

Cerebras System

remarkable hardware accelerator architecture & its programming model

Yuri Takigawa

July 1, 2025

The university of Tokyo, EEIC, Taura Lab

Contents

Installation and Setup

A Conceptual View: Hardware organization

PE (Processing Elements)

A Conceptual View: Programming model

Programming Language

Communication

Code Execution Unit: Task

Another use of wavelet arrival : Data Struct

layout and host code

Installation and Setup

Installation and Setup

A Conceptual View: Hardware organization

PE (Processing Elements)

A Conceptual View: Programming model

Programming Language

Communication

Code Execution Unit: Task

Another use of wavelet arrival : Data Struct

layout and host code

How to get an access to SDK

Fill the form on this [link](#).

- Human operators reply, thus it takes more than half a day.
- The reply includes **Dropbox link** to the **SDK files**.

Most of the things about **installation and setup** are [here](#).

- The most important thing is that this SDK is for **amd64** architecture.

Overview of installation and setups

Some of the (critical) things below are **NOT** explicitly written in the [guide](#).

1. **Azure VM** is highly recommended as an environment (**Miyabi** is not available here...)
 - Recommended instance type is **Standard B4ms** (4vcpu, 16GiB memory) with 64GiB disk.
2. When you connect via **ssh**,

```
ssh -i ~/.ssh/YOUR_PRIVATE_KEY -Y -L 8000:localhost:8000 YOUR_USERNAME@PUBLICIP
```

 - transfer **port 8000** to remote **port 8000**
 - YOUR_USERNAME is **NOT** a resource name.
 - YOUR_PUBLICIP can be seen on the resource via [azure home](#)
 - You can also refer past [spring-training](#) by gotonao

Overview of installation and setups

3. You can follow the [guide](#) at installation/setup, but some filenames are changed.
4. Finally, you can try remote GUI debug (step7 of the guide).
 - `sdk_debug_shell` visualize after running the test, open `http://localhost:8000/sdk-gui` at your local browser (eg. chrome)
 - Sometimes, version conflict occurs and show a kind of bugs.

The screenshot displays the Cerebras SDK GUI interface. The top bar shows the title 'cerebras SDK GUI' and a 'Work directory' path. The main area is divided into three panels:

- PE Coordinate:** A grid visualization showing a small cluster of blue squares on an orange background, representing the physical layout of the device.
- Instruction Trace:** A table showing execution cycles, tasks, and source code snippets. The table has columns for Cycle, Task, OP Addr, OP Name, Dest, Src0, Src1, Src2, Source, and Source Line.
- Wavelet Trace:** A table showing execution cycles, colors, and data. The table has columns for Cycle, Color, Ctrl, Link, Header, and Data.

The Instruction Trace table shows the following data:

Cycle	Task	OP Addr	OP Name	Dest	Src0	Src1	Src2	Source	Source Line
108	22	0x004	NOP		0			./lib/rosc-hal/clang-hal-sd-1.4.0/20190512/2200-1442-a23e2918.html	88
111	22	0x004	NOP		0			./lib/rosc-hal/clang-hal-sd-1.4.0/20190512/2200-1442-a23e2918.html	88
112	22	0x004	NOP		0			./lib/rosc-hal/clang-hal-sd-1.4.0/20190512/2200-1442-a23e2918.html	88

The Wavelet Trace table shows the following data:

Cycle	Color	Ctrl	Link	Header	Data
116	22	1	E	37	31
127	22	1	E	37	33
128	22	1	E	37	34
129	22	1	E	37	35
130	22	1	E	37	36

A Conceptual View: Hardware organization

Installation and Setup

A Conceptual View: Hardware organization

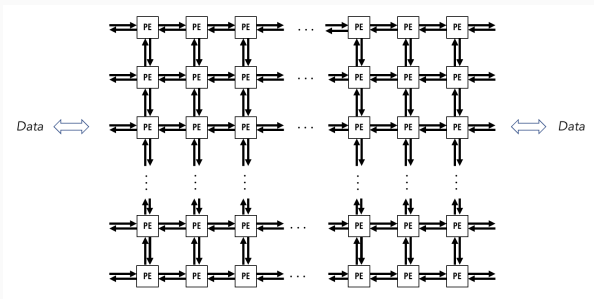
PE (Processing Elements)

A Conceptual View: Programming model

Wafer Scale Engine

Cerebras refers its hardware accelerator as a **WSE (Wafer Scale Engine)**.

- WSE consists of hundreds (dies) of thousands (数千万個) of independent **PE** (processing element)s (\sim cores).¹
- The PEs are interconnected by communication links, and they form a **two-dimensional rectangular mesh** on one single silicon wafer.

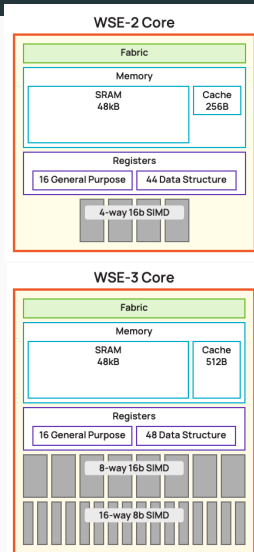


¹uses TSMC 5nm processed node

Characteristics of a PE: memory

- Each PE has its own **physically-local SRAM** (called **local PE memory**)^a with single-cycle access.
 - is 48kB total, consists of 8 banks, and has full datapath bandwidth: 2 64(128)bit read + 1 64(128)bit write per cycle
 - each bank has 6kB, and 32bit wide, has single port
- All the code and data related to the execution on the PE are stored **within** this memory.
- This physically-local memory is **logically local** as well (i.e., No other PE are directly accessible to this memory)

^a200x normalized memory BW vs. GPU

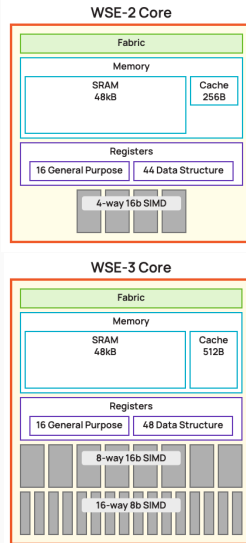


Characteristics of a PE: processor

- Each PE has a **processor** called **CE (Compute Engine)**.
 - 16 general purpose registers, 48 data structure registers
 - Compact 6-stage pipeline
 - Flexible **general ops** (e.g., arithmetic, logical, load/store, compare, branch) for control processing
- CE has **SIMD** computing unit
 - In the WSE2, **4-way 16 bit SIMD^a**, each way has ALU for FADD, FMUL, FMAC^b etc.,
 - In the WSE3, **8-way 16 bit SIMD** and **16-way 8 bit SIMD**

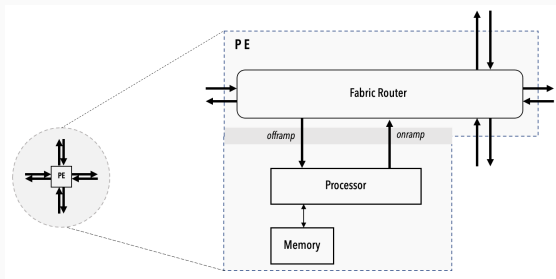
^awhich means execute single instruction with 4 different data simultaneously

^bFused Multiply-Add



Characteristics of a PE: processor

- Each PE has its own independent PC (program counter).
 - Thus, each PE can execute codes **asynchronously** by default.



Characteristics of a PE: router

- Each PE has the hardware unit for communication (send, receive) called **Router**.
- A **Router** is directly connected to its own **CE** via **bidirectional** link called **RAMP (Router ALU Messaging Path)**.²
- A **Router** is directly connected to the routers of the four nearest neighboring PEs (north, south, east, west)
 - Thus, a router has 4 ports
- A **Router** has 8 **input queues** and **output queues** inside the PE.
 - An **input queue** is a **hardware buffer** where data is temporarily stored before the entering the CE. (I will explain what this means later)

²You may have heard this word as インターチェンジ of highway

A Conceptual View: Programming model

Installation and Setup

A Conceptual View: Hardware organization

A Conceptual View: Programming model

Programming Language

Communication

Code Execution Unit: Task

Another use of wavelet arrival : Data Struct
layout and host code

Programming Language: CSL

- To develop code for the WSE, write *device code* in the **CSL (Cerebras Software Language)**, and *host code* in *Python*.
 - The host code is responsible for **copying data to and from the device**
 - **CSL** gives programmers full control of the **WSE**.
- Then, compile the *device code* with `cs1c`, and run your program on either **Cerebras fabric simulator** (or the actual network-attached device).
 - The usage is [here](#).

Programs and Tasks

A **CSL program** consists of one or more *subprograms*.

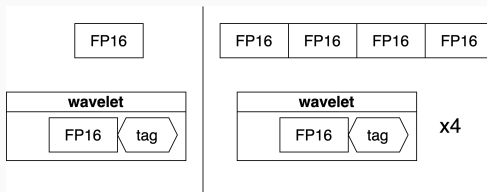
Subprogram has two **types** (declaration).

- **function**: callable
- **task**: a procedure that cannot be called from other code
 - something like **atomic** (i.e., unsplittable) code block managed by hardware
 - tasks are managed at *the specialized hardware unit* of **CE** similar to rich *NPC generator*
 - A task can be **activated** (i.e., ready for running) by some *hardware trigger* (imagine a flip of a flag bit)
 - tasks are started by PE hardware (specifically *NPC generator* of **CE**)
 - Only **one** task can be executed at a time on the CE
 - Once a task is started by hardware, it runs until it complete. Then, the *NPC generator* chooses a new task to run.

The unit of communication between PEs: wavelet

32-bit messages (\sim packets), called **wavelets**, can be sent to or received by neighboring PEs **in a single clock cycle**.

- Arrivals of wavelets trigger something inside the PE
 - **Task Activation**
 - **Stored in a data struct managed with a fabric DSD**
- transferring data of massive size (like array, tensor) is **split into multiple wavelets**, and data of wavelets that arrive at the destination PE first is **buffered** in the **input queue** until all wavelets arrive.



Virtual Communication Channel: Color

The virtual communication path (channel) through which **wavelets** (packets) travel is called **color**.

- There exists 24 virtual channels used by hardware.
- All colors transfer data on a single physical channel.
 - If multiple colors have wavelets to send via physical path from PE X to PE Y, those **colors (not wavelets)** are scheduled by **hardware arbiter** on router.
 - **IMPORTANT:** The congestion of one color does **NOT** block traffic of another color; **Fairness** between colors (not wavelets)
 - For example, RR algorithm could be used as a policy.
- Each **wavelet** has a 5-bit **tag** that encodes its **color**

Installation and Setup

A Conceptual View: Hardware organization

A Conceptual View: Programming model

Programming Language

Communication

Code Execution Unit: Task

Another use of wavelet arrival : Data Struct
layout and host code

Task IDs and Types of Tasks

Each task can be associated with **task ID** from 0 to 63.

- **Data Task**: the arrival of wavelet triggers its activation, its **ID** is associated with a **input queue** on the **router**³
- **Local Task**: the `@activate(task_id)` in some other codes within the same PE triggers its activation,
- **Control Task**: controls other tasks on the same PE as follows, its **ID** can take any values from 0 to 63.
 - unblock other **data task**
 - conditional launch of **local task**

³In the WSE2, task ID is directly associated with the **color** with implicit linkage between **input queue** and task ID

The conditions to be ready for execution

There are two **conditions** for tasks be scheduled by **task picker** (hardware selector)

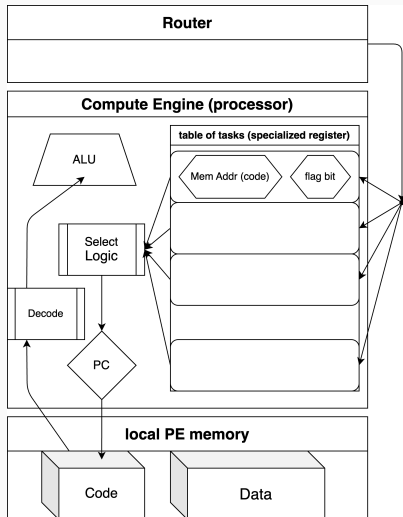
- **Activated**

- every task is **inactive** by default
- programmers can activate the task within the same PE, with `@activate(task_id)`.
- programmers can activate the task in another PE, with `@send_to_color(output_queue_id)`.

- **Unblocked**

- every task is **unblocked** by default
- but, programmers can block the ID of a task at compile time, with `@block(task_id)`.

Pseudo Image of hardware that manages Data Task



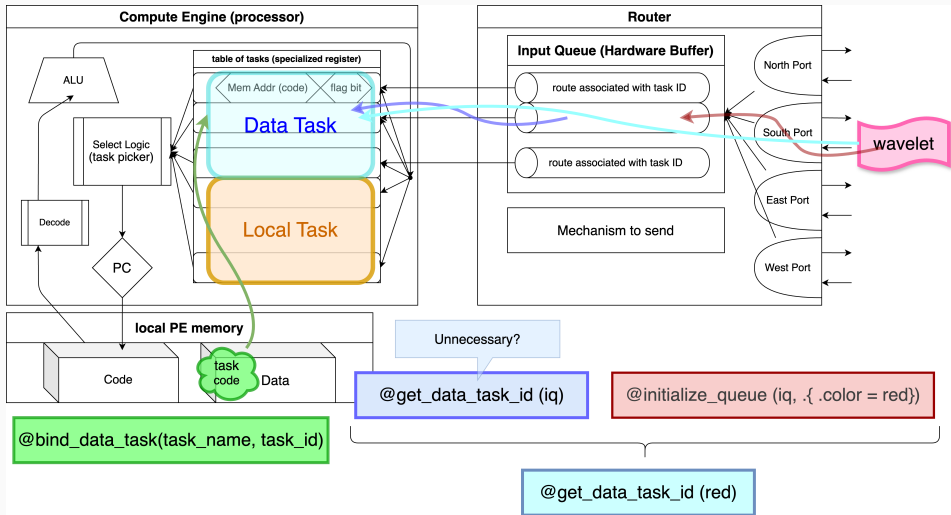
Communication via router: Task activation

1. Secure **color** for communication with `const color_name: color = @get_color(.)`
2. Secure **input queue** with `const iq_name: input_queue = @get_input_queue(.)`
3. Create **data task ID** from **an input queue**

```
const task_id_name: data_task_id = @get_data_task_id(iq_name)
```

- Each data task is bound to an input queue
4. Define what to do in the task with `task task_name(wavelet_data){ ... }`
 5. Define `comptime{ ... }` block:
 - Bind defined task `task_name` to created task ID `task_id_name` with
`@bind_data_task(task_name, task_id_name)`
 - Bind input queue(s) to which data task is bound to color
`@initialize_queue(iq_name, {.color = color_name});`

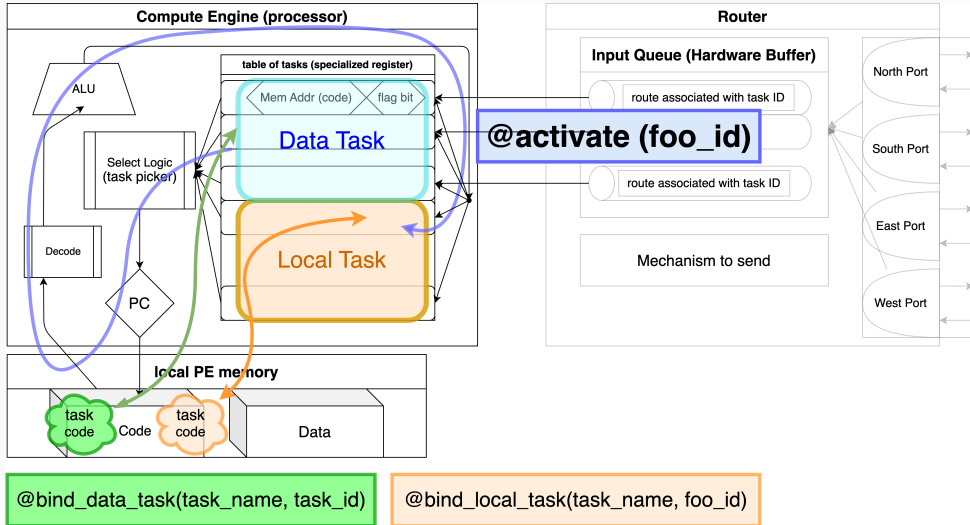
Pseudo Image of task activation from Router



Code Template: link computation and communication using Task

```
1 // 7 is the ID of a color with some defined data routing
2 const red: color = @get_color(7);
3
4 // On WSE-3, 2 is the ID of an input queue which will be bound to our data task.
5 const iq: input_queue = @get_input_queue(2);
6
7 // For WSE-2, the ID for this task is created from a color.
8 // For WSE-3, the ID for this task is created from an input queue.
9 const red_task_id: data_task_id =
10     if (@is_arch("wse3")) @get_data_task_id(iq)
11     else                    @get_data_task_id(red);
12
13 var result: f32 = 0.0;
14
15 task main_task(wavelet_data: f32) {
16     result = wavelet_data;
17 }
18
19 comptime {
20     @bind_data_task(main_task, red_task_id);
21
22     // For WSE-3, input queue to which our data task is bound must be bound to color red.
23     if (@is_arch("wse3")) @initialize_queue(iq, .{ .color = red });
24 }
```

Pseudo Image of task activation from another task



Code Template: link computations using Task

A data task (`main_task`) activates a local task (`foo_task`)

```
1  const red: color = @get_color(7);
2  const iq: input_queue = @get_input_queue(2); // Used only on WSE-3
3
4  const red_task_id: data_task_id =
5      if (@is_arch("wse3")) @get_data_task_id(iq)      else      @get_data_task_id(red);
6
7  const foo_task_id: local_task_id = @get_local_task_id(8);
8
9  var result: f32 = 0.0;  var sum: f32 = 0.0;
10
11 task main_task(wavelet_data: f32) {
12     result = wavelet_data;
13     @activate(foo_task_id);      // special instructions to flip activation flag of foo_task
14 }
15
16 task foo_task() {    sum += result;  }
17
18 comptime {
19     @bind_data_task(main_task, red_task_id);
20     // bind foo_task to the hardware to be managed
21     @bind_local_task(foo_task, foo_task_id);
22     if (@is_arch("wse3")) @initialize_queue(iq, .{ .color = red });
23 }
```

Contents

Installation and Setup

A Conceptual View: Hardware organization

PE (Processing Elements)

A Conceptual View: Programming model

Programming Language

Communication

Code Execution Unit: Task

Another use of wavelet arrival : Data Struct

layout and host code

Preliminaries: the hardware instruction of Tensor

CSL supports several **tensor ops**

- Every tensor op has its options
 - asynchronous execution means, release software serialization
 - i.e., start execution even before the previous op is completed, considering whether or not hazard (data, structural) exist

```
1 .{  
2   .async = true,           // asynchronous execution  
   flag  
3   .activate = task_id,     // task id of the task  
   activated when this op completes  
4   .element_type = f32,     // type  
5   // other options specific to each op  
6 }
```

- The format of these ops are ([here](#)):

```
1 // ARITHMETIC  
2 // dst = src1 * src2 + acc  
3 @fmacs(dst, acc, src1, src2 [, options])  
4 // dst = src1 {+, -, *, /} src2  
5 @fadds(dst, src1, src2 [, options])  
6 @fsubs(dst, src1, src2 [, options])  
7 @fmuls(dst, src1, src2 [, options])  
8 @fdivs(dst, src1, src2 [, options])  
9  
10 // COPY  
11 // dst = src  
12 @fmovs(dst, src [, options])  
13 // dst = src(size)  
14 @copy(dst, src, size_in_bytes)  
15  
16 // ELEMENT-WISE OPS  
17 @fexps(dst, src [, options]) // exp  
18 @flogs(dst, src [, options]) // ln  
19 @frelus(dst, src [, options]) // ReLU  
20  
21 // REDUCTION OPS  
22 @fsum(dst, src [, options]) // sum of items  
23 @fmax(dst, src [, options]) // dst is scaler
```

DSD: Abstraction of data consists of multiple values like tensor

Data Structure Descriptors (DSDs) [More details are available [here](#).]

- are a **compact representation** of
 - a chunk of memory (or)
 - a sequence of incoming or outgoing wavelets
- enable various **repeated operations** to be expressed using just **one hardware instruction**
- is an software object which consists of
 - `dst_type`: one of `mem1d_dsd`, `mem4d_dsd`, `fabin_dsd`, Or `fabout_dsd`
 - `properties`: different `dst_type` has different properties

```
1 // A 'mem1d_dsd' created through explicitly
  specifying properties
2 const dsd1 = @get_dsd(mem1d_dsd, .{
3     .base_address = access_of_A.base_address,
4     .offset = access_of_A.offset,
5     .stride = access_of_A.stride,
6     .extent = access_of_A.extent});
```

```
1 // 'access_of_A' is exactly equivalent to:
2 // .{.base_address = &A, .offset = 42, .stride =
  .{2}, .extent = .{10}}
3 const access_of_A: comptime_struct
4     = |i|{10} -> A[2*i + 42];
5
6 const dsd2 = @get_dsd(mem1d_dsd,
7     .{.tensor_access = access_of_A});
```


ADVANCED: hardware microthread

hardware **microthread** is an mechanism to distribute and manage hardware resources for enabling asynchronous execution [Details are [Microthread IDs](#), [Async DSD Ops](#)]

- Arbitration of hardware resource like ALU, Memory Access Unit, Router Interface (~ scheduling)
- An asynchronous DSD operation can be assigned a **microthread ID** through `.ut_id = @get_ut_id(n)`
 - microthread ID could be different from input/output queue ID⁴
 - If multiple DSR/DSD operands have the `.ut_id` setting specified, the hardware will pick one of them according to the order: `dst > src1 > src2`
- programmers can attach priority to each thread (including main thread).
- **IMPORTANT:** The programmer is responsible for ensuring that [no two concurrent DSD operations share a microthread](#).

⁴In WSE2, the same as output queue ID when using `fabs_dsd`, otherwise the same as input queue ID

The hardware mechanisms to manage DSD: DSR

There have to be the hardware mechanism that utilize DSDs: **Data Structure Registers (DSRs)**

- **DSRs** are **physical registers** that are used to store DSD values
- All DSD operations will actually operate on DSRs behind the scenes, thus all DSD operands to DSD operations **must be loaded to DSRs** before executing
- Each DSR belongs to one of three DSR files, namely `dest`, `src0`, `src1` DSR files (i.e., physically distinguished)
 - `dsrc_dest`: DSR number (レジスタ番地) that can only be used to store a destination operand DSD of a DSD operation
 - `dsrc_src0`: DSR number that can be used to store a source DSD as well as a destination operand DSD
 - `dsrc_src1`: DSR number that can only be used to store a source operand DSD
- Basically, compiler allocate DSR (and extra DSR) automatically, but programmers can use them directly with `dsrc = @get_dsrc, @load_to_dsrc(dsrc, dsd [, option])`

sender

1. set colors used to send/recv data between PEs
2. assign task ID used by a local task to unblock cmd stream

Code Template: link computation and communication using DSD

Contents

Installation and Setup

A Conceptual View: Hardware organization

PE (Processing Elements)

A Conceptual View: Programming model

Programming Language

Communication

Code Execution Unit: Task

Another use of wavelet arrival : Data Struct

layout and host code

Writing the top-level CSL file

There are a few things that programmers need for our device code to form a complete program

- **Initialization the infrastructure of the memcpy library** with `@import_module`
 - In order to allow the host to **launch kernels** and **copy data** to and from the device
 - has to specify **width** and **height** parameters which correspond to the **dimensions of the program rectangle**
- **A top-level “layout” file**
 - **define the program rectangle** on which our kernel will run, with
`@set_rectangle(columns_dim, rows_dim)`
 - **assign a code file** to the single PE in our rectangle, with
`@set_tile_code(column_idx, row_idx, "pe_program.csl" [, parameters])`
 - **pass memcpy parameters** as a parameter, which are parameterized by the PE's column number(idx), with `.{ .memcpy_params = memcpy.get_params(column_idx)}`

Code template of the top-level CSL file

```
1 // Import memcpy layout module for 1 x 1 grid of PEs
2 const memcpy = @import_module("<memcpy/get_params>",
3                               .{ .width = 1, .height = 1 });
4
5 layout {
6
7     // Use just one 1 PE (columns=1, rows=1)
8     @set_rectangle(1, 1);
9
10    // The lone PE in this program should execute the code in "pe_program.csl"
11    @set_tile_code(0, 0,
12                  "pe_program.csl",
13                  .{ .memcpy_params = memcpy.get_params(0) });
14
15    // Export device symbol for array "y"
16    // Last argument is mutability: host can read y, but not write to it
17    @export_name("y", [*]f32, false);
18
19    // Export host-callable device function
20    @export_name("init_and_compute", fn() void);
21 }
```

Writing the host code

1. Import libraries which is required

- SdkRuntime is the **library** containing the functionality necessary for **loading and running the device code**, as well as **copying data on and off the wafer**.
- MmcpyDataType and MmcpyOrder are **enums** containing types for use with *memcpy* calls

2. Instantiate (=construct) runner objects like

```
runner = SdkRuntime(args.name, cmaddr=args.cmaddr)
```

- name: specify the directory containing the compilation output
- cmaddr: attach **IP address** of targetted real accelerator obtained from command-line like `--cmaddr $CS_IP_ADDR:9000`⁵

3. Load and Run device kernel (named `init_and_compute` here)

- Before loading the program, get symbol for copying y result off device
- `runner.load()` → `runner.run()` → `runner.launch('init_and_compute' [, option])`

⁵CS use port 9000 to connect to the system and launch the program

Writing the host code

4. Copy back result (with many arguments attached as follows)

- Before copying, **allocate space** on the host to hold the result

- 4.1 the array on the host to hold the result `y_result`, the symbol on device that points to the y array `y_symbol`
- 4.2 To specify the location of rectangle of PEs from which to copy (called **ROI**), give the northwest corner of the ROI `0, 0` and the width and height of the ROI `1, 1`
- 4.3 how many elements to copy back from each PE in the ROI `M` (if 2darray, $M*N$)
- 4.4 `ROW_MAJOR` specifies that the data is ordered by (height, width, element)
- 4.5 `data_type` keyword specifies the width of the data copied back
- 4.6 `nonblock=False` specifies that this call will not return control to the host until the copy into `y_result` has finished

```
1 y_result = np.zeros([1*1*M], dtype=np.float32)
2 runner.memcpy_d2h(y_result, y_symbol, 0, 0, 1, 1, M, streaming=False,
3     order=MemcpyOrder.ROW_MAJOR, data_type=MemcpyDataType.MEMCPY_32BIT,
4     nonblock=False)
```

[Appendix] ROW MAJOR and COLUMN MAJOR

"Learn by example" is the best way to understand the concept.

Here is the example of copying from multiple PEs.

- Configuration

- Size of ROI: **height** = 2, **width** = 3
- Calculated Data in each PE

```
1    PE0 = [[1, 2, 3],
2           [4, 5, 6]]
3    PE1 = [[7, 8, 9],
4           [10, 11, 12]]
```

- ROW_MAJOR case

- Each PE data is copied continuously
- Copied result (1d): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

- COLUMN_MAJOR case

- Elements with the same index in each PE are copied continuously
- Copied result (1d): [1, 7, 2, 8, 3, 9, 4, 10, 5, 11, 6, 12]

Finally, Compile and run it!

1. compile device code `layout.cs1`
2. run the host code `run.py`

1. [cerebras SDK Documentation \(1.4.0\)](#)
2. [Cerebras AI Day Deck: A closer look at the world's fastest AI Chip](#)
3. [Cerebras Architecture Deep Dive: First Look Inside the HW/SW Co-Design for Deep Learning](#)