COMPREHENSIVE GENERAL LUMI COURSE
24/4/2024

AMD HARDWARE AND SOFTWARE

Suyash Tandon, Justin Chang, Julio Maia, Noel Chalmers, Paul T. Bauman, Nicholas Curtis, Nicholas Malaya, Alessandro Fanfarillo, Jose Noudohouenou, Chip Freitag, Damon McDougall, Noah Wolfe, Jakub Kurzak, Samuel Antao, George Markomanolis, Bob Robey

ADVANCED MICRO DEVICES, INC.

AMD
together we advance_

slides on LUMI in /project/project_465001098/Slides/AMD/

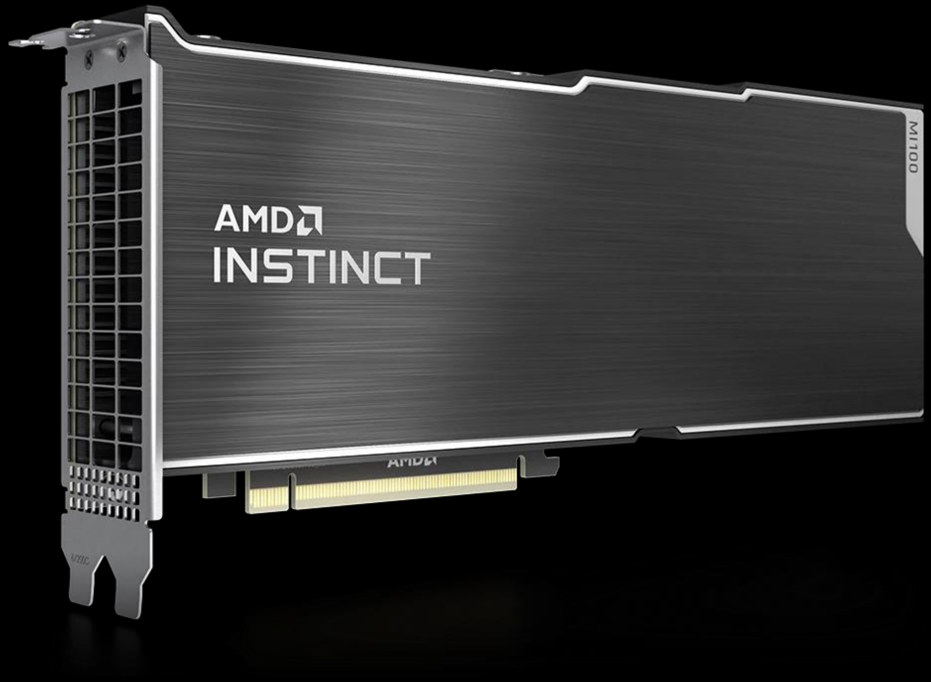
hands-on exercises: https://hackmd.io/@gmarkoma/lumi_finland

hands-on source code: /project/project_465001098/Exercises/AMD/HPCTrainingExamples/

AMD HARDWARE FOR HPC AND AI

CDNA ARCHITECTURE

AMD GPUS



Radeon™ Graphics Cards
RDNA architecture
E.g.:

- RX 6000 Series
- RX 7000 Series

AMD Instinct™ Accelerators
CDNA architecture
E.g.:

- MI100
- MI200
- MI300

AMD IN HPC

Frontier@ORNL

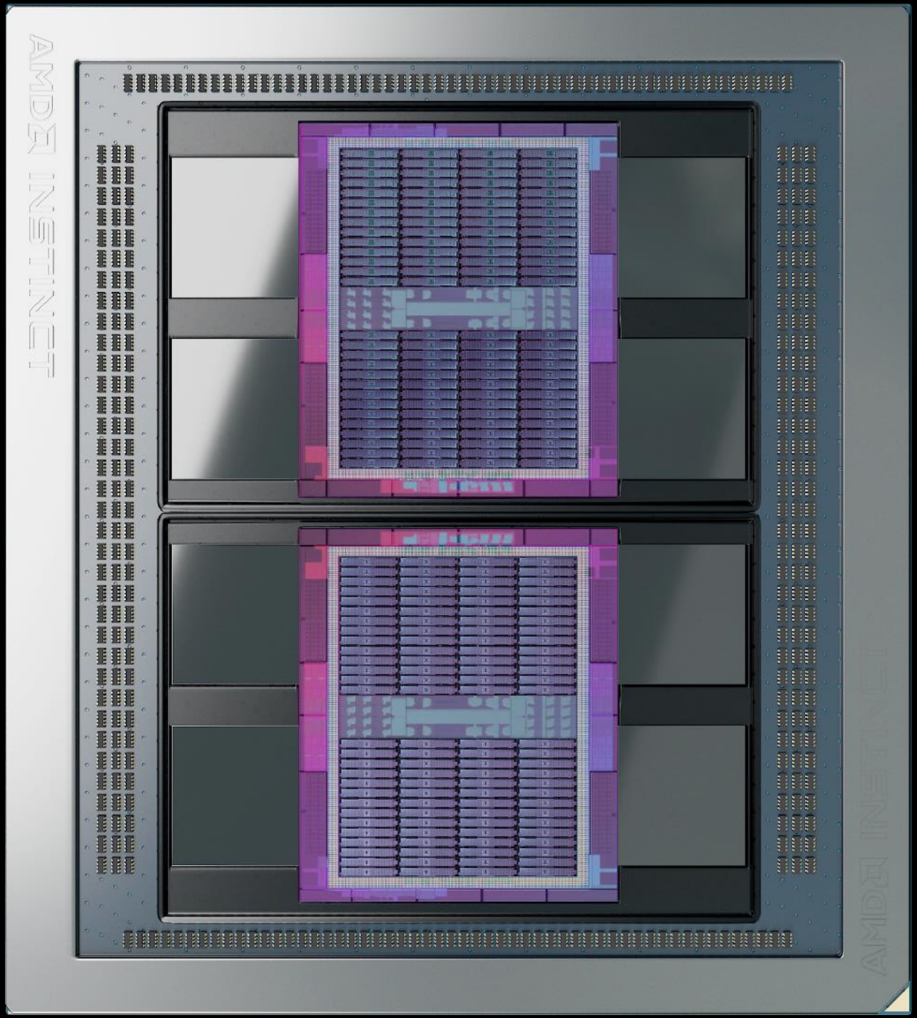
- currently the largest machine in the world
- the first computer to cross 1 exaFLOPS
- AMD EPYC CPUs
- AMD Instinct GPUs

LUMI@CSC

- currently the largest machine in Europe
- 5th fastest in the world
- AMD EPYC CPUs
- AMD Instinct GPUs

AMD
together we advance_

AMD INSTINCT™ MI200



AMD INSTINCT™ MI250X

ONE OF THE WORLD'S MOST ADVANCED DATA CENTER ACCELERATOR

58B
Transistors in 6nm

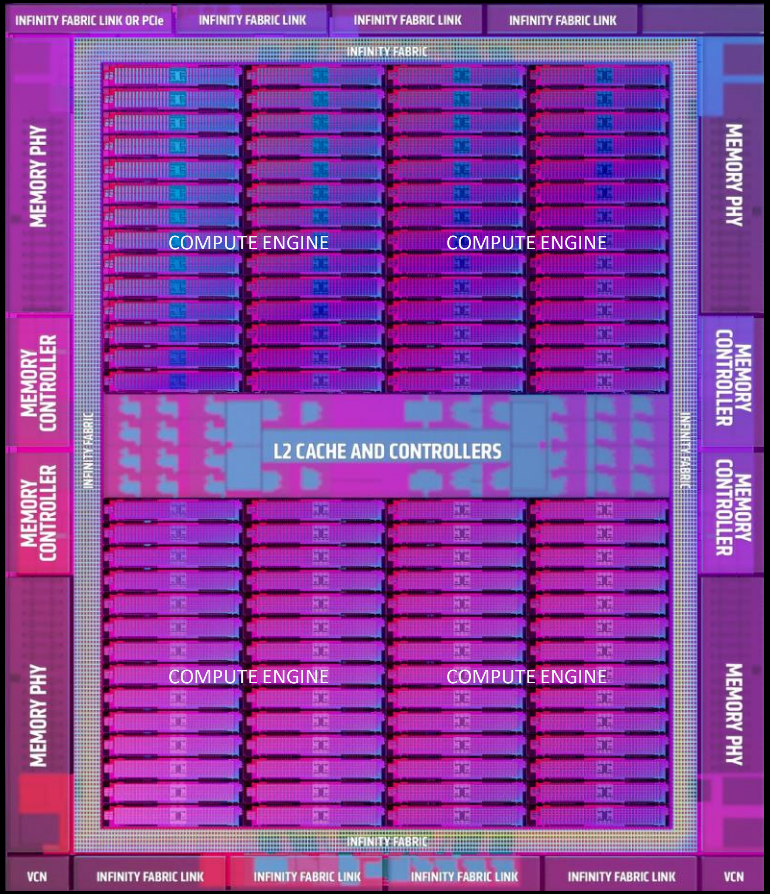
220
Compute Units

880
2nd Gen Matrix Cores

128
GB HBM2E @ 3.2 TB/s

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

AMD INSTINCT™ MI200



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_

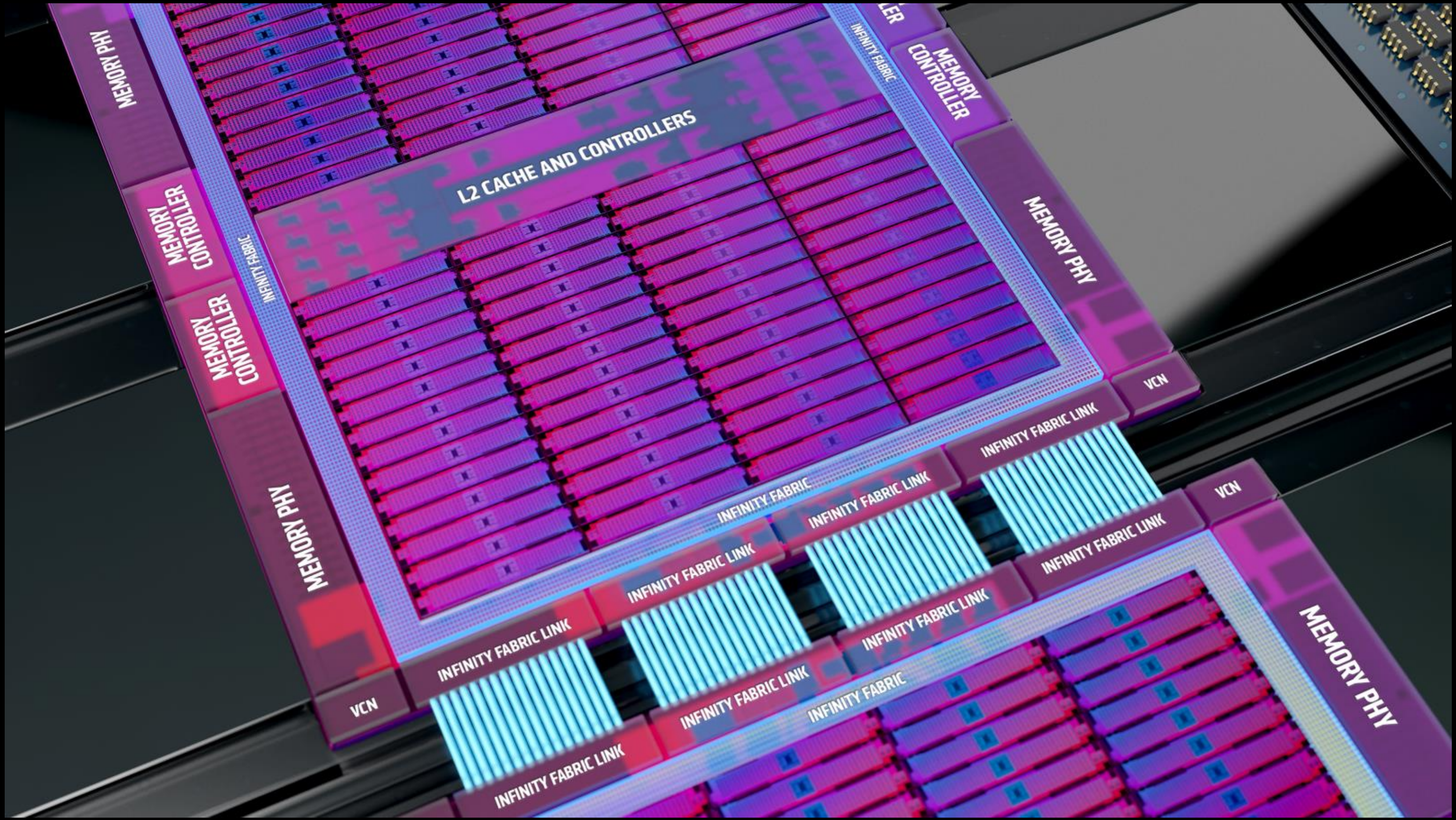
MULTI-CHIP DESIGN

TWO GPU DIES IN PACKAGE TO MAXIMIZE COMPUTE & DATA THROUGHPUT

INFINITY FABRIC FOR CROSS-DIE CONNECTIVITY

4 LINKS RUNNING AT 25GBPS

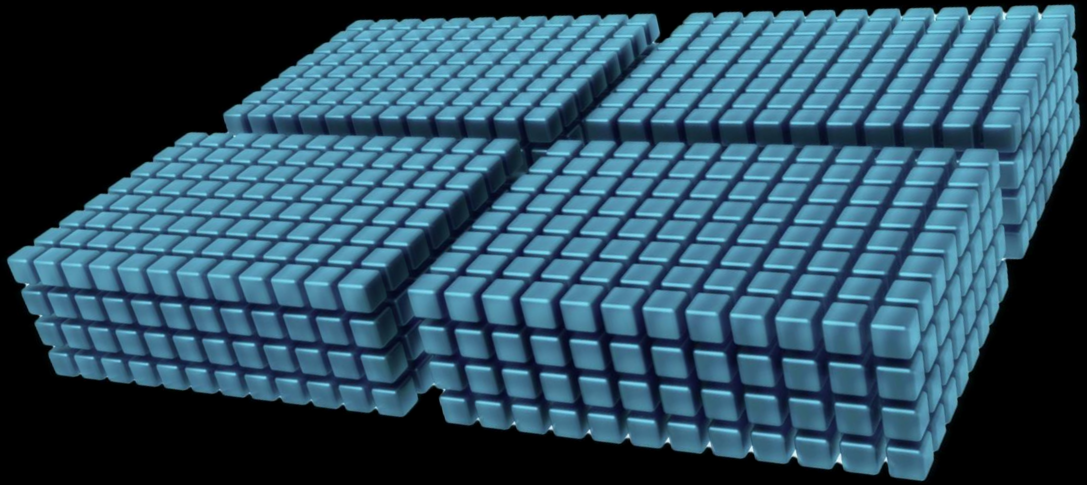
400GB/S OF BI-DIRECTIONALBANDWIDTH



AMD
together we advance_

2nd GENERATION MATRIX CORES

OPTIMIZED COMPUTE UNITS FOR SCIENTIFIC COMPUTING

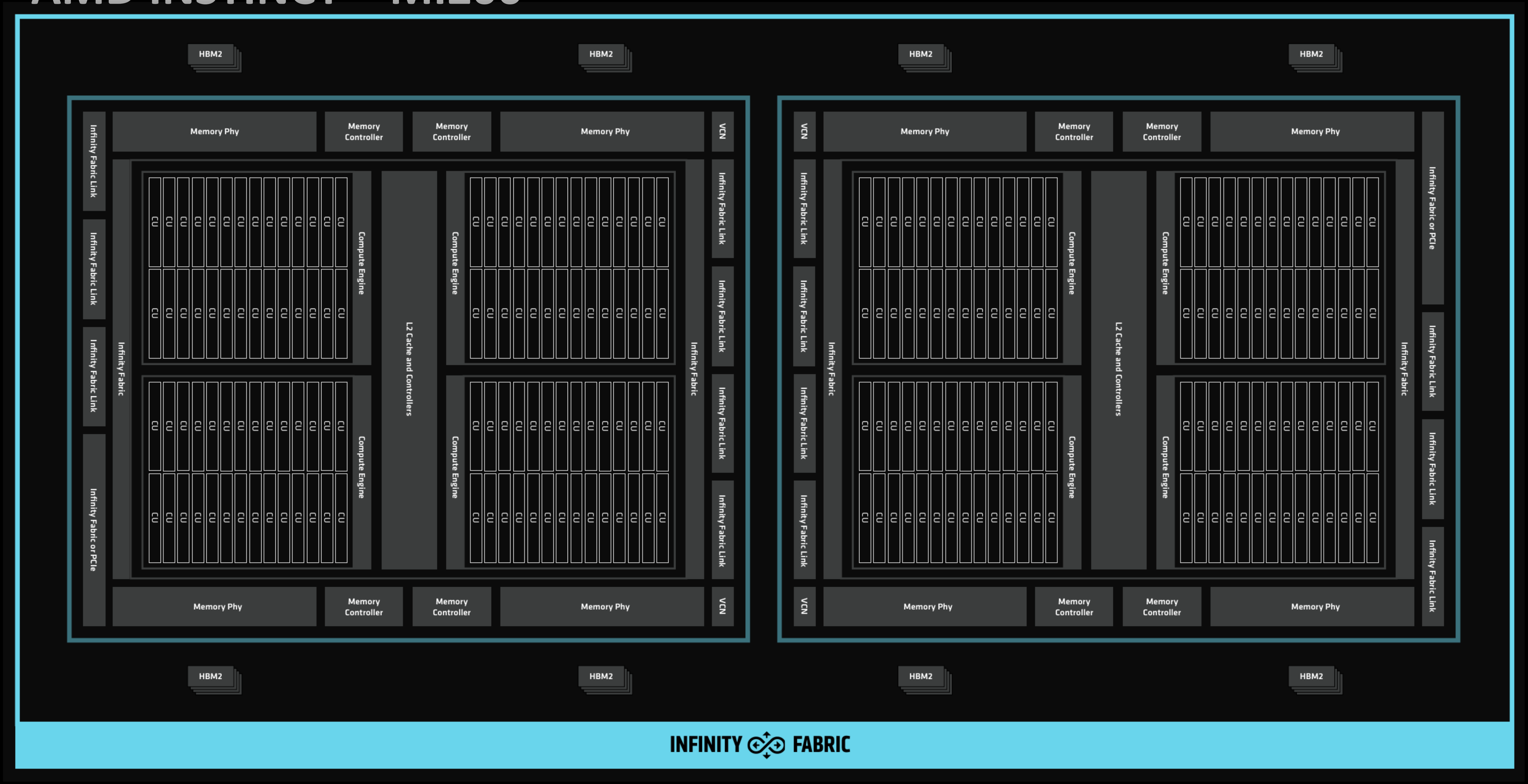


DOUBLE PRECISON (FP64)
MATRIX CORE THROUGHPUT
REPRESENTATION

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

AMD
together we advance_

AMD INSTINCT™ MI200



AMD
together we advance_

AMD MI250X specifications

- Two graphic compute dies (GCDs)
- 64GB of HBM2e memory per GCD (total 128GB)
- 26.5 TFLOPS peak performance per GCD
- 1.6 TB/s peak memory bandwidth per GCD
- 110 CU per GCD, total 220 CU per GPU
- The interconnection is attached to the GPU (not on the CPU)
- Both GCDs are interconnected with 200 GB/s per direction
- 128 single precision FMA operations per cycle
- AMD CDNA 2 Matrix Core supports double-precision data
- Memory coherency

AMD CDNA™ 2 white paper: https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf

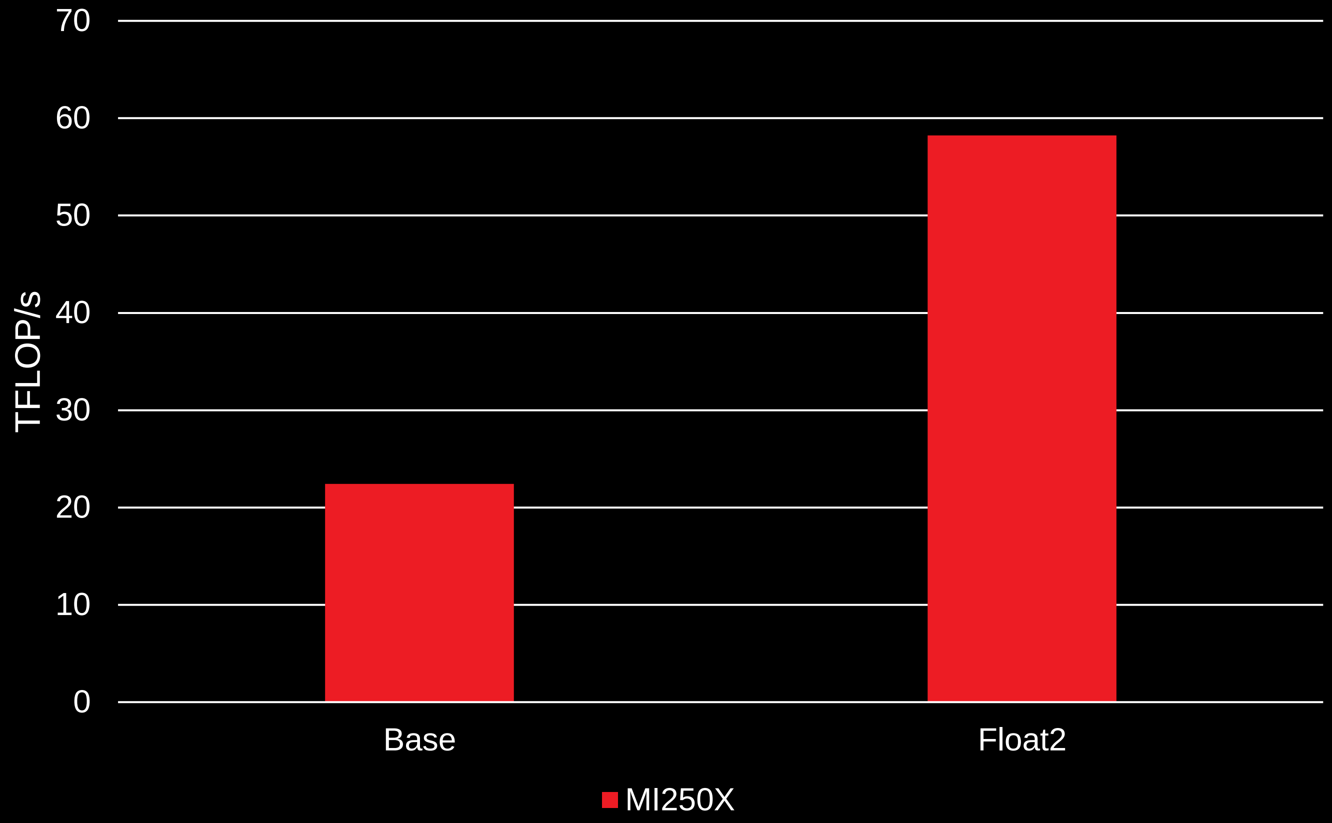
NEW IN AMD INSTINCT MI250X

PACKED FP32

FP64 PATH USED TO EXECUTE TWO COMPONENT VECTOR INSTRUCTIONS ON FP32

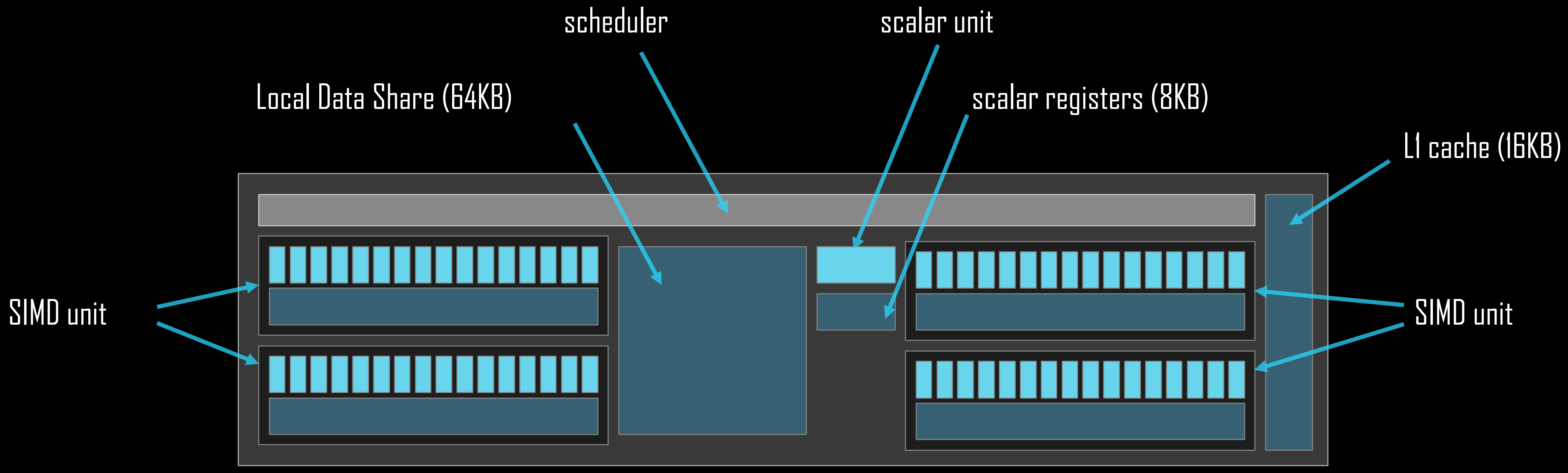
DOUBLES FP32 THROUGHPUT PER CLOCK PER COMPUTE UNIT

pk_FMA, pk_ADD, pk_MUL, pk_MOV operations



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

MI200 COMPUTE UNIT



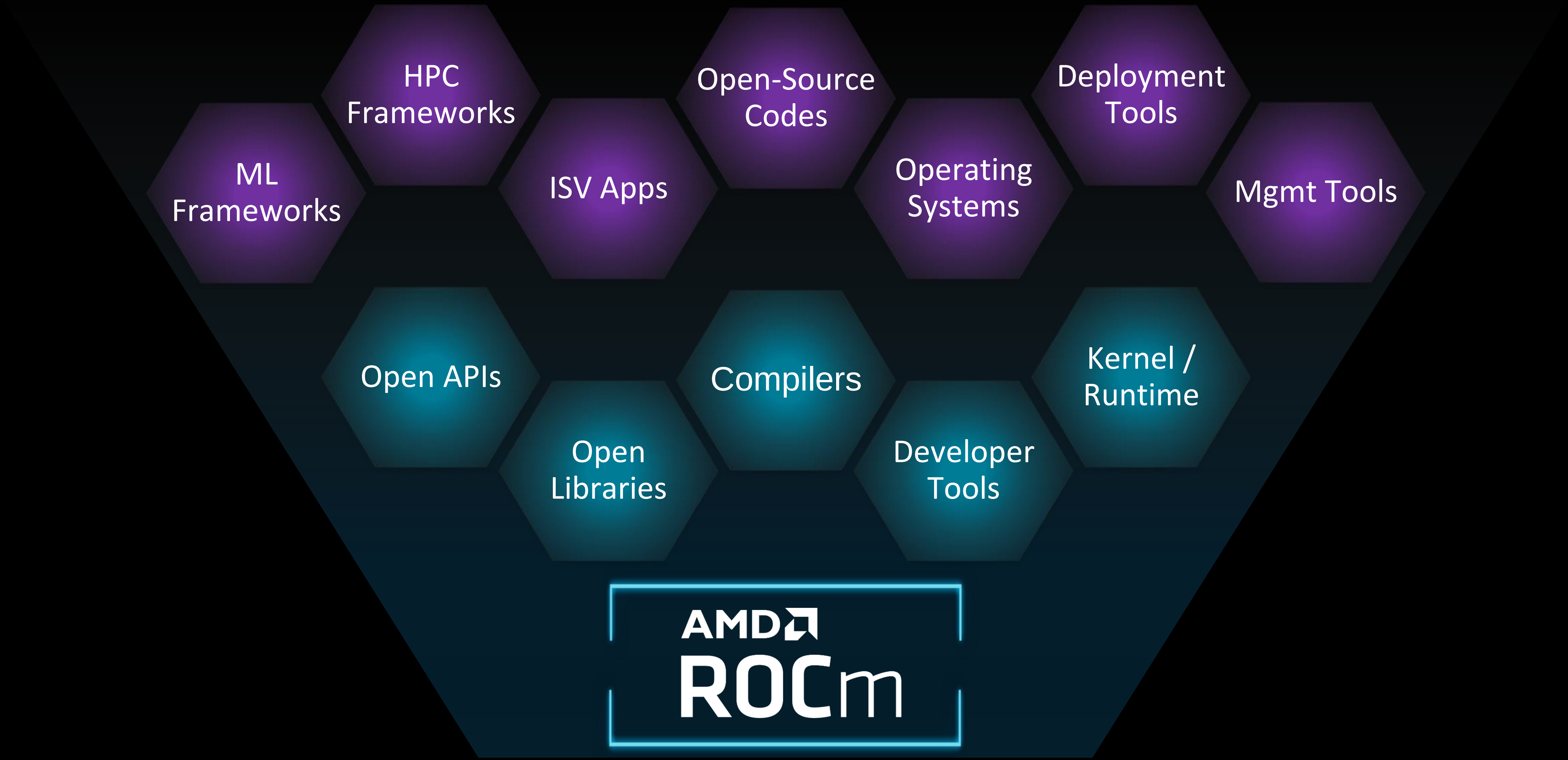
each SIMD unit

- has 16 SIMD lanes
- operates on vectors (waves) of size 64
- handles up to 10 waves simultaneously

AMD SOFTWARE FOR HPC AND AI

ROCM PLATFORM

AMD ROCm™ Open Software Platform For GPU Compute



Open Software Platform For GPU Compute

AMD ROCm

- Unlocked GPU Power To Accelerate Computational Tasks
- Optimized for HPC and Deep Learning Workloads at Scale
- Open Source Enabling Innovation, Differentiation, and Collaboration

Benchmarks & App Support	Optimized Training/Inference Models & Applications: MLPERF, HPL/HPCG, Life Science, Geo Science, Physics
Operating Systems Support	RHEL, CentOS, SLES, Ubuntu
Cluster Deployment	Singularity, Kubernetes®, Docker®, SLURM
Framework Support	Kokkos/RAJA, PyTorch, TensorFlow
Libraries	BLAS, RAND, FFT, MIGraphX, MIVisionX, PRIM, SOLVER, ALUTION, SPARSE, THRUST, MIOpen, RCCL
Programming Models	OpenMP® API, OpenCL™, HIP API
Development Toolchain	Compiler, Profiler, Tracer, Debugger, hipify, GPUFort
Drivers & Runtime	GPU Device Drivers and ROCm Run-Time
Deployment Tools	ROCm Validation Suite, ROCm Data Center Tool, ROCm SMI

AMD

ROCm 5.0

DEMOCRATIZING EXASCALE FOR ALL

EXPANDING SUPPORT & ACCESS

- Support for Radeon Pro W6800 Workstation GPUs
- Remote access through the AMD Accelerator Cloud

OPTIMIZING PERFORMANCE

- MI200 Optimizations: FP64 Matrix ops, Improved Cache
- Improved launch latency and kernel performance

ENABLING DEVELOPER SUCCESS

- HPC Apps & ML Frameworks on AMD InfinityHub
- Streamlined and improved tools increasing productivity

LIBRARIES

rocBLAS / hipBLAS
- basic operations on dense matrices

https://github.com/ROCmSoftwarePlatform/rocBLAS
https://github.com/ROCmSoftwarePlatform/hipBLAS

rocSOLVER
- dense linear algebra solvers

https://github.com/ROCmSoftwarePlatform/rocSOLVER

rocSPARSE / hipSPARSE
- basic operations on sparse matrices

https://github.com/ROCmSoftwarePlatform/rocSPARSE
https://github.com/ROCmSoftwarePlatform/hipSPARSE

rocALUTION
- sparse linear algebra solvers

https://github.com/ROCmSoftwarePlatform/rocALUTION

rocFFT / hipFFT
- Fast Fourier transforms

https://github.com/ROCmSoftwarePlatform/rocFFT
https://github.com/ROCmSoftwarePlatform/hipFFT

rocRAND / hipRAND
- random number generation

https://github.com/ROCmSoftwarePlatform/rocRAND
https://github.com/ROCmSoftwarePlatform/hipRAND

rocPRIM / hipCUB / rocThrust
- scan, sort, reduction, etc.

https://github.com/ROCmSoftwarePlatform/rocPRIM
https://github.com/ROCmSoftwarePlatform/hipCUB
https://github.com/ROCmSoftwarePlatform/rocThrust

ALSO OPEN SOURCE

the compiler
- https://github.com/ROCmSoftwarePlatform/llvm-project

the runtime
- https://github.com/RadeonOpenCompute/ROCR-Runtime

the debugger
- https://github.com/ROCm-Developer-Tools/ROCgdb

the profiler
- https://github.com/ROCm-Developer-Tools/rocprofiler

the HPL benchmark
- https://github.com/ROCmSoftwarePlatform/rocHPL

the HPCG benchmark
- https://github.com/ROCmSoftwarePlatform/rocHPCG

etc.

AMD SOFTWARE FOR HPC AND AI

HIP PROGRAMMING

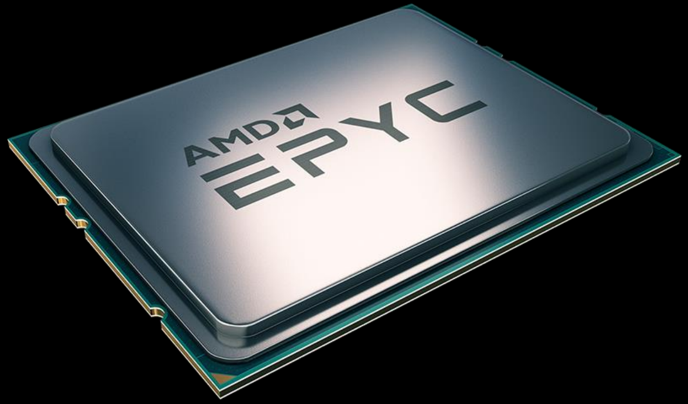
GPU ACCELERATION
HOST AND DEVICE

the host is the CPU

- host code runs here
- usual C++ syntax and features
- entry point is the “main” function
- use the HIP API to
  - create device buffers
  - moved data between host and device
  - launch device code

the device is the GPU

- device code runs here
- C/C++ syntax and features
- device code is launched as “kernels”
- instructions from the host are sent to streams



FUNCTION QUALIFIERS
HOST AND DEVICE

__global__

- “kernels”
- execute the GPU
- can be called from the CPU

__device__

- execute the GPU
- can be called from device code (kernels or a __device__ functions)

__host__ __device__

- executes on the CPU when called from CPU code
- executes on the GPU when called from GPU code

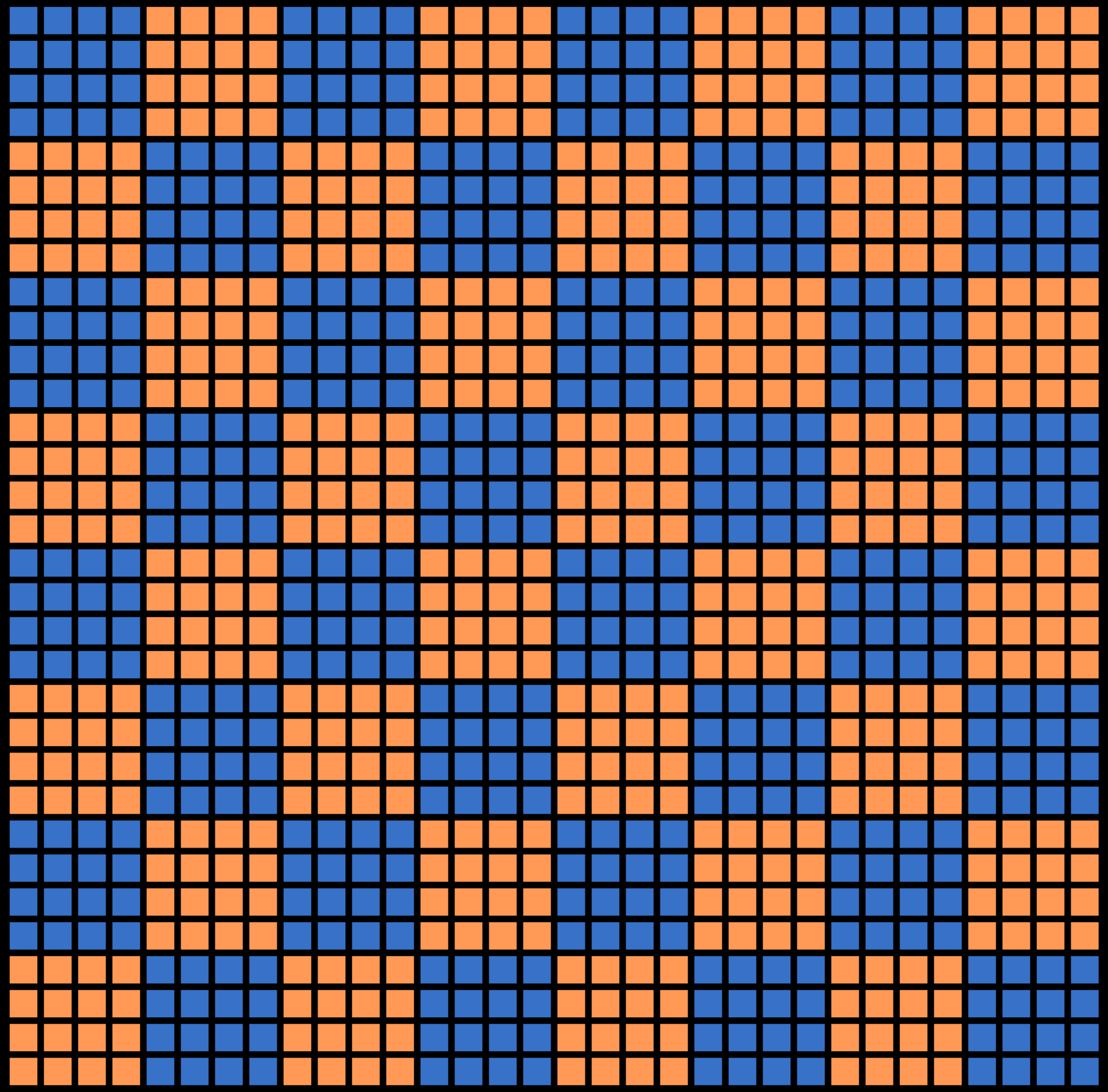
HIP KERNEL LANGUAGE

GPU CODE

in 2D

- each colored box is a block
- each block has an index - blockIdx.[xyz]
- each small square is a thread
- each thread has a 2D index - threadIdx.[xyz]
- grid dimensions in - blockDim.[xyz]

in 2D



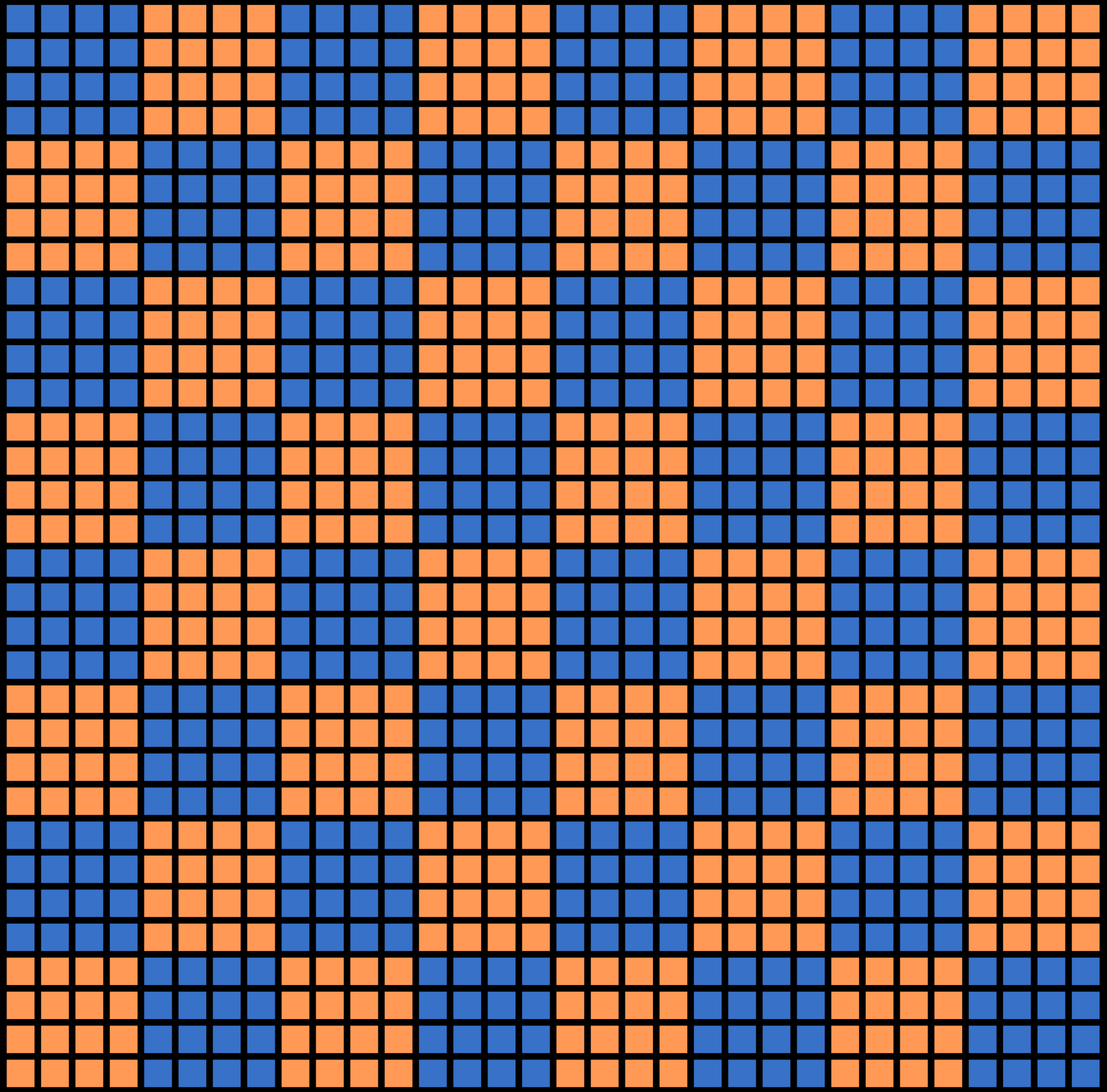
https://rocm.docs.amd.com/projects/HIP/en/latest/reference/kernel_language.html

HIP KERNEL LANGUAGE

GPU CODE

- all local variables and arrays are thread-private
- threads can exchange data through shared memory (LDS)
- declare using the __shared__ keyword
- use __syncthreads() to synchronize

in 2D



https://rocm.docs.amd.com/projects/HIP/en/latest/reference/kernel_language.html

HIP KERNEL LANGUAGE

GPU CODE

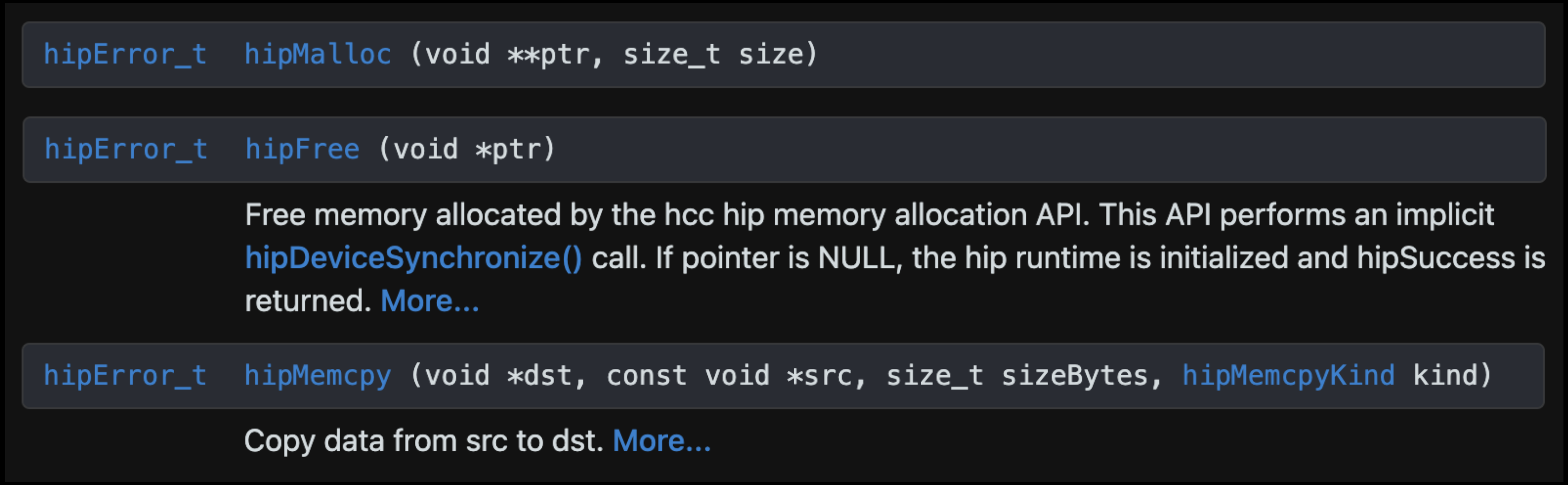
saxpy loop

- two 1D arrays
- the y[i] += a*x[i] operation
- mapped to 1D grid of threads/blocks
- each thread takes on index

```
#include <cuda.h>

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}
```

https://rocm.docs.amd.com/projects/HIP/en/latest/reference/kernel_language.html

HIP API
MEMORY MANAGEMENT



```
hipError_t  hipMalloc (void **ptr, size_t size)
```

```
hipError_t  hipFree (void *ptr)
```

Free memory allocated by the hcc hip memory allocation API. This API performs an implicit hipDeviceSynchronize() call. If pointer is NULL, the hip runtime is initialized and hipSuccess is returned. More...

```
hipError_t  hipMemcpy (void *dst, const void *src, size_t sizeBytes, hipMemcpyKind kind)
```

Copy data from src to dst. More...

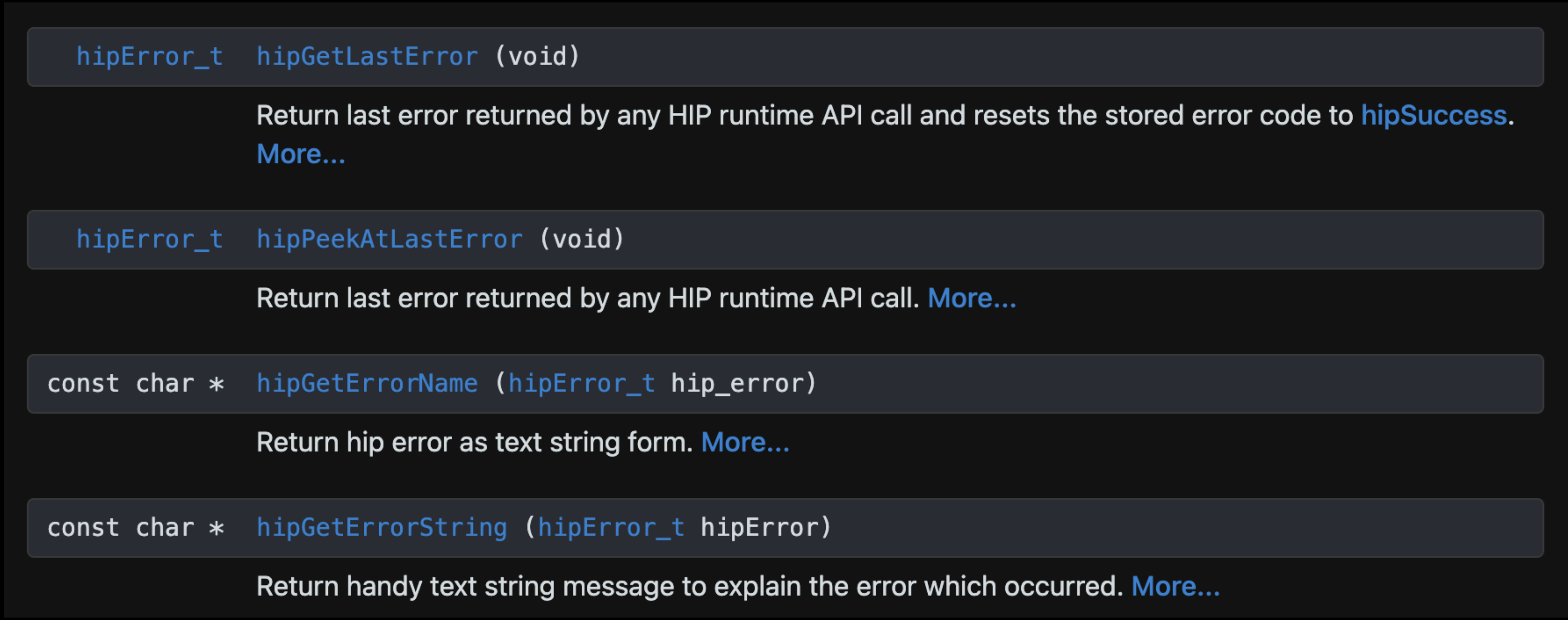
- GPU operates on GPU memory
- need to allocate GPU memory
- need to copy data between the CPU memory and the GPU memory

https://rocm.docs.amd.com/projects/HIP/en/latest/.doxygen/docBin/html/group___memory.html

AMD together we advance_

HIP API
ERROR HANDLING

- check last error
- get error name
- get error string



```
hipError_t   hipGetLastError (void)
```

Return last error returned by any HIP runtime API call and resets the stored error code to hipSuccess. More...

```
hipError_t   hipPeekAtLastError (void)
```

Return last error returned by any HIP runtime API call. More...

```
const char *   hipGetErrorName (hipError_t hip_error)
```

Return hip error as text string form. More...

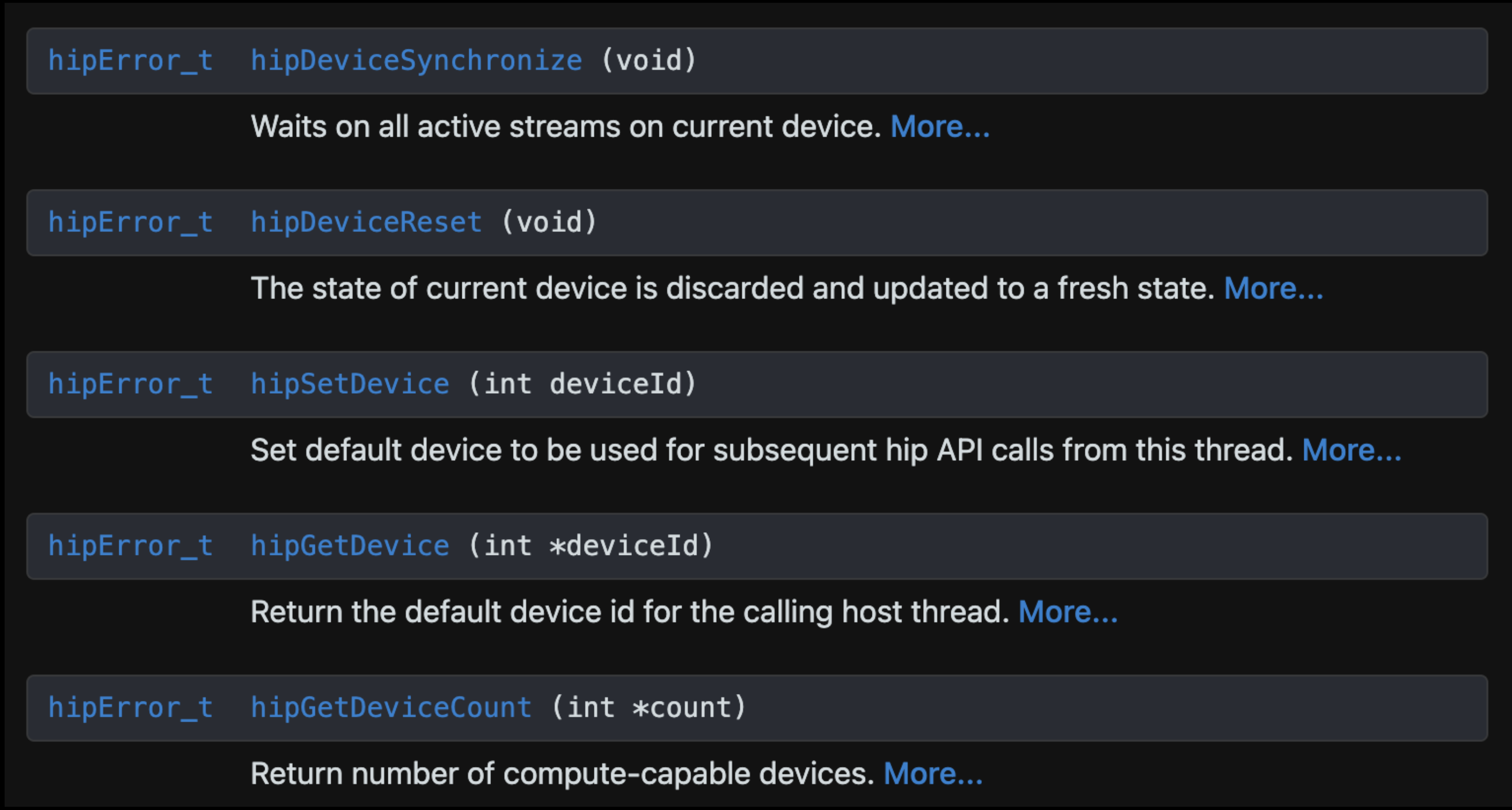
```
const char *   hipGetErrorString (hipError_t hipError)
```

Return handy text string message to explain the error which occurred. More...

https://rocm.docs.amd.com/projects/HIP/en/latest/.doxygen/docBin/html/group___error.html

HIP API
DEVICE MANAGEMENT

- check number of devices
- switch devices
- synchronize devices



```
hipError_t  hipDeviceSynchronize (void)
```
Waits on all active streams on current device. More...

```
hipError_t  hipDeviceReset (void)
```
The state of current device is discarded and updated to a fresh state. More...

```
hipError_t  hipSetDevice (int deviceId)
```
Set default device to be used for subsequent hip API calls from this thread. More...

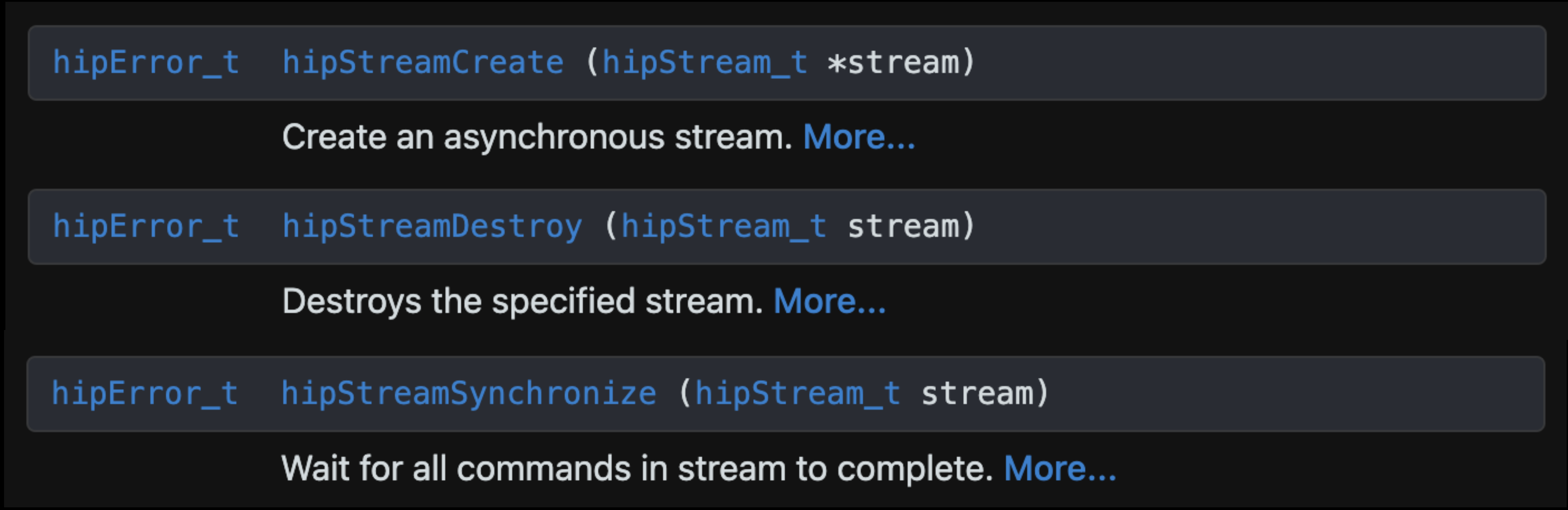
```
hipError_t  hipGetDevice (int *deviceId)
```
Return the default device id for the calling host thread. More...

```
hipError_t  hipGetDeviceCount (int *count)
```
Return number of compute-capable devices. More...

https://rocm.docs.amd.com/projects/HIP/en/latest/.doxygen/docBin/html/group___device.html

HIP API
STREAM MANAGEMENT

- create stream
- destroy stream
- synchronize stream
- etc.
- etc.
- etc.



```
hipError_t   hipStreamCreate (hipStream_t *stream)
```

Create an asynchronous stream. More...

```
hipError_t   hipStreamDestroy (hipStream_t stream)
```

Destroys the specified stream. More...

```
hipError_t   hipStreamSynchronize (hipStream_t stream)
```

Wait for all commands in stream to complete. More...

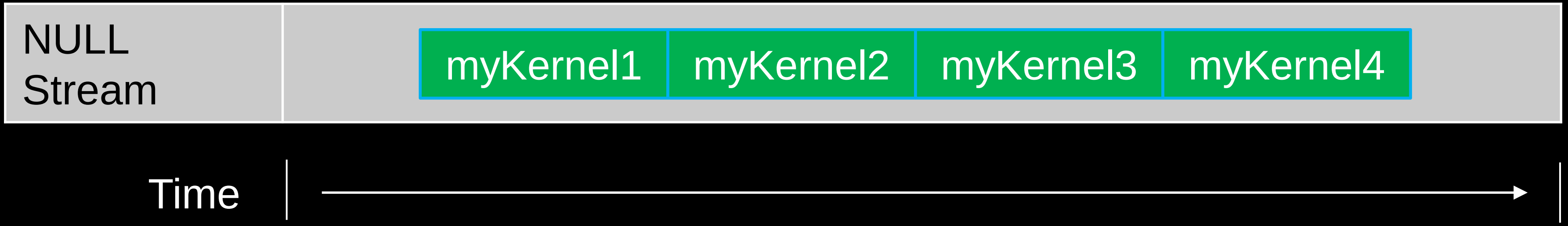
https://rocm.docs.amd.com/projects/HIP/en/latest/.doxygen/docBin/html/group___stream.html

STREAMS

- Suppose we have 4 small kernels to execute:

```
hipLaunchKernelGGL(myKernel1, dim3(1), dim3(256), 0, 0, 256, d_a1);
hipLaunchKernelGGL(myKernel2, dim3(1), dim3(256), 0, 0, 256, d_a2);
hipLaunchKernelGGL(myKernel3, dim3(1), dim3(256), 0, 0, 256, d_a3);
hipLaunchKernelGGL(myKernel4, dim3(1), dim3(256), 0, 0, 256, d_a4);
```

- Even though these kernels use only one block each, they'll execute in serial on the NULL stream:



AMD
together we advance_

STREAMS

- With streams we can effectively share the GPU’s compute resources:

```
hipLaunchKernelGGL(myKernel1, dim3(1), dim3(256), 0, stream1, 256, d_a1);
hipLaunchKernelGGL(myKernel2, dim3(1), dim3(256), 0, stream2, 256, d_a2);
hipLaunchKernelGGL(myKernel3, dim3(1), dim3(256), 0, stream3, 256, d_a3);
hipLaunchKernelGGL(myKernel4, dim3(1), dim3(256), 0, stream4, 256, d_a4);
```

NULL Stream			
Stream1		myKernel1	
Stream2		myKernel2	
Stream3		myKernel3	
Stream4		myKernel4	

Note 1: Kernels must modify different parts of memory to avoid data races.

Note 2: With large kernels, overlapping computations may not help performance.

STREAMS

- There is another use for streams besides concurrent kernels:
  - Overlapping kernels with data movement.

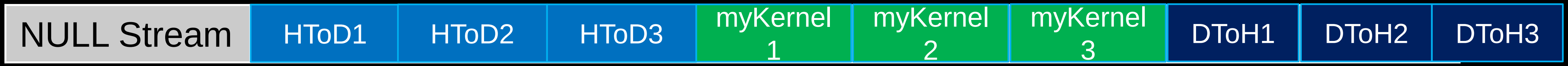
- AMD GPUs have separate engines for:
  - Host->Device memcpys
  - Device->Host memcpys
  - Compute kernels.

- These three different operations can overlap without dividing the GPU’s resources.
  - The overlapping operations should be in separate, non-NULL, streams.
  - The host memory should be **pinned.**

STREAMS

Suppose we have 3 kernels which require moving data to and from the device:

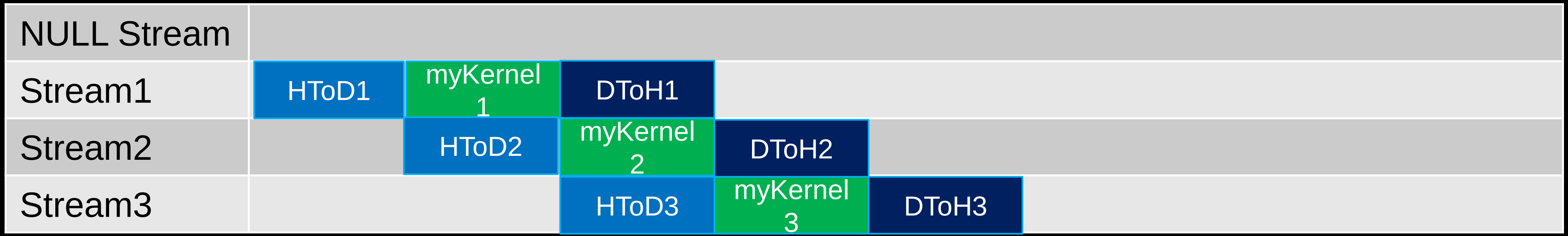
```
hipMemcpy(d_a1, h_a1, Nbytes, hipMemcpyHostToDevice));
hipMemcpy(d_a2, h_a2, Nbytes, hipMemcpyHostToDevice));
hipMemcpy(d_a3, h_a3, Nbytes, hipMemcpyHostToDevice));

hipLaunchKernelGGL(myKernel1, blocks, threads, 0, 0, N, d_a1);
hipLaunchKernelGGL(myKernel2, blocks, threads, 0, 0, N, d_a2);
hipLaunchKernelGGL(myKernel3, blocks, threads, 0, 0, N, d_a3);

hipMemcpy(h_a1, d_a1, Nbytes, hipMemcpyDeviceToHost);
hipMemcpy(h_a2, d_a2, Nbytes, hipMemcpyDeviceToHost);
hipMemcpy(h_a3, d_a3, Nbytes, hipMemcpyDeviceToHost);
```



STREAMS

Changing to asynchronous memcpys and using streams:

```
hipMemcpyAsync(d_a1, h_a1, Nbytes, hipMemcpyHostToDevice, stream1);
hipMemcpyAsync(d_a2, h_a2, Nbytes, hipMemcpyHostToDevice, stream2);
hipMemcpyAsync(d_a3, h_a3, Nbytes, hipMemcpyHostToDevice, stream3);

hipLaunchKernelGGL(myKernel1, blocks, threads, 0, stream1, N, d_a1);
hipLaunchKernelGGL(myKernel2, blocks, threads, 0, stream2, N, d_a2);
hipLaunchKernelGGL(myKernel3, blocks, threads, 0, stream3, N, d_a3);

hipMemcpyAsync(h_a1, d_a1, Nbytes, hipMemcpyDeviceToHost, stream1);
hipMemcpyAsync(h_a2, d_a2, Nbytes, hipMemcpyDeviceToHost, stream2);
hipMemcpyAsync(h_a3, d_a3, Nbytes, hipMemcpyDeviceToHost, stream3);
```



AMD LINGO

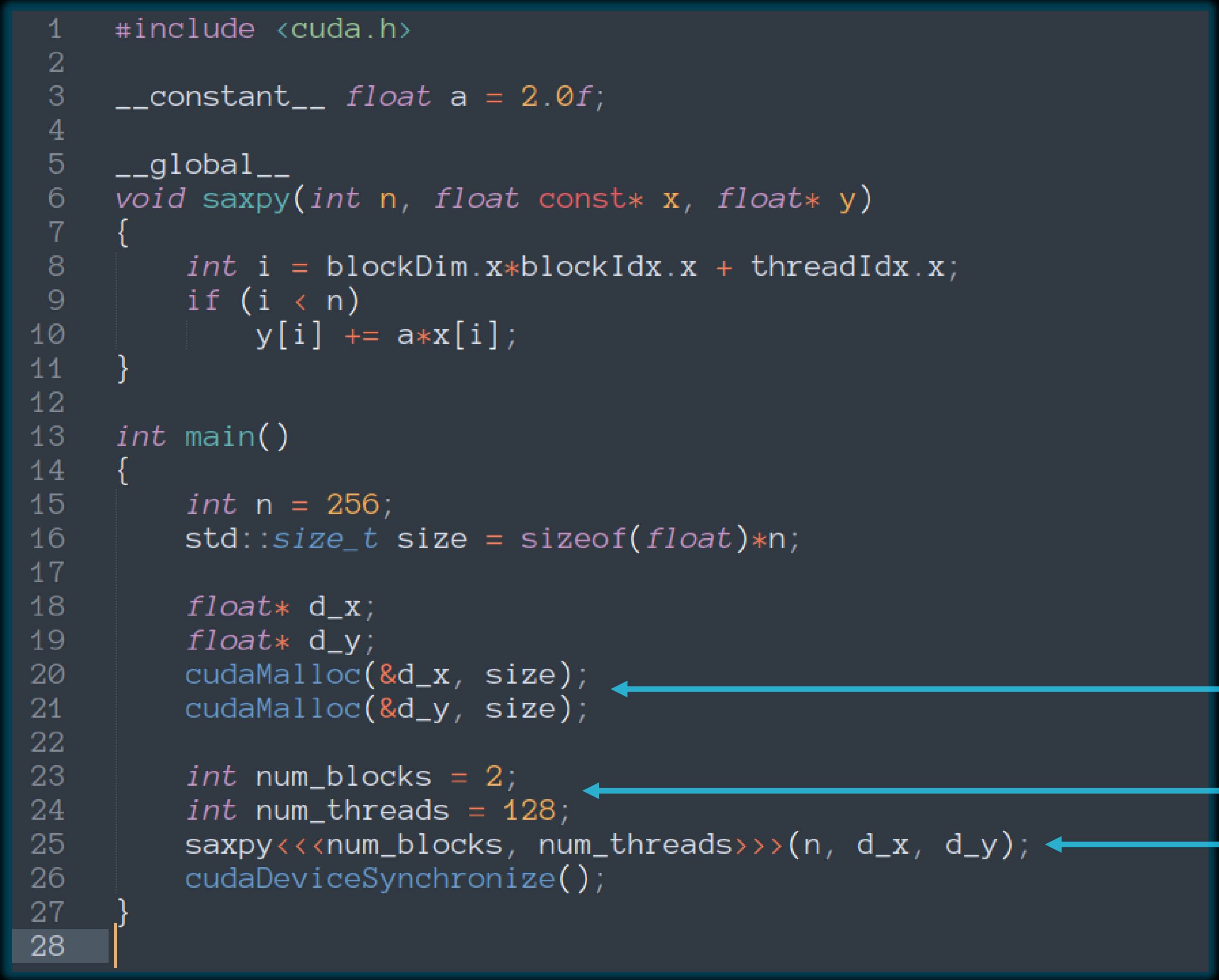
CUDA lingo		AMD lingo
block	→	work group
thread	→	work item
warp	→	wavefront

SIMPLE SAXPY KERNEL

```
#include <cuda.h>

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}
```

- vector addition kernel in CUDA
- each thread takes one array index
- and performs one multiply-and-add operation

ADDING THE CPU CODE



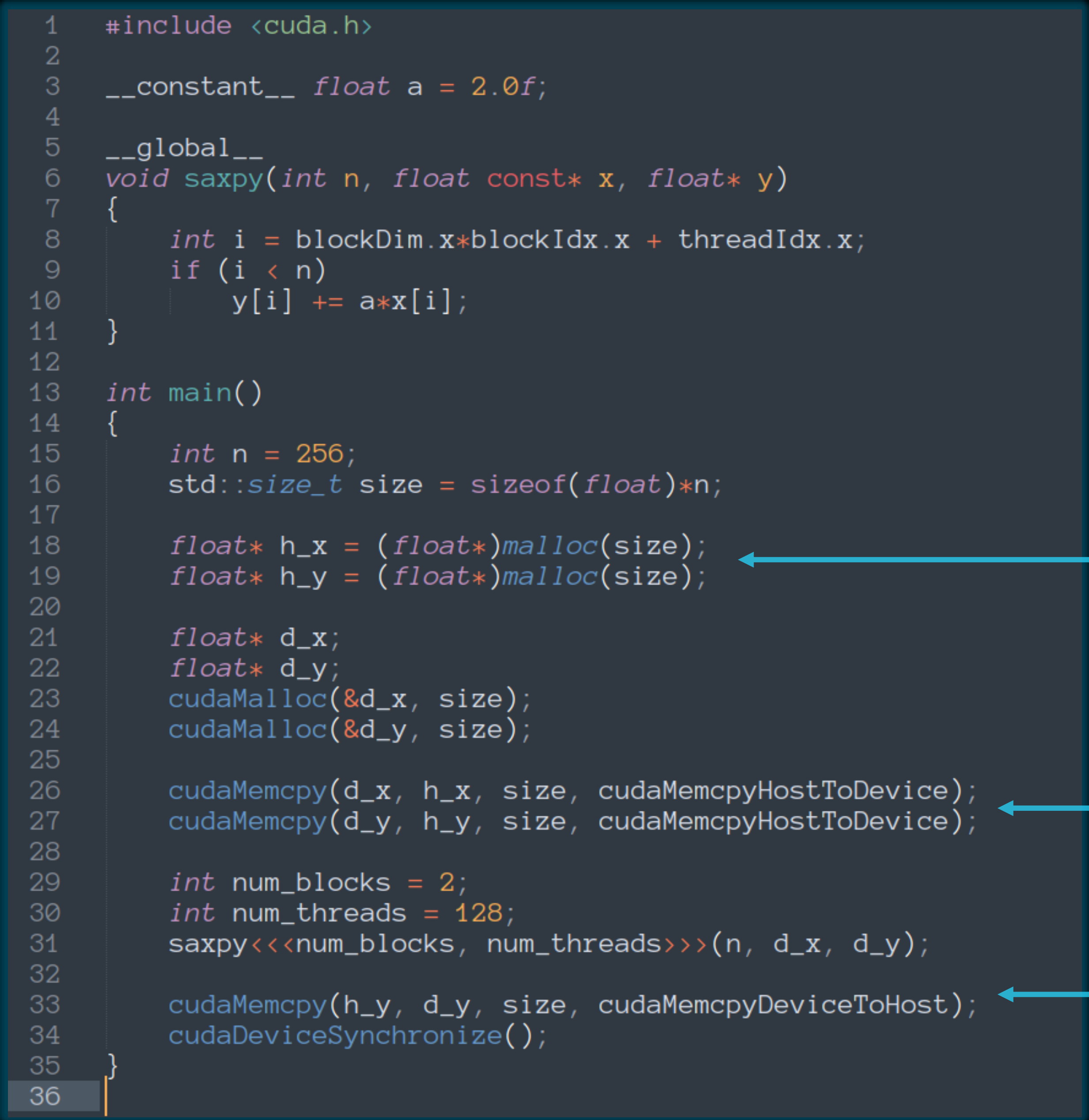
```
#include <cuda.h>

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}

int main()
{
    int n = 256;
    std::size_t size = sizeof(float)*n;

    float* d_x;
    float* d_y;
    cudaMalloc(&d_x, size);
    cudaMalloc(&d_y, size);

    int num_blocks = 2;
    int num_threads = 128;
    saxpy<<<num_blocks, num_threads>>>(n, d_x, d_y);
    cudaDeviceSynchronize();
}
```

allocate arrays in device memory

set up the grid

launch the kernel

ADDING HOST↔DEVICE COPIES



```
#include <cuda.h>

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}

int main()
{
    int n = 256;
    std::size_t size = sizeof(float)*n;

    float* h_x = (float*)malloc(size);
    float* h_y = (float*)malloc(size);

    float* d_x;
    float* d_y;
    cudaMalloc(&d_x, size);
    cudaMalloc(&d_y, size);

    cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, h_y, size, cudaMemcpyHostToDevice);

    int num_blocks = 2;
    int num_threads = 128;
    saxpy<<<num_blocks, num_threads>>>(n, d_x, d_y);

    cudaMemcpy(h_y, d_y, size, cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();
}
```

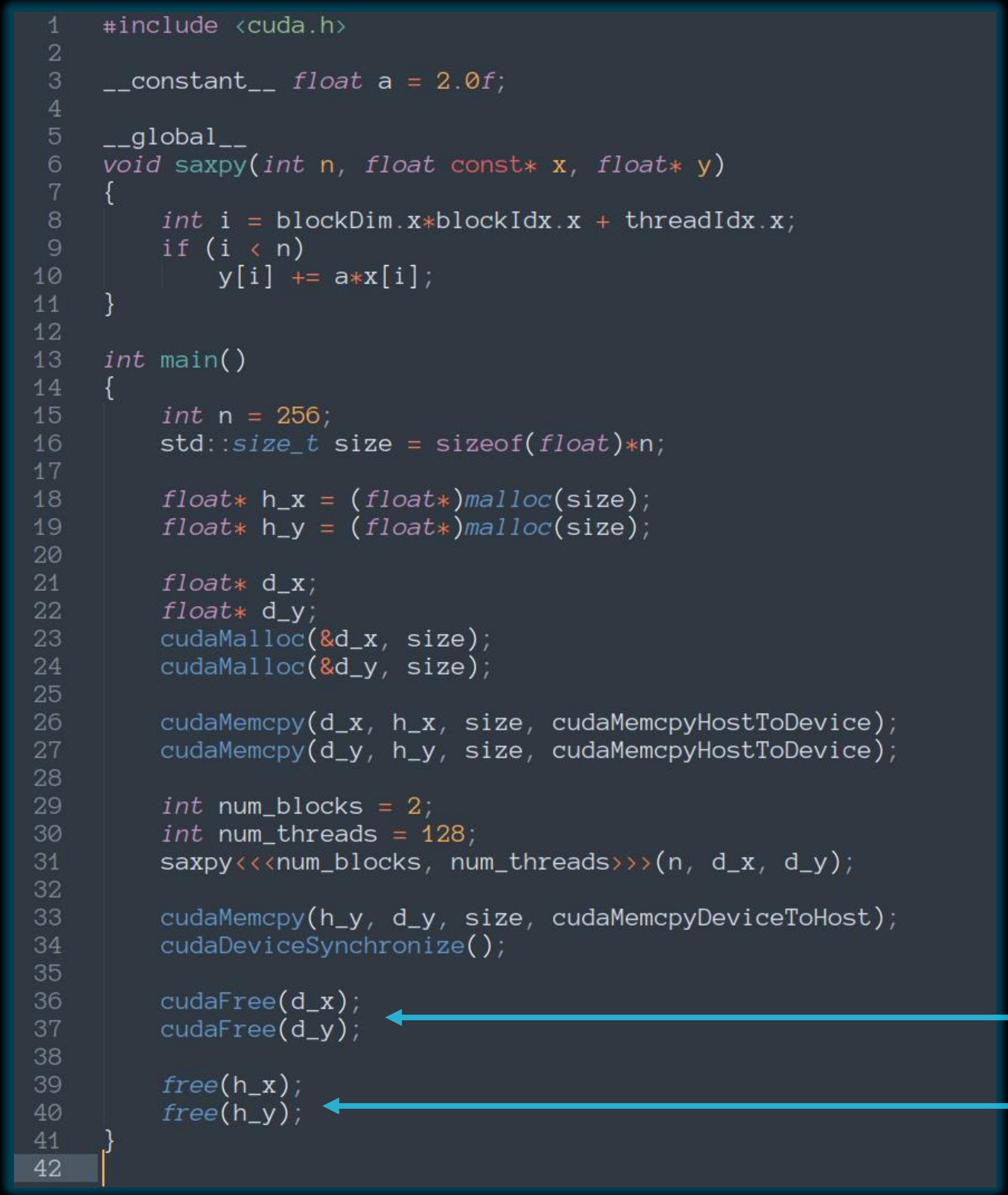
allocate arrays in host memory

copy content to device memory

copy results back to host memory

AMD
together we advance_

ADDING MEMORY CLEANUP

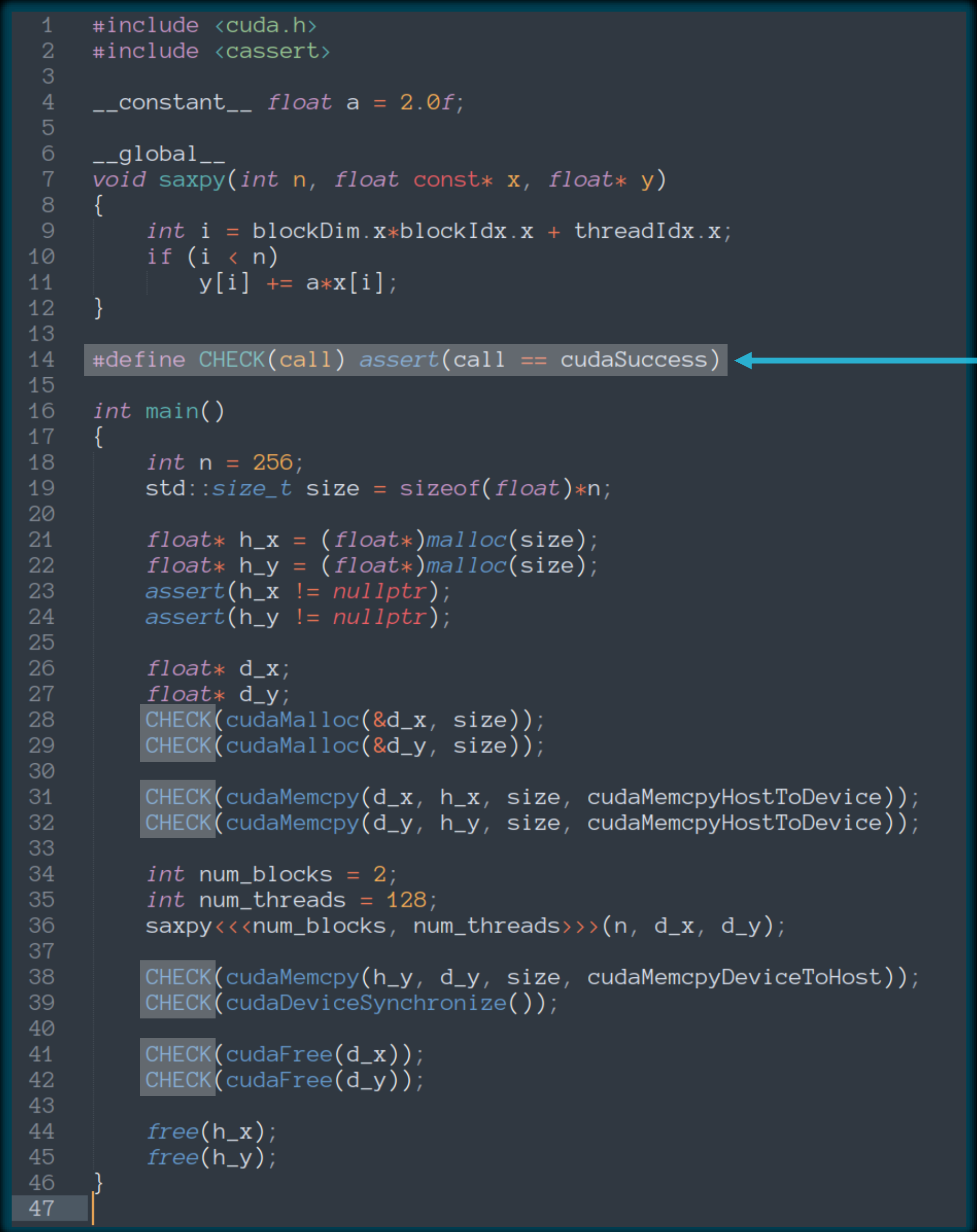


```
#include <cuda.h>

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}

int main()
{
    int n = 256;
    std::size_t size = sizeof(float)*n;

    float* h_x = (float*)malloc(size);
    float* h_y = (float*)malloc(size);

    float* d_x;
    float* d_y;
    cudaMalloc(&d_x, size);
    cudaMalloc(&d_y, size);

    cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, h_y, size, cudaMemcpyHostToDevice);

    int num_blocks = 2;
    int num_threads = 128;
    saxpy<<<num_blocks, num_threads>>>(n, d_x, d_y);

    cudaMemcpy(h_y, d_y, size, cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();

    cudaFree(d_x);
    cudaFree(d_y);

    free(h_x);
    free(h_y);
}
```

free arrays in device memory

free arrays in host memory

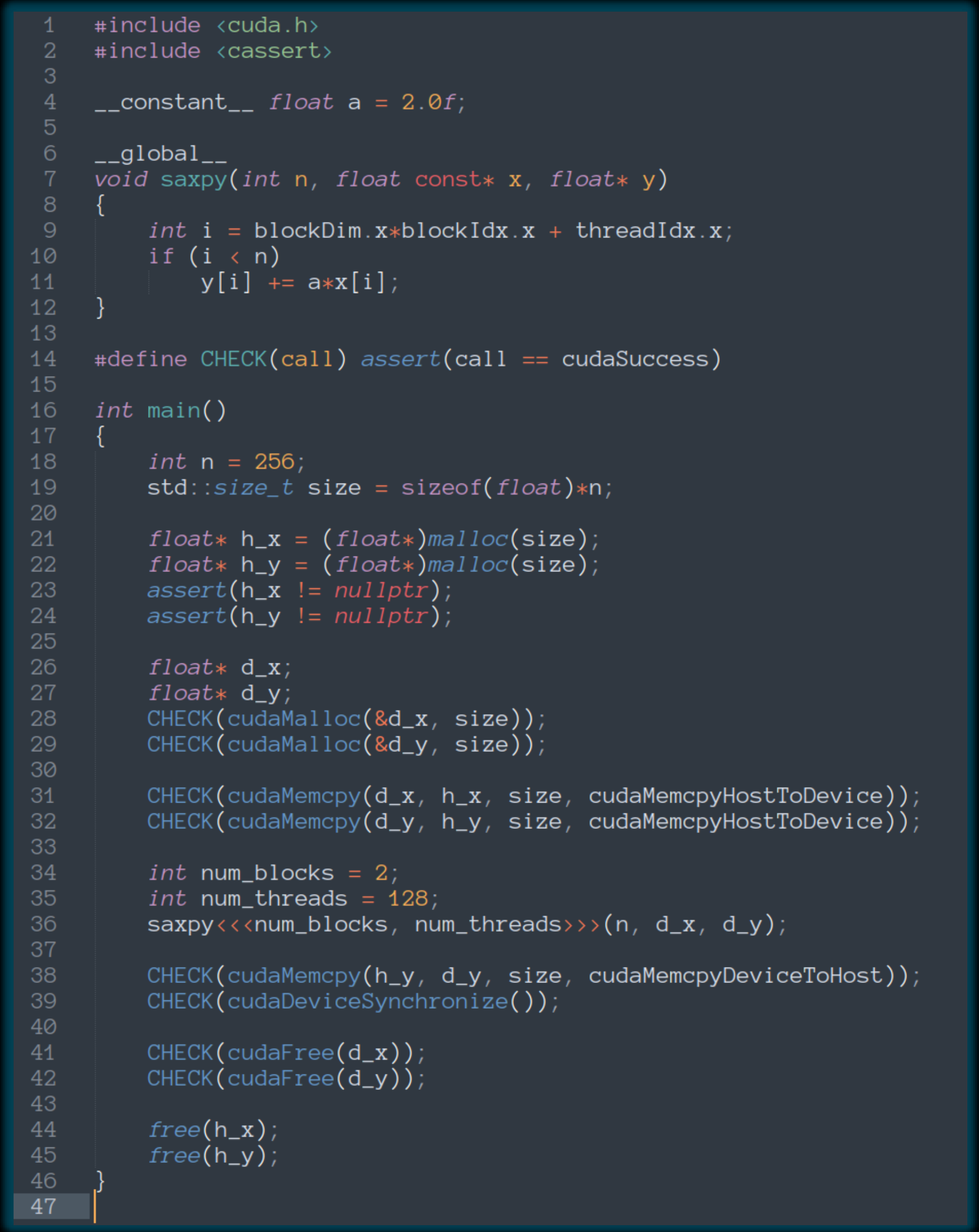
ADDING ERROR CHECKS



```
#include <cuda.h>
#include <cassert>

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}

#define CHECK(call) assert(call == cudaSuccess)

int main()
{
    int n = 256;
    std::size_t size = sizeof(float)*n;

    float* h_x = (float*)malloc(size);
    float* h_y = (float*)malloc(size);
    assert(h_x != nullptr);
    assert(h_y != nullptr);

    float* d_x;
    float* d_y;
    CHECK(cudaMalloc(&d_x, size));
    CHECK(cudaMalloc(&d_y, size));

    CHECK(cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_y, h_y, size, cudaMemcpyHostToDevice));

    int num_blocks = 2;
    int num_threads = 128;
    saxpy<<<num_blocks, num_threads>>>(n, d_x, d_y);

    CHECK(cudaMemcpy(h_y, d_y, size, cudaMemcpyDeviceToHost));
    CHECK(cudaDeviceSynchronize());

    CHECK(cudaFree(d_x));
    CHECK(cudaFree(d_y));

    free(h_x);
    free(h_y);
}
```

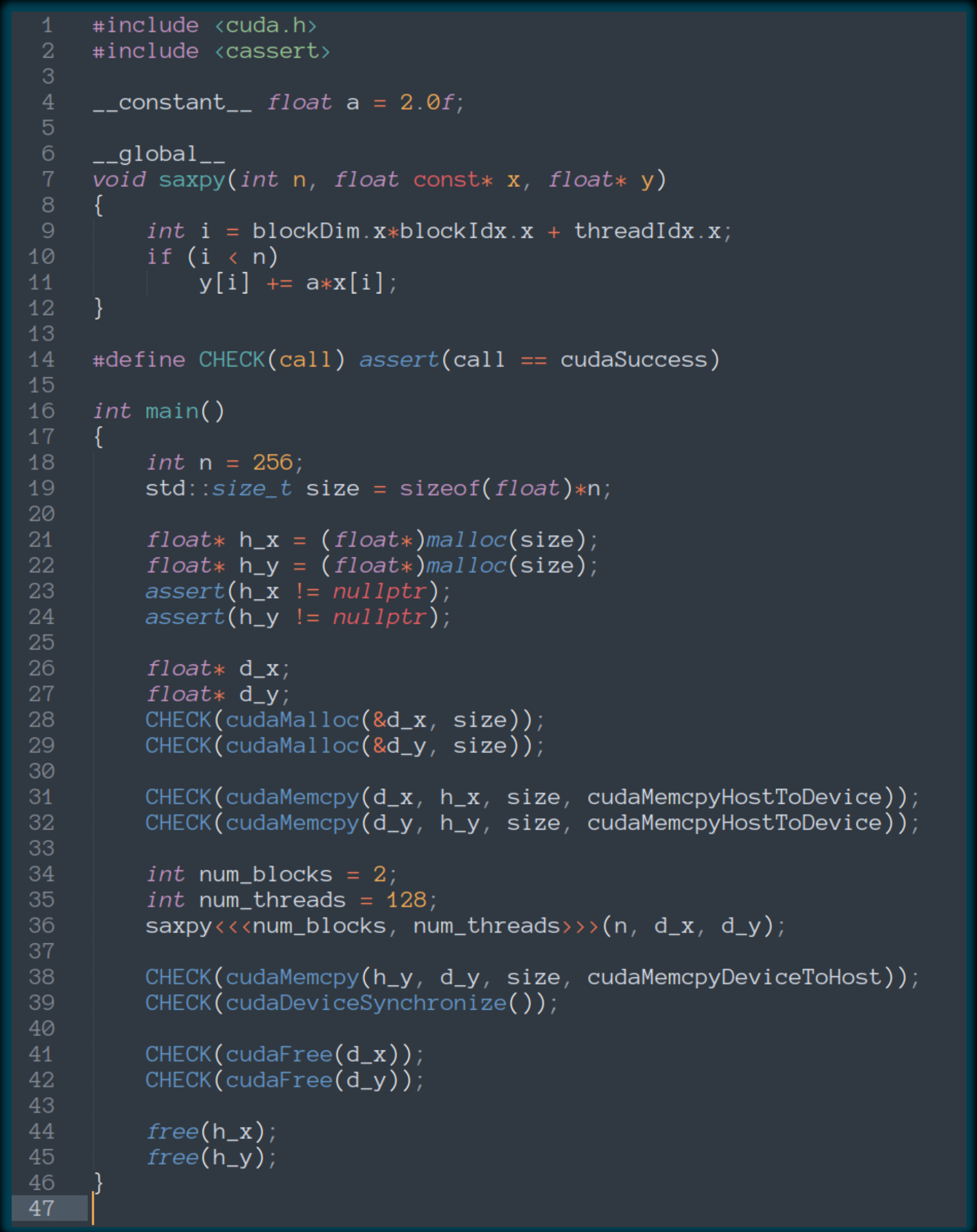
simple error checking macro

simple CUDA code



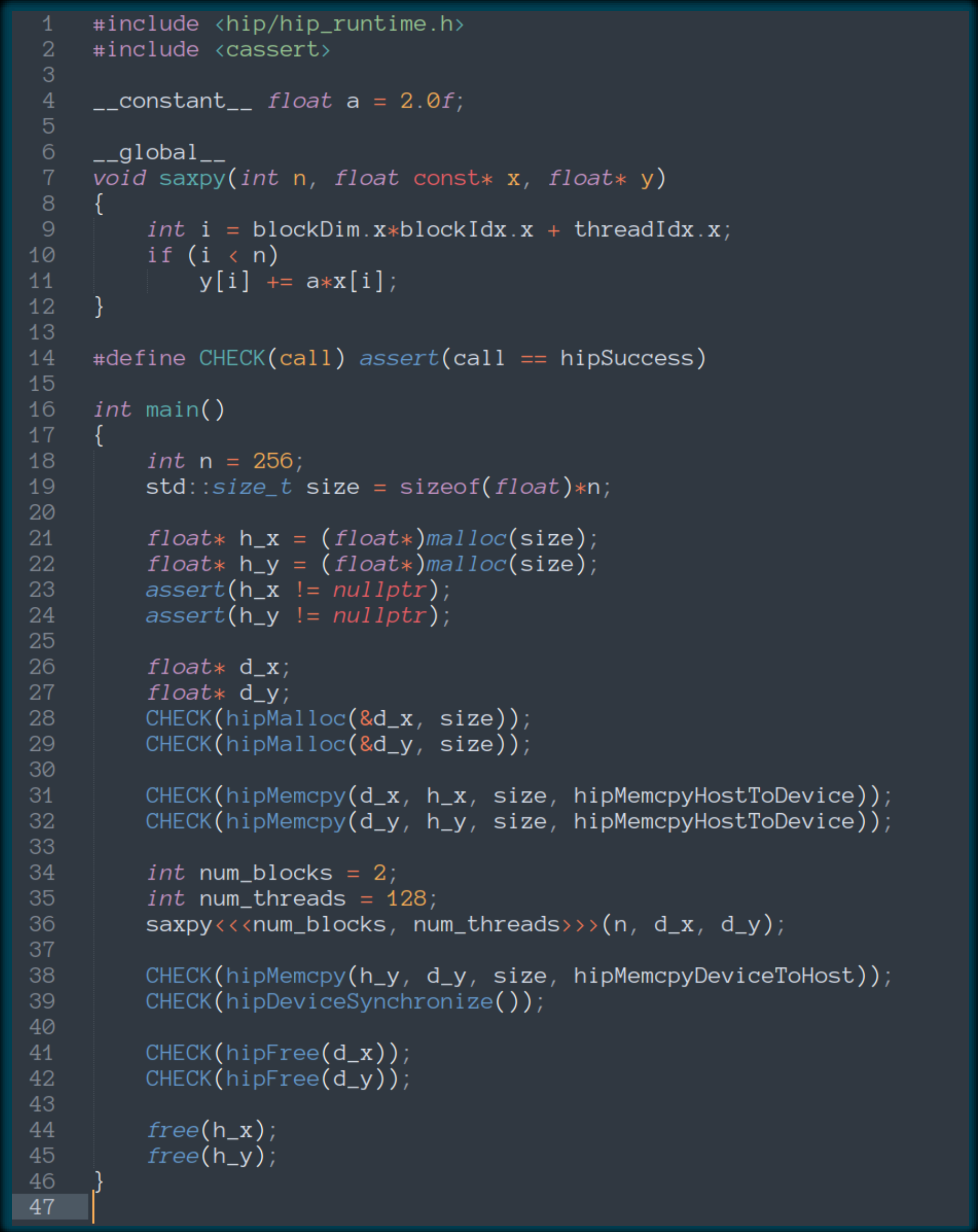
```
#include <cuda.h>
#include <cassert>

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}

#define CHECK(call) assert(call == cudaSuccess)

int main()
{
    int n = 256;
    std::size_t size = sizeof(float)*n;

    float* h_x = (float*)malloc(size);
    float* h_y = (float*)malloc(size);
    assert(h_x != nullptr);
    assert(h_y != nullptr);

    float* d_x;
    float* d_y;
    CHECK(cudaMalloc(&d_x, size));
    CHECK(cudaMalloc(&d_y, size));

    CHECK(cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_y, h_y, size, cudaMemcpyHostToDevice));

    int num_blocks = 2;
    int num_threads = 128;
    saxpy<<<num_blocks, num_threads>>>(n, d_x, d_y);

    CHECK(cudaMemcpy(h_y, d_y, size, cudaMemcpyDeviceToHost));
    CHECK(cudaDeviceSynchronize());

    CHECK(cudaFree(d_x));
    CHECK(cudaFree(d_y));

    free(h_x);
    free(h_y);
}
```

simple CUDA code



```cpp
#include <cuda.h>
#include <cassert>

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}

#define CHECK(call) assert(call == cudaSuccess)

int main()
{
    int n = 256;
    std::size_t size = sizeof(float)*n;

    float* h_x = (float*)malloc(size);
    float* h_y = (float*)malloc(size);
    assert(h_x != nullptr);
    assert(h_y != nullptr);

    float* d_x;
    float* d_y;
    CHECK(cudaMalloc(&d_x, size));
    CHECK(cudaMalloc(&d_y, size));

    CHECK(cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_y, h_y, size, cudaMemcpyHostToDevice));

    int num_blocks = 2;
    int num_threads = 128;
    saxpy<<<num_blocks, num_threads>>>(n, d_x, d_y);

    CHECK(cudaMemcpy(h_y, d_y, size, cudaMemcpyDeviceToHost));
    CHECK(cudaDeviceSynchronize());

    CHECK(cudaFree(d_x));
    CHECK(cudaFree(d_y));

    free(h_x);
    free(h_y);
}
```

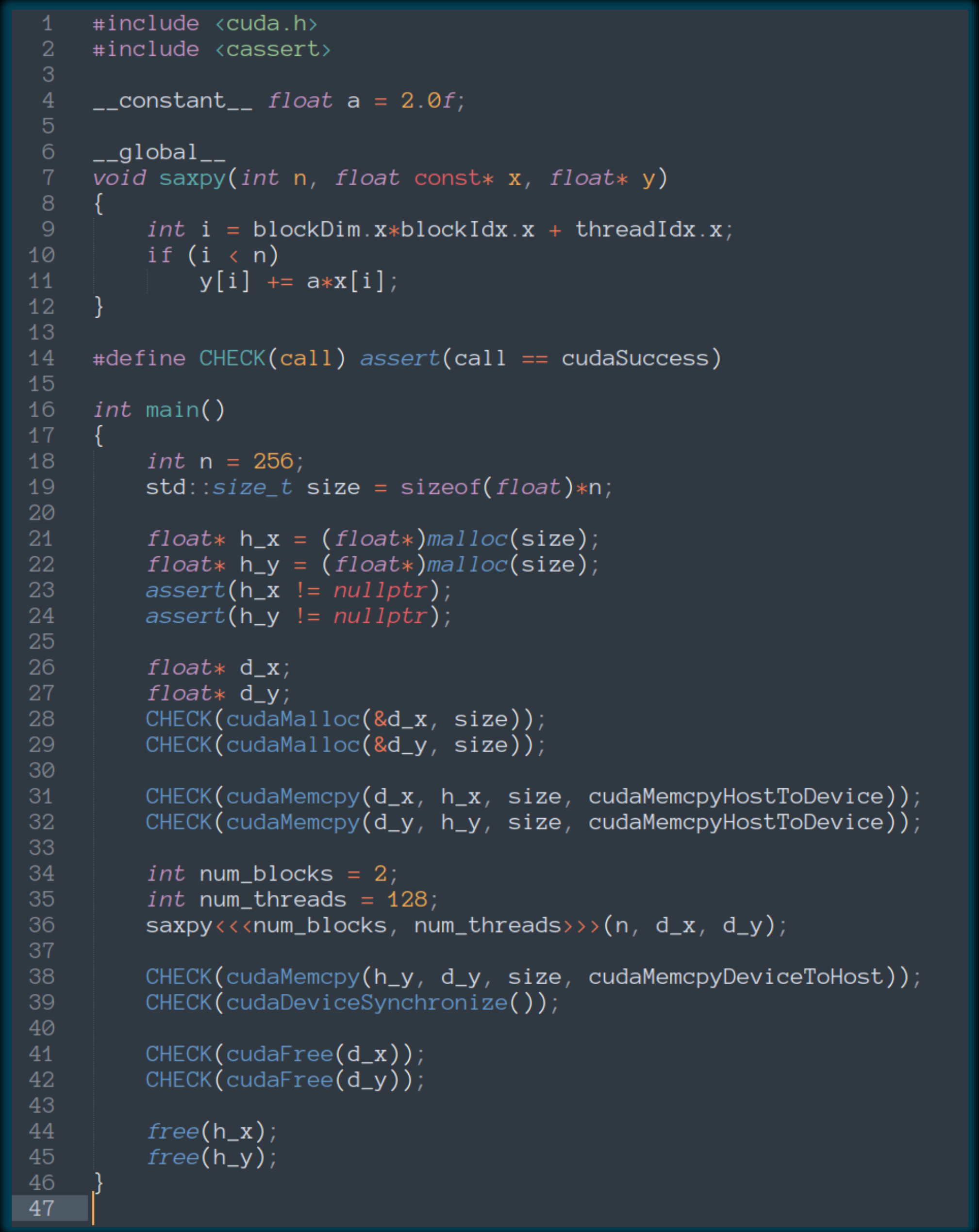
same code in HIP



```cpp
#include <hip/hip_runtime.h>
#include <cassert>

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}

#define CHECK(call) assert(call == hipSuccess)

int main()
{
    int n = 256;
    std::size_t size = sizeof(float)*n;

    float* h_x = (float*)malloc(size);
    float* h_y = (float*)malloc(size);
    assert(h_x != nullptr);
    assert(h_y != nullptr);

    float* d_x;
    float* d_y;
    CHECK(hipMalloc(&d_x, size));
    CHECK(hipMalloc(&d_y, size));

    CHECK(hipMemcpy(d_x, h_x, size, hipMemcpyHostToDevice));
    CHECK(hipMemcpy(d_y, h_y, size, hipMemcpyHostToDevice));

    int num_blocks = 2;
    int num_threads = 128;
    saxpy<<<num_blocks, num_threads>>>(n, d_x, d_y);

    CHECK(hipMemcpy(h_y, d_y, size, hipMemcpyDeviceToHost));
    CHECK(hipDeviceSynchronize());

    CHECK(hipFree(d_x));
    CHECK(hipFree(d_y));

    free(h_x);
    free(h_y);
}
```

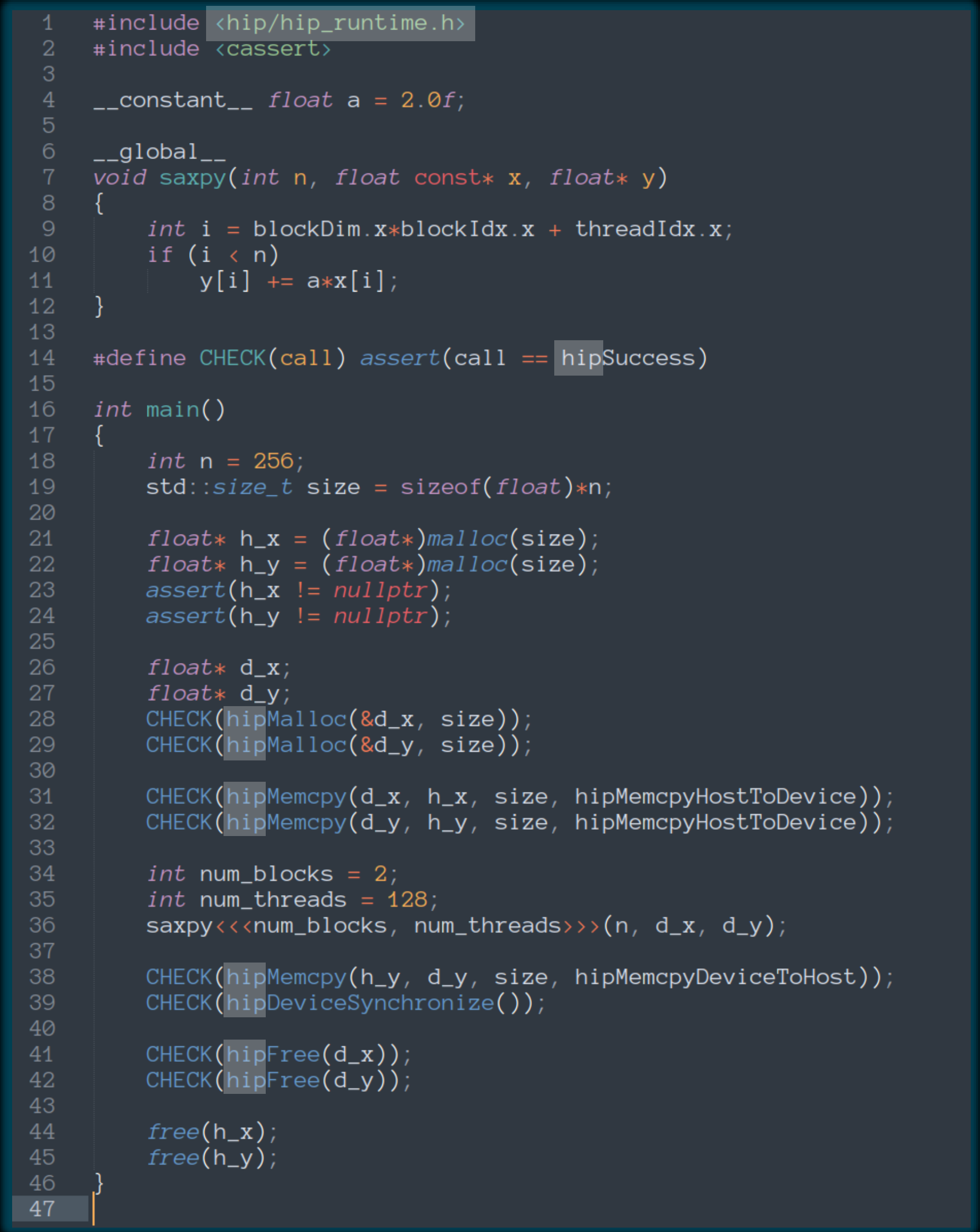
spot the differences

simple CUDA code



```
#include <cuda.h>
#include <cassert>

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}

#define CHECK(call) assert(call == cudaSuccess)

int main()
{
    int n = 256;
    std::size_t size = sizeof(float)*n;

    float* h_x = (float*)malloc(size);
    float* h_y = (float*)malloc(size);
    assert(h_x != nullptr);
    assert(h_y != nullptr);

    float* d_x;
    float* d_y;
    CHECK(cudaMalloc(&d_x, size));
    CHECK(cudaMalloc(&d_y, size));

    CHECK(cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_y, h_y, size, cudaMemcpyHostToDevice));

    int num_blocks = 2;
    int num_threads = 128;
    saxpy<<<num_blocks, num_threads>>>(n, d_x, d_y);

    CHECK(cudaMemcpy(h_y, d_y, size, cudaMemcpyDeviceToHost));
    CHECK(cudaDeviceSynchronize());

    CHECK(cudaFree(d_x));
    CHECK(cudaFree(d_y));

    free(h_x);
    free(h_y);
}
```

same code in HIP



```
#include <hip/hip_runtime.h>
#include <cassert>

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}

#define CHECK(call) assert(call == hipSuccess)

int main()
{
    int n = 256;
    std::size_t size = sizeof(float)*n;

    float* h_x = (float*)malloc(size);
    float* h_y = (float*)malloc(size);
    assert(h_x != nullptr);
    assert(h_y != nullptr);

    float* d_x;
    float* d_y;
    CHECK(hipMalloc(&d_x, size));
    CHECK(hipMalloc(&d_y, size));

    CHECK(hipMemcpy(d_x, h_x, size, hipMemcpyHostToDevice));
    CHECK(hipMemcpy(d_y, h_y, size, hipMemcpyHostToDevice));

    int num_blocks = 2;
    int num_threads = 128;
    saxpy<<<num_blocks, num_threads>>>(n, d_x, d_y);

    CHECK(hipMemcpy(h_y, d_y, size, hipMemcpyDeviceToHost));
    CHECK(hipDeviceSynchronize());

    CHECK(hipFree(d_x));
    CHECK(hipFree(d_y));

    free(h_x);
    free(h_y);
}
```

HIPIFY TOOLS

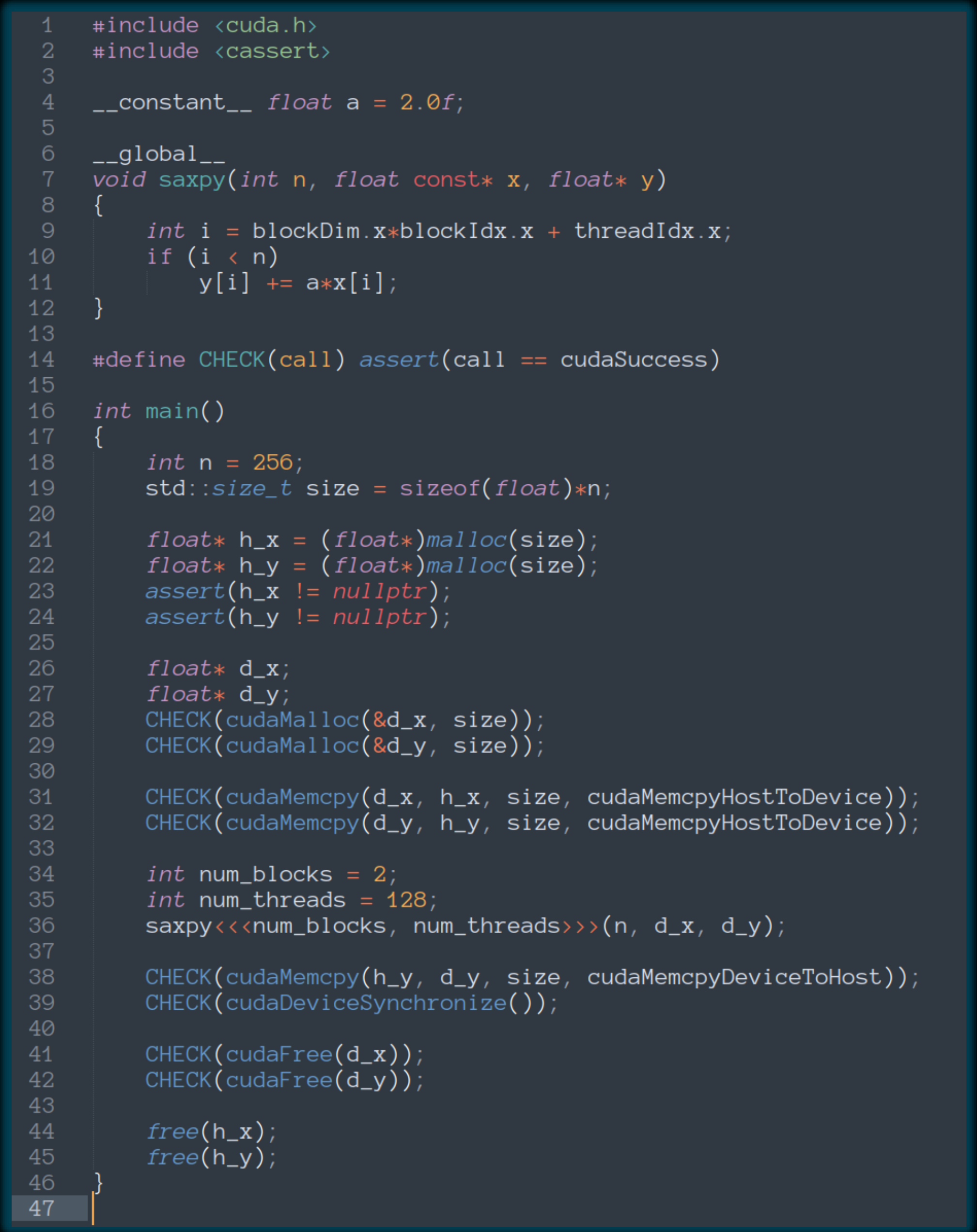
hipify-clang

- compiler (clang) based translator
- handles very complex constructs
- prints an error if not able to translate
- supports clang options
- requires CUDA

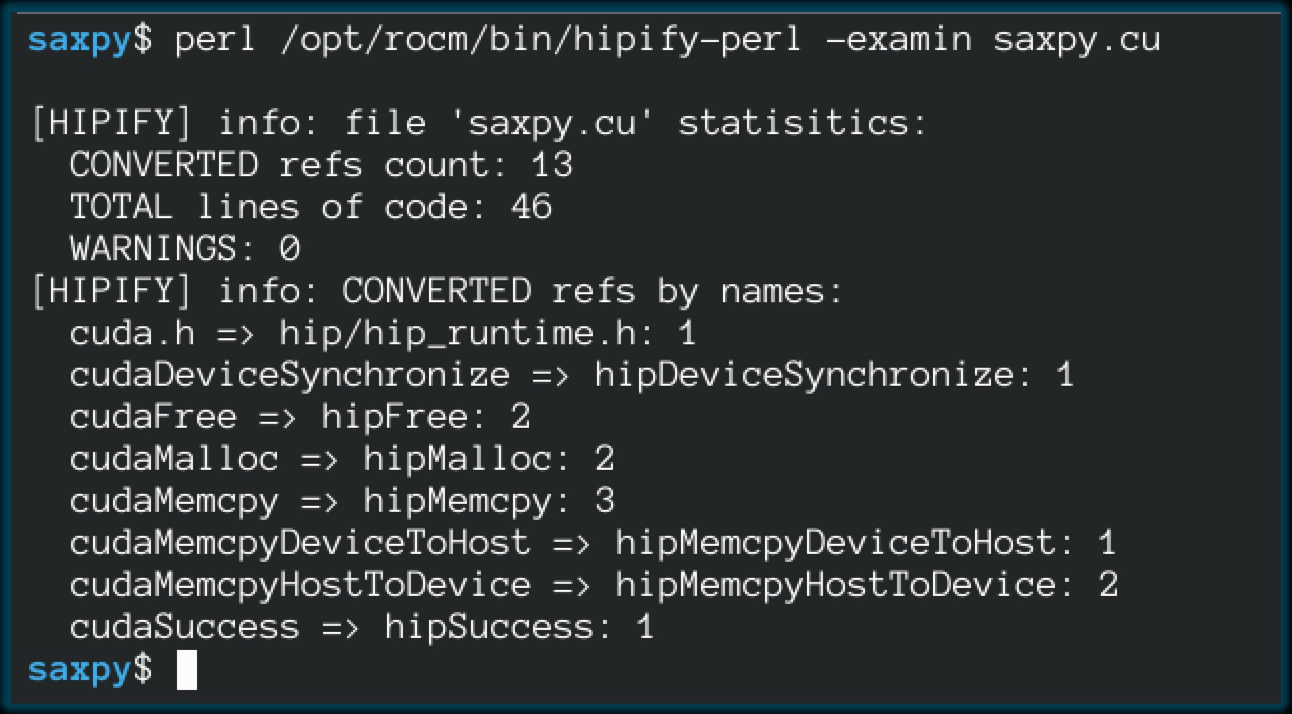
hipify-perl

- Perl® script
- relies on regular expressions
- may struggle with complex constructs
- does not require CUDA

https://github.com/ROCm-Developer-Tools/HIPIFY



```
#include <cuda.h>
#include <cassert>

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}

#define CHECK(call) assert(call == cudaSuccess)

int main()
{
    int n = 256;
    std::size_t size = sizeof(float)*n;

    float* h_x = (float*)malloc(size);
    float* h_y = (float*)malloc(size);
    assert(h_x != nullptr);
    assert(h_y != nullptr);

    float* d_x;
    float* d_y;
    CHECK(cudaMalloc(&d_x, size));
    CHECK(cudaMalloc(&d_y, size));

    CHECK(cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_y, h_y, size, cudaMemcpyHostToDevice));

    int num_blocks = 2;
    int num_threads = 128;
    saxpy<<<num_blocks, num_threads>>>(n, d_x, d_y);

    CHECK(cudaMemcpy(h_y, d_y, size, cudaMemcpyDeviceToHost));
    CHECK(cudaDeviceSynchronize());

    CHECK(cudaFree(d_x));
    CHECK(cudaFree(d_y));

    free(h_x);
    free(h_y);
}
```



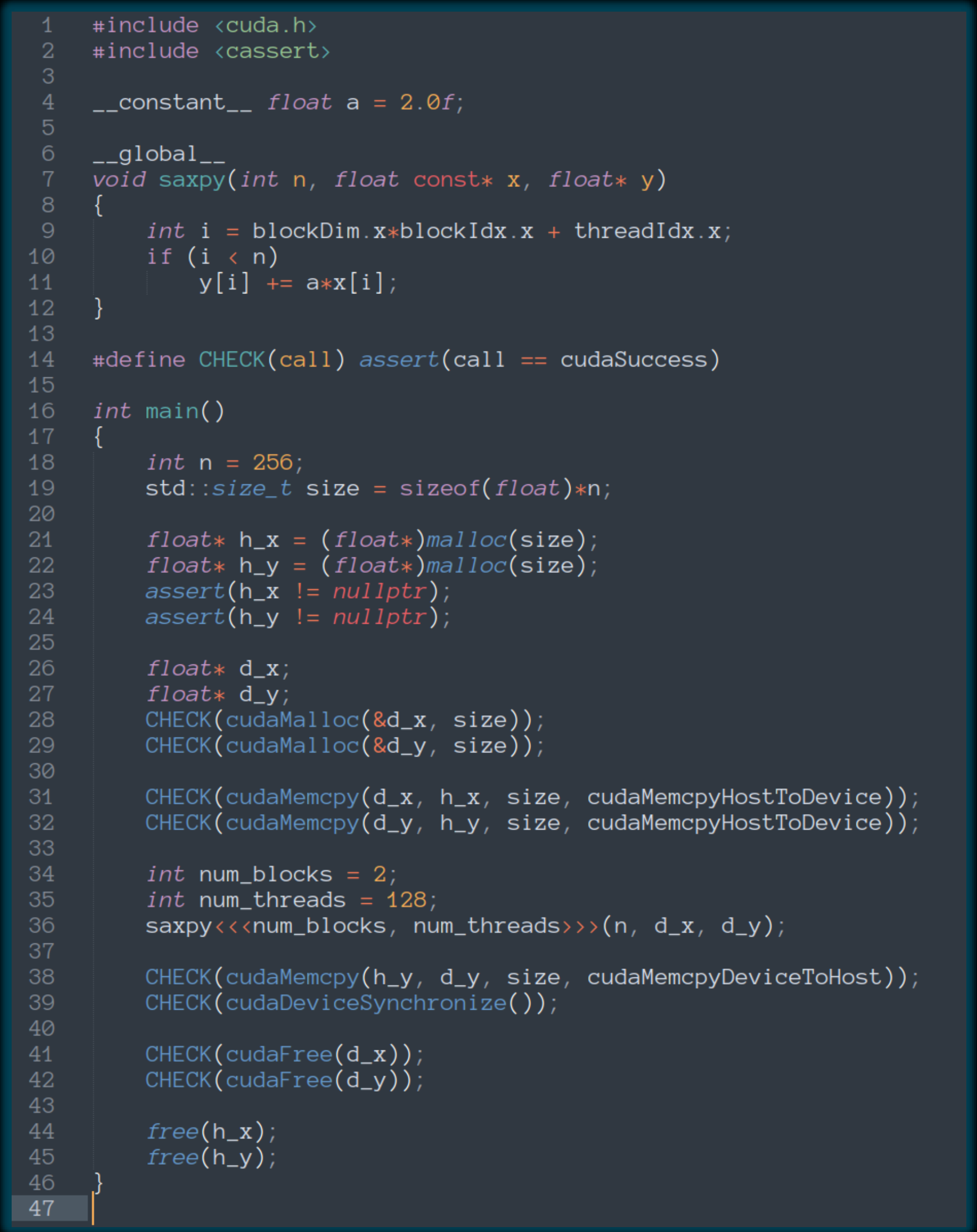
```
saxpy$ perl /opt/rocm/bin/hipify-perl -examin saxpy.cu

[HIPIFY] info: file 'saxpy.cu' statisitics:
  CONVERTED refs count: 13
  TOTAL lines of code: 46
  WARNINGS: 0
[HIPIFY] info: CONVERTED refs by names:
  cuda.h => hip/hip_runtime.h: 1
  cudaDeviceSynchronize => hipDeviceSynchronize: 1
  cudaFree => hipFree: 2
  cudaMalloc => hipMalloc: 2
  cudaMemcpy => hipMemcpy: 3
  cudaMemcpyDeviceToHost => hipMemcpyDeviceToHost: 1
  cudaMemcpyHostToDevice => hipMemcpyHostToDevice: 2
  cudaSuccess => hipSuccess: 1
saxpy$
```

hipify-perl

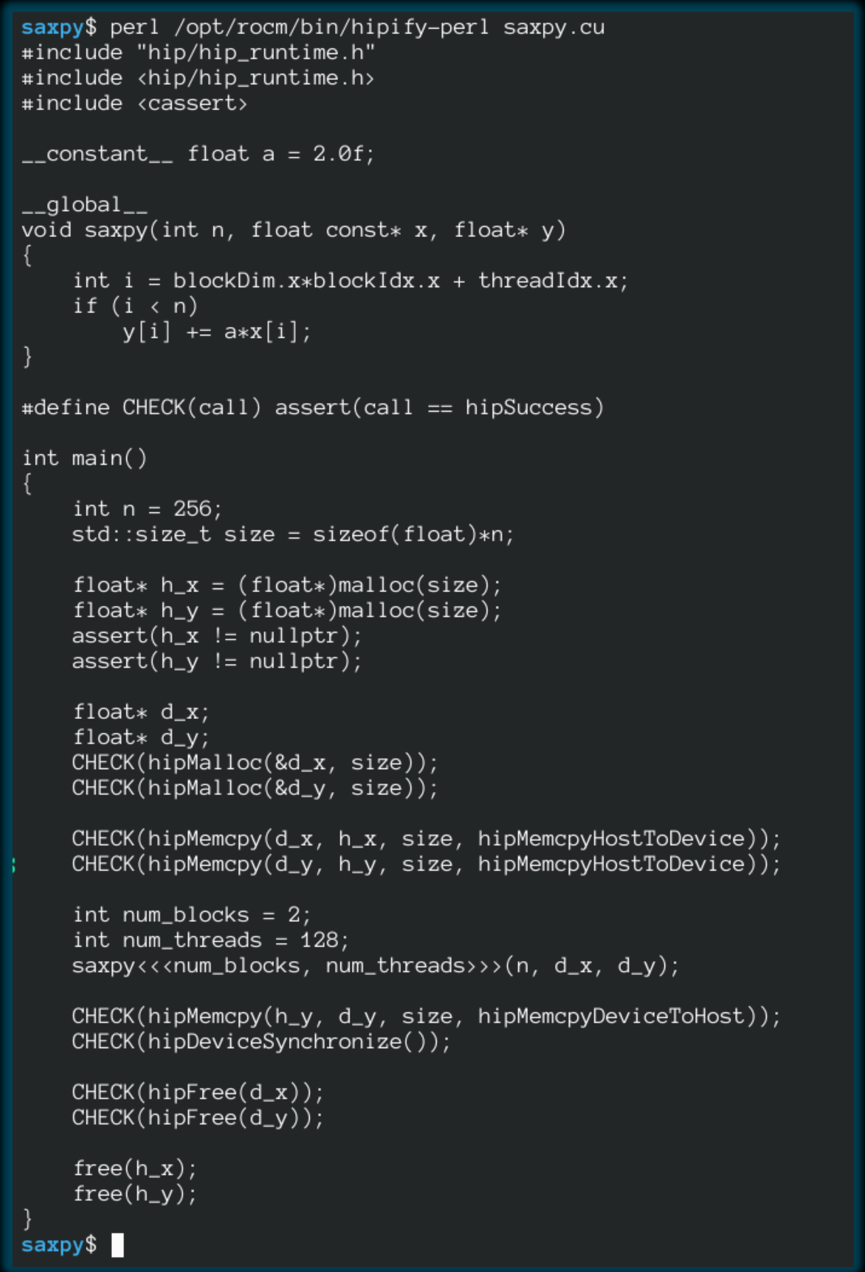
`hipify-perl -examin`

- for initial assessment
- no replacements done
- prints basic statistics and the number of replacements

hipify-perl



```
#include <cuda.h>
#include <cassert>

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}

#define CHECK(call) assert(call == cudaSuccess)

int main()
{
    int n = 256;
    std::size_t size = sizeof(float)*n;

    float* h_x = (float*)malloc(size);
    float* h_y = (float*)malloc(size);
    assert(h_x != nullptr);
    assert(h_y != nullptr);

    float* d_x;
    float* d_y;
    CHECK(cudaMalloc(&d_x, size));
    CHECK(cudaMalloc(&d_y, size));

    CHECK(cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_y, h_y, size, cudaMemcpyHostToDevice));

    int num_blocks = 2;
    int num_threads = 128;
    saxpy<<<num_blocks, num_threads>>>(n, d_x, d_y);

    CHECK(cudaMemcpy(h_y, d_y, size, cudaMemcpyDeviceToHost));
    CHECK(cudaDeviceSynchronize());

    CHECK(cudaFree(d_x));
    CHECK(cudaFree(d_y));

    free(h_x);
    free(h_y);
}
```



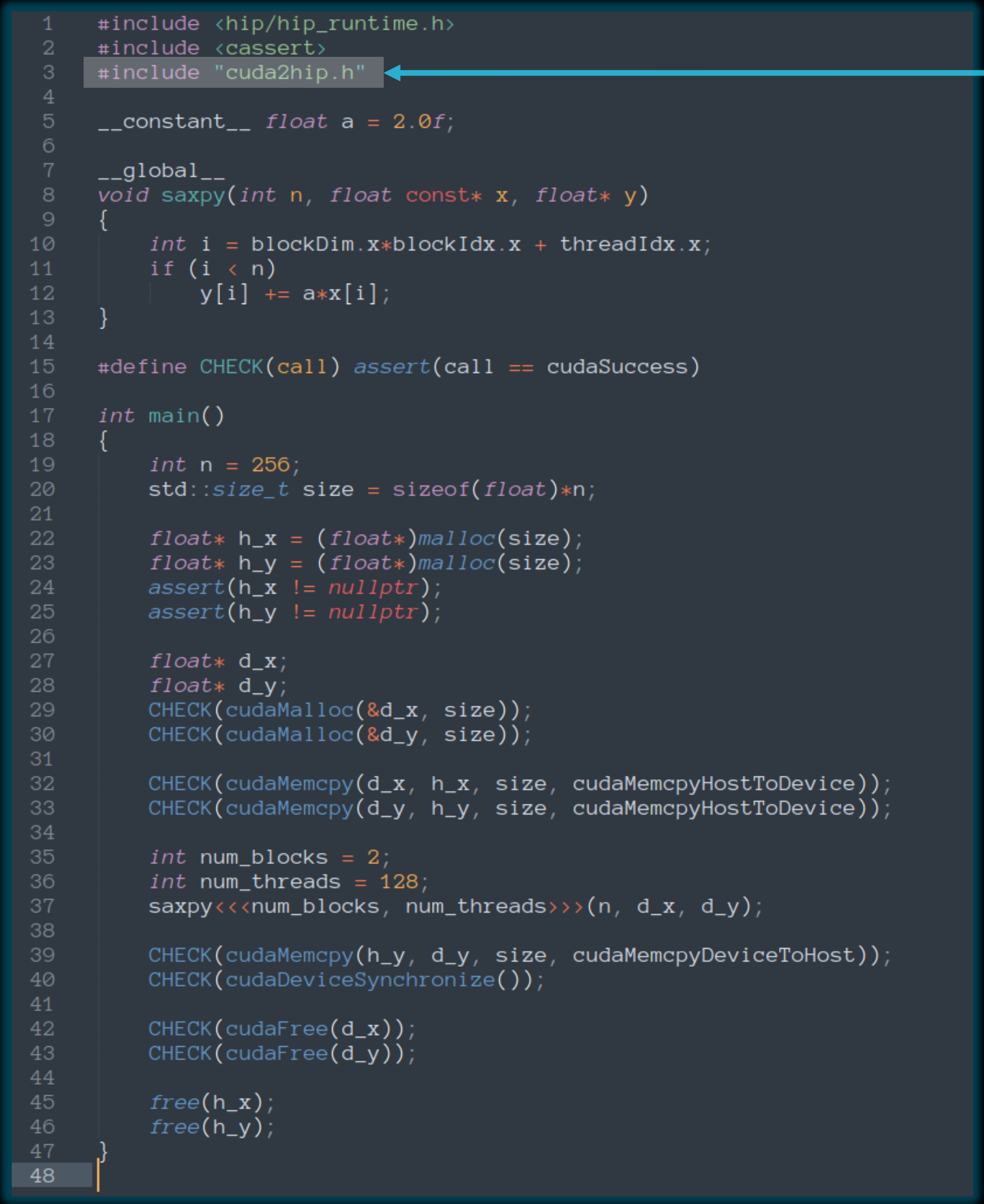
```
saxpy$ perl /opt/rocm/bin/hipify-perl saxpy.cu
#include "hip/hip_runtime.h"
#include <hip/hip_runtime.h>
#include <cassert>

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}

#define CHECK(call) assert(call == hipSuccess)

int main()
{
    int n = 256;
    std::size_t size = sizeof(float)*n;

    float* h_x = (float*)malloc(size);
    float* h_y = (float*)malloc(size);
    assert(h_x != nullptr);
    assert(h_y != nullptr);

    float* d_x;
    float* d_y;
    CHECK(hipMalloc(&d_x, size));
    CHECK(hipMalloc(&d_y, size));

    CHECK(hipMemcpy(d_x, h_x, size, hipMemcpyHostToDevice));
    CHECK(hipMemcpy(d_y, h_y, size, hipMemcpyHostToDevice));

    int num_blocks = 2;
    int num_threads = 128;
    saxpy<<<num_blocks, num_threads>>>(n, d_x, d_y);

    CHECK(hipMemcpy(h_y, d_y, size, hipMemcpyDeviceToHost));
    CHECK(hipDeviceSynchronize());

    CHECK(hipFree(d_x));
    CHECK(hipFree(d_y));

    free(h_x);
    free(h_y);
}
saxpy$
```

translating a file to standard output

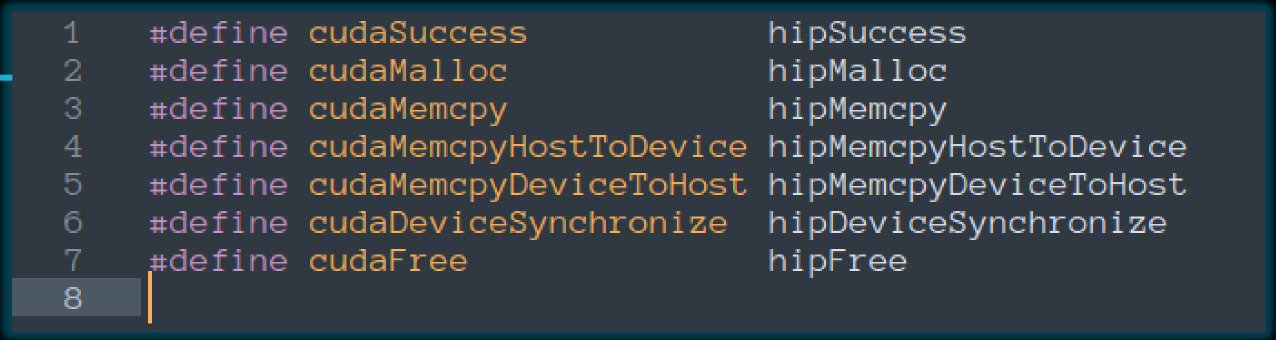
but can also

- translate in place
- preserve orig copy
- recursively do folders

together we advance_



```cpp
#include <hip/hip_runtime.h>
#include <cassert>
#include "cuda2hip.h"

__constant__ float a = 2.0f;

__global__
void saxpy(int n, float const* x, float* y)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < n)
        y[i] += a*x[i];
}

#define CHECK(call) assert(call == cudaSuccess)

int main()
{
    int n = 256;
    std::size_t size = sizeof(float)*n;

    float* h_x = (float*)malloc(size);
    float* h_y = (float*)malloc(size);
    assert(h_x != nullptr);
    assert(h_y != nullptr);

    float* d_x;
    float* d_y;
    CHECK(cudaMalloc(&d_x, size));
    CHECK(cudaMalloc(&d_y, size));

    CHECK(cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_y, h_y, size, cudaMemcpyHostToDevice));

    int num_blocks = 2;
    int num_threads = 128;
    saxpy<<<num_blocks, num_threads>>>(n, d_x, d_y);

    CHECK(cudaMemcpy(h_y, d_y, size, cudaMemcpyDeviceToHost));
    CHECK(cudaDeviceSynchronize());

    CHECK(cudaFree(d_x));
    CHECK(cudaFree(d_y));

    free(h_x);
    free(h_y);
}
```



```cpp
#define cudaSuccess                hipSuccess
#define cudaMalloc                 hipMalloc
#define cudaMemcpy                 hipMemcpy
#define cudaMemcpyHostToDevice     hipMemcpyHostToDevice
#define cudaMemcpyDeviceToHost     hipMemcpyDeviceToHost
#define cudaDeviceSynchronize      hipDeviceSynchronize
#define cudaFree                   hipFree
```

alternatively

- create a file with renaming macros
- include conditionally, depending on target

AMD
together we advance_

OPTIMIZATION TECHNIQUES

basic

- thread divergence / SIMDzation
- reuse in shared memory & bank conflicts
- coalescing of global memory accesses
- resource partitioning / occupancy / spills
- L1, L2 cache blocking
- ...

advanced

- atomics
- warp primitives
- CPU-GPU coherence
- inter-stream synchronization
- ...

DIFFERENCES FROM CUDA

- warpSize
  - 64 on AMD
  - 32 on NVIDIA
- dynamic parallelism not supported
- exercise caution:
  - atomics
  - managed memory
  - warp-level primitives
  - inter-process communication

AMD RESOURCES

DOCUMENTATION AND TRAINING

AMD ROCM DEVELOPER HUB

Engage with ROCm Experts

- Participate in ROCm Webinar Series
- Post questions, view FAQ's in Community Forum

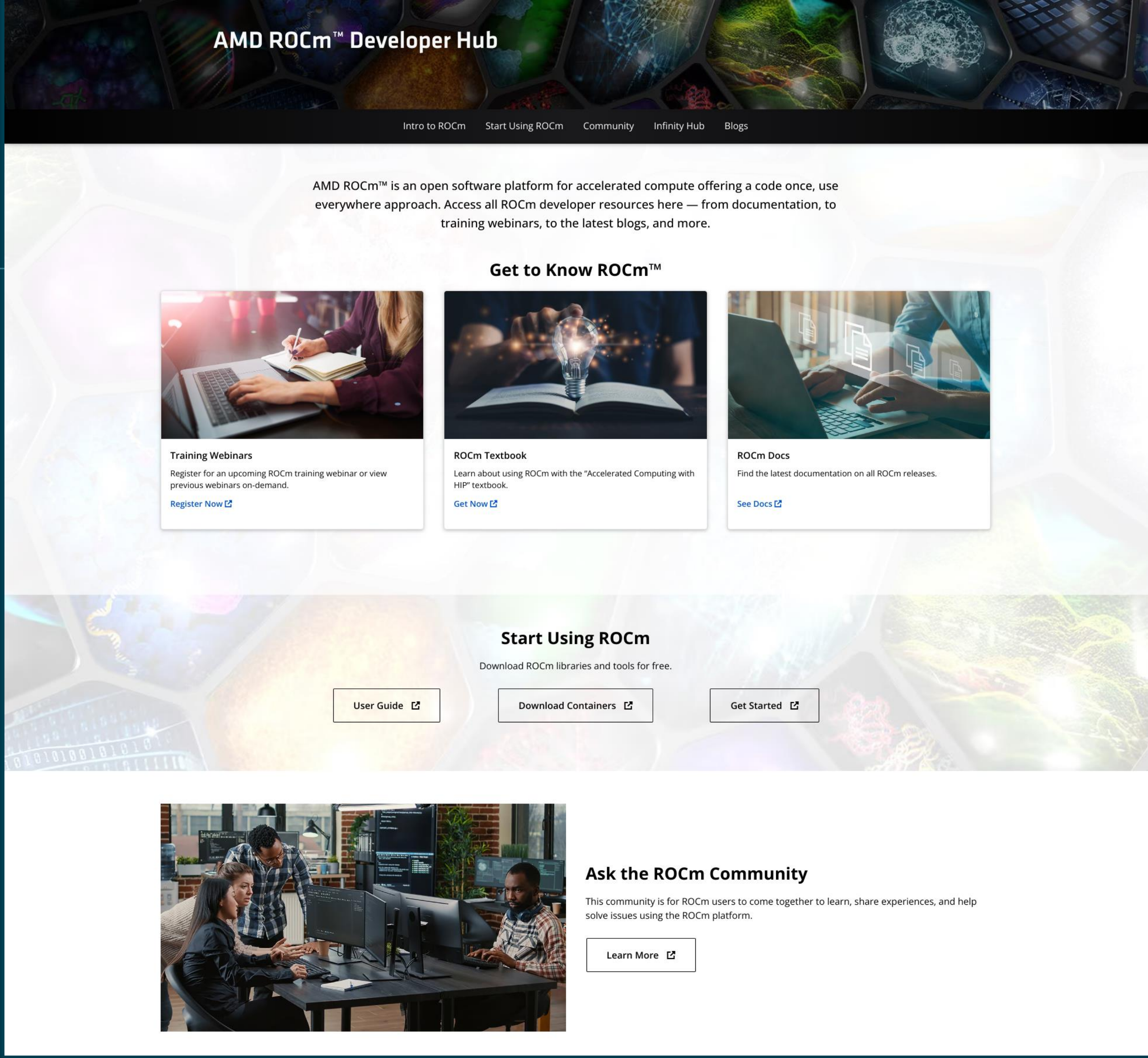
Increase Understanding

- Purchase ROCm Text Book
- View the latest news in the Blogs

Get Started Using ROCm

- ROCm Documentation on GitHub
- Download the Latest Version of ROCm

https://www.amd.com/en/developer/rocm-hub.html



AMD ROCm™ is an open software platform for accelerated compute offering a code once, use everywhere approach. Access all ROCm developer resources here — from documentation, to training webinars, to the latest blogs, and more.

Get to Know ROCm™

Training Webinars
Register for an upcoming ROCm training webinar or view previous webinars on-demand.
Register Now

ROCm Textbook
Learn about using ROCm with the "Accelerated Computing with HIP" textbook.
Get Now

ROCm Docs
Find the latest documentation on all ROCm releases.
See Docs

Start Using ROCm
Download ROCm libraries and tools for free.
User Guide | Download Containers | Get Started

Ask the ROCm Community
This community is for ROCm users to come together to learn, share experiences, and help solve issues using the ROCm platform.
Learn More

AMD
together we advance_

NEW ROCM DOCS

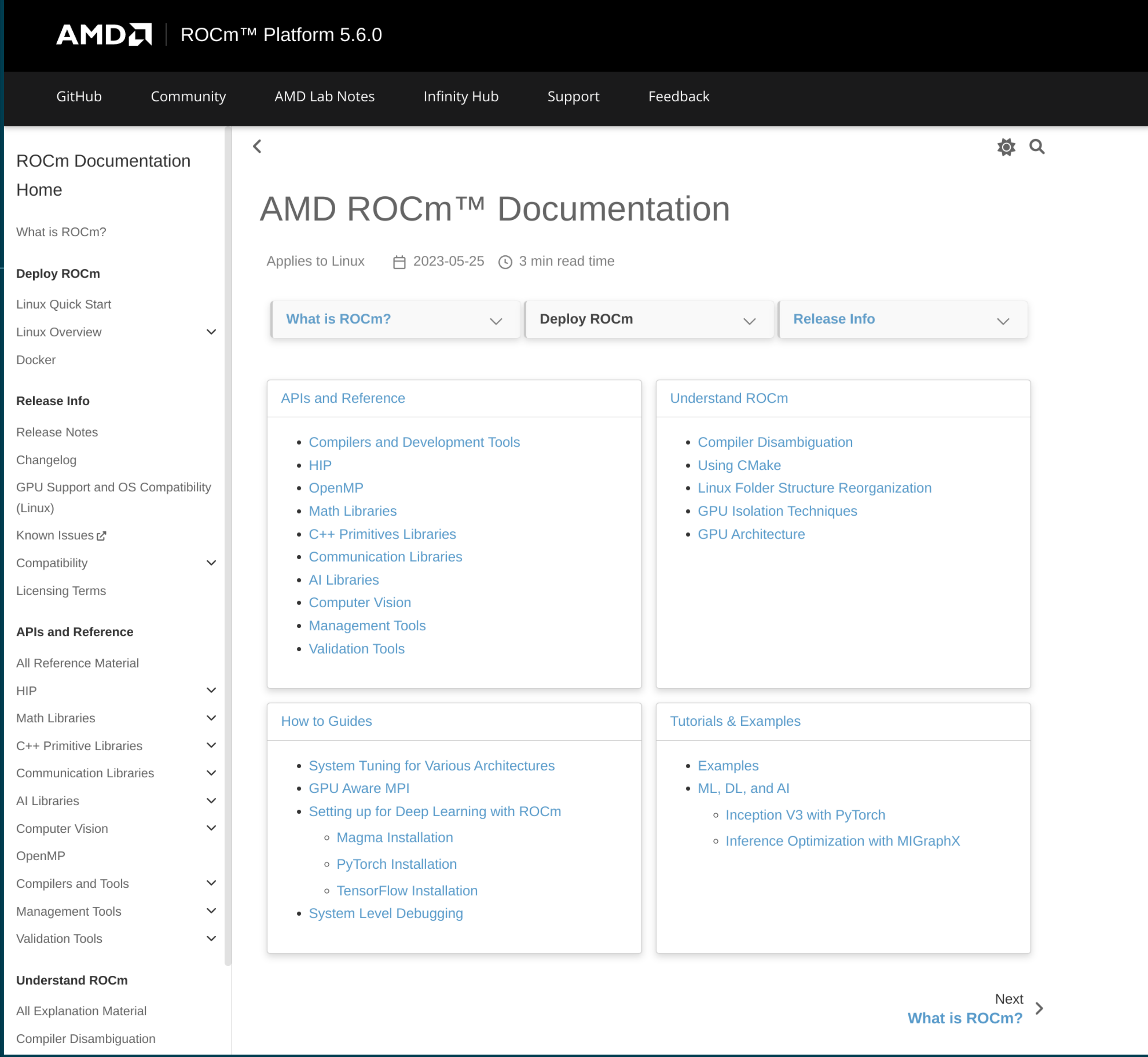
Comprehensive Coverage

- Compilers and Frameworks
- Math libraries, communication libraries
- Management tools, validation tools
- …

Howto Guides

- Installation
- Tunning
- Debugging
- …

https://rocm.docs.amd.com/



AMD | ROCm™ Platform 5.6.0

GitHub Community AMD Lab Notes Infinity Hub Support Feedback

ROCm Documentation Home

What is ROCm?

Deploy ROCm

Linux Quick Start
Linux Overview
Docker

Release Info

Release Notes
Changelog
GPU Support and OS Compatibility (Linux)
Known Issues
Compatibility
Licensing Terms

APIs and Reference

All Reference Material
HIP
Math Libraries
C++ Primitive Libraries
Communication Libraries
AI Libraries
Computer Vision
OpenMP
Compilers and Tools
Management Tools
Validation Tools

Understand ROCm

All Explanation Material
Compiler Disambiguation

AMD ROCm™ Documentation

Applies to Linux 2023-05-25 3 min read time

What is ROCm? | Deploy ROCm | Release Info

APIs and Reference

- Compilers and Development Tools
- HIP
- OpenMP
- Math Libraries
- C++ Primitives Libraries
- Communication Libraries
- AI Libraries
- Computer Vision
- Management Tools
- Validation Tools

Understand ROCm

- Compiler Disambiguation
- Using CMake
- Linux Folder Structure Reorganization
- GPU Isolation Techniques
- GPU Architecture

How to Guides

- System Tuning for Various Architectures
- GPU Aware MPI
- Setting up for Deep Learning with ROCm
  - Magma Installation
  - PyTorch Installation
  - TensorFlow Installation
- System Level Debugging

Tutorials & Examples

- Examples
- ML, DL, and AI
  - Inception V3 with PyTorch
  - Inference Optimization with MIGraphX

Next
What is ROCm?

AMD together we advance_

HIP TEXTBOOK

Comprehensive Coverage

- HIP Language
- AMD GPU Internals
- Performance Analysis
- Debugging
- Programming Patterns
- ROCm Libraries
- Porting to HIP
- Multi-GPU Programming
- Third Party Tools
- CDNA Assembly
- ML with ROCm

https://www.barnesandnoble.com/w/accelerated-computing-with-hip-yifan-sun/1142866934

AMD INFINITY HUB

AMD Instinct™ MI200 SUPPORT

29 key applications & frameworks on Infinity Hub & a catalogue supporting over 90 applications, frameworks & tools

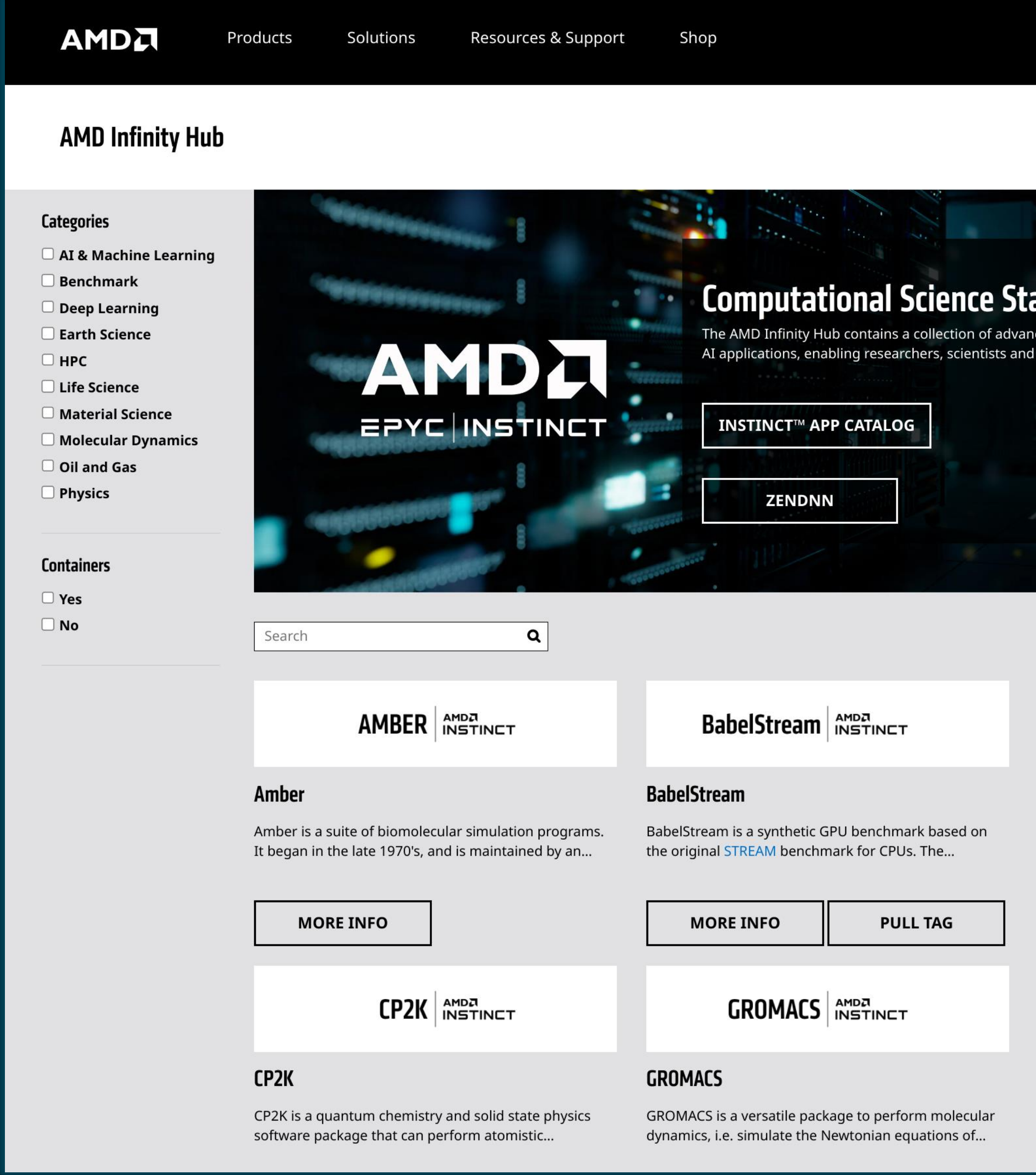
Accelerating Instinct™ adoption

Over 17000 application pulls. 10000+ since last year

PERFORMANCE RESULTS

Published Performance Results for Select Apps / Benchmarks

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



AMD Infinity Hub

Categories

- AI & Machine Learning
- Benchmark
- Deep Learning
- Earth Science
- HPC
- Life Science
- Material Science
- Molecular Dynamics
- Oil and Gas
- Physics

Containers

- Yes
- No

Computational Science Sta

The AMD Infinity Hub contains a collection of advan
AI applications, enabling researchers, scientists and

Amber

Amber is a suite of biomolecular simulation programs. It began in the late 1970's, and is maintained by an...

BabelStream

BabelStream is a synthetic GPU benchmark based on the original STREAM benchmark for CPUs. The...

CP2K

CP2K is a quantum chemistry and solid state physics software package that can perform atomistic...

GROMACS

GROMACS is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of...

AMD
together we advance_

SOFTWARE CATALOG

STRONG MOMENTUM AND INCREASING LIST OF SUPORTED APPLICATION, LIBRARIES & FRAMEWORKS

Life Science

AMBER
GROMACS
NAMD
LAMMPS
Hoomd-Blue
VASP

Physics

MILC
GRID
QUANTUM ESPRESSO
N-Body
CHROMA
PIConGPU
QuickSilver

Chemistry

CP2K
QUDA
NWCHEM
TERACHEM
QMCPACK

CFD

OpenFOAM®
AMR-WIND
NEKBONE
LAGHOS
NEKO
NEKRS
PeleC

Earth Science

EXAGO
DEVITO
OCCA
SPECFEM3D-GLOBE
SPECFEM3D-CARTESIAN
ACECAST (WRF)
MPAS
ICON

Benchmarks

HPL
HPCG
AMG
ML - TORCHBENCH
ML - SUPERBENCH

Libraries

AMR-EX
Ginkko
HYPRE
TRILINOS

ML Frameworks

PYTORCH
TENSORFLOW
JAX
ONNX
OPENAI TRITON

ISV Applications

ANSYS MECHANICAL
CADENCE CHARLES
ANSYS FLUENT*
SIEMENS® STAR-CCM+*
SIEMENS® CALIBRE*

+ MANY MORE

* Porting/optimization in progress

AMD
together we advance_

AMD LAB NOTES

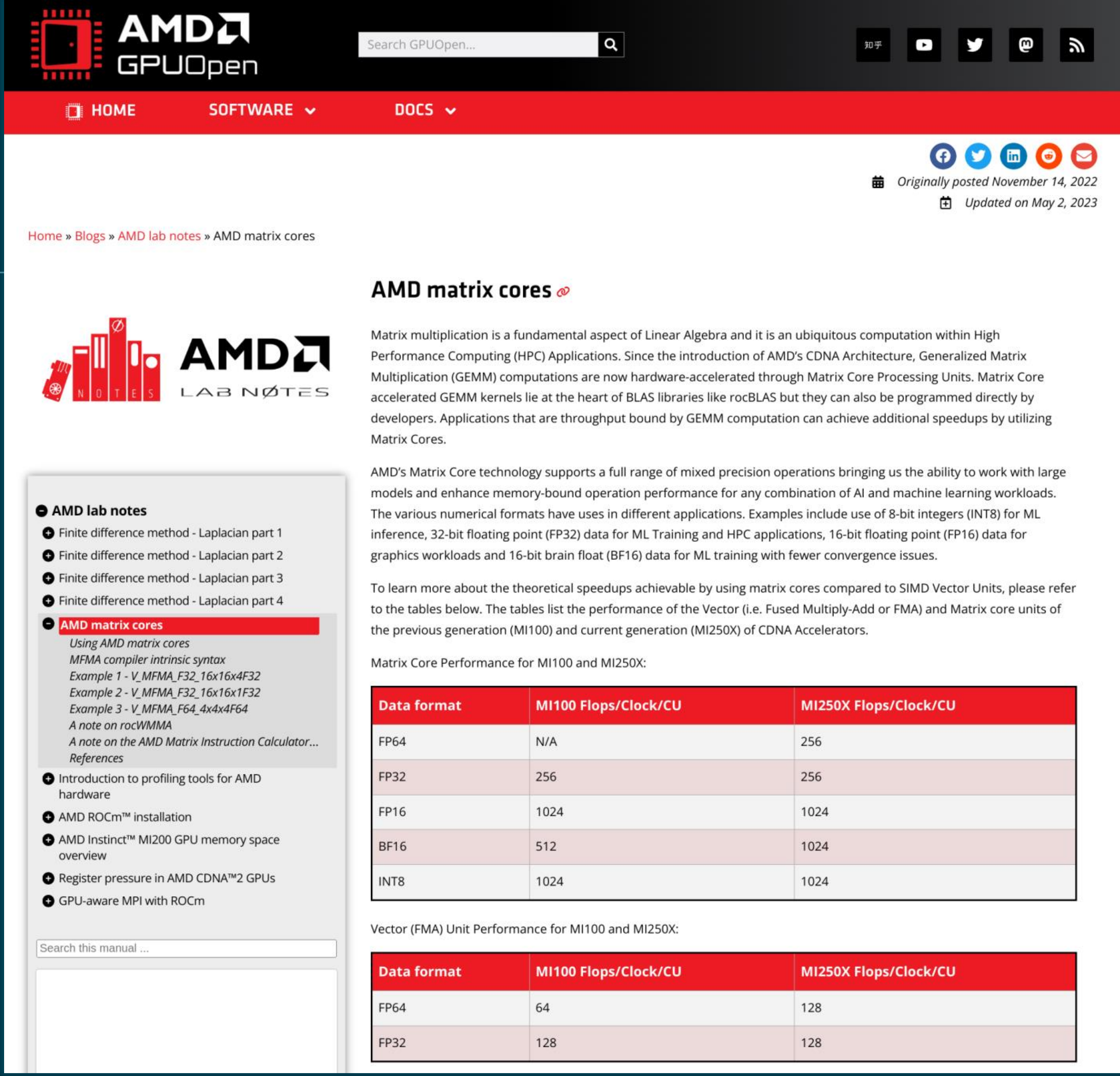
Introductory Topics

- ROCm installation
- Basics of HIP programming
- …

Advanced Topics

- Matrix Cores
- Register pressure
- GPU-aware MPI
- …

https://gpuopen.com/learn/amd-lab-notes/
https://github.com/AMD/amd-lab-notes



Originally posted November 14, 2022
Updated on May 2, 2023

Home » Blogs » AMD lab notes » AMD matrix cores

Search this manual ...

AMD matrix cores

Matrix multiplication is a fundamental aspect of Linear Algebra and it is an ubiquitous computation within High Performance Computing (HPC) Applications. Since the introduction of AMD's CDNA Architecture, Generalized Matrix Multiplication (GEMM) computations are now hardware-accelerated through Matrix Core Processing Units. Matrix Core accelerated GEMM kernels lie at the heart of BLAS libraries like rocBLAS but they can also be programmed directly by developers. Applications that are throughput bound by GEMM computation can achieve additional speedups by utilizing Matrix Cores.

AMD's Matrix Core technology supports a full range of mixed precision operations bringing us the ability to work with large models and enhance memory-bound operation performance for any combination of AI and machine learning workloads. The various numerical formats have uses in different applications. Examples include use of 8-bit integers (INT8) for ML inference, 32-bit floating point (FP32) data for ML Training and HPC applications, 16-bit floating point (FP16) data for graphics workloads and 16-bit brain float (BF16) data for ML training with fewer convergence issues.

To learn more about the theoretical speedups achievable by using matrix cores compared to SIMD Vector Units, please refer to the tables below. The tables list the performance of the Vector (i.e. Fused Multiply-Add or FMA) and Matrix core units of the previous generation (MI100) and current generation (MI250X) of CDNA Accelerators.

Matrix Core Performance for MI100 and MI250X:

Data format	MI100 Flops/Clock/CU	MI250X Flops/Clock/CU
FP64	N/A	256
FP32	256	256
FP16	1024	1024
BF16	512	1024
INT8	1024	1024

Vector (FMA) Unit Performance for MI100 and MI250X:

Data format	MI100 Flops/Clock/CU	MI250X Flops/Clock/CU
FP64	64	128
FP32	128	128

DISCLAIMERS

ATTRIBUTIONS

AMD
together we advance_