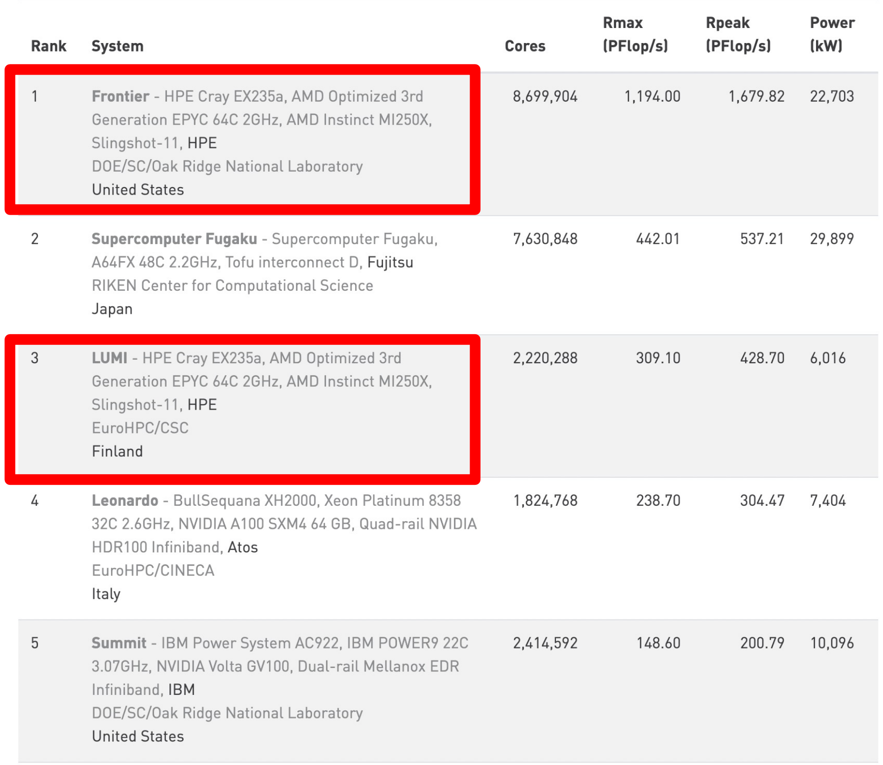
Introduction to AMD GPUs

Ivy Peng
Assistant Professor in Computer Science
Scalable Parallel System (ScaLab)
Department of Computer Science, KTH

AMD GPUs

U.S. DEPARTMENT OF ENERGY
NNSA
Lawrence Livermore National Laboratory
AMD
CRAY
EL CAP

TOP500 The List.

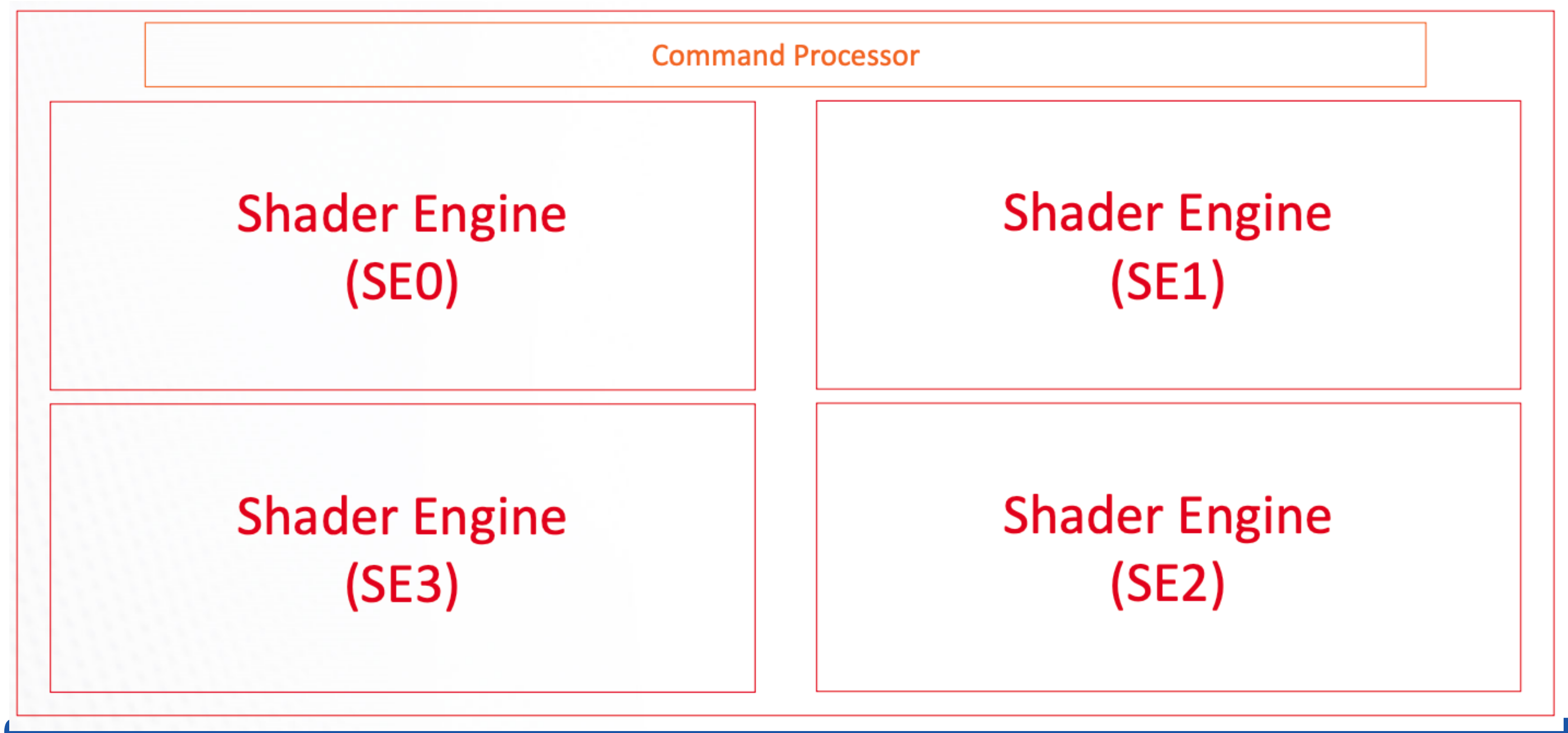


Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,824,768	238.70	304.47	7,404
5	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096

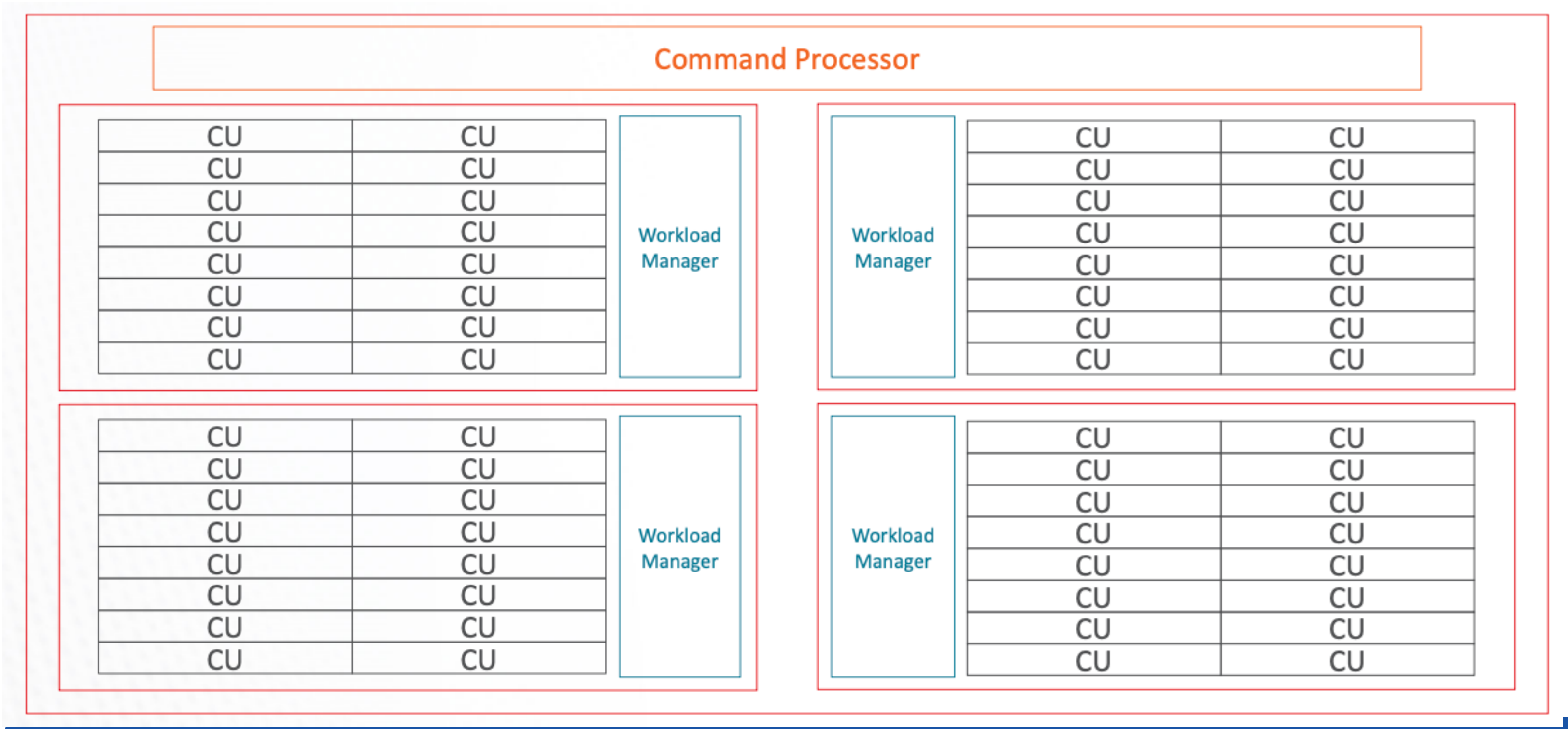
AMD GPUs

- AMD is a strong Nvidia competitor for High-performance data centers
- AMD Instinct MI250x powers upcoming supercomputers, such as Frontiers, El Capitan, LUMI and Dardel
  - Its performance is better / on-par than Nvidia A100.

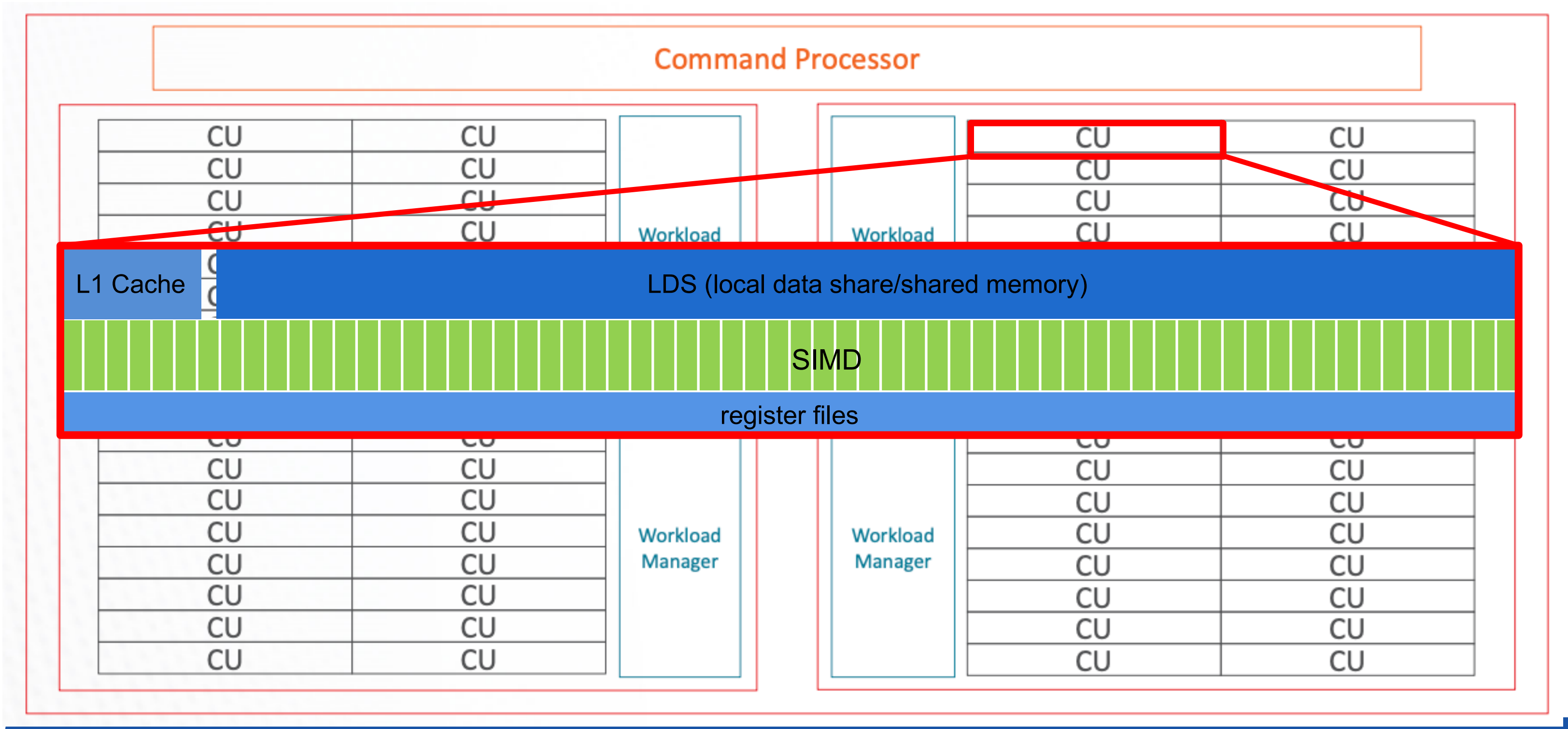
AMD GCN Architecture



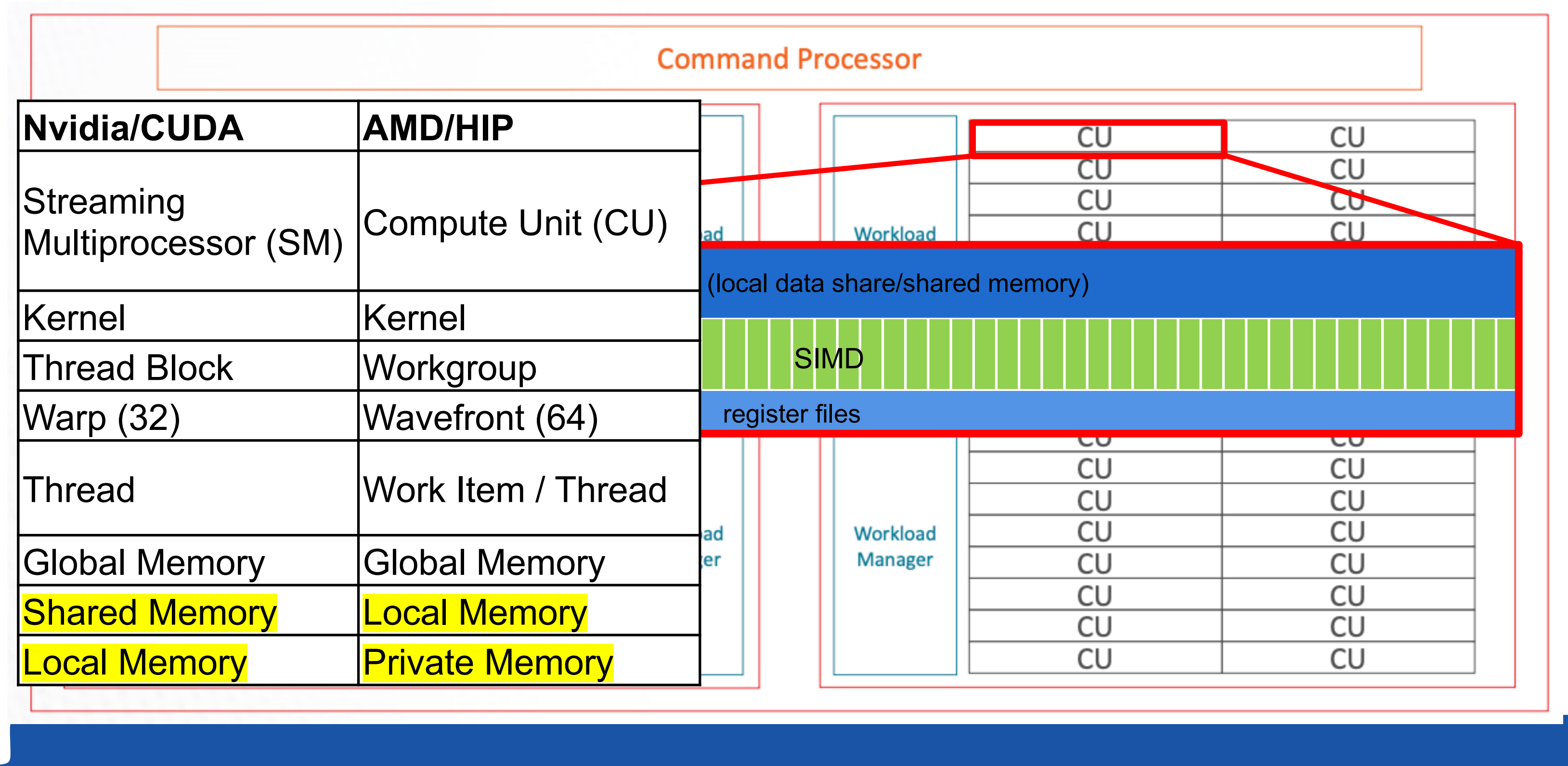
AMD GCN Architecture



AMD Instinct MI250x GPU



AMD Instinct MI250x GPU



Nvidia/CUDA	AMD/HIP
Streaming Multiprocessor (SM)	Compute Unit (CU)
Kernel	Kernel
Thread Block	Workgroup
Warp (32)	Wavefront (64)
Thread	Work Item / Thread
Global Memory	Global Memory
Shared Memory	Local Memory
Local Memory	Private Memory

AMD Instinct MI250x GPU

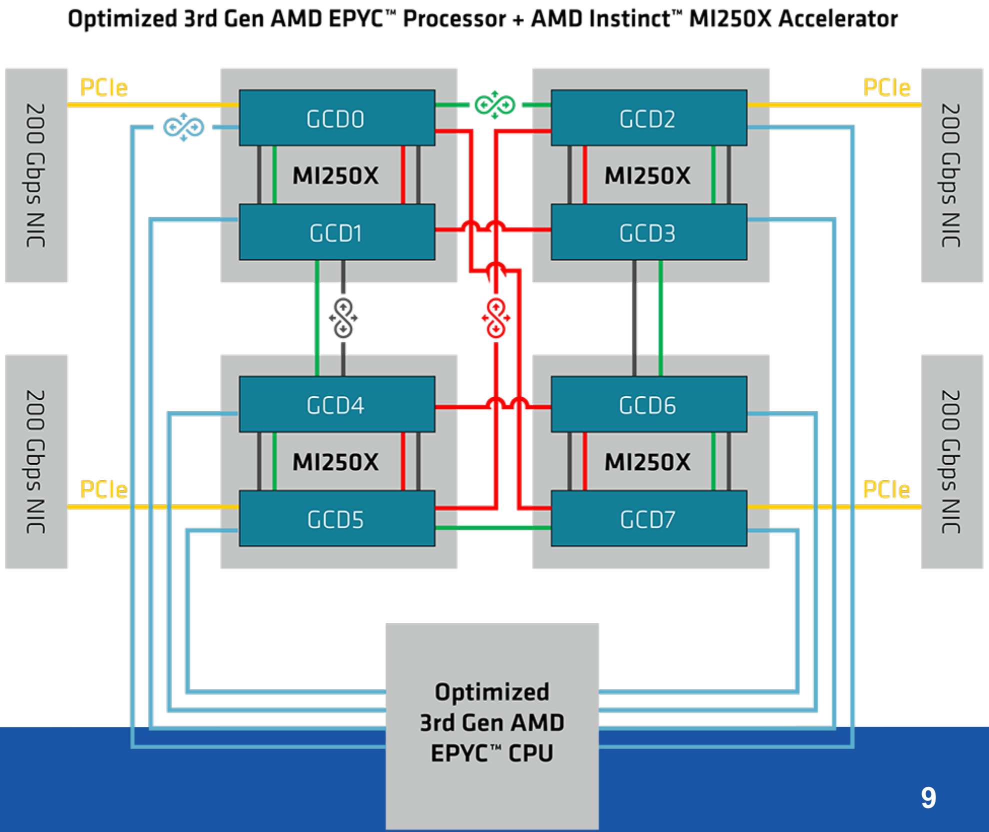
- Each GPU has 2 Graphic Compute Dies (GCD)
- The CUs feature Matrix Core Engines optimized for the matrix in machine learning, similar to Nvidia Tensor Cores

MI250X	
Compute Units	220 (110 per GCD)
Stream Processors	14080
MEMORY	
Memory Size	128GB HBM2e
Memory Interface	8,192 bits
Memory Clock	1.6GHz
Memory Bandwidth	up to 3.2TB/sec2
PERFORMANCE	
Peak FP64/FP32 Vector	47.9 TFLOPS
Peak FP64/FP32 Matrix	95.7 TFLOPS
Peak FP16/BF16	383.0 TFLOPS
Peak INT4/INT8	383.0 TOPS

The Dardel Supercomputer

The GPU partition comprises 56 GPU nodes. Each node has:

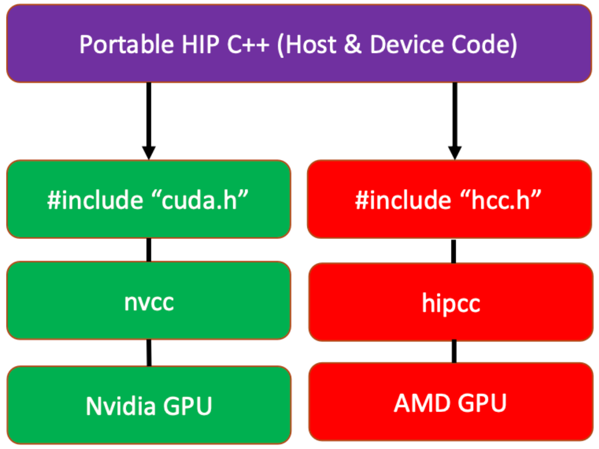
- CPU: one AMD EPYC™ 64-core processors (128 hardware threads)
- 512 GB of shared fast HBM2e memory (128GB x 4 GPUs)
- 4 AMD Instinct™ MI250X GPUs (8 GCD) connected by AMD Infinity Fabric® links



HIP, AMD’s portable layer

AMD’s Heterogeneous-compute Interface for Portability - HIP

- HIP is a C++ runtime API and kernel language
- Provides an API for an application to leverage GPU acceleration for both AMD and CUDA devices.
- Syntactically similar to CUDA.
    - Many CUDA applications can be easily converted to HIP using *hipify*



source: introduction to AMD GPU programming with hip”, paul bauman et al.

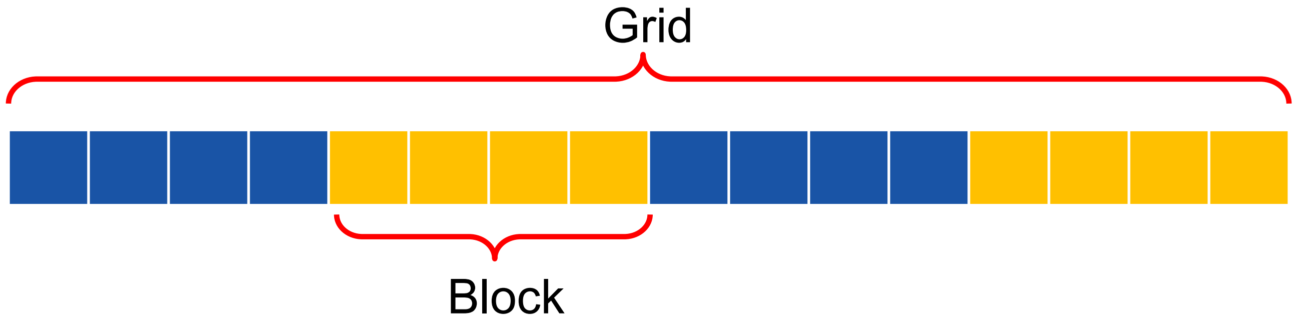
HIP API (v.s. CUDA Runtime API)

	HIP	CUDA
Device Management	hipSetDevice(), hipGetDevice(), hipGetDeviceProperties()	cudaSetDevice(), cudaGetDevice(), cudaGetDeviceProperties()
Memory Management	hipMalloc(), hipMemcpy(), hipMemcpyAsync(), hipFree()	cudaMalloc(), cudaMemcpy(), cudaMemcpyAsync(), cudaFree()
Device Kernels	__global__, __device__ hipLaunchKernelGGL()	__global__, __device__ aKernel<<<,>>>()
Synchronization	hipDeviceSynchronize()	cudaDeviceSynchronize();

Computational Grid – 1D

Similar to CUDA, we can calculate a thread's global id by

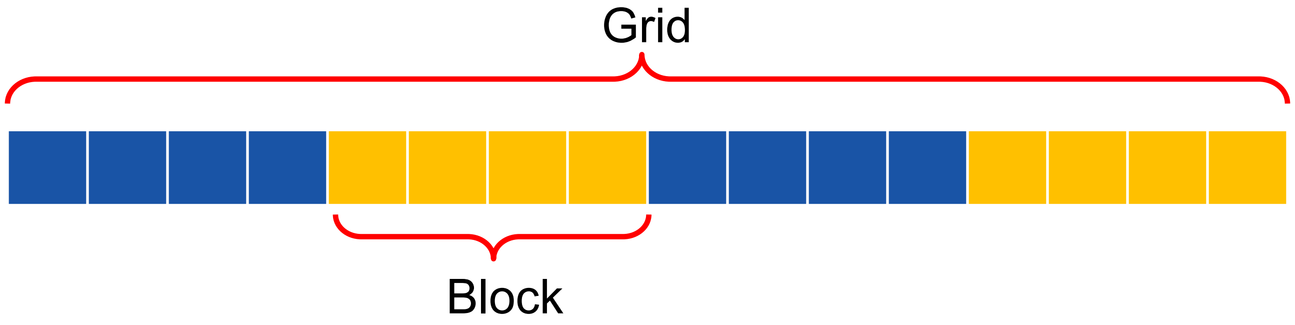
- Its block ID: blockIdx.x
- the block's dimension: blockDim.x
- Its local thread ID in a block: threadIdx.x



Computational Grid – 2D, 3D

Similar to CUDA, we use the .y .z index

- Its block ID: blockIdx.x , blockIdx.y , blockIdx.z
- the block's dimension: blockDim.x , blockDim.y , blockDim.z
- Its local thread ID in a block: threadIdx.x , threadIdx.y , threadIdx.z



A good practice is to make the block size a multiple of 64 (wavefront=64)

Device Memory

Similar to Nvidia GPU and CUDA:

```
int main() {
   ...
   //allocate on device memory
   double *d_a = NULL;
   hipMalloc(&d_a, Nbytes);
   ...
   //copy data into device memory
   hipMemcpy(d_a,h_a,Nbytes,hipMemcpyHostToDevice);
   ...
   //copy data from device memory to host
   hipMemcpy(h_a,d_a,Nbytes,hipMemcpyDeviceToHost);
   ...
   //free device memory
   hipFree(d_a);
}
```

Kernel Launch

```
__global__ void myKernel(int N, double *d_a) { . . .}

dim3 threads(256,1,1); //3D dimensions of a block of threads
dim3 blocks((N+256-1)/256,1,1); //3D dimensions the grid of blocks

hipLaunchKernelGGL(myKernel,
                   blocks, threads,
                   bytes_LDS, //equal:CUDA dynamic shared memory
                   stream_id, //equal:CUDA stream
                   kernel_args…);
```

Equivalent to CUDA:

```
myKernel<<<blocks, threads, SMEM, stream_id>>>(N,a);
```

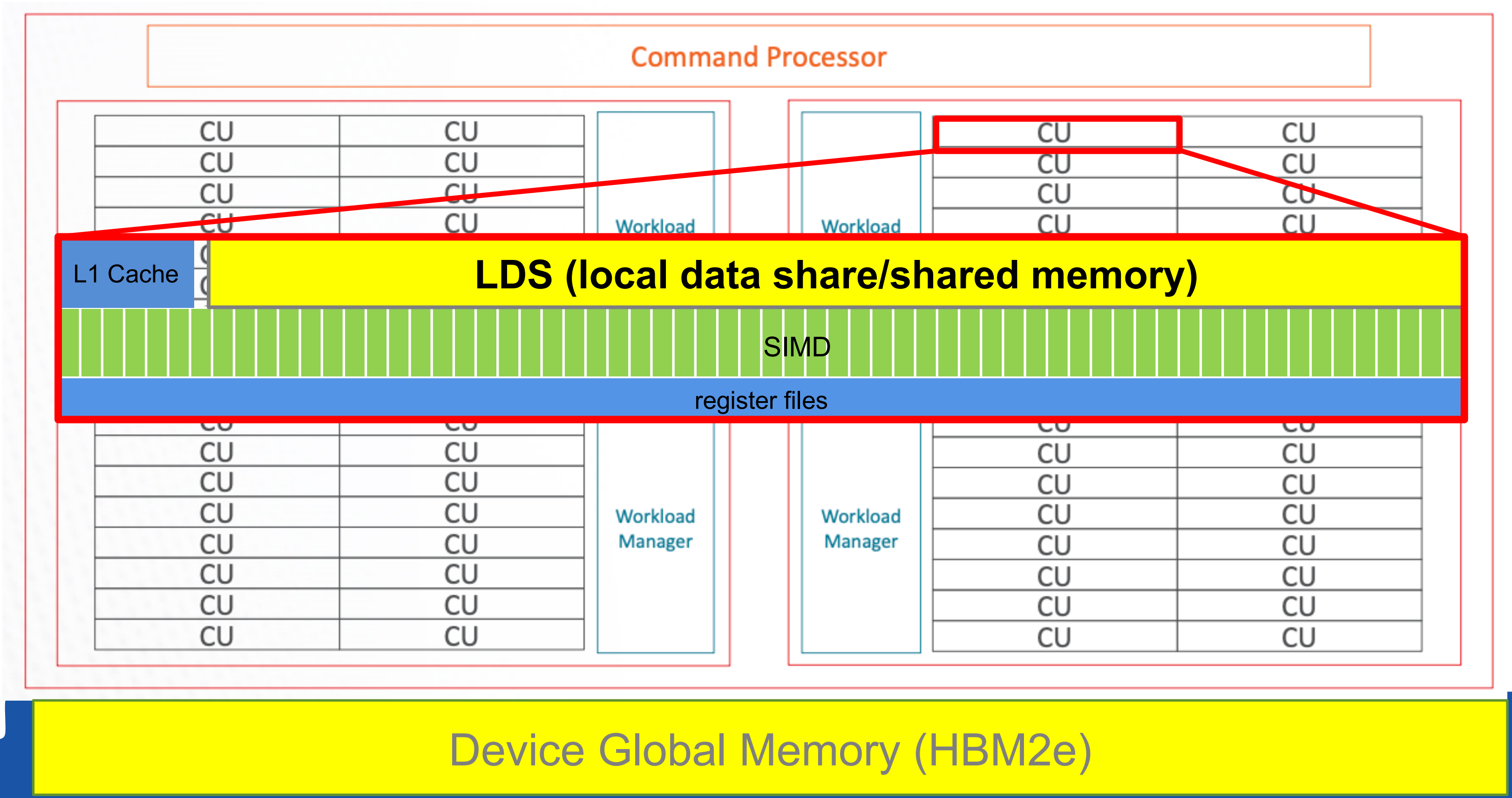
HIP Code Skeleton

```
__global__ void myKernel(args…)
{
  ...
  //calculate threads's global ID
  int tid = threadIdx.x + blockIdx.x*blockDim.x;
  ...
}
```

```
#include "hip/hip_runtime.h"
int main()
{
   ...
   //allocate on device memory
   double *d_a = NULL;
   hipMalloc(&d_a, Nbytes);
   ...
   //copy data into device memory

hipMemcpy(d_a,h_a,Nbytes,hipMemcpyHostToDevice);
   ...
   //define thread block and launch kernel
   dim3 threads(256,1,1);
   dim3 blocks((N+256-1)/256,1,1);
   hipLaunchKernelGGL(myKernel...)
   ...
   //copy data from device memory to host

hipMemcpy(h_a,d_a,Nbytes,hipMemcpyDeviceToHost);
   ...
   //free device memory
   hipFree(d_a);
}
```

Shared Memory (LDS)

Access is faster than device memory but slower than register



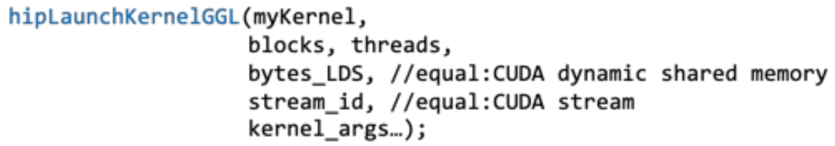
Command Processor
CU
Workload Manager
L1 Cache
LDS (local data share/shared memory)
SIMD
register files
Device Global Memory (HBM2e)

Shared Memory (LDS)

The same keyword as in CUDA, variables are declared as :

```
__shared__ double s_a[256];
```

- ➢ Allocated on LDS memory
- ➢ Shared and accessible by all threads in the same block
- ➢ Best practice to use __syncthreads to ensure all threads in the same block reached the same step
- ➢ Either use known size at compile time or dynamic allocate in kernel launch



```
hipLaunchKernelGGL(myKernel,
                   blocks, threads,
                   bytes_LDS, //equal:CUDA dynamic shared memory
                   stream_id, //equal:CUDA stream
                   kernel_args…);
```

Atomics Operations

- Perform a read+write of a single 32 or 64-bit word in device global or LDS memory
- Work as a solution to race condition when data is updated by multiple threads

Operation	**Type T**
T atomicAdd(T* address, T val)	int, long long int, float, double
T atomicExch(T* address, T val)	int, long long int, float
T atomicMin(T* address, T val)	int, long long int
T atomicMax(T* address, T val)	int, long long int
T atomicAnd(T* address, T val)	int, long long int
T atomicOr(T* address, T val)	int, long long int
T atomicXor(T* address, T val)	int, long long int

Compile a HIP code

Compile using ***hipcc*** *and --offload to target AMD GPU, e.g.,*

```
    hipcc --offload-arch=gfx90a vecAdd.cpp -o vecAdd
```

Set HIPCC_VERBOSE=7 to see compilation information

```
    HIPCC_VERBOSE=7 hipcc --offload-arch=gfx90a
vecAdd.cpp -o vecAdd
```

Hipify a CUDA code to HIP

Many applications are already written in CUDA, i.e., *.cu

AMD provides 'Hipify' tools to automatically convert most CUDA code to HIP code.

- Hipify-perl is a script that comes in rocm module on Dardel

```
module load rocm/5.3.3
hipify-perl -print-stats hello.cu > hello.cpp
```

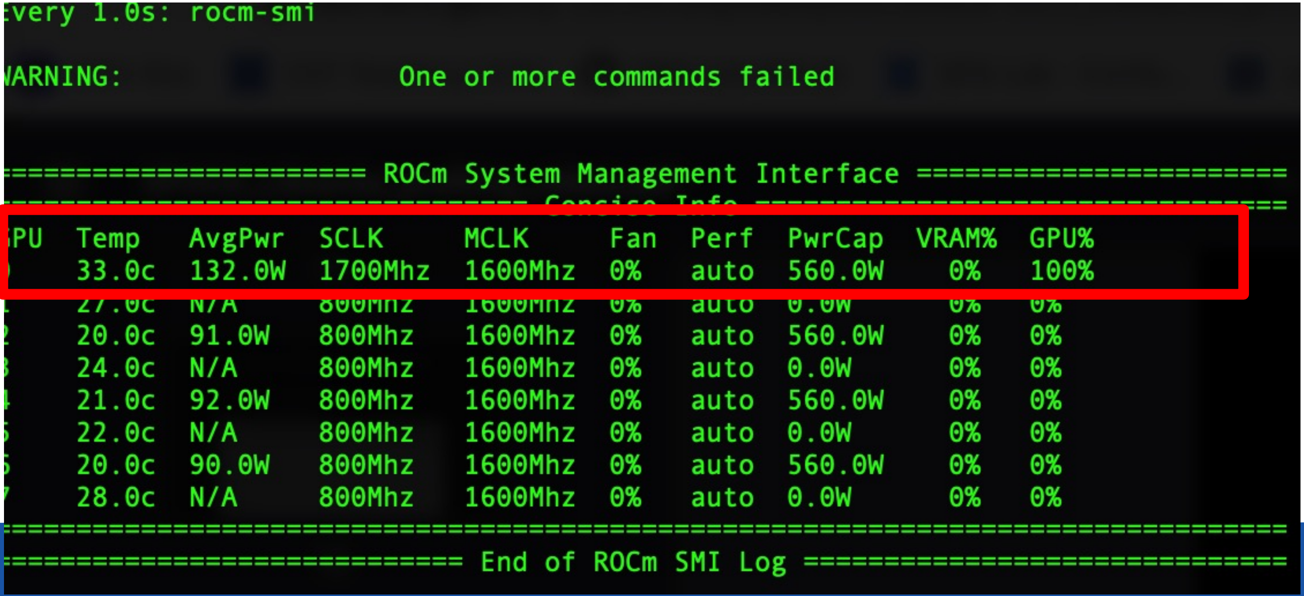
Most codes (>90%) in large applications can be automatically converted.
What cannot be hipified?

- Intrinsic code
- User inerted assembly
- Nvidia GPU specific hardcoded

Make Sure your code is running on GPU

While your code is running, check the GPU activities using rocm-smi.

In the following example, only 1 (GPU 0) out of 8 GPUs are being utilized – each GCD is presented as a GPU device



```
very 1.0s: rocm-smi

ARNING:                 One or more commands failed


====================== ROCm System Management Interface ======================
================================ Concise Info ================================
PU   Temp    AvgPwr   SCLK      MCLK      Fan   Perf  PwrCap  VRAM%  GPU%
     33.0c   132.0W   1700Mhz   1600Mhz   0%    auto  560.0W    0%   100%
     27.0c   N/A      800Mhz    1600Mhz   0%    auto  0.0W      0%   0%
     20.0c   91.0W    800Mhz    1600Mhz   0%    auto  560.0W    0%   0%
     24.0c   N/A      800Mhz    1600Mhz   0%    auto  0.0W      0%   0%
     21.0c   92.0W    800Mhz    1600Mhz   0%    auto  560.0W    0%   0%
     22.0c   N/A      800Mhz    1600Mhz   0%    auto  0.0W      0%   0%
     20.0c   90.0W    800Mhz    1600Mhz   0%    auto  560.0W    0%   0%
     28.0c   N/A      800Mhz    1600Mhz   0%    auto  0.0W      0%   0%
==============================================================================
========================== End of ROCm SMI Log ===========================
```

AMD GPU Profiling with rocprof

Profiling Reduction with rocprof - 1

Run rocprof with --stats for a summary in *results.stats.csv*

```
4_reduction> srun -n 1 rocprof --stats ./reduction
RPL: on '230811_145940' from '/cfs/klemming/root/rocm/opt/rocm-5.3.3' in ‘./amd_part2/4_reduction'
RPL: profiling '"./reduction"'
RPL: input file ''
RPL: output dir '/tmp/rpl_data_230811_145940_117170'
RPL: result dir '/tmp/rpl_data_230811_145940_117170/input_results_230811_145940'
Usage: ./reduction num_of_elems
using default value: 52428800
ARRAYSIZE: 52428800
Array size: 200 MB
ROCProfiler: input from "/tmp/rpl_data_230811_145940_117170/input.xml"
  0 metrics
The average performance of reduction is 396.195 GBytes/sec
VERIFICATION: result is CORRECT

ROCPRofiler: 20 contexts collected, output directory /tmp/rpl_data_230811_145940_117170/input_results_230811_145940
File '/amd_part2/4_reduction/results.csv' is generating
File '/amd_part2/4_reduction/results.stats.csv' is generating
```

Profiling Reduction with rocprof - 2

Run rocprof with a selected list of hardware counters specified in *my_counters.txt*

```
4_reduction> cat my_counters.txt
pmc: Wavefronts VALUInsts VFetchInsts VWriteInsts VALUUtilization VALUBusy WriteSize

4_reduction> srun -n 1 rocprof -i my_counter.txt ./reduction
…

ROCProfiler: input from "/tmp/rpl_data_230810_164433_16749/input0.xml"
  gpu_index =
  kernel =
  range =
  7 metrics
    Wavefronts, VALUInsts, VFetchInsts, VWriteInsts, VALUUtilization, VALUBusy, WriteSize

The average performance of reduction is 91.2052 GBytes/sec
VERIFICATION: result is CORRECT

ROCPRofiler: 20 contexts collected, output directory
/tmp/rpl_data_230810_164433_16749/input0_results_230810_164433
File ‘4_reduction/my_counter.csv' is generating
```

Profiling Reduction with rocprof - 3

Run rocprof to collect a trace of events and visualize the trace

1. Collect traces, a set of json files are generated

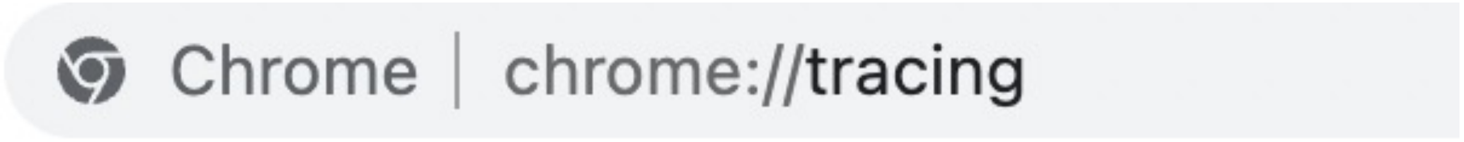
```
> srun -n 1 rocprof --hip-trace --hsa-trace ./helloworld
RPL: on '230811_161421' from '/cfs/klemming/root/rocm/opt/rocm-5.3.3' in
. . .

RPL: output dir '/tmp/rpl_data_230811_161421_1594'
RPL: result dir '/tmp/rpl_data_230811_161421_1594/input_results_230811_161421'
. . .

ROCProfiler: input from "/tmp/rpl_data_230811_161421_1594/input.xml"
  0 metrics
ROCtracer (1615):
    HSA-trace(*)
    HSA-activity-trace()
    HIP-trace(*)
. . .
```

Profiling Reduction with rocprof - 3

Run rocprof to collect a trace of events and visualize the trace

2. download result.json to your laptop

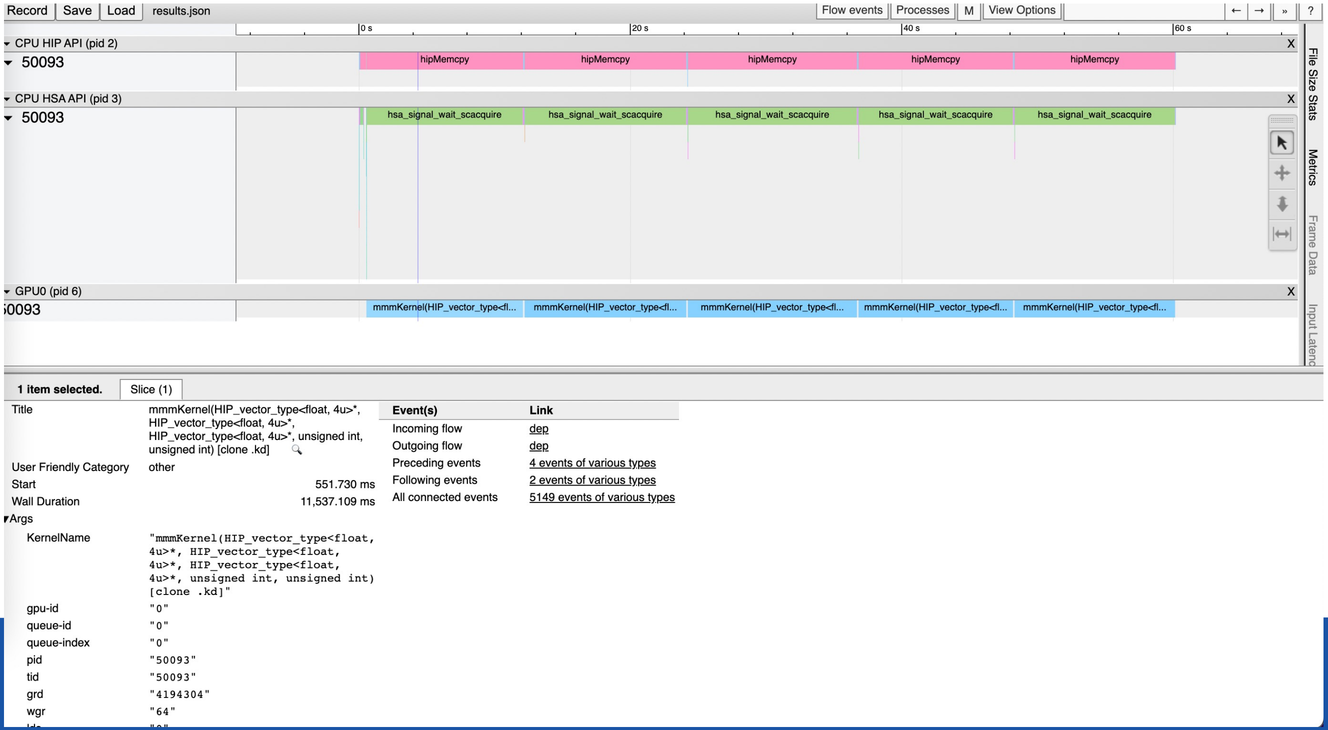
3. Open Chrome browser



Chrome | chrome://tracing

Profiling Reduction with rocprof - 3

4. Load the json file in Chrome browser



1 item selected. Slice (1)

Title	mmmKernel(HIP_vector_type<float, 4u>*, HIP_vector_type<float, 4u>*, HIP_vector_type<float, 4u>*, unsigned int, unsigned int) [clone .kd]
User Friendly Category	other
Start	551.730 ms
Wall Duration	11,537.109 ms

Event(s)	Link
Incoming flow	dep
Outgoing flow	dep
Preceding events	4 events of various types
Following events	2 events of various types
All connected events	5149 events of various types

Args

KernelName	"mmmKernel(HIP_vector_type<float, 4u>*, HIP_vector_type<float, 4u>*, HIP_vector_type<float, 4u>*, unsigned int, unsigned int) [clone .kd]"
gpu-id	"0"
queue-id	"0"
queue-index	"0"
pid	"50093"
tid	"50093"
grd	"4194304"
wgr	"64"

Use Math Libraries

Optimized Math Libraries

- **BLAS**
  - rocBLAS(https://github.com/ROCmSoftwarePlatform/rocBLAS)
  - cuBLAS(https://docs.nvidia.com/cuda/cublas/index.html)
- **FFTs**
  - rocFFT(https://github.com/ROCmSoftwarePlatform/rocFFT)
  - cuFFT(https://developer.nvidia.com/cufft)
- **Sparse linear algebra**
  - rocSPARSE(https://github.com/ROCmSoftwarePlatform/rocSPARSE)
  - cuSPARSE(https://docs.nvidia.com/cuda/cusparse/index.html)
- **Solvers**
  - rocALUTION(https://github.com/ROCmSoftwarePlatform/rocALUTION)
  - cuSOLVER(https://docs.nvidia.com/cuda/cusolver/index.html#)

BLAS (Basic Linear Algebra Subprograms)

A, B, C are matrices, x, y are vectors, alpha, beta are scalars

- BLAS level 1 vector-vector operations
  - *e.g.*, *axpy* computes constant alpha multiplied by vector x, plus vector y: vector-vector

$$y := alpha * x + y$$

- BLAS level 2 matrix-vector operations

$$y := alpha*A*x + beta*y$$

- BLAS level 3 matrix-matrix operations

$$C := alpha*A * B + beta*C$$

rocBLAS level 1: ***axpy***

rocblas_Xaxpy: X represent data types, e.g., s = single precision

y := alpha * x + y

```
rocblas_saxpy(rocblas_handle handle,
              rocblas_int n, const float *alpha,
              const float *x, rocblas_int incx, float *y, rocblas_int incy)

rocblas_daxpy(rocblas_handle handle,
              rocblas_int n, const double *alpha,
              const double *x, rocblas_int incx, double *y, rocblas_int incy)
```

Nvidia provides similar cuBLAS

```
cublasStatus_t cublasSaxpy(cublasHandle_t handle,
              int n, const float *alpha, const float *x, int incx, float *y, int incy)
```

rocBLAS level 3: *gemm*

rocblas_Xgemm: X represent data types, e.g., s = single precision

C := alpha*A * B + beta*C

```
rocblas_sgemm(rocblas_handle handle, r
              ocblas_operation transA, rocblas_operation transB,
              rocblas_int m, rocblas_int n, rocblas_int k, const float *alpha,
              const float *A, rocblas_int lda, const float *B, rocblas_int ldb,
              const float *beta, float *C, rocblas_int ldc)
```

A is a m x k matrix, B is a k x n matrix, C is a m x n matrix

Nvidia provides similar cuBLAS

```
cublasSgemm(cublasHandle_t handle,
            cublasOperation_t transa, cublasOperation_t transb,
            int m, int n, int k, const float *alpha, const float *A, int lda, const float *B,
            int ldb, const float *beta, float *C, int ldc)
```

Use rocBLAS: an example

rocblas_sgemm(*rocblas_handle* handle, *rocblas_operation* transA, *rocblas_operation* transB, *rocblas_int* m, *rocblas_int* n, *rocblas_int* k, const float *alpha, const float *A, *rocblas_int* lda, const float *B, *rocblas_int* ldb, const float *beta, float *C, *rocblas_int* ldc)

```
int main(){

//1. create rocblas handle
rocblas_handle handle;
rocblas_create_handle(&handle);

//2. memory allocation on device
hipMalloc(…)
hipMemcpy(…hipMemcpyHostToDevice)

//3. calculation using rocblas
rocblas_sgemm(handle, transA, transB, M, N, K,
&hAlpha, dA, lda, dB, ldb, &hBeta, dC, ldc);


//4. copy results from device to host
hipMemcpy(…hipMemcpyDeviceToHost)

…
}
```

Q&A