Introduction to High-Performance GPU Programming

Dr. Hari Sadasivan,
AI Group, AMD

About Me

Dr. Hari Sadasivan
MTS SDE in AI Group, AMD |
Faculty (part-time), UW Seattle |
Sr Member, IEEE

- Co-founder of AMD –omics group
- Tech-lead, AMD CoE in AI at UW Seattle
    - CSE/ECE PMP 590: Applied Parallel Programming on GPU

Home: https://harisadasivan.com/ Contact: hari.sadasivan@amd.com

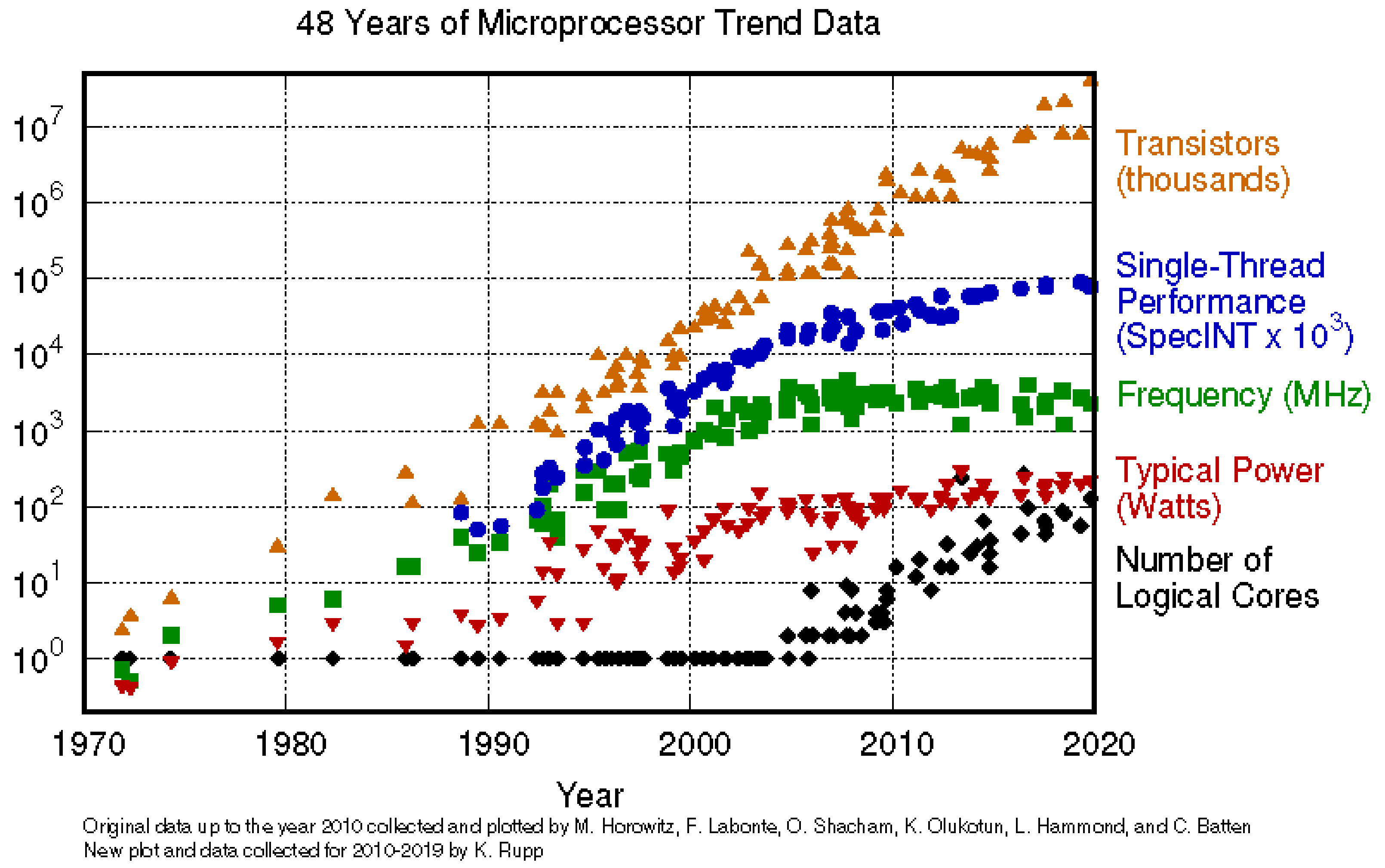
Agenda

GPUs

- Evolution
- CPU vs GPU
- SIMT Programming Model
- Programming GPUs
  - Put together a complete HIP code
- Architecture
- Profiling

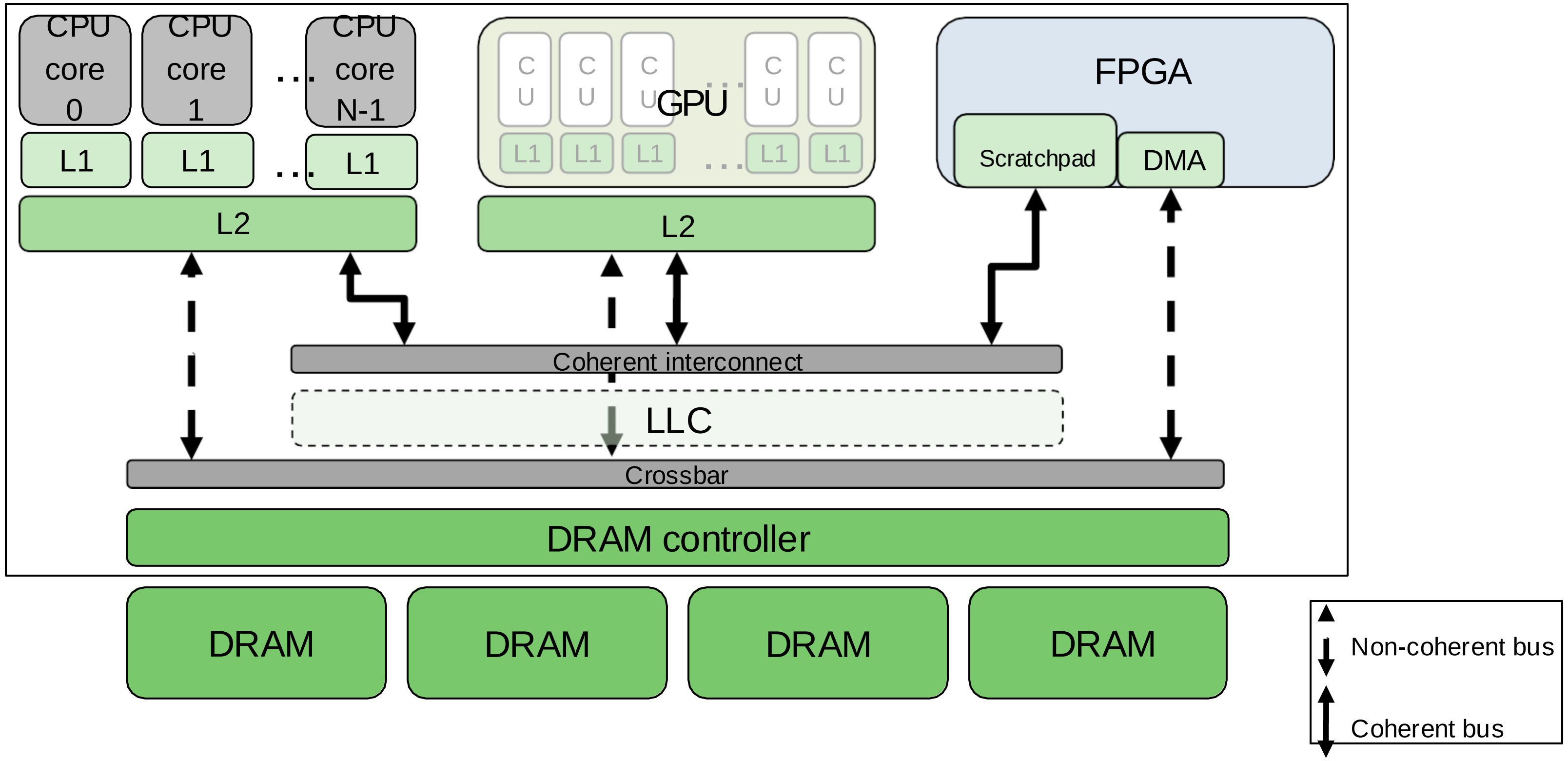
- The world of GPU Optimizations for AI
  - AMD Hiring

HW: End of Moore’s Law



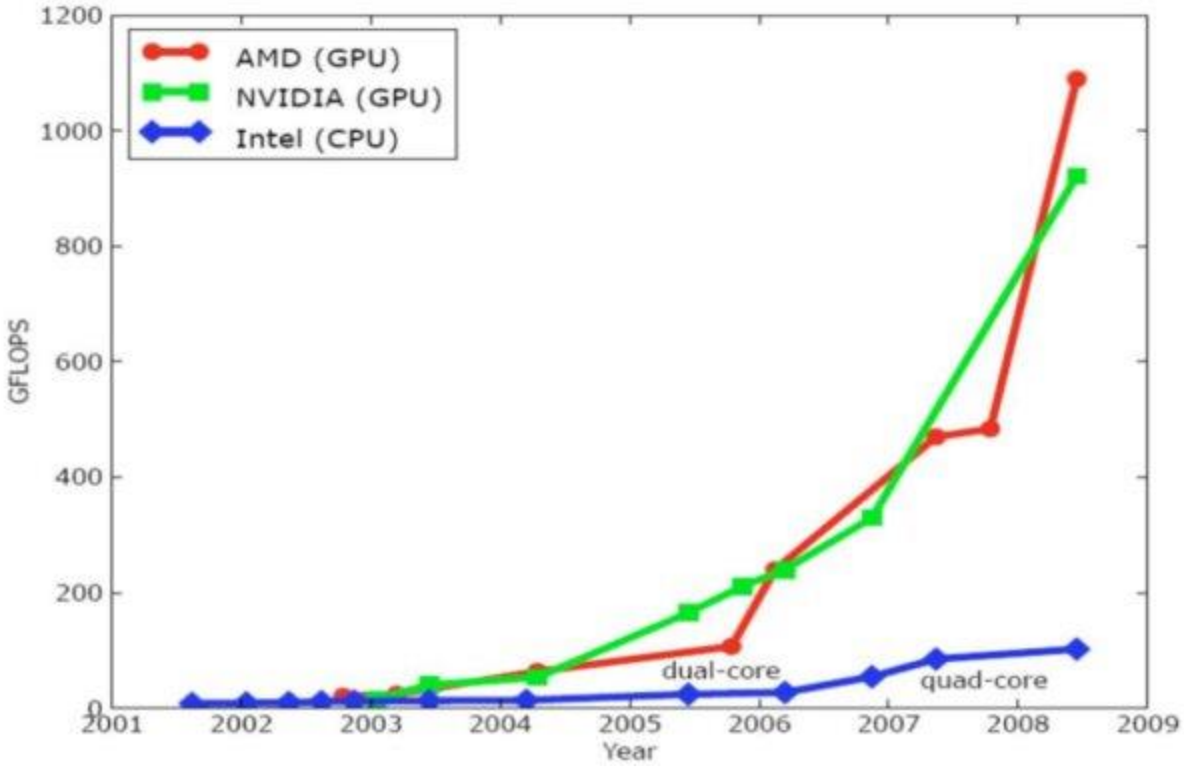
HW: Heterogeneous Computing Systems

- The end of Moore's law created the need for heterogeneous systems
  - More suitable devices for each type of workload
  - Increased performance and energy efficiency



Credits: Prof. Reetu Das, UM

HW: Silent revolution in Graphics



An Effective Dynamic Scheduling Runtime and Tuning System for Heterogeneous Multi and Many-Core Desktop Platforms, Binotto et al.

HW: GPUs in High Performance Computing

Computing at Exascale

El Capitan at Lawrence Livermore National Laboratory (LLNL) uses AMD’s MI300A

Performance expected to exceed 2ExaFLOPs, which comes with a $600 million price tag.

Frontier- World's fastest supercomputer

- Oak Ridge National Lab
- Uses MI250X and 3rd Gen AMD Zen CPU for 1.6ExaFLOPs

CPUs: Latency Oriented Design

- High clock frequency
- Large caches

–Convert long latency memory accesses to short latency cache accesses

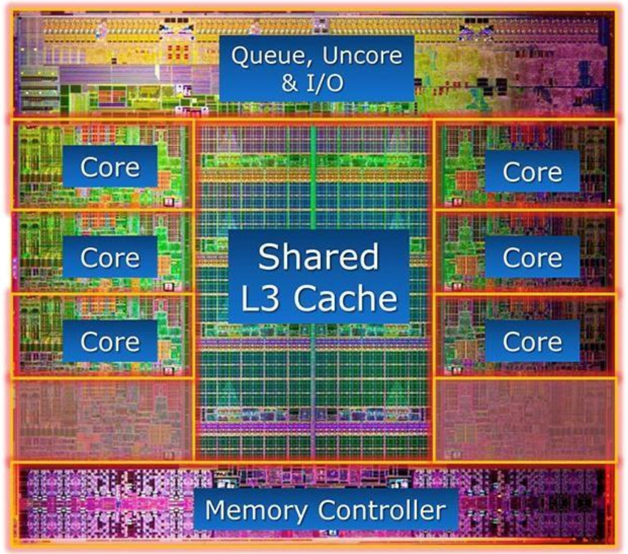
- Sophisticated control

–Branch prediction for reduced branch latency

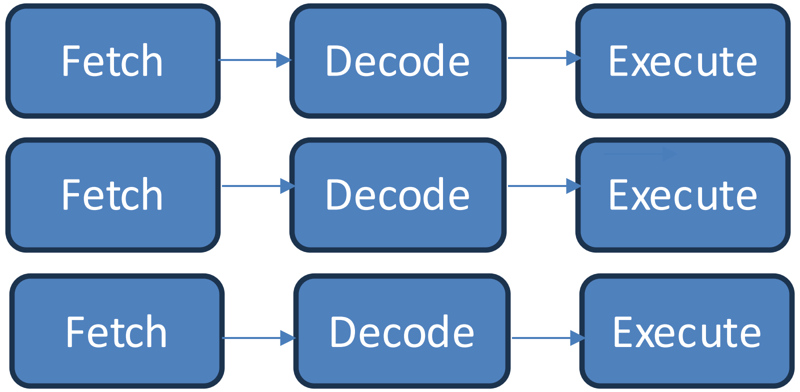
- Powerful ALU

–Reduced operation latency

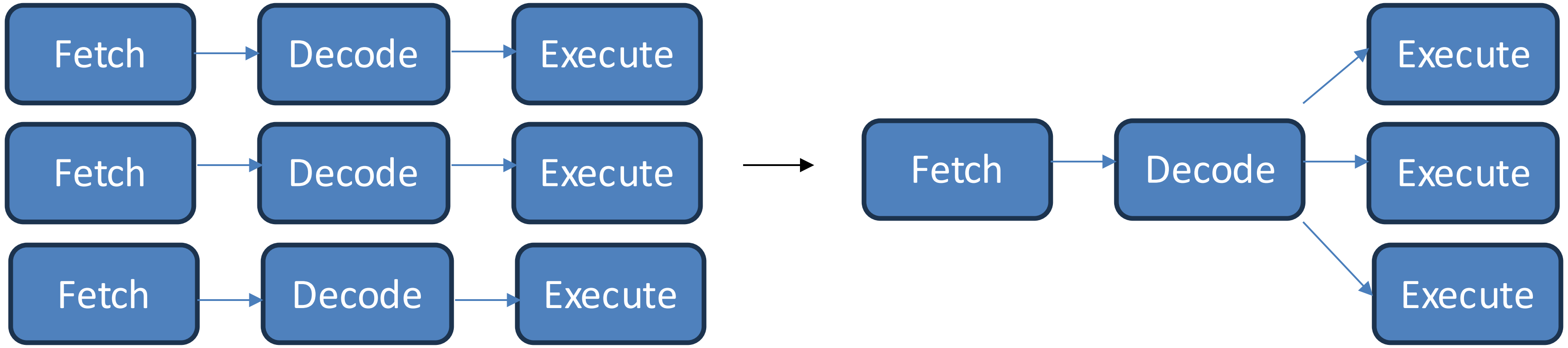
Intel® Core™ i7-3960X Processor Die Detail



CPUs: Instruction is fetched, decoded & executed per data element



Efficiency: Fetch once & decode once



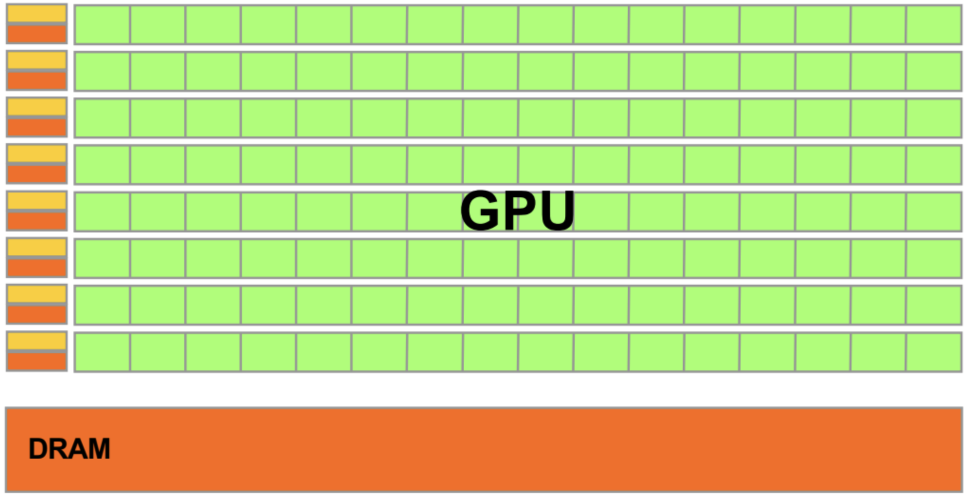
GPUs: Throughput Oriented Design

- Moderate clock frequency

- Small caches
  –To boost memory throughput

- Simple control
  –No branch prediction

- Energy efficient ALUs
  –Many, long latency but
  heavily pipelined for high
  throughput



- Require massive number of threads to tolerate latencies

Credits: Prof. Reetu Das, UM

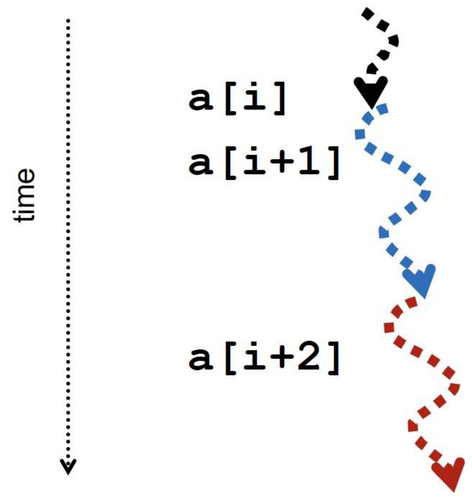
Few more observations on CPU vs GPU

- Common fetch & decode for multiple data elements saves chip area & energy
- GPU can do zero overhead context-switching.
- In a CPU, instructions executed in a pipeline come from same thread.
    - In a GPU, instructions executed at different stages of the pipeline originate from different wavefronts.
- GPUs are a more energy-efficient solution if we can completely utilize the GPUs

Sequential Execution Model

- One instruction at the time
- Optimizations possible at the machine level

```
int a[N]; // a is image, N is large
for (i =0; i < N; i++)
{
   a[i] = a[i] * fade;
}
```



Parallel Execution Model: SIMD

- Example: image fading
  - Some instructions executed concurrently

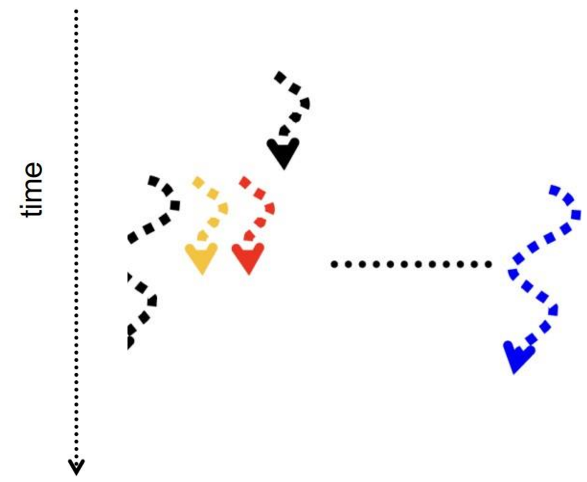
```
int a[N]; // N is large

for all elements do in parallel
{ a[i] = a[i] * fade;
}
```

This has been tried before: ILLIAC III, UIUC, 1966
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4038028&tag=1
http://ed-thelen.org/comp-hist/vs-illiac-iv.html

Parallel Execution Model: SPMD

- Programming Model
- Procedures can synchronize at certain points in program, e.g. barriers
- Each program/procedure 1) works on different data, 2) can execute a different control-flow path, at run-time

- Example: image fading
  - The code is identical across all threads
  - The execution path may differ



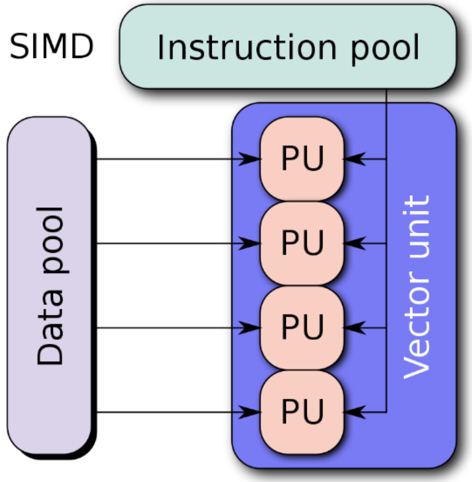

```
int a[N]; // N is large

for all elements do in parallel
{   if (a[i] > threshold) a[i]*= fade;
}
```

Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
  - Sequential instruction execution; lock-step operations in a SIMD instruction
  - Programming model is SIMD (no extra threads) □ SW needs to know vector length
  - ISA contains vector/SIMD instructions

- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (i.e., placed in a different warp) □ programming model not SIMD
    - SW does not need to know vector length
    - Enables multithreading and flexible dynamic grouping of threads
  - ISA is scalar □ SIMD operations can be formed dynamically
  - Essentially, it is SPMD programming model implemented on SIMD hardware

SIMD vs SIMT



- SIMD: A single sequential stream of SIMD vector instructions
  - each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

- SIMT/ warp-based SIMD: Multiple instruction streams of scalar instructions
  - threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

SIMD vs. SIMT Execution Model

- SIMD: A single sequential instruction stream of SIMD instructions
  - each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN
- SIMT: Multiple instruction streams of scalar instructions
  - threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads
- Two Major SIMT Advantages:
  - Can treat each thread separately i.e., can execute each thread independently (on any type of scalar pipeline)
  - Can group threads into warps flexibly
    - i.e., can group threads that are supposed to truly execute the same instruction
    - dynamically obtain and maximize benefits of SIMD processing

Sample GPU SIMT Code (Simplified)

CPU code

```
for (ii = 0; ii < 100000; ++ii) {
C[ii] = A[ii] + B[ii];
}
```

HIP code

```
// there are 100000 threads
__global__void KernelFunction(…) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  int varA = aa[tid];
  int varB = bb[tid];
  C[tid] = varA + varB;
}
```


Slide credit: Hyesoon Kim

Warps *not* Exposed to GPU Programmers

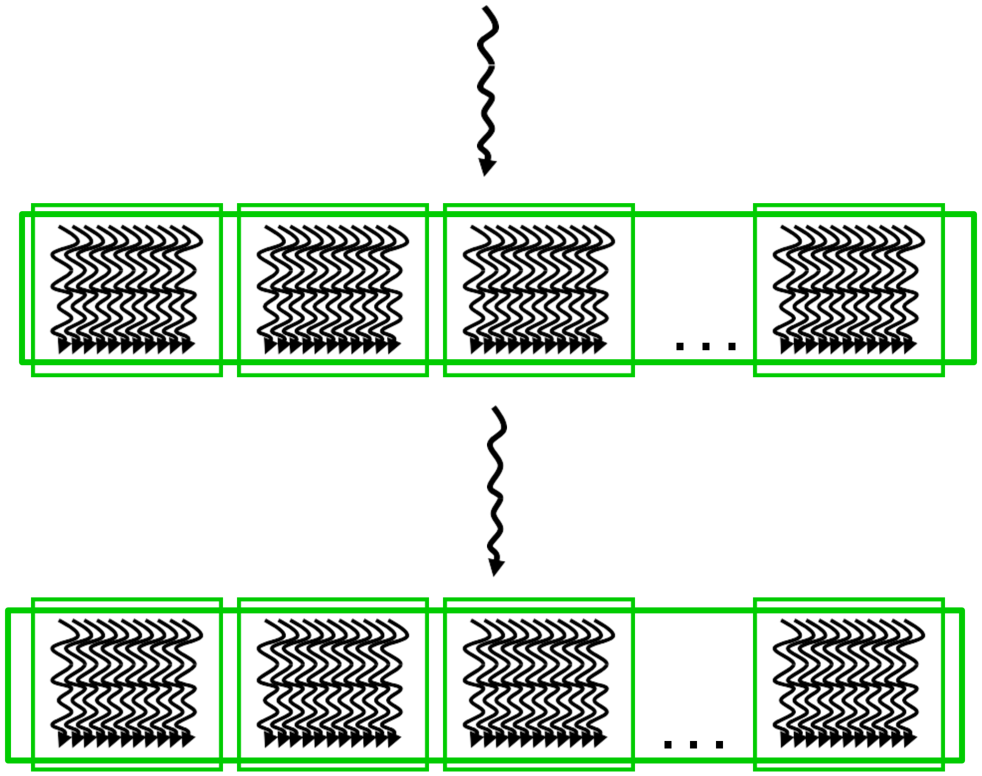
- CPU threads and GPU kernels
  - Sequential or modestly parallel sections on CPU
  - Massively parallel sections on GPU: Blocks of threads

Serial Code (host)

Parallel Kernel (device)
KernelA<<<nBlk, nThr>>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<<nBlk, nThr>>>(args);

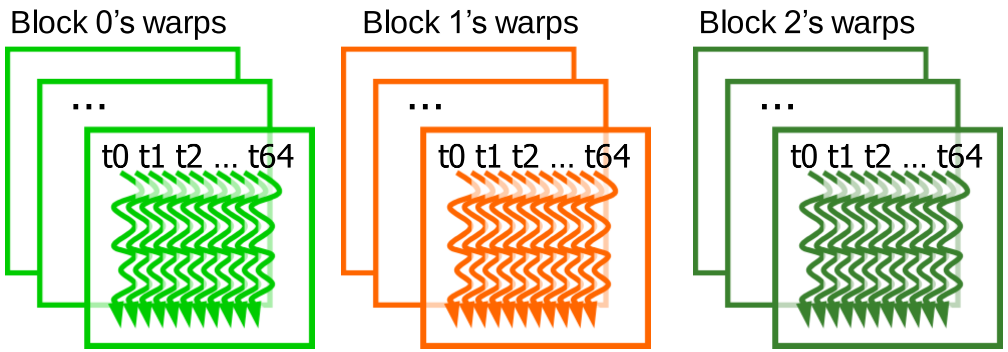


Slide credit: Hwu & Kirk

From Blocks/Wavegroups to Warps/Wavefronts

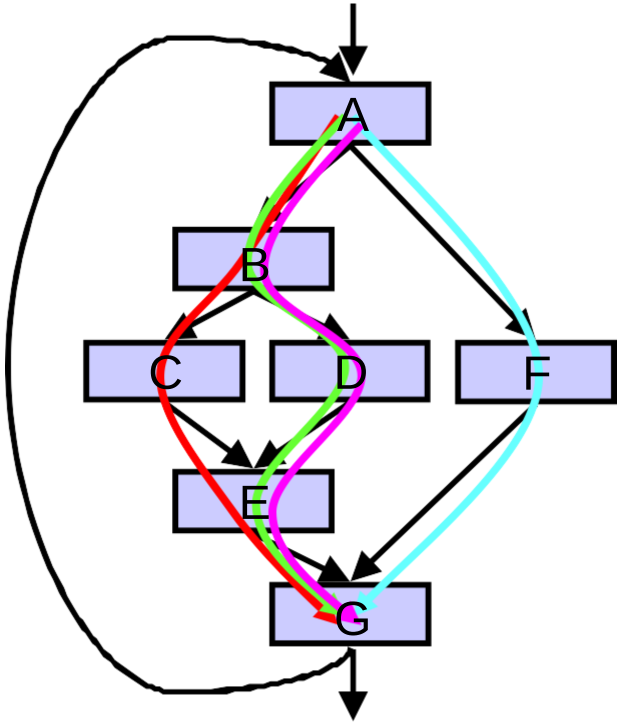
- GPU cores: SIMD pipelines
  - 16-lane wide Compute Units

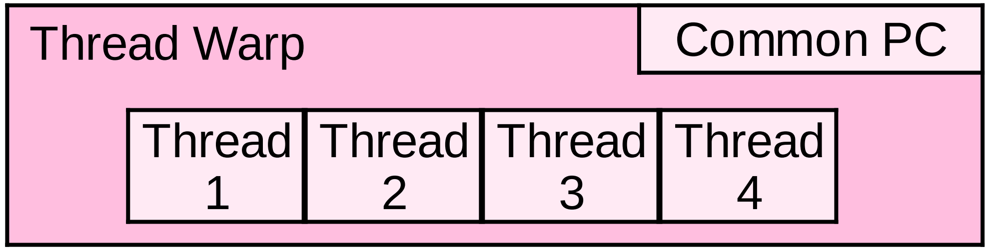
- Workgroups/Blocks are divided into wavefronts/warps
  - SIMD unit (64 threads)



Threads Can Take Different Paths in Warp-based SIMD

- Each thread can have conditional control flow instructions
- Threads can execute different control flow paths

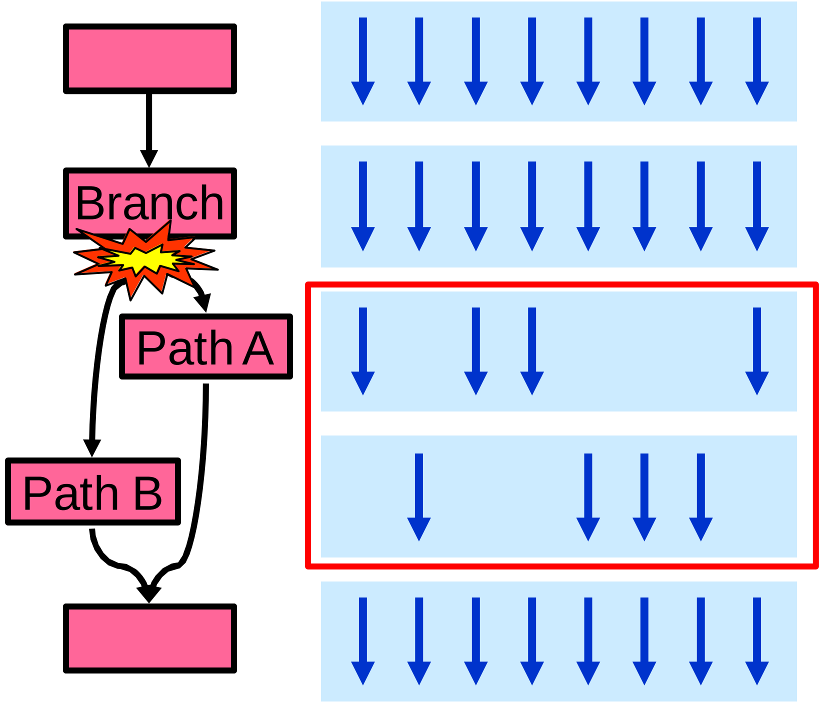




Slide credit: Tor Aamodt

Control Flow Problem in GPUs/SIMT

- A GPU uses a SIMD pipeline to save area on control logic
  - Groups scalar threads into warps/wavefronts
- Branch divergence occurs when threads inside warps branch to different execution paths



Applications optimal for CPU and GPU

Strategies: Use Both CPU and GPU

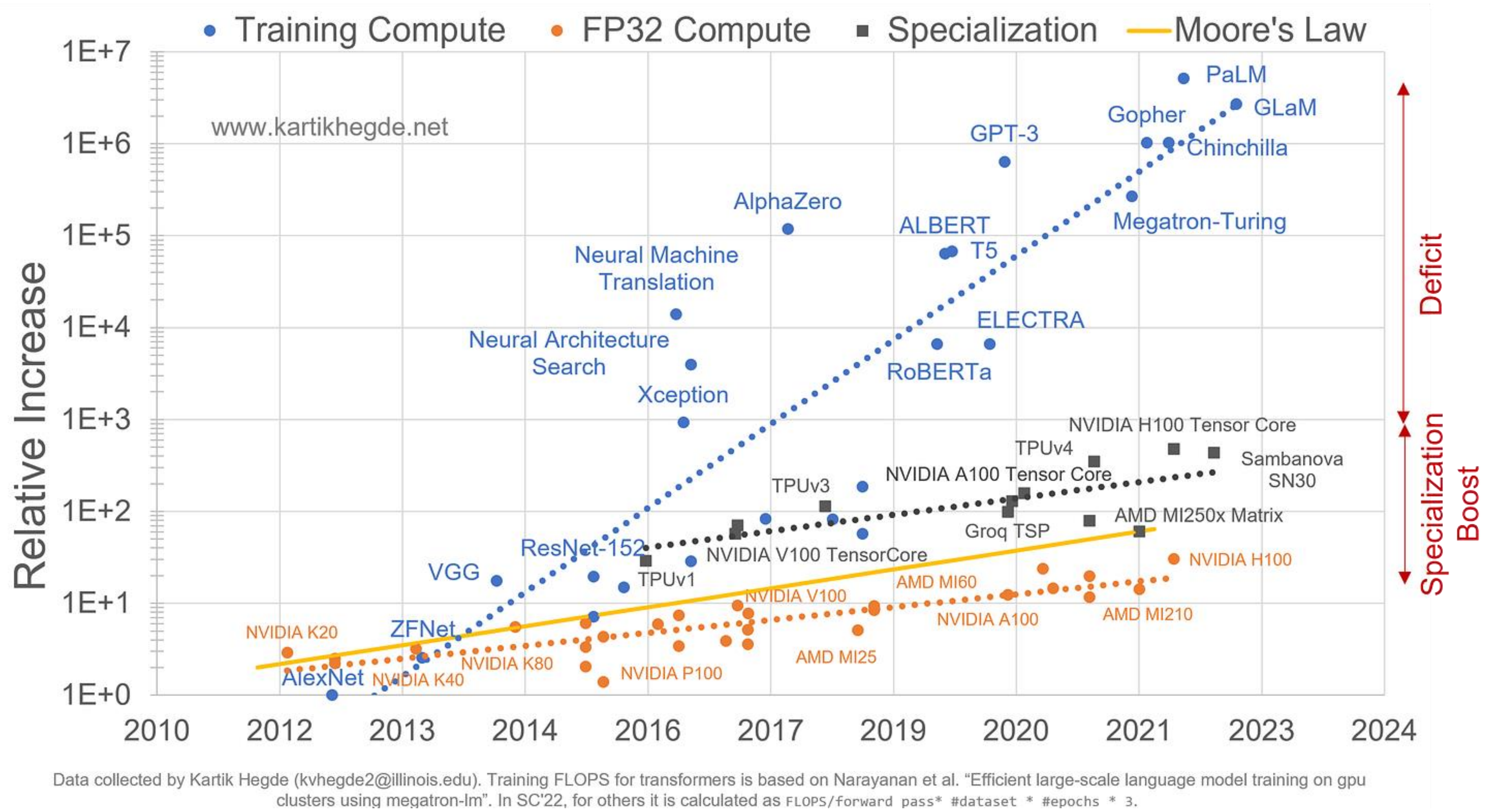
- CPUs for sequential parts where latency matters
  – CPUs can be 10+X faster than GPUs for sequential code

- GPUs for parallel parts where throughput wins
  – GPUs can be 10+X faster than CPUs for parallel code

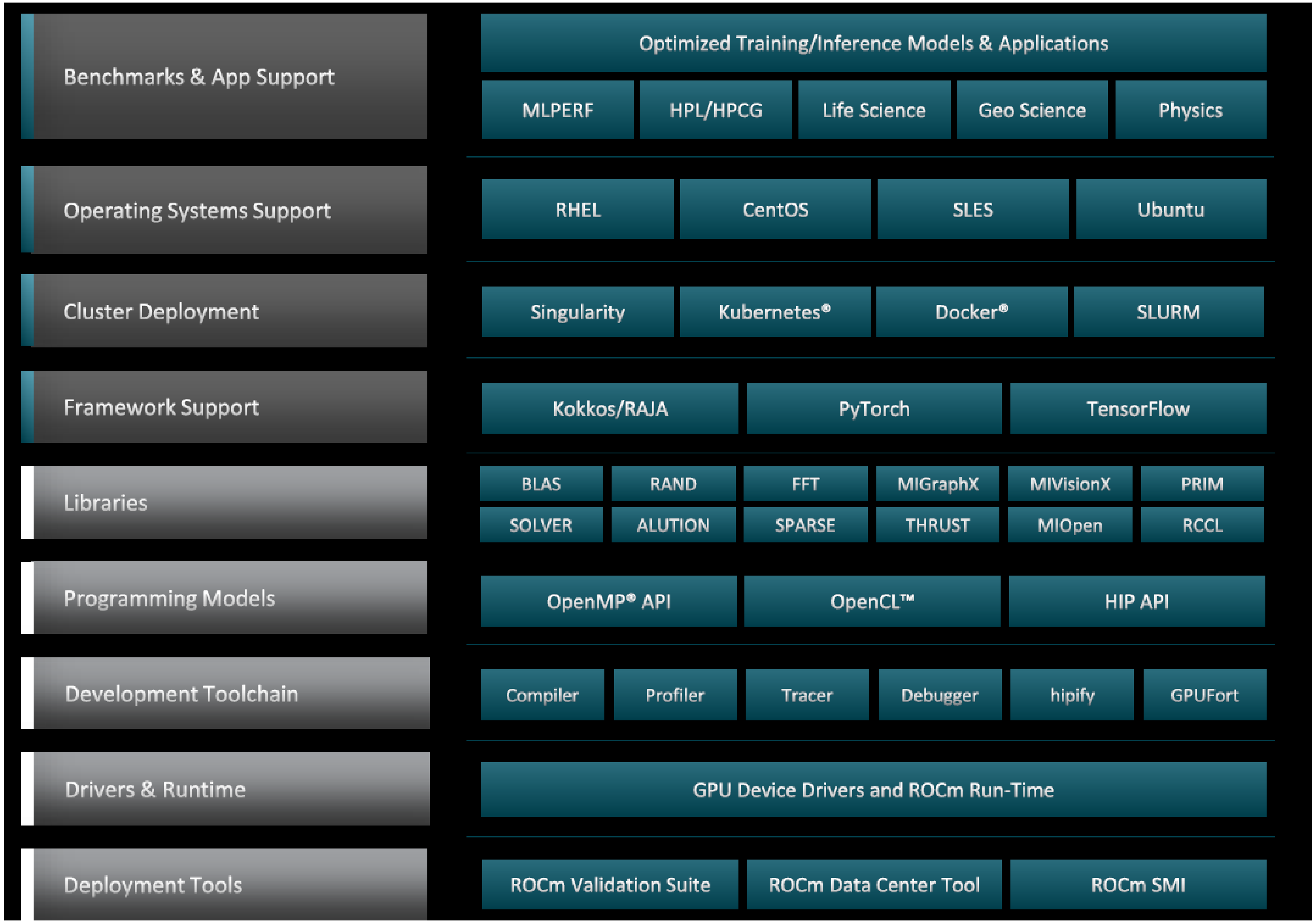
Parallel Programming Workflow

- Identify compute-intensive parts of an application
- Adopt scalable algorithms
- Optimize data arrangements to maximize locality
- Performance Tuning
- Pay attention to code portability, scalability, and maintainability

Improve HW & SW to meet growing compute demand



ROCm SW stack for HPC & AI workloads

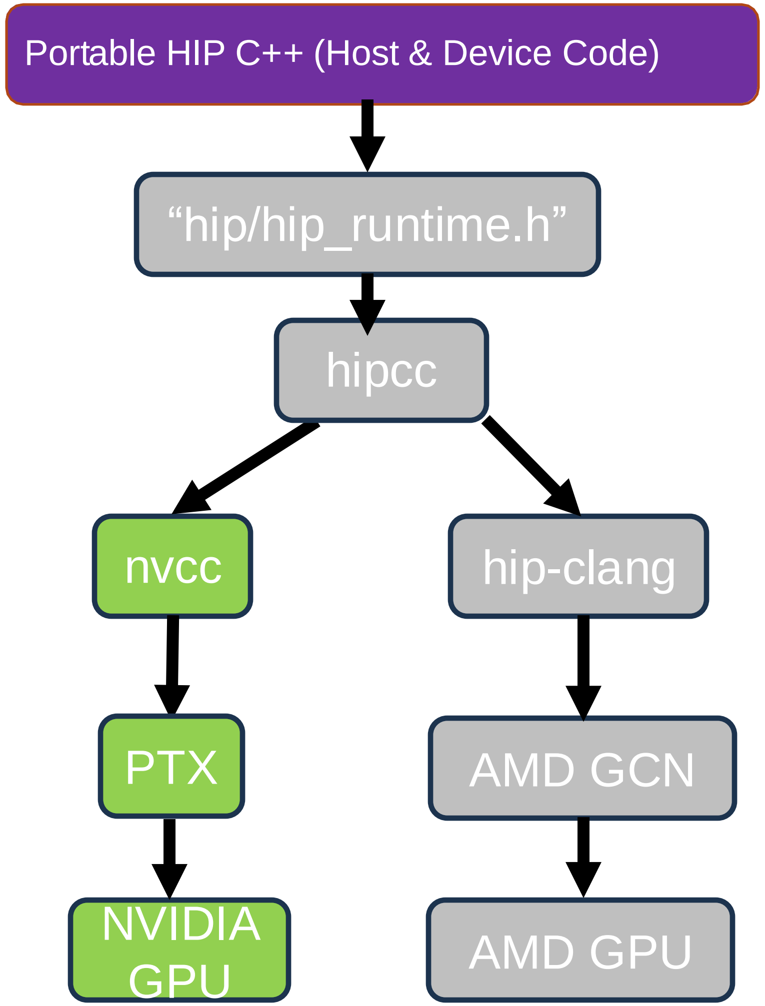


AMD GPU Programming Concepts

Programming with HIP:
Kernels, blocks, threads, and more

What is AMD's HIP?

Heterogeneous-compute **I**nterface for **P**ortability, or **HIP**, is a C++ runtime API & kernel language for AMD & NVIDIA devices.



HIP:

- Open-source.
- Provides an API for an application to leverage GPU acceleration for both AMD and CUDA devices.
- Syntactically similar to CUDA. Most CUDA API calls can be converted in place: cuda -> hip
- Supports a strong subset of CUDA runtime functionality.

A Tale of Host and Device

Source code in HIP has two flavors: Host code and Device code

- The Host is the CPU
- Host code runs here
- Usual C++ syntax and features
- Entry point is the ‘main’ function
- HIP API can be used to create device buffers, move between host and device, and launch device code.

- The Device is the GPU
- Device code runs here
- C-like syntax
- Device codes are launched via “kernels”
- Instructions from the Host are enqueued into “streams”

HIP API

- Device Management:
  - hipSetDevice(), hipGetDevice(), hipGetDeviceProperties()
- Memory Management
  - hipMalloc(), hipMemcpy(), hipMemcpyAsync(), hipFree()
- Streams
  - hipStreamCreate(), hipSynchronize(), hipStreamSynchronize(), hipStreamFree()
- Events
  - hipEventCreate(), hipEventRecord(), hipStreamWaitEvent(), hipEventElapsedTime()
- Device Kernels
  - __global__,__device__, hipLaunchKernelGGL()
- Device code
  - threadIdx, blockIdx, blockDim,__shared__
  - 200+ math functions covering entire CUDA math library.
- Error handling
  - hipGetLastError(), hipGetErrorString()

Kernels, memory, and structure of host code

Device Kernels: The Grid

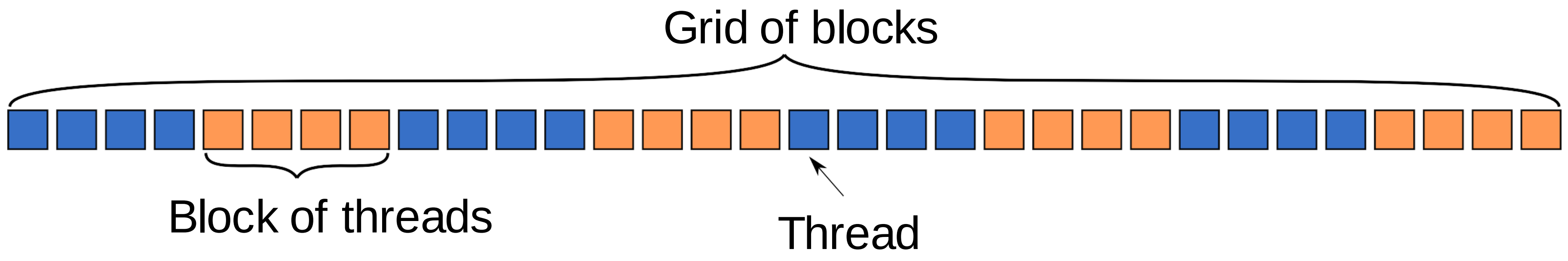
- In HIP, kernels are executed on a 3D ”grid”
- The “grid” is what you can map your problem to
  - It’s not a physical thing, but it can be useful to think that way
- AMD devices (GPUs) support 1D, 2D, and 3D grids,
  but most work maps well to 1D
- Each dimension of the grid partitioned into equal sized “blocks”
- Each block is made up of multiple “threads”
- The grid and its associated blocks are just organizational constructs
  - The threads are the things that do the work
- If you’re familiar with CUDA already, the grid+block structure is very similar in HIP

Device Kernels: The Grid

Some Terminology:

CUDA	HIP	OpenCL™
grid	grid	NDRange
block	workgroup	work group
warp	wavefront	sub-group
thread	work item / thread	work item

The Grid: blocks of threads in 1D

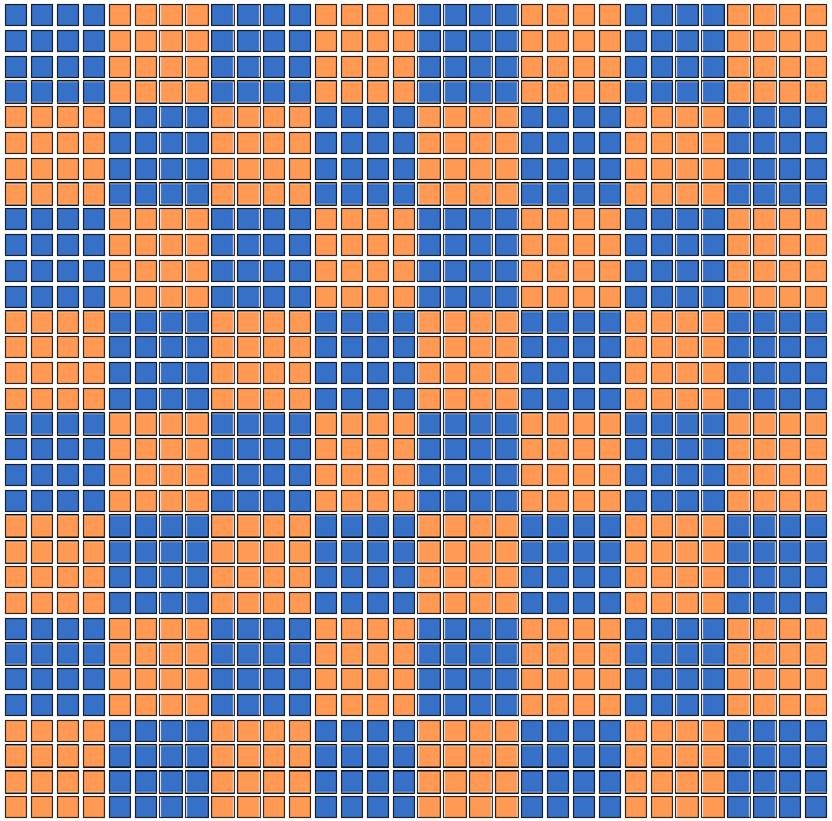


Threads in grid have access to:

- Their respective block: blockIdx.x
- Their respective thread ID in a block: threadIdx.x
- Their block's dimension: blockDim.x
- The number of blocks in the grid: gridDim.x

The Grid: blocks of threads in 2D

- Each color is a block of threads
- Each small square is a thread
- The concept is the same in 1D and 2D
- In 2D each block and thread now has a two-dimensional index



Threads in grid have access to:

- Their respective block IDs: `blockIdx.x, blockIdx.y`
- Their respective thread IDs in a block: `threadIdx.x, threadIdx.y`

Kernel Properties

A simple embarrassingly parallel loop:

```
for (int i=0;i<N;i++)
   h_a[i] *= 2.0;
```

Can be translated into a GPU kernel:

```
__global__ void myKernel(int N, double *d_a)
{
int i = threadIdx.x + blockIdx.x*blockDim.x;
if (i<N)  d_a[i] *= 2.0;
}
```

- A device function that will be launched from the host program is called a kernel & is declared with the __global__ attribute
- Kernels should be declared void
- All pointers passed to kernels must point to memory on the device
- All threads execute the kernel's body "simultaneously"
- Each thread uses its unique thread and block IDs to compute a global ID
- There could be more than N threads in the grid

Kernel launch

Kernels are launched from the host:

```
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,            //Kernel name (_global_void function)
                   blocks,              //Grid dimensions
                   threads,             //Block dimensions
                   0,                   //Bytes of dynamic LDS space
                   0,                   //Stream (0=NULL stream)
                   N, a);               //Kernel arguments
```

Analogous to CUDA kernel launch syntax:

```
myKernel<<<blocks, threads, 0, 0>>>(N,a);
```

SIMD operations

Why blocks and threads?

Natural mapping of kernels to hardware:

- Blocks are dynamically scheduled onto CUs
- All threads in a block execute on the same CU
- Threads in a block share LDS memory and L1 cache
- Threads in a block are executed in **64-wide** chunks called "wavefronts"
- Wavefronts execute on SIMD units (Single Instruction Multiple Data)
- If a wavefront stalls (e.g. data dependency) CUs can quickly context switch to another wavefront

A good practice is to make the block size a multiple of 64 and have several wavefronts (e.g. 256 threads)

Device Memory

The host instructs the device to allocate device memory and records a pointer to device memory:

```
int main() {
    …
    int N = 1000;
    size_t Nbytes = N*sizeof(double);
    double *h_a = (double*) malloc(Nbytes);          //Host memory

    double *d_a = NULL;
    hipMalloc(&d_a, Nbytes);                         //Allocate Nbytes on device

    …

    free(h_a);                                       //free host memory
    hipFree(d_a);                                    //free device memory
}
```

Device Memory

The host queues memory transfers:

```
//copy data from host to device
hipMemcpy(d_a, h_a, Nbytes, hipMemcpyHostToDevice);

//copy data from device to host
hipMemcpy(h_a, d_a, Nbytes, hipMemcpyDeviceToHost);

//copy data from one device buffer to another
hipMemcpy(d_b, d_a, Nbytes, hipMemcpyDeviceToDevice);
```

Putting it all together

```
#include "hip/hip_runtime.h"
int main() {  int N =
   1000;
   size_t Nbytes = N*sizeof(double);
   double *h_a = (double*) malloc(Nbytes);             //host memory
   double *d_a = NULL;
   HIP_CHECK(hipMalloc(&d_a, Nbytes));
   …
   HIP_CHECK(hipMemcpy(d_a, h_a, Nbytes, hipMemcpyHostToDevice));          //copy data to device

   hipLaunchKernelGGL(myKernel, dim3((N+256-1)/256,1,1), dim3(256,1,1), 0, 0, N, d_a); //Launch kernel
   HIP_CHECK(hipGetLastError());

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

```
_global_____void myKernel(int N, double *d_a) {
   int i = threadIdx.x + blockIdx.x*blockDim.x; if (i<N) {
      d_a[i] *= 2.0;
   }
}
```

```
#define HIP_CHECK(command) {                                  \
   hipError_t status = command; if                            \
   (status!=hipSuccess) {                                     \
      std::cerr << "Error: HIP reports "                      \
                    << hipGetErrorString(status) \
                    << std::endl;                              \
      std::abort(); } }
```

Vector Addition

```
__global__void vecAddkernel(float* A_d, float* B_d, float* C_d, int n) {

    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i < n)  {
        C_d[i] = A_d[i] + B_d[i];
    }
}
```

Vector Addition

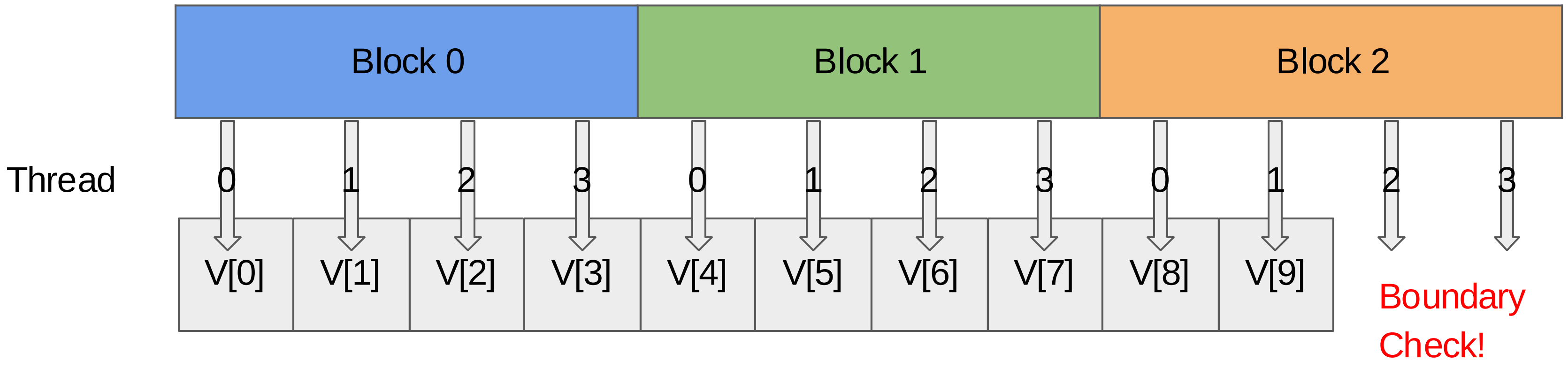
```
void vecAdd(float* A, float* B, float* C, int n) {
        int size = n * sizeof(float);
        float* A_d, B_d, C_d;

        // Allocate device memory
        hipMalloc((void **) &A_d, size);
        hipMalloc((void **) &B_d, size);
        hipMalloc((void **) &C_d, size);

        // Transfer A and B to device memory
        hipMemcpy(A_d, A, size, hipMemcpyHostToDevice);
        hipMemcpy(B_d, B, size, hipMemcpyHostToDevice);

        // Kernel invocation code
        vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
        hipDeviceSynchronize();

        // Transfer C from device to host
        hipMemcpy(C, C_d, size, hipMemcpyDeviceToHost);

        // Free device memory for A, B, C
        hipFree(A_d); hipFree(B_d); hipFree (C_d);
}
```

Another way to launch kernel

```
int vecAdd(float* A, float* B, float* C, int n) {
    …
    // Kernel invocation code
    dim3 DimGrid(ceil(n/256), 1, 1);
    dim3 DimBlock(256,1,1);

    vecAddKernel<<<DimGrid, DimBlock>>>(A_d, B_d, C_d, n);
    hipDeviceSynchronize();

    // Transfer C from device to host
    ...
}
```

Mapping to the data

Suppose we use 1d thread blocks of size 4



i = blockIdx.x * blockDim.x + threadIdx.x

Questions

1. How many floating operations are being performed in the vector add kernel? Give your answer in terms of N and explain.

2. How many global memory reads and writes are being performed by the vector add kernel? Give your answer in terms of N.

Questions

1. How many floating operations are being performed in the vector add kernel? Give your answer in terms of N and explain.

   N, one for each pair of input vector elements

2. How many global memory reads and writes are being performed by the vector add kernel? Give your answer in terms of N.

   Reads: 2N, one for each of the two input vectors elements.

   Writes: N, one for each output vector element.

Function Qualifiers

hipcc makes two compilation passes through source code. One to compile host code, and one to compile device code.

- __global__ functions:
  - These are entry points to device code, called from the host
  - Code in these regions will execute on SIMD units
- __device__ functions:
  - Can be called from __global__ and other __device__ functions.
  - Cannot be called from host code.
  - Not compiled into host code – essentially ignored during host compilation pass
- __host__ __device__ functions :
  - Can be called from __global__, __device__ , and host functions.
  - Will execute on SIMD units when called from device code!

Memory declarations in Device Code

- Malloc/free not supported in device code.
- Variables/arrays can be declared on the stack.
- Stack variables declared in device code are allocated in registers and are private to each thread.
- Threads can all access common memory via device pointers, but otherwise do not share memory.
  - Important exception: __shared__ memory
- Stack variables declared as __shared__ :
  - Allocated once per block in LDS memory
  - Shared and accessible by all threads in the same block
  - Access is faster than device global memory (but slower than register)
  - Must have size known at compile time

Thread Synchronization

- _syncthreads():
  - Blocks a wavefront from continuing execution until all wavefronts have reached __syncthreads()
  - Memory transactions made by a thread before __syncthreads() are visible to all other threads in the block after __syncthreads()
  - Can have a noticeable overhead if called repeatedly

Shared Memory

```
_global__    void reverse(double *d_a) {
__shared __   double s_a[256]; //array of doubles, shared in this block

    int tid = threadIdx.x;
    s_a[tid] = d_a[tid];                    //each thread fills one entry

    //all wavefronts must reach this point before any wavefront is allowed to continue.
                                            //something is missing here…
_syncthreads();

    d_a[tid] = s_a[255-tid]; //write out array in reverse order
}

int main() {
    …
    hipLaunchKernelGGL(reverse, dim3(1), dim3(256), 0, 0, d_a); //Launch kernel
    …
}
```

Device Management

Multiple GPUs in system? Multiple host threads/MPI ranks? What device are we running on?

- Host can query number of devices visible to system:

```
int numDevices = 0;  hipGetDeviceCount(&numDevices);
```

- Host tells the runtime to issue instructions to a particular device:

```
int deviceID = 0;  hipSetDevice(deviceID);
```

- Host can query what device is currently selected:

```
hipGetDevice(&deviceID);
```

- The host can manage several devices by swapping the currently selected device during runtime.
- MPI ranks can set different devices or over-subscribe (share) devices.

Device Properties during run-time

The host can also query a device's properties:

```
hipDeviceProp_t props;
hipGetDeviceProperties(&props, deviceID);
```

- hipDeviceProp_t is a struct that contains useful fields like the device's name, warpsize, max workgroup size, max grid size, #CUs, cache, clock speed, and GCN arch architecture.
  - See "hip/hip_runtime_api.h" for full list of fields.

Querying System from terminal

- rocminfo: Queries and displays information on the system's hardware
  - More info at: https://github.com/RadeonOpenCompute/rocminfo
- Querying ROCm version:
  - If you install ROCm in the standard location (/opt/rocm) version info is at: /opt/rocm/.info/version-dev
  - Can also run the command 'dkms status' and the ROCm version will be displayed
- rocm-smi: Queries and sets AMD GPU frequencies, power usage, and fan speeds
  - sudo privileges are needed to set frequencies and power limits
  - sudo privileges are not needed to query information
  - Get more info by running 'rocm-smi -h' or looking at: https://github.com/RadeonOpenCompute/ROC-smi

```
$ /opt/rocm/bin/rocm-smi
========================ROCm System Management Interface========================
================================================================================
GPU  Temp   AvgPwr  SCLK     MCLK    Fan   Perf    PwrCap  VRAM%  GPU%
1    38.0c  18.0W   1440Mhz  945Mhz  0.0%  manual  220.0W    0%   0%
================================================================================
==============================End of ROCm SMI Log ==============================
```

Blocking vs Nonblocking API functions

- The kernel launch function, hipLaunchKernelGGL, is **non-blocking** for the host.
  - After sending instructions/data, the host continues immediately while the device executes the kernel
  - If you know the kernel will take some time, this is a good area to do some work (i.e. MPI comms) on the host

- However, hipMemcpyis **blocking**.
  - The data pointed to in the arguments can be accessed/modified after the function returns.

- The non-blocking version is hipMemcpyAsync

```
hipMemcpyAsync(d_a, h_a, Nbytes, hipMemcpyHostToDevice, stream);
```

- Like hipLaunchKernelGGL, this function takes an argument of type hipStream_t

- It is not safe to access/modify the arguments of hipMemcpyAsyncwithout some sort of synchronization.

Streams

- A stream in HIP is a queue of tasks (e.g. kernels, memcpys, events).
  - Tasks enqueued in a stream **complete in order on that stream**.
  - Tasks being executed in different streams are allowed to overlap and share device resources.

- Streams are created via:

  hipStream_t stream;
  hipStreamCreate(&stream);

- And destroyed via:

  hipStreamDestroy(stream);

- Passing 0 or NULL as the hipStream_t argument to a function instructs the function to execute on a stream called the 'NULL Stream':
  - No task on the NULL stream will begin until **all previously enqueued tasks in all other streams have completed**.
  - Blocking calls like hipMemcpy run on the NULL stream.

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:

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: Check that the kernels 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.**

Pinned Memory

Host data allocations are pageable by default. The GPU can directly access Host data if it is pinned instead.

- Allocating pinned host memory:

```
double *h_a = NULL;
hipHostMalloc(&h_a, Nbytes);
```

- Free pinned host memory:

```
hipHostFree(h_a);
```

- Host<->Device memcpy **bandwidth increases significantly when host memory is pinned**.
  - It is good practice to allocate host memory that is frequently transferred to/from the device as pinned memory.

Streams

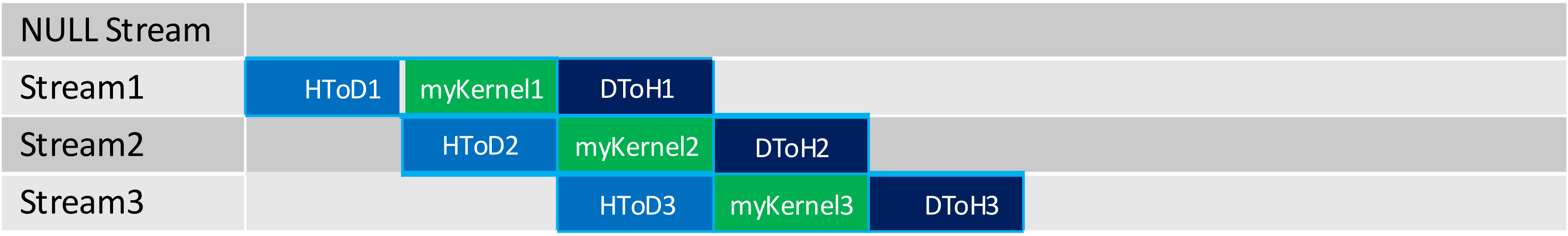
Suppose we have 3 kernels which require moving data to and from the device:

```
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);
```



Synchronization

How do we coordinate execution on device streams with host execution?
Need some synchronization points.

- hipDeviceSynchronize();
  - Heavy-duty sync point.
  - Blocks host until **all work** in **all device streams** has reported complete.

- hipStreamSynchronize(stream);
  - Blocks host until all work in stream has reported complete.

Can a stream synchronize with another stream? For that we need ‘Events’.

Events

A hipEvent_t object is created on a device via:

```
hipEvent_t event;
hipEventCreate(&event);
```

We queue an event into a stream:

```
hipEventRecord(event, stream);
```

- The event records what work is currently enqueued in the stream.
- When the stream’s execution reaches the event, the event is considered ‘complete’.

At the end of the application, event objects should be destroyed:

```
hipEventDestroy(event);
```

Events

What can we do with queued events?

- hipEventSynchronize(event);
  - Block host until event reports complete.
  - Only a synchronization point with respect to the stream where event was enqueued.
- hipEventElapsedTime(&time, startEvent, endEvent);
  - Returns the time in ms between when two events, startEvent and endEvent, completed
  - Can be very useful for timing kernels/memcpys
- hipStreamWaitEvent(stream, event);
  - Non-blocking for host.
  - Instructs all future work submitted to stream to wait until event reports complete.
  - Primary way we enforce an ‘ordering’ between tasks in separate streams.

Questions

1. How would one declare a variable err that can appropriately receive the returned value of a HIP API call?

Questions

1. How would one declare a variable err that can appropriately receive the returned value of a HIP API call?

Ans: hipError_t err = {stmt}; where {stmt} is a HIP API call such as hipMalloc() or a kernel function call

Questions

2. If we want to copy 5000 bytes of data from host array h_A (h_A is a pointer to element 0 of the source array) to device array d_A (d_A is a pointer to element 0 of the destination array), what would be an appropriate API call for the data copy in HIP?

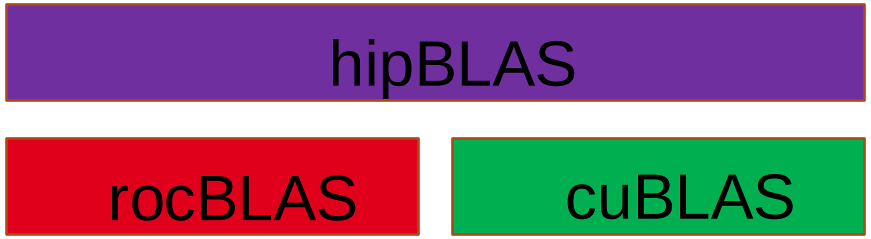
Questions

2. If we want to copy 5000 bytes of data from host array h_A (h_A is a pointer to element 0 of the source array) to device array d_A (d_A is a pointer to element 0 of the destination array), what would be an appropriate API call for the data copy in HIP?

Ans: hipMemcpy(d_A, h_A, 5000, hipMemcpyHostToDevice);

AMD GPU Libraries

- A note on naming conventions:
  - roc* -> AMGCN library usually written in HIP
  - cu* -> NVIDIA PTX libraries
  - hip* -> usually interface layer on top of roc*/cu* backends
- hip* libraries:
  - Can be compiled by hipcc and can generate a call for the device you have:
    - hipcc->hip-clang->AMD GCN ISA
    - hipcc->nvcc (inlined)->NVPTX
  - Just a thin wrapper that marshals calls off to a “backend” library:
    - corresponding roc* library backend containing optimized GCN
    - corresponding cu* library backend containing NVPTX for NVIDIA devices
  - E.g., hipBLAS is a marshalling library:



Why libraries?

- Code reuse
- High Performance
  - Maximize compute
  - Maximize memory bandwidth
- No need to deal with low level GPU code

Math library equivalents

CUBLAS	**ROCBLAS**	Basic Linear Algebra Subroutines
CUFFT	**ROCFFT**	Fast Fourier Transforms
THRUST	**ROCTHRUST**	STL Library for parallel algos
CUB	**ROCPRIM**	Optimized Parallel Primitives
EIGEN	**EIGEN**	C++ Template Library for Linear Algebra

MORE INFO AT: GITHUB.COM/ROCM-DEVELOPER-TOOLS/HIP → HIP_PORTING_GUIDE.MD