Performance Modeling in DAREXA-F

Compression & Lower Precision Computing

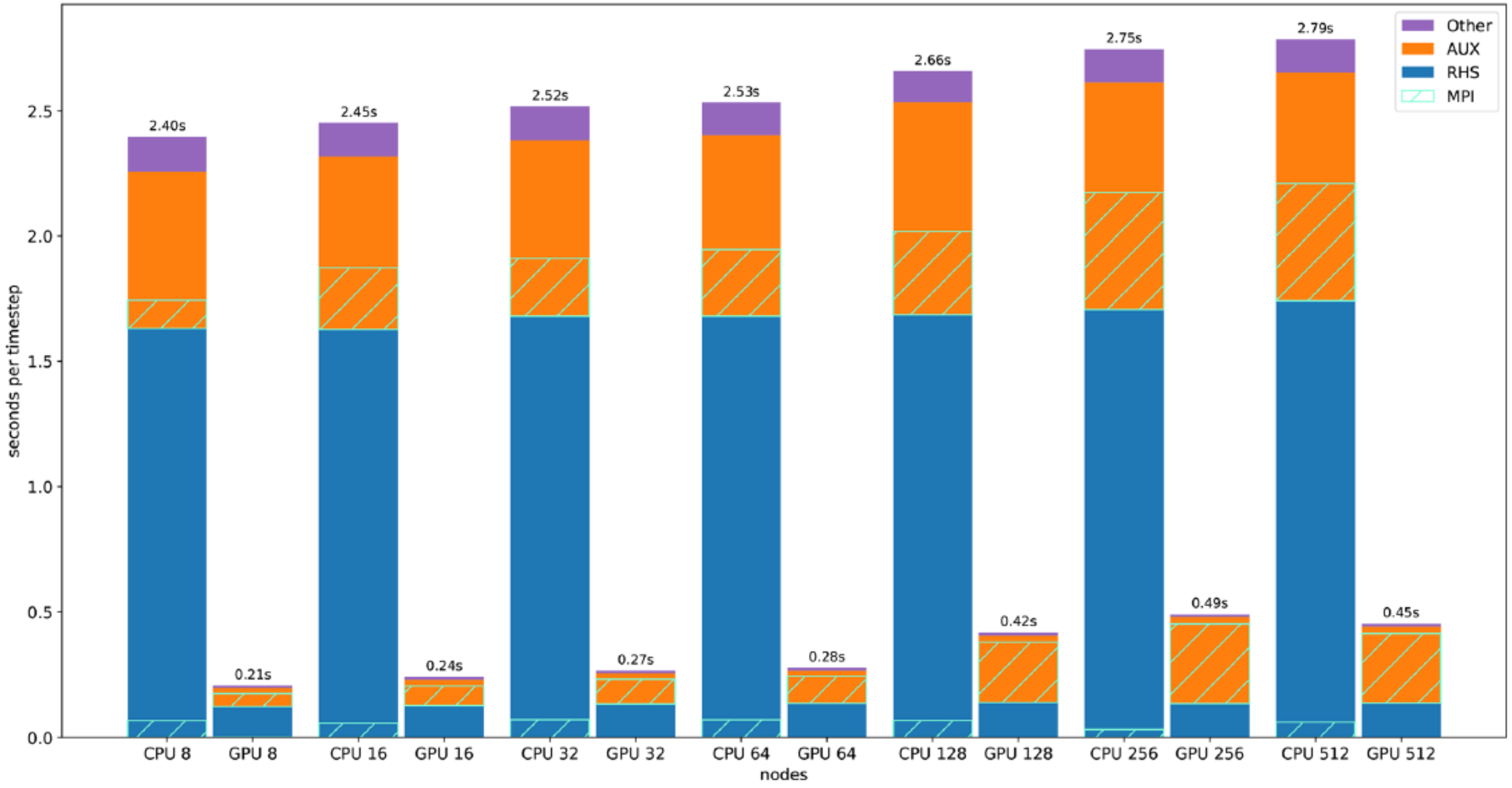
J. Laukemann, G. Wellein

Erlangen National High Performance Computing Center (NHR@FAU)

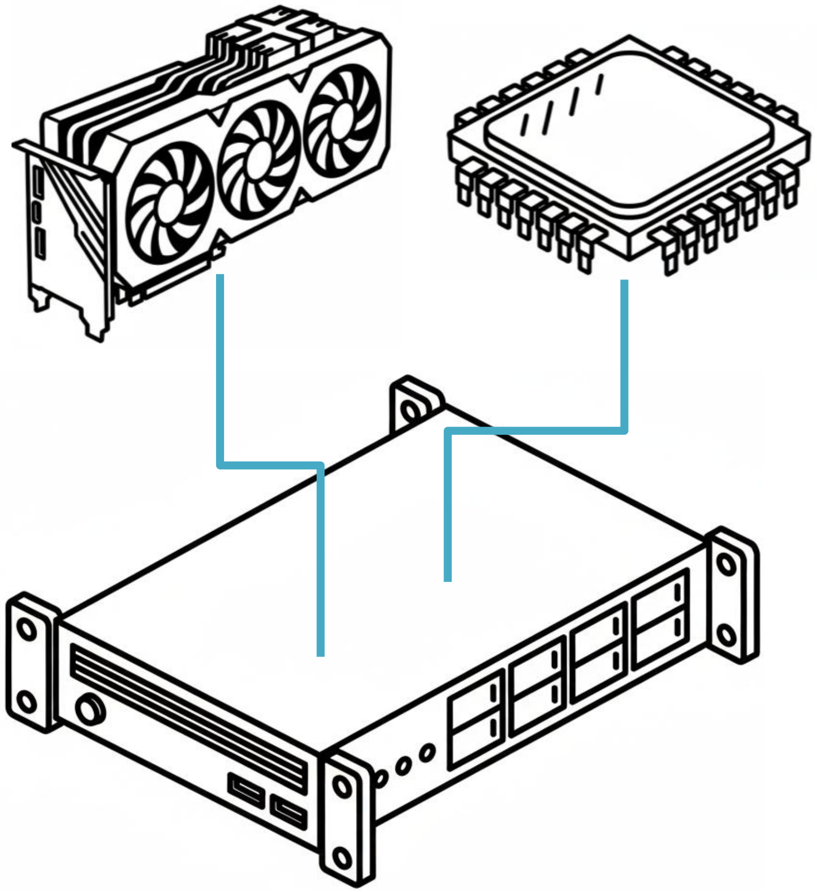
Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg

Motivation – lower precision & compression

weak scaling to 512 Summit nodes (CPU vs. GPU)



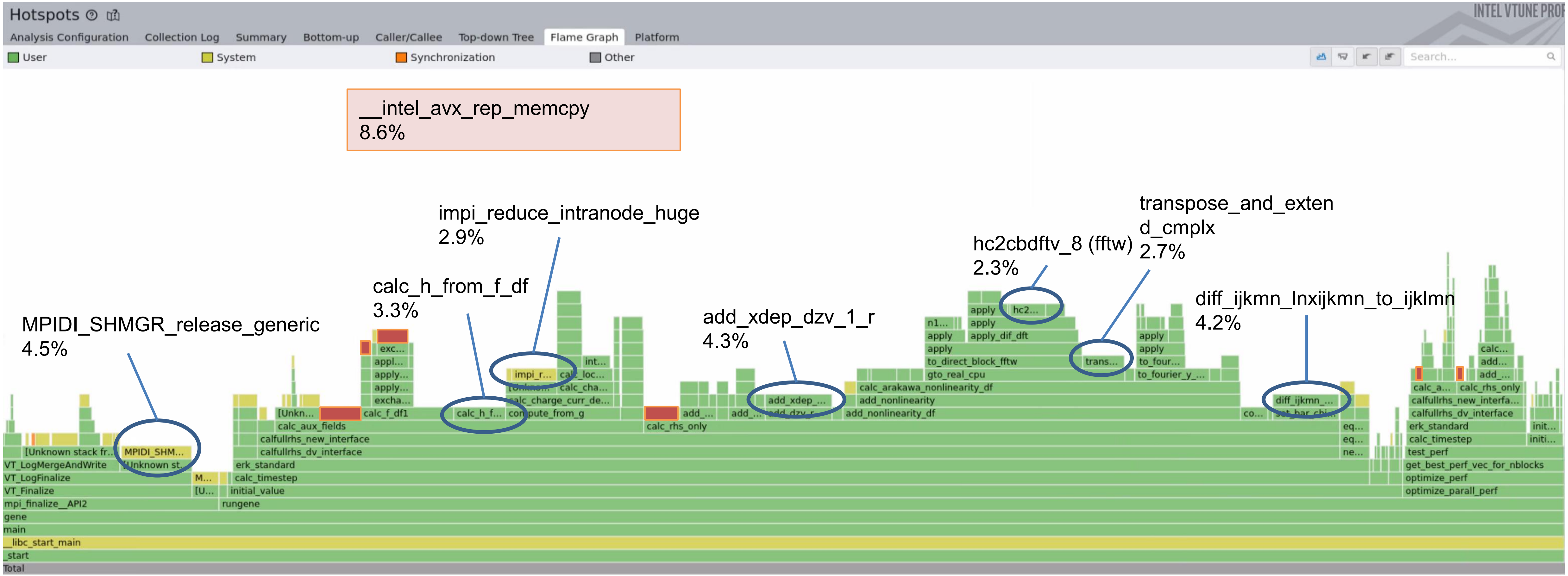
[Germaschewski et al., PoP 2021]



What about node-level performance?

Today: Pitfalls and experiences during DAREXA-F

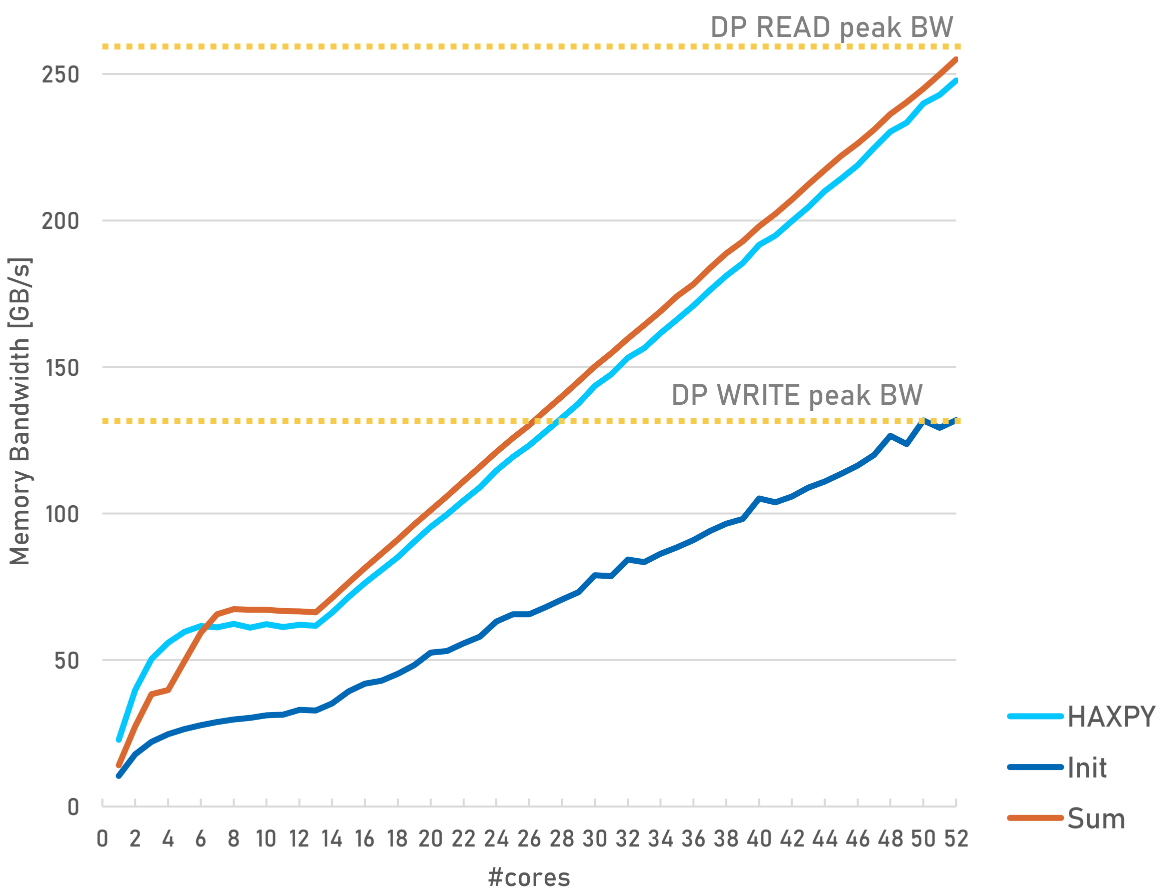
Node level performance CPU (single node)



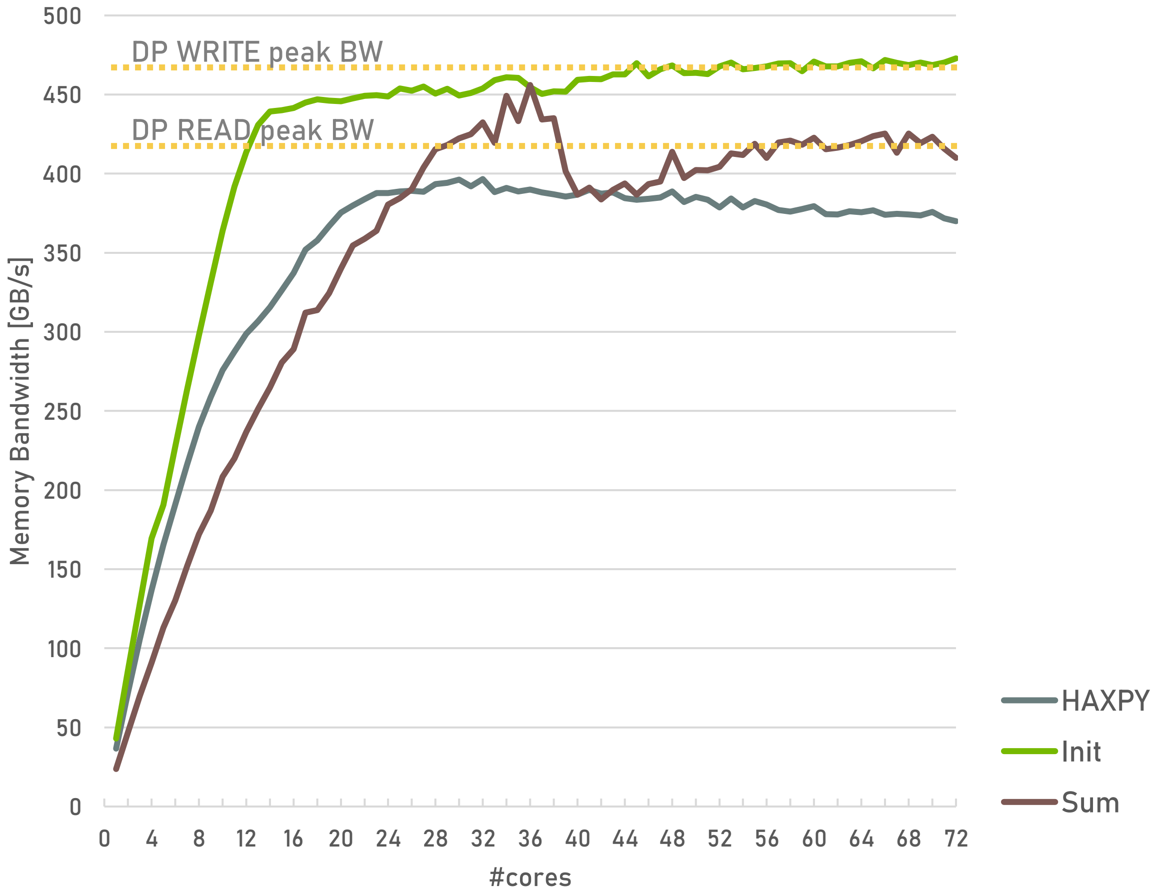
→ No free lunch

Node level performance – FP16-readiness

Intel Sapphire Rapids



NVIDIA Grace Superchip



FP16 analysis - GTENSOR

gtensor

CPU CI, CUDA/HIP build CI passing | SYCL CI passing

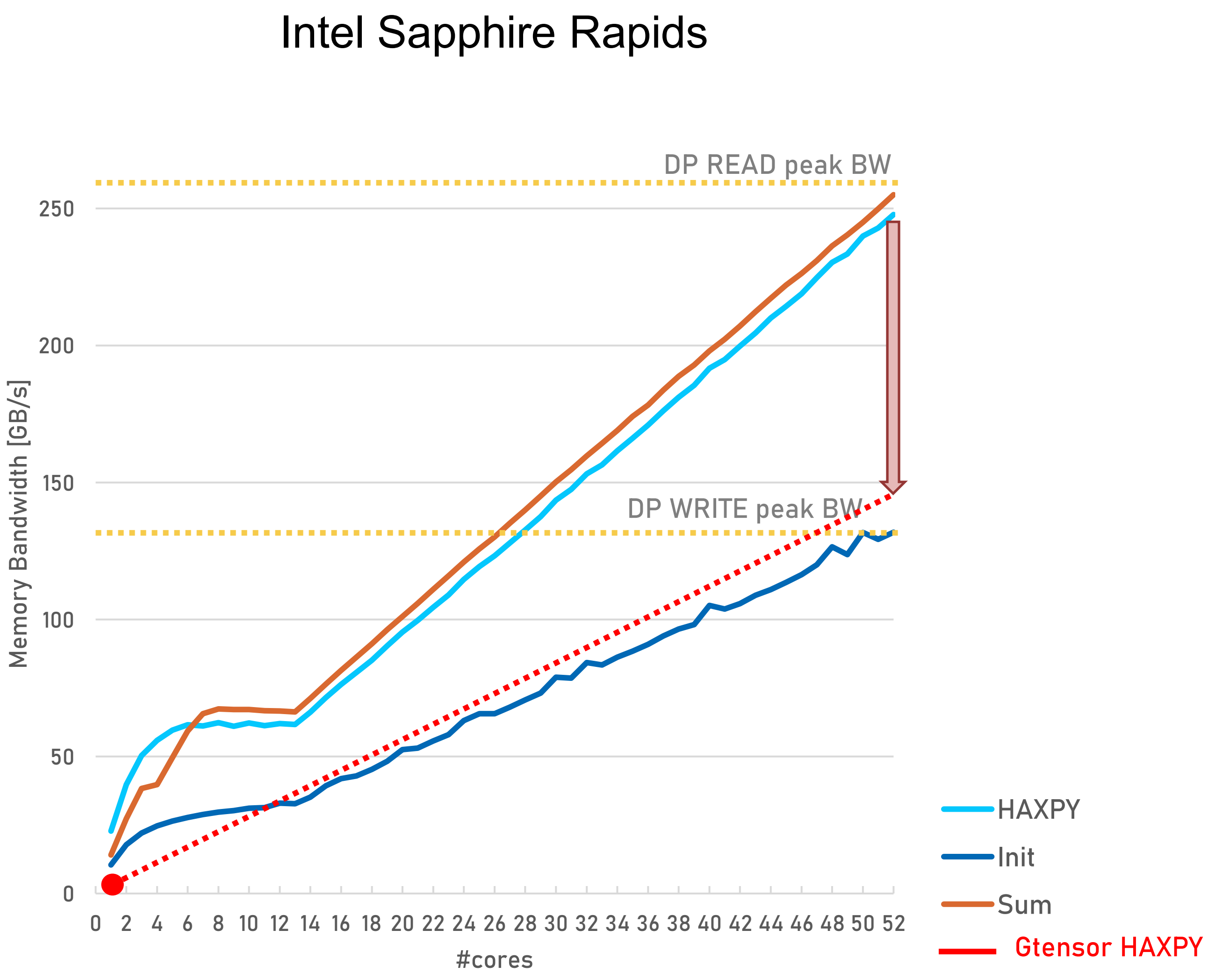
multi-dimensional arrays and array views, with easy interoperability with Fortran and thrust for GPU and CPU

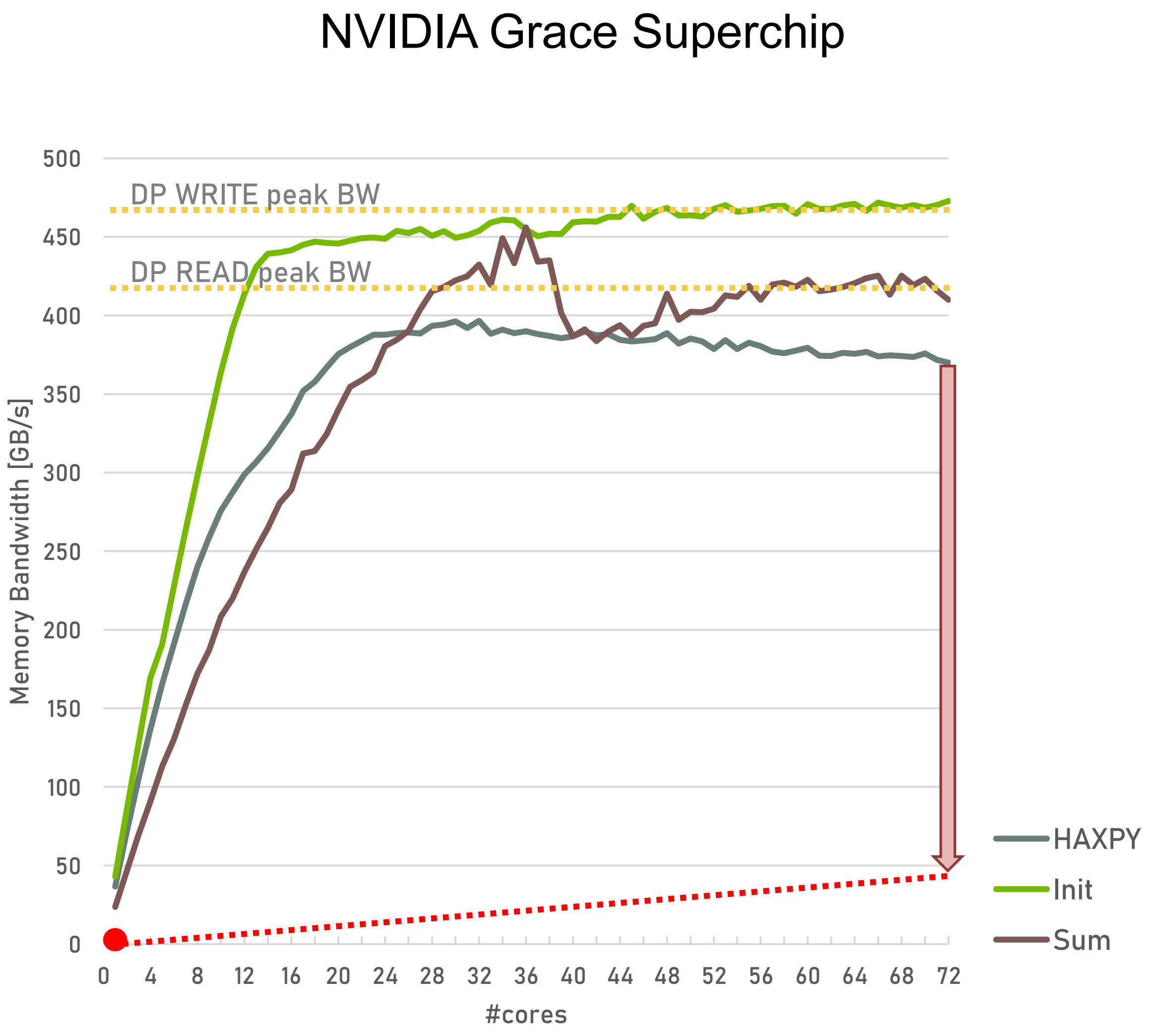
written in C++

```
// AXPY example GTENSOR

// Declaration + auto-init device
gt::gtensor<storage_type, 1, gt::space::device> xx(gt::shape(N), 0.6123);
gt::gtensor<storage_type, 1, gt::space::device> yy(gt::shape(N), 0.3877);

auto k_xx = xx.to_kernel();
auto k_yy = yy.to_kernel();

// Implicit benchmark device
for(int r = 0; r < REPS; ++r)
    yy = yy + a * xx;
```

HAXPY measurements: (single core, clang17):	Intel SPR	NVIDIA Grace
	2.86 GB/s	0.62 GB/s

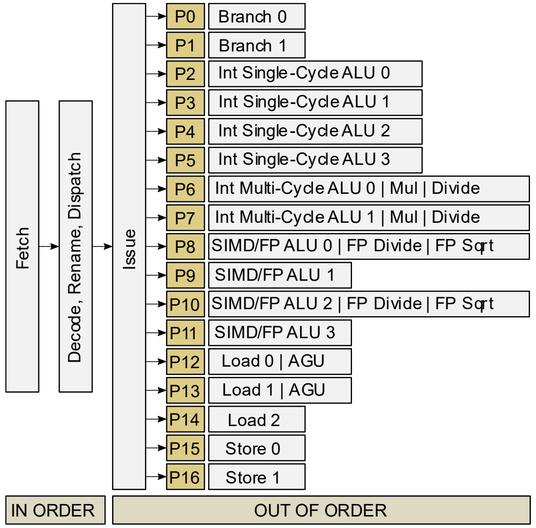
Node level performance – FP16-readiness





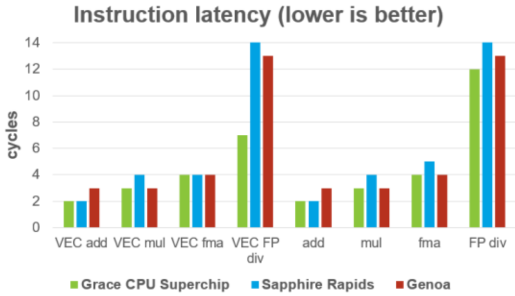
Building an in-core performance model

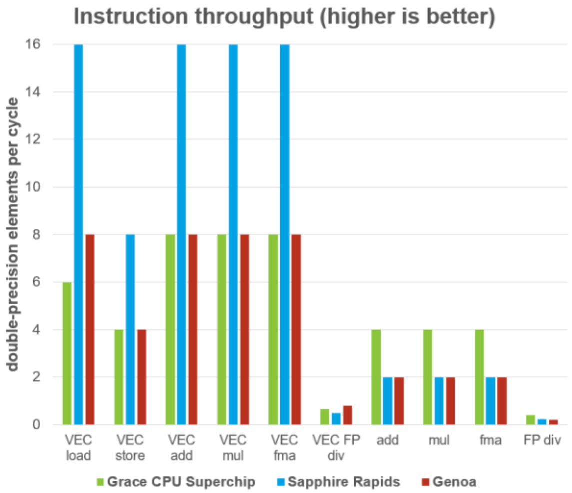
- port model
  - abstraction of superscalar and OoO abilities



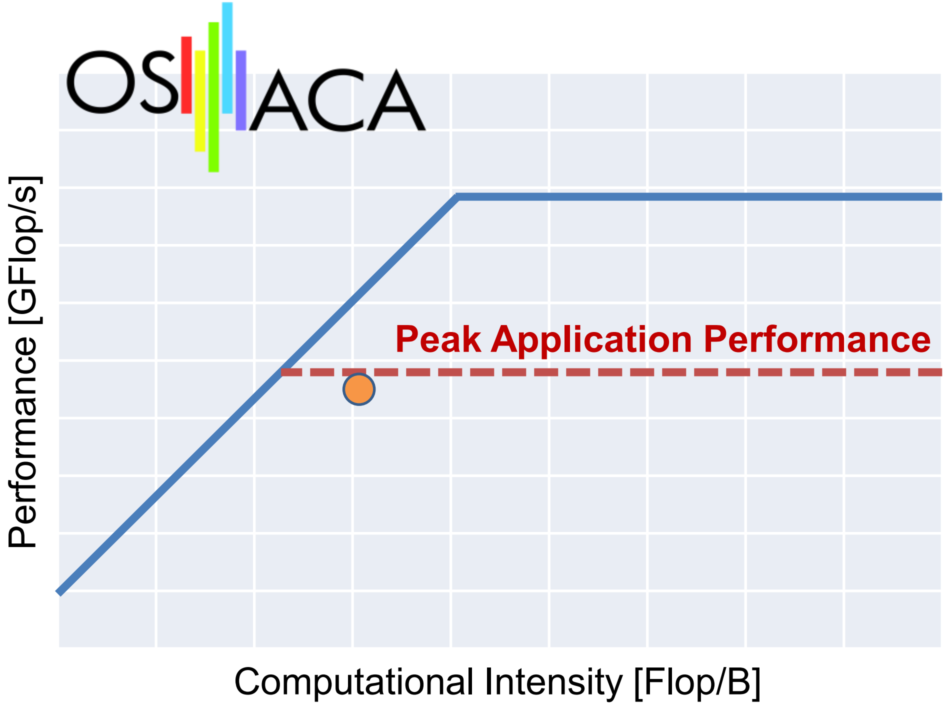
- ASM benchmarks

	GCS	SPR	Genoa
#ports	17	12	13
SIMD-width	16 B	64 B	32 B
INT units	6	5	4
FP/vector units	4	3	4





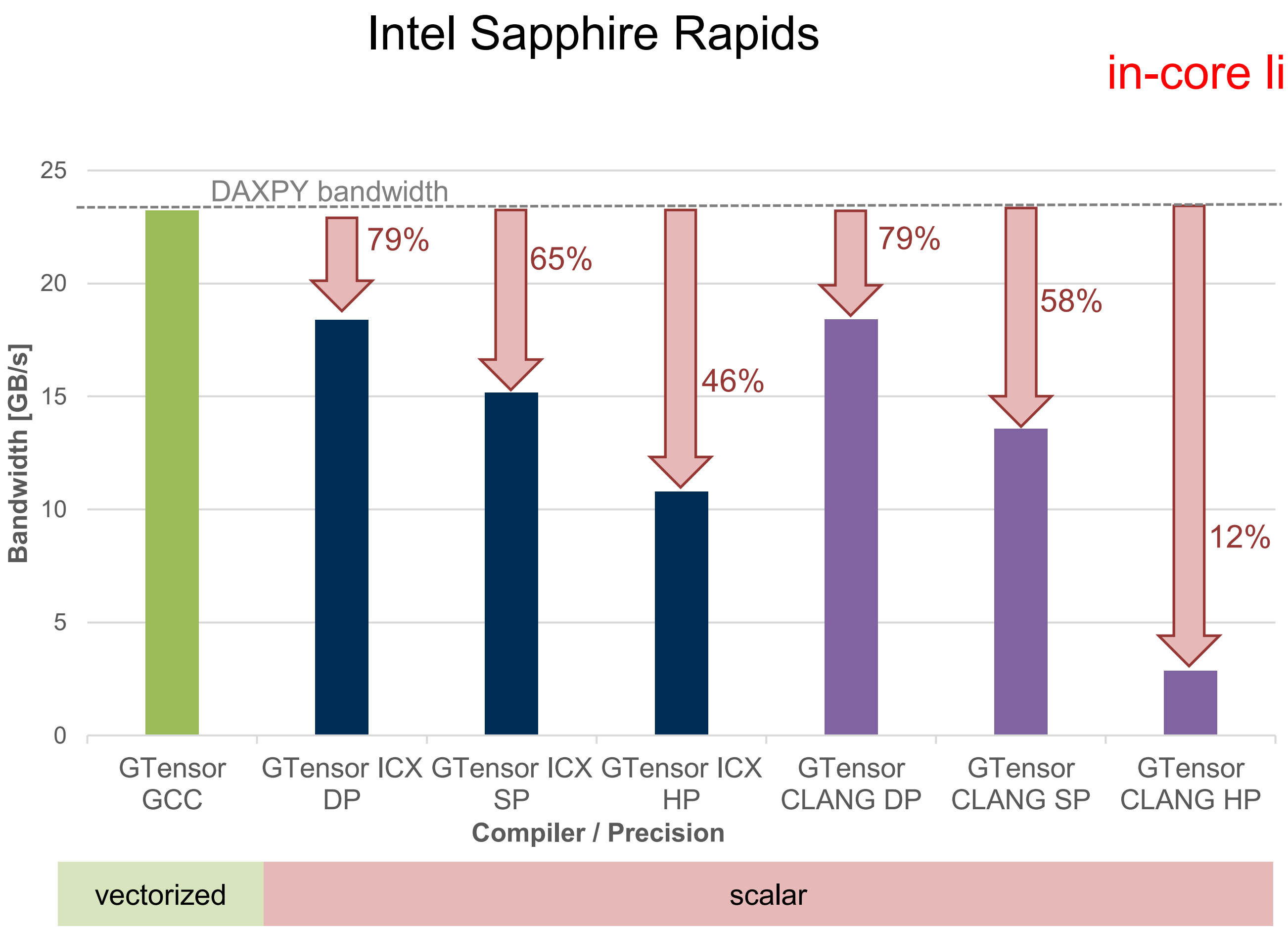
- Performance model
  - throughput
  - latency & dependencies
  - port occupation

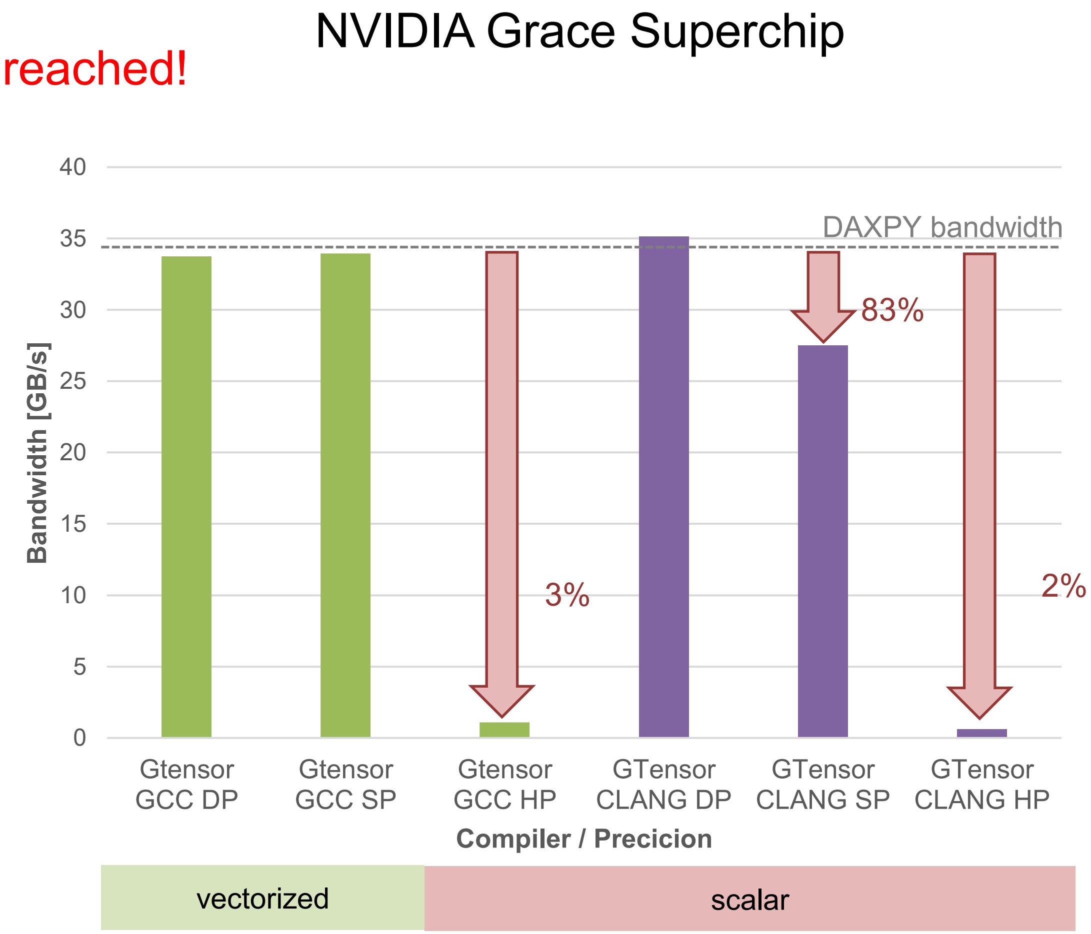


[Laukemann et al., PMBS 2024, PARCO 2025]

CPU performance – FP16 support

in-core limit reached!





! Vectorization much more important

! FP16 instr. slower than DP instr.

CPU performance - FP16 support

- Bottlenecks can easily shift when migrating from DP to FP16/BFloat16
  - Harder to saturate main memory
  - More instructions per Byte
    - → Fetching/Decoding/Issuing/Retirement limit easier reachable
  - 4x more elements per vector does NOT automatically give 4x speedup

- Compiler “mistakes” can have more impact
  - bad inlining
  - un-aligned data access
  - ...

CPU performance – extracted GENE kernels – 1/2

```
subroutine add_xdep_dzv_1_r(h_6d, rhs_6d)
  complex, dimension(:,:,:,:,:,:), contiguous, &
           pointer, intent(in) :: h_6d, rhs_6d
  integer :: j, k, l, m, n, bl(6), bu(6)

  bl = lbound(rhs_6d); bu = ubound(rhs_6d)

  // update RHS
  do concurrent(n=bl(6):bu(6), m=bl(5):bu(5), l=bl(4)+2:bu(4)-2, &
     &        k=bl(3)+2:bu(3)-2, j=bl(2):bu(2))
    rhs_6d(:,j,k,l,m,n) = rhs_6d(:,j,k,l,m,n) + &
         dh_pref(:, 1,k,l,m,n)*h_6d(:,j,k  ,l-2,m,n) + &
         dh_pref(:, 2,k,l,m,n)*h_6d(:,j,k-1,l-1,m,n) + &
         dh_pref(:, 3,k,l,m,n)*h_6d(:,j,k  ,l-1,m,n) + &
         dh_pref(:, 4,k,l,m,n)*h_6d(:,j,k+1,l-1,m,n) + &
         dh_pref(:, 5,k,l,m,n)*h_6d(:,j,k-2,l  ,m,n) + &
         dh_pref(:, 6,k,l,m,n)*h_6d(:,j,k-1,l  ,m,n) + &
         dh_pref(:, 7,k,l,m,n)*h_6d(:,j,k  ,l  ,m,n) + &
         dh_pref(:, 8,k,l,m,n)*h_6d(:,j,k+1,l  ,m,n) + &
         dh_pref(:, 9,k,l,m,n)*h_6d(:,j,k+2,l  ,m,n) + &
         dh_pref(:,10,k,l,m,n)*h_6d(:,j,k-1,l+1,m,n) + &
         dh_pref(:,11,k,l,m,n)*h_6d(:,j,k  ,l+1,m,n) + &
         dh_pref(:,12,k,l,m,n)*h_6d(:,j,k+1,l+1,m,n) + &
         dh_pref(:,13,k,l,m,n)*h_6d(:,j,k  ,l+2,m,n)
end do
```

- Highly in-core bound (FMA, FMS) with long critical paths
  - AVX prediction vs measurement: 50-55 cy/it vs 57 cy/it
  - AVX-512 pred. vs meas.: 54-58 cy/it vs 83 cy/it
- Possible optimizations through
  - use AVX-512 on default (📈1.38x)
  - change in data layout (RIRI → RRII)
  - larger grid (cur: 65536) for more overlap
  - intrinsics (avoid permutations)

CPU performance – extracted GENE kernels – 2/2

```
subroutine add_i_sten_slice_i_global (rhs_slice,klmn,&
                                       factor,stencil,stencil_lower)
  complex, intent(in) :: klmn
  complex, intent(inout), dimension(:,:) :: rhs_slice
  real, intent(in) :: factor, stencil(:)
  integer, intent(in), optional :: stencil_lower
  integer :: sb(2),sten,j,i1,i2,j1,j2,s_shift
  complex, dimension(:,:), pointer :: dist_func_slice

  sb = get_stencil_range(size(stencil),stencil_lower)
  s_shift = 1-sb(1)
  dist_func_slice => get_2D_block_slice(klmn)

  // stencil update
  do j=j1,j2
    do sten=sb(1), sb(2)
      rhs_slice(:,j-j1+1) = rhs_slice(:,j-j1+1) &
        *factor*stencil(s_shift+sten) &
        *dist_func_slice(i1+sten:i2+sten,j)
    end do
  end do
```

- Highly in-core bound (FMA, FMS) for cache-sized working sets
  - AVX prediction vs measurement: 24 cy/it vs 24 cy/it
  - AVX-512 pred. vs meas.: 48 cy/it vs 49 cy/it
- Possible optimizations through
  - use AVX-512 on default (2x)
  - change in data layout (RIRI → RRII)
  - intrinsics (avoid permutations)
  - blocking (local klmn)

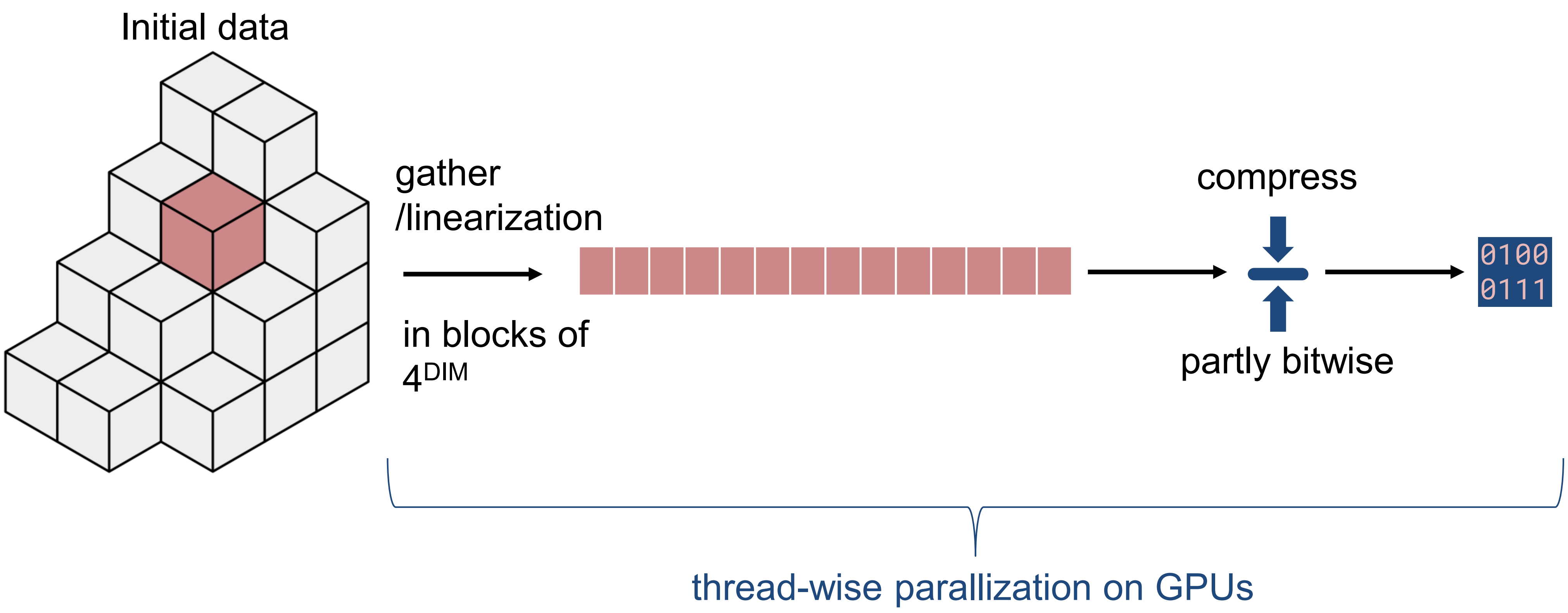
GPU performance – ZFP compression

- **ZFP: Lossy compression** library for **multidimensional arrays**
- Efficient number format for **fixed-size blocks** of **real values**
- **Streaming (de)compression** via OpenMP and **CUDA**

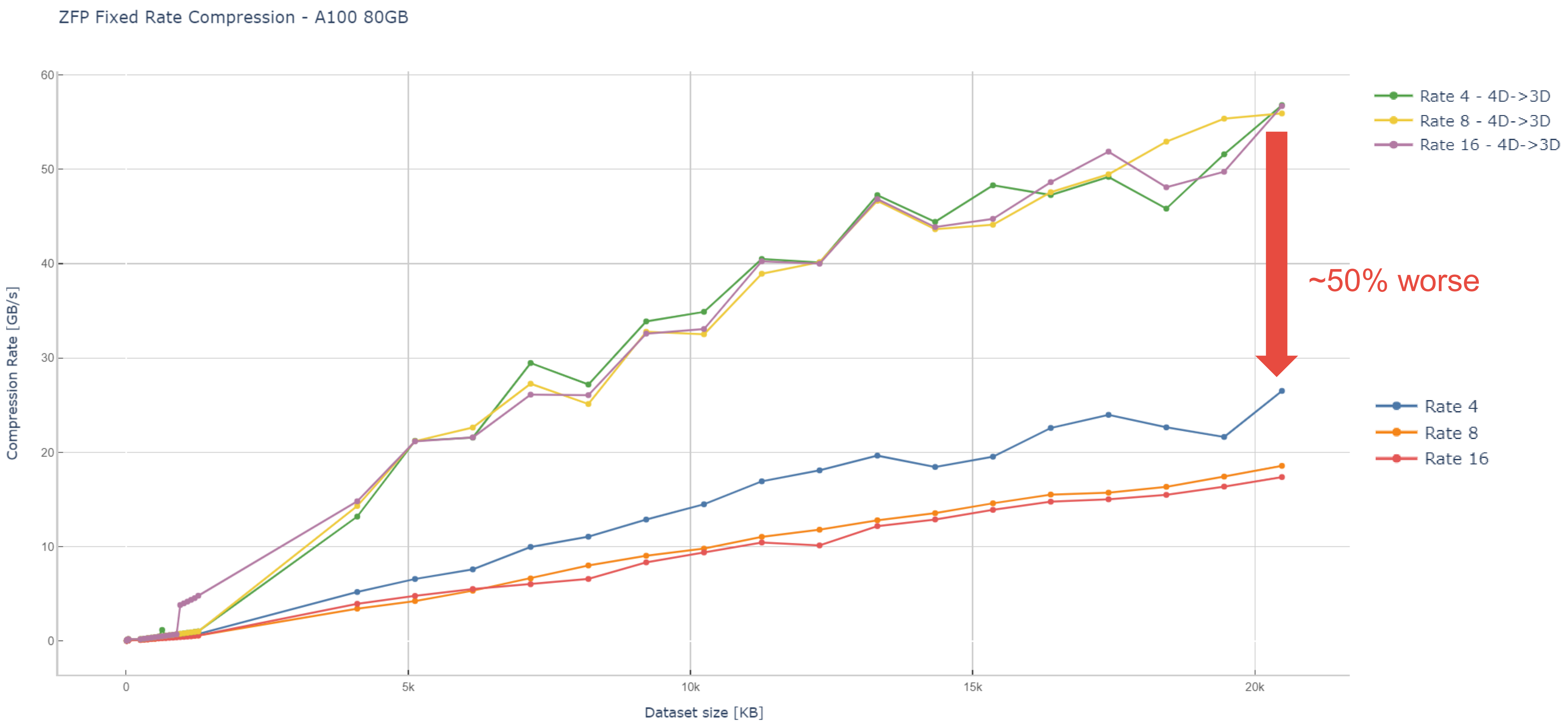
(de)compression mode		serial	OpenMP	CUDA
compression	fixed rate	x	x	x
	fixed precision	x	x	
	fixed accuracy	x	x	
decompression	fixed rate	x		x
	fixed precision	x		
	fixed accuracy	x		

GPU performance – ZFP compression

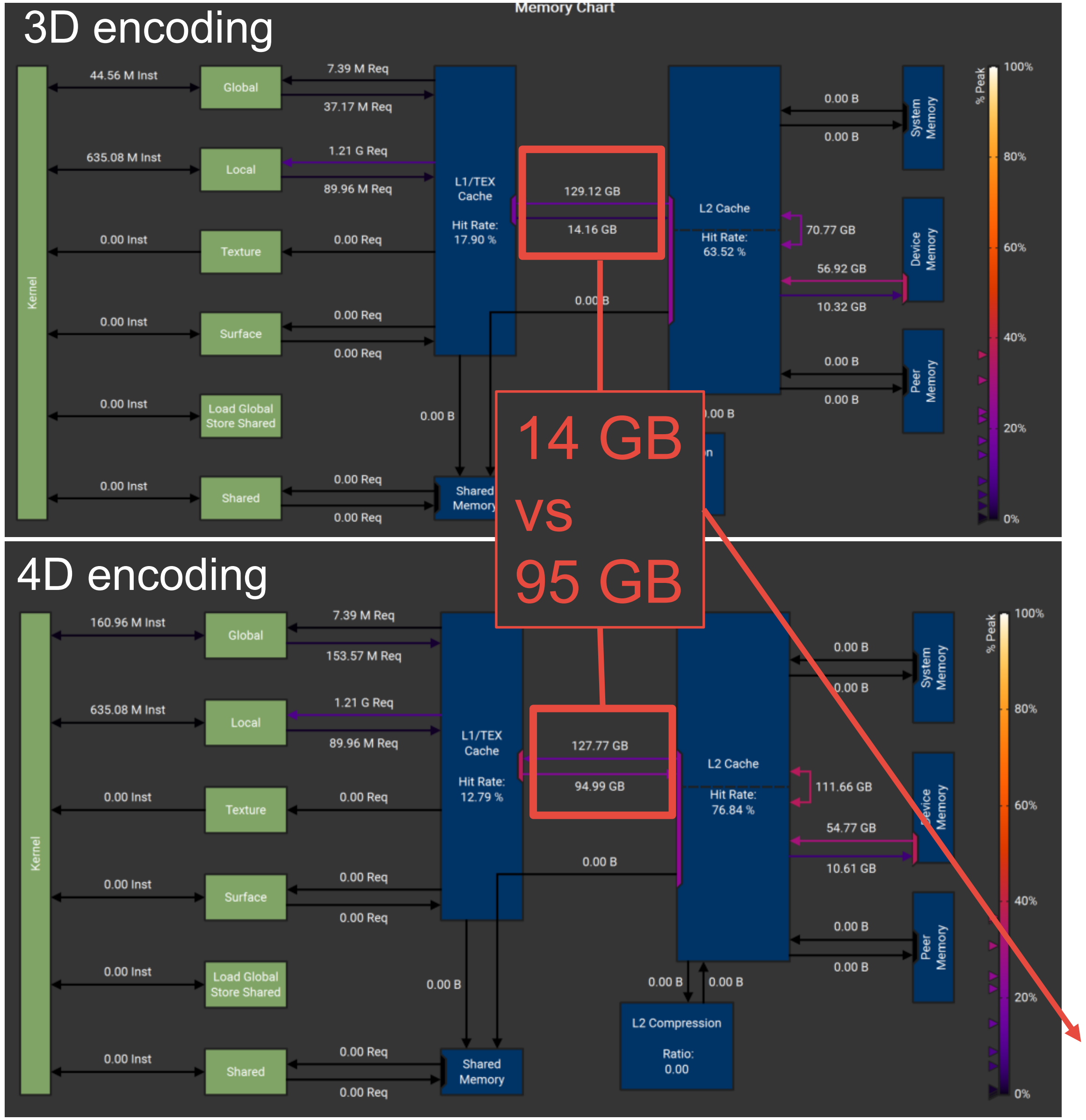
Algorithm

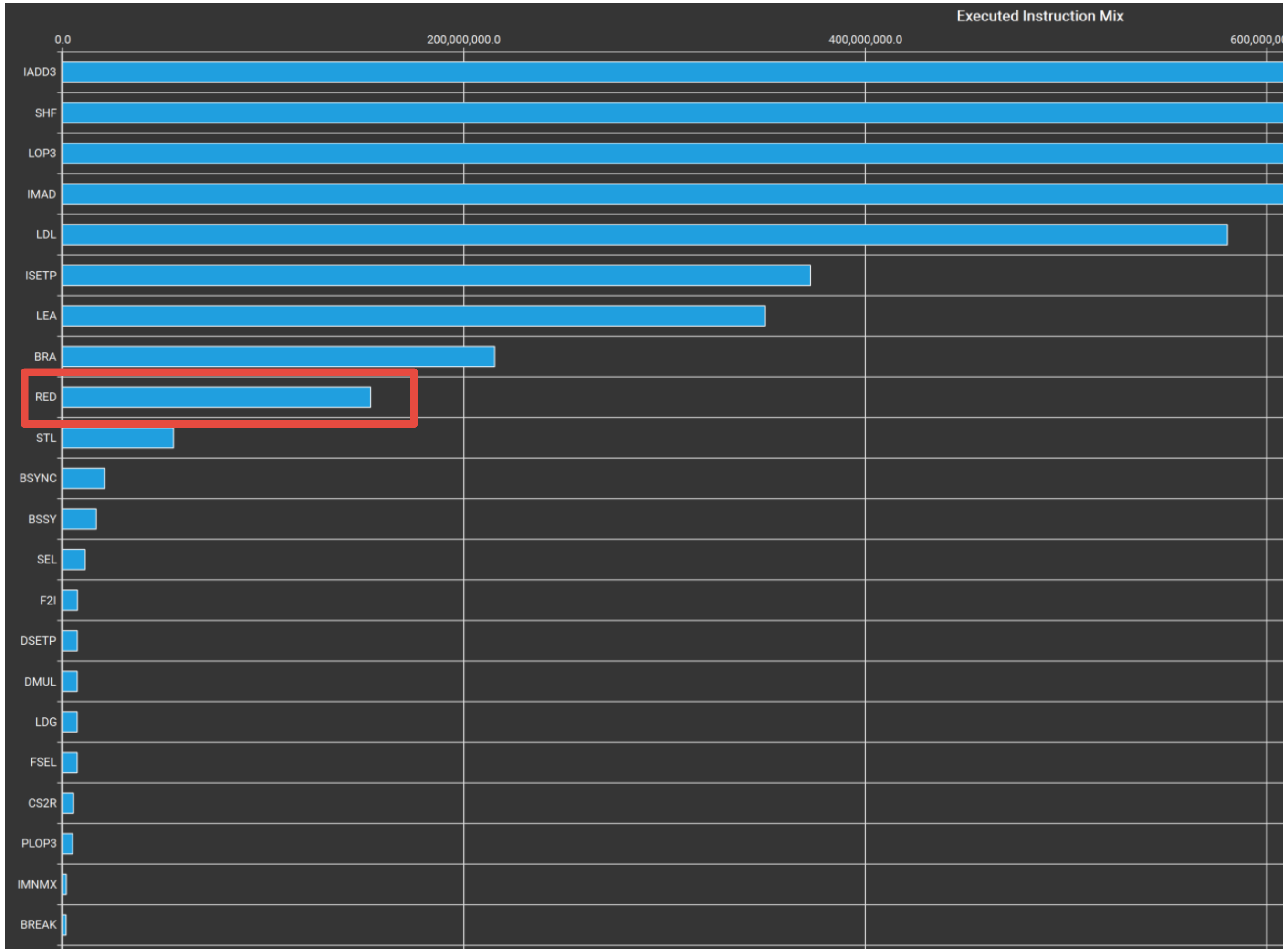


GPU performance – 4D support for CUDA



GPU performance – 4D support for CUDA





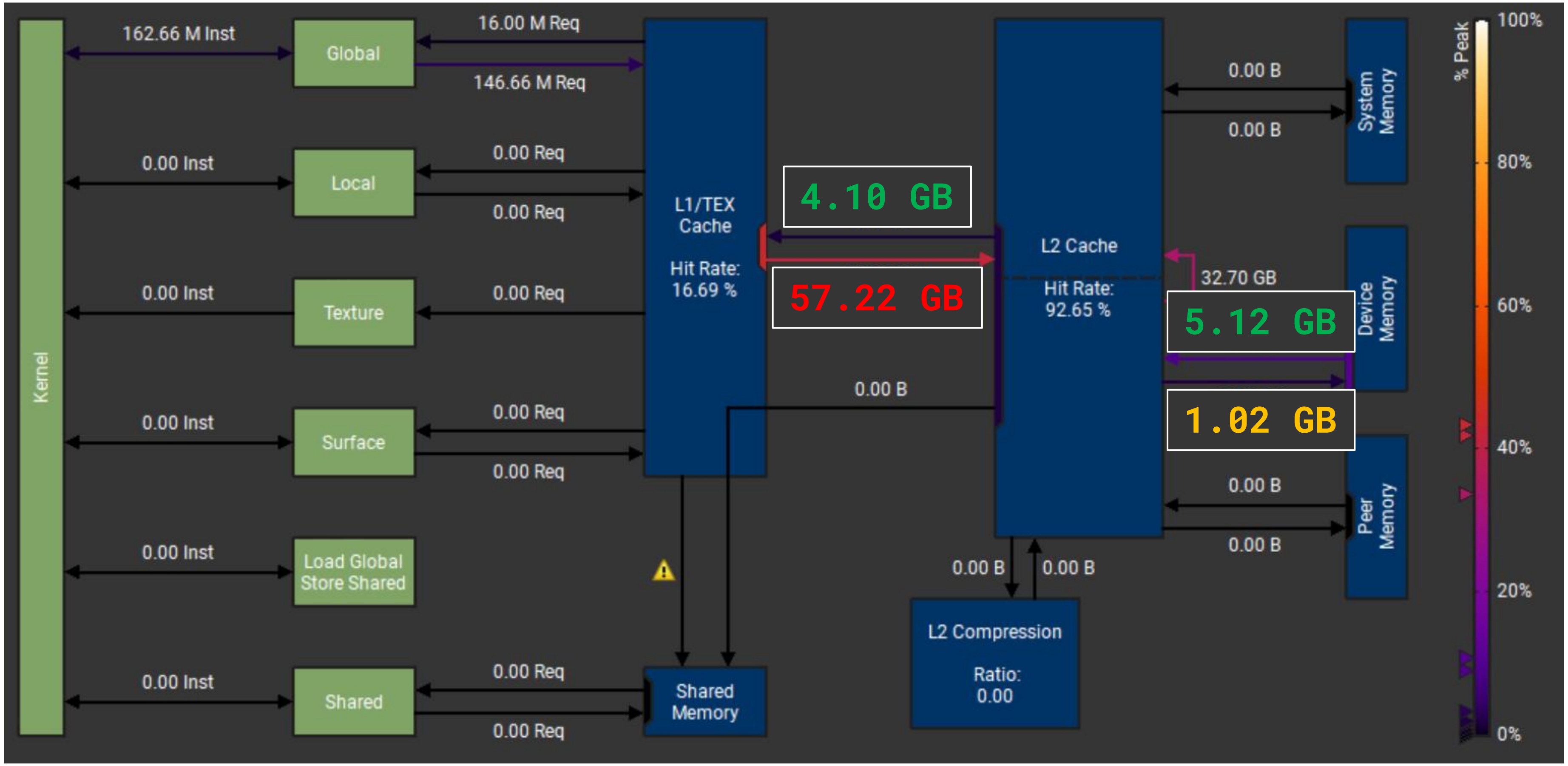
4D compression on blocks of 4^4=256 elements → 512+ registers needed (max. 255 regs per thread on a A100)

GPU performance – optimizing ZFP

1GB data set w/ fixed rate 16 → expected data traffic:

READ: 1 GB
WRITE: 256 MB

A100 (40GB)



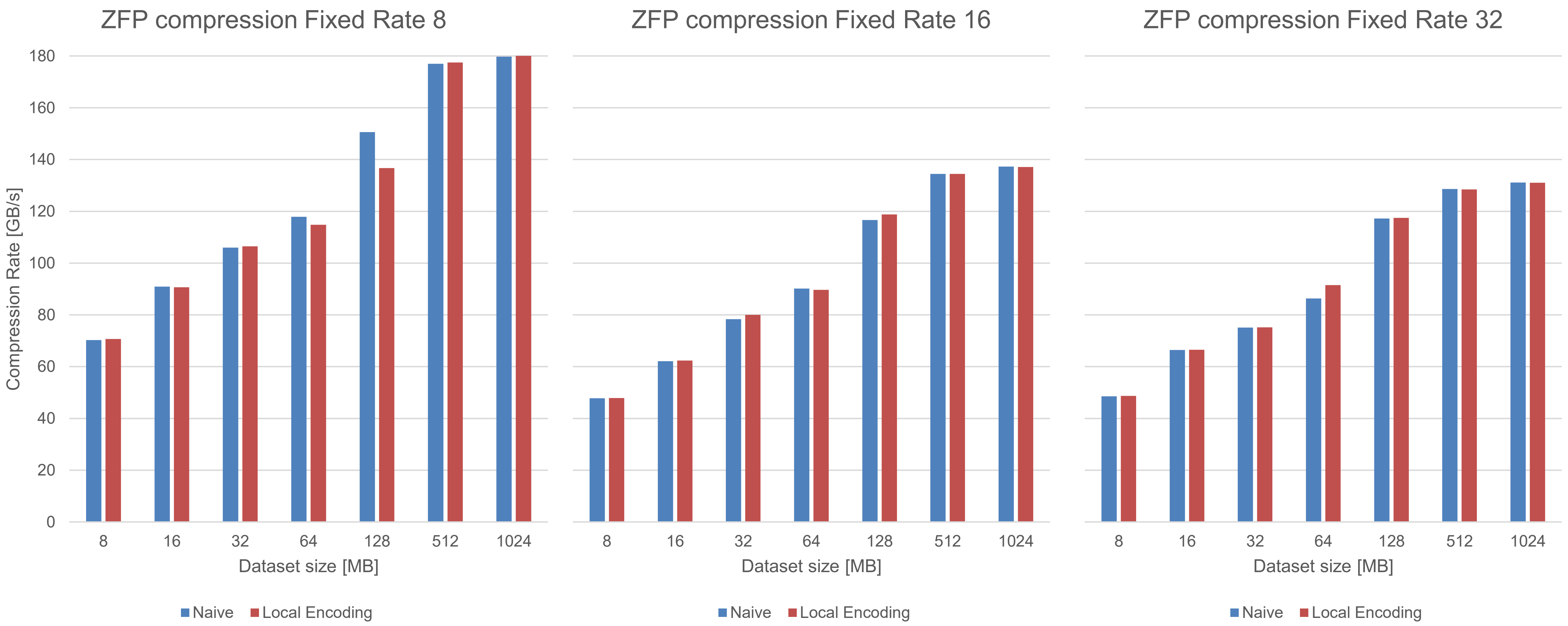
GPU performance – optimizing ZFP

Idea: use local memory instead of global memory for encoding buffer

```
template<>
void inline __device__ zfp_encode_block<long long int, 64>(
            long long int *fblock,
            const int maxbits,
            const uint block_idx,
            Word *stream)
{
  BlockWriter<64> block_writer(stream, maxbits, block_idx);
  const int intprec = get_precision<long long int>();
#ifdef LOCAL_ENCODE
  Word local_stream[64];
  for (int i=0; i<64; ++i) {
      local_stream[i] = 0;
  }
  BlockWriter<64> local_bwriter(local_stream, maxbits, 0);

  encode_block<long long int, 64>(local_bwriter, maxbits, intprec, fblock);
  for (int i=0; i<64; i++) {
      block_writer.write_bits(*local_stream+i, sizeof(Word));
  }
#else
  encode_block<long long int, 64>(block_writer, maxbits, intprec, fblock);
#endif

}
```

GPU performance – optimizing ZFP

Idea: use local memory instead of global memory for encoding buffer



GPU performance – optimizing ZFP

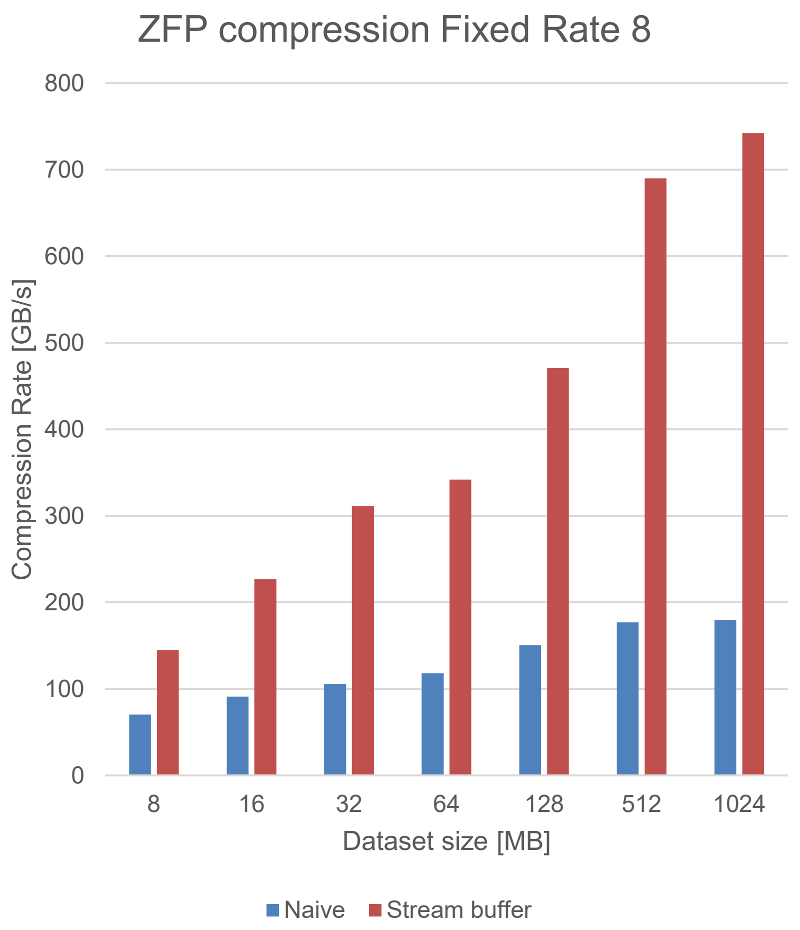
Idea: eliminate bite-wise write by using temporary buffer

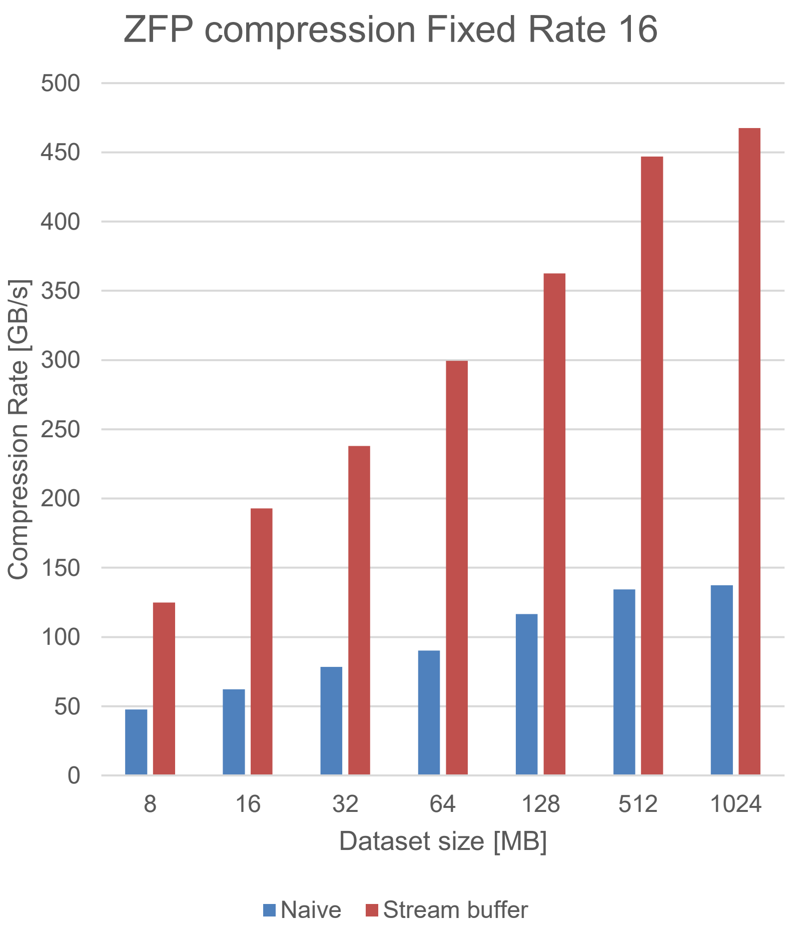
```
template<typename Int, int BlockSize>
void inline __device__ encode_block(
           BlockWriter<BlockSize> &stream,
           int maxbits,
           int maxprec,
           Int *iblock)
{
// ...
#ifndef DIFFLOOP
    for (; n < BlockSize && bits && (bits--, stream.write_bit(!!x)); x >>= 1, n++)
      for (; n < BlockSize - 1 && bits && (bits--, !stream.write_bit(x & 1u)); x >>= 1, n++)
        ;
#else
    uint64 buffer = 0;
    uint buffer_pos = 0;
    // write into buffer and send to stream as chunk
    // ...
    // ...
}
```

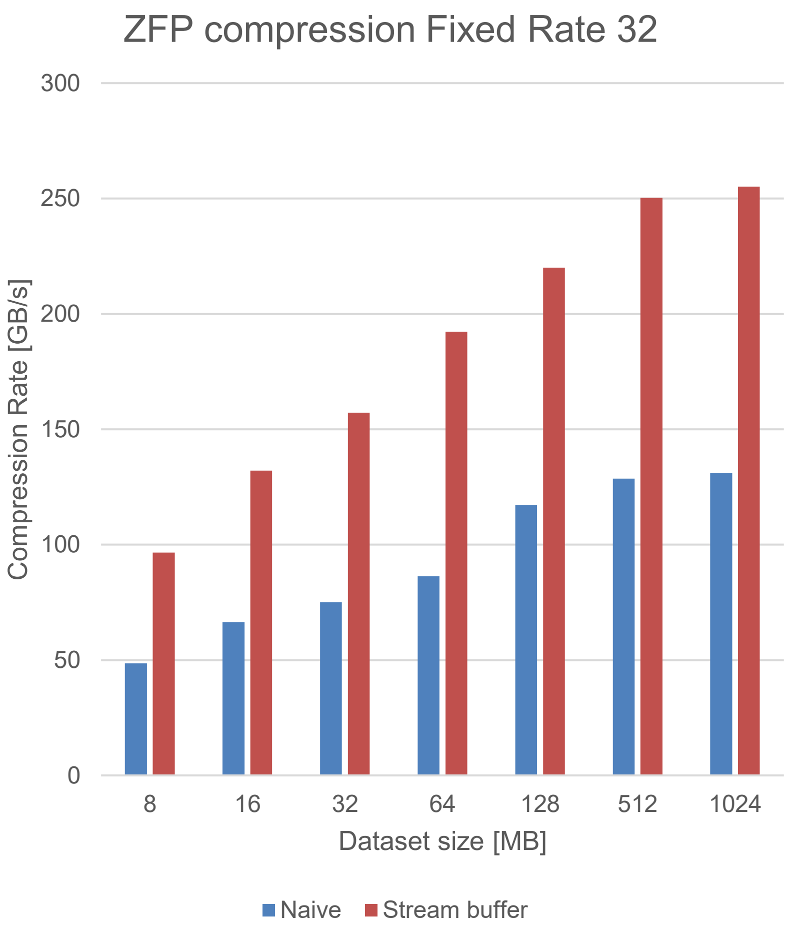
GPU performance – optimizing ZFP

Idea: eliminate bite-wise write by using temporary buffer

1.99-2.6x speedup for 8MB data sets
up to 4.1x speedup (1GB/FR8)







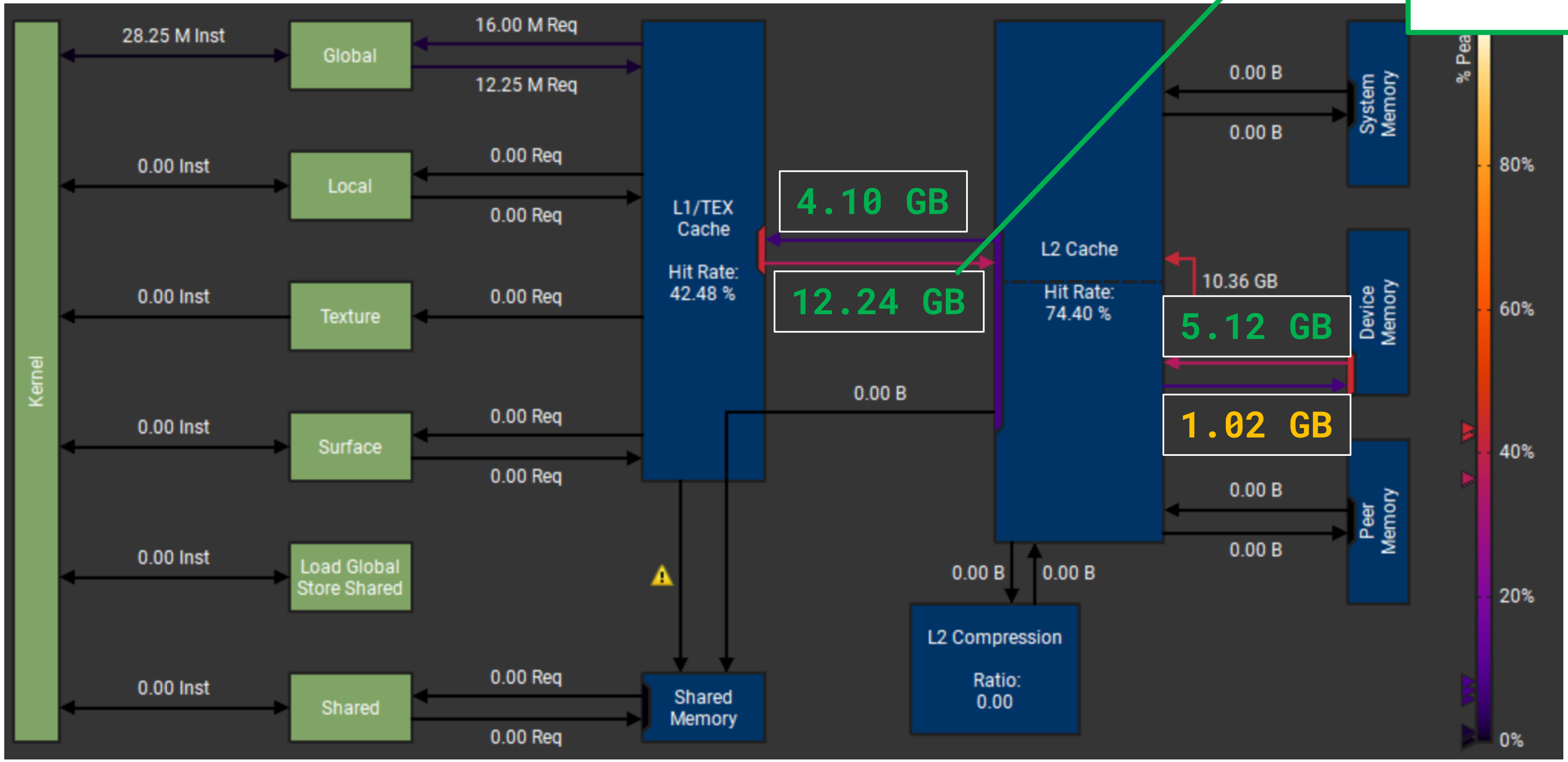
GPU performance – optimizing ZFP

1GB data set w/ fixed rate 16 → expected data traffic:

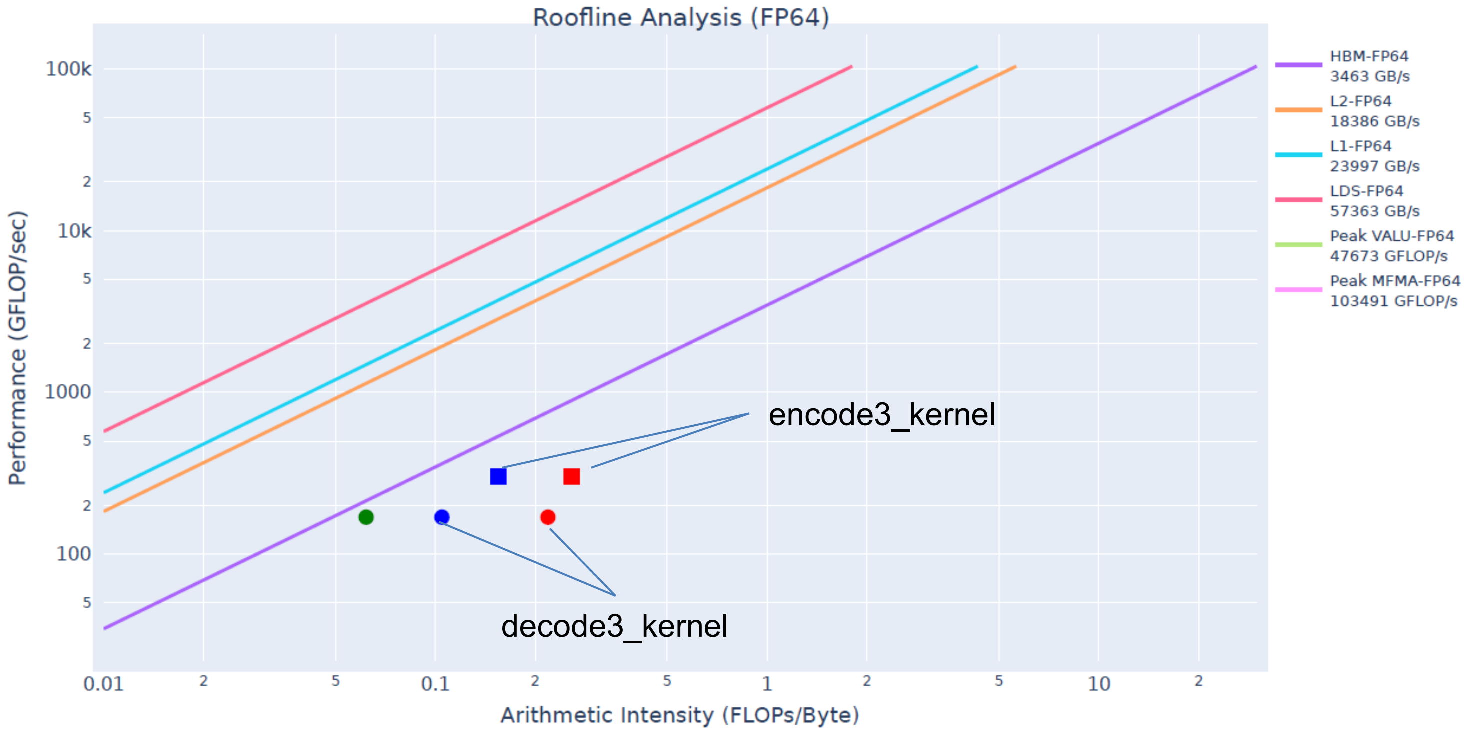
```
READ: 1 GB
WRITE: 256 MB
```

L1->L2 data volume 4.6x smaller

A100 (40GB)

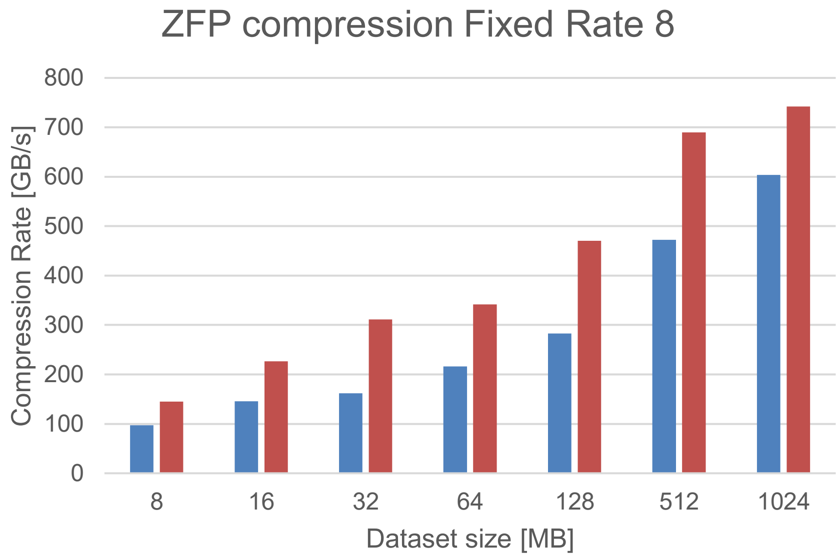


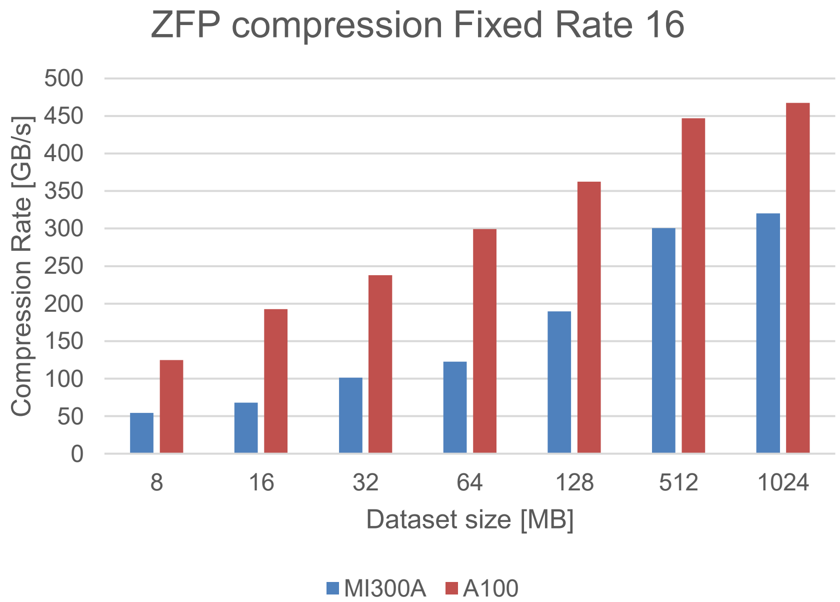
GPU performance – AMD



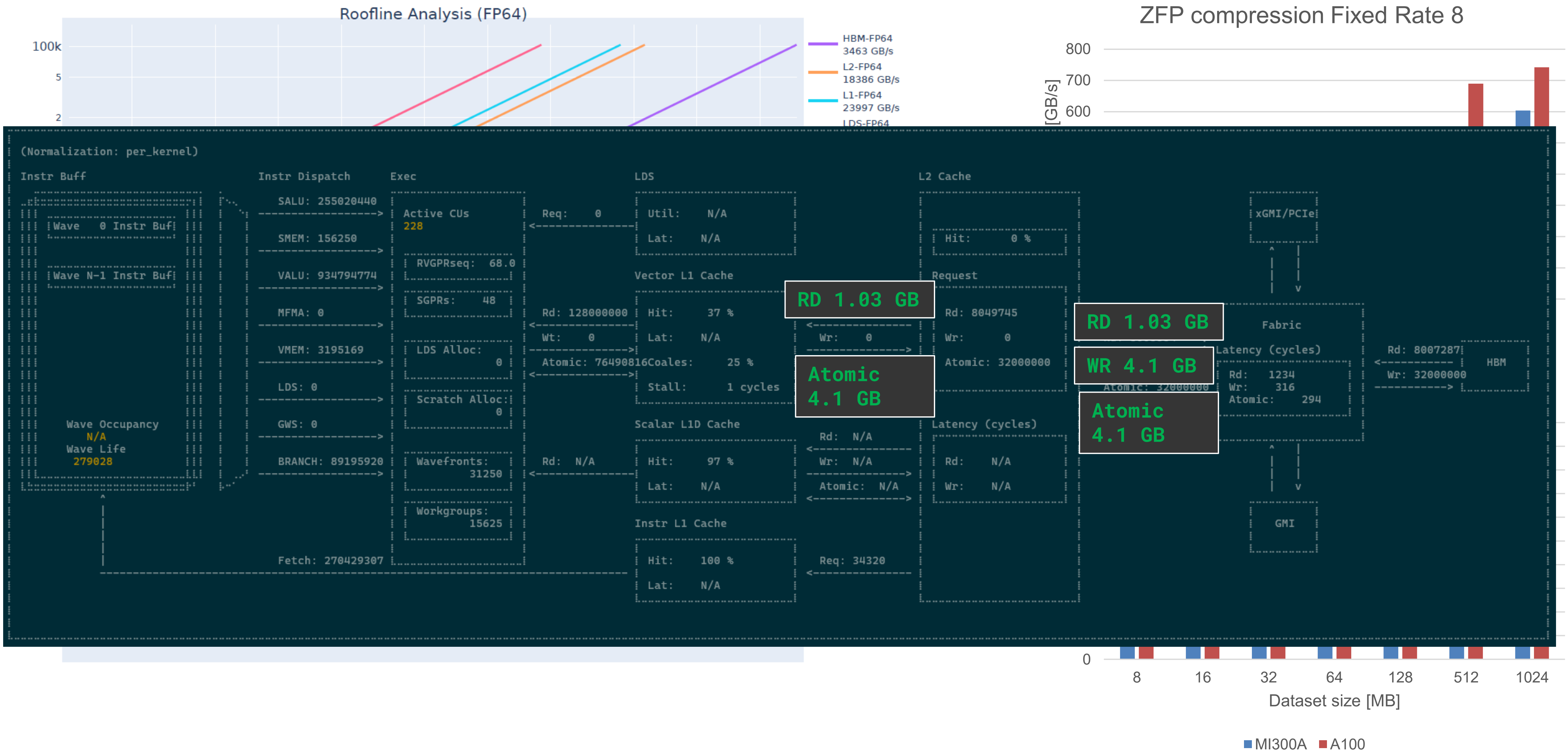
Plot Points & Values

Symbol	FLOPs/Byte	GFLOP/s	Status	Cache Level
● (blue)	0.10	169.51	Memory Bound	L1
■ (blue)	0.15	303.60	Memory Bound	L1
● (green)	0.06	169.51	Memory Bound	L2
■ (green)	0.26	303.60	Memory Bound	L2
● (red)	0.22	169.51	Memory Bound	HBM
■ (red)	0.26	303.60	Memory Bound	HBM





GPU performance – AMD



Summary

- Porting code to lower precision is difficult! (especially on CPUs)
  - Be aware of dumb compiler

- ZFP is fast, but it can be faster! (especially on NVIDIA)
  - adjust the algorithm to your hardware

- Stay up-to-date with ongoing developments
  - new compressors (cuSZp, HoSZp, ZFP branches, ...)
  - NCCL/RCCL

jan.laukemann@fau.de

Performance Modeling in DAREXA-F

Compression & Lower Precision Computing

J. Laukemann, G. Wellein

Erlangen National High Performance Computing Center (NHR@FAU)

Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg