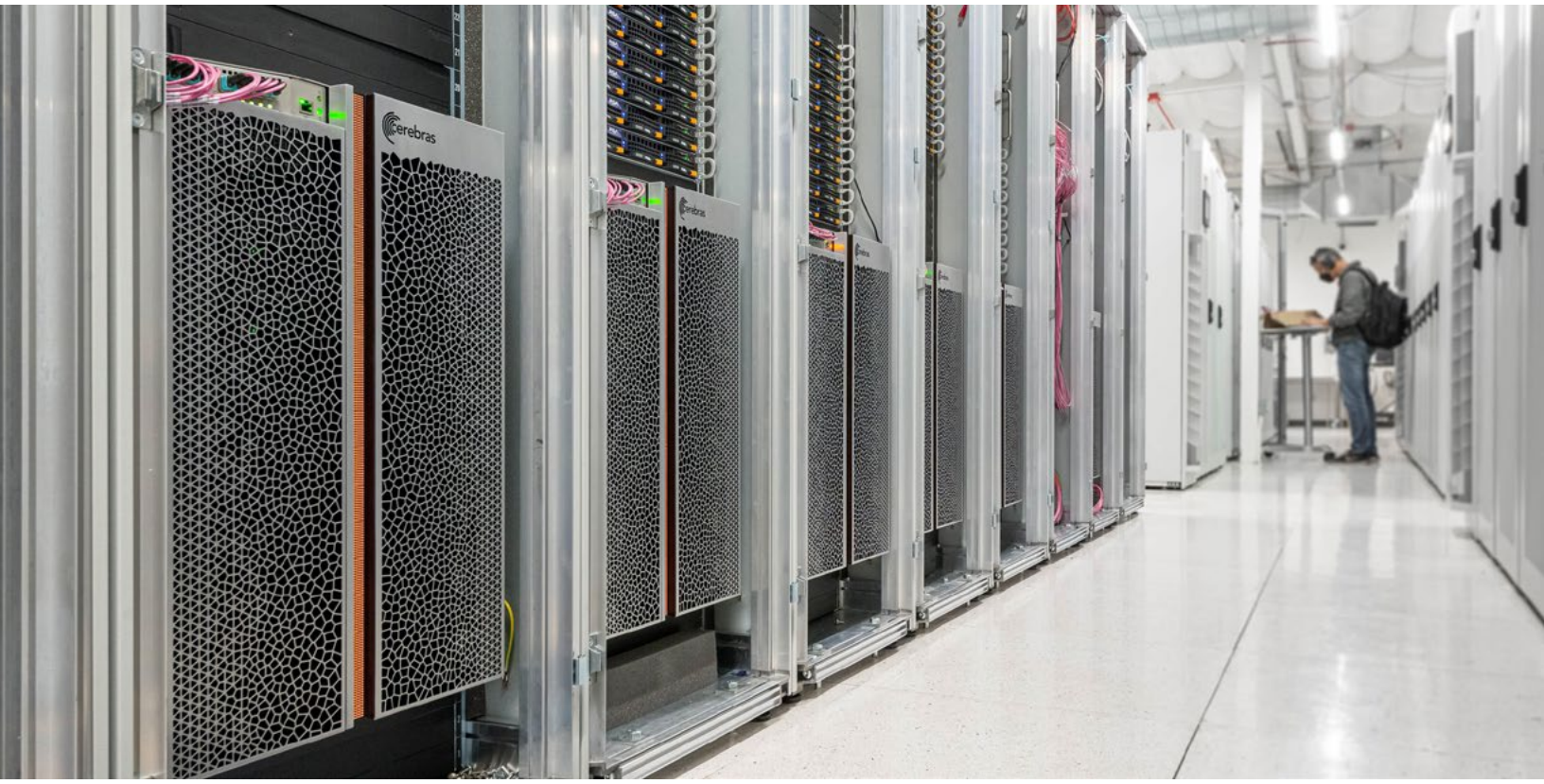
The Cerebras Software Development Kit: A Technical Overview

Justin Selig, Cerebras SDK Product Manager

Abstract

The Cerebras CS-2 system is the world's largest and most powerful HPC and AI accelerator. It excels on workloads that are constrained by the limitations of conventional computers: poor parallel efficiency, bandwidth constraints in the face of heavy local or global communication, long memory access latency, and imbalances between various compute and communication resources. Sparse linear algebra and tensor workloads, stencil-based partial differential equation (PDE) solvers, n-body problems, and signal processing algorithms such as the Fast Fourier Transform, are workloads where Cerebras systems have demonstrated superior performance compared to conventional scale-out solutions. We expect to find many more use cases that are well-suited for wafer-scale computing using the Cerebras architecture.

Cerebras has introduced a new software development kit (SDK) which allows anyone to take advantage of the strengths of the CS-2 system. Developers can use the Cerebras SDK to create custom kernels for their standalone applications or modify the kernel libraries provided for their unique use cases. The SDK enables developers to harness the power of wafer-scale computing with the tools and software used by the Cerebras development team.



Removing Data Access and Scaling Constraints

A distributed problem

There is a growing realization in the high-performance computing (HPC) field that the traditional scale-out approach of hundreds or thousands of identical compute nodes loaded up with GPUs or other accelerators has limitations. The efficiency of algorithms tends to decrease as they are split, or "sharded" across many nodes, because moving data between those nodes is such a slow process. Writing code for massively parallel systems is a difficult and time consuming.

A far better solution is to scale up the power of individual compute nodes, that is to add very powerful accelerators to the network as complete, independent compute nodes that are capable of fitting problems of interest within a single chip. The Cerebras CS-2 system is the world's most powerful network-attached accelerator.

Why is the Cerebras system so fast?

At the heart of the Cerebras CS-2 is the second-generation Wafer-Scale Engine (WSE-2). The specs are impressive: the WSE-2 is a massive parallel processor built from a single 300mm wafer. It offers 850,000 cores, optimized for sparse linear algebra with support for FP16, FP32, and INT16 data types. It contains 40GB of onboard SRAM divided between its cores, with 220Pb/s of interconnect bandwidth and 20PB/s of memory bandwidth, ensuring that each core can access its local memory in a single clock cycle.

All of these numbers are orders of magnitude greater than those of conventional architectures. But how do they translate to the kind of real-world performance that makes HPC developers so excited?

First, wafer-scale integration collapses time by collapsing space. By putting a roomful of cores on one chip, data spends less time in flight. The wires are short – millimeters or less, instead of tens of meters. And moreover, we eliminate all the hardware and layers of software in between: no DDR, no PCIe, no InfiniBand or Ethernet.

Second, the architecture was designed expressly to excel at workloads that demand high-speed memory access and data transfer:

We don't waste time accessing data – the entirety of each core's memory is one clock cycle away.

We don't waste time waiting for data to crawl between nodes – even the fastest external network is a thousand times slower than the single clock cycle it takes to move from core to core on the WSE-2.

We don't have to split data objects, such as graphs and arrays, across nodes at the wrong end of a tiny, long, I/O pipe.

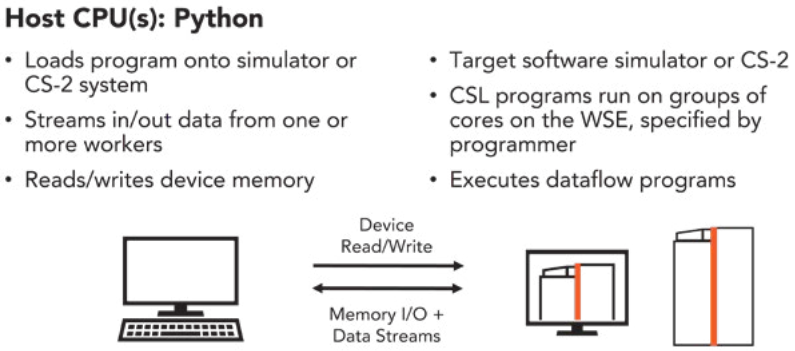
These attributes allow unequaled acceleration for sparse linear algebra, the math underlying neural networks and some HPC algorithms, as well as dense linear algebra and spectral algorithms.

In the remainder of this paper, we'll take a detailed look at the programming model and introduce the key concepts that enable developers to harness the power of the Cerebras architecture.

Programming Model

While the CS-2 system already supports frameworks like PyTorch and TensorFlow, the Cerebras SDK allows programmers to write lower-level code that targets the WSE's microarchitecture directly using a domain-specific programming language called the Cerebras Software Language, or CSL.

In CSL, the cores on the WSE-2 are referred to as Processing Elements (PEs). The programmer can write code that targets every PE of the wafer such that compute and memory are optimally utilized. The programmer communicates to the device via a set of runtime APIs executed on one or more host computers.

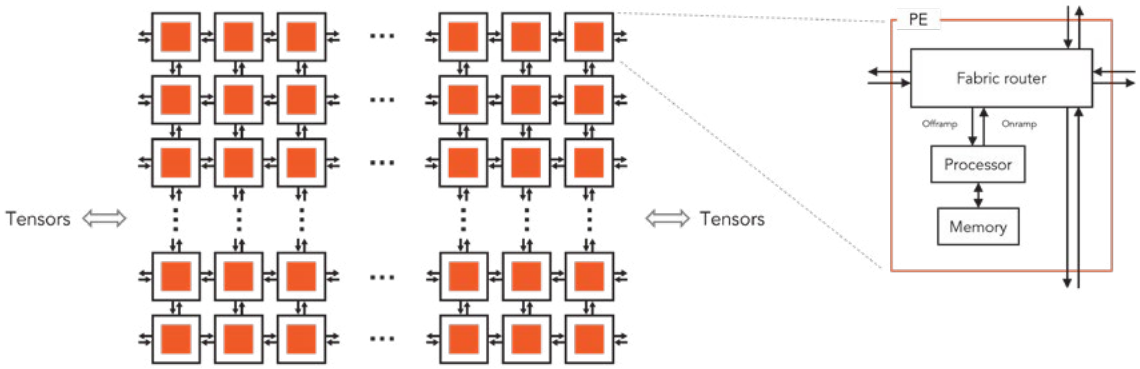


The CS-2 system is programmed similarly to most accelerators today. A host system is used to load programs on to the CS-2 system for execution. The host is capable of targeting either an actual CS-2 system or a software simulator for testing and debugging. Programs intended for the CS-2 system are written in the CSL programming language.

In addition to its focus on programming the individual PE, CSL also allows the developer to specify a group of PEs that each run a copy of a single PE program, and to then arrange several such groups, with different programs running in different groups. It allows the programmer to specify the communication paths through the on-wafer network for data communication within each group and between the groups. This additional flexibility helps CS-2 system programmers take full advantage of what the platform has to offer.

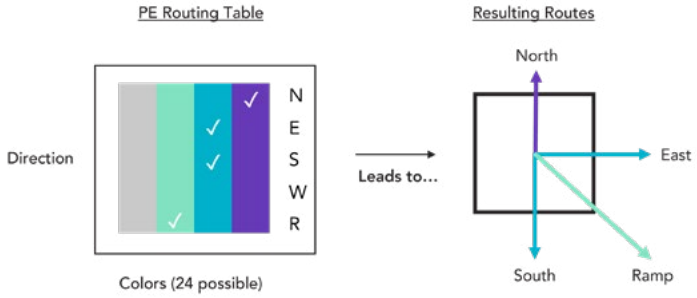
Instruction Set Architecture: Hardware and Data Structures

The 850,000 PEs on the WSE-2 are structured in a 2D grid. Each PE contains a general-purpose compute element (CE), a fabric router, and 48kB of local SRAM memory with single-cycle read/write access latency. PE-to-PE communication latency is also one cycle. The PE contains a network router with links to the CE and to the routers of the four nearest PEs in the north, south, east, and west directions. Communication is integrated into the instruction set, at single 32-bit word granularity, and is accordingly as fast as arithmetic.



The CE's instruction set supports FP32, FP16, and INT16 data types. The Cerebras ISA supports optimized vector operations for processing tensors as well as general-purpose control instructions. A CE can execute vector instructions that perform up to eight operations per clock for FP16 operands.

Data moves from PE to PE in 32-bit packets called wavelets. Software tags each wavelet with a 5 bit virtual channel identifier called a color. Every router is configured to route each of the colors to any subset of its five outgoing links. In this way, for every color a path through the network can be set up to send data from any source to any set of destinations. The router can be statically configured at compile time, or dynamically reconfigured at runtime. The figure below shows how a routing table in each PE maps colors to directions.



For communicating sparse data, a wavelet can be formed from a 16-bit index and a 16-bit data field containing the index i and the value x(i) of a nonzero element of a sparse vector.

Programming the CS-2 using CSL

CSL is designed for writing high-performance programs for PEs on the WSE. CSL is a medium-level language, in that it exposes both low and high-level programming features. CSL is low-level in that it exposes the capabilities of the instruction set and some of the hardware structures that the ISA controls. On the other hand, CSL is a high-level language in that it supports structured control flow, loops, if-then-else constructs, and hides processor registers, performing automatic register allocation. CSL allows for compile time execution of code blocks that take compile-time constant objects as input, a powerful feature it inherits from Zig, on which CSL is based. CSL will be largely familiar to anyone who is comfortable with C/C++, but there are some new capabilities on top of the C-derived basics.

CSL is a statically typed language with syntax similar to languages like C/C++. It supports signed and unsigned integers (16 and 32-bit), half-precision and single-precision floating point operations, and booleans. Functions are supported. CSL also contains built-ins for accessing special compiler features, including type checking, compile-time directives, and pseudo-random number generators. Common control structures like if/else, case/switch, and while are also supported.

Types

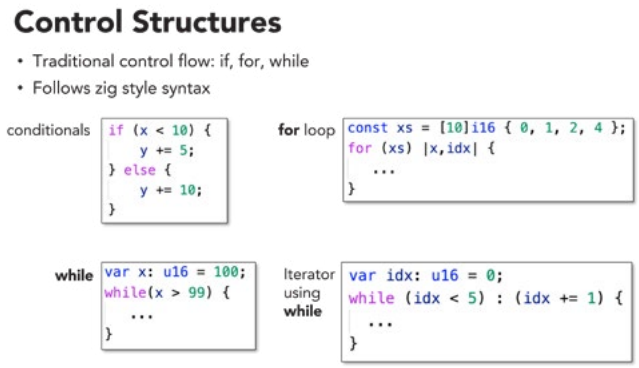
- Syntax similar to other modern languages
  - Go, Swift, Scala, Rust
- Usual list of base types
  - Signed integers: **i16, i32**
  - Unsigned integers: **u16, u32**
  - Floating point: **f16, f32**
  - Booleans: **bool**

```
var x : i16;
const y = 42;
var arr : [16, 4]f32;
var ptr : *i16;
```

Functions

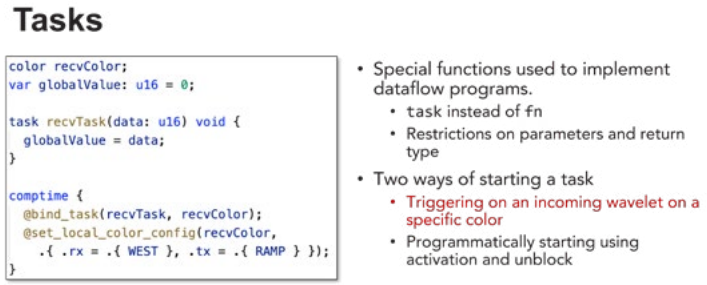
- Call-by-value/reference and inline automatically handled by compiler.

```
fn factorial(x : i32) i32 {
    if (x <= 2) return x;
    return x * factorial(x - 1);
}
```



CSL supports the use of modules. Modules allow one CSL file to include another CSL file. Using modules, programmers can invoke functions or refer to constants within the imported module. Modules can also invoke library functions.

One major new capability of CSL is the introduction of tasks, which are special functions used to implement dataflow programs. Tasks are mapped to colors. When a wavelet arrives, an associated task can be automatically executed. This function operates similarly to a conventional interrupt, with one important difference: we don't suspend running tasks to handle the interrupt, which means we don't have to save and restore states.

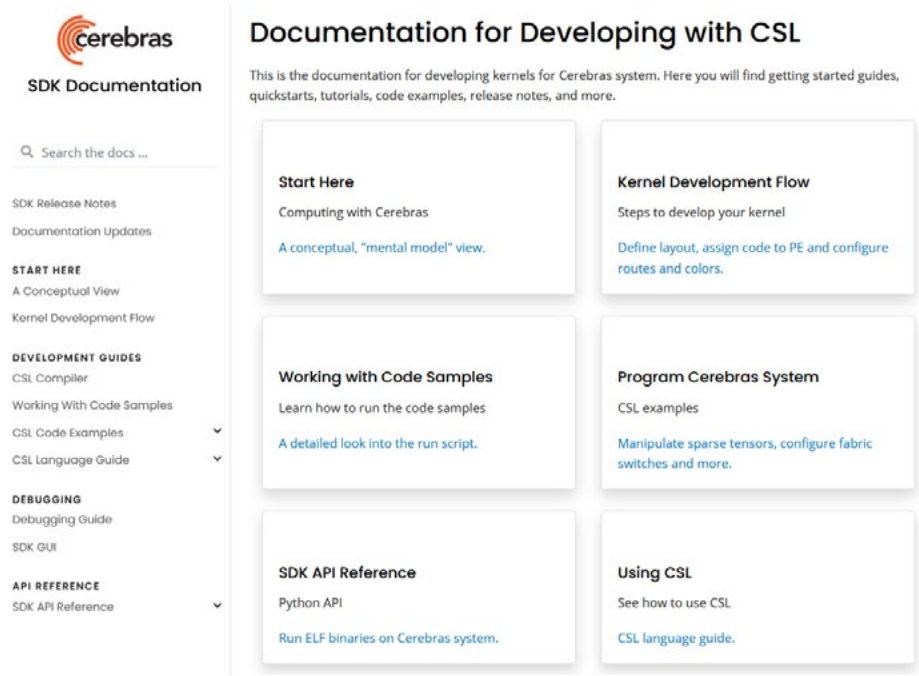


Instructions work on tensors whose operands are specified in data structure descriptors or DSDs. Registers called Data Structure Registers (DSRs) are used at runtime to hold DSDs so that DSDs may be used as operands in any instruction. An instruction such as add or move can take one or more DSD operands, and when it has one, the instruction runs for multiple cycles to complete operations on all of the data referenced in the DSD. Thus, a single instruction can access and work on an in-memory data array. This allows hardware to reduce the runtime cost of address generation and loop control and can be used to encode complex indexing patterns.

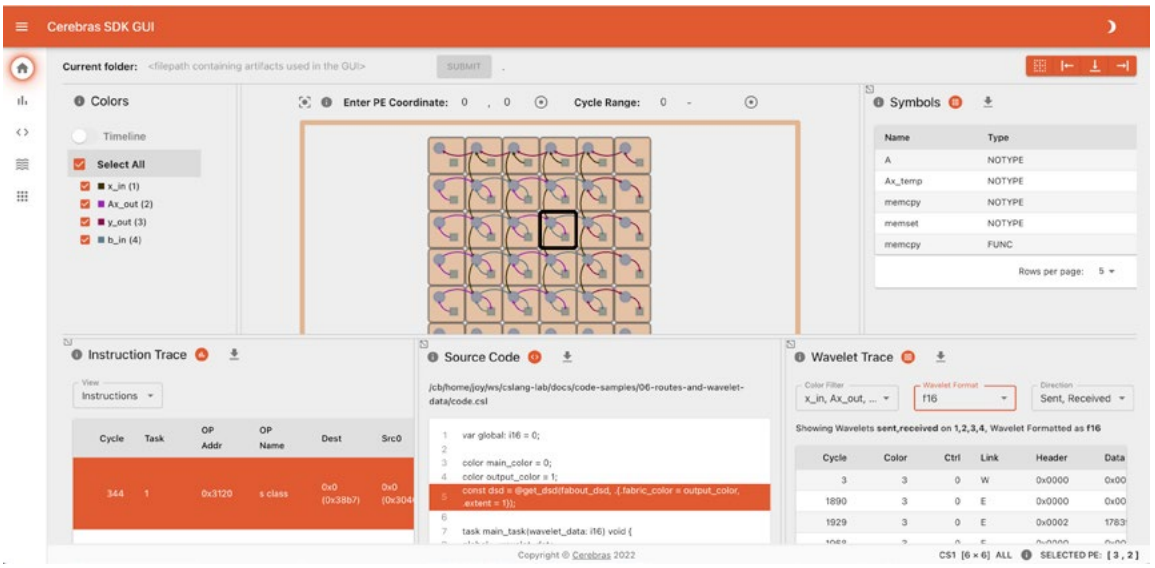
A DSD may also describe a vector of data elements arriving (or departing) as a sequence of wavelets of a certain color. Thus, in addition to triggering tasks, incoming data can be directly consumed or produced by the CE as data operands for an operation that runs while that data arrive. A fabric vector operation of this kind may be run as a background thread, allowing other work to be done while waiting for more data to arrive or depart.

Offered in the SDK

The Cerebras SDK includes code samples, tutorials, getting started guides, a language guide, and information on how to use tools like the included debugger. There are examples of basic tasks and colors, how to use multi-PE kernels, and best practices when programming the CS-2 system for specific workloads.



One important feature of the SDK is a simulator and debugging tool. This allows developers to debug code intended to run on the CS-2 system even when the hardware platform is not available. This simulator is cycle-accurate and can run on most platforms.



Code Examples

Device Side

This code snippet is an example of a program intended to run on a single PE, written in standard Python. This is the code that would execute on the CS-2 system itself. In this example, the program defines both a global variable and a color. At compile time, main_task and main_color are bound together before main_color is activated. The code runs on a reserved 1x1 PE grid. At runtime, the main_task thread assigns the variable "global" a value of 42.

Define variable named global, initializing it to 0.Define a color `main_color`, using id 0.

Define a task, executed at runtime, which writes the value 42 to `global`.

At compile-time, bind main_task to `main_color`. Then, activate `main_color` so that `main_task` runs when execution at runtime begins.

Reserve a 1x1 rectangular region of PEs.At the upper-left-hand corner of the rectangular region (x=0, y=0), place the `device.csl` code (i.e., current code file)

```
// Define global variables and constants
var global: i16 = 0;
color main_color = 0;

// Executed at runtime
task main_task() void {
  global = 42;
}

// Compile-time functions
comptime {
  @bind_task(main_task, main_color);
  @activate(main_color);
}

// Define code layout
layout {
  @set_rectangle(1, 1);
  @set_tile_code(0, 0, "device.csl");
}
```

Host side

This code snippet shows the complimentary host-side code. The host-side program provides a path to the binaries created by the CSL compiler and specifies the name of the (simulated) core that should be written to post simulation. Once the application has finished running on the simulator, the host reads the result from the core. The result is then compared to the aforementioned reference value "42" to confirm its accuracy.

SDK uses numpy data types on the host.

Provide path to binaries.Specify name of core generated by simulator.

Load the binary onto the simulator (or CS-2) and start execution.

Read the result of the program by loading the variable named `global` written on the device.

Compare the result with a reference value 42.

```
import numpy as np

from cerebras.elf.cs_elf_runner import CSELFRunner
from cerebras.sdk.elf_inspector import ELFInspector


# Path to ELF and core files
elf_paths = ["out_0_0.elf"]
core_path = "core"

# Simulate ELF file and produce a core
runner = CSELFRunner(elf_paths)
runner.run_on_simfabric(core_path)

# Read the result from the core
loader = ELFInspector(elf_paths[0], core_path)
result = loader.get_as_array("global", 1, np.int16)

print("*****************")
print(f"Result is {result[0,0,0]}")
print("*****************")

np.testing.assert_equal(result[0,0,0], 42)
```

Starting Your Collaboration with Cerebras

If you are curious about programming for wafer-scale or want to evaluate whether the CS-2 system would be a good fit for your organization, we encourage you to get in touch.

The following data points will help us answer your questions most effectively:

1. What limits your performance today? Arithmetic? Memory latency or bandwidth? Communication latency or bandwidth? Something else?
2. Can you give us specific algorithms or example code and data?
3. How do you develop code today?
4. In what languages?
5. What libraries do you use?
6. What floating point precision do you need? What integer word length?
7. Are any open benchmarks of interest to you?

www.cerebras.net/sdk