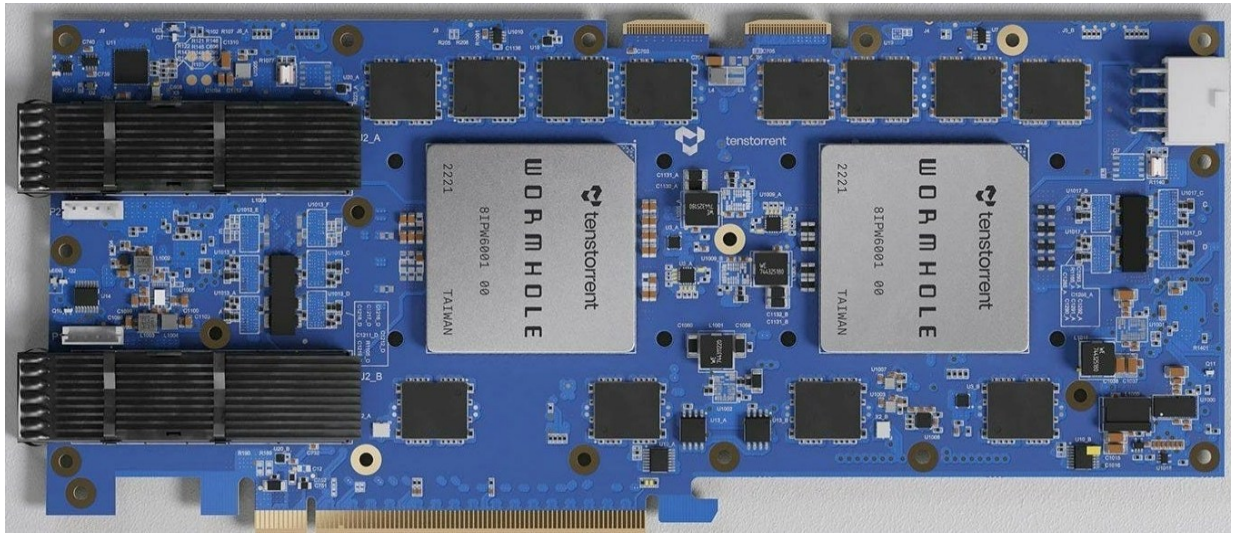
Overview of TT-Metalium SDK: Part one



epcc

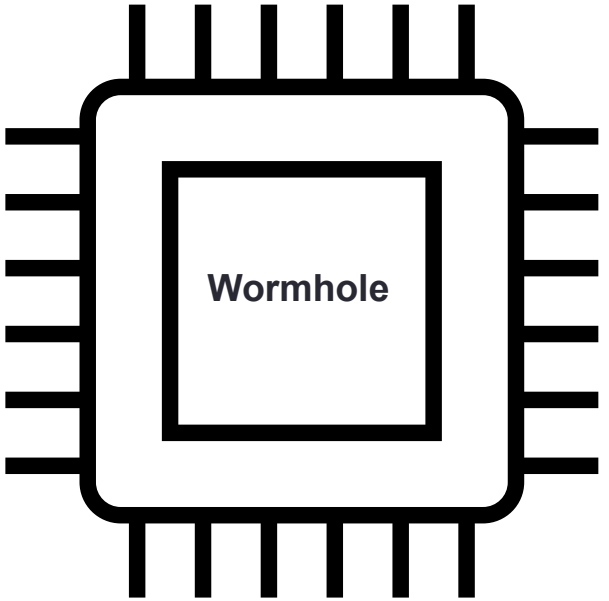
tenstorrent

Host and device overview

Host

PCIe accelerator

- Configure DDR data
- Create CBs and L1 buffers
- Set up kernels
- Copy input data to DDR



- Results from DDR

The structure of the host code

```
• Initialise the device
• Allocate DDR-DRAM memory
• Allocate L1 buffers
• Allocate circular buffers (CBs)
• Create kernels and set their runtime arguments

• Copy input data onto device
• Queue up kernels to run
• Wait for kernels to complete

• Copy results back to host
```

- There is quite a bit of boilerplate code in the host C++ file (we will see this in the practical) but in-fact it's actually quite simple
  - These are the major activities that the host is doing
  - We will look at these in more detail now….

Host: Allocating memory in DDR-DRAM

- There is 24GB of GDDR6 DRAM available on the Wormhole board
  - This is visible to the host and typically Tensix cores will load from, and write to, this

```
// Create descriptor of DRAM allocation
constexpr uint32_t single_tile_size = 4 * DATA_SIZE;
InterleavedBufferConfig dram_config {
    .device = device,
    .size = single_tile_size,
    .page_size = single_tile_size,
    .buffer_type = BufferType::DRAM };

// Use descriptor configuration to allocate buffers in DRAM on the device
std::shared_ptr<Buffer> src0_dram_buffer = CreateBuffer(dram_config);
```

Create a memory configuration that defines the overall size, page size and location

- All memory is allocated in pages, here we have a single page which equals the overall size

Performs the memory allocation based upon the configuration. We can then use this handle on the host when referring to the buffer

Host: Transferring data between host and DRAM

Writes data from the host to the device's DDR-DRAM

Both these calls operate in a blocking (true) or non-blocking form. Blocking will wait for the data transfer to complete, non-blocking will add this to the command queue and that is then waited on later

```
EnqueueWriteBuffer(cq, src0_dram_buffer, src0_data, false);
```

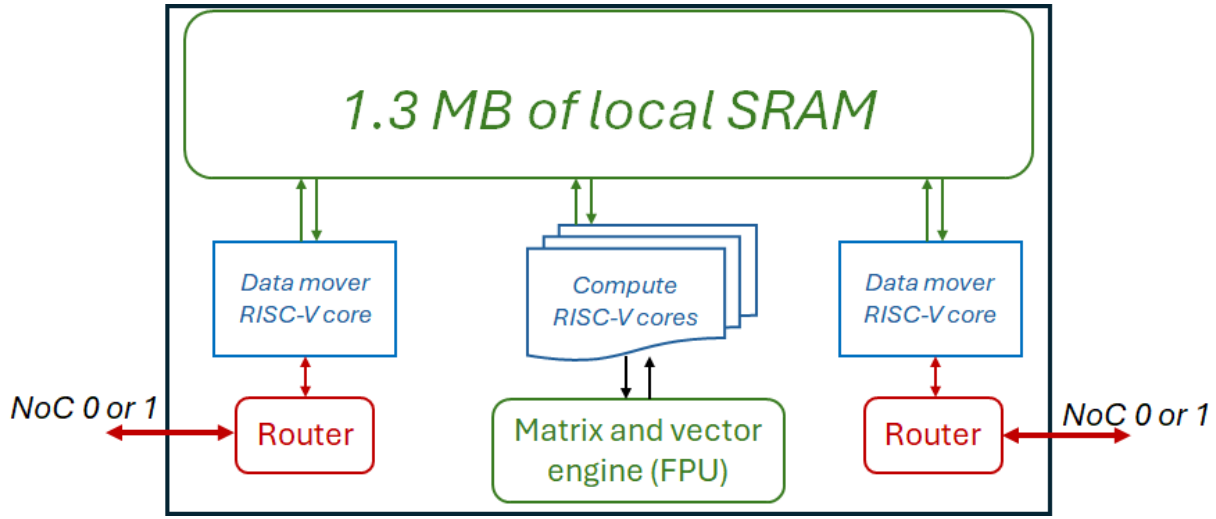
The command queue *The buffer handle* *Host memory* *Whether to block*

```
EnqueueReadBuffer(cq, dst_dram_buffer, result_data, true);
```

Reads data from the DDR-DRAM to host memory

Host: Allocating memory in L1 SRAM

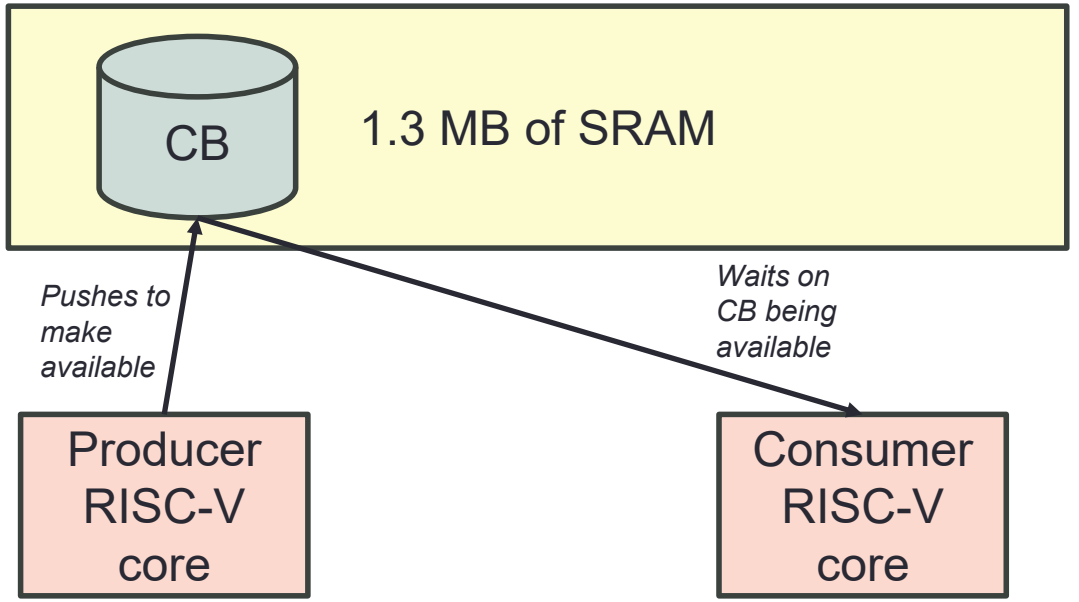
```
tt::tt_metal::InterleavedBufferConfig l1_config {
    .device= device,
    .size = single_tile_size,
    .page_size = single_tile_size,
    .buffer_type = tt::tt_metal::BufferType::L1 };

std::shared_ptr<tt::tt_metal::Buffer> l1_buffer_1 = CreateBuffer(l1_config);
```



- Each Tensix unit has 1.3MB of local SRAM
  - Can allocate memory into here and then use this on the device, for instance as a temporary scratch pad.
- Approach is similar to DDR allocation, but with a buffer type of L1

Host: Circular Buffers

- In a way are similar to L1 buffers
  - In addition to memory, provide a producer-consumer model to share data between RISC-V cores in a structured manner
  - Are created on the host for each Tensix unit




```
constexpr uint32_t src0_cb_index = CBIndex::c_0;
CircularBufferConfig cb_src0_config =
    CircularBufferConfig(single_tile_size, {{src0_cb_index, tt::DataFormat::UInt32}})
        .set_page_size(src0_cb_index, single_tile_size);
CBHandle cb_src0 = tt_metal::CreateCircularBuffer(program, core, cb_src0_config);
```

Configuration of CB (size, data type and index)

Create CB based on configuration on a specific core

Host: Configuring and launching kernels

```
KernelHandle reader_kernel_id = CreateKernel(
    program,
    "kernels/dataflow/read_kernel.cpp",
    core,
    DataMovementConfig{.processor = DataMovementProcessor::RISCV_0, .noc = NOC::RISCV_0_default});

// Configure reader runtime kernel arguments
SetRuntimeArgs(
    program,
    reader_kernel_id,
    core,
    {src0_dram_buffer->address(),
     src1_dram_buffer->address(),
     l1_buffer_1->address(),
     l1_buffer_2->address(),
     DATA_SIZE});
```

Create a kernel based upon the source filename, specify it will be placed on a specific RISC-V baby core of a Tensix core

Set runtime arguments for the kernel (we will look at the device side unpacking those in a moment….)

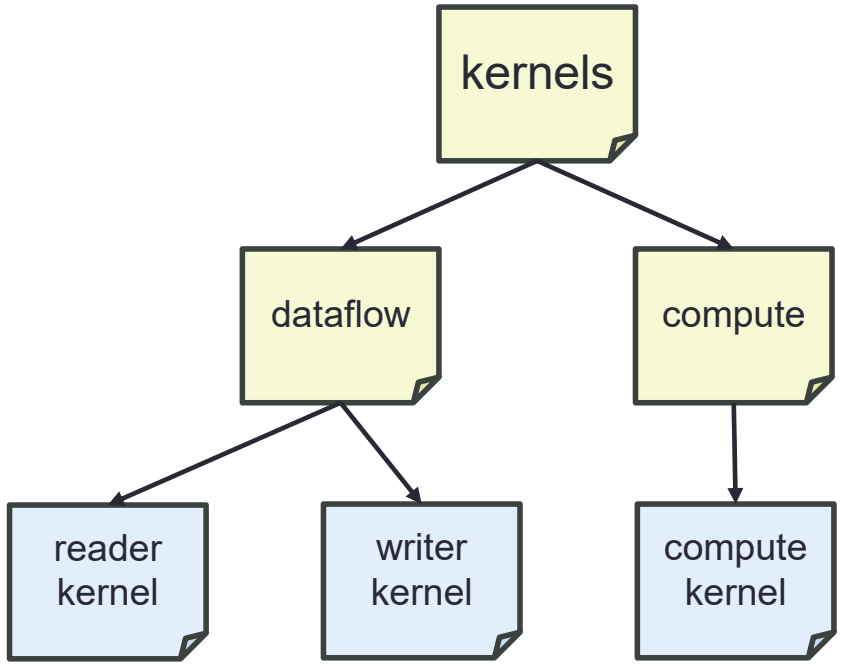
```
EnqueueProgram(cq, program, false);
// Wait on the command queue to complete
Finish(cq);
```

Add the program to the command queue, running the kernels. The Boolean determines whether this is blocking (true) or non-blocking (false)

Wait for command queue to finish all it's tasks

Device side

- Typically three kernels are provided by the programmer
  - Reader kernel is launched on data movement baby RISC-V core 0
  - Writer kernel is launched on data movement baby RISC-V core 1
  - Compute kernel is launched on the three compute RISC-V cores
- This structure is typical, but not fixed



- Importantly, the kernel is only compiled when the host code is executed (and not when the host code is compiled)
- Here we concentrate on the dataflow kernels, the compute kernel is in the next lecture

Device side

Read runtime arguments set from the host

Read data from DRAM

Push data into circular buffer

```
#include "dataflow_api.h"

#define DATA_TYPE_BYTES 4

void kernel_main() {
    // Load in runtime arguments provided by the host
    uint32_t src0_dram = get_arg_val<uint32_t>(0);
    uint32_t src1_dram = get_arg_val<uint32_t>(1);
    uint32_t buffer_1_addr = get_arg_val<uint32_t>(2);
    uint32_t buffer_2_addr = get_arg_val<uint32_t>(3);
    uint32_t data_size = get_arg_val<uint32_t>(4);

    // NoC coords (x,y) depending on DRAM location on-chip
    uint64_t src0_dram_noc_addr = get_noc_addr_from_bank_id<true>(0, src0_dram);
    uint64_t src1_dram_noc_addr = get_noc_addr_from_bank_id<true>(0, src1_dram);

    // Index of circular buffer 0 (to communicate between this and the other data mover core)
    constexpr uint32_t cb_id_in0 = tt::CBIndex::c_0;

    // Internal L1 buffers to receive data into
    uint32_t * buffer_1 = (uint32_t*) buffer_1_addr;
    uint32_t * buffer_2 = (uint32_t*) buffer_2_addr;

    // Bytes that are read from DDR for the data
    uint32_t bytes_data_size=DATA_TYPE_BYTES * data_size;

    // Read data from DRAM into L1 buffers
    noc_async_read(src0_dram_noc_addr, buffer_1_addr, bytes_data_size);
    noc_async_read(src1_dram_noc_addr, buffer_2_addr, bytes_data_size);
    noc_async_read_barrier();

    // Reserve a single page in the CB
    cb_reserve_back(cb_id_in0, 1);
    // Now we have the page grab the write pointer to this
    uint32_t l1_write_addr_in0 = get_write_ptr(cb_id_in0);
    // Push the page to make it available to consumer of the CB
    cb_push_back(cb_id_in0, 1);
}
```

Device: Runtime arguments and reading DDR-DRAM

- We are showing quite a few things here
  - Reading the five runtime arguments (set by the host)
  - Obtaining the “global” Network on Chip (NoC) address of the DDR memory to read from
  - Reading data from DDR into the L1 SRAM buffer and then blocking for this to complete

```
uint32_t src0_dram = get_arg_val<uint32_t>(0);
uint32_t src1_dram = get_arg_val<uint32_t>(1);
uint32_t buffer_1_addr = get_arg_val<uint32_t>(2);
uint32_t buffer_2_addr = get_arg_val<uint32_t>(3);
uint32_t data_size = get_arg_val<uint32_t>(4);
```

Each runtime argument is read based upon its index

```
uint64_t src0_dram_noc_addr = get_noc_addr_from_bank_id<true>(0, src0_dram);
```

Obtain the global, NoC, address of the memory that we will read from

```
noc_async_read(src0_dram_noc_addr, buffer_1_addr, bytes_data_size);
```

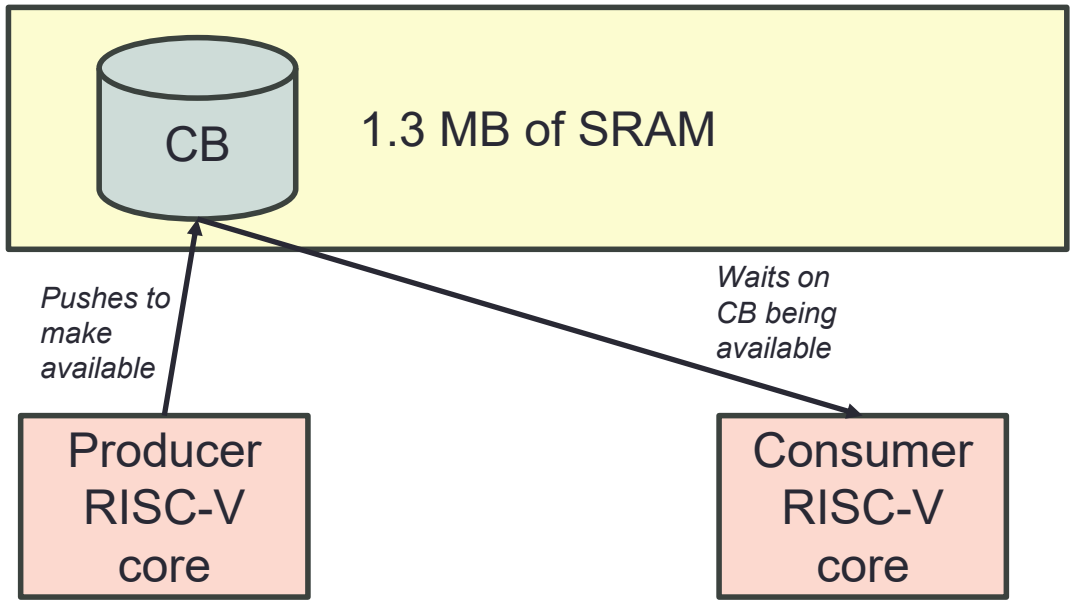
Read from DDR using the NoC address into the L1 SRAM buffer

```
noc_async_read_barrier();
```

Block until this data reading has completed

Device: Circular buffers

- One RISC-V core will produce data to add into the CB, the other will consumer data from the CB.
- The API marshals access to the CB as well as making available the underlying memory




```
cb_reserve_back(cb_id_in0, 1);
uint32_t l1_write_addr_in0 = get_write_ptr(cb_id_in0);
cb_push_back(cb_id_in0, 1);
```

On the producer, will reserve a page in the CB, get the write pointer to this (and then can use that to write to the memory), before pushing it to the CB and making it available.

```
cb_wait_front(cb_id_out0, 1);
uint32_t l1_read_addr = get_read_ptr(cb_id_out0);
cb_pop_front(cb_id_out0, 1);
```

On the consumer, will wait until a page has been pushed into the CB, then will retrieve the read pointer from this and work with the data. Once the consumer has completed it will pop the data so the producer can add some more if required.

Writing results to DDR

```
uint64_t dst_noc_addr = get_noc_addr_from_bank_id<true>(0, dst_addr);
noc_async_write(l1_read_addr, dst_noc_addr, bytes_data_size);
noc_async_write_barrier();
```

- Writing data from L1 to DDR via the NoC is very similar to reading data
  - Is also non-blocking and we need to then wait for completion

- Typically:
  - Reader kernel will reserve page(s) in CB(s), read data from DDR memory into these CB(s) and then push the CB(s) to make them available
  - Compute kernel waits for CBs, then uses these to drive compute on the FPU and reserves page(s) in output CB(s), writing results to these CB(s) and making them available
  - Writer kernel will wait for the output CB(s), then writes the data here to DDR
    - All these run concurrently in a pipelined fashion, working on separate chunks (or tiles)

Printing for debugging

- An extremely helpful feature is the ability to print from kernels on the device to the terminal. This is super helpful during debugging
  - Need to explore the environment variable via *export TT_METAL_DPRINT_CORES=0,0*

```
DPRINT << "Hello " << ENDL();

DPRINT_MATH(DPRINT << "Hello " << ENDL());
```

- *DPRINT* in the kernel will print, furthermore there are function calls such as *DPRINT_MATH* that select which specific core will do the printing. Also calls to print contents of CBs too.
  - As it's a debugging thing, I tend to find *DPRINT* sufficient for most things

Lots more details at https://docs.tenstorrent.com/tt-metal/latest/tt-metalium/tools/kernel_print.html

Building and running

- Device kernels are JIT compiled by Metalium when the host code is launched

```
[user-id@tenstorrent1 ~/tt-tutorial/practical/one/sample_solutions]$ make
……
[user-id@tenstorrent1 ~/tt-tutorial/practical/one/sample_solutions]$ ./ex_one
2025-09-01 18:38:54.566 | info     |   SiliconDriver | Opened PCI device 0; KMD version: 1.29.0; API: 1; IOMMU: disabled (pci_device.cpp:197)
2025-09-01 18:38:54.579 | info     |          Device | Opening user mode device driver (tt_cluster.cpp:192)
2025-09-01 18:38:54.579 | info     |   SiliconDriver | Opened PCI device 0; KMD version: 1.29.0; API: 1; IOMMU: disabled (pci_device.cpp:197)
2025-09-01 18:38:54.590 | info     |   SiliconDriver | Opened PCI device 0; KMD version: 1.29.0; API: 1; IOMMU: disabled (pci_device.cpp:197)
2025-09-01 18:38:54.601 | info     |   SiliconDriver | Harvesting mask for chip 0 is 0x300 (NOC0: 0x300, simulated harvesting mask: 0x0).
(cluster.cpp:295)
2025-09-01 18:38:54.603 | info     |   SiliconDriver | Opened PCI device 0; KMD version: 1.29.0; API: 1; IOMMU: disabled (pci_device.cpp:197)
2025-09-01 18:38:54.699 | info     |   SiliconDriver | Harvesting mask for chip 1 is 0x204 (NOC0: 0x204, simulated harvesting mask: 0x0).
(cluster.cpp:295)
2025-09-01 18:38:54.702 | info     |   SiliconDriver | Opening local chip ids/pci ids: {0}/[0] and remote chip ids {1} (cluster.cpp:157)
2025-09-01 18:38:54.705 | info     |   SiliconDriver | All devices in cluster running firmware version: 255.255.0 (cluster.cpp:138)
2025-09-01 18:38:54.705 | info     |   SiliconDriver | Software version 6.0.0, Ethernet FW version 6.9.0 (Device 0) (cluster.cpp:935)
2025-09-01 18:38:54.705 | info     |   SiliconDriver | Software version 6.0.0, Ethernet FW version 6.9.0 (Device 1) (cluster.cpp:935)
Completed successfully on the device, with 100 elements
2025-09-01 18:38:58.053 | info     |          Device | Closing user mode device drivers (tt_cluster.cpp:383)
```

- The *make* command here builds the *ex_one* host executable but nothing else
- When this host executable is launched, the device kernels are then built
  - Hence we need to provide the source file when calling *CreateKernel* on the host

Next steps: Getting hands on

- We will get you logged into the Tenstorrent host that is in the RISC-V testbed

- Practicals one and two at this point:
  - Practical one explores adding pairs of numbers on one of the data mover RISC-V cores
    - Getting data between the host and the device
    - Allocating L1 buffers on the device
    - Passing runtime arguments to the device
    - Reading and writing data between DDR-DRAM and the L1 buffer on the device
    - Working with L1 buffer on the device

  - Practical two explores using the second RISC-V data mover core to write results to DDR
    - Circular buffers between RISC-V baby cores to enable communication of data between them

All practicals have the exercise itself with the readme walking you through the steps required. There is also a sample solution in-case you get stuck.