CnuasCC, Datasheet

Device compiler toolchain for CnuasGPU kernels

Document DS-CNU-015 • **Revision** C • **Issued** 15 August 2026 • **Status** Preview, front end implemented

Item	Value
Part	`cnuascc`
Type	Compiler driver and toolchain
Models	NVIDIA `nvcc`
Status	Preview, front end implemented, no code generation
Repo	PacketFive/CnuasGPU

Preview datasheet

`cnuascc` can read a device source file, check it and report what is wrong with it. It cannot yet produce an object, so there is still no way to compile and launch a user written kernel. The functions the accelerator can execute today are the fixed set listed in CnuasRT section 4.5. Section 3 below records what is implemented and section 4 what the v0.1 route leaves out.

1. Overview

`cnuascc` is the intended device compiler for CnuasGPU. It splits a single source file into host and device parts, compiles the device part to CnuasIR, compiles the host part with a standard pipeline, and links the two so that one binary carries both.

It is the largest remaining piece of the accelerator stack. The runtime can load and launch a CnuasIR object, the interpreting backend can run one, and the compiler can now check a source file, but until it can emit an object every kernel still has to be written as an instruction stream by hand. That is what stands between the tree and the numerical libraries, a profiler, and user written kernels.

2. Eventual pipeline

Stage	Function
Frontend	Clang, with `--cnuascc-host` and `--cnuascc-device` selection flags
Splitter	Separates host and device code and emits two object files
Device backend	LLVM RISC-V Vector plus the Cnuas tensor opcodes, producing a CnuasIR object
Host backend	Standard host LLVM pipeline, linking the Cnuas runtime stubs
Linker	Embeds the CnuasIR bundle into the host executable, which the runtime extracts at load time

This is the target for version 1. It is not what version 0.1 does, and section 3 says why.

3. What is implemented, and how it differs

Roadmap 8.3b built a self contained front end rather than the Clang stage above: a lexer, a recursive descent parser, and a type checker, written in Python and living in `tools/cnuascc/` in the CnuasGPU repository. It reads the device language described in `src/cnuasgpu/docs/CnuasCC_Language.md`, a subset of C cut down to what the CnuasIR v0.1 instruction set can express.

Two reasons for the divergence, both about cost rather than taste.

1. Clang accepts all of C. Since CnuasIR v0.1 can express only a fraction of it, a Clang based front end would accept a program and then fail during code generation, a long way from the cause and in terms the author of the source did not use. Cutting the grammar instead puts every rejection on a token, with a note saying what to write.
2. An LLVM backend is a larger piece of work than a v0.1 instruction set justifies, and the instruction set is not frozen yet. Freezing it against a hand written code generator is cheaper than freezing it against an LLVM target that then has to be reworked.

The intent is that the Clang and LLVM pipeline replaces this once CnuasIR reaches version 1. The device language is specified so that a program written against version 0.1 keeps compiling when it does.

Stage	File	Status
Lexer	`tools/cnuascc/lexer.py`	Implemented
Parser	`tools/cnuascc/parser.py`	Implemented
Type checker and name resolution	`tools/cnuascc/sema.py`	Implemented
Driver	`tools/cnuascc-compile.py`, installed as `cnuascc`	Implemented, writes a CnuasIR object
Code generator	`tools/cnuascc/codegen.py`, `tools/cnuascc/encode.py`	Implemented, scalar subset
Host and device splitting	not written	Requires the Clang pipeline

Validation is `tools/test/test_cnuascc.py`, 126 checks, run by `make -C tools check`. The tests are weighted towards programs that must be rejected and towards the wording of the rejection, because a front end for a restricted language is mostly a machine for saying no. Code generation is checked three ways: every emitted word must decode inside the CnuasIR v0.1 subset, the lowerings the subset cannot express must be refused with a diagnostic rather than emitted wrongly, and compiled kernels are run on the interpreter and compared against the same arithmetic evaluated outside the compiler.

4. Language restrictions worth knowing

The full list is in the language reference. These three are the ones that change how a kernel is written.

Restriction	Reason
A kernel may not take a scalar `float` argument; pass a `float *` and read element zero	CnuasIR v0.1 has no move between the integer and floating point register files, so a float in an argument register could not be got out of it. The launch API has no floating point argument kind for the same reason
A pointer may only be indexed, never assigned, compared, offset or tested	A pointer is a buffer argument resolved by the runtime before the kernel starts, which is what makes the bounds check at launch meaningful
No function calls	There is no call ABI in v0.1, so a kernel is a leaf

5. Companion components

Component	Function	Status
`libcnuasccrt`	Device side runtime, formatted output, mathematics and synchronisation	Not implemented

Component	Function	Status
CnuasIR	Target instruction set	Preview
Runtime module and launch API	`cnuasModuleLoad`, `cnuasModuleUnload` and `cnuasLaunchKernel`	Implemented, see CnuasRT; the triple angle bracket launch syntax is not

6. What this unblocks

Blocked component	Datasheet
CnuasDNN, CnuasFFT, CnuasSPARSE, CnuasSOLVER, and CnuasBLAS beyond its first version	Math Libraries
CnuasCCL beyond its first version	CnuasCCL
`cnuas-prof` kernel profiler	Management Tools
User written kernels of any kind	This datasheet

A first CnuasBLAS and a first CnuasCCL are not on this list. Both can be built over kernels and a transport that already ship, so they are held up by nothing here.

7. Dependencies

Remaining for a usable v0.1 compiler:

1. Vector code generation. The compiler lowers to the scalar subset only, so a kernel that needs the vector unit is still written by hand.
2. Doubleword memory in CnuasIR, without which an `int *` cannot be subscripted, since a CnuasCC `int` is 64 bits.

Remaining for the pipeline in section 2:

1. Freeze the CnuasIR instruction set at version 1.
2. Land an LLVM backend that emits it.
3. Define the bundle container and the loader path in the runtime.

The runtime module and launch entry points, which were on this list, shipped in roadmap 6.4c.

8. Revision history

Revision	Notes
A	First publication as a preview. Pipeline taken from the CnuasGPU design document, section 7.4.
B	Records roadmap 6.4c, in which the runtime module and launch API ships as the companion component.
C	Records roadmap 8.3b. The front end is implemented, as a self contained parser and type checker rather than the Clang stage of section 2; the reasons are in section 3. Adds the language restrictions in section 4.

PacketFive, Packet Five Networks Ltd., Dublin, Ireland.