ACCELERATING STENCILS ON THE TENSTORRENT GRAYSKULL RISC-V ACCELERATOR

Nick Brown (EPCC) and Ryan Barton (Tenstorrent)

n.brown@epcc.ed.ac.uk

epcc

THE UNIVERSITY OF EDINBURGH

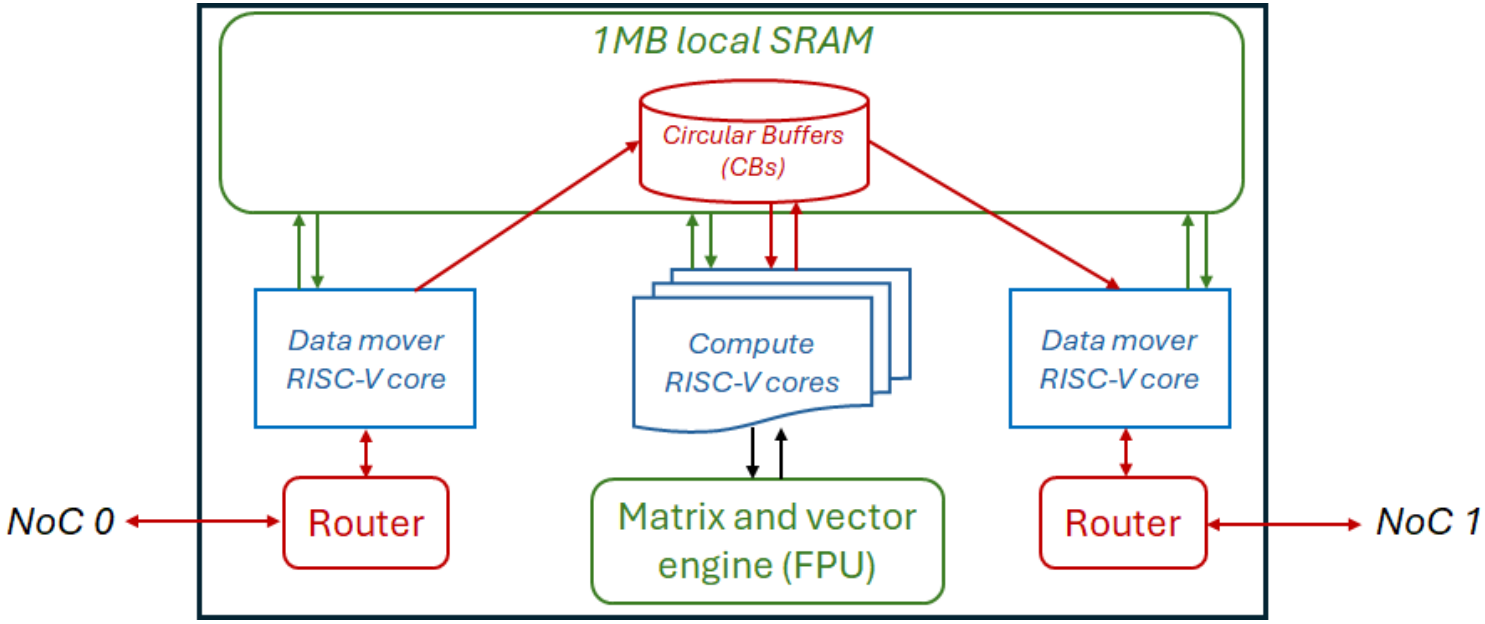
Tenstorrent Grayskull

- PCIe accelerator card, initially designed for AI workloads but can program directly using TT-Metalium API.
  - Hence potential for scientific computing workloads

- In this work we use the e150 which contains 8 GiB of DRAM which is split across eight banks
- Running at 1.2 GHz
- 120 Tensix cores (108 of these are usable as workers)
- Quoted as providing up to 332 FP8 TFLOP/s
- Available for purchase and very affordable

The Tensix core

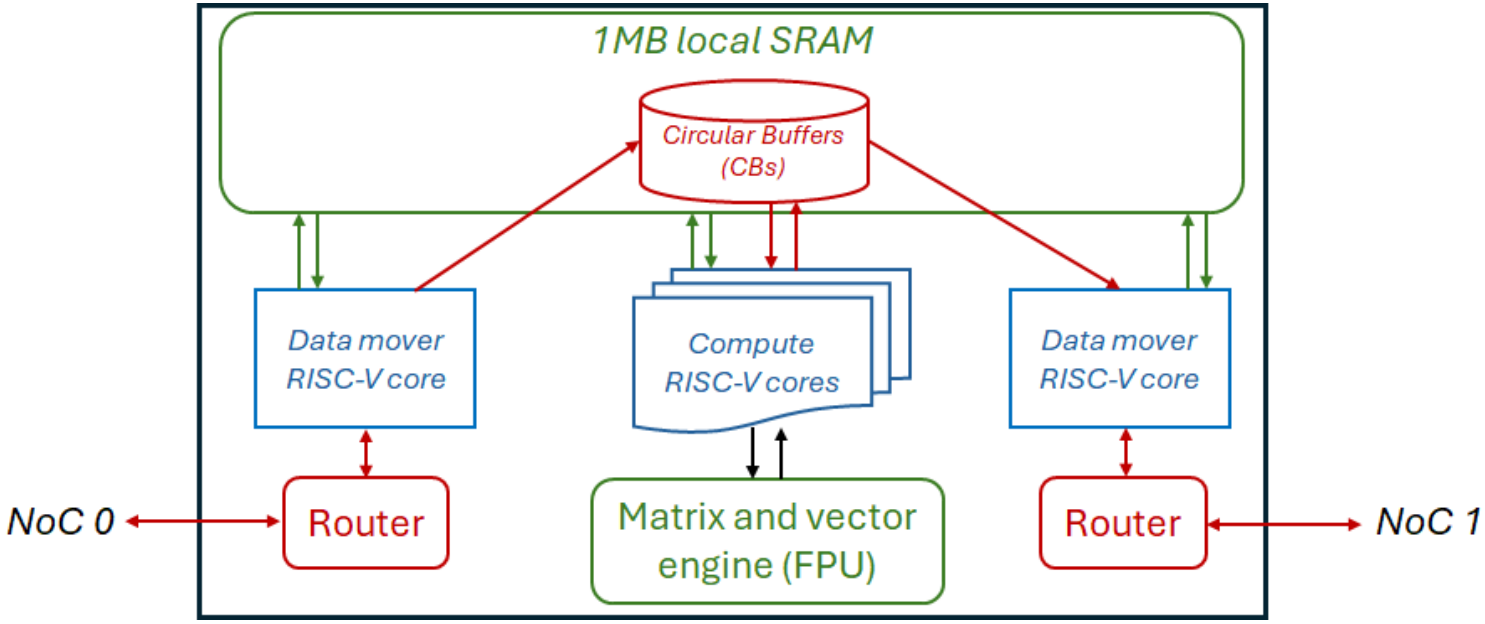
- Contains five *baby* RISC-V cores, two for data movement and three for compute



  - Matrix and vector FPU can be viewed as a 16384-bit wide SIMD unit
    - Element wise addition, subtraction, multiplication and division along with other mathematical and logical operations e.g. calculating squares, logs, trigonometric functions, conditionals and reductions, as well as higher level operations required for ML such as matrix multiplication, ReLU, sigmoid, and transposition
    - The Grayskull supports maximum half precision (both BF16 and FP16) however Wormhole has FP32 support
  - 1MB of SRAM and the cores communicate via Circular Buffers (CBs)
    - Follow producer and consumer model, working in pages of a predefined size
    - Provides pipelining between the cores

The Tensix core

- The programmer writes three kernels
  - Data movement in
  - Data movement out
  - Compute



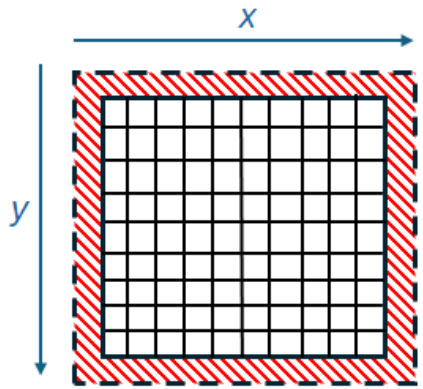
- The three compute cores are for data packing, maths core (feeding FPU) and data unpacking
  - However this distinction is made behind the scenes in the TT-Metal framework, the programmer just sees them as one core and uses the API.
- The compute cores are not created equally (or the compilers aren't)
  - It's possible to do a fair bit on the data movement cores, such as floating point arithmetic but this isn't supported by the compute core. However arithmetic will be (very) slow if it doesn't use the FPU.

epcc
THE UNIVERSITY OF EDINBURGH

Our focus here…..

- Jacobi iterative method for solving LaPlace's equation for diffusion
  - This is the simplest iterative method, and convergence is inferior to others however this simplicity is beneficial here when taking first steps
- Represents a wider class of stencil algorithms
  - Where the calculation depends on neighbouring values
  - Ubiquitous in scientific computing!

```
for all iterations:
    for all grid points i and j:
        unew(i,j) = 0.25 * ( u(i+1,j)+u(i−1,j)+ u(i,j
            +1)+u(i,j−1) )
    swap unew and u
```



- Domain is surrounded by boundary conditions (red) which vary from one side to the other, and the unknowns (white) are calculated from these

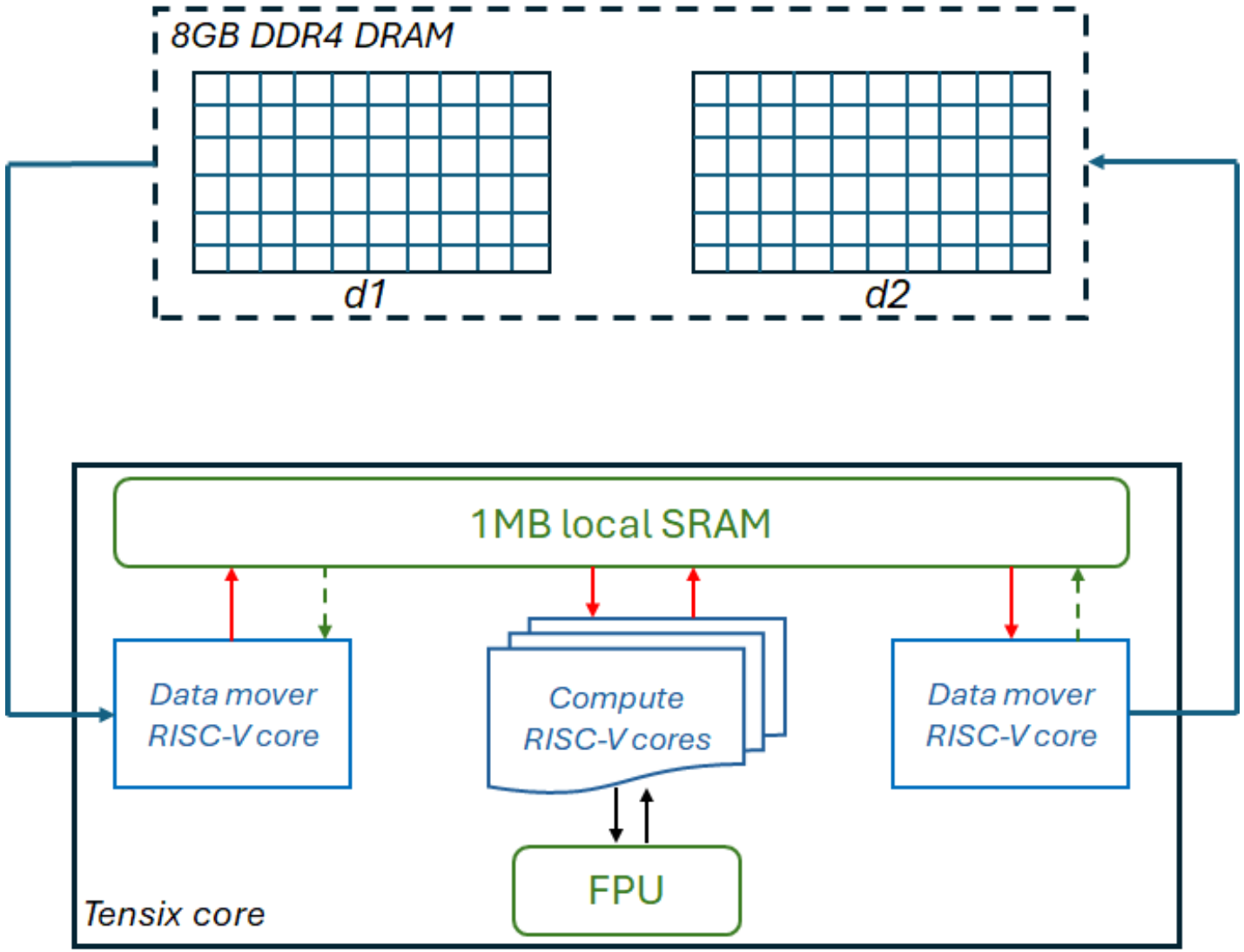
As data movement and compute is separated, and a fair bit of SRAM, can the Tensix provide benefits?

epcc

THE UNIVERSITY OF EDINBURGH

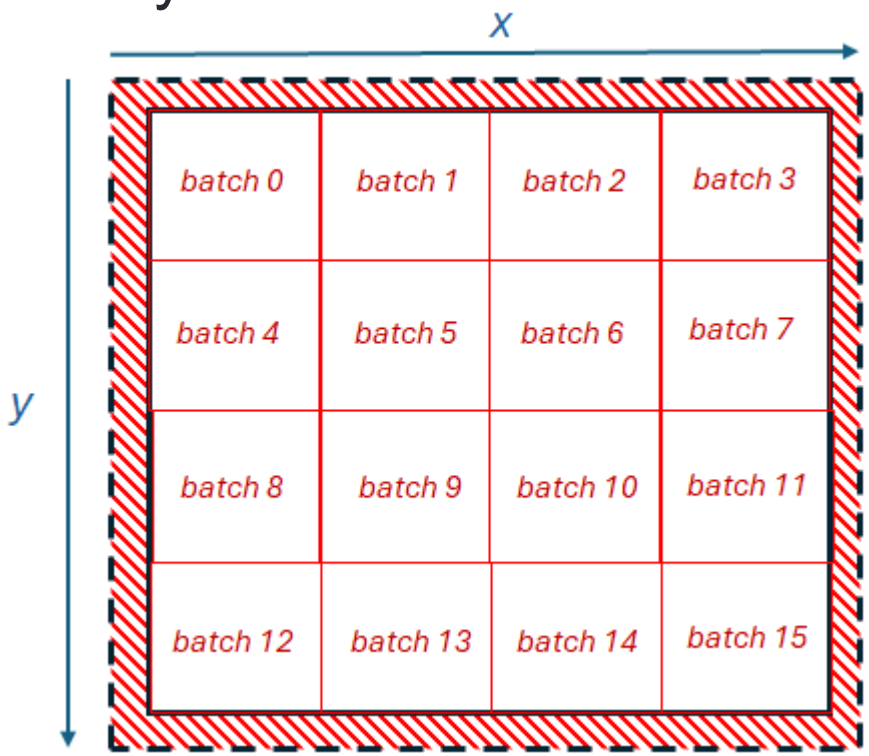
Initial approach

- Data mover (in) reads a tile of data and stores this in CB held in SRAM
- This tile of data is made available to the compute core, fed into the FPU
- Results are provided via CD to data mover (out) core which writes back into DDR
  - Two separate data areas, *d1* and *d2* and swaps between these
- Dashed green line is semaphore, to start next iteration



Compute kernel

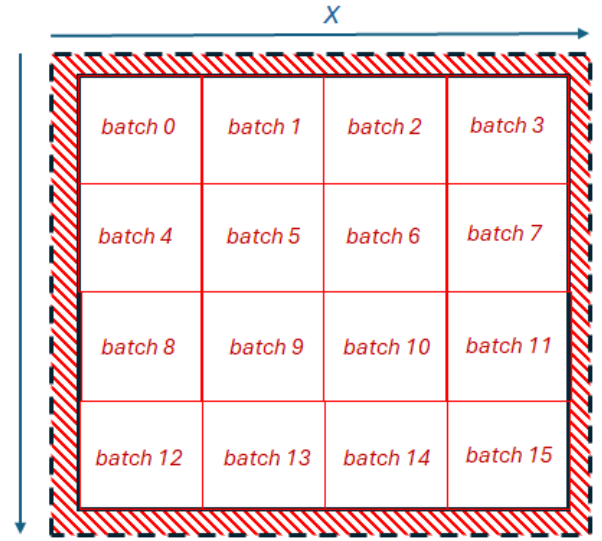
- Decompose domain into tiles of 1024 BF16 elements (32 by 32)
- Require four neighbouring values so have four separate tiles, each offset by -1 or +1 in X or Y
- A fifth tile, all values 0.25 (to average at the end)
- Use an intermediate CB for intermediate results




```
constexpr uint32_t dst0 = 0;

cb_wait_front(cb_in0, 1);
cb_wait_front(cb_in1, 1);
add_tiles(cb_in0, cb_in1, 0, 0, dst0);
cb_pop_front(cb_in1, 1);
cb_pop_front(cb_in0, 1);

cb_reserve_back(cb_intermediate, 1);
pack_tile(dst0, cb_intermediate);
cb_push_back(cb_intermediate, 1);

cb_wait_front(cb_in2, 1);
cb_wait_front(cb_intermediate, 1);
add_tiles(cb_in2, cb_intermediate, 0, 0, dst0);
cb_pop_front(cb_intermediate, 1);
cb_pop_front(cb_in2, 1);

cb_reserve_back(cb_intermediate, 1);
pack_tile(dst0, cb_intermediate);
cb_push_back(cb_intermediate, 1);

// Undertaking the same addition for the third CB

cb_wait_front(cb_intermediate, 1);
mul_tiles(cb_scalar, cb_intermediate, 0, 0, dst0);
cb_pop_front(cb_intermediate, 1);

cb_reserve_back(cb_out0, 1);
pack_tile(dst0, cb_out0);
cb_push_back(cb_out0, 1);
```

Data movement approach

- Require not just 32 by 32 elements in the tile, but also the halos so we can apply the offsets – therefore need 34 by 34, which is 34 non-contiguous reads from DDR each 68 bytes




```
for (uint32_t j=0;j<BATCH_SIZE_IN_Y;j++) {
   std::uint32_t addr_offset=(j*total_size_in_x)+
         batch_offset;
   uint64_t noc_addr = get_noc_addr(noc_x, noc_y,
         ddr_addr+(addr_offset*2));
   noc_async_read(noc_addr, local_buffer+(j*(
         BATCH_SIZE_IN_X)*2), 34*2);
}
noc_async_read_barrier();
// Issue memory copies to four CBs based on the local
      buffer
```

- Writing is simpler as have a 32 by 32 result tile so just write this directly
  - Still results in 32 non-contiguous memory accesses, each 64 bytes
- But this didn't work!
  - Compiled and ran without any errors
  - After the first memory access gave incorrect results, both in terms of data read in and also data written out

epcc | THE UNIVERSITY OF EDINBURGH

(Fixing) Data movement approach

- Through experimentation we found that all DDR memory accesses must be 256 bit aligned
  - The boundary conditions resulted in 68 bytes being read at a time, which is not 256 bit aligned and-so resulted in incorrect values

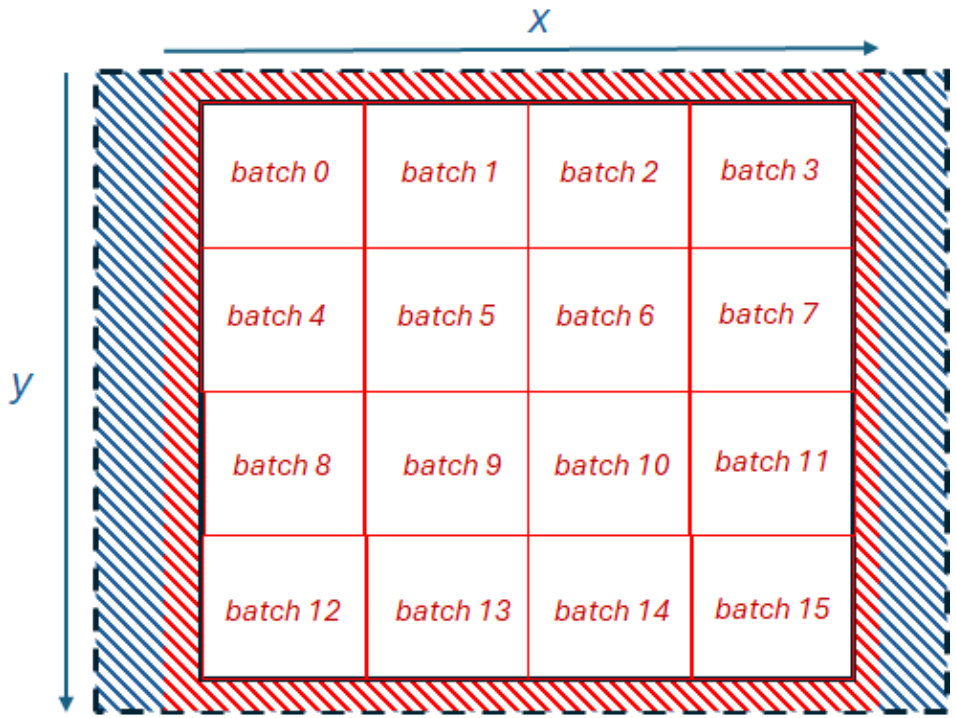
```
std::uint32_t read_data(std::uint32_t address, std::
    uint32_t starting_address, std::uint32_t noc_x, std::
    uint32_t noc_y, std::uint32_t size, std::uint32_t
    buffer_addr) {
    std::uint32_t offset=(address – starting_address) %
        ALIGNMENT;
    std::uint32_t offset_start=address–offset;
    std::uint32_t read_size=size+offset;

    uint64_t noc_addr = get_noc_addr(noc_x, noc_y,
        offset_start);
    noc_async_read(noc_addr, buffer_addr, read_size);
    noc_async_read_barrier();
    return offset;
}
```

- Modified approach so that read is always 256 bit aligned, reading additional preliminary bits if required
- The additional offset is returned to the caller which then can ignore this initial data by starting from the *offset* index in the array

epcc | THE UNIVERSITY OF EDINBURGH

(Fixing) Data movement approach

- Not that simple for data writing however
  - Calculated additional number of elements that had to be written, read these from DDR and packed them into a temporary buffer with the rest of the data to be written
  - Corrupted values – likely because of lack of ordering constraints between read and write



- Did lots of experimentation, and does work if comes from separate locations in a buffer that aren't overwritten
  - Suspect DDR controllers are merging data write operations
- To solve, limit the domain size to a power of two, and allocate an initial 256 bit wide area of memory on the left and right, these are mainly empty but contain the boundary conditions

Initial performance

- Running on one Tensix core using a problem size of 512 by 512 BF16 elements and 10000 iterations

Version	Performance (GPt/s)
CPU single core	1.41
Initial	0.0065
Data write optimised	0.0072
Double buffering	0.0140

Tensix core 217 times slower then CPU core

Tensix core 101 times slower then CPU core

- The *data write optimised* issues the *noc_async_write_barrier* call on the batch level rather than for each access
- The *double buffering* approach blocks for outstanding reads only at the start of the batch, then issues calls to retrieve data for the next batch into the next buffer in local SRAM. Whilst reads are on-going, memory copies are undertaken to copy data into the four CBs from the current batch held in the current buffer, before iterating onto the next batch
  - Theory is we can hide overhead of memory access by doing this

Where is the bottleneck?

- Went through and commented out parts of the code to see the performance that this would deliver

Read	Memcpy	Compute	Write	Performance (GPt/s)
N	N	N	N	7.574
N	N	Y	N	1.387
N	N	N	Y	0.278
Y	N	N	N	0.205
N	Y	N	N	0.014
Y	Y	N	N	0.013

- If data access had zero overhead, then we would get 1.387 GPt/s
  - This is very similar to the CPU core performance and realistically what we should be aiming to obtain here
- Data movement is the bottleneck here
  - Especially our memory copy approach, which reads a specific 34 by 34 tile from memory and then issues *memcpy* to copy this into each of the four CBs with the offsets applied

Exploring data access strategies

- Developed a streaming benchmark which loads 4096 by 4096 32-bit integers from DRAM as quickly as possible by one data mover core, passes these onto the other data mover core which writes them back to DRAM.

Batch size (bytes)	DRAM requests / row	Read Runtime (s)		Write Runtime (s)	
		no sync	sync	no sync	sync
16384	1	0.011	0.011	0.011	0.011
8192	2	0.011	0.011	0.011	0.016
4096	4	0.012	0.013	0.011	0.020
2048	8	0.012	0.020	0.011	0.023
1024	16	0.016	0.034	0.011	0.031
512	32	0.031	0.074	0.011	0.038
256	64	0.039	0.201	0.011	0.053
128	128	0.067	0.327	0.014	0.093
64	256	0.122	0.802	0.027	0.182
32	512	0.238	1.571	0.052	0.360
16	1024	0.470	3.150	0.104	0.718
8	2048	0.916	6.331	0.206	1.436
4	4096	1.761	12.659	0.411	2.873

- What is the overhead in one large memory access vs lots of small ones?
- Overhead of synchronisation per access or per row?
- Lots of accesses are bad after a point, especially when synchronising!
- Read is worse than write

Exploring data access strategies

- Same benchmark but all requests are non-contiguous (work downwards in the Y dimension rather than across in the X dimension)
  - Is there additional overhead in accessing data non-contiguously per request?

Batch size (bytes)	DRAM requests / row	Read Runtime (s)		Write Runtime (s)	
		no sync	sync	no sync	sync
16384	1	0.011	0.011	0.011	0.011
8192	2	0.011	0.011	0.011	0.014
4096	4	0.012	0.012	0.011	0.020
2048	8	0.013	0.021	0.011	0.021
1024	16	0.016	0.042	0.012	0.029
512	32	0.031	0.077	0.017	0.032
256	64	0.042	0.201	0.022	0.052
128	128	0.082	0.340	0.040	0.095
64	256	0.148	0.809	0.074	0.182
32	512	0.275	1.597	0.143	0.361
16	1024	0.544	3.219	0.280	0.721
8	2048	1.081	6.491	0.556	1.441
4	4096	1.969	13.013	0.715	2.882

- Performance degrading between 1024 and 512 byte size.
- Not really an impact of reading non-contigiously
- Suspect bottleneck is not number of NoC requests but the DRAM access width

epcc | THE UNIVERSITY OF EDINBURGH

Exploring data access strategies

- Instead of reading into CB directly we repeated experiment by reading 16384 bytes at a time into a buffer and issued a memory copy.
  - Runtime of 0.106 seconds, which is around ten times slower than when reading directly into the CB. Therefore, clearly we must avoid memory copies by the data movement cores as this results in significant overhead.

Replication factor	Runtime (s)
1	0.011
2	0.017
4	0.033
8	0.055
16	0.098
32	0.185

- Alternative is to issue four reads per tile, each with offset applied
- To understand overhead, repeated experiment (16384 byte width) with replication factor for each
- Results in overhead so ideally avoid memory copy and additional reads if possible

Exploring data access strategies

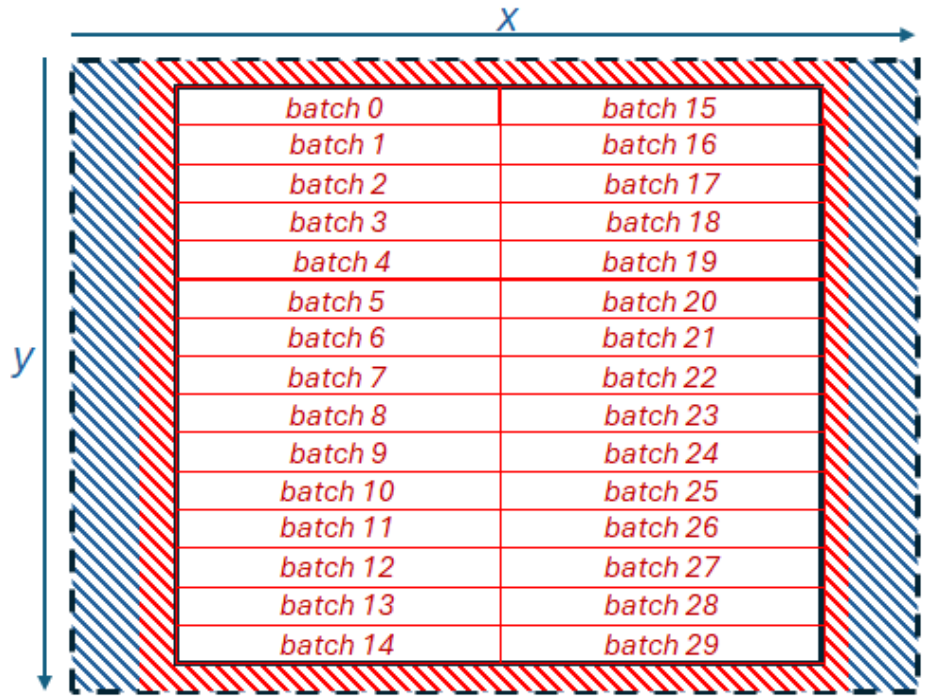
- Have allocated in single bank of DDR to this point, but there are 8 banks and can interleave across them
  - Theory is that this could potentially alleviate some of the memory pressure and improve performance

Page size (bytes)	Runtime (s) with replication factor 0	8	16	32
none	0.010	0.047	0.086	0.162
64K	0.013	0.034	0.050	0.084
32K	0.012	0.030	0.046	0.079
16K	0.013	0.030	0.046	0.079
8K	0.015	0.042	0.072	0.131
4K	0.015	0.075	0.136	0.258
2K	0.021	0.148	0.274	0.527
1K	0.038	0.302	0.565	1.094

- *none* is the existing approach
- When replicating memory accesses there is a benefit in interleaving
  - E.g. For page size of 32KB or 16KB performance with replication factor of 32 is double a single bank
- Conclusion: No real downside of interleaving if page size is sensible

Optimised approach

- Remove memory copies and avoid data access replication
- Moved from tiles of 32 by 32 to one dimension chunks of 1026 elements (1024 values and two halos) to read data contiguously for each tile in one large access.



- Compute kernel requires current batch and previous (upper) and next batch (lower). To avoid duplicate reading of data, we allocate enough memory in the core's local memory buffer for four batches and when working in a column of batches in the Y dimension read batches 0 and 1 and 2 immediately. Then, starting at the first batch (batch zero above) synchronise memory reads immediately, issue a non-blocking read for two batches ahead (batch 2) and make available to the compute cores data that has been read for the current batch

Modifying (hacking) the API to avoid memory copies

- Whilst this results in fewer, contiguous memory accesses, we still need to memory copy between SRAM and CBs
  - Which is very expensive so must be avoided!

- The Metalium framework is open source, so we were able to go in and modify the CB API to avoid this.
  - Added in *cb_set_rd_ptr* function call to arbitrarily point CBs at any memory in SRAM, thus being able to flexibly point to these different chunks and apply the offsets in the current dimension too

Optimised performance and energy efficiency

- 163 times better performance than our initial version!
  - Not quite 1.387 GPt/s that we aimed for, so there is still some data access overhead
- Across the entire e150 we slightly outperform the Xeon Platinum but at around 5 times less energy

Type	Total cores	Cores in Y	Cores in X	Performance (GPt/s)	Energy (Joules)
CPU	1	-	-	1.41	1657
CPU	24	-	-	21.61	588
e150	1	1	1	1.06	2094
e150	2	1	2	2.48	893
e150	4	1	4	2.92	744
e150	8	4	4	7.99	276
e150	32	8	4	9.20	240
e150	64	8	8	12.96	170
e150	72	8	9	17.26	128
e150	108	12	9	22.06	110
e150 x 2	216	24	9	44.12	102
e150 x 4	432	48	9	86.75	108

Conclusions and future work

- RISC-V based PCIe accelerators are very interesting and something that we should be actively exploring
- The Tenstorrent family have significant potential in HPC, however the devil is in the detail here!
  - Need to think carefully how to map algorithm to the architecture and must be very careful to avoid replicated memory accesses and memory copies – which can be difficult
  - More flexibility in the API around CBs could really help here

- Next step is to explore the Wormhole and move to FP32
- Move complex kernels too, not just systolic array but also decomposing the calculation in a grid cell across Tensix cores