Exploring Fast Fourier Transforms on the Tenstorrent Wormhole

Nick Brown[1], Jake Davies[1], and Felix Le Clair[2]

[1] EPCC
[2] Tenstorrent

Today

- The Tenstorrent Wormhole is a highly promising and widely available RISC-V PCIe accelerator (built for machine learning)
- The architecture and programming model are very different to traditional HPC
- The Fast Fourier Transform (FFT) was named the most important numerical algorithm of our time

RISC-V Accelerators

- Try RISC-V for HPC without having to rebuild the whole ecosystem
- Various custom architectures
- Worth exploring what these accelerators are capable of

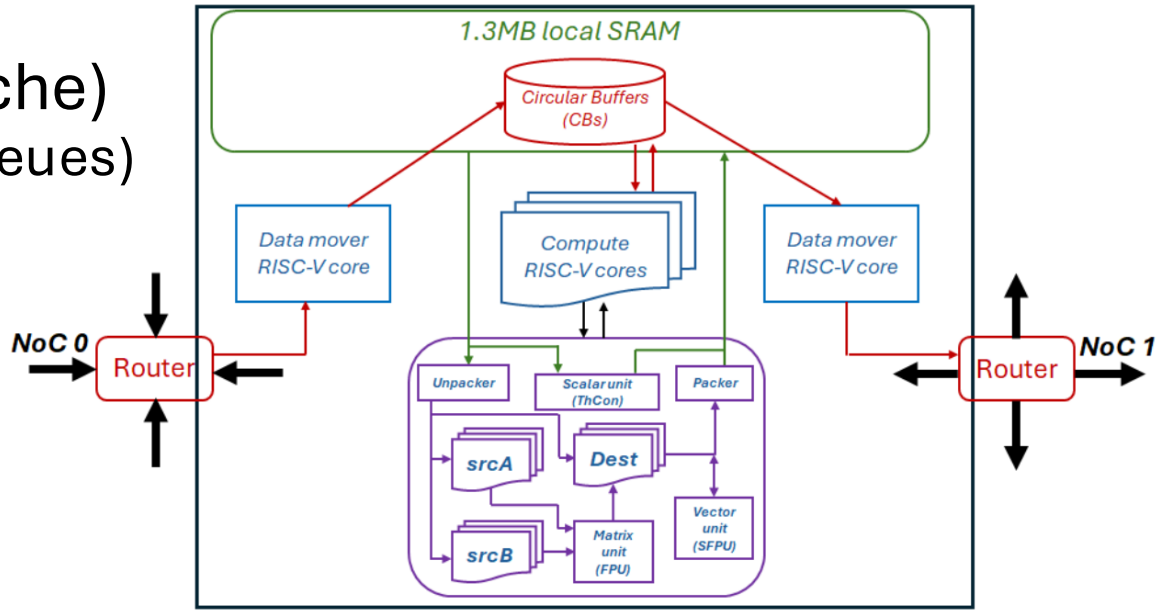
Tenstorrent Wormhole

- PCIe accelerator built upon **Tensix cores**

- Each has 5 “baby” RISC-V cores which drive custom hardware
  - 3 cores handle computing
  - 2 cores handle data movement into/out of the Tensix core

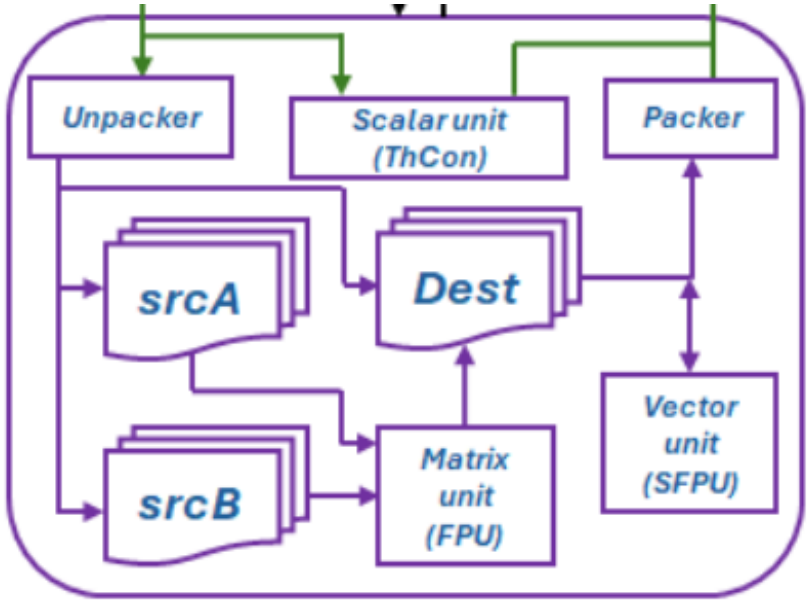
- 1.3MB of fast, local SRAM (explicit cache)
  - Abstracted with circular buffers (FIFO queues)

- Network-on-Chip (NoC)
- Compute engine (up to FP32)
  - Scalar
  - Vector (SIMD)
  - Matrix



Tenstorrent Wormhole

- One compute kernel compiled into unpack, math, pack
- Dest / dst register used for matrix result or vector ops
- No results in baby cores, only pack/unpack
- Matrix unit can perform 2048 multiplies and 2048 adds per cycle

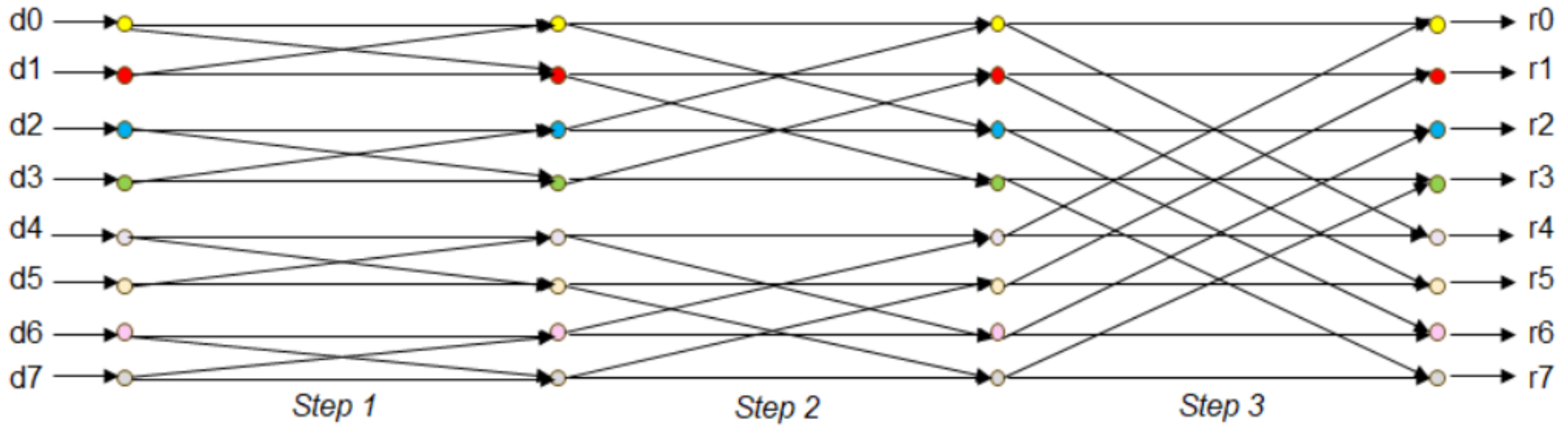


Tenstorrent Wormhole

- Tenstorrent Wormhole n300 has 120 Tensix cores
  - Logically arranged in a grid
- Programmed using TT-Metalium, a C++ framework
  - Direct access to hardware
  - Circular buffers, NoC requests, etc.

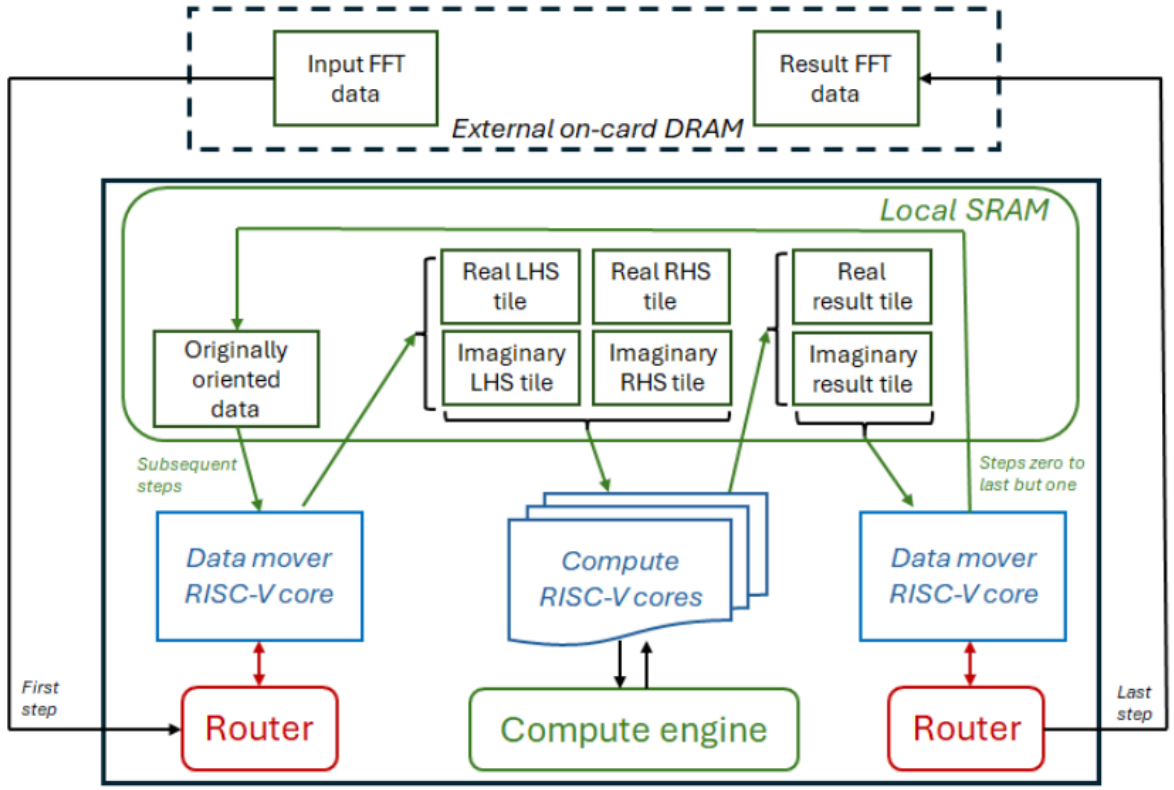
Fast Fourier Transform

- Cooley-Tukey algorithm is the most common
- Compute driven by needing different pairings of complex numbers at each step
  - Data movement significant



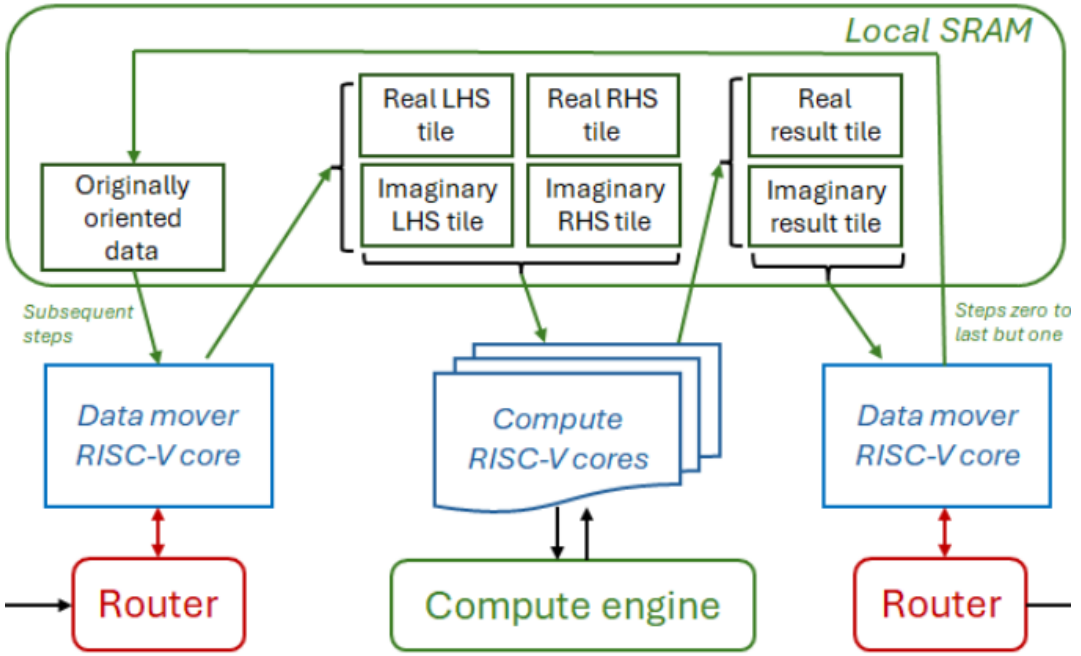
Single Core Kernel

- Reader core issues read
- Reader core reorders data
- Compute engine does computation and stores results in CB
- Writer core returns data order
- Writer core issues write
- Data must be reordered on read and write each step



“Computation”

- FFT is a binary operation on complex numbers
- Challenge: Wormhole doesn’t support complex numbers
- We can handle real and imaginary parts individually
- We make four circular buffers
  - LHS Real
  - RHS Real
  - LHS Imaginary
  - RHS Imaginary
- Makes the code more complicated



“Computation”

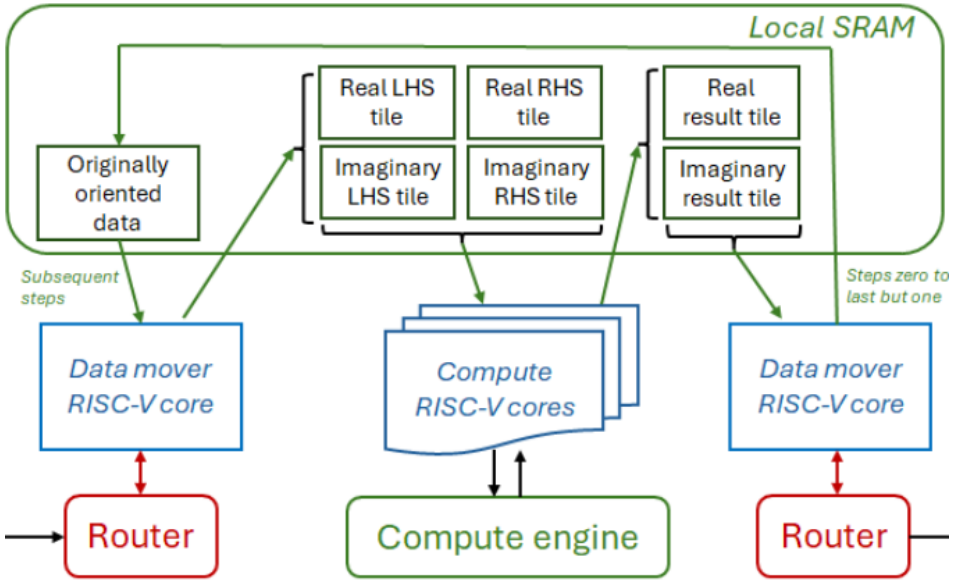
```
for (int step = 0; step <= num_steps; step++) {
    cb_wait_front(cb_data1_r, 1);
    cb_wait_front(cb_data1_i, 1);

    maths_sfpu_op<MUL>(cb_data1_r, cb_twiddle_r, cb_intermediate0);
    maths_sfpu_op<MUL>(cb_data1_i, cb_twiddle_i, cb_intermediate1);
    maths_spfu_op<SUB, true, true>(cb_intermediate0, cb_intermediate1, cb_c0);

    cb_wait_front(cb_data0_r, 1);
    cb_wait_front(cb_data0_i, 1);

    ... // repeat for next constant

    cb_pop_front(cb_data0_r, 1);
    // repeat for other buffers
}
```



Performance

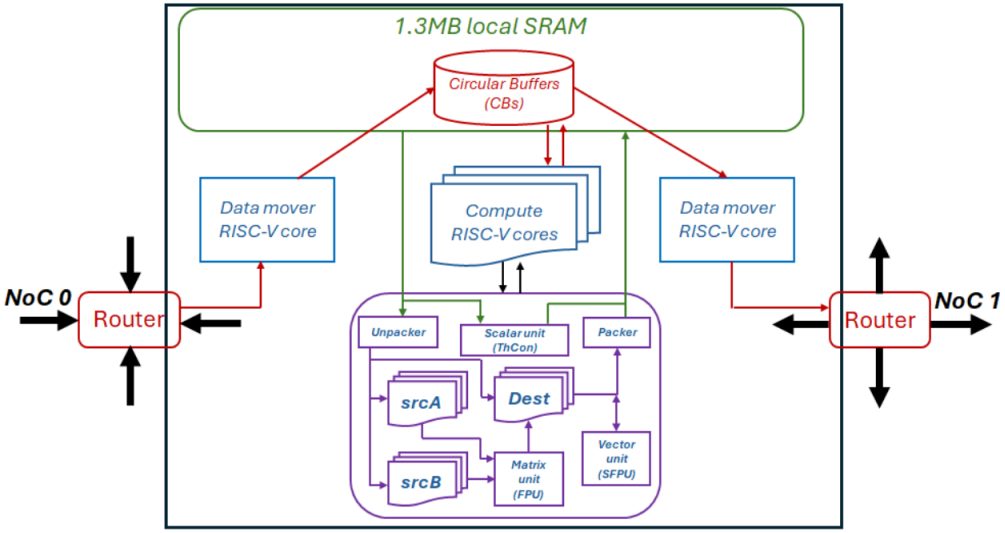
- Compared to a single core of a server-grade CPU[3]
- Initial single-core implementation ~8x slower
- Hard to debug these accelerators
- Disabled components to see where bottleneck is
  - Compute alone is very fast
  - Data reordering was the overhead
- Vector vs Matrix units comparable

External read	Read reorder	Compute	Write reorder	External write	Runtime (ms)
Y	Y	Y	Y	Y	14.4
Y	N	Y	Y	Y	7.3
N	N	Y	Y	Y	7.3
N	Y	Y	N	N	10.5
Y	Y	Y	N	N	10.6
N	N	Y	N	Y	0.9
N	N	Y	N	N	0.9

[3]24-core 8260M Intel Cascade Lake Xeon Platinum CPU

Data Chunking

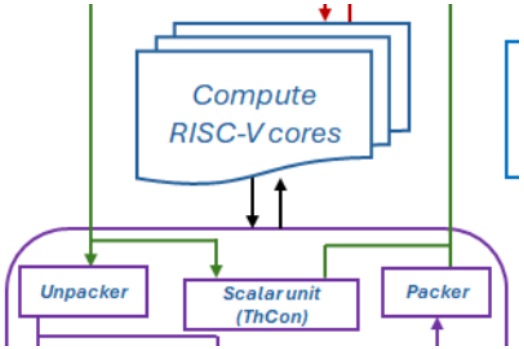
- Initially we placed entire domain into a single circular buffer page
- Simple, but means the baby cores weren't running concurrently
- So we enhanced the code by chunking the domain



Version	Runtime (ms)
Xeon Platinum CPU single core	1.85
Initial	14.39
Chunked	9.38

Data Reordering

- The baby RISC-V cores themselves are simple, they just dispatch instructions to custom hardware
  - Not designed for more involved tasks themselves
- Wrote custom data reordering kernel for the scalar unit (ThCon)
  - Unavailable through the Metalium API
  - Programmed via intrinsics from the Tenstorrent Low Level Kernels library
- Reduces runtime by ~1ms



Version	Runtime (ms)
Xeon Platinum CPU single core	1.85
Initial	14.39
Chunked	9.38
Data copy by ThCon	7.56

Data Reordering

```
uint32_t base_addr = from_addr / 16;
uint32_t addr_offset = from_addr − (base_address ∗ 16);

TT_SETDMAREG(0, LOWER_HALFWORD(addr_offset), 0, LO_16(0));
TT_SETDMAREG(0, UPPER_HALFWORD(addr_offset), 0, HI_16(0));

TT_SETDMAREG(0, LOWER_HALFWORD(base_addr), 0, LO_16(1));
TT_SETDMAREG(0, UPPER_HALFWORD(base_addr), 0, HI_16(1));

TT_LOADIND(p_ind::LD_32bit, LO_16(0), p_ind::INC_4B, 2, 1);
```

Moved the data reordering from RISC-V to compute engine

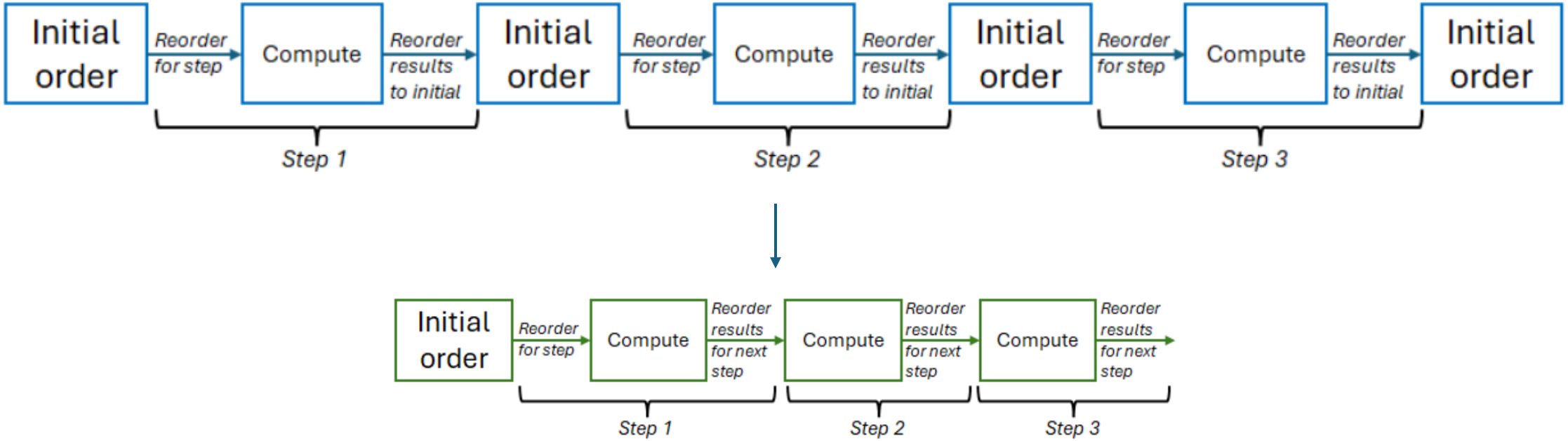
128-Bit Copies

- Initially anticipated true random access
- Tenstorrent suggested we chunk into larger reads
- We are working with FP32, so all data accesses were 32-bit
- However, when reordering data is contiguous (writing into CB tile)
- Unrolled the kernel loop by 4, and performed 128-bit accesses

Version	Runtime (ms)
Xeon Platinum CPU single core	1.85
Initial	14.39
Chunked	9.38
Data copy by ThCon	7.56
128-bit copies	6.61

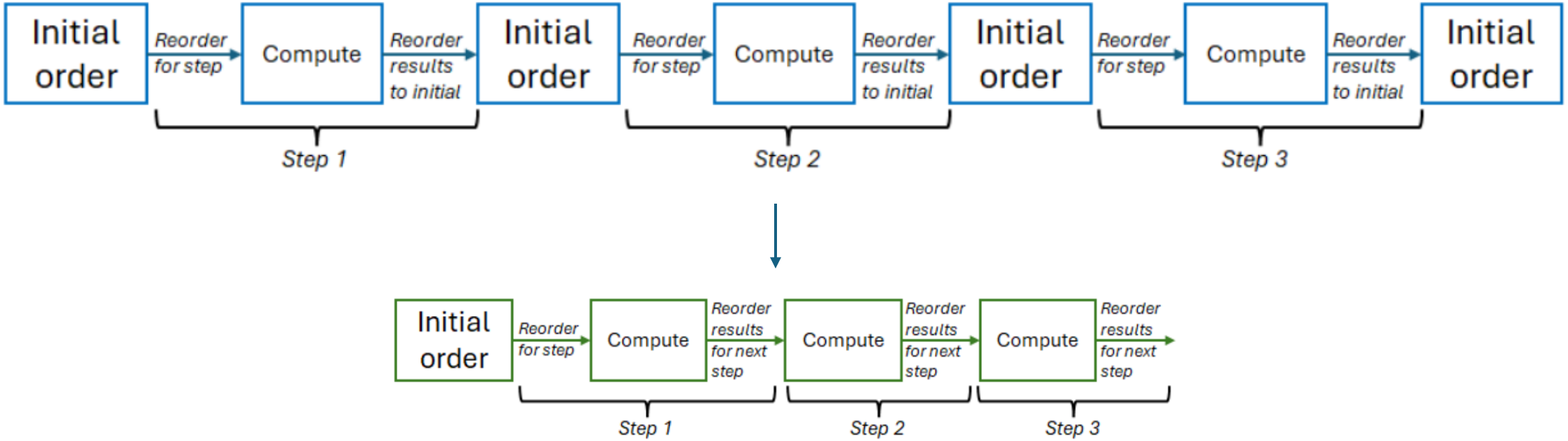
Single Data Copy

- Currently doing two data reorderings per step
- Simplest to code, but still expensive
- Instead, prepare the order for the next step



Single Data Copy

- Changed the code and to support this and wouldn't compile
- Adjusted linker script to add to the bss section of our program
- Most complicated compute kernel?



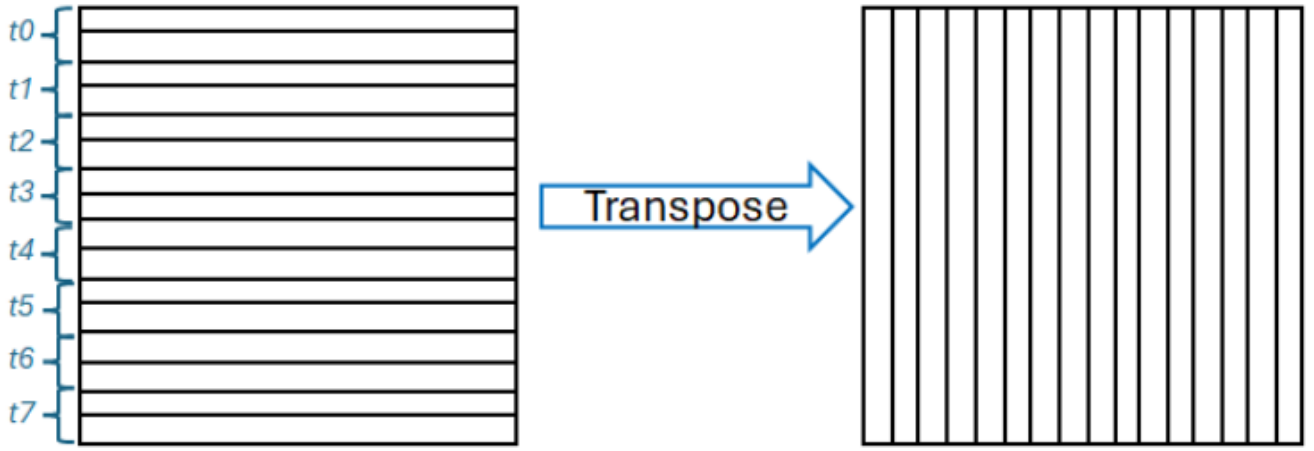
Single Data Copy

- We saw improvement, but expected more
- Upon investigation we found that all data accesses were now non-contiguous, and so 32 rather than 128-bits wide
- We had increased number of individual accesses with more narrow accesses, whereas before had more wider accesses

Version	Runtime (ms)
Xeon Platinum CPU single core	1.85
Initial	14.39
Chunked	9.38
Data copy by ThCon	7.56
128-bit copies	6.61
Single data copy	5.31

2D FFT

- Then looked at scaling up across multiple Tensix cores
- 2D FFT is a 1D FFT across rows, and then a matrix transposition, then another 1D FFT
  - All-to-all communication across cores



Implementation

- Made our FFT code work across several locally held rows
- Leveraged the provided multicore transpose operation[4]
- 1024 x 1024 (even) problem size uses 64 of the 120 Tensix cores
  - FFT must be of size 2^n, 64 cores gives an even workload distribution
- Each with 16 local rows

[4]comes from the higher level tt-nn library

Results

- 64-Tensix cores vs 24-core CPU (OpenMP)
- CPU is 2.3x faster
- But Wormhole has 6x less power draw
- Wormhole is 3.6x more energy efficient

Version	Number of cores	Runtime (ms)	Average Power (Watts)	Energy usage (J)
Xeon Platinum CPU	24	10.24	353	3.62
Wormhole n300	64	23.56	42	0.99

Future Opportunities

- Larger problem size integrating DRAM would be interesting
  - Requires uneven distribution
- Supporting uneven transpositions would allow to use all 120 Tensix cores, possibly catching up to CPU performance
- Reworking one-copy data reordering: contiguous 128-bit accesses

Summary

- Still a challenge is understanding how to adapt algorithms and tooling to best suit general purpose HPC, rather than ML
- Great potential for Tenstorrent technology to benefit HPC workloads with its energy efficient nature