against `objdump`. Status moves from object format only to object format plus build tools.
C	Records roadmap 6.4c. New sections 0.2 and 0.3 covering the interpreting compute backend, its machine parameters, its trust model, and the module and launch API. Status moves from build tools only to executes.
D	Records roadmap 8.3a, the scalar subset completion. Section 0.1 gains the rationale and the list of additions, and the check counts move to 242 and 230.