Introduction to GPU programming

Felice Pantaleo
Experimental Physics Department, CERN

felice@cern.ch

Content of the theoretical session

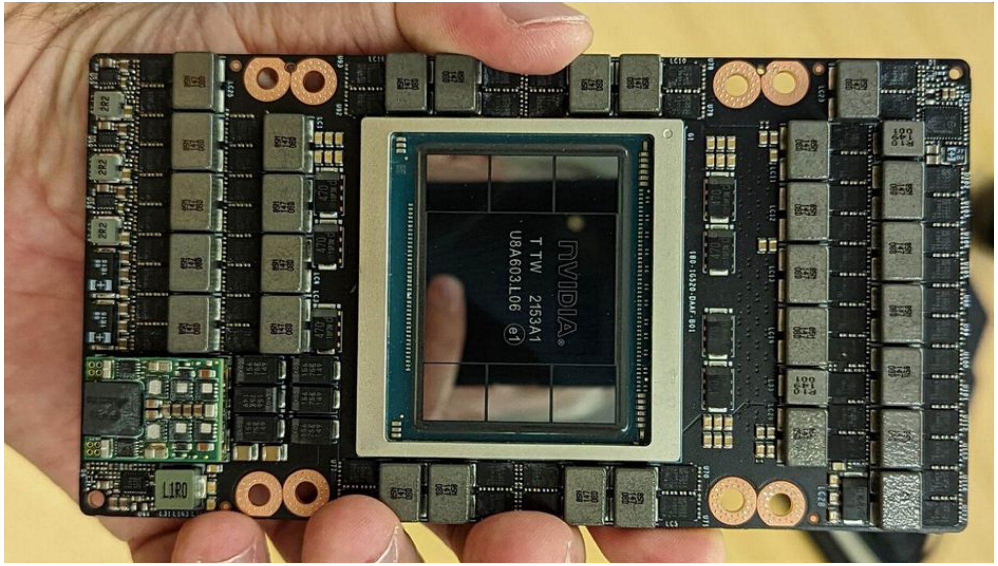
- Heterogeneous Parallel computing systems
- Basics
- Parallel kernels
- Shared Memory and Synchronization
- Device Management

Content of the tutorial session

- Write and launch kernels
- Manage GPU memory
- Manage communication and synchronization

Accelerators

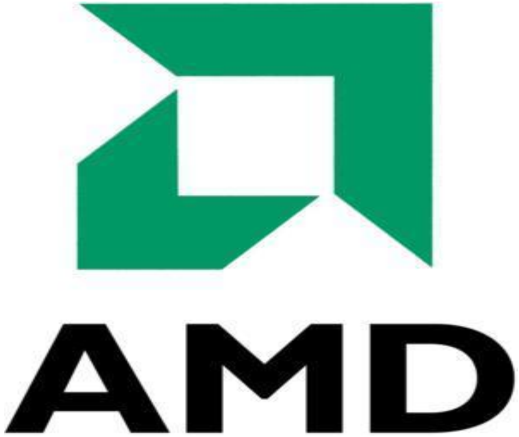
- Exceptional raw power and memory bandwidth wrt CPUs
- Lower energy to solution
- Massively parallel architecture
- Low Memory/core

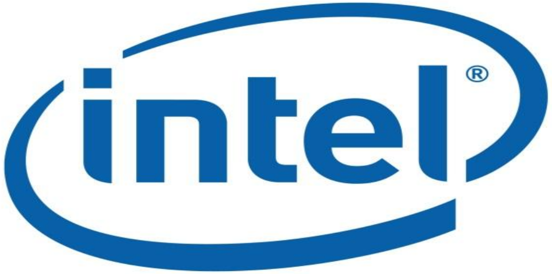


Accelerators

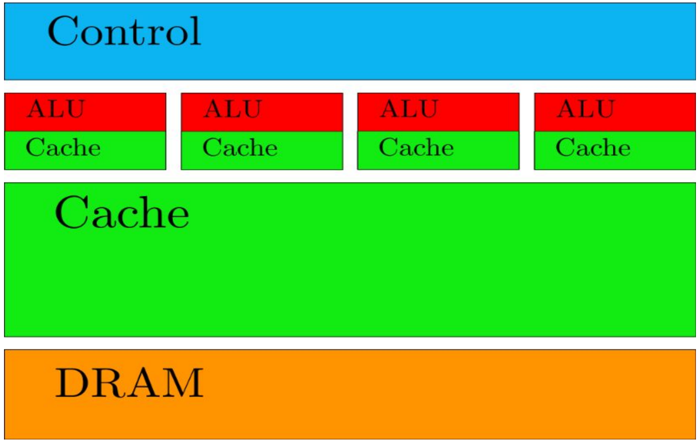
- GPUs were traditionally used for real-time rendering/gaming.

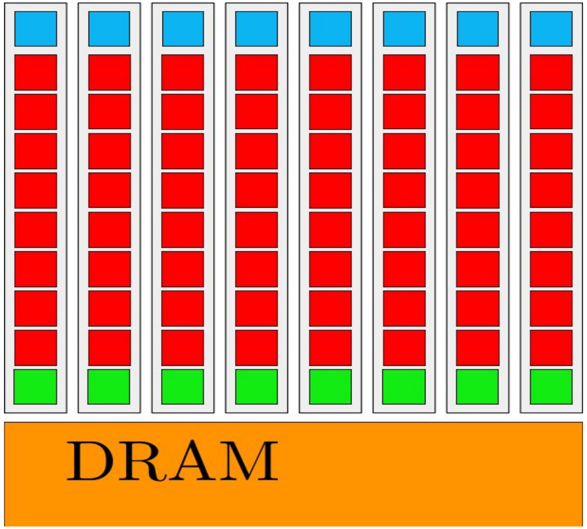
AMD and NVIDIA main manufacturers for discrete GPUs, Intel for integrated ones





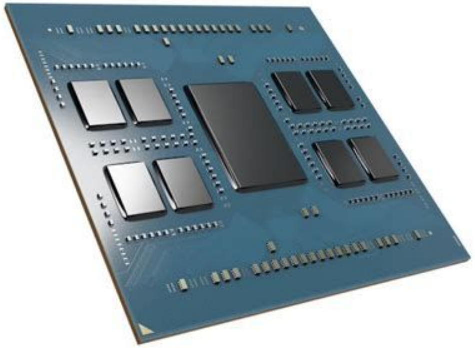
CPU vs GPU architectures

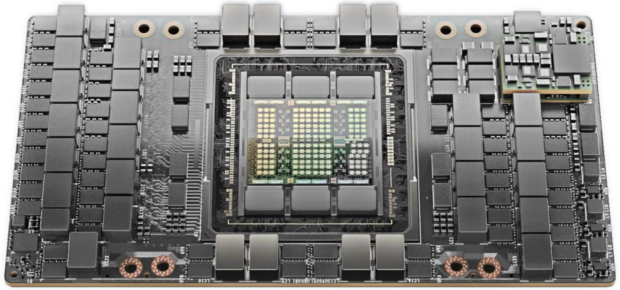




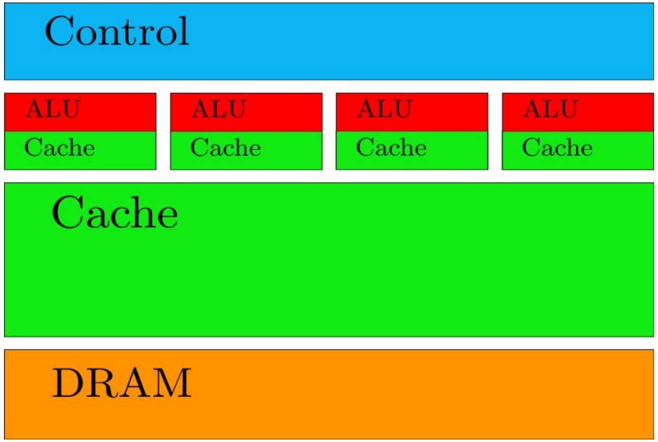
CPU

GPU

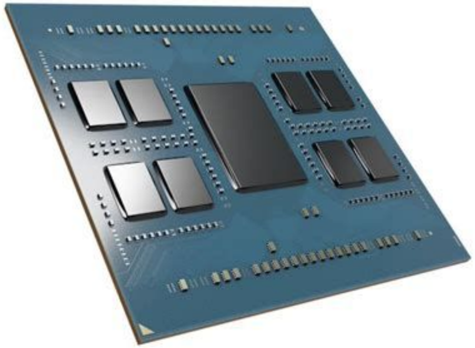




CPU vs GPU architectures



CPU

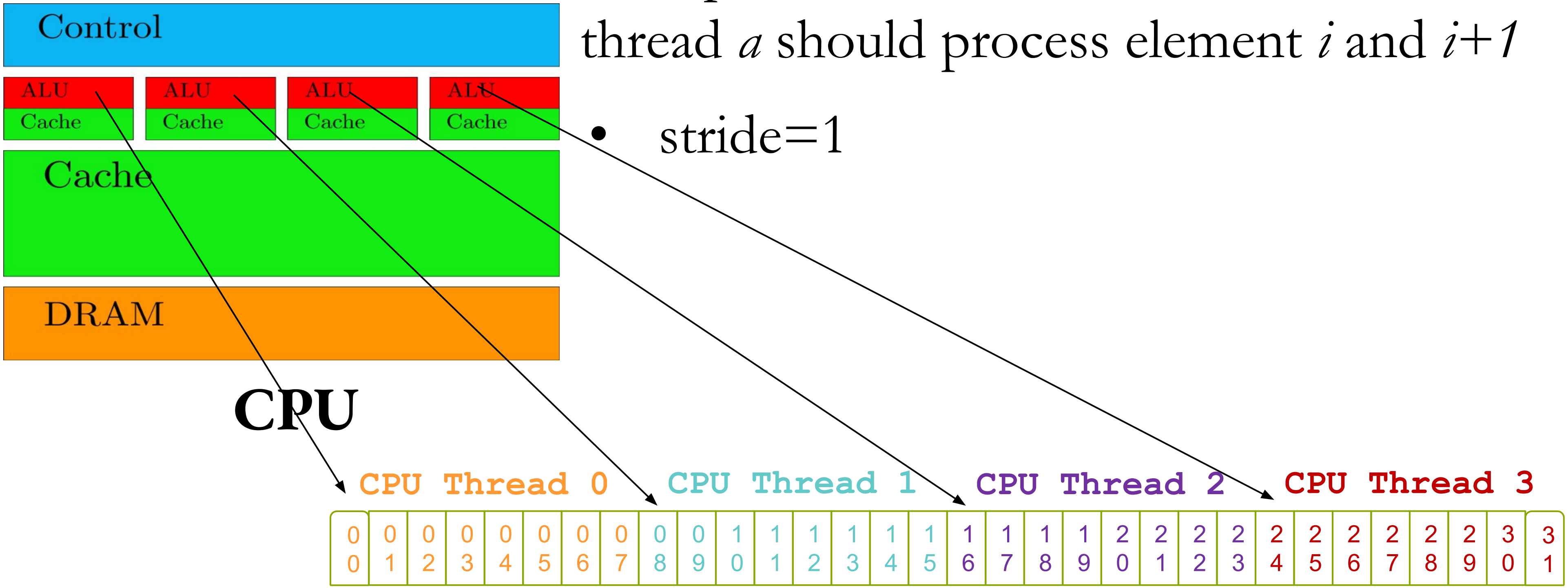


- Large caches (slow memory accesses to quick cache accesses)
- SIMD
- Branch prediction/speculative
- Powerful ALU
- Pipelining

Memory access patterns: cached

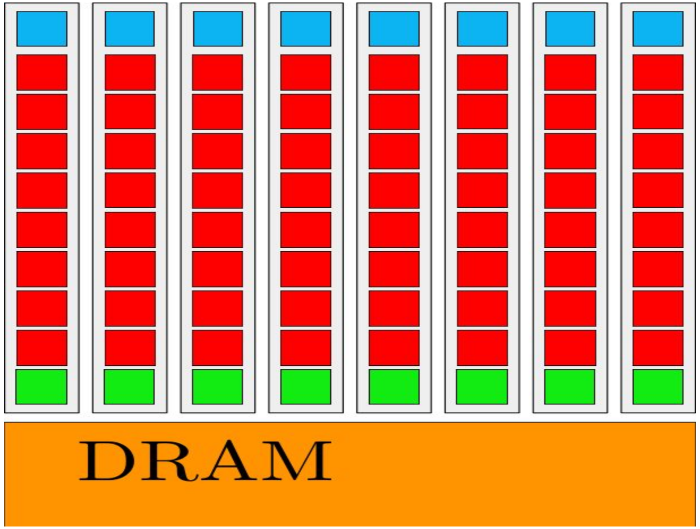
For optimal CPU cache utilization, the thread *a* should process element *i* and *i+1*

- stride=1

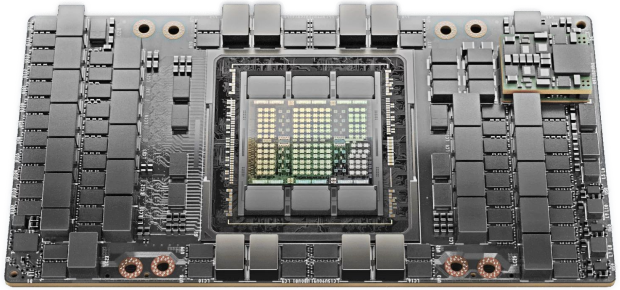


CPU vs GPU architectures

- Hundreds of "cores" (e.g. streaming multiprocessors, Xe cores, compute units)
- SIMT (Single-Instruction, Multiple-Thread) with hundreds of SIMD-like warps in fly
- Instructions pipelined
- Thread-level parallelism
- Instructions issued in order
- Branch predication

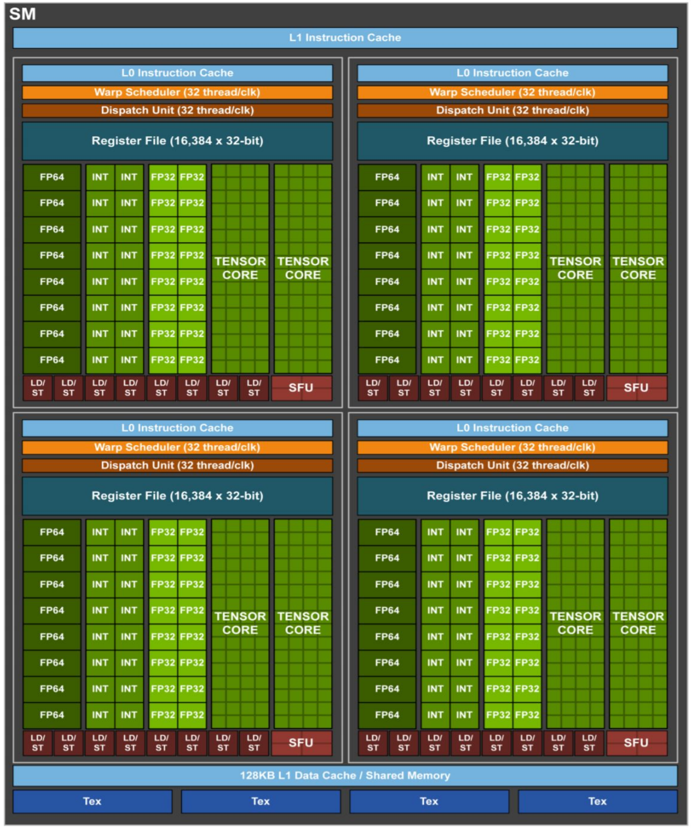


GPU

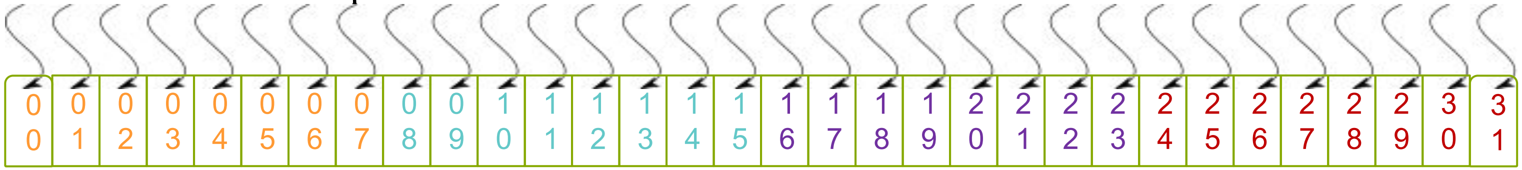


Inside a GPU SM: coalesced

- L1 data cache shared among ALUs
- ALUs work in SIMD mode in groups of 32 (warps)
  - Think about it as vectors on the same CPU core
- If a *load* is issued by each thread, they have to wait for all the loads in the same warp to complete before the next instruction can execute
- Coalesced memory access pattern optimal for GPUs: thread *a* should process element *i*, thread *a+1* the element and *i+1*
  - Lose an order of magnitude in performance if cached access pattern used on GPU

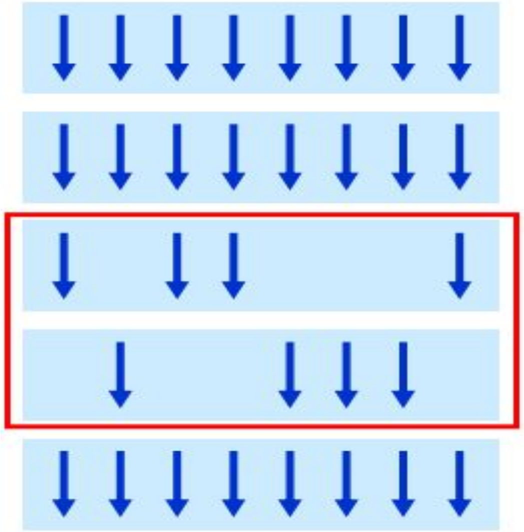


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT INT
FP32 FP32
TENSOR CORE
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
Tex
Tex
Tex



Warps

- Once a block is assigned to an SM, it is divided into units called warps.
- Thread IDs within a warp are consecutive and increasing
- Threads within a warp are executed in a SIMD fashion
- If an operand is not ready the warp will stall
- Context switch between warps when stalled
- Context switch must be very fast
- Typical values of warp size 16, 32, 64 depending on vendor



Throughput

Theoretical peak throughput: the maximum amount of data that a kernel can read and produce in the unit time.

$\text{Throughput}_{\text{peak}}$ (GB/s) = 2 x access width (byte) x mem_freq (GHz)

This means that if your device comes with a memory clock rate of 1GHz DDR (double data rate) and a 384-bit wide memory interface, the amount of data that a kernel can process and produce in the unit time is at most:

$\text{Throughput}_{\text{peak}}$ (GB/s) = 2 x (384/8)(byte) x 1 (GHz)= 96GB/s

Global memory

Volta V100:

- 7.8 TFLOPS DPFP peak throughput
- 900 GB/s peak off-chip memory access bandwidth
- 112 G DPFP operands per second
- To achieve peak throughput, a program must perform 7800/112 = ~70 FP arithmetic operations for each operand value fetched from off-chip memory

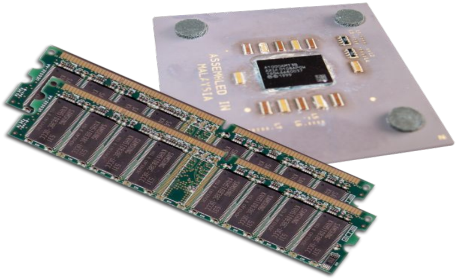
Bandwidth

Bandwidth

Heterogeneous Parallel Computing Systems

Heterogeneous Computing

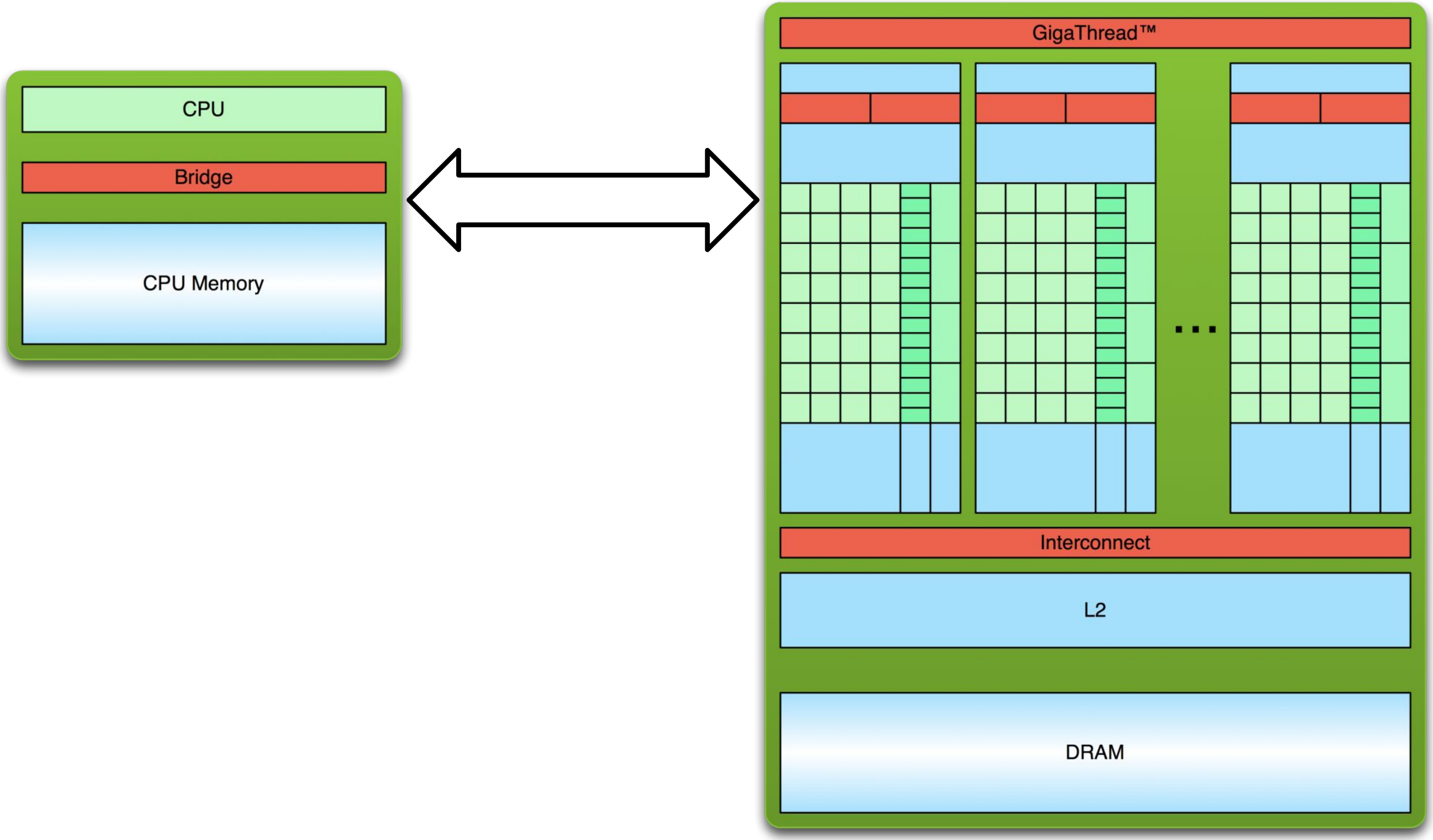
- Terminology
  - Host The CPU and its memory space
  - Device The GPU and its memory space



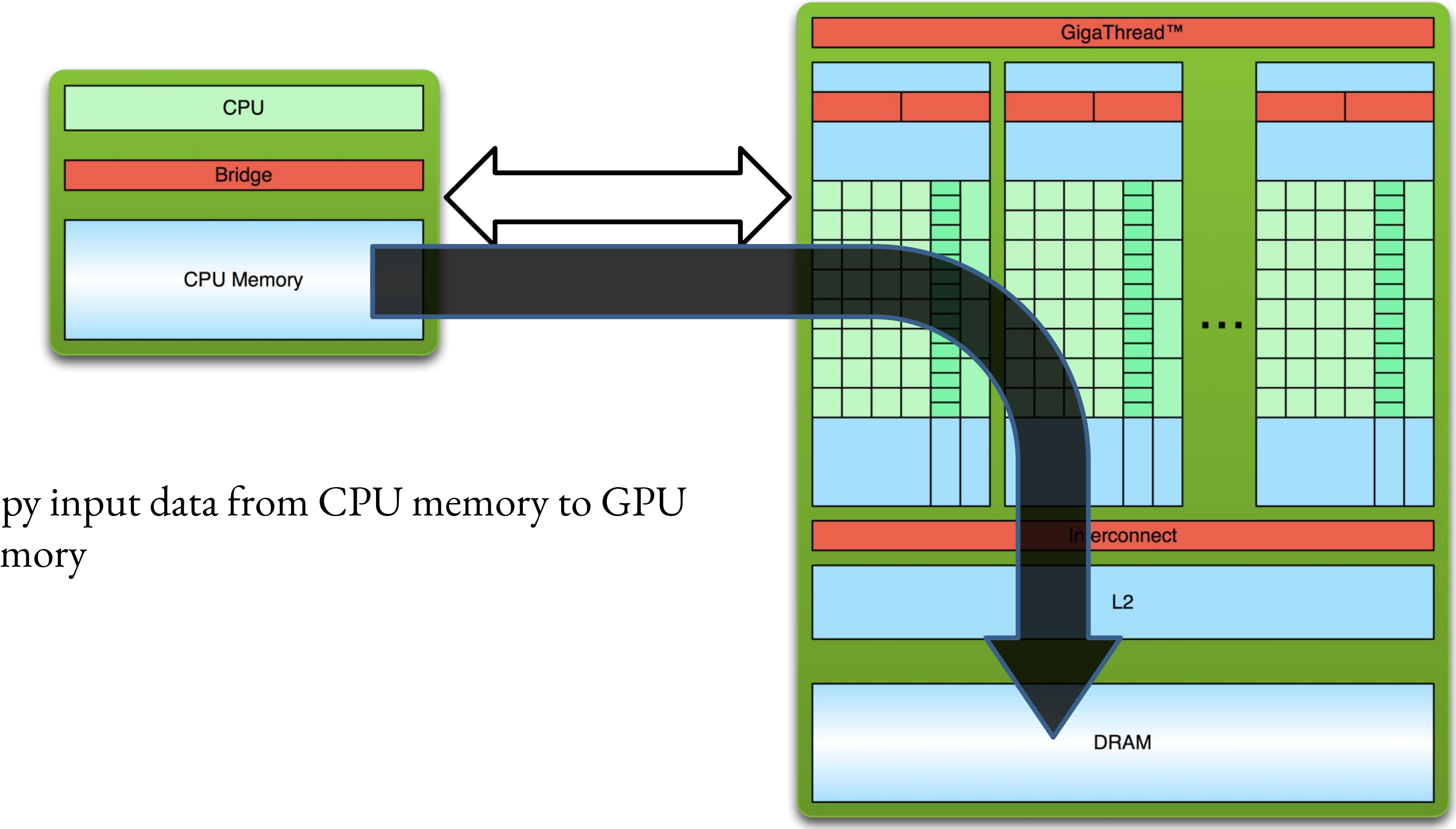
Host

Device

Simple Processing Flow

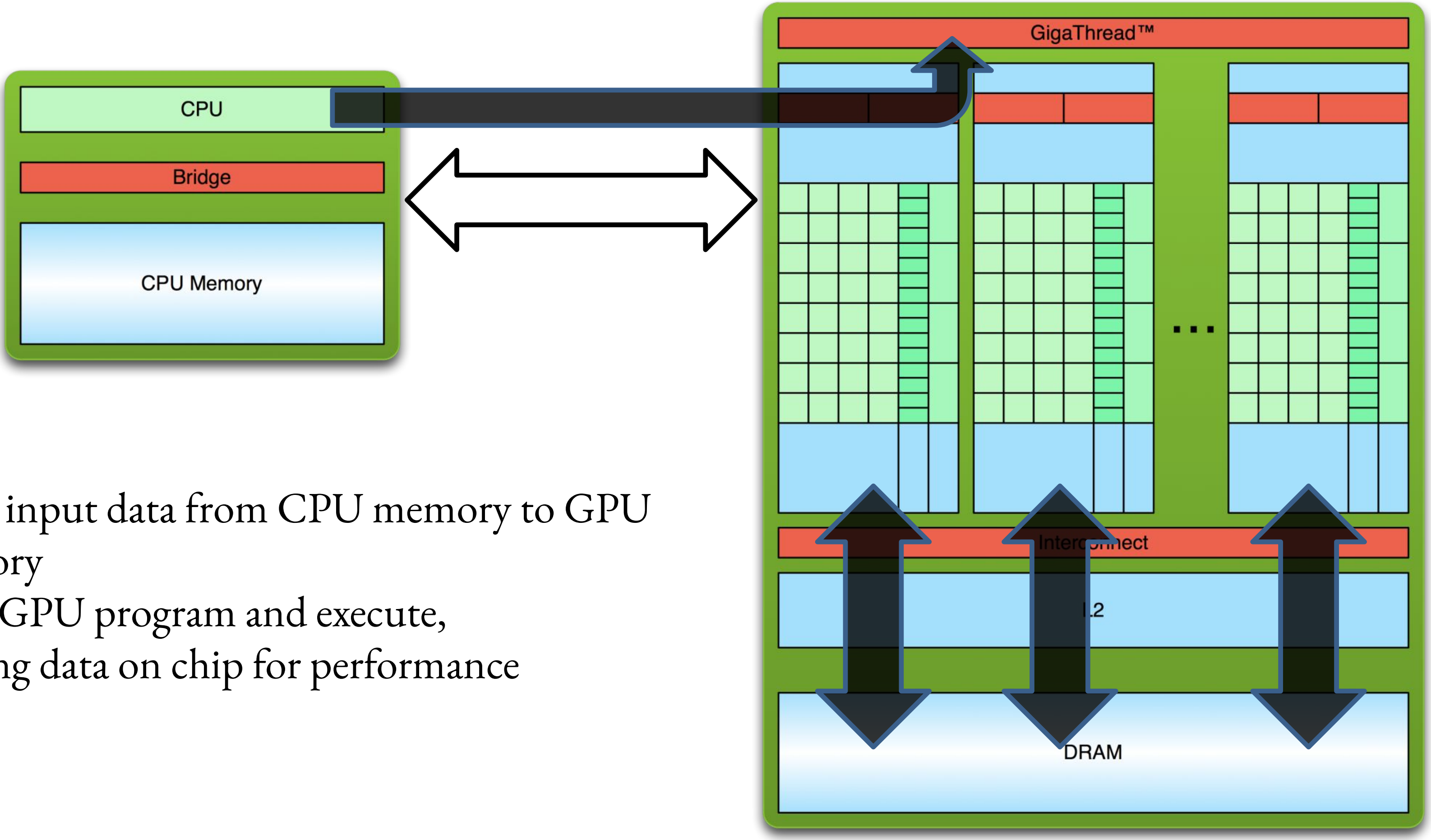


Simple Processing Flow



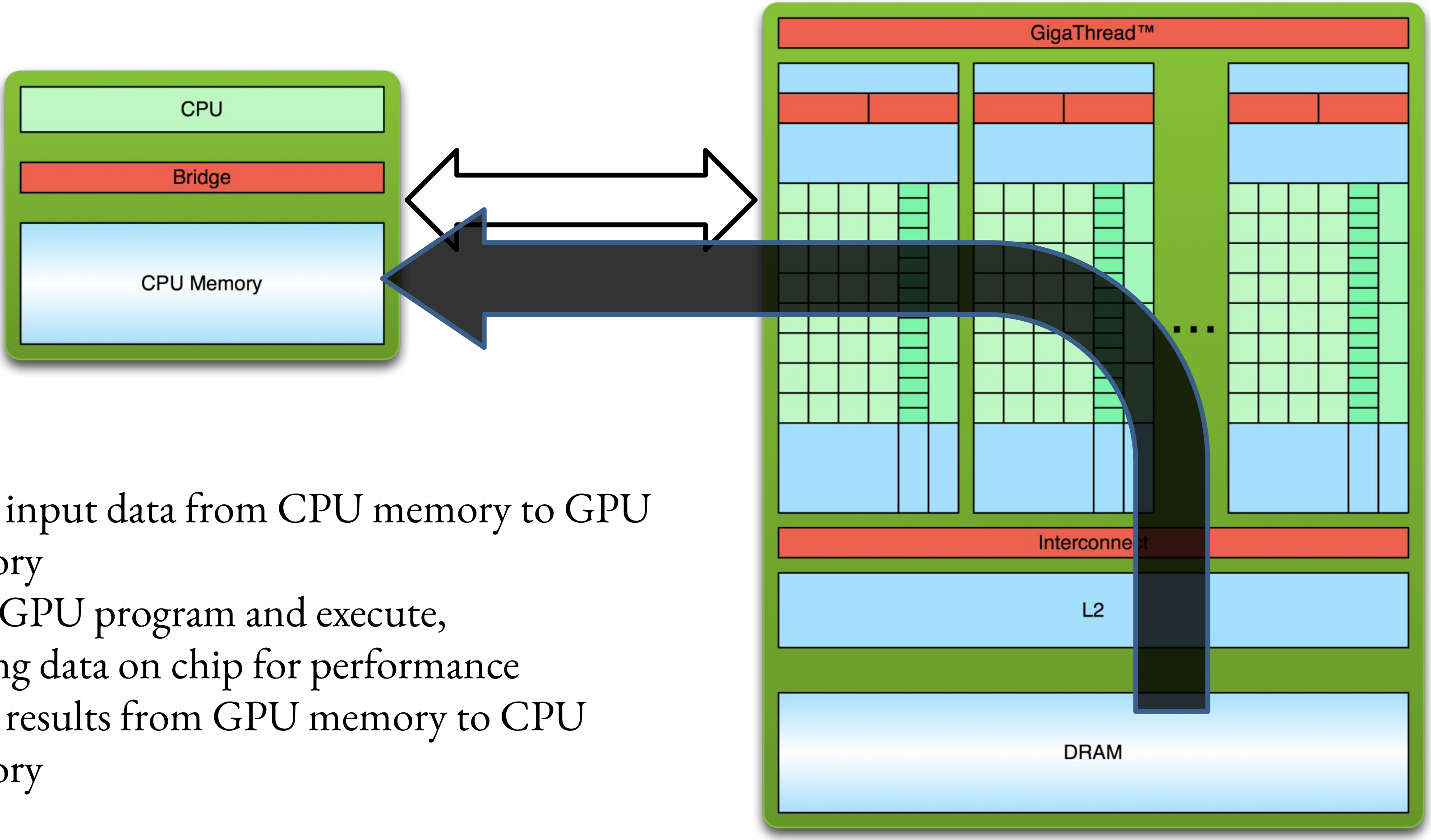
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Basics

CUDA

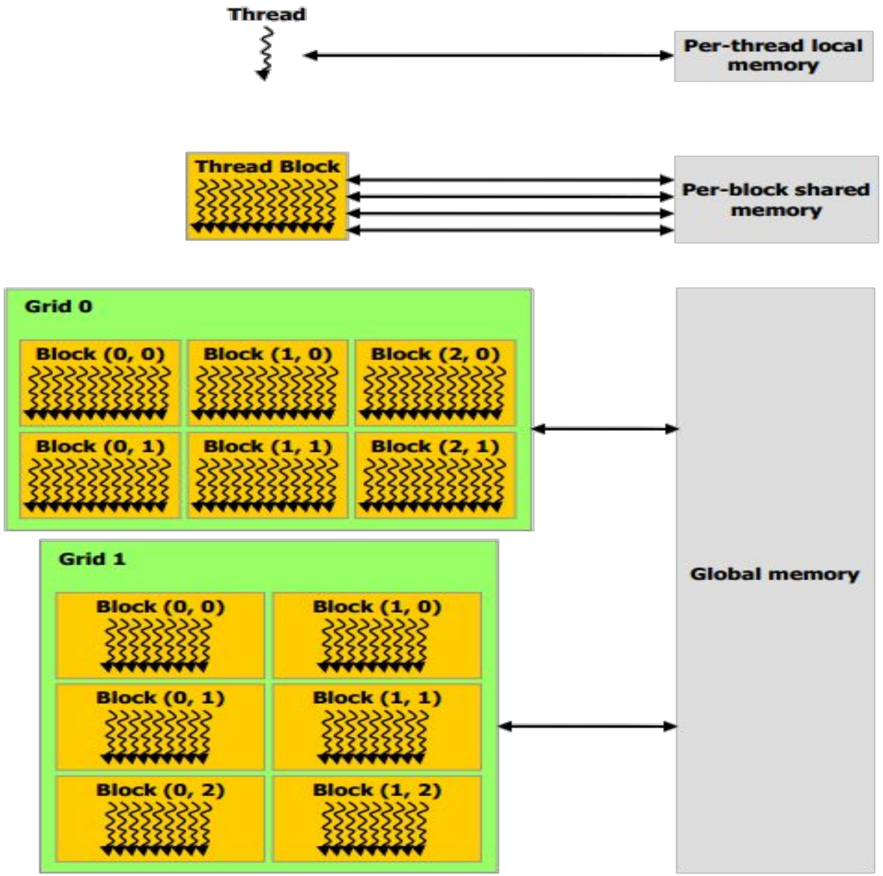
- Small set of extensions to enable asynchronous heterogeneous computing using NVIDIA GPUs
- Straightforward APIs to manage devices, memory etc.
- Concepts learned in CUDA apply to other parallel frameworks (OpenMP, SYCL, HIP).
- Learning Curve
  - Initial effort focuses on understanding new APIs and language constructs.
  - Core programming logic often remains unchanged.

SPMD Phases

- Initialize
  - Establish localized data structure and communication channels
- Obtain a unique identifier
  - Each thread acquires a unique identifier, typically range from 0 to N-1, where N is the number of threads
- Distribute Data
  - Decompose global data into chunks and localize them, or
  - Sharing/replicating major data structure using thread ID to associate subset of the data to threads
- Run the core computation
- Finalize
  - Reconcile global data structure, prepare for the next major iteration

Memory Hierarchy in GPU programming

- Registers/Shared memory:
  - Fast
  - Only accessible by the thread/block
  - Lifetime of the thread/block
- Global memory:
  - Potentially 150x slower than register or shared memory
  - Accessible from either the host or device
  - Lifetime of the application



Hello World!

```cpp
#include <iostream>
int main() {
   std::cout << "Hello World!\n";
}
```

Hello World!

```
#include <iostream>
int main() {
   std::cout << "Hello World!\n";
}
```

Standard C++ that runs on the host

nvcc can be used to compile programs with no *device* code

```
Output:
$ nvcc hello_world.cu
$ ./a.out
Hello World!
$
```

Hello World! with Device Code

```
#include <iostream>
__global__ void mykernel() {
}

int main() {
  cudaStream_t stream; cudaStreamCreate(&stream);
  mykernel<<<1,1,0,stream>>>();
  std::cout << "Hello World!\n";
  cudaStreamSynchronize(stream);
  cudaStreamDestroy(stream);
}
```

Hello World! with Device Code

```
__global__ void mykernel() {
}
```

- CUDA keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by nvcc compiler
  - Host functions (e.g. `main()`) processed by `gcc`

Hello World! with Device Code

```
mykernel<<<1,1,0,stream>>>();
```

- Triple angle brackets mark a call from host code to device code
  - Also called a “kernel launch”
  - We’ll return to the parameters in a moment

- That’s all that is required to execute a function on the GPU!

Parallel constructs in CUDA

Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(const int *a, const int *b, int *c) {
    *c = *a + *b;
}
```

- As before `__global__` is a CUDA keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

Addition on the Device

- Note that we use pointers for the variables

```
__global__ void add(const int *a, const int *b, int *c) {
    *c = *a + *b;
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory

- We need to allocate memory on the GPU

Memory Management

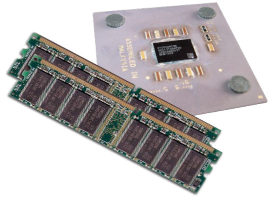
- Host and device memory are separate entities
  - Device pointers point to GPU memory

May be passed to/from host code
May not be dereferenced in host code

  - Host pointers point to CPU memory

May be passed to/from device code
May not be dereferenced in device code

- Simple CUDA API for handling device memory
  - `cudaMalloc(), cudaFree(), cudaMemcpy()`
  - Similar to `malloc(), free(), memcpy()`



Addition on the Device: add()

- Returning to our `add()` kernel

```
__global__ void add(const int *a, const int *b, int *c) {
    *c = *a + *b;
}
```

- Let's take a look at `main()` …

Addition on the Device: main()

```
int main() {
   cudaStream_t stream;
   cudaStreamCreate(&stream);
   int *a, *b, *c;   // host copies of a, b, c
   int *d_a, *d_b, *d_c;// device copies of a, b, c
   int size = sizeof(int);
   // Allocate space for device copies of a, b, c
   cudaMallocHost(&a,size);
   cudaMallocHost(&b,size);
   cudaMallocHost(&c,size);
   *a = 2; *b = 7;
```

Addition on the Device: main()

```
   cudaMallocAsync(&d_a, size, stream);
   cudaMallocAsync(&d_b, size, stream);
   cudaMallocAsync(&d_c, size, stream);
    // Copy inputs to device
   cudaMemcpyAsync(d_a, a, size, cudaMemcpyHostToDevice, stream);
   cudaMemcpyAsync(d_b, b, size, cudaMemcpyHostToDevice, stream);
   // Launch add() kernel on GPU
   add<<<1,1,0,stream>>>(d_a, d_b, d_c);
   // Copy result back to host
   cudaMemcpyAsync(c, d_c, size, cudaMemcpyDeviceToHost, stream);
   cudaFreeAsync(d_a,stream);
   cudaFreeAsync(d_b,stream);
   cudaFreeAsync(d_c,stream);
     // Synchronize to be able to use c...
    cudaStreamSynchronize(stream);
    cudaStreamDestroy(stream);
    cudaFreeHost(a); cudaFreeHost(b); cudaFreeHost(c);
}
```

Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1, 0, stream >>>();
```

```
add<<< N, 1, 0, stream >>>();
```

- Instead of executing `add()` once, execute N times in parallel

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using `blockIdx.x`

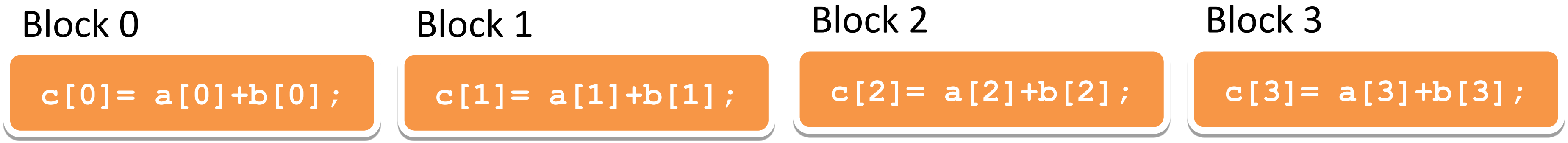
```
__global__ void add(const int *a, const int *b, int *c)

{
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index

Remember SPMD?

```
__global__ void add(const int *a, const int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- On the device, each block can execute in parallel:



Block 0

```
c[0]= a[0]+b[0];
```

Block 1

```
c[1]= a[1]+b[1];
```

Block 2

```
c[2]= a[2]+b[2];
```

Block 3

```
c[3]= a[3]+b[3];
```

Vector Addition on the Device: add()

- Returning to our parallelized `add()` kernel

```
__global__ void add(const int *a, const int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
  }
```

- Let’s take a look at `main()` …

Vector Addition on the Device: main()

```
int main() {
 cudaStream_t stream; cudaStreamCreate(&stream);
 int N = 512;
 std::vector<int> a, b, c;
 a.resize(N); b.resize(N); c.resize(N);
 int *d_a, *d_b, *d_c; // device copies of a, b, c
 int size = N * sizeof(int);
 // Alloc space for host copies of a, b, c and
   // setup input values
    my_favorite_random_ints(a, N);
    my_favorite_random_ints(b, N);
 // Alloc memory for device copies of a, b, c
 cudaMallocAsync(&d_a, size, stream);
 cudaMallocAsync(&d_b, size, stream);
 cudaMallocAsync(&d_c, size, stream);
```

Vector Addition on the Device:

```
// Copy inputs to device
cudaMemcpyAsync(d_a, a.data(), size, cudaMemcpyHostToDevice, stream);
cudaMemcpyAsync(d_b, b.data(), size, cudaMemcpyHostToDevice, stream);
// Launch add() kernel on GPU with N blocks
add<<<N, 1, 0, stream>>>(d_a, d_b, d_c);
// Copy result back to host
cudaMemcpyAsync(c.data(), d_c, size, cudaMemcpyDeviceToHost, stream);
// Cleanup
cudaFreeAsync(d_a,stream);
cudaFreeAsync(d_b,stream);
cudaFreeAsync(d_c,stream);
cudaStreamSynchronize(stream);
// Now you can use content of the c vector…
cudaStreamDestroy(stream);
}
```

CUDA Threads

- Terminology: a block can be split into parallel threads
- Let’s change add() to use parallel threads instead of parallel blocks

```
__global__ void add(const int *a, const int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- We use `threadIdx.x` instead of `blockIdx.x`
- Need to make one change in `main()…`

Combining Blocks and Threads

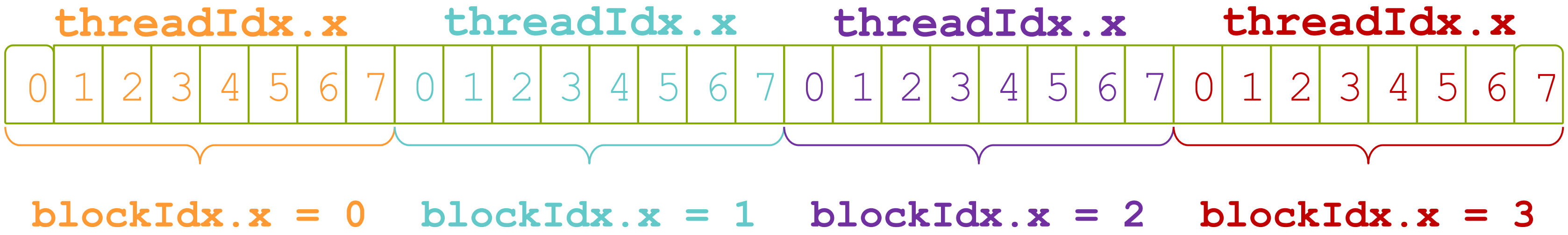
- We’ve seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads

Let’s adapt vector addition to use both blocks and threads

Why? We’ll come to that…
First let’s discuss data indexing…

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)



With `blockDim.x` threads/block a unique index for each thread is given by:

```
auto index = threadIdx.x + blockIdx.x * blockDim.x;
```

Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

```
auto index = threadIdx.x + blockIdx.x * blockDim.x;
```

- Combined version of add() to use parallel threads *and* parallel blocks

```
__global__ void add(const int *a, const int *b, int *c) {
    auto index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

What changes need to be made in main()?

Vector Addition on the Device: main()

```
int main() {
  cudaStream_t stream; cudaStreamCreate(&stream);
  int N = 2048*2048;
  int threads_per_block = 512;
  std::vector<int> a, b, c;
  a.resize(N); b.resize(N); c.resize(N);
  int *d_a, *d_b, *d_c; // device copies of a, b, c
  int size = N * sizeof(int);
  // Alloc space for host copies of a, b, c and
  // setup input values
  my_favorite_random_ints(a, N);
  my_favorite_random_ints(b, N);
  // Alloc memory for device copies of a, b, c
  cudaMallocAsync(&d_a, size, stream);
  cudaMallocAsync(&d_b, size, stream);
  cudaMallocAsync(&d_c, size, stream);
```

Vector Addition on the Device:

```
  // Copy inputs to device
  cudaMemcpyAsync(d_a, a.data(), size, cudaMemcpyHostToDevice, stream);
  cudaMemcpyAsync(d_b, b.data(), size, cudaMemcpyHostToDevice, stream);
  // Launch add() kernel on GPU with N blocks
  add<<<N/threads_per_block,threads_per_block, 0, stream>>>(d_a, d_b, d_c);
  // Copy result back to host
  cudaMemcpyAsync(c.data(), d_c, size, cudaMemcpyDeviceToHost, stream);
  // Cleanup
  cudaFreeAsync(d_a,stream); cudaFreeAsync(d_b,stream);cudaFreeAsync(d_c,stream);
  cudaStreamSynchronize(stream);
  // Now you can use content of the c vector…
  cudaStreamDestroy(stream);
}
```

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`

- Avoid accessing beyond the end of the arrays:

```
__global__ void add(const int *a, const int *b, int *c, int n) {
    auto index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

Update the kernel launch:

```
add<<<(n + nThPerBlock - 1)/nThPerBlock, nThPerBlock >>>(d_a,d_b, d_c, n);
```

Time for exercises!

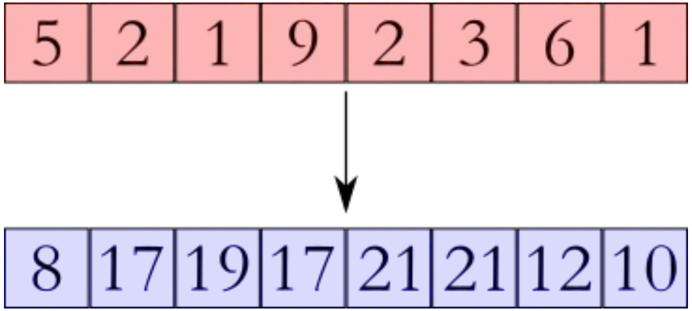
Shared Memory

Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize
- To understand the gain, we need a new example…

1D Stencil

- Consider applying a 1D stencil sum to a 1D array of elements
  - Each output element is the sum of input elements within a radius
  - Example of stencil with radius 2:



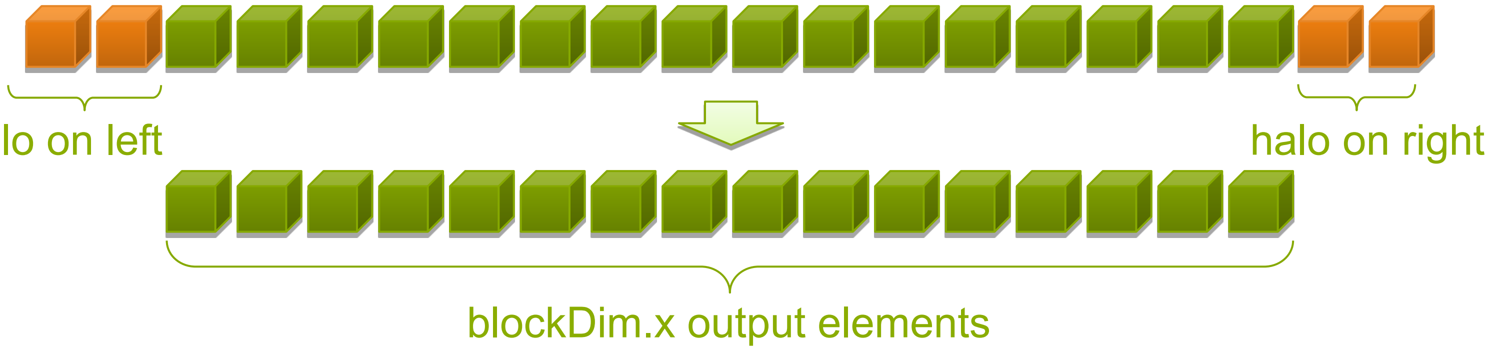
5	2	1	9	2	3	6	1
8	17	19	17	21	21	12	10

Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

Implementing With Shared Memory

- Cache data in shared memory
  - Read `(blockDim.x + 2 * radius)` input elements from global memory to shared memory
  - Compute `blockDim.x` output elements
  - Write `blockDim.x` output elements to global memory
  - Each block needs a halo of radius elements at each boundary



Stencil Kernel

```
__global__ void stencil_1d(const int *in, int *out) {
```

Stencil Kernel

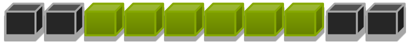
```
__global__ void stencil_1d(const int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
```

Stencil Kernel

```
__global__ void stencil_1d(const int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  auto g_index = threadIdx.x + blockIdx.x * blockDim.x;
  auto s_index = threadIdx.x + RADIUS;
```

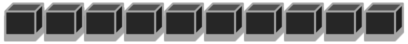
Stencil Kernel

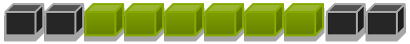
```
__global__ void stencil_1d(const int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  auto g_index = threadIdx.x + blockIdx.x * blockDim.x;
  auto s_index = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[s_index] = in[g_index];
```

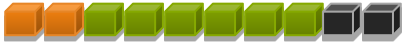


Stencil Kernel

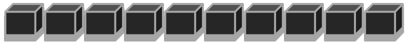
```
__global__ void stencil_1d(const int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  auto g_index = threadIdx.x + blockIdx.x * blockDim.x;
  auto s_index = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[s_index] = in[g_index];
  if (threadIdx.x < RADIUS) {
    temp[s_index - RADIUS] = in[g_index - RADIUS];
```

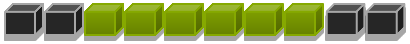


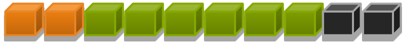




Stencil Kernel



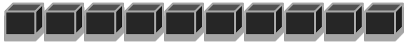


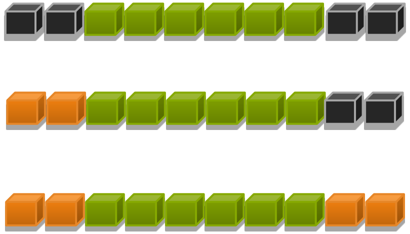


```
__global__ void stencil_1d(const int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  auto g_index = threadIdx.x + blockIdx.x * blockDim.x;
  auto s_index = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[s_index] = in[g_index];
  if (threadIdx.x < RADIUS) {
    temp[s_index - RADIUS] = in[g_index - RADIUS];
    temp[s_index + BLOCK_SIZE] =
```

Stencil Kernel

```
__global__ void stencil_1d(const int *in, int *out, int n) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
 auto g_index = threadIdx.x + blockIdx.x * blockDim.x;
 if (g_index < n) {
  auto s_index = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[s_index] = in[g_index];
  if (threadIdx.x < RADIUS) {
    temp[s_index - RADIUS] = g_index - RADIUS < 0? 0 :
                             in[g_index - RADIUS];
    temp[s_index + BLOCK_SIZE] = g_index + BLOCK_SIZE < n ?
                                 in[g_index + BLOCK_SIZE] : 0;
  }
```





Stencil Kernel

```
  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[s_index + offset];

  // Store the result
  out[g_index] = result;
 }
}
```

Race condition

- The stencil example will not work…
- A race condition occurs when multiple tasks read from and write to the same memory without proper synchronization.
- The "race" may finish correctly sometimes and therefore complete without errors, and at other times it may finish incorrectly.
- If a data race occurs, the behavior of the program is undefined.

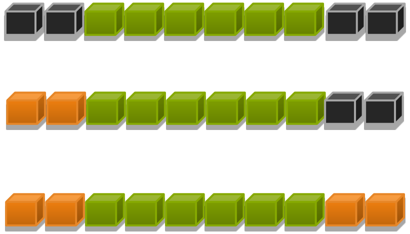
`__syncthreads()`

- `void __syncthreads();`

- Synchronizes all threads within a block
  - Ensuring correct execution order when threads share data.
  - Used to prevent race conditions
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

Stencil Kernel

```
__global__ void stencil_1d(const int *in, int *out, int n) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
 auto g_index = threadIdx.x + blockIdx.x * blockDim.x;
 if (g_index < n) {
  auto s_index = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[s_index] = in[g_index];
  if (threadIdx.x < RADIUS) {
    temp[s_index - RADIUS] = g_index - RADIUS < 0? 0 :
                             in[g_index - RADIUS];
    temp[s_index + BLOCK_SIZE] = g_index + BLOCK_SIZE < n ?
                                 in[g_index + BLOCK_SIZE] : 0;
  }
__syncthreads();
```



Stencil Kernel

```
  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[s_index + offset];

  // Store the result
  out[g_index] = result;
 }
}
```

Review (1 of 2)

- Launching parallel threads
  - Launch N blocks with M threads per block with `kernel<<<N,M,0,stream>>>(…);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block

- Allocate elements to threads:

```
auto index = threadIdx.x + blockIdx.x * blockDim.x;
```

Review (2 of 2)

- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks

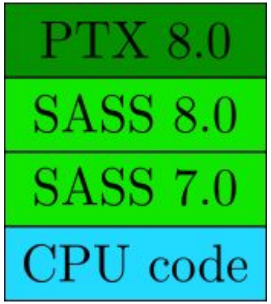
- Use `__syncthreads()` as a barrier to prevent data hazards

Device Management

Compute Capability

- The compute capability of a device describes its architecture, e.g.
  - Number of registers
  - Sizes of memories
  - Features & capabilities
- By running the application deviceQuery in the practical part you will be able to know useful information like
  - The maximum number of threads per block
  - The amount of shared memory
  - The frequency of the memory
- The compute capability is given as a major.minor version number (i.e: Pascal: 6.0, Volta: 7.0, Ampere: 8, Hopper: 9)

CUDA Binary



PTX 8.0

SASS 8.0

SASS 7.0

CPU code

- Exact match of SASS runs natively
  - Many copies of SASS may exist in one fat binary
  - This binary will just work on Ampere (8) and Volta (7)
- When running on a GPU for which SASS does not exist in the binary, CUDA PTX compiler recompiles for the new GPUs
  - Forward compatibility guaranteed by JIT compilation of PTX to future compute capabilities

Hardware vs Software

- From a programmer’s perspective:
  - Blocks
  - Kernel
  - Threads
  - Grid
- Hardware implementation:
  - Streaming multiprocessors (SM)
  - Warps and Lanes

CUDA Runtime system

- Threads assigned to execution resources on a block-by-block basis.
- CUDA runtime automatically reduces number of blocks assigned to each SM until resource usage is under limit.
- Runtime system:
  - maintains a list of blocks that need to execute
  - assigns new blocks to SM as they compute previously assigned blocks
- Example of SM resources:
  - threads/block or threads/SM or blocks/SM
  - number of threads that can be simultaneously tracked and scheduled
  - shared memory

Coordinating Host & Device

- Kernel launches are asynchronous
  - control is returned to the host thread before the device has completed the requested task
  - CPU needs to synchronize before consuming the results

`cudaMemcpy(), cudaMalloc()`	Blocks the CPU thread until the copy/allocation is complete Copy/allocation begins when all preceding CUDA calls have completed
`cudaMemcpyAsync(), cudaMallocAsync()`	Asynchronous, does not block the CPU thread
`cudaDeviceSynchronize()`	Blocks the CPU thread until all preceding CUDA calls have completed
`cudaStreamSynchronize(stream)`	Blocks the CPU thread until all preceding CUDA calls in the stream have completed

Pinned memory

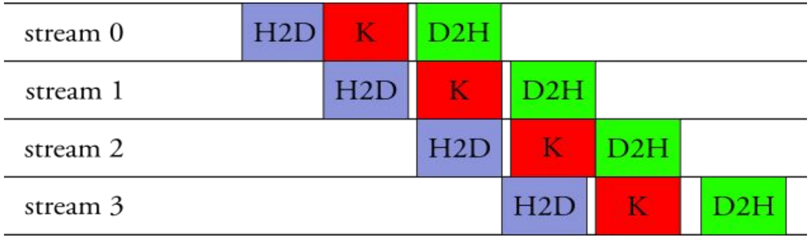
- Pinned memory is a main memory area that is not pageable by the operating system
- Ensures faster transfers (the DMA engine can work without CPU intervention)
- The only way to get closer to PCI peak bandwidth
- Allows CUDA asynchronous operations to work correctly

```
// allocate pinned memory
cudaMallocHost(&area, sizeof(double) * N);
// free pinned memory
cudaFreeHost(area);
```

CUDA streams enable concurrency

- Simultaneous support:
  - CUDA kernels on GPU
  - 2 cudaMemcpyAsync (in opposite directions)
  - Computation on the CPU
- Requirements for Concurrency:
  - CUDA operations must be in different, non-0, streams
  - cudaMemcpyAsync with host from 'pinned' memory

CUDA Streams



```
std::vector<cudaStream_t> streams(4);
for (auto& s: streams) cudaStreamCreate(&s);
std::vector<float*> hPtrs(4); std::vector<float*> dPtrs(4);
for (int i=0; i<4; ++i) cudaMallocHost(&hPtrs[i],memSize);
for (int i=0; i<4; ++i) {
    cudaMallocAsync(&dPtrs[i],memSize, streams[i]);
    cudaMemcpyAsync(dPtrs[i],hPtrs[i], memSize, cudaMemcpyHostToDevice, streams[i]);
    kernelA<<<100,512,0,streams[i]>>>(dPtrs[i]);
    kernelB<<<100,512,0,streams[i]>>>(dPtrs[i]);
    cudaMemcpyAsync(hResults[i],dPtrs[i],memSize, cudaMemcpyDeviceToHost, streams[i]);
}
for (auto& s: streams) {
    cudaStreamSynchronize(s);
    cudaStreamDestroy(s); // if the stream is not needed any longer
}
```

Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself

  OR
  - Error in an earlier asynchronous operation (e.g. kernel)

- Get the error code for the last error:
  ```
  cudaError_t cudaGetLastError(void)
  ```
- Get a string to describe the error:
  ```
  char *cudaGetErrorString(cudaError_t)

  cudaGetErrorString(cudaGetLastError());
  ```

Timing

- You can use the standard timing facilities (host side) in an almost standard way...

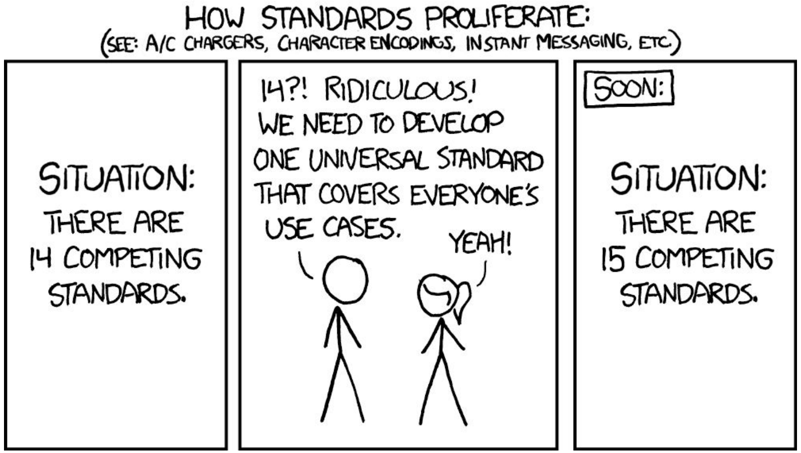
  ...but remember that in general you want to operate on the GPU as asynchronously as possible

Performance portability

- Started effort to make CMS online and offline event reconstruction heterogeneous in 2016

- Ability to write code that can target different hardware (NVIDIA, AMD, Intel GPUs, CPUs).
- Reduces vendor lock-in and expands deployment options.
  - While keeping more than an eye on SYCL, we ported our CUDA code to alpaka portability library

- Fortunately GPUs all work in very similar ways and once you learn one programming model and know how/if to map logical names to the hardware you can program any GPU
  - https://github.com/CHIP-SPV/chipStar
  - https://github.com/ROCm/HIPIFY


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

Emphasizing Parallel Algorithm Design

- Algorithm Adaptability
  - Focusing on data partitioning strategies that work across models.
  - Designing algorithms that minimize inter-thread communication and maximize concurrency.

- Optimization Techniques
  - Memory coalescing, minimizing divergence, and efficient use of shared memory are universal concerns.
  - Profiling and tuning performance using model-specific tools.

Conclusion

- Programming GPUs forces you to think parallel
  - CUDA is very well mapped to the properties of the hardware
- Portable code is key for long-term maintainability, testability and support for new accelerator devices
  - It improves the CPU performance as well if done properly, aiding automatic vectorization
- Many possible solutions, not so many viable ones, even less production ready or compatible with existing infrastructure
- Starting from a CUDA code rather than sequential C++ made life so much easier in the parallelization and portability

References

- CUDA Training material
- CMS plots by CMS collaboration or Patatrack team

Context Switching

- Registers and shared memory are allocated for a block as long as that block is active
- Once a block is active it will stay active until all threads in that block have completed
- Context switching is very fast because registers and shared memory do not need to be saved and restored
- Goal: Have enough transactions in flight to saturate the memory bus
- Latency can be hidden by having more transactions in flight
- Increase active threads or Instruction Level Parallelism