Hardware, Software, and Application Co-Design on the Frontier and Lumi Systems

Jakub Kurzak, Gina Sitaraman, Leopold Grinberg, Asitav Mishra, Nicholas Malaya, Paul Bauman, George Markomanolis, Samuel Antao

ZIH Colloquium – TU Dresden
27/04/2023

together we advance_

Enabled by AMD Instinct™ GPUs

- OLCF Frontier reached 1.1 ExaFLOPS on the High Performance Linpack (HPL) benchmark becoming the first system ever to reach an ExaFLOP of 64-bit (Double Precision) performance on the June 2022 Top500 list.
- Most FLOPs coming from AMD Instinct™ GPU accelerators
- It is officially the world's first Exascale Supercomputer
- It is officially the most power efficient Supercomputer in the world **at scale:** More science per watt

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

TOP500 The List.

THE GREEN 500

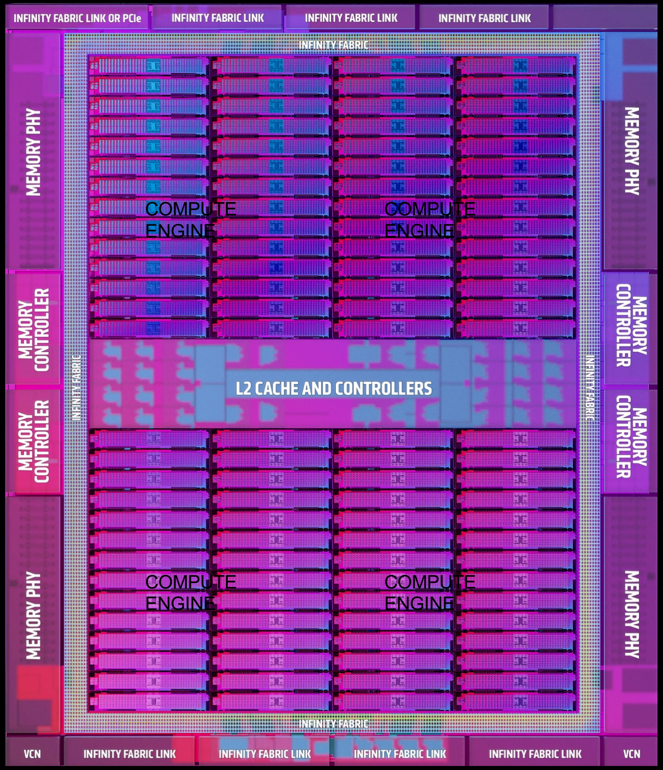
AMD together we advance_

Enabled by AMD Instinct™ GPUs

- LUMI is hosted in Finland by CSC with many other countries in the consortium alongside the EuroHPC Joint Undertaking initiative
- Most FLOPs coming from AMD Instinct™ GPU accelerators

[Public]

AMD Instinct™ GPUs



2ND GENERATION CDNA ARCHITECTURE

TAILORED-BUILT FOR HPC & AI

TSMC 6NM TECHNOLOGY

UP TO 110 CU PER GRAPHICS CORE DIE

4 MATRIX CORES PER COMPUTE UNIT

MATRIX CORES ENHANCED FOR HPC

8 INFINITY FABRIC LINKS PER DIE

SPECIAL FP32 OPS FOR DOUBLE THROUGHPUT

AMD
together we advance_

Agenda

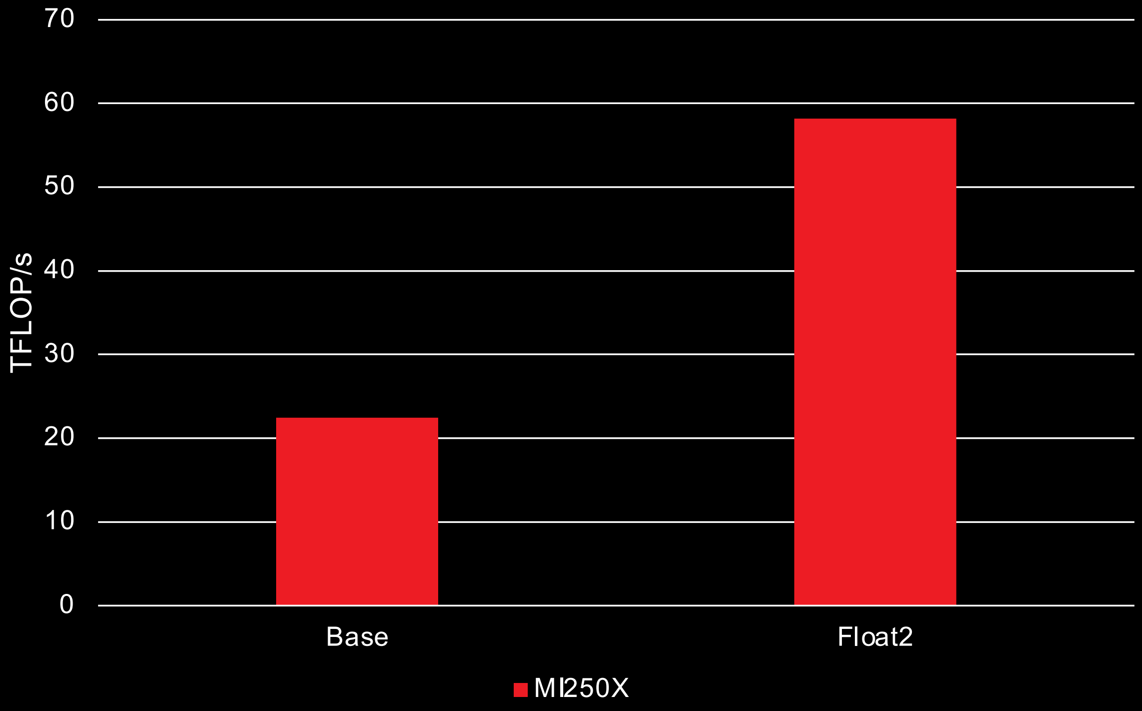
1. Cases about packed FP32 operations
2. Matrix Cores and MPI scalability
3. Optimizing CP2K for AMD GPUs
4. Tuning and profiling around SHOC
5. Evolving programming models
6. Communication tunning in ML workloads

Packed FP32

FP64 PATH USED TO EXECUTE TWO COMPONENT VECTOR INSTRUCTIONS ON FP32

DOUBLES FP32 THROUGHPUT PER CLOCK PER COMPUTE UNIT

pk_FMA, pk_ADD, pk_MUL, pk_MOV operations



https://www.amd.com/en/technologies/infinity-hub/mini-hacc

AMD together we advance_

Refactoring code to emit PACKED FP32 instructions

Original

```
float vxi = 0.0f, vyi = 0.0f, vzi = 0.0f;
  for (int j = hipThreadIdx_x; j < count1; j += hipBlockDim_x) {
    float dx = xx1[j] - xxi;
    float dy = yy1[j] - yyi;
    float dz = zz1[j] - zzi;
    float dist2 = dx*dx + dy*dy + dz*dz;
    if (dist2 < fsrrmax2) {
      float rtemp = (dist2 + rsm2)*(dist2 + rsm2)*(dist2 + rsm2);
      float f_over_r = massi*mass1[j]*(1.0f/sqrt(rtemp) - (ma0 +
dist2*(ma1 + dist2*(ma2 + dist2*(ma3 + dist2*(ma4 + dist2*ma5))))));

      vxi += fcoeff*f_over_r*dx;
      vyi += fcoeff*f_over_r*dy;
      vzi += fcoeff*f_over_r*dz;
    }
  }
```

Modified to use Packed FMA32

```
float2 vxi = 0.0f, vyi = 0.0f, vzi = 0.0f;
  for (int j = hipThreadIdx_x; j < count1; j += 2*hipBlockDim_x) {
    float2 dx = {xx1[j] - xxi, xx1[j+ hipBlockDim_x] – xxi};
    float2 dy = {yy1[j] - yyi, yy1[j+ hipBlockDim_x] – yyi};
    float2 dz = {zz1[j] - zzi, zz1[j+ hipBlockDim_x] – zzi};
    float2 dist2 = dx*dx + dy*dy + dz*dz;
    if (dist2 < fsrrmax2) {
      float2 rtemp = (dist2 + rsm2)*(dist2 + rsm2)*(dist2 + rsm2);
      float2 f_over_r = massi*mass1[j]*(1.0f/sqrt(rtemp) - (ma0 +
dist2*(ma1 + dist2*(ma2 + dist2*(ma3 + dist2*(ma4 + dist2*ma5))))));

      vxi += fcoeff*f_over_r*dx;
      vyi += fcoeff*f_over_r*dy;
      vzi += fcoeff*f_over_r*dz;
    }
  }
```

https://www.amd.com/en/technologies/infinity-hub/mini-hacc

AMD
together we advance_

Biomedical Knowledge Base Analysis

1 ExaFLOPS Biomedical Knowledge Base Analysis

- COAST / SnapShot project at ORNL
  - mining large datasets of biomedical literature (PubMed, SPOKE)
  - tens of millions of publications
  - hundreds of thousands of concepts

- discovering unknown connections, e.g.
  - new link between a symptom and a toxin
  - new candidate drug for a disease

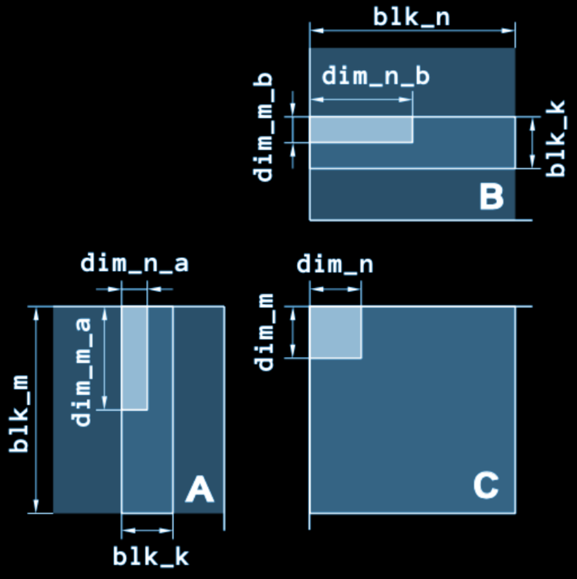
- classic graph-theoretical approach to data mining
  - solving the All-Pairs-Shortest-Path problem
  - Floyd-Warshall algorithm (classic example of dynamic programming)
  - semiring algebra (GEMM / matrix multiply-like kernels)

- SC22 Gordon Bell finalist
  - focus on COVID research
  - https://dl.acm.org/doi/abs/10.5555/3571885.3571892
  - https://www.computer.org/csdl/proceedings-article/sc/2022/544400a061/1I0bSLIULiU

1 ExaFLOPS Biomedical Knowledge Base Analysis

- key architectural extension – packed FP32 math
  - double-throughput of FP32 operations
  - without the use of Matrix Engines
- main kernel – implementing a semiring operation
  - GEMM-like kernel using addition and minimum
  - implementation in HIP using the `float2` type
  - 15.3 TF per GCD
  - 30.6 TF per MI250X OAM

```
__device__
void compute_minplus(float2 a[THR_M],
                     float2 b[THR_N],
                     float  c[THR_N][THR_M])
{
    for (int j = 0; j < THR_N; ++j)
        for (int i = 0; i < THR_M; ++i)
            c[j][i] = fminf(fminf((a[i]+b[j]).x,
                                  (a[i]+b[j]).y),
                            c[j][i]);
}
```



AMD
together we advance_

1 ExaFLOPS Biomedical Knowledge Base Analysis

Nodes	Vertices (Million)	Memory (in TB)	Time (Secs)	PFlops/s	Fractional Peak
Summit Fastest Run (in GPU)					
4096	4.43	209	1280	135.9	70%
Frontier Fastest Run					
9200	7.06	399	702	1004	75%
Frontier Largest Run					
9025	18.6	2,768	13800	945	73%

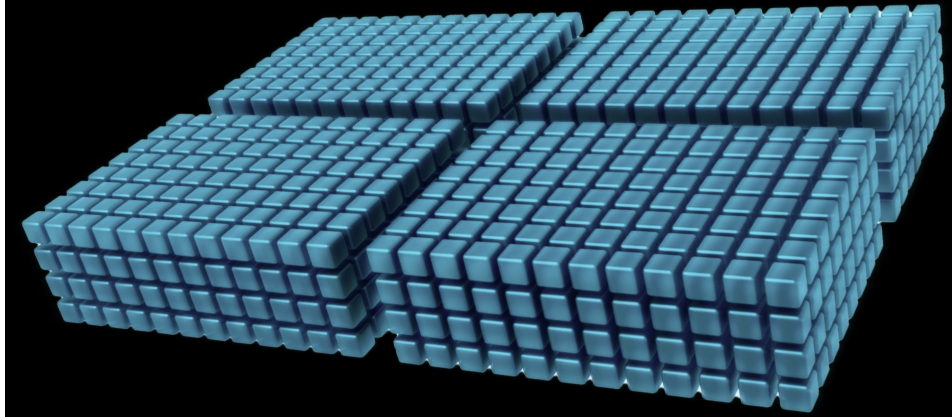
- 1.004 EF using 9,200 nodes of Frontier
  - 36,800 MI250X GPUs
  - 73,600 GCDs
- compared to 136 PF on Summit
  - 7x faster

Matrix Cores

2nd GENERATION MATRIX CORES

OPTIMIZED COMPUTE UNITS FOR MATRIX OPERATIONS

DOUBLE PRECISON (FP64) MATRIX CORE THROUGHPUT REPRESENTATION



MI100 MATRIX CORES OPS/CLOCK/COMPUTE UNIT	MI250X MATRIX CORES OPS/CLOCK/COMPUTE UNIT
No FP64 Matrix Core	256 FP64
256 FP32	256 FP32
1024 FP16	1024 FP16
512 BF16	1024 BF16
512 INT8	1024 INT8

https://developer.amd.com/wp-content/resources/CDNA2_Shader_ISA_18November2021.pdf

[Public]

2nd GENERATION MATRIX CORES

OPTIMIZED COMPUTE UNITS FOR MATRIX OPERATIONS

```
#define M 16
#define N 16
#define K 4

using float4 = __attribute__( (__vector_size__(K * sizeof(float)) )) float;

__global__ void sgemm_16x16x4(const float *A, const float *B, float *D)
{
 float4 dmn = {0};

 int mk = threadIdx.y + K * threadIdx.x;
 int kn = threadIdx.x + N * threadIdx.y;

 float amk = A[mk];
 float bkn = B[kn];
 dmn = __builtin_amdgcn_mfma_f32_16x16x4f32(amk, bkn, dmn, 0, 0, 0);

 for (int i = 0; i < 4; ++i) {
 const int idx = threadIdx.x + i * N + threadIdx.y * 4 * N;
 D[idx] = dmn[i];
 }
}
```

- Current support for using MFMA instructions:
  - AMD libraries: rocBLAS
  - AMD rocWMMA library
  - LLVM™ builtin compiler intrinsic functions
  - Inline assembly

https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-matrix-cores-readme/

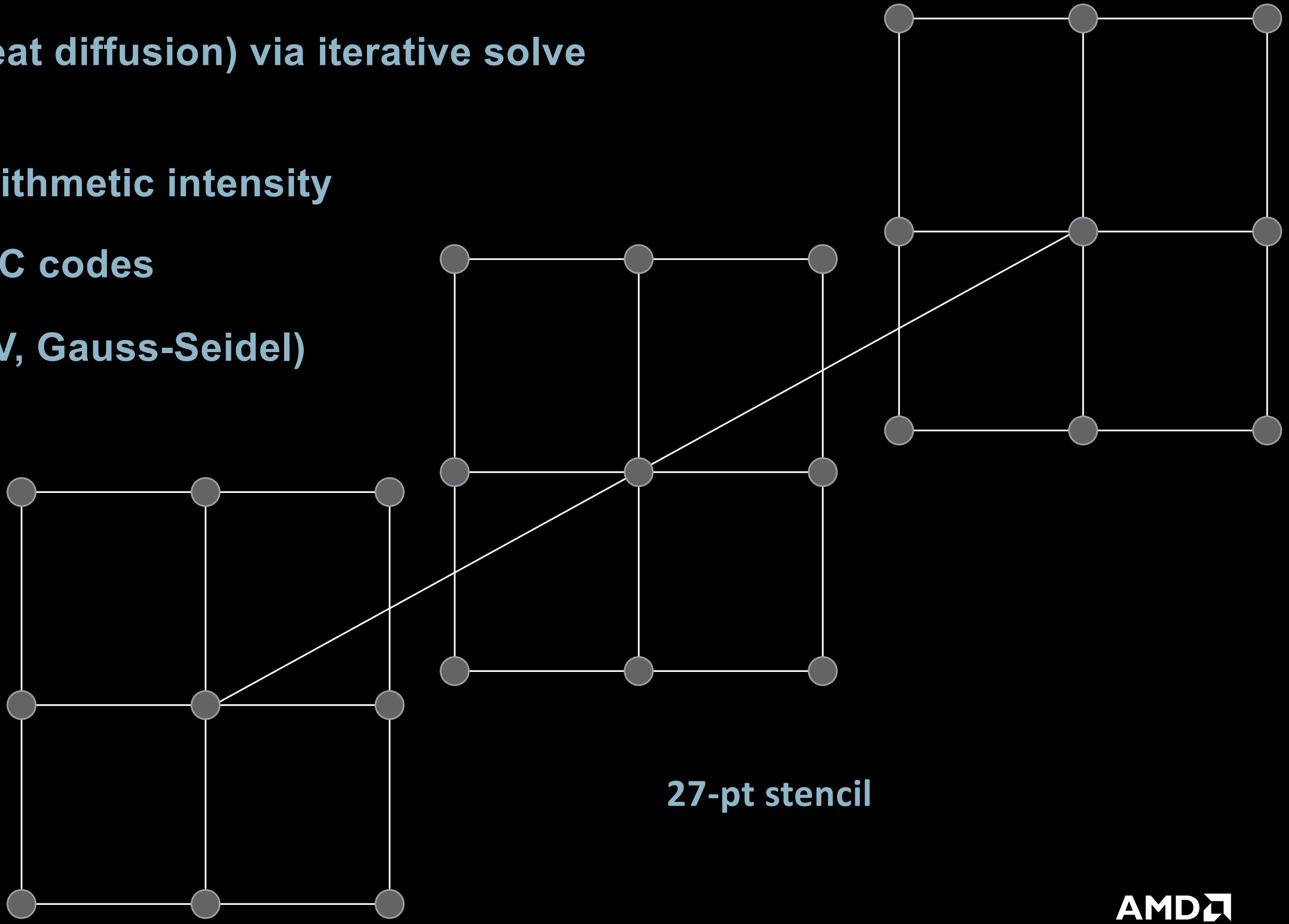
https://github.com/ROCmSoftwarePlatform/rocWMMA

AMD together we advance_

[Public]

THE HPCG BENCHMARK

- High Performance Conjugate Gradient (HPCG)
  - Solves the 3d Poisson equation (heat diffusion) via iterative solve (conjugate gradient)
  - Sparse linear solver (SpMV), low arithmetic intensity
  - Broadly representative of many HPC codes
    - Memory bandwidth scaling (SpMV, Gauss-Seidel)
    - MPI collectives (all-reduce)
    - MPI sendrecvs (halo exchange)
  - Weak scaling benchmark


27-pt stencil

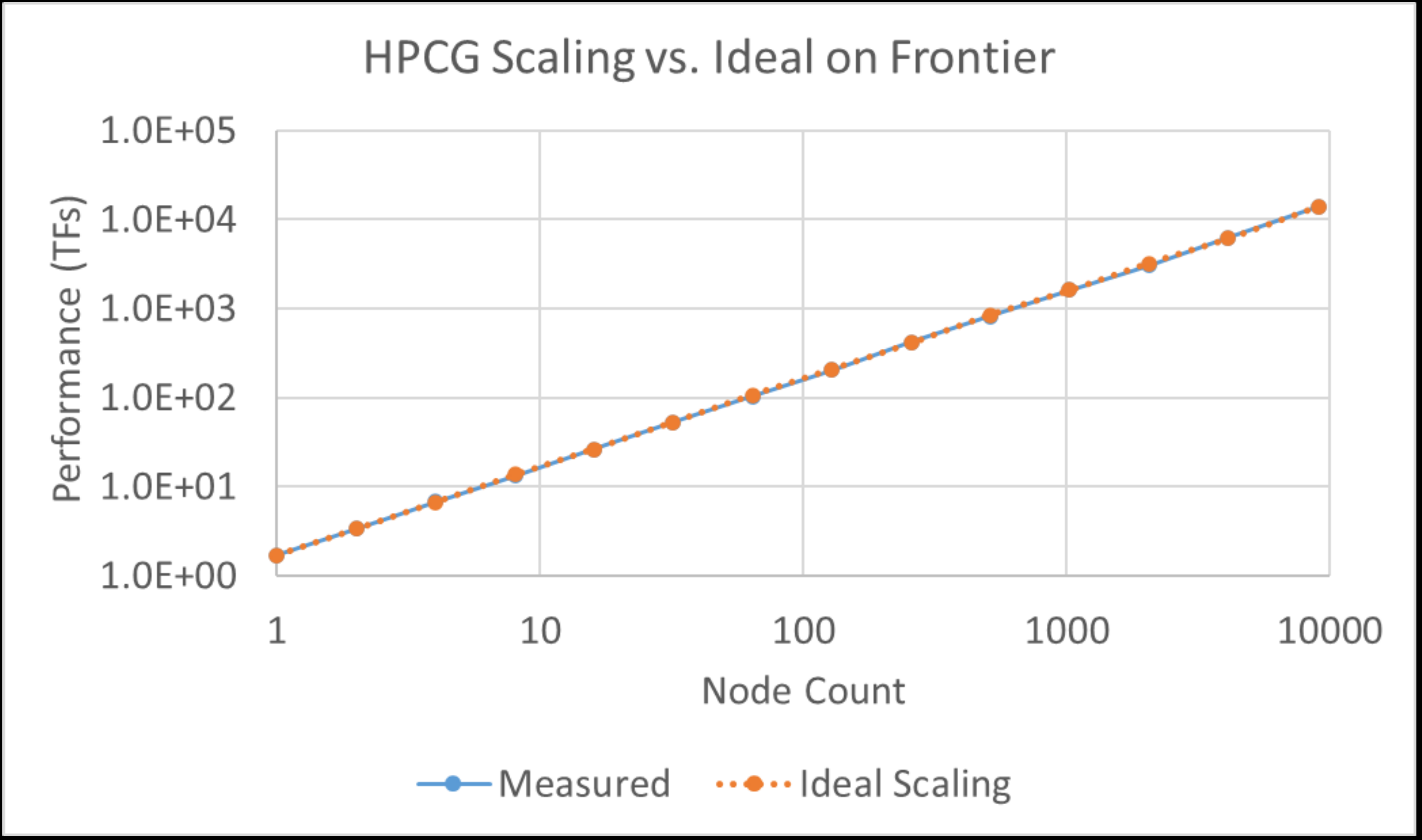
AMD
together we advance_

[Public]

HPCG SCALING ON FRONTIER

Data Gathered By: Paul Bauman, AMD

- HPCG
  - Weak scaling benchmark
  - Nearly ideal scalability
  - From 1 -> 9000+ nodes
  - Implemented via MPI
- HPE Cray MPI
  - Slingshot interconnect
  - GPU-aware MPI
- Why does it scale well?
  - Low-latency
  - Low tail-latency



AMD
together we advance_

AMD

Optimizing CP2K for AMD GPUs

CP2K Introduction

- CP2K Project: https://www.cp2k.org
- Quantum Chemistry and Solid-State Physics package
  - Performs simulations of solid state, liquid, molecular and biological systems
- Written in Fortran 2008, a little C++
- 16 dependencies and counting (several with GPU backends)
- Parallelized with multi-threading (OpenMP®), MPI, and HIP/CUDA

Key Contributions

- Contributions from CP2K developer community:
  - Several HIP backends: GRID, PW, DBCSR, COSMA
- Setting CPU thread affinity and GPU affinity based on node topology
- Using GPU aware MPI or RCCL for collectives in COSMA
  - Fast interconnects between GPU devices lower communication latency
  - With NICs directly connected to the GPUs, inter-node communication latency is lower
- Using GPU aware MPI in DBCSR
  - Keeping data in GPU memory and moving some computations from CPU to GPU reduced contention on CPU resources and sped up other asynchronous CPU activity
- OpenMP® parallelization of CPU intensive regions
  - Better utilization of CPU cores

Lessons Learned

- Use NPS4 configuration as opposed to NPS1 and set CPU thread affinity and GPU affinity
  - Maximized use of system resources such as memory controllers, CPU cores, GPUs, and NICs
  - Improved CPU memory bandwidth utilization
  - Threads of a process are mapped to the same NUMA domain, thereby sharing L3 cache
  - Processes' threads should be mapped to CPU cores closest to the device assigned to that process in order to improve H2D and D2H data transfers rates
- Use GPU aware MPI where applicable
  - NICs attached directly to the GCDs lower inter-node communication latency
  - Fast Infinity Fabric™ links between GCDs on MI250X GPUs lower communication latency between GPU devices
- Use multiple processes sharing the same device (GCD)
  - Improved CPU resource utilization
- Use __launch_bounds__ to reduce register spills
- Application Specific Tuning
  - Select a square grid of processes, for example, running 16 ranks vs 8 ranks on 8 MI250X GCDs
    - Lower communication overhead and improved CPU memory bandwidth utilization in COSMA
  - Using larger tile sizes in COSMA resulted in fewer calls to rocBLAS and higher computational load in each call
- Avoid use of complicated schemes involving numerous streams and event-based synchronization across CPU threads -keep code simple

Optimizing SHOC for AMD GPU

SHOC – What is it?

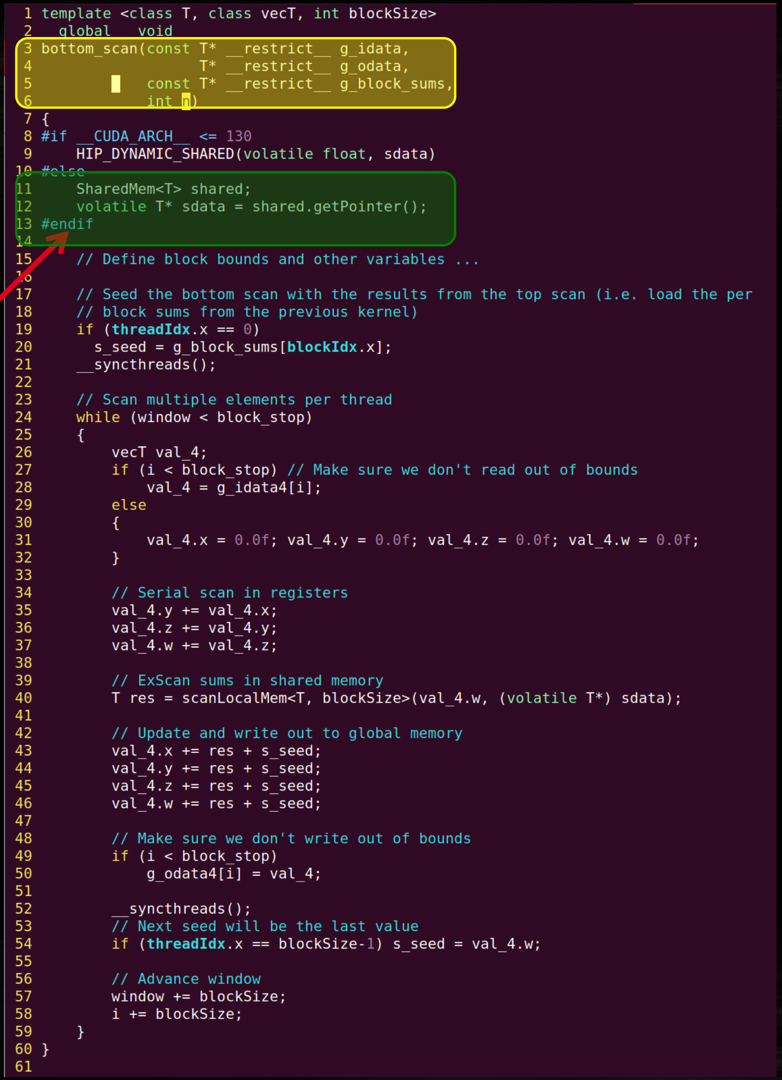
- **S**calable **H**eter**O**geneous **C**omputing **B**enchmarks
- Collection of HPC specific benchmarks to test a system for:
  - performance (stress tests)
  - stability (correctness)
- Levels of Parallelization:
  - Serial (per device)
  - Embarrassingly Parallel (multi-node/devices but no communication)
  - Truly Parallel (multi-node/devices with communication)
- Accelerator Programming Models:
  - CUDA
  - OpenCL™
- Multi-node, multi-devices, MPI
- Classes of Benchmarks
  - Level0 (baseline synthetic tests): BusSpeed, DeviceMemory, MaxFlops
  - Level1 (fundamental algorithms): bfs, fft, gemm, md, spmv, and more
  - Level2 (proxy applications): s3d, qtclustering

Optimization Approach

- Optimization Process
  1. Roofline Analysis
     - Omniperf to place current performance of your kernels on the roofline and if you've reached peak expected numbers for your hardware
  2. Profile hotspots
     - First find where runtime is being spent to focus optimization efforts (largest ROI)
     - Two options:
       - rocprof with tracing flags turned on
       - Omnitrace
  3. Profile HW counters
     - Find what pieces of the hardware are stressed the most. What piece of hardware is limiting performance?
     - Two options:
       - rocprof
       - Omniperf
  4. Adjust algorithm or make changes
  5. Test performance of changes.
  6. Repeat

SHOC – Scan

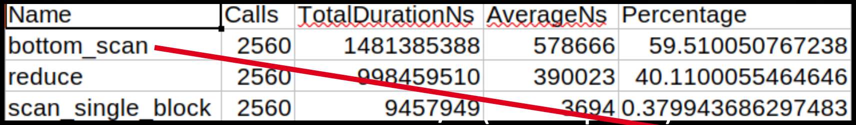
- Scan performs the 'parallel prefix sum'
  - Serial O(n) => Parallel O(n/p + log(p)) with p=procs, n=problem size
- Memory bound kernels
  - Reduction kernel
    - Blocks read coalesced array, but in non-contiguous pattern (strided by grid size)
    - Shared memory for intrablock summation
  - Bottom Scan kernel
    - Shared memory for intrablock summation



```
template <class T, class vecT, int blockSize>
__global__ void
bottom_scan(const T* __restrict__ g_idata,
                  T* __restrict__ g_odata,
            const T* __restrict__ g_block_sums,
            int n)
{
#if __CUDA_ARCH__ <= 130
    HIP_DYNAMIC_SHARED(volatile float, sdata)
#else
    SharedMem<T> shared;
    volatile T* sdata = shared.getPointer();
#endif

    // Define block bounds and other variables ...

    // Seed the bottom scan with the results from the top scan (i.e. load the per
    // block sums from the previous kernel)
    if (threadIdx.x == 0)
      s_seed = g_block_sums[blockIdx.x];
    __syncthreads();

    // Scan multiple elements per thread
    while (window < block_stop)
    {
        vecT val_4;
        if (i < block_stop) // Make sure we don't read out of bounds
            val_4 = g_idata4[i];
        else
        {
            val_4.x = 0.0f; val_4.y = 0.0f; val_4.z = 0.0f; val_4.w = 0.0f;
        }

        // Serial scan in registers
        val_4.y += val_4.x;
        val_4.z += val_4.y;
        val_4.w += val_4.z;

        // ExScan sums in shared memory
        T res = scanLocalMem<T, blockSize>(val_4.w, (volatile T*) sdata);

        // Update and write out to global memory
        val_4.x += res + s_seed;
        val_4.y += res + s_seed;
        val_4.z += res + s_seed;
        val_4.w += res + s_seed;

        // Make sure we don't write out of bounds
        if (i < block_stop)
            g_odata4[i] = val_4;

        __syncthreads();
        // Next seed will be the last value
        if (threadIdx.x == blockSize-1) s_seed = val_4.w;

        // Advance window
        window += blockSize;
        i += blockSize;
    }
}
```

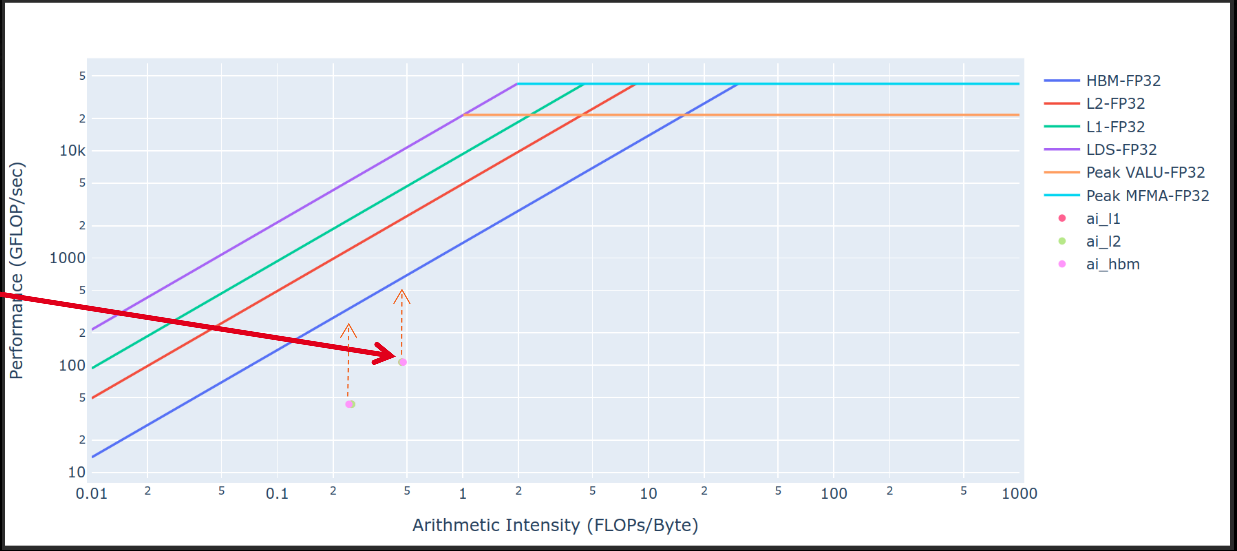
SHOC – Scan Performance Analysis

- Kernels timestamps using rocprof stats



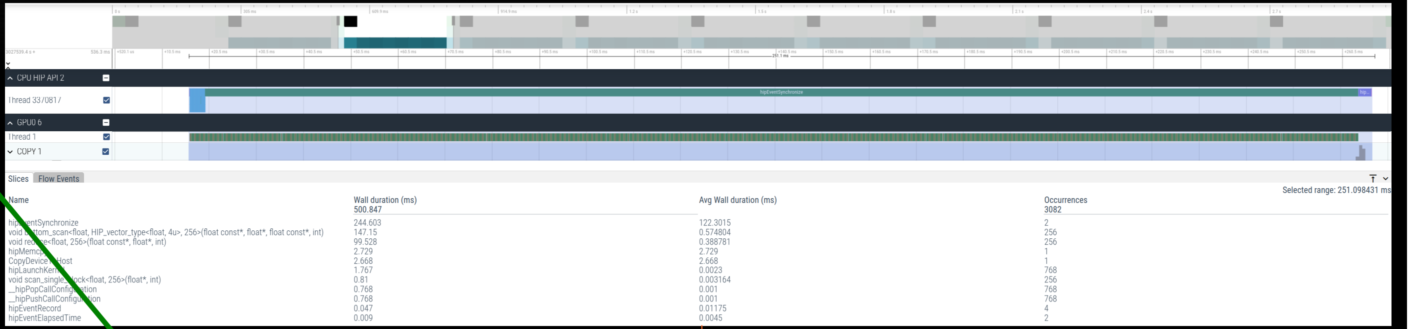
Name	Calls	TotalDurationNs	AverageNs	Percentage
bottom_scan	2560	1481385388	578666	59.510050767238
reduce	2560	998459510	390023	40.1100055464646
scan_single_block	2560	9457949	3694	0.379943686297483

- Memory bound region
  - Largely latency bound
- HBM: 69.3 GB/s



SHOC – Scan timelines

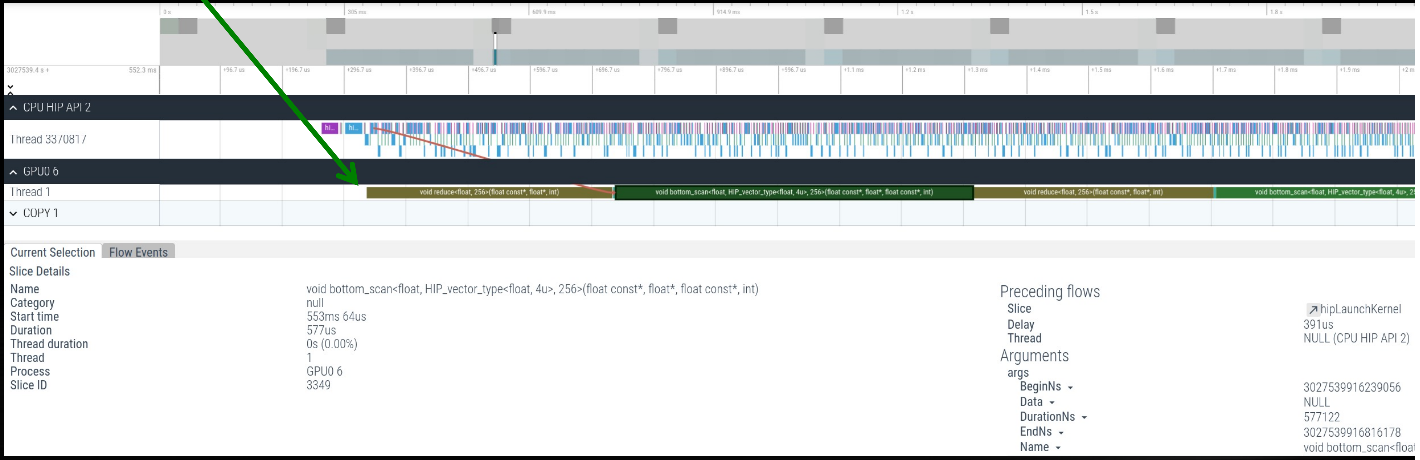
- Kernels time trace using rocprof --hip-trace or Omnitrace
- GPU kernels busy (no gaps)
- Delays in hipLaunchKernels



Selected range: 251.098431 ms

Name	Wall duration (ms) 500.847	Avg Wall duration (ms)	Occurrences 3082
hipEventSynchronize	244.603	122.3015	2
void bottom_scan<float, HIP_vector_type<float, 4u>, 256>(float const*, float*, float const*, int)	147.15	0.574804	256
void reduce<float, 256>(float const*, float*, int)	99.528	0.388781	256
hipMemcpy	2.729	2.729	1
CopyDeviceToHost	2.668	2.668	1
hipLaunchKernel	1.767	0.0023	768
void scan_single_block<float, 256>(float*, int)	0.81	0.003164	256
__hipPopCallConfiguration	0.768	0.001	768
__hipPushCallConfiguration	0.768	0.001	768
hipEventRecord	0.047	0.01175	4
hipEventElapsedTime	0.009	0.0045	2

Zoomed



CPU HIP API 2
Thread 33708 1 7
GPU0 6
Thread 1
void reduce<float, 256>(float const*, float*, int)
void bottom_scan<float, HIP_vector_type<float, 4u>, 256>(float const*, float*, float const*, int)
void reduce<float, 256>(float const*, float*, int)
COPY 1
Current Selection
Flow Events

Slice Details

Name	void bottom_scan<float, HIP_vector_type<float, 4u>, 256>(float const*, float*, float const*, int)
Category	null
Start time	553ms 64us
Duration	577us
Thread duration	0s (0.00%)
Thread	1
Process	GPU0 6
Slice ID	3349

Preceding flows

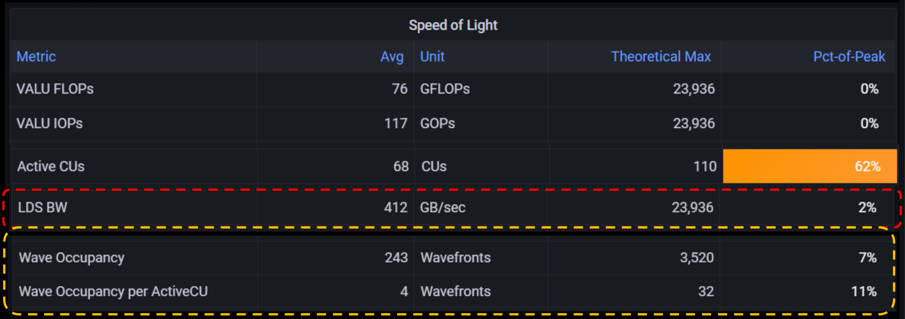
Slice	↗hipLaunchKernel
Delay	391us
Thread	NULL (CPU HIP API 2)

Arguments

args	
BeginNs	3027539916239056
Data	NULL
DurationNs	577122
EndNs	3027539916816178
Name	void bottom_scan<floa

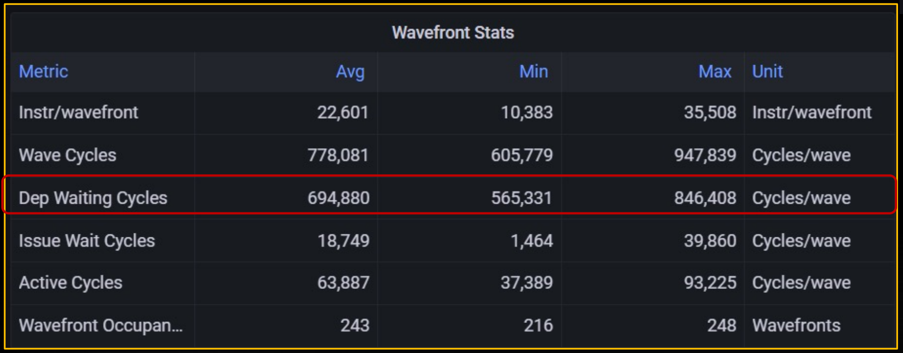
SHOC – Scan Analysis Continued…

- More performance data from Omniperf



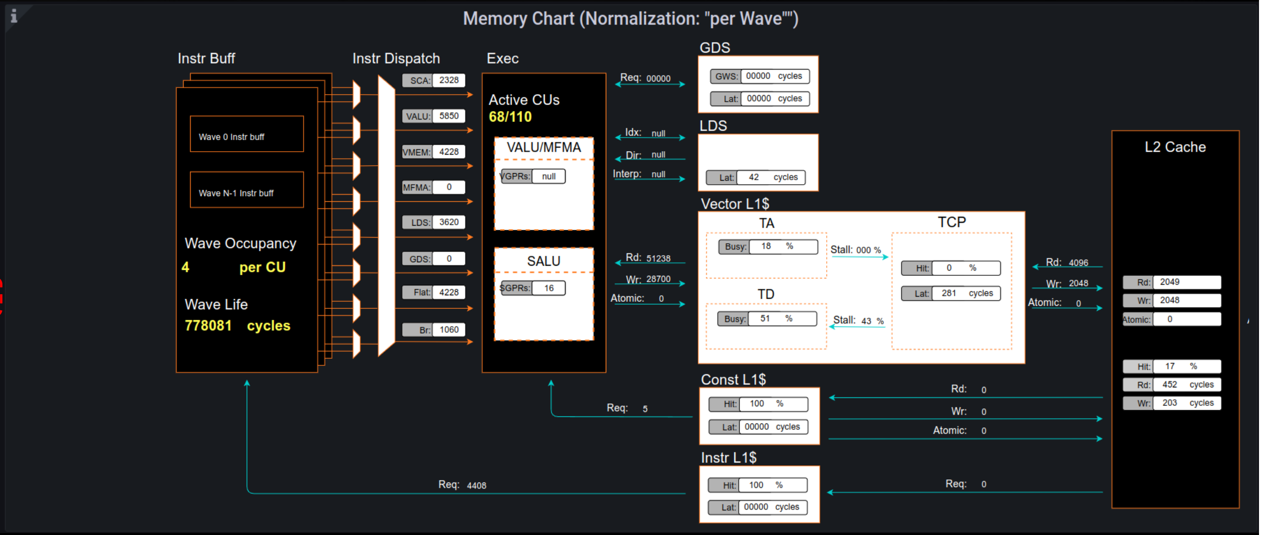
Speed of Light

Metric	Avg	Unit	Theoretical Max	Pct-of-Peak
VALU FLOPs	76	GFLOPs	23,936	0%
VALU IOPs	117	GOPs	23,936	0%
Active CUs	68	CUs	110	62%
LDS BW	412	GB/sec	23,936	2%
Wave Occupancy	243	Wavefronts	3,520	7%
Wave Occupancy per ActiveCU	4	Wavefronts	32	11%



Wavefront Stats

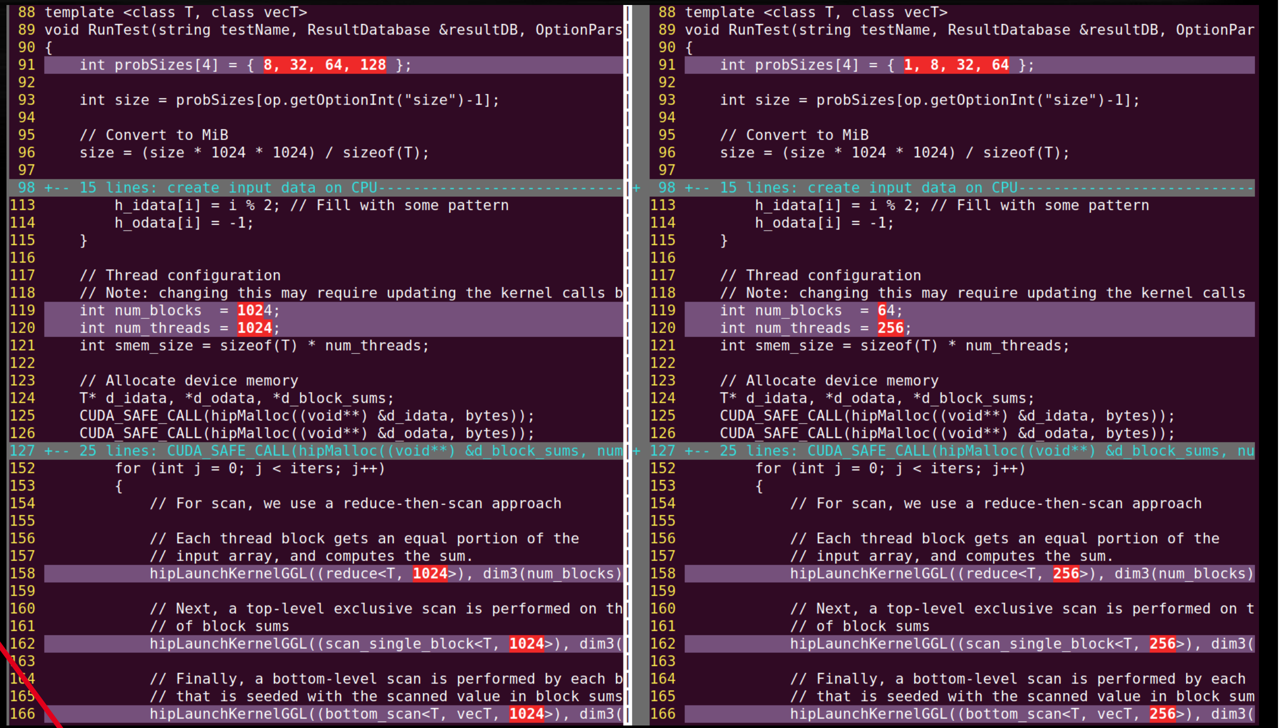
Metric	Avg	Min	Max	Unit
Instr/wavefront	22,601	10,383	35,508	Instr/wavefront
Wave Cycles	778,081	605,779	947,839	Cycles/wave
Dep Waiting Cycles	694,880	565,331	846,408	Cycles/wave
Issue Wait Cycles	18,749	1,464	39,860	Cycles/wave
Active Cycles	63,887	37,389	93,225	Cycles/wave
Wavefront Occupan...	243	216	248	Wavefronts



[Public]

SHOC – Scan Optimization/Tuning

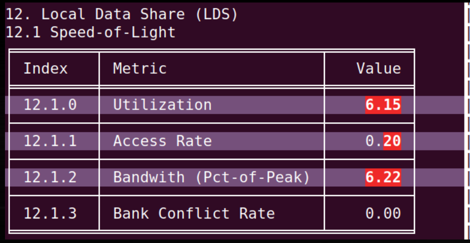
- Increase global problem size
- LDS utilization improvements
  - Increase workgroup size (blocksize)
  - Launch bounds
- Total number of blocks



```
template <class T, class vecT>
void RunTest(string testName, ResultDatabase &resultDB, OptionPars
{
    int probSizes[4] = { 8, 32, 64, 128 };

    int size = probSizes[op.getOptionInt("size")-1];

    // Convert to MiB
    size = (size * 1024 * 1024) / sizeof(T);

+-- 15 lines: create input data on CPU-----------------------------
        h_idata[i] = i % 2; // Fill with some pattern
        h_odata[i] = -1;
    }

    // Thread configuration
    // Note: changing this may require updating the kernel calls b
    int num_blocks  = 1024;
    int num_threads = 1024;
    int smem_size = sizeof(T) * num_threads;

    // Allocate device memory
    T* d_idata, *d_odata, *d_block_sums;
    CUDA_SAFE_CALL(hipMalloc((void**) &d_idata, bytes));
    CUDA_SAFE_CALL(hipMalloc((void**) &d_odata, bytes));
+-- 25 lines: CUDA_SAFE_CALL(hipMalloc((void**) &d_block_sums, num
        for (int j = 0; j < iters; j++)
        {
            // For scan, we use a reduce-then-scan approach

            // Each thread block gets an equal portion of the
            // input array, and computes the sum.
            hipLaunchKernelGGL((reduce<T, 1024>), dim3(num_blocks)

            // Next, a top-level exclusive scan is performed on th
            // of block sums
            hipLaunchKernelGGL((scan_single_block<T, 1024>), dim3(

            // Finally, a bottom-level scan is performed by each b
            // that is seeded with the scanned value in block sums
            hipLaunchKernelGGL((bottom_scan<T, vecT, 1024>), dim3(
```

```
template <class T, class vecT>
void RunTest(string testName, ResultDatabase &resultDB, OptionPar
{
    int probSizes[4] = { 1, 8, 32, 64 };

    int size = probSizes[op.getOptionInt("size")-1];

    // Convert to MiB
    size = (size * 1024 * 1024) / sizeof(T);

+-- 15 lines: create input data on CPU-----------------------------
        h_idata[i] = i % 2; // Fill with some pattern
        h_odata[i] = -1;
    }

    // Thread configuration
    // Note: changing this may require updating the kernel calls
    int num_blocks  = 64;
    int num_threads = 256;
    int smem_size = sizeof(T) * num_threads;

    // Allocate device memory
    T* d_idata, *d_odata, *d_block_sums;
    CUDA_SAFE_CALL(hipMalloc((void**) &d_idata, bytes));
    CUDA_SAFE_CALL(hipMalloc((void**) &d_odata, bytes));
+-- 25 lines: CUDA_SAFE_CALL(hipMalloc((void**) &d_block_sums, nu
        for (int j = 0; j < iters; j++)
        {
            // For scan, we use a reduce-then-scan approach

            // Each thread block gets an equal portion of the
            // input array, and computes the sum.
            hipLaunchKernelGGL((reduce<T, 256>), dim3(num_blocks)

            // Next, a top-level exclusive scan is performed on t
            // of block sums
            hipLaunchKernelGGL((scan_single_block<T, 256>), dim3(

            // Finally, a bottom-level scan is performed by each
            // that is seeded with the scanned value in block sum
            hipLaunchKernelGGL((bottom_scan<T, vecT, 256>), dim3(
```

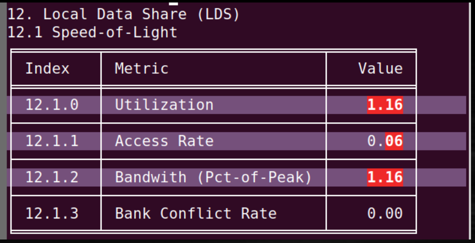
Optimized



12. Local Data Share (LDS)
12.1 Speed-of-Light

Index	Metric	Value
12.1.0	Utilization	6.15
12.1.1	Access Rate	0.20
12.1.2	Bandwith (Pct-of-Peak)	6.22
12.1.3	Bank Conflict Rate	0.00

Baseline

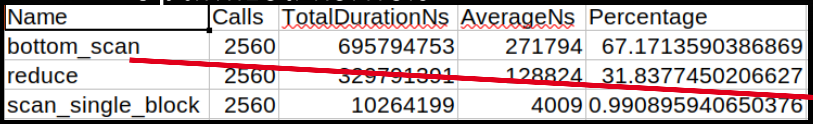


12. Local Data Share (LDS)
12.1 Speed-of-Light

Index	Metric	Value
12.1.0	Utilization	1.16
12.1.1	Access Rate	0.06
12.1.2	Bandwith (Pct-of-Peak)	1.16
12.1.3	Bank Conflict Rate	0.00

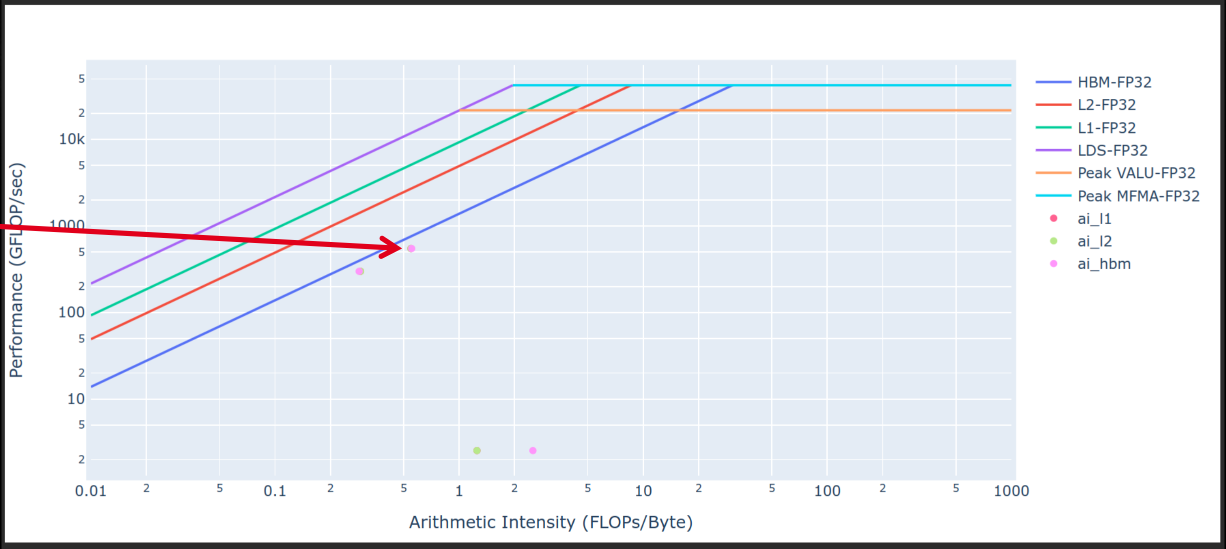
SHOC – Optimized Scan Rooflines

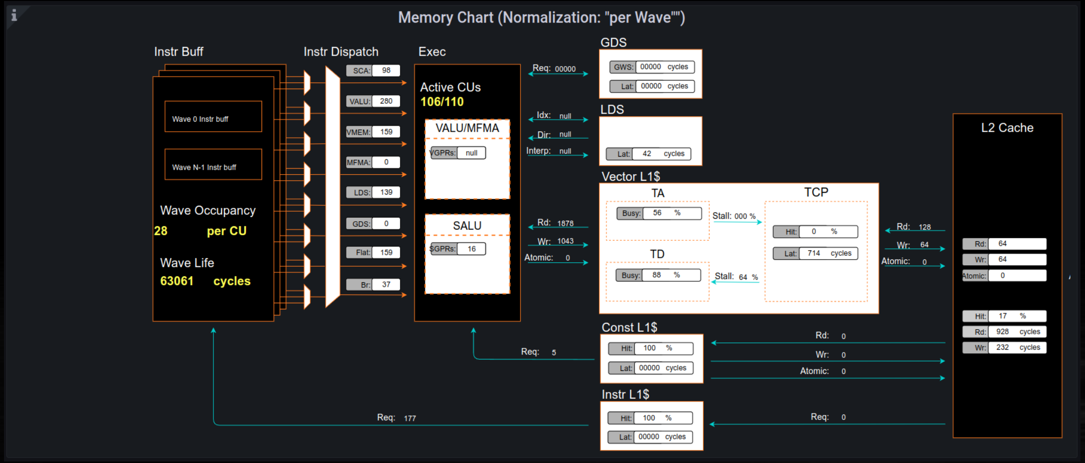
- Scan
  - Optimized kernels



Name	Calls	TotalDurationNs	AverageNs	Percentage
bottom_scan	2560	695794753	271794	67.1713590386869
reduce	2560	329791391	128824	31.8377450206627
scan_single_block	2560	10264199	4009	0.990895940650376

- Better LDS Utils: 2217 Gb/s (9% peak)
- Occupancy improved
  - 106/110 active CUs (96% peak)
  - 2970 Wavefronts (84% peak)
- HBM: 333.5 GB/s

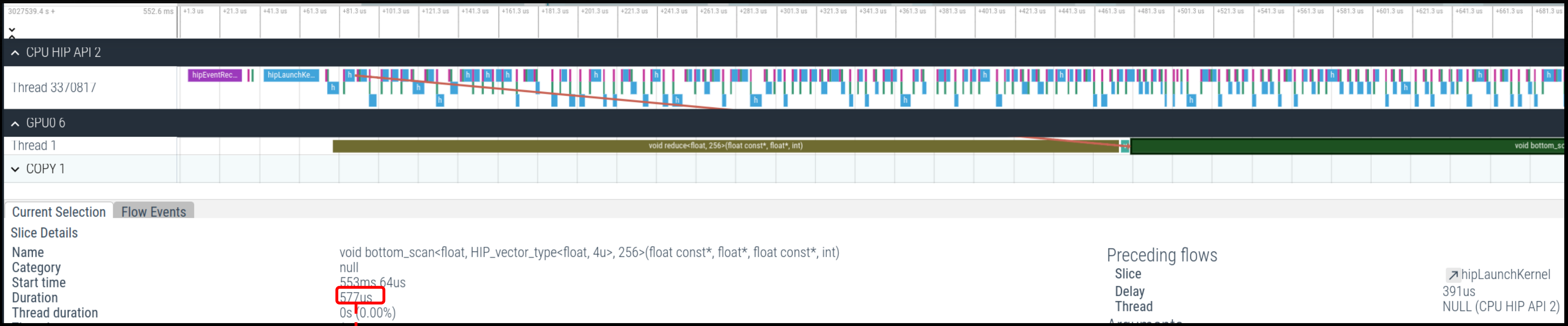



Memory Chart (Normalization: "per Wave"")

SHOC – Optimized Scan timelines

- Reduced kernel duration and launch times

Baseline Scan



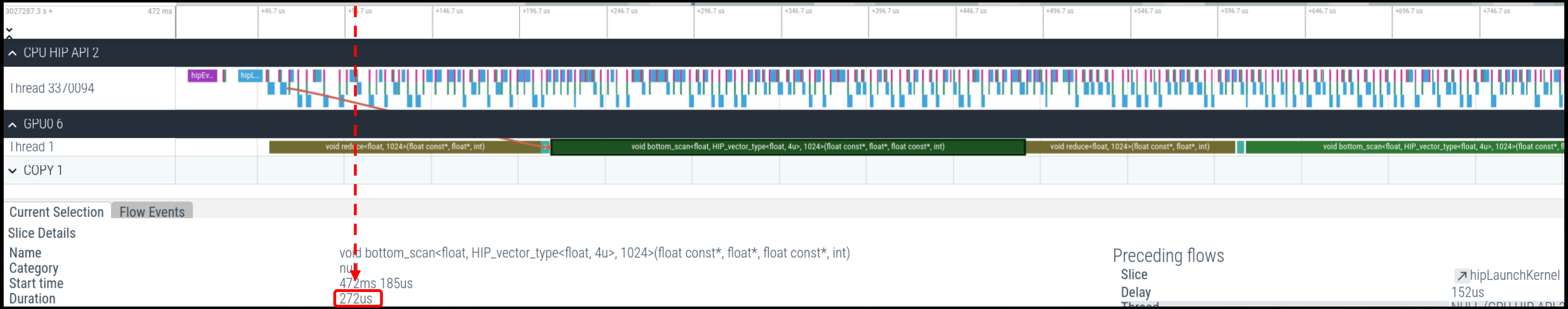
Slice Details

Name	void bottom_scan<float, HIP_vector_type<float, 4u>, 256>(float const*, float*, float const*, int)
Category	null
Start time	553ms 64us
Duration	577us
Thread duration	0s (0.00%)

Preceding flows

Slice	↗hipLaunchKernel
Delay	391us
Thread	NULL (CPU HIP API 2)

Optimized Scan


3027287.3 s +
472 ms
CPU HIP API 2
Thread 3370094
hipEv...
hipL...
GPU0 6
Thread 1
void reduce<float, 1024>(float const*, float*, int)
void bottom_scan<float, HIP_vector_type<float, 4u>, 1024>(float const*, float*, float const*, int)
void reduce<float, 1024>(float const*, float*, int)
void bottom_scan<float, HIP_vector_type<float, 4u>, 1024>(float const*, f
COPY 1
Current Selection
Flow Events

Slice Details

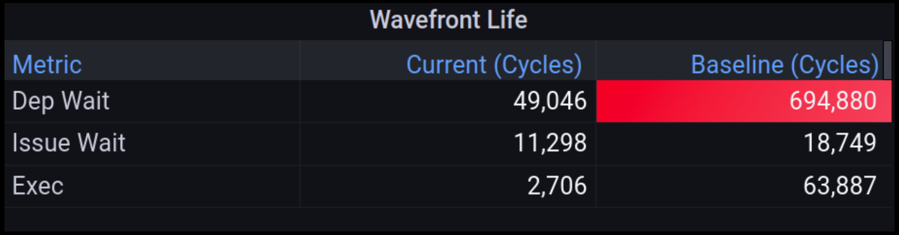
Name	void bottom_scan<float, HIP_vector_type<float, 4u>, 1024>(float const*, float*, float const*, int)
Category	null
Start time	472ms 185us
Duration	272us

Preceding flows

Slice	↗hipLaunchKernel
Delay	152us

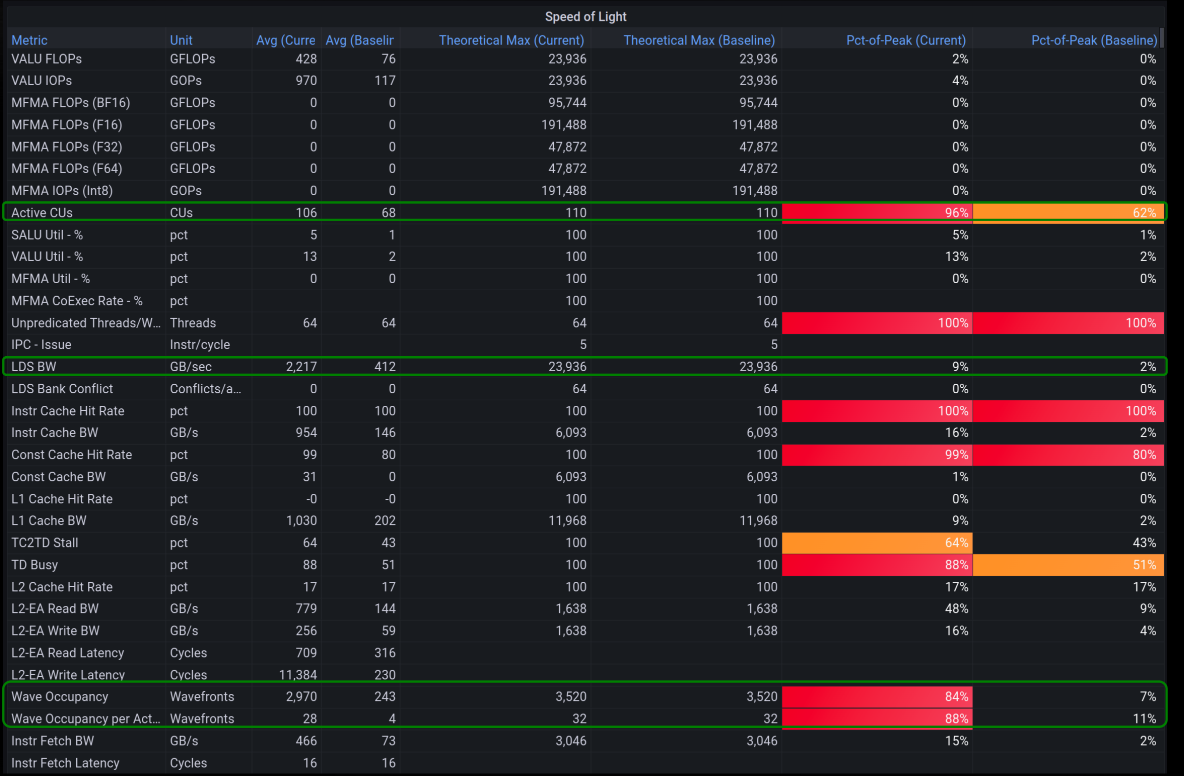
SHOC – Optimized vs Baseline Scan

- Improved VALU FLOPs/IOPs
- Increased LDS BW
- Increased occupancy (Active CUs, Wave occupancy)



Wavefront Life

Metric	Current (Cycles)	Baseline (Cycles)
Dep Wait	49,046	694,880
Issue Wait	11,298	18,749
Exec	2,706	63,887



Speed of Light

Metric	Unit	Avg (Curre	Avg (Baselir	Theoretical Max (Current)	Theoretical Max (Baseline)	Pct-of-Peak (Current)	Pct-of-Peak (Baseline)
VALU FLOPs	GFLOPs	428	76	23,936	23,936	2%	0%
VALU IOPs	GOPs	970	117	23,936	23,936	4%	0%
MFMA FLOPs (BF16)	GFLOPs	0	0	95,744	95,744	0%	0%
MFMA FLOPs (F16)	GFLOPs	0	0	191,488	191,488	0%	0%
MFMA FLOPs (F32)	GFLOPs	0	0	47,872	47,872	0%	0%
MFMA FLOPs (F64)	GFLOPs	0	0	47,872	47,872	0%	0%
MFMA IOPs (Int8)	GOPs	0	0	191,488	191,488	0%	0%
Active CUs	CUs	106	68	110	110	96%	62%
SALU Util - %	pct	5	1	100	100	5%	1%
VALU Util - %	pct	13	2	100	100	13%	2%
MFMA Util - %	pct	0	0	100	100	0%	0%
MFMA CoExec Rate - %	pct			100	100		
Unpredicated Threads/W...	Threads	64	64	64	64	100%	100%
IPC - Issue	Instr/cycle			5	5		
LDS BW	GB/sec	2,217	412	23,936	23,936	9%	2%
LDS Bank Conflict	Conflicts/a...	0	0	64	64	0%	0%
Instr Cache Hit Rate	pct	100	100	100	100	100%	100%
Instr Cache BW	GB/s	954	146	6,093	6,093	16%	2%
Const Cache Hit Rate	pct	99	80	100	100	99%	80%
Const Cache BW	GB/s	31	0	6,093	6,093	1%	0%
L1 Cache Hit Rate	pct	-0	-0	100	100	0%	0%
L1 Cache BW	GB/s	1,030	202	11,968	11,968	9%	2%
TC2TD Stall	pct	64	43	100	100	64%	43%
TD Busy	pct	88	51	100	100	88%	51%
L2 Cache Hit Rate	pct	17	17	100	100	17%	17%
L2-EA Read BW	GB/s	779	144	1,638	1,638	48%	9%
L2-EA Write BW	GB/s	256	59	1,638	1,638	16%	4%
L2-EA Read Latency	Cycles	709	316				
L2-EA Write Latency	Cycles	11,384	230				
Wave Occupancy	Wavefronts	2,970	243	3,520	3,520	84%	7%
Wave Occupancy per Act...	Wavefronts	28	4	32	32	88%	11%
Instr Fetch BW	GB/s	466	73	3,046	3,046	15%	2%
Instr Fetch Latency	Cycles	16	16				

Evolving programming models

COHERENT PROGRAMMING MODEL: SIMPLICITY

CPU CODE

```
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);



for (int i=0; i<M; i++) //initialize
   in_h[i] = …;

cpu_func(in_d, out_d, M);



for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```

GPU CODE

```
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);
hipMalloc(&in_d, Msize);
hipMalloc(&out_d, Msize);

for (int i=0; i<M; i++) //initialize
   in_h[i] = …;
hipMemcpy(in_d,in_h,Msize);
gpu_func<< >>(in_d, out_d, M);
hipDeviceSynchronize();
hipMemcpy(out_h,out_d,Msize);

for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```

COHERENT CODE

```
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);



for (int i=0; i<M; i++) //initialize
   in_h[i] = …;

gpu_func<< >>(in_h, out_h, M);
hipDeviceSynchronize();


for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```

- GPU memory allocation on Device
- Explicit memory management between CPU & GPU
- Synchronization Barrier

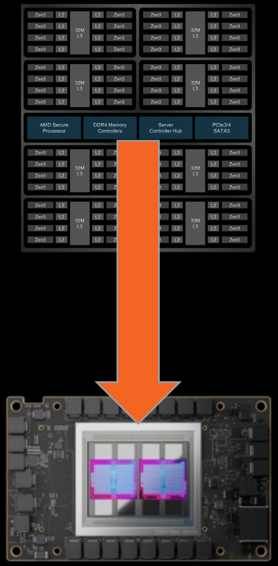
AMD
together we advance_

[Public]

COHERENT PROGRAMMING MODEL: PERFORMANCE

COHERENT CODE

```
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);



for (int i=0; i<M; i++) //initialize
   in_h[i] = …;

gpu_func<< >>(in_d, out_d, M);
hipDeviceSynchronize();


for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```



Operation	MI250X (MCM)
Coherent access over Infinity Fabric	56 GB/s

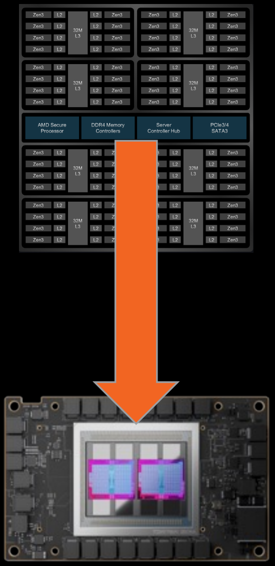
- ~~GPU memory allocation on Device~~
- ~~Explicit memory management between CPU & GPU~~
- Synchronization Barrier

AMD
together we advance_

OPENMP® TARGET OFFLOAD

COHERENT CODE

```
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);



for (int i=0; i<M; i++) //initialize
   in_h[i] = …;

#pragma omp requires unified_shared_memory

#pragma omp target
{
…
}


for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```



- Runtime knows it can omit copies and map clauses
- (Implicit) Synchronization Barrier

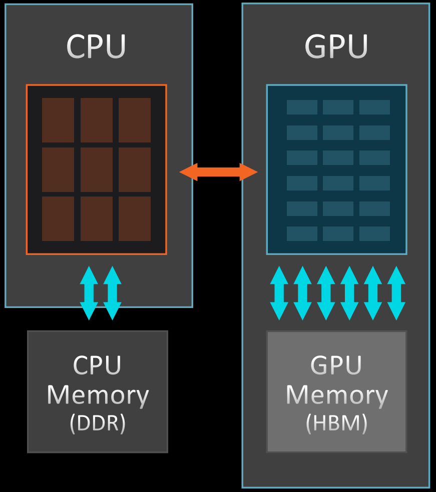
AMD
together we advance_

AMD
MI300

UNIFIED MEMORY APU ARCHITECTURE BENEFITS

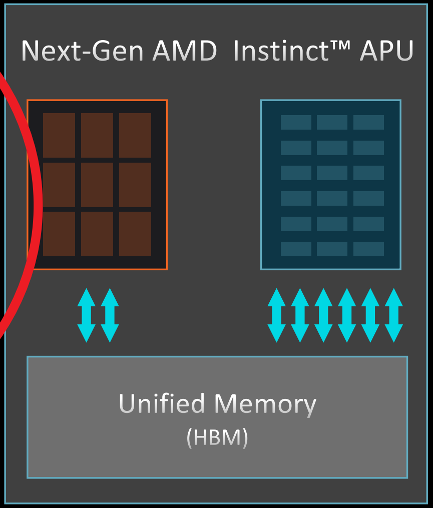
AMD CDNA™ 2 Coherent Memory Architecture

- Simplifies Programming
- Low Overhead 3rd Gen Infinity Interconnect
- Modular System Design



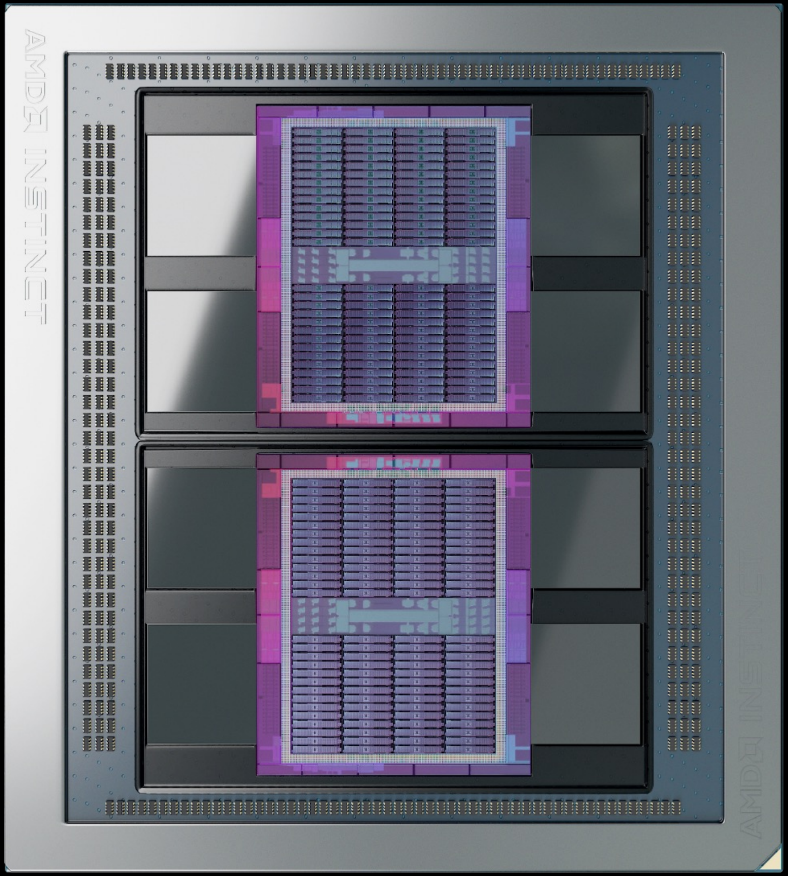
AMD CDNA™ 3 Unified Memory APU Architecture

- Eliminates Redundant Memory Copies
- High-Efficiency 4th Gen AMD Infinity Architecture
- Low TCO with Unified Memory APU Package



AMD

Improving communication in distribute ML



AMD INSTINCT™ MI250X layout

58B
Transistors in 6nm

220
Compute Units

880
2nd Gen Matrix Cores

128
GB HBM2E @ 3.2 TB/s

https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf

[Public]

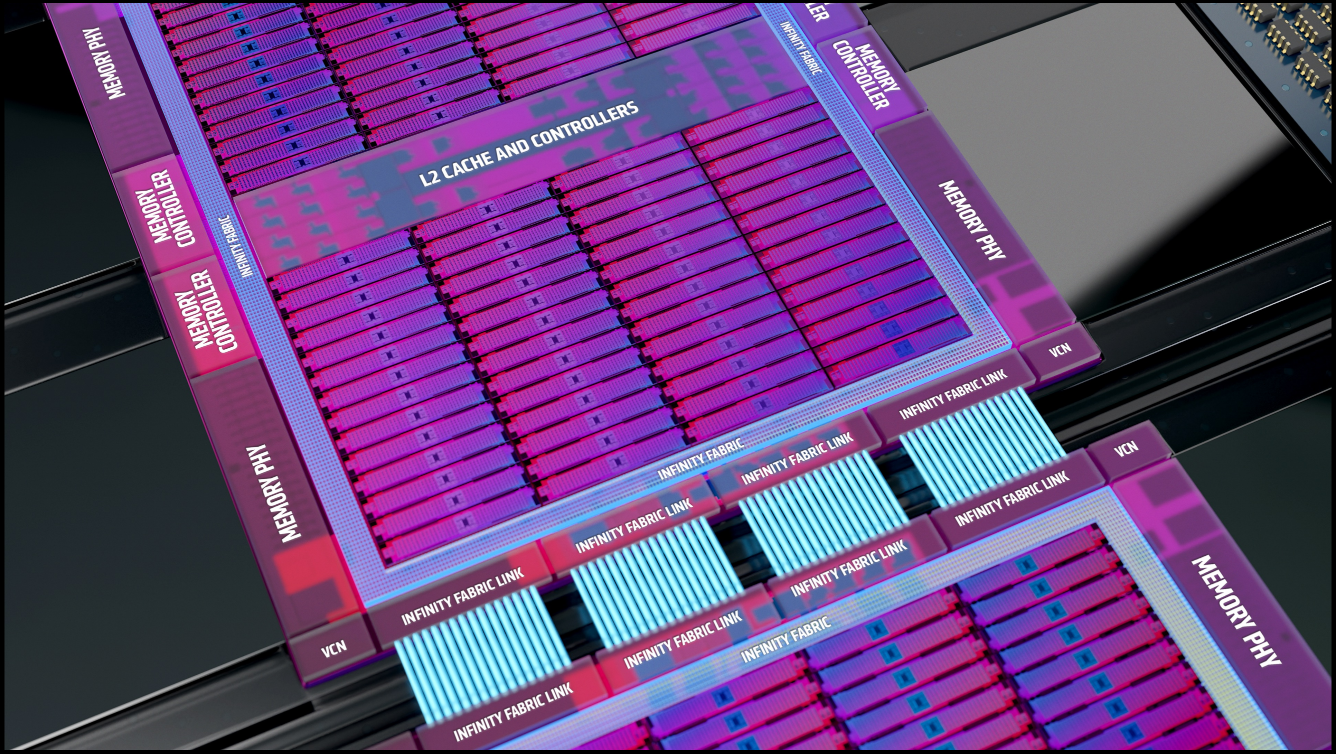
MULTI-CHIP DESIGN

TWO GPU DIES IN PACKAGE TO MAXIMIZE COMPUTE & DATA THROUGHPUT

INFINITY FABRIC FOR CROSS-DIE CONNECTIVITY

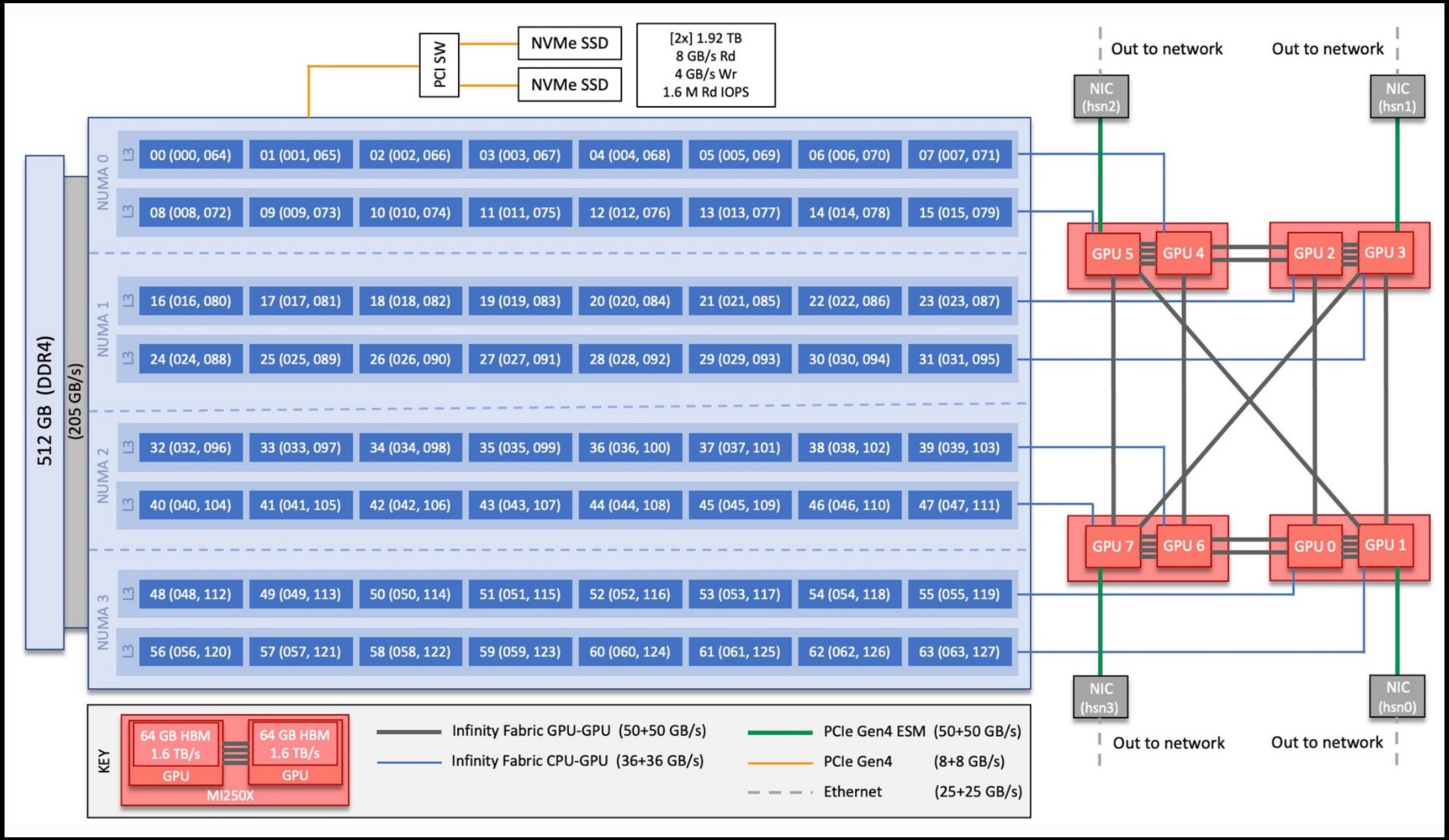
4 LINKS RUNNING AT 25GBPS

400GB/S OF BI-DIRECTIONALBANDWIDTH



AMD
together we advance_

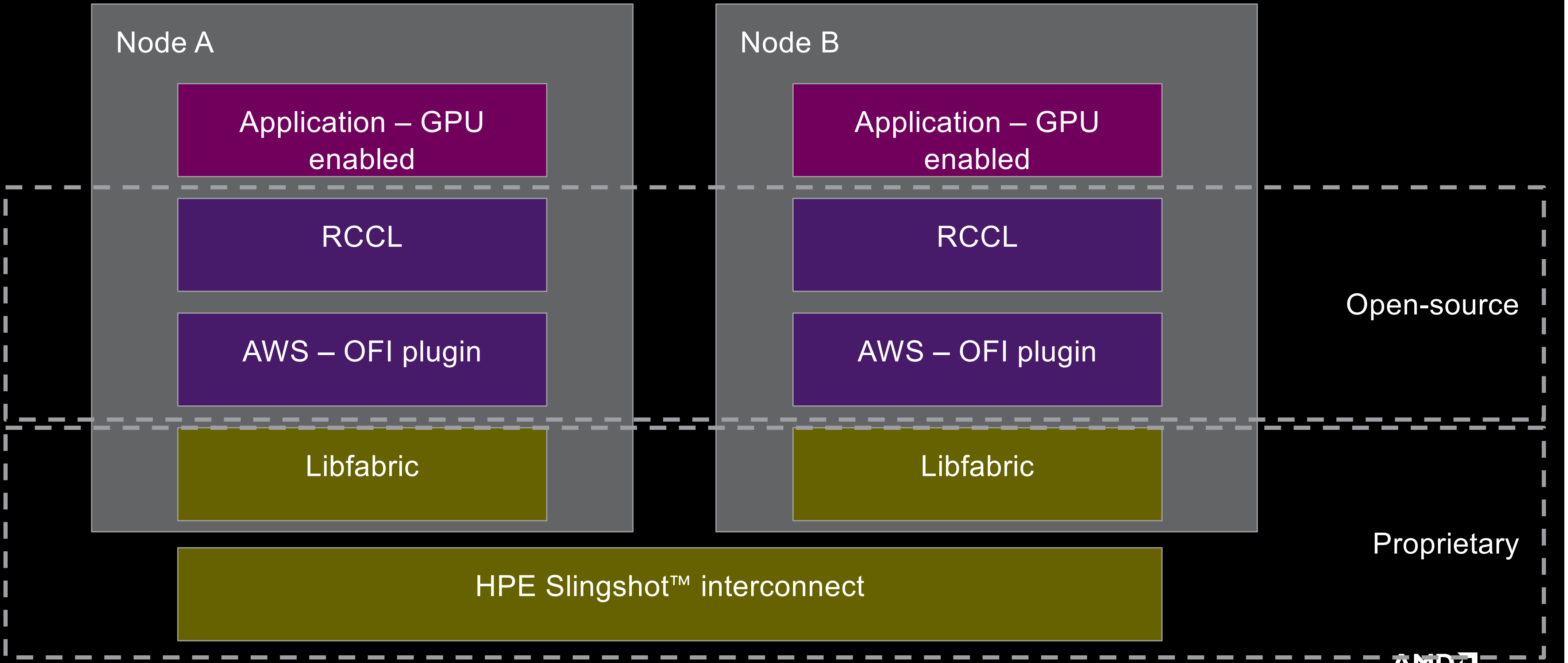
Frontier/LUMI node layout



https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html

Frontier/Lumi network stack

- **Libfabric/OFI (Open Fabrics Interface) RCCL plug in**
- **https://github.com/ROCmSoftwarePlatform/aws-ofi-rccl**

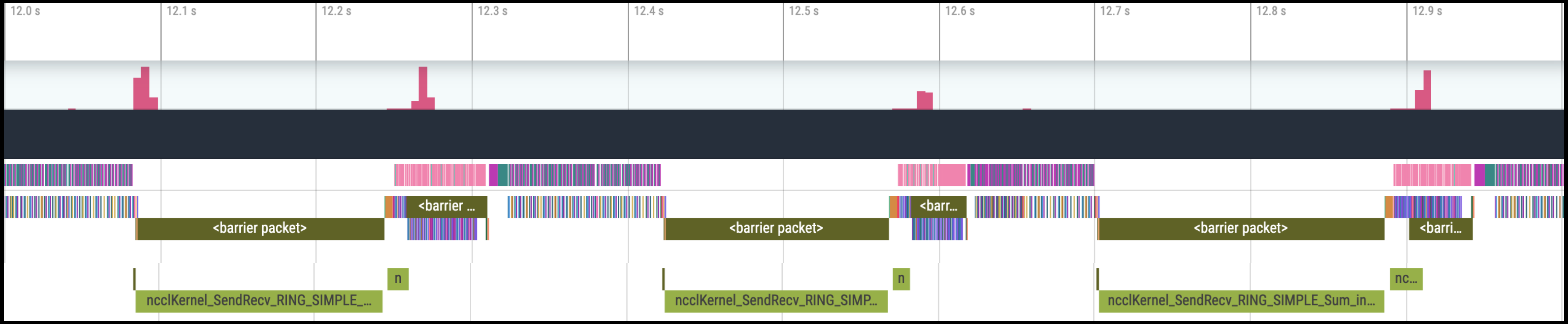


AMD
together we advance_

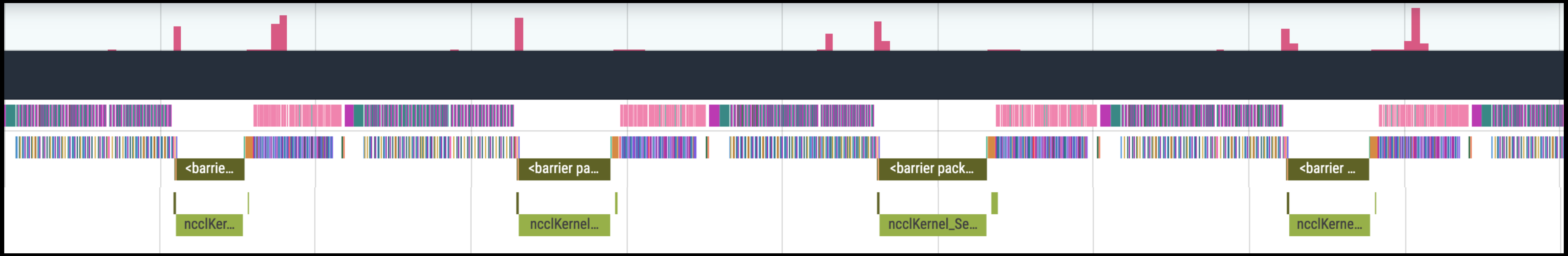
Frontier/Lumi network stack

- **Libfabric/OFI (Open Fabrics Interface) RCCL plug in – over 3-4x speedup**

No plugin – sockets-based implementation



Plug-in enabled – leveraging Libfabric implementation



Questions?

DISCLAIMERS AND ATTRIBUTIONS

AMD

Back-up