GPU Architecture & Programming

High Performance Computing, Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Outline

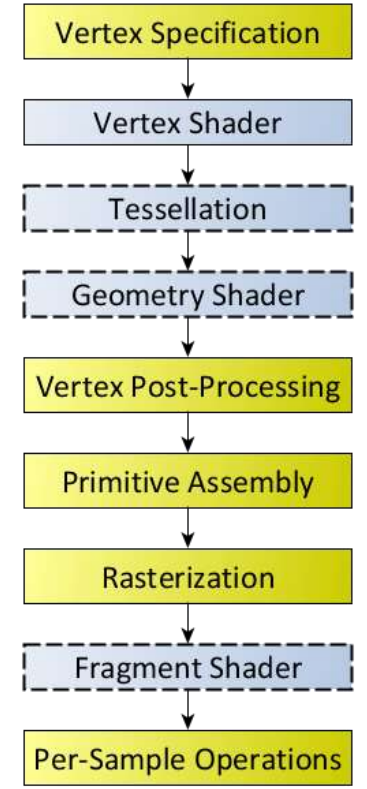
- GPU Architecture
  - SIMT
  - Thread synchronization
  - Memory model
- Introduction to OpenCL
  - Platform model
  - Execution model
  - Memory model

GPU

- GPU: Graphics Processing Unit
  - accelerates graphics
  - graphics processing as pipeline (rasterization pipeline)
  - programmer can program specific stage of the pipeline
- Example: fragment shader in the OpenGL Shading Language

```
uniform sampler2D texture1;
uniform sampler2D texture2;

void main()
{
 FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord), 0.2);
}
```



GPU Computing

- GPU Computing
  - using GPUs for general purpose computing other than graphics
- Early GPGPU (General Purpose GPU) programming: map the problem into the rasterization pipeline
  - arrays as textures
  - kernels as shaders
  - computing as drawing (rendering pass)
- Modern GPGPU programming
  - open, by Khronos: OpenCL, SYCL
  - proprietary, by NVIDIA (CUDA) and AMD (ROCm)

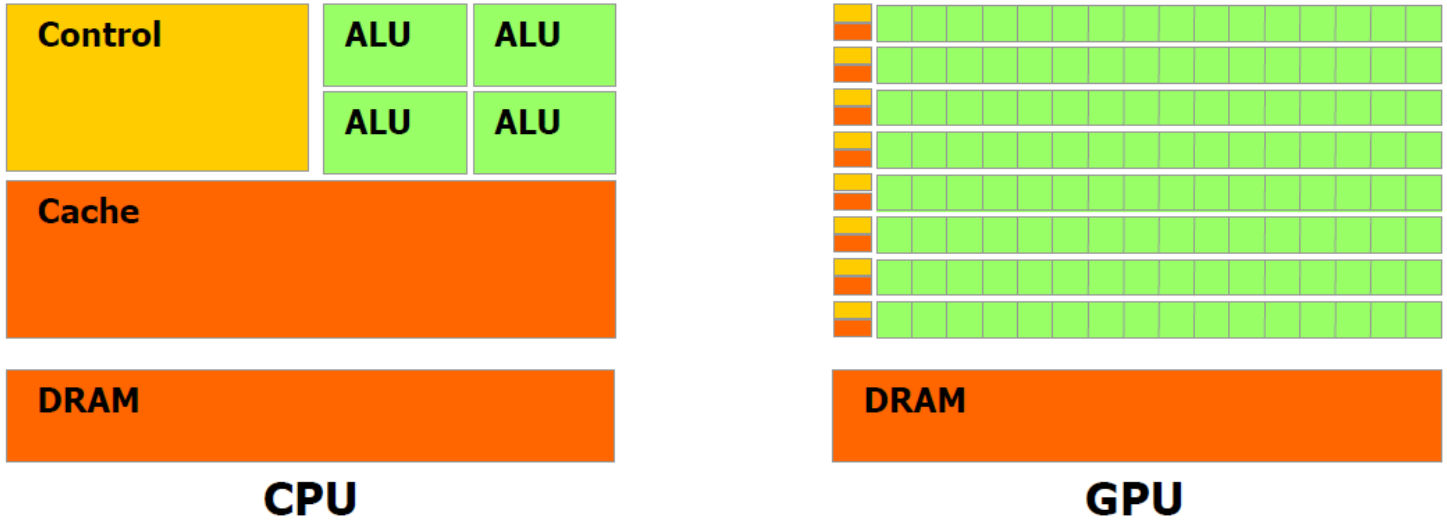
A saxpy kernel in OpenGL SL

```
uniform sampler2D textureY;
uniform sampler2D textureX;
uniform float alpha;

void main(void) {
  vec4 y = texture2D(textureY, gl_TexCoord[0].st);
  vec4 x = texture2D(texture, gl_TexCoord[0].st);
  gl_FragColor = y + alpha*x;
}
```

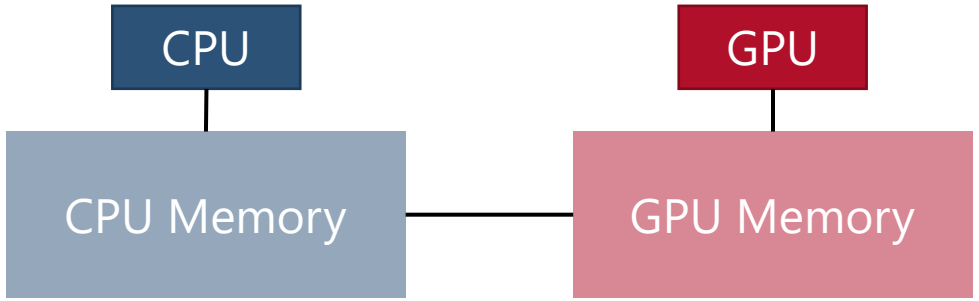
GPU Architecture

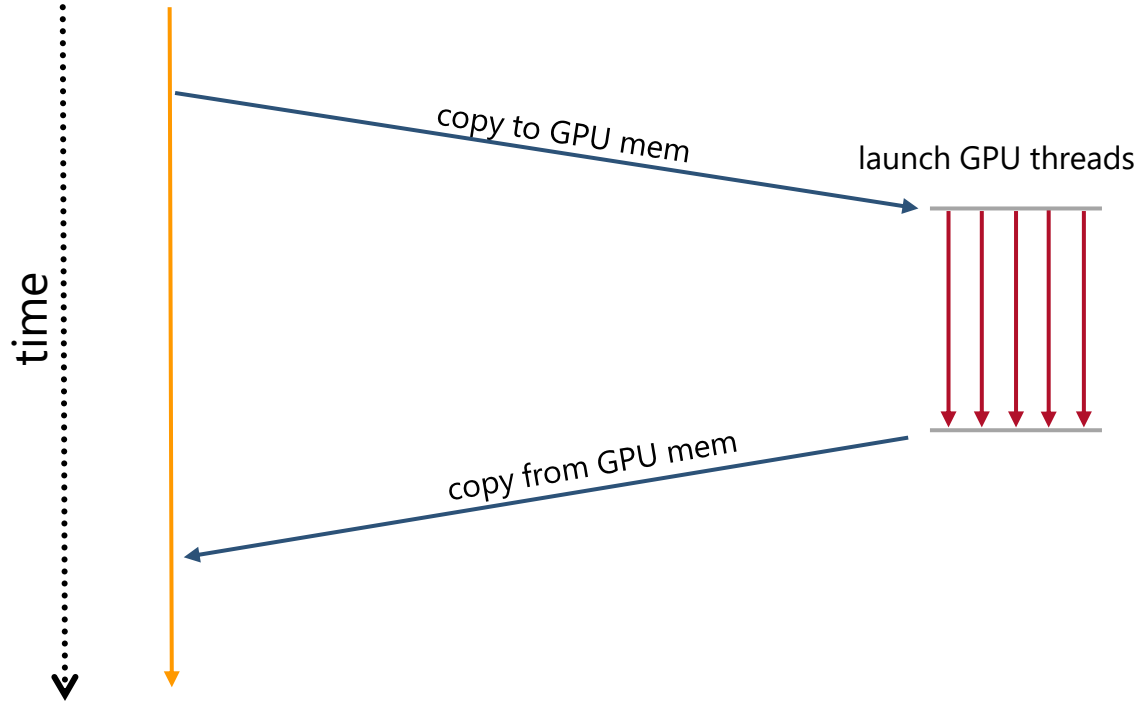
- Control and cache shared among several processing units (ALUs, …)



GPU Architecture

- GPU cards connected to the CPU via bus (e.g., PCI bus)
  - accelerator
- Typical steps
  1. data copied to the GPU memory
  2. computation on the GPU
  3. copy data back from GPU to CPU





GPU Architecture

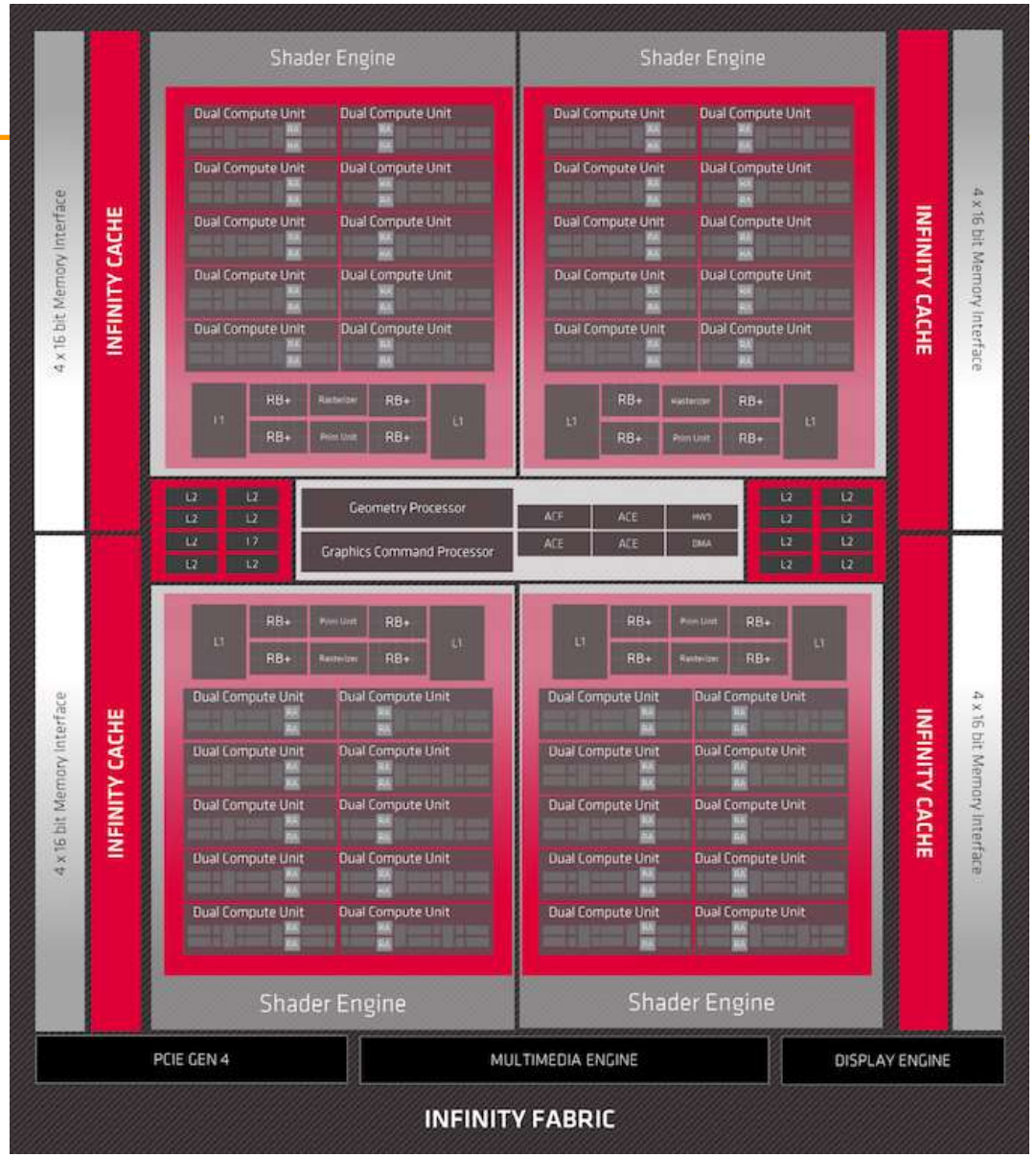
- Warning: different hardware vendors use different words
  - we refer to the OpenCL terminology
- GPU has one or more compute units (CU)
  - NVIDIA streaming multiprocessors (SM)
  - AMD compute units (CU)
- Each compute unit is composed of one or more processing elements (PE)
  - NVIDIA CUDA core
  - AMD ALU/processing element/stream core

GPU Architectures: NVIDIA GV100

- NVIDIA GV100
  - Volta architecture
  - 80 SM
    - streaming multiprocessor
  - tensor core: 8/SM, 640/GPU
  - GPU boost clock 1530MHz
  - peak FP32 15.7 TFLOPS
  - shared memory
    - (configurable up to) 96 KB
  - L2 cache 6144 KB
  - memory size 16GB
  - memory interface 4096-bit HBM2

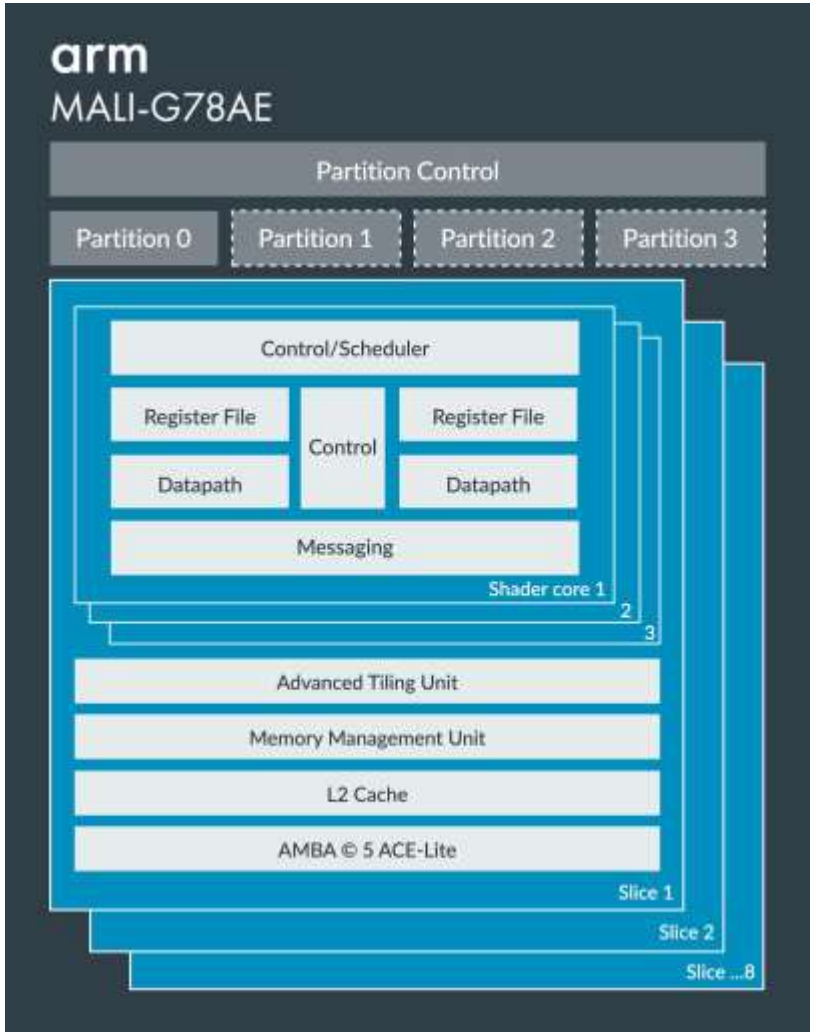
GPU Architectures: AMD RX 6900 XT

- AMD RX 6900 XT
  - RDNA2 architecture
  - streaming multiprocessor
    - 80 compute units
    - 5120 ROP
  - GPU boost clock 2250MHz
  - peak FP32 20.6 TFLOPS
  - Infinity cache 128 MB
  - memory VRAM 16GB
  - memory clock 16 Gbps GDDR6



GPU Architectures: ARM Mali-G78AE GPU

- Arm Mali-G78AE GPU
  - embedded GPU architecture
    - several implementation for autonomous systems
  - support for workload partitioning
    - dedicated hardware resources to different workloads, enabling the full separation of safe and non-safe, or time sensitive, workloads
  - 1 to 24 cores
  - compatible with a wide range of bus interconnect and peripheral IP

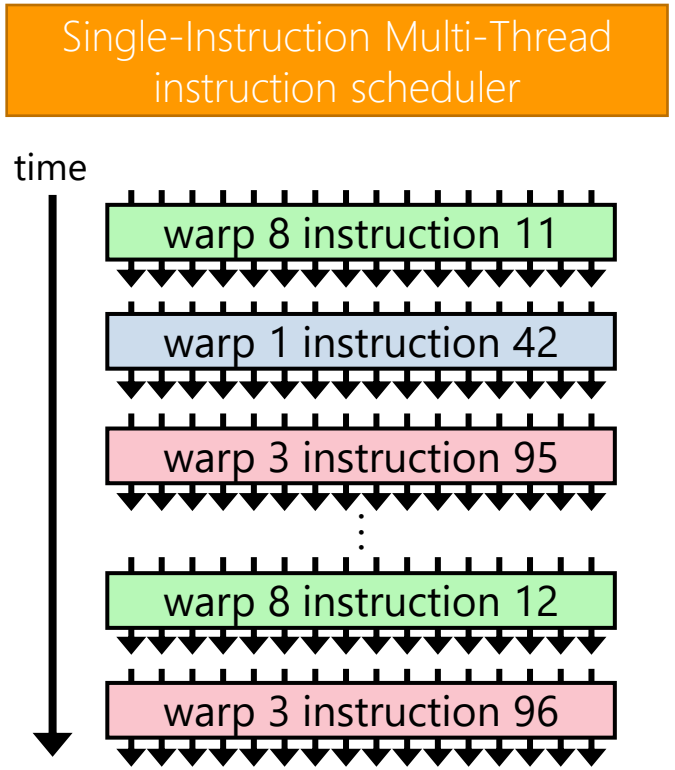


GPU Key Concepts

- Hardware thread management
  - hardware thread launch and monitoring
  - hardware thread switching
  - tens of thousands of lightweight, concurrent threads
- SIMT execution model
- Multiple memory scopes
  - per-thread private memory
  - shared local memory
  - global memory
- Using threads to hide memory latency
- Coarse grain thread synchronization

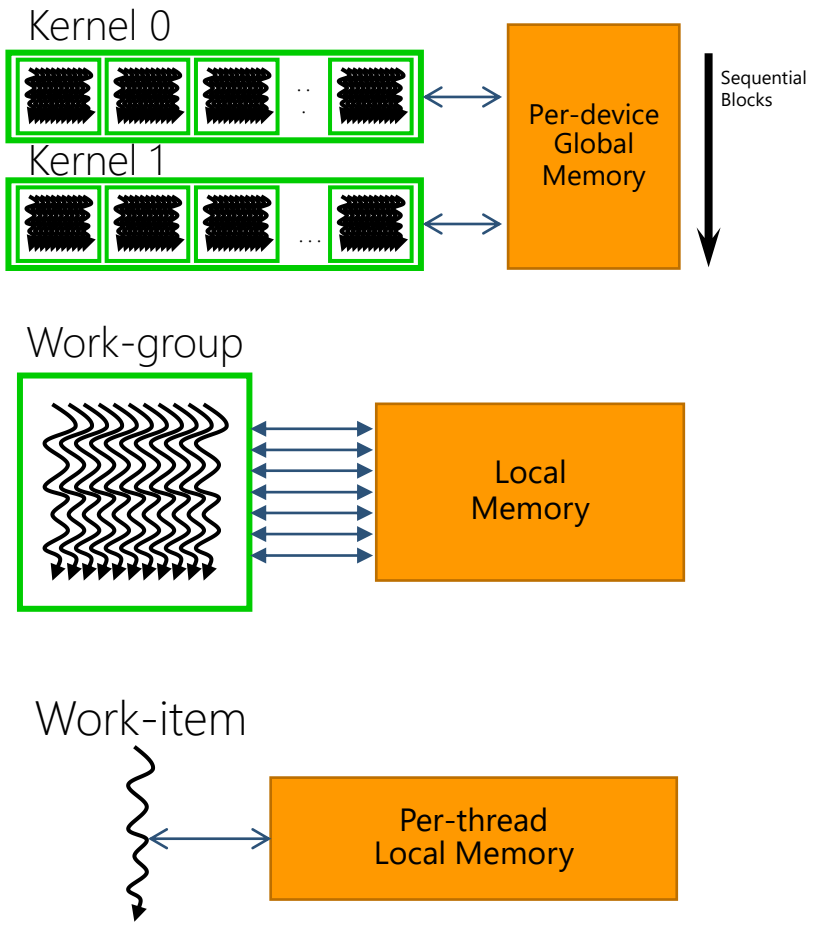
SIMT Multithreaded Execution

- SIMT: Single-Instruction Multi-Thread
  - executes one instruction across many independent threads
  - NVIDIA Warp: 32 parallel threads execute a SIMT instruction
  - AMD Wavefront: 64 parallel threads execute a SIMT instruction
  - SIMT provides easy single-thread scalar programming with SIMD efficiency
  - hardware implements zero-overhead thread scheduling
- SIMT threads can execute independently
  - SIMT warp/wavefront diverges and converges when threads branch independently
  - best efficiency and performance when threads of a warp execute together



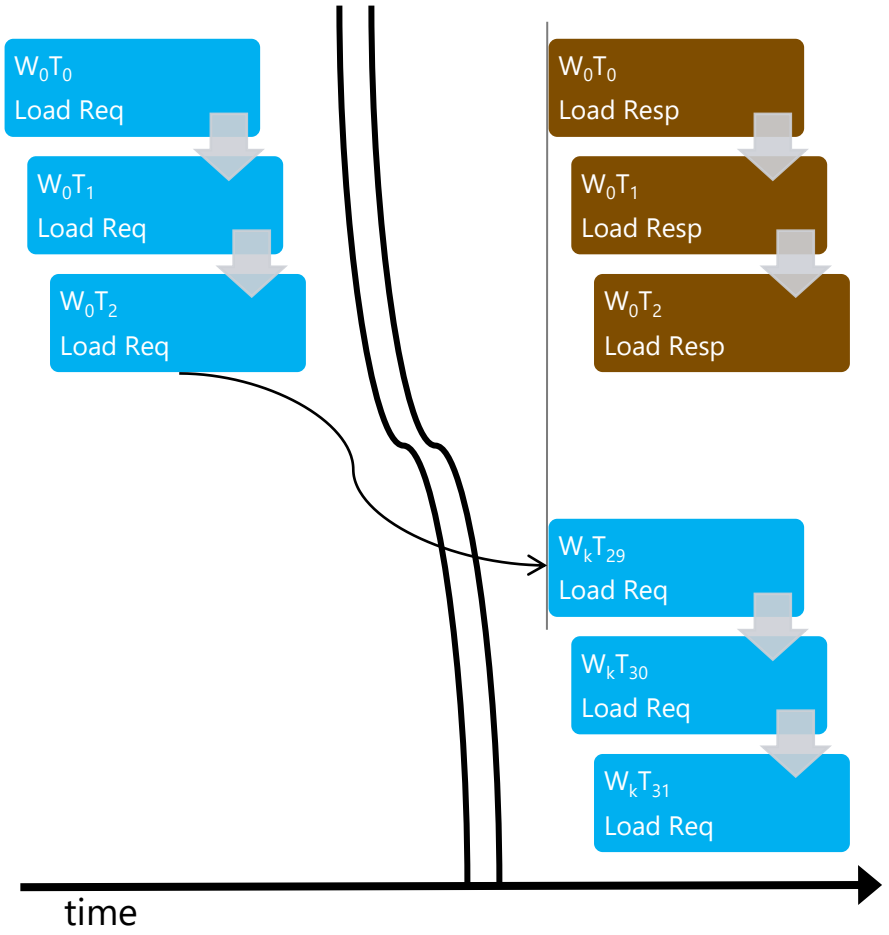
GPU Memory

- Global memory
  - traditionally, the GPU frame buffer
  - can be accessed by any thread in any thread block
- Local memory
  - small memory close to the processor, low latency
  - allocated per thread block
  - NVIDIA shared memory
- Private memory
  - each thread has its own local memory
  - stacks, other private data



GPU Latency Hiding

- From queuing theory: Little's Law: $N = \lambda L$
  - N the average number of threads
  - λ average arrival rate
  - L memory latency
- Arrival rate product of:
  - desired execution rate (IPC)
  - density of `load` instructions (%)
- N = # of threads needed to cover latency L



Batching

- Hiding `load` latency with fewer threads
- Use batching
  - group independent `load`s together
  - modified law: $N = \lambda L / B$
    - B = batch size
- Example
  - the values `a`, `b`, and `c` are loaded independently before being used
  - implication is that we can execute 3 loads from one thread before the first use (`a` in this case) causes a stall

```
// batch size 3 example
float *d_A, *d_B, *d_C;
float a, b, c, result;

a = *d_A; b = *d_B; c = *d_C;
result = a * b + c;
```

Thread Synchronization

- Barrier synchronization among threads of a group
  - fast single-instruction barrier
    - OpenCL `barrier();`
    - NVIDIA CUDA `__syncthreads();`
  - synchronizes all threads in a work group
  - once all threads have reached this point, kernel execution resumes normally
  - use before reading local memory written by another thread in the same work group
- Global synchronization between dependent kernels
  - waits for all work group of kernel to complete
  - fast synchronization and kernel launch

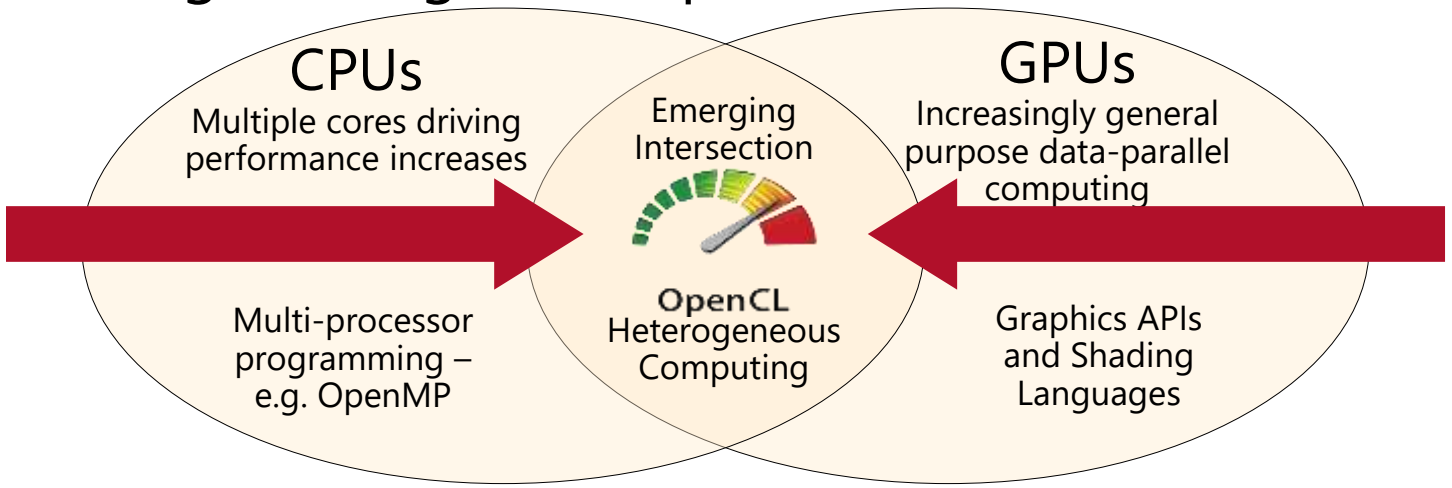
```
__kernel foo(…){
  int lid = get_local_id(0);
  __local int l[WG_SIZE+1];
  l[WG_SIZE] = 0;
  l[lid] = f(…);
  barrier(LK_LOCAL_MEM_FENCE);
  int v = l[lid+1]
}
```

GPU Programming

- First research prototypes
  - Brook (Stanford, 2004)
  - GPGPU with shading languages: OpenGL SL (Khronos), DirectX (Microsoft), Cg (NVIDIA)
- Proprietary
  - CUDA: NVIDIA GPU only
  - ROCm / HIP: AMD GPU only
- Portable languages
  - OpenCL and SYCL, by Khronos
  - Target Intel, AMD, CUDA, ARM, Xilinx, …

OpenCL

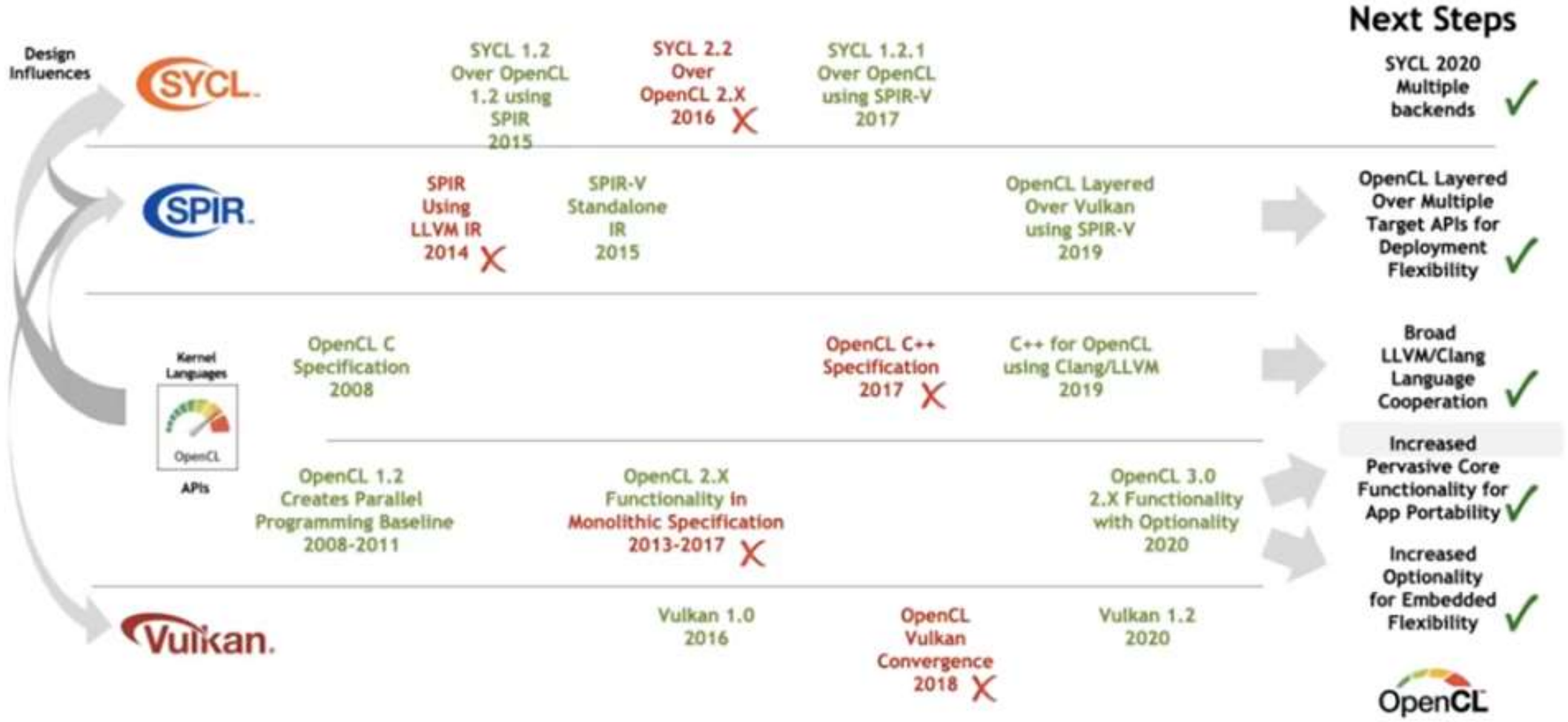
- OpenCL – Open Computing Language
  - open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors
- Industry standards for programming heterogeneous platforms



- Current version 3.0
  - reference card https://www.khronos.org/files/opencl30-reference-guide.pdf
  - API specification https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html

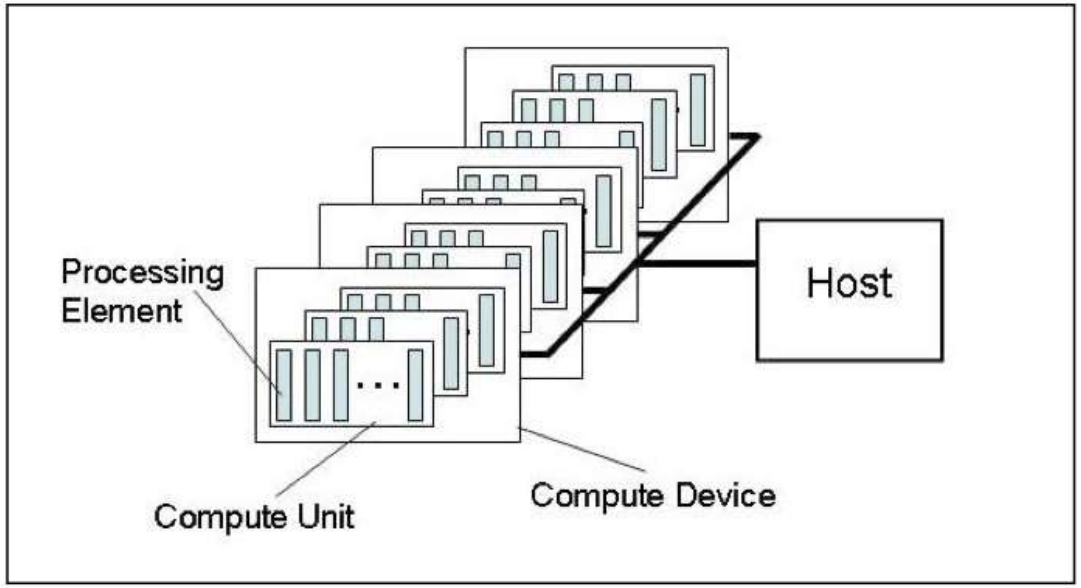
OpenCL Standards Evolution

courtesy of Neil Trevett, Khronos / VP NVIDIA



OpenCL Platform Model

- Platform model
  - one host and one or more OpenCL devices
  - a device is divided into one or more compute units
  - compute units (CU) are divided into one or more processing elements
  - each compute unit is divided into one or more processing elements (PE)
- Memory divided into host memory and device memory

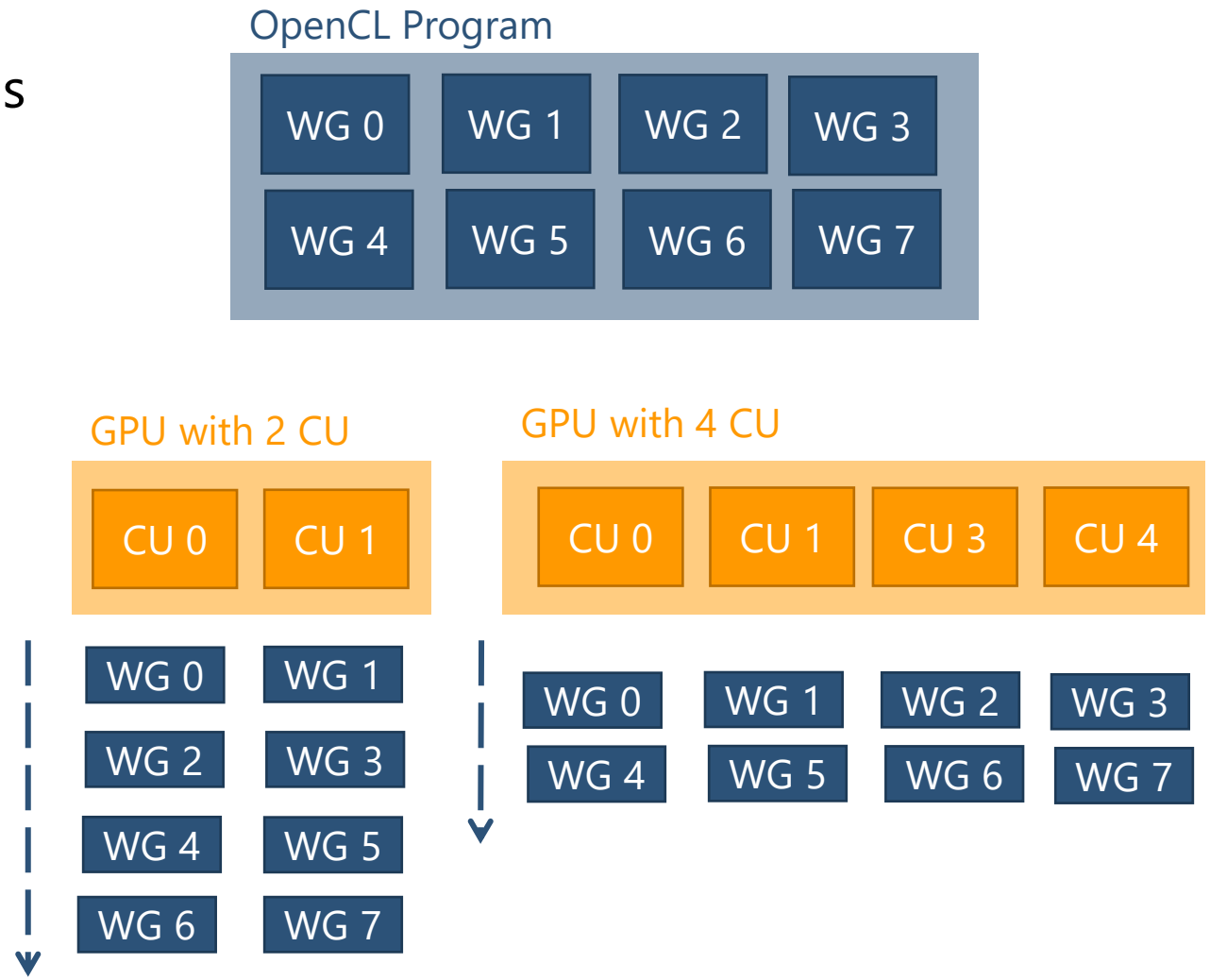


OpenCL Platform Example

- Example: one node, two CPU sockets, two GPUs
- CPUs
  - treated as one OpenCL device
  - one CU per core
  - 1 PE per CU, or if PEs mapped to SIMD lanes, `n` PEs per CU, where `n` matches the SIMD width
  - Remember: the CPU will also have to be its own host!
- GPUs
  - Each GPU is a separate OpenCL device
  - Can use CPU and all GPU devices concurrently through OpenCL

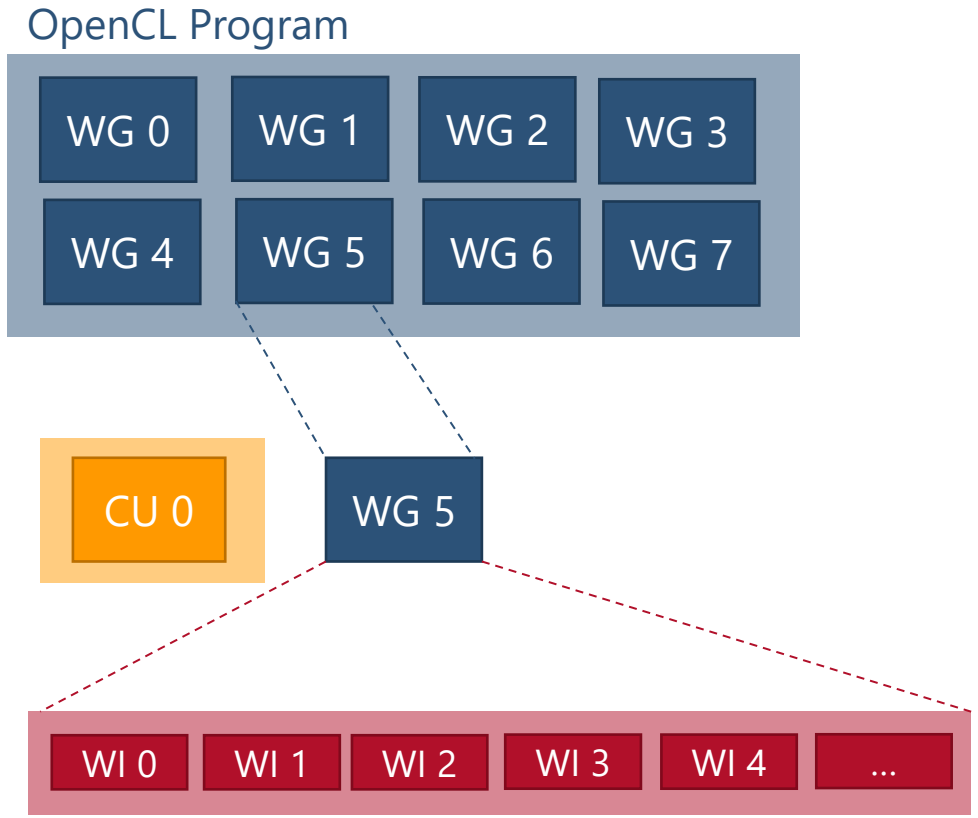
OpenCL Work-group

- Work-group: a collection of work-items that execute on a single compute unit
  - NVIDIA block
- The work-items in the same group
  - execute the same kernel instance
  - share local memory and work-group functions
- We can specify the number of work-items in a work-group
  - this is called the local (work-group) size
  - alternatively, the OpenCL run-time can choose the work-group size for you
    - usually not optimally



OpenCL Work-item

- Work-item: a collection of parallel executions of a kernel invoked on a device by a command
  - NVIDIA CUDA thread
- A work-item
  - is executed by one or more processing elements as part of a work-group executing on a compute unit
  - is distinguished from other work-items by its global ID or the combination of its work-group ID and its local ID within a work-group



Data Parallelism in OpenCL: Kernel

- Replace loops with functions (a kernel) executing at each point in a problem domain
  - e.g., process an array of 1024 elements with one kernel invocation per
  - many instances of the kernel, called work-items, are executed in parallel

Traditional loops

```
void
mul(const int n,
    const float *a,
    const float *b,
          float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

OpenCL Kernel

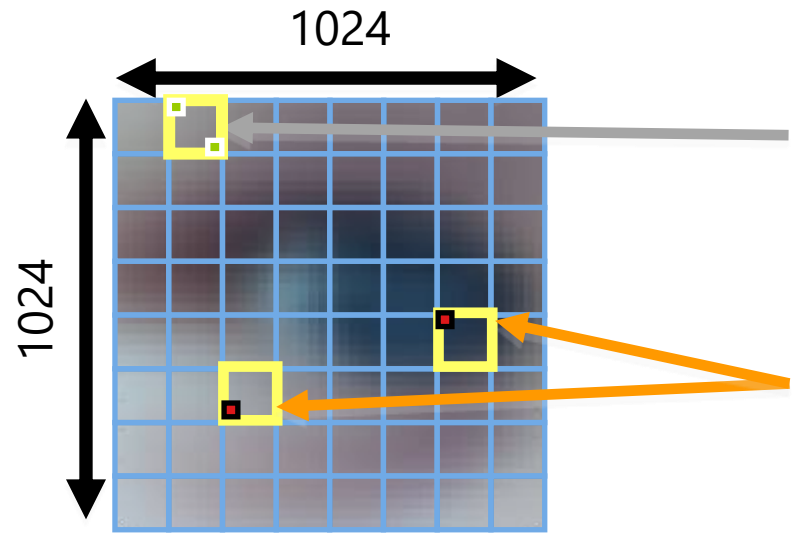
```
__kernel void
mul(__global const float *a,
    __global const float *b,
    __global       float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
```

N-dimensional Domain of Work-items

- The problem we want to compute should have some dimensionality
- For example
  - 1D: vector addition
  - 2D: image filtering
  - 3D: compute a kernel on all points in a cube
- When we execute the kernel we specify up to 3 dimensions
  - and specify the total problem size in each dimension – this is called the global size
- We associate each point in the iteration space with a work-item

Synchronization

- Example: 2D kernel

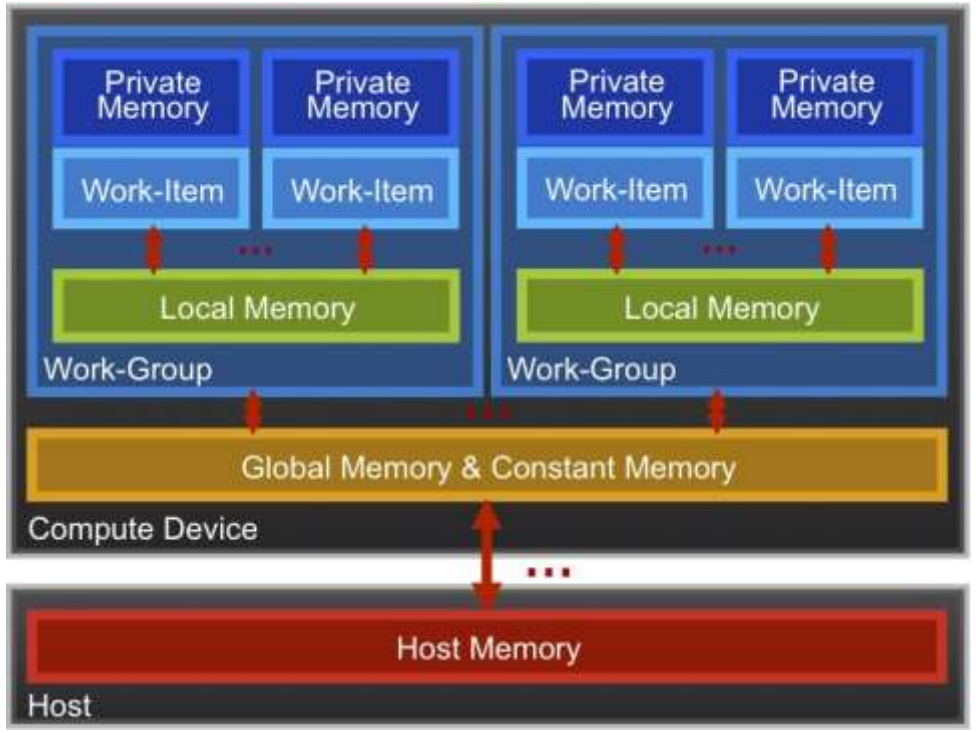


Synchronization between work-items possible only within work-groups: barriers and memory fences

Cannot synchronize between work-groups within a kernel

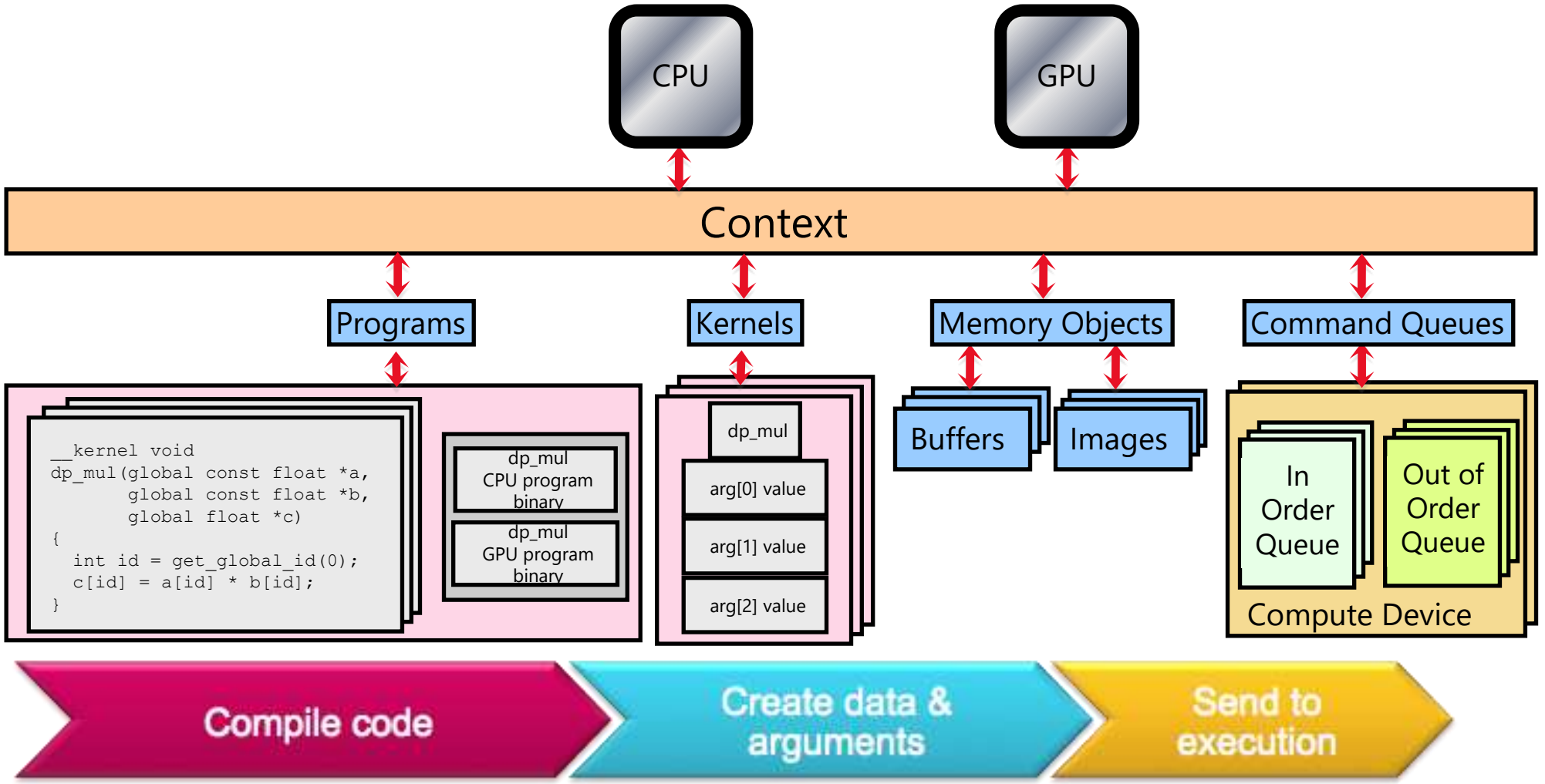
OpenCL Memory Model

- Private memory
  - per work-item
- Local memory
  - shared within a work-group
- Global memory / constant memory
  - visible to all work-groups
- Host memory
  - on the CPU



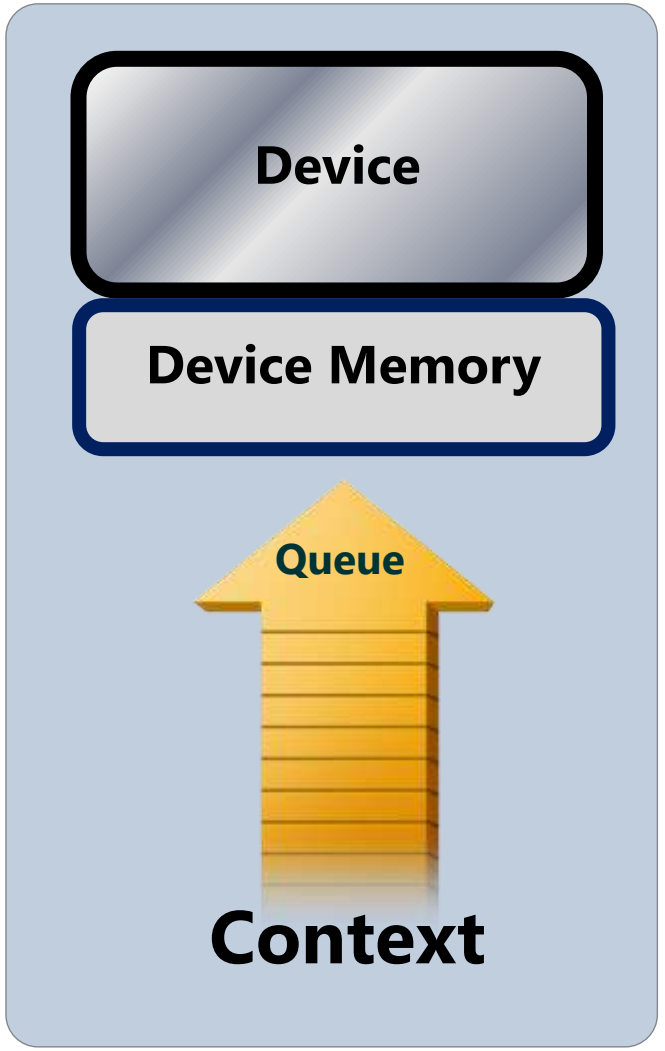
- Memory management is explicit: the programmer is responsible for moving data
  - host → global → local and back

OpenCL Platform Model



Platform Model: Context and Command-Queue

- Context
  - the environment within which kernels execute and in which synchronization and memory management is defined.
  - includes
    - one or more devices
    - device memory
    - one or more command-queues
- All commands for a device are submitted through a command-queue
  - commands: kernel execution, synchronization, and memory transfer operations
- Each command-queue points to a single device within a context

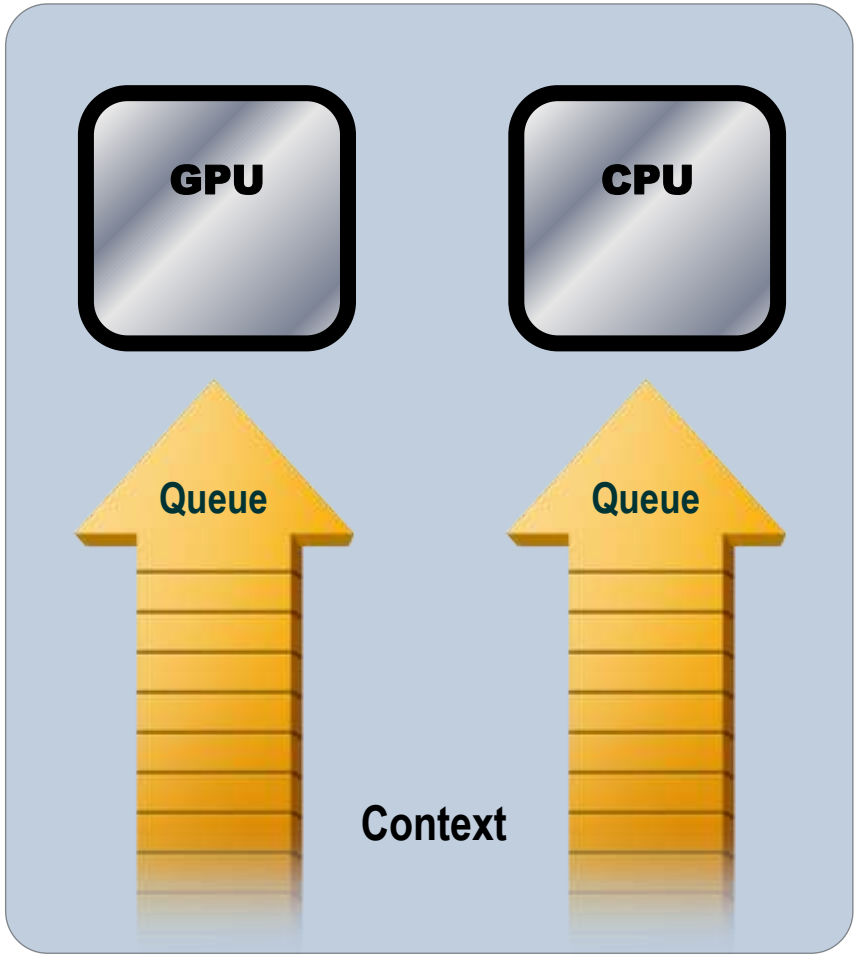


OpenCL Platform Definition

- Grab the first available platform:
  - `err = clGetPlatformIDs(1, &firstPlatformId,  &numPlatforms);`
- Use the first CPU device the platform provides:
  - `err = clGetDeviceIDs(firstPlatformId, CL_DEVICE_TYPE_CPU, 1, &device_id, NULL);`
- Create a simple context with a single device:
  - `context = clCreateContext(firstPlatformId, 1, &device_id, NULL, NULL, &err);`
- Create a simple command-queue to feed our device:
  - `commands = clCreateCommandQueue(context, device_id, 0, &err);`

Command queues

- Commands include:
  - kernel executions
  - memory object management
  - synchronization
- The only way to submit commands to a device is through a command-queue
- Each command-queue points to a single device within a context
  - multiple command-queues can feed a single device
- Used to define independent streams of commands that don't require synchronization



OpenCL Create & Build the Program

- Define source code for the kernel-program as a string literal (great for toy programs) or read from a file (for real applications)
- Build the program object from the source code in `strings`:

```
cl_program clCreateProgramWithSource(cl_context context,
  cl_uint count, const char** strings, const size_t* lengths,
  cl_int* errcode_ret);
```

- Compile the program to create an executable from which specific kernels can be pulled:

```
cl_int clBuildProgram(cl_program program,
  cl_uint num_devices, const cl_device_id* device_list,
  const char* options,
  void (CL_CALLBACK*pfn_notify)(cl_program program,void*user_data),
  void* user_data);
```

OpenCL Setup Memory Objects

- For vector addition we need 3 memory objects, one each for input vectors A and B, and one for the output vector C
- Create input vectors and assign values on the host:

```
float h_a[LENGTH], h_b[LENGTH], h_c[LENGTH];
for (i = 0; i < length; i++) {
    h_a[i] = rand() / (float)RAND_MAX;
    h_b[i] = rand() / (float)RAND_MAX;
}
```

- Define OpenCL memory objects:

```
d_a = clCreateBuffer(context,CL_MEM_READ_ONLY,sizeof(float)*count,NULL, NULL);
d_b = clCreateBuffer(context,CL_MEM_READ_ONLY,sizeof(float)*count,NULL, NULL);
d_c = clCreateBuffer(context,CL_MEM_WRITE_ONLY,sizeof(float)*count,NULL, NULL);
```

What do we put in device memory?

- Memory Objects
  - a handle to a reference-counted region of global memory
- There are two kinds of memory object
- Buffer object:
  - defines a linear collection of bytes (“just a C array”)
  - the contents of buffer objects are fully exposed within kernels and can be accessed using pointers
- Image object:
  - defines a two- or three-dimensional region of memory
  - image data can only be accessed with read and write functions, i.e. these are opaque data structures
  - the read functions use a sampler

Buffer Creation + Copy into Device

- Buffers are declared on the host as type: `cl_mem`
- Arrays in host memory hold your original host-side data:
  - `float h_a[LENGTH], h_b[LENGTH];`
- Cope-host memory flag: create the buffer (`d_a`), assign sizeof(float)*count bytes from `h_a` to the buffer and copy it into device memory

```
cl_mem d_a = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  sizeof(float)*count, h_a, NULL);
```

Memory Flag	Behavior
CL_MEM_READ_WRITE	Specifies memory read / write behavior
CL_MEM_WRITE_ONLY	
CL_MEM_READ_ONLY	
CL_MEM_USE_HOST_PTR	Implementations can cache the contents pointed to by host_ptr in device memory. This cached copy can be used when kernels are executed on a device.
CL_MEM_ALLOC_HOST_PTR	Specifies to the implementation to allocate memory from host accessible memory.
CL_MEM_COPY_HOST_PTR	Specifies to allocate memory for the object and copy the data from memory referenced by host_ptr.

Blocking vs Non-blocking

- Most command queue commands have a `cl_bool` blocking parameter
  - `CL_TRUE` = blocking
  - `CL_FALSE` = non-blocking
- Example
  - Submit command to copy the buffer back to host memory at `h_c`:
    - ```
      clEnqueueReadBuffer(queue, d_c, CL_TRUE,
          sizeof(float)*count, h_c, NULL, NULL, NULL);
      ```
  - If blocking_read is `CL_TRUE` i.e. the read command is blocking, `clEnqueueReadBuffer` does not return until the buffer data has been read and copied into memory pointed to by ptr.
  - If blocking_read is `CL_FALSE` i.e. the read command is non-blocking, `clEnqueueReadBuffer` queues a non-blocking read command and returns
    - the contents of the buffer that ptr points to cannot be used until the read command has completed
    - the event argument returns an event object which can be used to query the execution status of the read command
    - when the read command has completed, the contents of the buffer that ptr points to can be used by the application
      ```

OpenCL Kernel Definition

- Create kernel object from the kernel function “vadd”:

```
kernel = clCreateKernel(program, “vadd”, &err);
```

- Attach arguments of the kernel function “vadd” to memory objects:

```
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &count);
```

Enqueue Commands

- Write buffers from host into global memory (as non-blocking operations):

```
err = clEnqueueWriteBuffer(commands, d_a, CL_FALSE, 0,
sizeof(float)*count, h_a, 0, NULL, NULL);

err = clEnqueueWriteBuffer(commands, d_b, CL_FALSE, 0,
sizeof(float)*count, h_b, 0, NULL, NULL);
```

- Enqueue the kernel for execution (note: in-order so OK):

```
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global,
&local, 0, NULL, NULL);
```

- Read back result (as a blocking operation). We have an in-order queue which assures the previous commands are completed before the read can begin.

```
err = clEnqueueReadBuffer(commands, d_c, CL_TRUE,
sizeof(float)*count, h_c, 0, NULL, NULL);
```

OpenCL Example: Vector Addition

Host code

```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0,CL_DEVICE_TYPE_GPU,
NULL,NULL,NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
cl_device_id[] devices = malloc(cb);
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);

// create a command-queue, buffers, program, kernel
cmd_queue = clCreateCommandQueue(context,devices[0],0,NULL);
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcb, NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_float)*n, NULL, NULL);
program = clCreateProgramWithSource(context, 1, &program_source, NULL, NULL);
err = clBuildProgram(program, 0, NULL,NULL,NULL,NULL);
kernel = clCreateKernel(program, "vec_add", NULL);

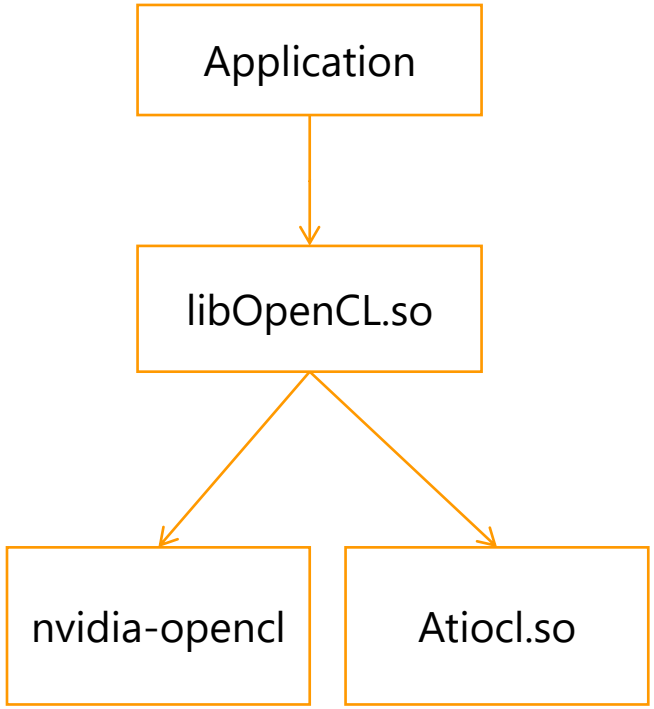
// set argument, run kernel, read back results
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0], sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *) &memobjs[1], sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2], sizeof(cl_mem));
global_work_size[0] = n;
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, global_work_size, NULL,0,NULL,NULL);
err = clEnqueueReadBuffer(cmd_queue, memobjs[2], CL_TRUE, 0, n*sizeof(cl_float), dst, 0, NULL, NULL);
```

Kernel code

```
__kernel void vadd(__global const float *a,
                   __global const float *b,
                   __global       float *c)
 {
     int gid = get_global_id(0);
     c[gid]  = a[gid] + b[gid];
 }
```

OpenCL Setup

- In order to co-ordinate which library each device uses, the runtime uses the ICD (Installable Client Driver)
  - ICD allows multiple implementations to co-exist
  - code only links to libOpenCL.so
    - if you compile your code against a generic runtime, the ICD will load the correct platform runtime when required
  - `clGetPlatformIDs()` and `clGetPlatformInfo()` examine the list of available implementations and select a suitable one
- on Linux: runtimes are listed in the `/etc/OpenCL/vendors` directory



Lab: Install OpenCL

1. OpenCL setup
   - on your local machine
   - on Google Colab
2. Run the vector addition kernel

- to inspect and verify that you can run an OpenCL kernel
- look at the host code and identify the API calls in the host code
- compare them against the API descriptions on the OpenCL reference card

GPU Memory

High Performance Computing, Summer 2021

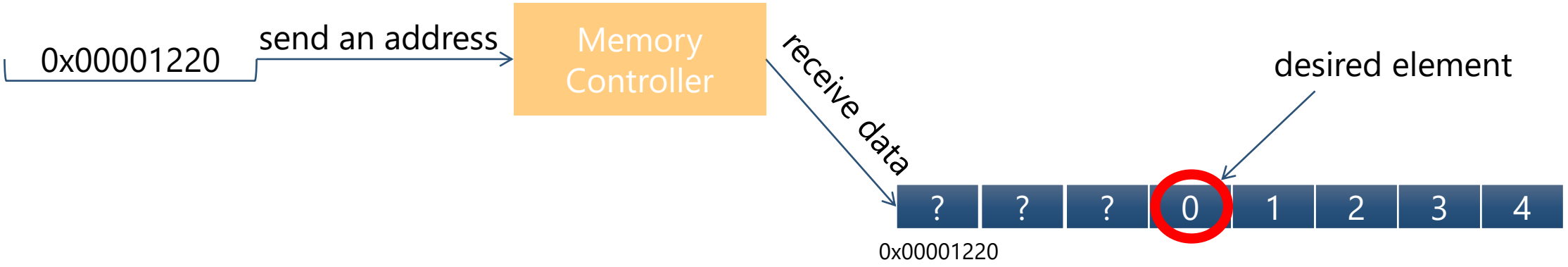
Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Outline

- GPU memory
  - coalescing memory access
  - local and private memory
- Memory optimization in OpenCL
  - local memory
  - atomics
- Example codes
  - n-body
  - matrix transpose

Memory Bus Addressing

- To fully utilize the bus, GPUs combine the accesses of multiple threads into fewer requests when possible
- Assume that the memory bus is 32-bytes (256-bits) wide
- The byte-addressable bus must make accesses that are aligned to the bus width, so the bottom 5 bits are masked off
  - all data in the range `0x00001220` to `0x0000123F` is returned on the bus
  - in this case, 4 bytes are useful, and 28 bytes are wasted

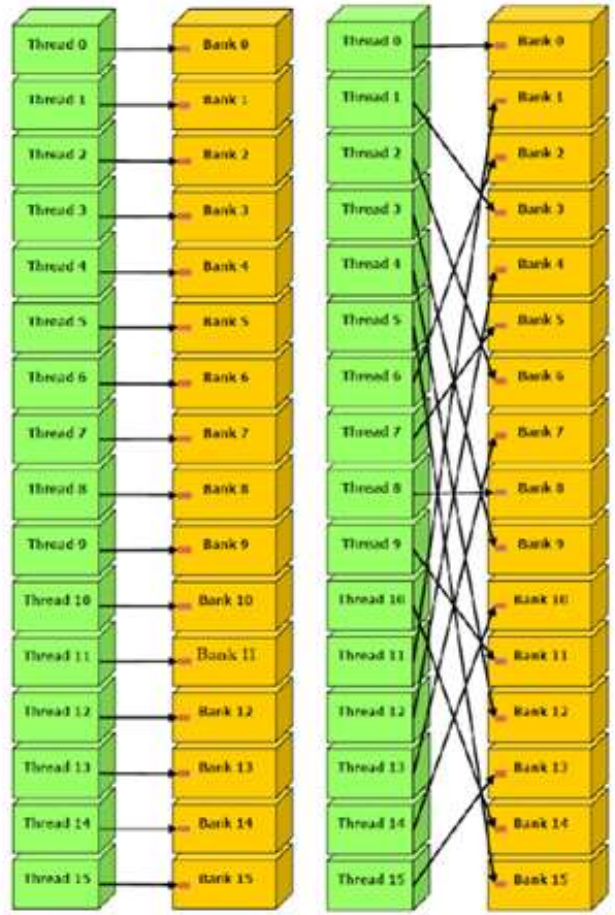


Coalescing Memory Accesses

- Coalesced memory access or memory coalescing
  - refers to combining multiple memory accesses into a single transaction
- Example: NVIDIA K20 GPUs
  - every successive 128 bytes (32 single precision words) memory can be accessed by a warp (32 consecutive threads) in a single transaction

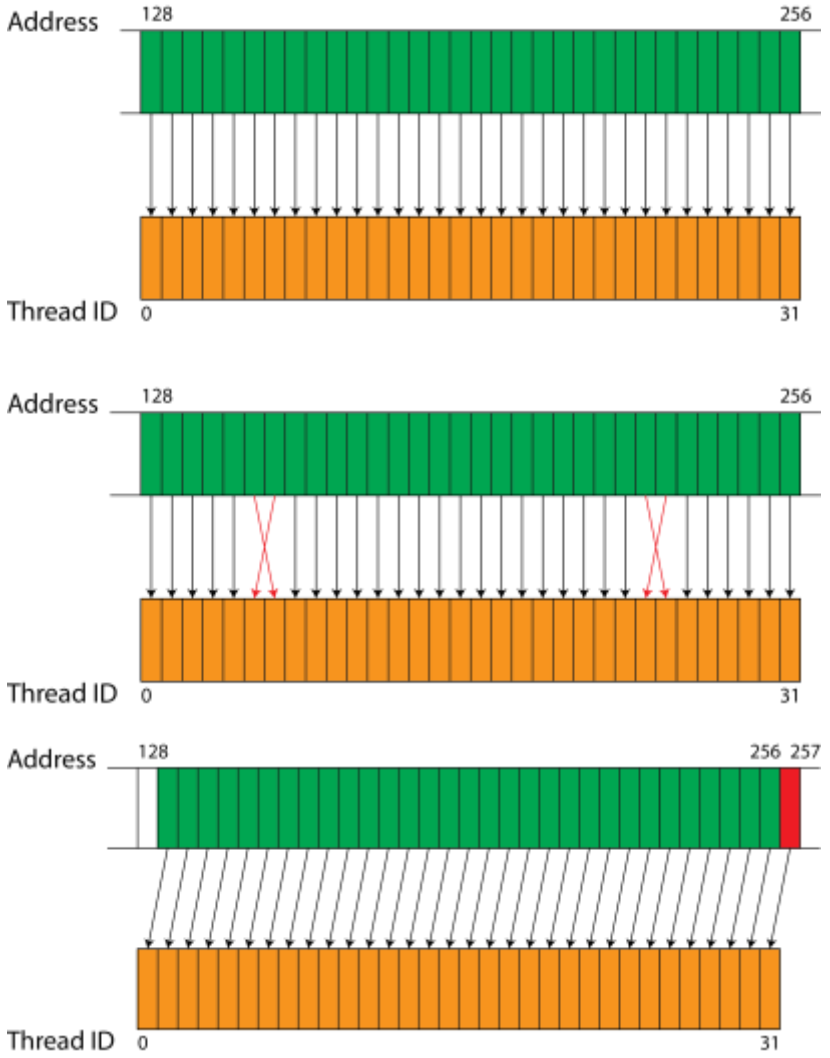
Memory Coalescing & Memory Banks

- Memory coalescing
  - ideally get work-item `i` to access `data[i]` and work-item `j` to access `data[j]` at the same time
- Memory bank
  - local memory is divided into equally sized memory modules, called banks that can be accessed simultaneously
  - any memory load or store of n addresses that spans n distinct memory banks can be serviced simultaneously, yielding an effective bandwidth that is n times as high as the bandwidth of a single bank
- Memory alignment
  - padding arrays to keep everything aligned to multiples of 16, 32 or 64 bytes



Memory Coalescing

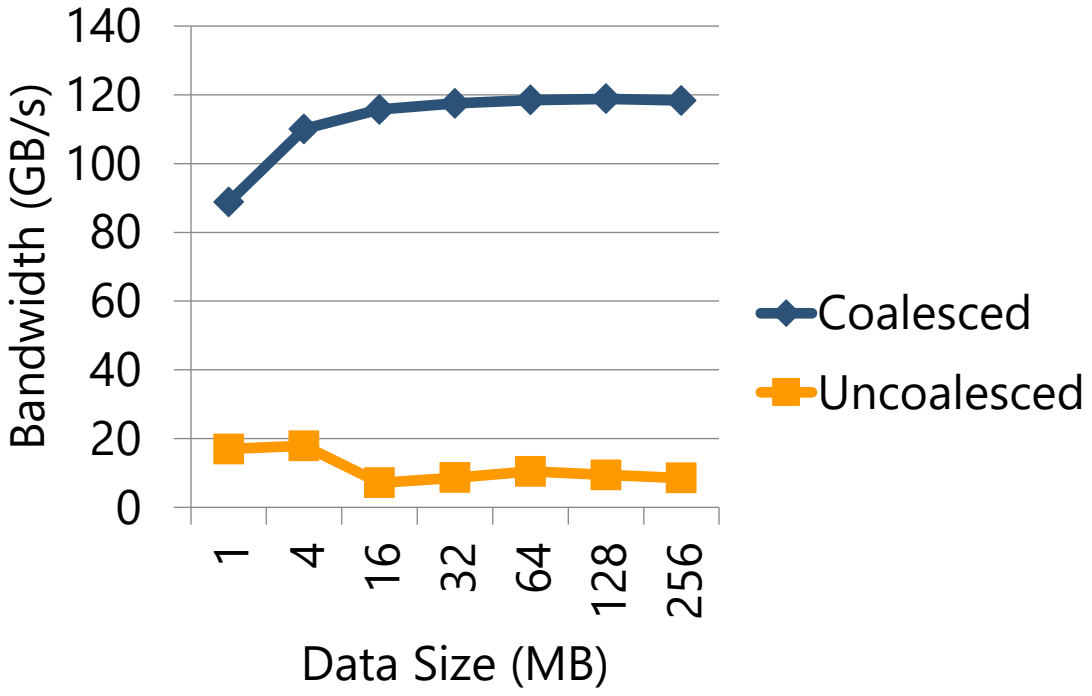
- Sequential and aligned access
  - 32 consecutive threads access 32 consecutive words
  - the memory access is coalesced
- Aligned but nonconsecutive sequential access
  - memory access is not sequential, but aligned
  - in modern GPUs, such access pattern can still be combined into a single transaction
- Unaligned memory access
  - the memory accessed is sequential but misaligned
  - two transactions are required
    - to load the first 31 words and another transaction to load the last word



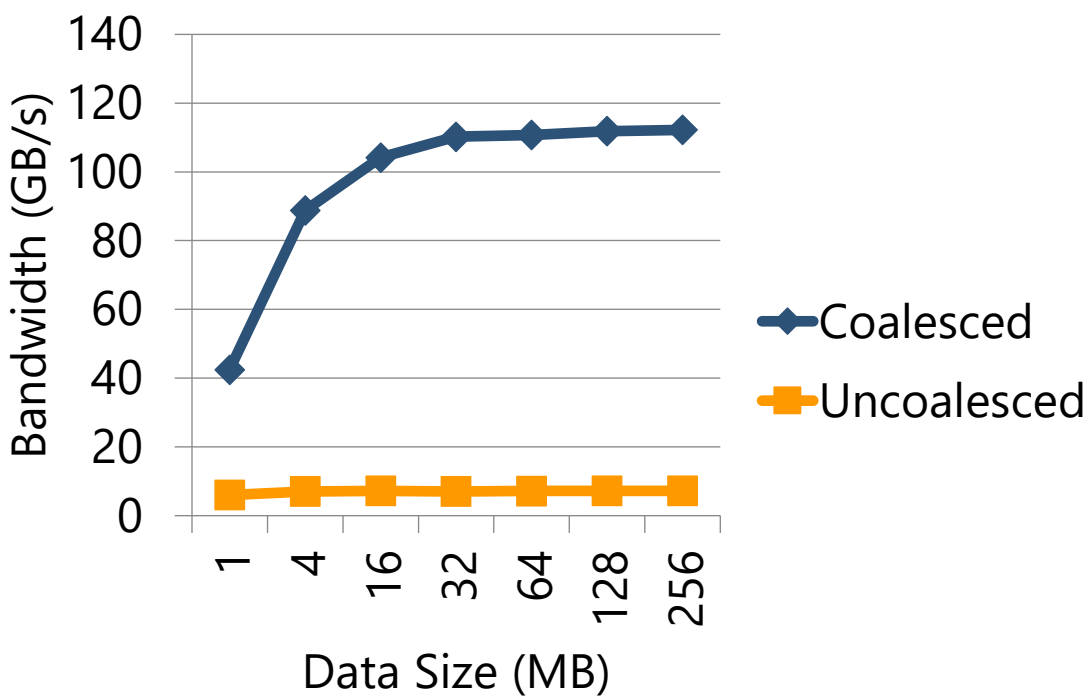
Coalescing Memory Accesses

- Global memory performance for a simple data copying kernel of entirely coalesced and entirely non-coalesced accesses on

ATI Radeon 5870



NVIDIA GTX 285



Private Memory

- Managing the memory hierarchy is one of the most important things to get right to achieve good performance
- Private memory
  - a very scarce resource, only a few tens of 32-bit words per work-Item at most
  - if you use too much it spills to global memory or reduces the number of work-Items that can be run at the same time, potentially harming performance
  - think of these like registers on the CPU

Local Memory

- Tens of KBytes per Compute Unit
  - multiple work-groups will be running on each CU
  - thus, only a fraction of the total local memory size is available to each work-group
- Assume up to 10 KBytes of local memory per work-group
  - kernels are responsible for transferring data between local and global/constant memories
    - e.g.: `async_work_group_copy()` and `async_workgroup_strided_copy()`
- Use local memory to hold data that can be reused by all work-items in a work-group
- Access patterns to local memory affect performance in a similar way to accessing global memory
  - think about coalescence & bank conflicts

Local Memory Limitations

- Local memory doesn't always help
  - CPUs don't have special hardware for it
  - this can mean excessive use of local memory might slow down kernels on CPUs
  - GPUs now have effective on-chip caches which *may* provide much of the benefit of local memory but without programmer intervention

Memory Consistency

- OpenCL uses a relaxed consistency memory model.
  - the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times
- Within a work-item
  - memory has load/store consistency to the work-item's private view of memory, i.e. it sees its own reads and writes correctly
- Within a work-group
  - local memory is consistent between work-items at a barrier
- Global memory is consistent within a work-group at a barrier, but not guaranteed across different work-groups
- Consistency of memory shared between commands (e.g. kernel invocations) is enforced by synchronization (barriers, events, in-order queue)

OpenCL Atomics

- Atomic: operations that at any point, and from any perspective, have either occurred completely, or not at all
  - guarantee no interference between threads during single memory location modification
- Since OpenCL 2.0
  - atomic initialization: `atomic_init()`
  - load and store: `atomic_load()` and `atomic_store()`
  - arithmetic, logical, min/max operations: `atomic_fetch_...()` function family
  - comparison & exchange: `atomic_exchange()`,`atomic_compare_exchange_...()`
- Memory orders associated with atomic operations may constrain the visibility of loads and stores with respect to the atomic operations

OpenCL Atomics Example & Memory Ordering

```
__global atomic_uint *counter;
atomic_init(counter, 0);    //initialize variable with zero
uint old_val = atomic_fetch_add_explicit(counter, 1, memory_order_relaxed, memory_scope_device);
```

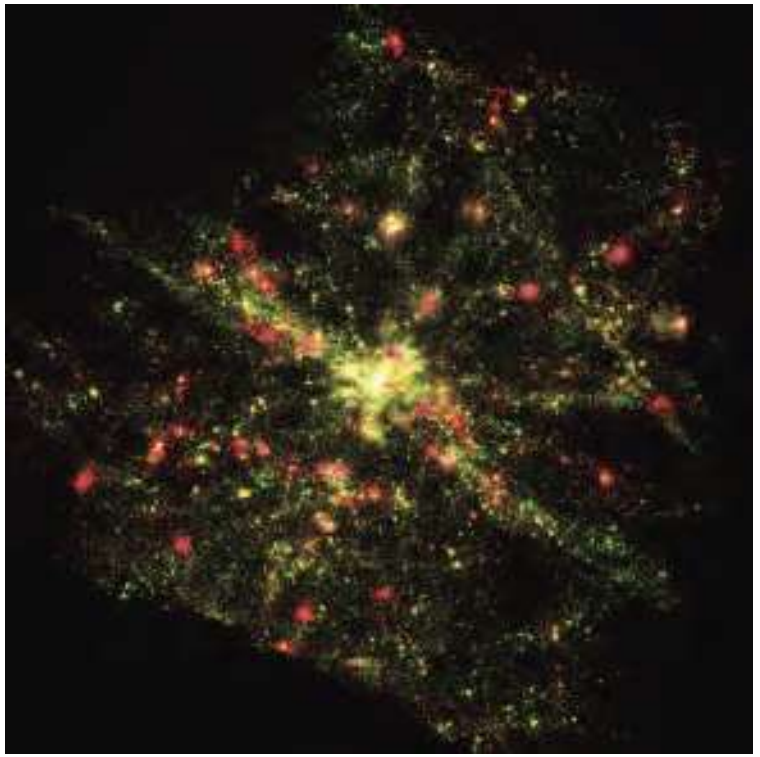
- Memory ordering
  - memory_order_relaxed: no additional memory synchronization, only atomicity is guaranteed
  - memory_order_acquire: memory fence is inserted right before atomic operation; all write results of other work-items within operation scope become visible to current work-item before atomic operation starts
  - memory_order_release: memory fence is inserted right after atomic operation; write results of the current work-item immediately become visible to others once atomic operation finishes
  - memory_order_acq_rel: both acquire and release fences are inserted
  - memory_order_seq_cst: all atomic accesses are serialized into single global sequence within a given scope
- Memory scope
  - memory_scope_work_item
  - memory_scope_work_group
  - memory_scope_device
  - memory_scope_all_svm_devices

OpenCL Kernels Restrictions

- Derived from ISO C99
  - preprocessing directives defined by C99 are supported
  - with some limitations
- Restrictions
  - recursion is not supported
  - pointers to functions are not allowed
  - pointers to pointers allowed within a kernel, but not as an argument to a kernel invocation
  - bit-fields are not supported
  - variable length arrays and structures are not supported

OpenCL Code Example: N-Body Simulation (1)

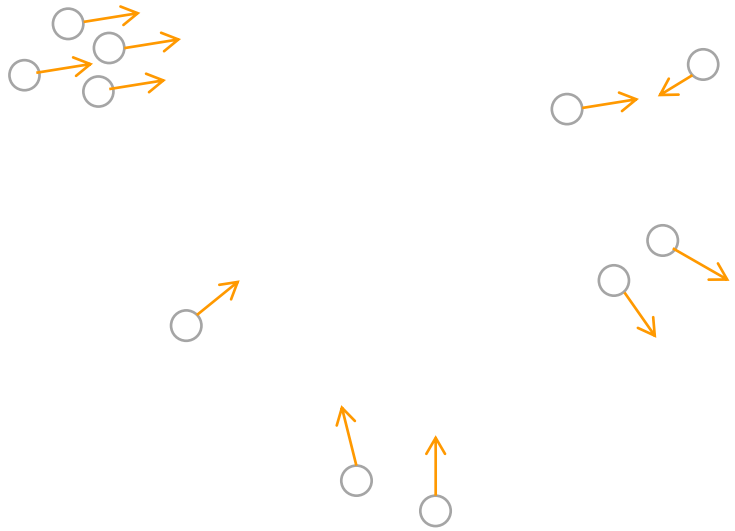
- An informal statement of the problem:
  - *Given only the present positions and velocities of a group of celestial bodies, predict their motions for all future time and deduce them for all past time*
- Consider n point masses m_1, m_2, ... m_n in a 3D space
  - suppose that the force of attraction experienced between each pair of particles is Newtonian
  - initial positions in space and initial velocities are specified for every particle at some present instant
  - determine the position of each particle at every future (or past) moment of time



- In mathematical terms, this means to find a global solution of the initial value problem for the differential equations describing the N-body problem

OpenCL Code Example: N-Body Simulation (2)

- Let's make it even simpler
  - we have n bodies
  - we should calculate $(n-1)^2$ pairwise interactions
- Smarter solutions use grid, nested grid, tree
  - e.g., Gadget2, Flash, etc.
- We just focus on a raw $O(n^2)$ method
  - brute force approach



OpenCL Code Example: N-Body Simulation (3)

- Data structure for body
  - simple AoS formulation
- Host code omitted
  1. Platform, device, command queue
  2. Program, Kernel
  3. Send to GPU
  4. Execute the kernel
     - simple 1D kernel
     - one work-item for each particle
  5. Read data back from the GPU to the CPU

```
typedef struct {
  double m;     // the mass of the body
  float3 pos; // the position in space
  float3 v;   // the velocity of the body
} body;
```

Kernel

```
__kernel void n_body(__global body* B_read, __global body* B_write, int size)
{
    uint gid = get_global_id(0);
    if(gid >= size) return;

    // set forces to zero
    float3 F = (float3)(0.0f, 0.0f, 0.0f);
    body bgid = B_read[gid];
    for(int k=0; k<size; k++) {
        body bk = B_read[k];
        float3 dist = bk.pos - bgid.pos;
        double r = fabs(dist);
        double f = (gid != k) ? 0 :   (B_read[gid].m * B_read[k].m) / (r*r);
        float3 cur = normalize(dist) * f;
        F = F + cur;
    }
    B_write[gid].m = bgid.m;
    B_write[gid].v = bgid.v + (F / bgid.m);
    B_write[gid].pos = bgid.pos + bgid.v;
}
```

N-Body Problem: Can we do better

- Is this kernel memory-bounded or computation-bounded?
- How can we improve memory access?
- Can we use local memory?

N-Body Simulation: Improving Memory Access

- Use SoA!
- Use tiling/blocking!
  - pre-load a “tile of particles” on local memory
  - tile size matches the local memory size

Optimized N-Body (1)

```
__kernel void nbody_sim(
    __global float4* pos , __global float4* vel,
    int numBodies, float deltaTime, float epsSqr,
    __local float4* localPos,
    __global float4* newPosition,
    __global float4* newVelocity)
{
    unsigned int tid = get_local_id(0);
    unsigned int gid = get_global_id(0);
    unsigned int localSize = get_local_size(0);

    // Number of tiles we need to iterate
    unsigned int numTiles = numBodies / localSize;

    // position of this work-item
    float4 myPos = pos[gid];
    float4 acc = (float4)(0.0f, 0.0f, 0.0f, 0.0f);
```

Optimized N-Body (2)

```
for(int i = 0; i < numTiles; ++i) {
  // load one tile into local memory
  int idx = i * localSize + tid;
  localPos[tid] = pos[idx];
  // Synchronize to make sure data is available for comp.
  barrier(CLK_LOCAL_MEM_FENCE);

  // calculate acceleration effect due to each body
  // a[i->j] = m[j] * r[i->j] / (r^2 + epsSqr)^(3/2)
  for(int j = 0; j < localSize; ++j) {
    // calculate acceleration caused by particle j on i
    float4 r = localPos[j] - myPos;
    float distSqr = r.x * r.x  +  r.y * r.y  +  r.z * r.z;
    float invDist = 1.0f / sqrt(distSqr + epsSqr);
    float invDistCube = invDist * invDist * invDist;
    float s = localPos[j].w * invDistCube;
    // accumulate effect of all particles
    acc += s * r;
  }
  // Synchronize so that next tile can be loaded
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

Optimized N-Body (3)

```
  float4 oldVel = vel[gid];

  // updated position and velocity
  float4 newPos = myPos +
                  oldVel * deltaTime +
                  acc *0.5f* deltaTime * deltaTime;
  newPos.w = myPos.w;
  float4 newVel = oldVel + acc * deltaTime;

  // write to global memory
  newPosition[gid] = newPos;
  newVelocity[gid] = newVel;
}
```

Lab Exercise: Matrix Transpose

- The transpose of a matrix is an operator which flips a matrix over its diagonal
  - it switches the row and column indices of the matrix A by producing another matrix, denoted by A^T
- Example

$$A = \begin{pmatrix} 2 & 4 & 8 \\ 3 & 2 & 0 \\ 5 & 3 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad A^T = \begin{pmatrix} 2 & 3 & 5 & 0 \\ 4 & 2 & 3 & 1 \\ 8 & 0 & 1 & 0 \end{pmatrix}$$

Matrix Transpose – Kernel Code

```
__kernel void transpose_naive(
    __global float *odata, __global float* idata,
    int width, int height)
{
    unsigned int xIndex = get_global_id(0);
    unsigned int yIndex = get_global_id(1);

    if (xIndex < width && yIndex < height)
    {
        unsigned int index_in  = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        odata[index_out] = idata[index_in];
    }
}
```

Lab Exercise

- Write an optimized matrix transpose
  - fully coalesced memory access
  - no bank conflicts
  - use of local memory

GPU Reduction

High Performance Computing, Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

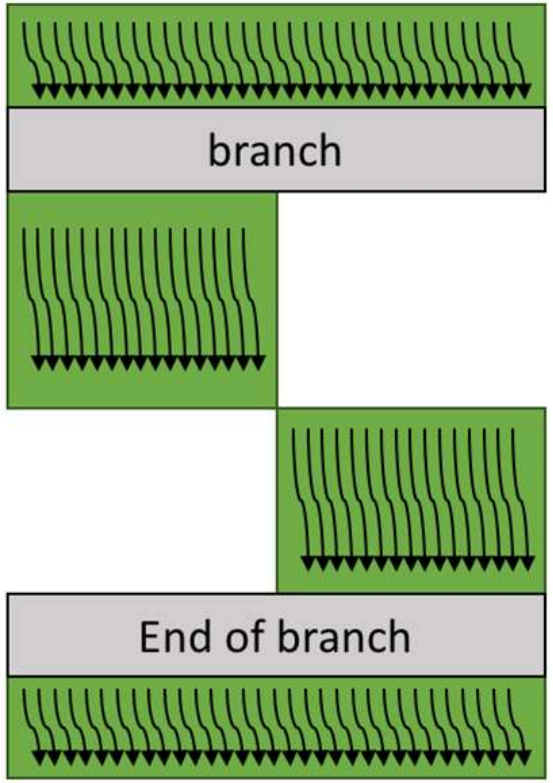
Outline

- Recap
  - GPU threads
  - GPU optimization issues
- OpenCL synchronization
- Example: Reduction

Divergent Control Flow

```
if(gid < 16){
  A
}
else {
  B
}
```

- Work-item divergence
  - What happens when work-items branch?
  - SIMD-like data parallel model: SIMT
  - both paths (`if-else`) may need to be executed (branch divergence)
  - avoid where possible (non-divergent branches are termed uniform)



Occupancy

- Occupancy is a measure of how active you're keeping each PE
  - occupancy measurements of >0.5 are good (>50% active)
  - occupancy = active warps(wavefront) / max active warps(wavefront)
- Potential occupancy limiters
  - Register and local memory usage
  - Work-group size
  - Unbalanced workload within and across work-groups

Optimization Issues

- Work-item divergence
  - What happens when work-items branch?
  - SIMT data parallel model
  - Both paths (if-else) may need to be executed (branch divergence)
    - avoid where possible (non-divergent branches are termed uniform)
- Efficient access to memory
  - Memory coalescing: Ideally get work-item i to access data[i] and work-item j to access data[j] at the same time etc.
  - Memory alignment: Padding arrays to keep everything aligned to multiples of 16, 32 or 64 bytes
- Number of work-items and work-group sizes
  - IMore is better, but diminishing returns, and there is an upper limit
    - Each work item consumes PE finite resources (registers etc)

Efficient Reduction on GPU

- Fundamental data-parallel primitive
- There are many kinds of reductions depending on
  - the type of data being reduced
  - the operator which is being used to perform the reduction (+, *, max, min, …)
  - memory layout (segmented, ..)
- Consider a sum reduction on float

```
float reduce_sum(float* input, int length){
  float accumulator = input[0];
  for(int i = 1; i < length; i++)
    accumulator += input[i];
  return accumulator;
}
```

Simple Parallel Reduction

- A reduction can be carried out in three steps:
  1. Each work-item sums its private values into a local array indexed by the work-item's local id
  2. When all the work-items have finished, one work-item sums the local array into an element of a global array (indexed by work-group id)
  3. When all work-groups have finished the kernel execution, the global array is summed on the host

Associativity and Commutativity

- The reduction operator may show some flexibility in terms of what order the operations must be performed
- Associative Property
  - If an operator allows us to regroup the operations and still get the same result
  - E.g. ((10 + 20) + 30) = (10 + (20 + 30))
- Commutative Property
  - If an operator allows us to reorder the operations and still get the same result
  - E.g. ((10 + 20) + 30) = ((30 + 10) + 20)

Towards a GPU-Optimized Parallel Reduction

Mark Harris

Optimizing Parallel Reduction in CUDA

NVIDIA
https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Reduction – Version 1 – Interleaved Addressing

- At each stage of the reduction tree, we'll be loading and storing partial reductions as we compute
- It's crucial to use local memory to communicate between work-items in the work group
- We'll then execute the reduction tree by using a for loop in conjunction with OpenCL barriers

Reduction #1

```
__kernel void reduce0(__global T* g_idata, __global T* g_odata, unsigned int n, __local T* sdata)
{
    // load local memory
    unsigned int tid = get_local_id(0);
    unsigned int i = get_global_id(0);

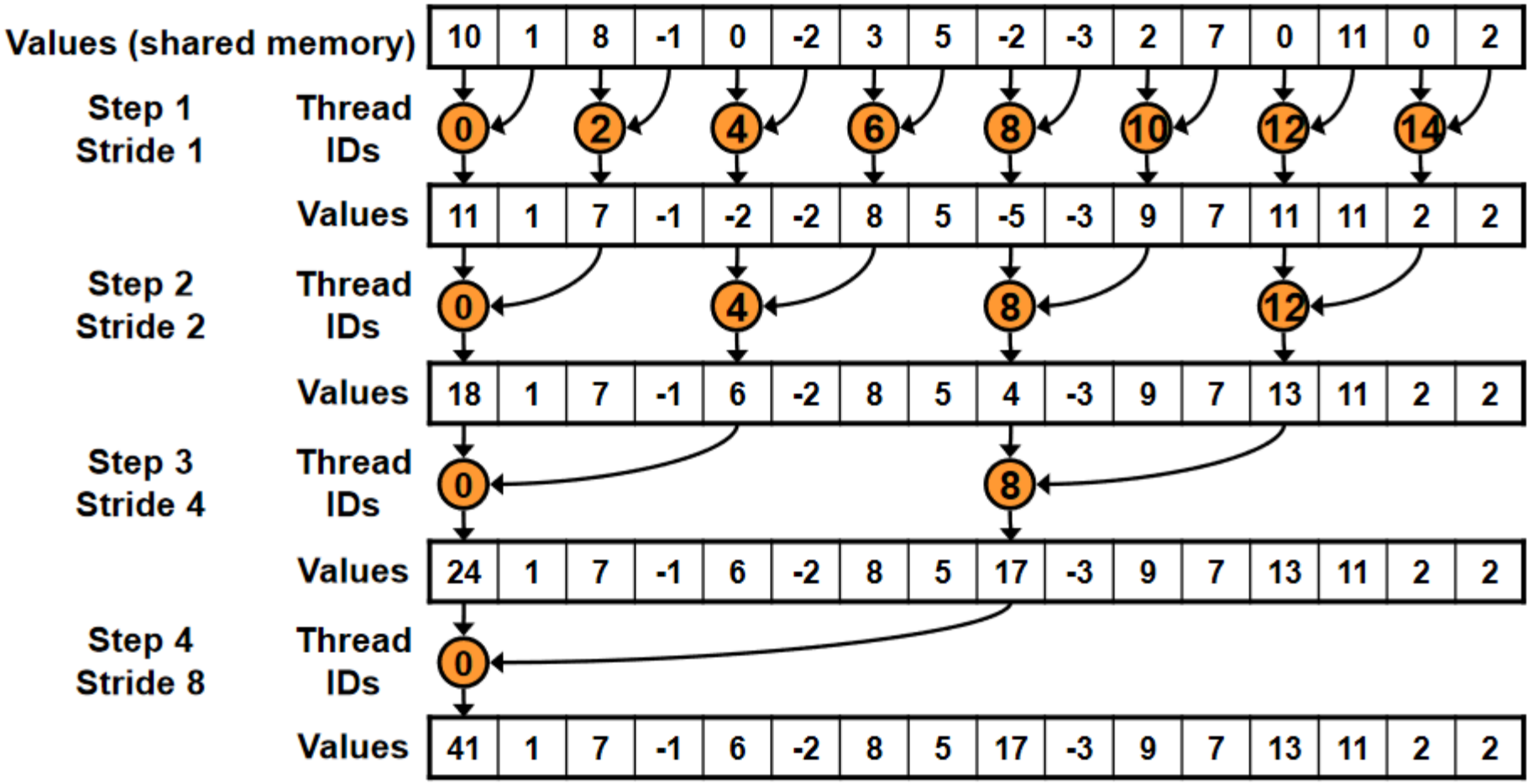
    sdata[tid] = (i < n) ? g_idata[i] : 0;

    barrier(CLK_LOCAL_MEM_FENCE);

    // do reduction in local memory
    for (unsigned int s = 1; s < get_local_size(0); s *= 2) {
        if ((tid % (2 * s)) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

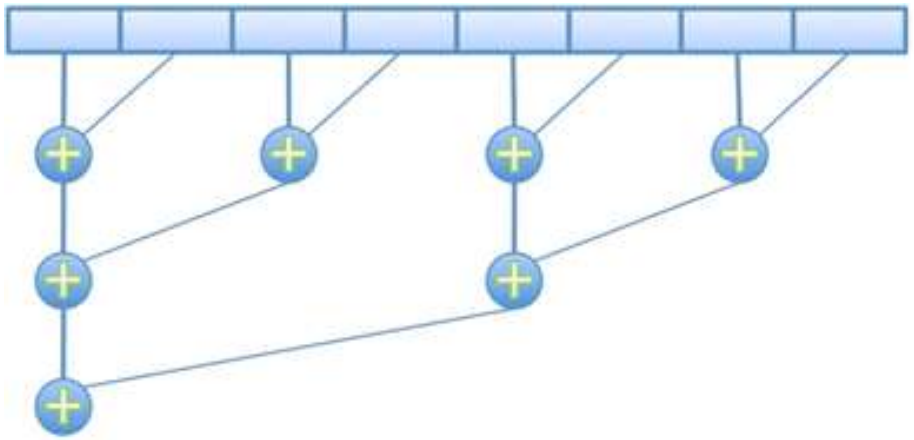
    // write result for this block to global mem
    if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
}
```

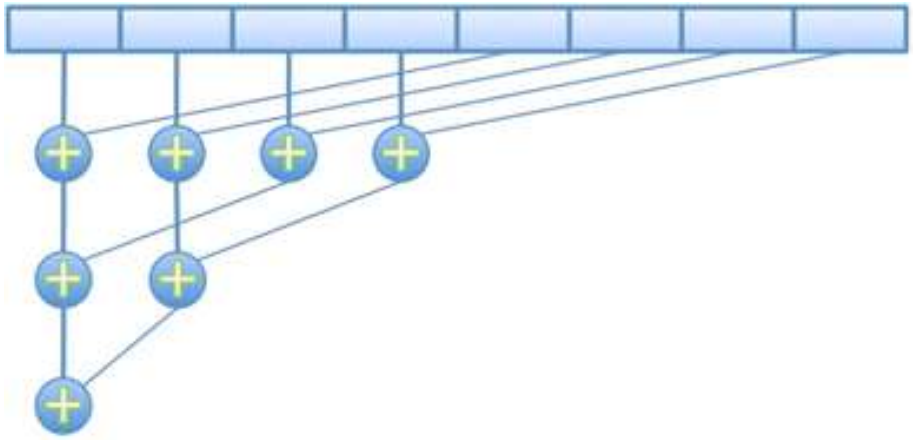
Reduction #1 – Interleaved Addressing

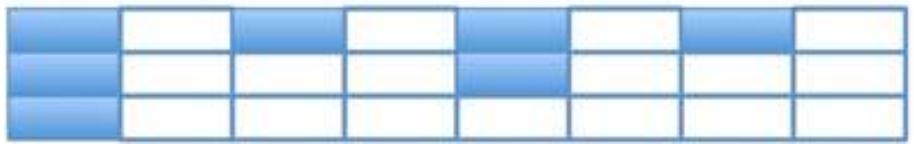


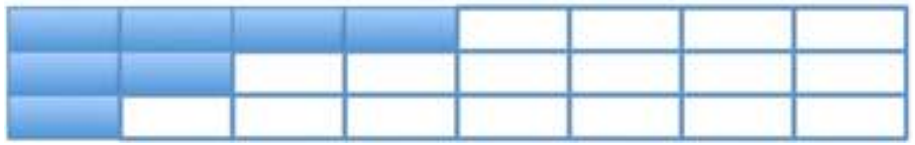
Taking Advantage of Commutativity

- What version is faster? Why?

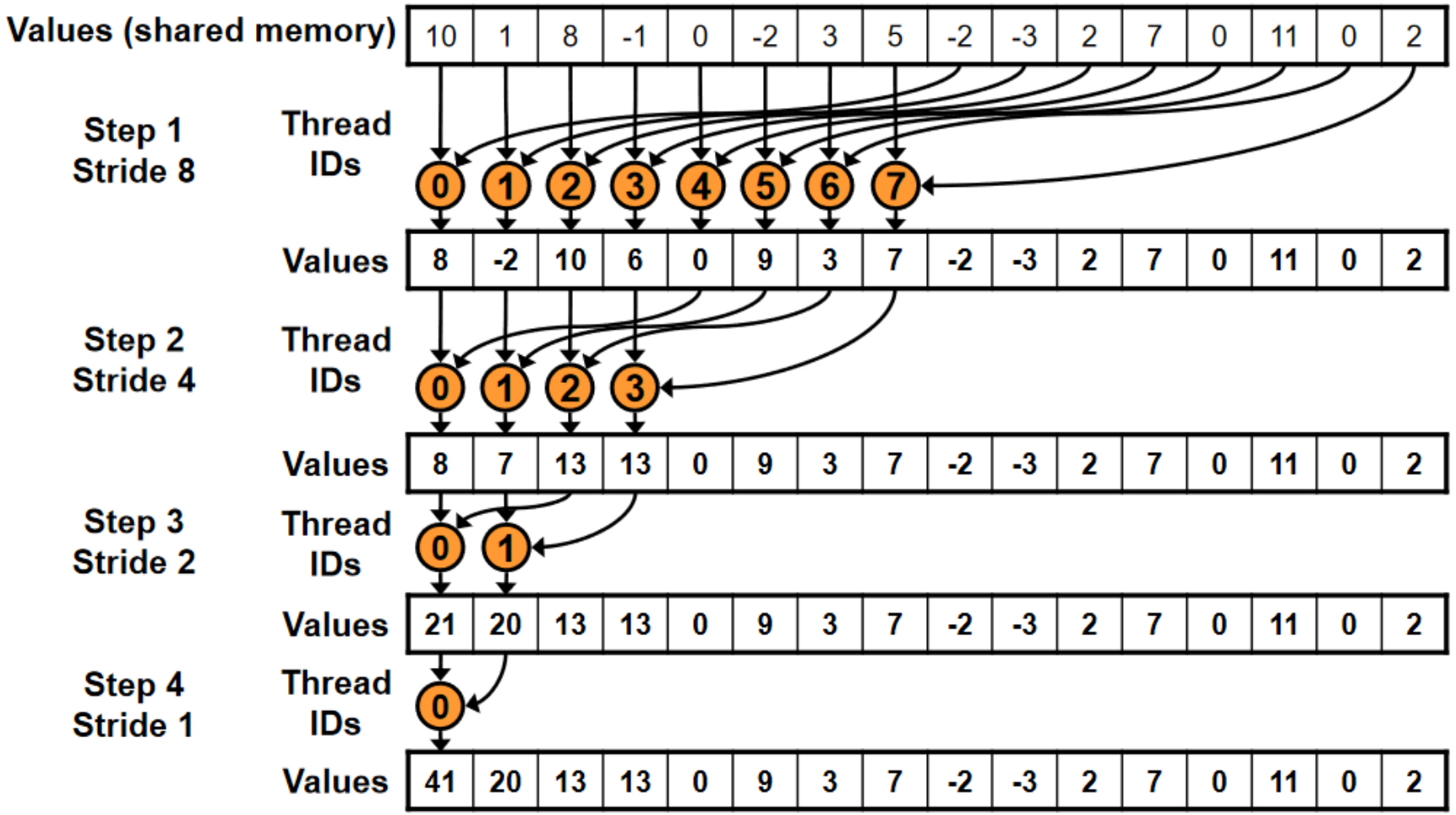








Reduction: Removing Non-Divergent Branch



Toward Reduction #2: Removing Non-Divergent Branch

- Divergent branch in inner loop

```
for (unsigned int s = 1; s < get_local_size(0); s *= 2) {
  if ((tid % (2 * s)) == 0) {
    sdata[tid] += sdata[tid + s];
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

- Replace with strided index and non-divergent branch

```
for (unsigned int s = 1; s < get_local_size(0); s *= 2) {
  int index = 2 * s * tid;
  if (index < get_local_size(0)) {
    sdata[index] += sdata[index + s];
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

Reduction #2

```
__kernel void reduce1(__global T* g_idata, __global T* g_odata, unsigned int n, __local T* sdata)
{
    // load local memory
    unsigned int tid = get_local_id(0);
    unsigned int i = get_global_id(0);

    sdata[tid] = (i < n) ? g_idata[i] : 0;
    barrier(CLK_LOCAL_MEM_FENCE);

    // do reduction in local mem
    for (unsigned int s = 1; s < get_local_size(0); s *= 2) {
        int index = 2 * s * tid;
        if (index < get_local_size(0)) {
            sdata[index] += sdata[index + s];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
}
```

Reduction: Sequential Addressing

- Strided indexing in inner loop

```
for (unsigned int s = 1; s < get_local_size(0); s *= 2) {
  int index = 2 * s * tid;
  if (index < get_local_size(0)) {
    sdata[index] += sdata[index + s];
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

- Replace with reversed loop and ID-based indexing

```
for (unsigned int s = get_local_size(0) / 2; s > 0; s >>= 1) {
  if (tid < s) {
    sdata[tid] += sdata[tid + s];
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

Reduction #3

```
__kernel void reduce2(__global T* g_idata, __global T* g_odata, unsigned int n, __local T* sdata)
{
    // load local memory
    unsigned int tid = get_local_id(0);
    unsigned int i = get_global_id(0);

    sdata[tid] = (i < n) ? g_idata[i] : 0;

    barrier(CLK_LOCAL_MEM_FENCE);

    // do reduction in local memory
    for (unsigned int s = get_local_size(0) / 2; s > 0; s >>= 1)
    {
        if (tid < s)
        {
            sdata[tid] += sdata[tid + s];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
}
```

Problem: Idle Threads

- Half of the threads are idle on first loop iteration!

```
for (unsigned int s = get_local_size(0) / 2; s > 0; s >>= 1) {
  if (tid < s) {
    sdata[tid] += sdata[tid + s];
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

Problem: Idle Threads

- Idea: First add during load
  - halve the number of workgroup
  - replace the single load with two loads
    - the two values added before local memory fence

```
unsigned int tid = get_local_id(0);
unsigned int i = get_group_id(0) * (get_local_size(0) * 2) + get_local_id(0);

sdata[tid] = (i < n) ? g_idata[i] : 0;
if (i + get_local_size(0) < n)
  sdata[tid] += g_idata[i + get_local_size(0)];

barrier(CLK_LOCAL_MEM_FENCE);
```

Reduction #4 - Version with n/2 threads

```
__kernel void reduce3(__global T* g_idata, __global T* g_odata, unsigned int n, __local T* sdata)
{
    // perform first level of reduction,
    // reading from global memory, writing to shared memory
    unsigned int tid = get_local_id(0);
    unsigned int i = get_group_id(0) * (get_local_size(0) * 2) + get_local_id(0);

    sdata[tid] = (i < n) ? g_idata[i] : 0;
    if (i + get_local_size(0) < n)
        sdata[tid] += g_idata[i + get_local_size(0)];

    barrier(CLK_LOCAL_MEM_FENCE);

    // do reduction in shared mem
    for (unsigned int s = get_local_size(0) / 2; s > 0; s >>= 1)
    {
        if (tid < s)
        {
            sdata[tid] += sdata[tid + s];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
}
```

two loads and first add of the reduction

Instruction Bottleneck

- Still far from the bandwidth bound
  - reduction has low arithmetic intensity
- Improve ILP
  - address arithmetic and loop overhead
- Solution: apply loop unrolling

Unrolling the Last Warp

- As reduction proceeds, # “active” threads decreases
  - when s <= 32, we have only one warp left
  - instructions are SIMD synchronous within a warp
- That means when s <= 32
  - we don’t need to `barrier(CLK_LOCAL_MEM_FENCE);`
  - we don’t need “`if (tid < s)`” because it doesn’t save any work
- Solution: Loop unrolling for the last 6 iterations of the inner loop

Reduction #5: Unrolling the Last Warp

- Unroll the last warp to avoid synchronization where it isn't needed
  - saves useless work in all warps, not just the last one
  - without unrolling, all warps execute every iteration of the for loop and if statement

```
for (unsigned int s = get_local_size(0) / 2; s > 32; s >>= 1) {
  if (tid < s){
    sdata[tid] += sdata[tid + s];
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}

if (tid < 32) {
  if (blockSize >= 64) { sdata[tid] += sdata[tid + 32]; }
  if (blockSize >= 32) { sdata[tid] += sdata[tid + 16]; }
  if (blockSize >= 16) { sdata[tid] += sdata[tid + 8]; }
  if (blockSize >= 8) { sdata[tid] += sdata[tid + 4]; }
  if (blockSize >= 4) { sdata[tid] += sdata[tid + 2]; }
  if (blockSize >= 2) { sdata[tid] += sdata[tid + 1]; }
}
// no need to add a barrier(CLK_LOCAL_MEM_FENCE)
```

Reduction #5 : Unrolling the Last Warp

```
__kernel void reduce4(__global T* g_idata, __global T* g_odata, unsigned int n, __local volatile T* sdata) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0) * 2) + get_local_id(0);

  sdata[tid] = (i < n) ? g_idata[i] : 0;
  if (i + get_local_size(0) < n)
    sdata[tid] += g_idata[i + get_local_size(0)];

  barrier(CLK_LOCAL_MEM_FENCE);

  for (unsigned int s = get_local_size(0) / 2; s > 32; s >>= 1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }

  if (tid < 32) {
    if (blockSize >= 64) { sdata[tid] += sdata[tid + 32]; }
    if (blockSize >= 32) { sdata[tid] += sdata[tid + 16]; }
    if (blockSize >= 16) { sdata[tid] += sdata[tid + 8]; }
    if (blockSize >= 8) { sdata[tid] += sdata[tid + 4]; }
    if (blockSize >= 4) { sdata[tid] += sdata[tid + 2]; }
    if (blockSize >= 2) { sdata[tid] += sdata[tid + 1]; }
  }
  // write result for this block to global mem
  if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
}
```

volatile needed for correctness

skip the last 32 iterations

unroll the last warp

Complete Unrolling

- If we knew the number of iterations at compile time, we could completely unroll the reduction
- We assume
  - workgroup size limited to 512 threads
  - power-of-2 workgroup sizes
  - unroll for a fixed workgroup size
- Problem: we need to know the workgroup size at compile time
  - e.g., using a kernel macro
    - equivalent to `#define blockSize 128`

Reduction #6: Complete Unrolling

```
__kernel void reduce5(__global T* g_idata, __global T* g_odata, unsigned int n, __local volatile T* sdata) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0) * 2) + get_local_id(0);

  sdata[tid] = (i < n) ? g_idata[i] : 0;
  if (i + blockSize < n)
    sdata[tid] += g_idata[i + blockSize];

  barrier(CLK_LOCAL_MEM_FENCE);

  if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } barrier(CLK_LOCAL_MEM_FENCE); }
  if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } barrier(CLK_LOCAL_MEM_FENCE); }
  if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (blockSize >= 64) { sdata[tid] += sdata[tid + 32]; }
    if (blockSize >= 32) { sdata[tid] += sdata[tid + 16]; }
    if (blockSize >= 16) { sdata[tid] += sdata[tid + 8]; }
    if (blockSize >= 8) { sdata[tid] += sdata[tid + 4]; }
    if (blockSize >= 4) { sdata[tid] += sdata[tid + 2]; }
    if (blockSize >= 2) { sdata[tid] += sdata[tid + 1]; }
  }
  // write result for this block to global mem
  if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
}
```

Note: all code in yellow will be evaluated at compile time

Final Optimizations

- Algorithm cascading: combine sequential and parallel reduction
  - each thread loads and sums multiple elements into shared memory
  - tree-based reduction in shared memory
  - benefits
    - possibly better latency hiding with more work per thread
    - more threads per workgroup reduces levels in tree of recursive kernel invocations
    - high kernel launch overhead in last levels with few blocks

- Final version
  - this version adds multiple elements per thread sequentially
  - this reduces the overall cost of the algorithm while keeping the work complexity O(n) and the step complexity O(log n)
    - Brent's Theorem

```
unsigned int tid = get_local_id(0);
unsigned int i = get_group_id(0) * (get_local_size(0) * 2) + get_local_id(0);
unsigned int gridSize = blockSize * 2 * get_num_groups(0);
sdata[tid] = 0;

while (i < n) {
  sdata[tid] += g_idata[i];
  // check read out of bounds, optimized for powerOf2 sized arrays
  if (nIsPow2 || i + blockSize < n)
    sdata[tid] += g_idata[i + blockSize];
  i += gridSize;
}
```

Reduction #7: Final Version: Algorithm Cascading

```
__kernel void reduce6(__global T* g_idata, __global T* g_odata, unsigned int n, __local volatile T* sdata) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0) * 2) + get_local_id(0);
  unsigned int gridSize = blockSize * 2 * get_num_groups(0);
  sdata[tid] = 0;

  while (i < n) {
    sdata[tid] += g_idata[i];
    // check read out of bounds, optimized for powerOf2 sized arrays
    if (nIsPow2 || i + blockSize < n)
      sdata[tid] += g_idata[i + blockSize];
    i += gridSize;
  }

  barrier(CLK_LOCAL_MEM_FENCE);

  if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } barrier(CLK_LOCAL_MEM_FENCE); }
  if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } barrier(CLK_LOCAL_MEM_FENCE); }
  if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } barrier(CLK_LOCAL_MEM_FENCE); }

  if (tid < 32) {
    if (blockSize >= 64) { sdata[tid] += sdata[tid + 32]; }
    if (blockSize >= 32) { sdata[tid] += sdata[tid + 16]; }
    if (blockSize >= 16) { sdata[tid] += sdata[tid + 8]; }
    if (blockSize >= 8) { sdata[tid] += sdata[tid + 4]; }
    if (blockSize >= 4) { sdata[tid] += sdata[tid + 2]; }
    if (blockSize >= 2) { sdata[tid] += sdata[tid + 1]; }
  }

  if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
}
```

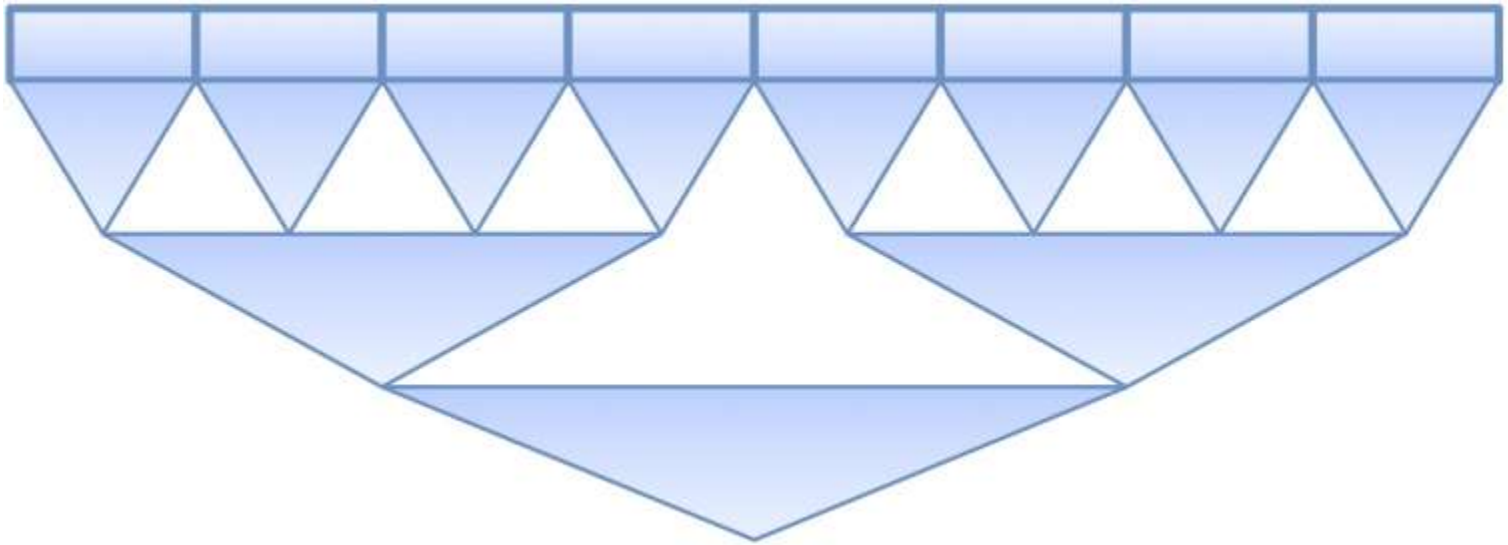
first level reduction
reading from global memory,
writing to shared memory

We reduce multiple elements per thread. The number is determined by the number of active thread blocks (via gridDim). More blocks will result in a larger gridSize and therefore fewer elements per thread

What is missing?

Multi-stage Reduction

- We saw a parallel reductions on vectors which are short enough to fit in a single work-group
- How can we do reduction of larger vectors?

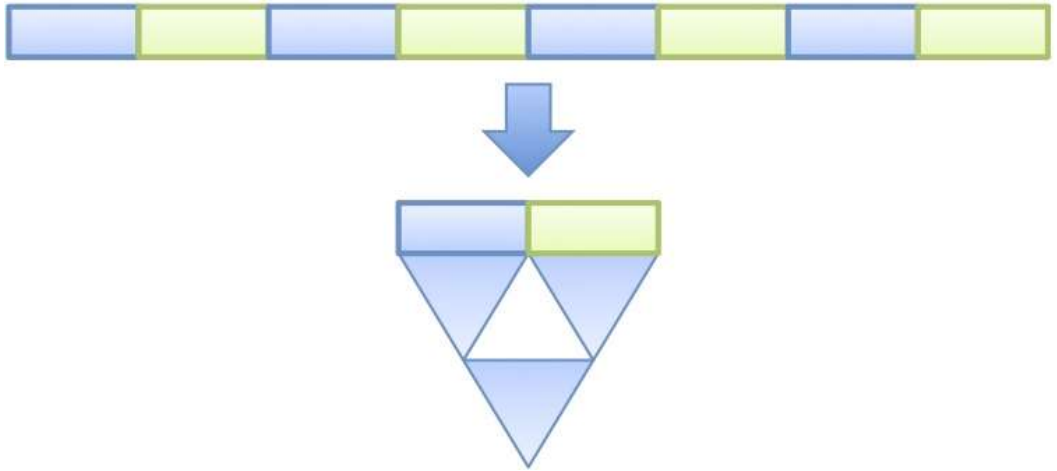


Multi-stage Reduction: Three Approaches

1. Recursive multi-stage reductions (tree based)
   - the results produced by each local reduction are gathered into a new vector, which is then reduced again
   - at each stage of the reduction, the amount of data remaining is reduced by a constant reduction factor.
   - e.g. if we worked with work-groups of 256 work-items, each stage would reduce the amount of data by a factor of 256. This style of reduction expresses the largest amount of parallelism, and it can be written to only take advantage of associativity, for operators which are not commutative. However, it can be less efficient.
2. Two-stage reductions
   - we express just enough parallelism to fill the machine, and then follow with a final global reduction
   - taking advantage of commutativity, we can then perform most of the work sequentially, which improves efficiency compared to the fully-parallel multi-stage reduction
   - additionally, we only have to launch two kernels per reduction
3. Reductions using atomics
   - use atomic memory operations to reduce the partial results from each local reduction
   - atomic transactions will limit you to the operators and data-types which are supported by the platform you're targeting

Two-stage Reduction

- The multi-stage reductions involves lots of synchronization and data-sharing amongst work-items in the work-group
- We can try to do as much of the reduction serially as possible, avoiding parallel reductions across the work-group as much as possible
  - Two-stage reduction, where the input is divided up into p chunks, where p is large enough to keep all of our processors busy



Lab Exercise

- Implement a parallel reduction in OpenCL
  - Experiment the same version on GPU and CPU

GPU Matrix Multiplication

High Performance Computing, Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

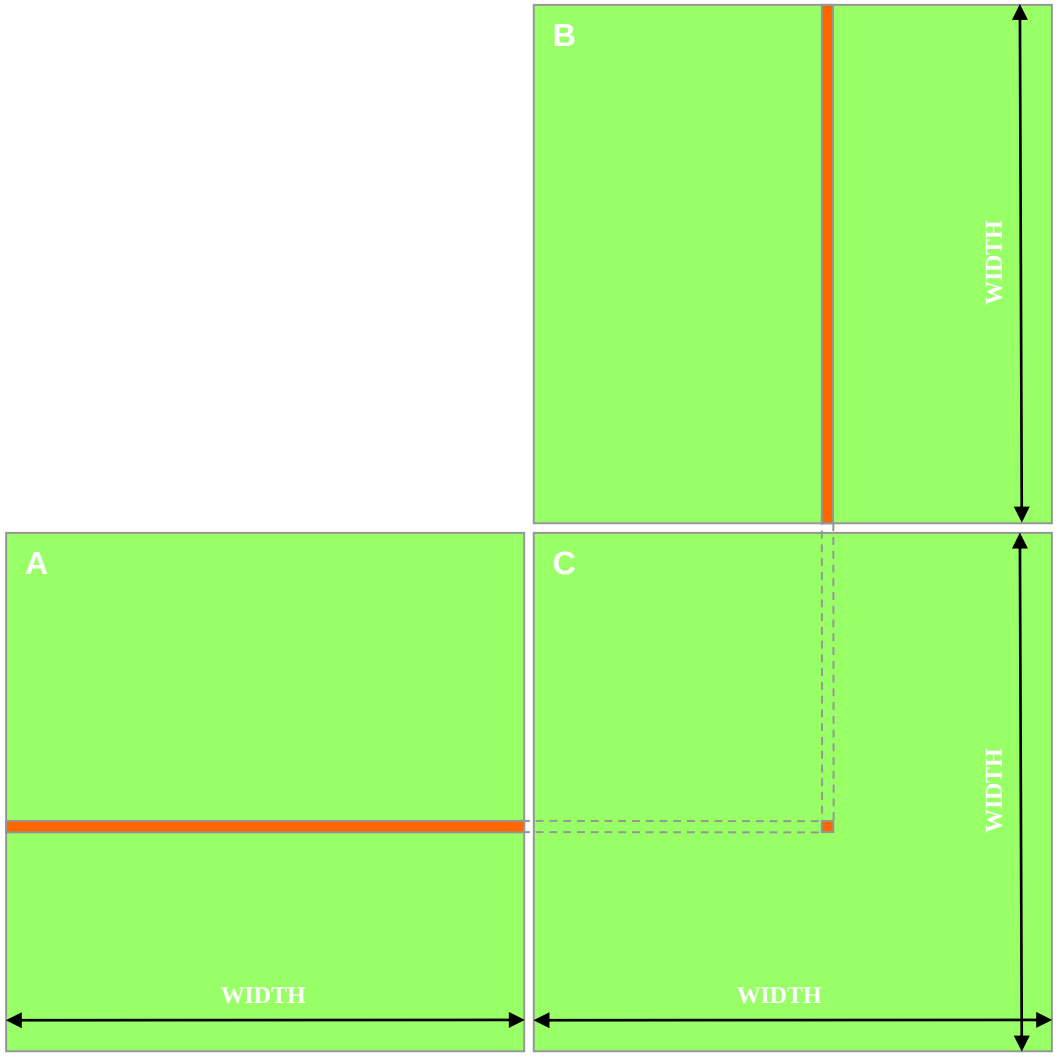
Outline

- Matrix Multiplication

```
for (int i=0; i<N; i++){
  for (int j=0; j<N; j++) {
    for (int k=0; k<N; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Matrix Multiplication Memory Layout

```
for (int i=0; i<N; i++){
  for (int j=0; j<N; j++) {
    for (int k=0; k<N; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```



Matrix Multiplication in OpenCL

- Two dimensions
- Global work size = $[C_{width} \times C_{height}]$
- We only show the kernel code

Matrix Multiplication

```
__kernel void matrixMul(
    __global float* A, __global float* B, __global float* C,
    int wA, int wB)
{
  int tx = get_global_id(0);
  int ty = get_global_id(1);

  float value = 0;
  for (int k = 0; k < wA; ++k)
  {
    float elementA = A[ty * wA + k];
    float elementB = B[k * wB + tx];
    value += elementA * elementB;
   }
   C[ty * wA + tx] = value;
}
```

Questions

- Why there is only 1 for loop in the OpenCL version and 3 for loops for the serial?
- What is the difference between
  - `get_global_id(0)` and `get_global_id(1)`
  - `get_global_id(0)` and `get_local_id(0)`
- Is there a way to replace `wA` and `wB`?
- What parameters do we use for the kernel invocation?

```
cl_int clEnqueueNDRangeKernel(
  cl_command_queue command_queue, cl_kernel kernel,
  cl_uint work_dim, const size_t *global_work_offset,
  const size_t *global_work_size,
  const size_t *local_work_size,
  cl_uint num_events_in_wait_list,
  const cl_event *event_wait_list, cl_event *event)
```

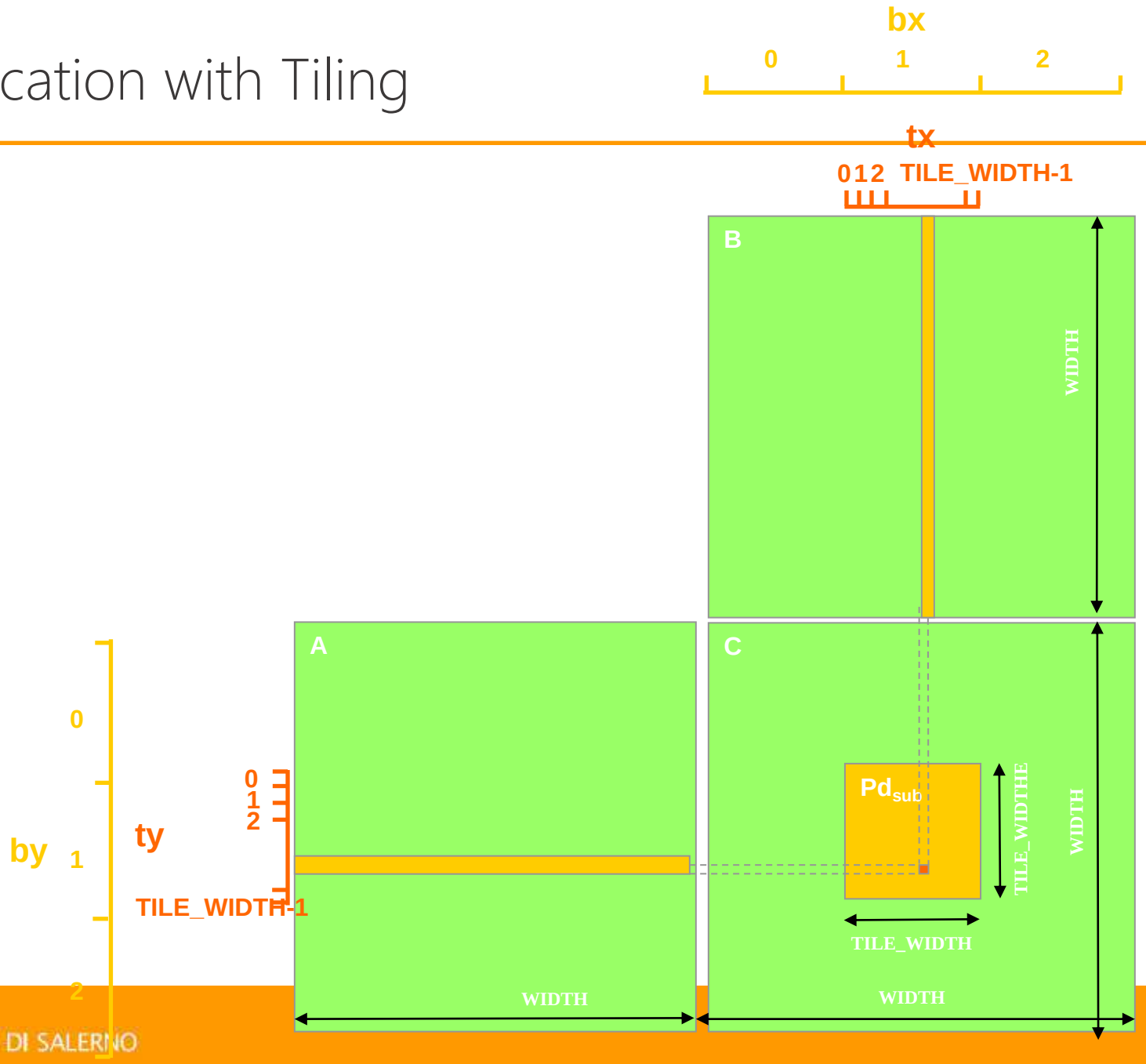
GPU Memory Speed Remarks

- Global vs local memory bandwidth
- The key is to use local memory and registers when possible

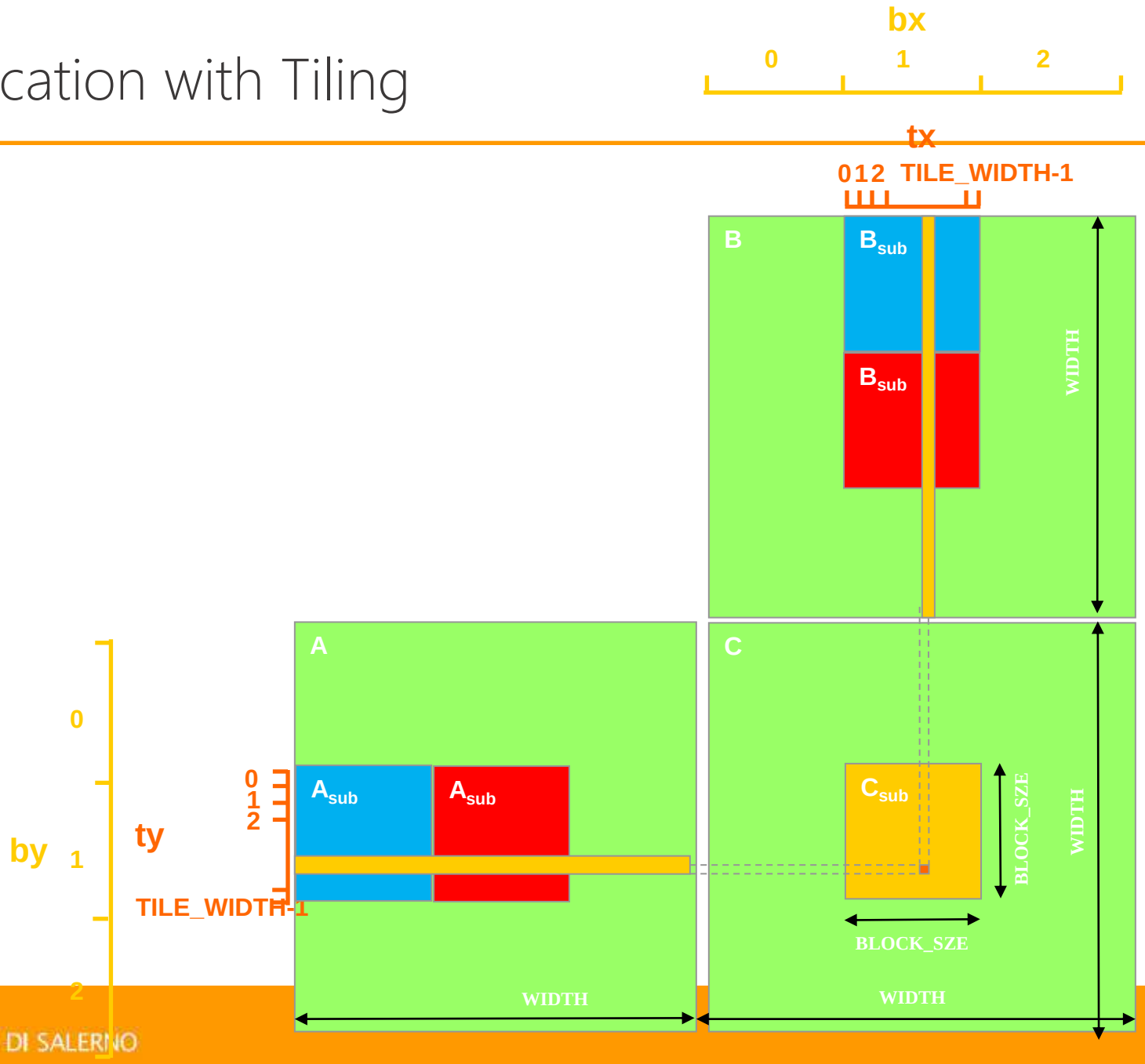
Tiling

- Split result matrix into smaller blocks
  - block as a two-dimensional tiling
  - improved performance
- Copy portion of matrix to local memory
  - maximize data reuse

Matrix Multiplication with Tiling



Matrix Multiplication with Tiling



Matrix Multiplication – Part 1

```
#define AS(i, j) As[j + i * BLOCK_SIZE]
#define BS(i, j) Bs[j + i * BLOCK_SIZE]

// Matrix multiplication on the device: C = A * B. uiWA is A's width and uiWB is B's width
__kernel void matrixMul( __global float* C, __global float* A, __global float* B,
                         __local float* As, __local float* Bs, int uiWA, int uiWB, int trueLocalSize1)
{
    // Block index
    int bx = get_group_id(0);
    int by = get_group_id(1);
    // Thread index
    int tx = get_local_id(0);
    int ty = get_local_id(1);
    // Index of the first sub-matrix of A processed by the block
    int aBegin = uiWA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + uiWA - 1;
    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * uiWB;
    // Csub is used to store the element of the block sub-matrix that is computed by the thread
    float Csub = 0.0f;
```

Matrix Multiplication – Part 2

```
    // Loop over all the sub-matrices of A and B, required to compute the block sub-matrix
    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {

        // Load the matrices from device memory to shared memory; each thread loads one element of each matrix
        AS(ty, tx) = A[a + uiWA * ty + tx];
        BS(ty, tx) = B[b + uiWB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        barrier(CLK_LOCAL_MEM_FENCE);

        // Multiply the two matrices together; each thread computes one element of the block sub-matrix
        #pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);

        // Synchronize to make sure that the preceding computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        barrier(CLK_LOCAL_MEM_FENCE);
    }

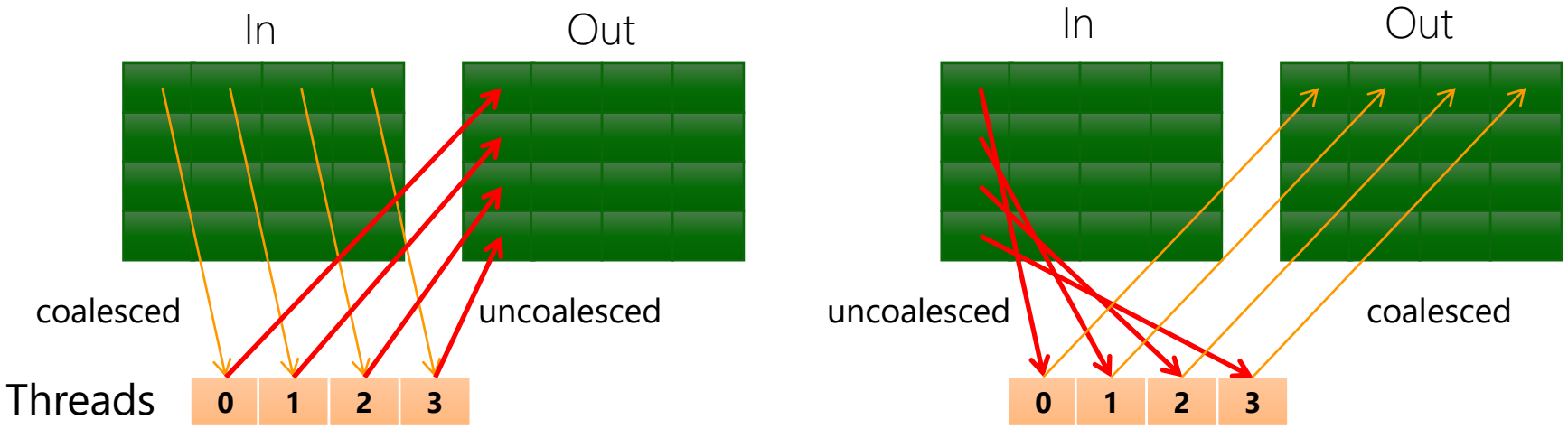
    if (get_global_id(1) < trueLocalSize1)
    // Write the block sub-matrix to device memory; each thread writes one element
    C[get_global_id(1) * get_global_size(0) + get_global_id(0)] = Csub;
}
```

Optimized Matrix Multiplication Results

- Note
  - Matrix size has to be a multiple of work-group size
- It’s faster!
  - Speed up of 6 over previous implementation

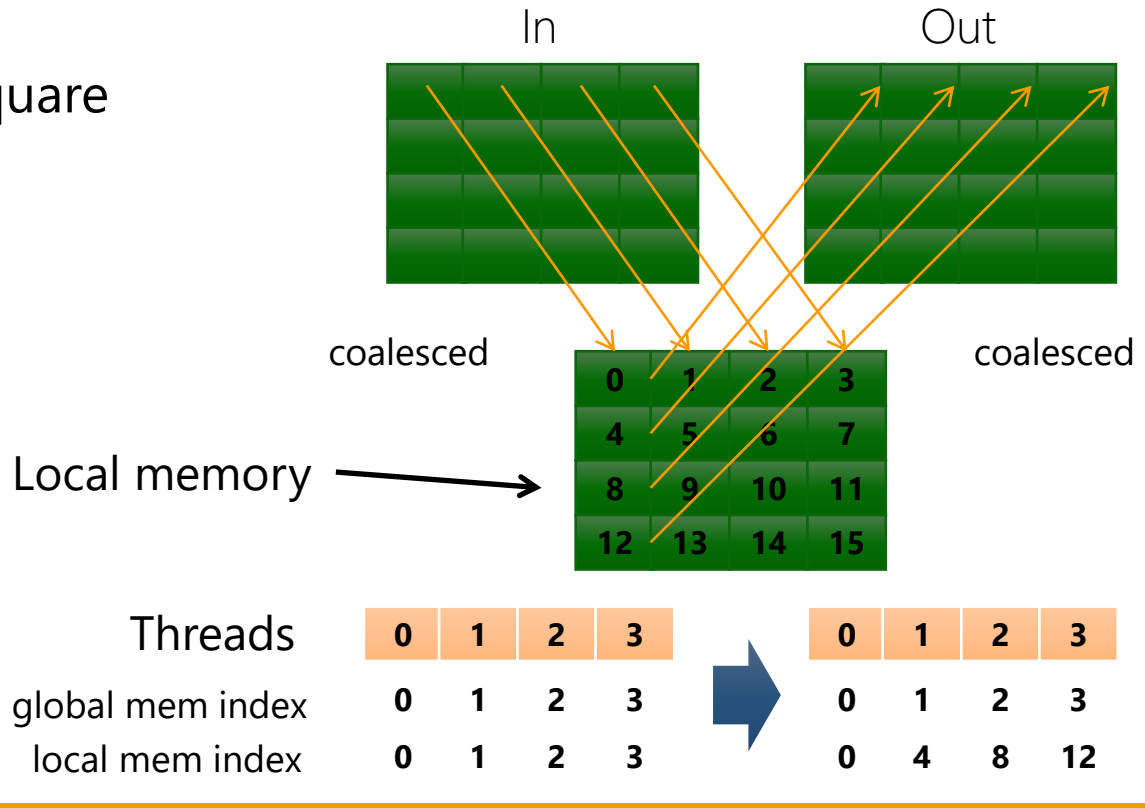
Lab Exercise: Matrix Transpose

- A matrix transpose is a straightforward technique
  - `Out(x,y) = In(y,x)`
- No matter which thread mapping is chosen, one operation (read/write) will produce coalesced accesses while the other (write/read) produces uncoalesced accesses
  - Note that data must be read to a temporary location (such as a register) before being written to a new location



Lab Exercise: Matrix Transpose

- If local memory is used to buffer the data between reading and writing, we can rearrange the thread mapping to provide coalesced accesses in both directions
  - Note that the work group must be square



Lab Exercise: Matrix Transpose

- Naïve implementation
  - suffers from completely non-coalesced writes

```
__kernel void transpose_naive(
    __global float* odata, __global float* idata,
    int width, int height)
{
    unsigned int xIndex = get_global_id(0);
    unsigned int yIndex = get_global_id(1);

    if (xIndex < width && yIndex < height)
    {
        unsigned int index_in = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        odata[index_out] = idata[index_in];
    }
}
```

- Exercise: Implement an optimized matrix transpose

Heterogenous Programming Models

High Performance Computing, Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Outline

- Programming models for heterogenous computing
  - OpenCL, CUDA, HIP
- SYCL
  - Introduction to SYCL
  - SYCL 2020: reduction, subgroups, unified shared memory

Programming Models for Heterogenous Computing

- PM for multicore CPU
  - OpenMP
  - pthreads
  - Intel TBB
- PM for GPU
  - NVIDIA CUDA
  - AMD HIP
- Heterogenous PM
  - Khronos OpenCL
  - Khronos SYCL

Kernel Addition in CUDA

- Host & device code in one file
- Kernel
  - definition: `__global`
  - invocation: `<<<nB,tB>>>`
- Allocation
  - `cudaMalloc`
  - `cudaFree`
- Copy to-from device
  - `cudaMemcpy`

```
#include <cuda.h>
#include <cuda_runtime.h>
#define N 256000
__global__ void vector_add(float *out, float *a, float *b, int n) {
  out[i] = a[i] + b[i];
}

int main(){
    float *a, *b, *out;
    float *d_a, *d_b, *d_out;
     // Allocate host memory
    a   = (float*)malloc(sizeof(float) * N);
    b   = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);
    // Initialize host arrays
    for(int i = 0; i < N; i++){
        a[i] = 1.0f;  b[i] = 2.0f;
    }
    // Allocate device memory
    cudaMalloc((void**)&d_a, sizeof(float) * N);
    cudaMalloc((void**)&d_b, sizeof(float) * N);
    cudaMalloc((void**)&d_out, sizeof(float) * N);
    // Transfer data from host to device memory
    cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);
    // Executing kernel
    vector_add<<<100,256>>>(d_out, d_a, d_b, N);
    // Transfer data back to host memory
    cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);
    /* print the out buffer */
    // Deallocate device memory
    cudaFree(d_a);    cudaFree(d_b);    cudaFree(d_out);
    // Deallocate host memory
    free(a);     free(b);     free(out);
}
```

OpenCL vs NVIDIA CUDA vs AMD HIP

	OpenCL	CUDA	HIP
	`#include "CL/cl.h"`	`#include "cuda.h"`	`#include "hcc.h"`
Kernel compilation	`clBuildFromSource/IL/…`	`nvcc`	`hipcc`
Allocate	`clCreateBuffer()`	`cudaMalloc()`	`hipMalloc()`
Host-to-device	`clEnqueueWriteBuffer()`	`cudaMemcpy(,,, cudaMemcpyHostToDevice)`	`hipMemcpy(,,, hipMemcpyHostToDevice)`
Device-to-host	`clEnqueueReadBuffer()`	`cudaMemcpy(,,, cudaMemcpyDeviceToHost)`	`hipMemcpy(,,, hipMemcpyDeviceTohost)`
Def. local memory	`__local int array[SZ];`	`__shared__ int array[SZ];`	`__shared__ int array[SZ];`
Kernel execution	`clEnqueueNDRangeKernel( q, &k, 2, 0, &global, &local,0, NULL, NULL);`	`kernel<<<num_blocks, threads_per_block>>>();`	`hipLaunchKernel(kernel, blocks,threadsPerBlock,…)`

Kernel Indexing: OpenCL vs NVIDIA CUDA vs AMD HIP

OpenCL	CUDA	HIP
`get_num_groups()`	`gridDim`	like CUDA
`get_group_id()`	`blockIdX`	
`get_local_size()`	`blockDim`	
`get_global_size()`	`gridDim*blockDim`	
`get_local_id()`	`threadIdx`	
`get_global_id()`	`blockIdx * blockdim + threadIdx`	

C++ Wrapup

- Assume C++ knowledge
- C++ features relevant for SYCL
  - templates
  - RAII (Resource Acquisition Is Initialization)
  - lambda
  - exception

C++ Fundamentals: Objects and Templates

- Assumption: general knowledge of basic C++ object and template usage

```
#include <iostream>
using namespace std;
class Rectangle {       // class
    int width, height; // members
  public:                // access specifier
    void set_values (int,int);
    int area(){ {return width*height;} //constructor
};
void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}
int main () {
  Rectangle rect; //object (instance of class)
  rect.set_values (3,4); //
  cout << "area: " << rect.area();
  return 0;
}
```

```
#include <iostream>
using namespace std;
template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}
int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  int p=1, q=2, r;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  r=GetMax(p,q); // automatic type inference
  cout <<k<<" "<<n<<" "<<r<< endl;
  return 0;
}
```

source: http://www.cplusplus.com/doc/tutorial/classes/

C++ Exception

- A C++ exception is a response to an exceptional circumstance that arises while a program is running
  - throw: exceptions can be thrown anywhere within a code block using throw statement
  - catch: a program catches an exception with an exception handler at the place in a program where you want to handle the problem
  - try: a try block identifies a block of code for which particular exceptions will be activated

```
double division(int a, int b) {
   if( b == 0 ) {
      throw "Div by zero!";
   }
   return (a/b);
}
```

```
int x = 50, y = 0;
double z = 0;
try {
  z = division(x, y);
  cout << z << endl;
} catch (const char* msg) {
  cerr << msg << endl;
}
```

RAII: Resource Acquisition Is Initialization

- Key C++ technique: Resource Acquisition Is Initialization
  - RAII uses classes with destructors to impose order on resource management
- For example

```
void old_fct(const char* s){
  // open the file
  FILE* f = fopen(s,"r");

  // use f

  fclose(f); // close the file
}
```

```
void fct(string s) {
  // constructor opens the file
  File_handle f(s,"r");

  // use f


} // destructor closes the file
```

Why RAII

- To avoid leaks and the complexity of manual resource management.
  - C++'s language-enforced constructor/destructor symmetry mirrors the symmetry inherent in resource acquire/release function pairs
    - such as `fopen`/`fclose`, `lock`/`unlock`, and `new`/`delete`
  - Whenever you deal with a resource that needs paired acquire/release function calls
    - encapsulate that resource in an object that enforces pairing
    - acquire the resource in its constructor, and release it in its destructor
- OpenCL semantic has the same issues
  - `clCreateContext`/`clReleaseContext`
  - `clCreateBuffer`/`clReleaseBuffer`
  - …

C++ Functors

- Functors (of function objects)
  - functors are objects that can be treated as though they are a function or function pointer
  - commonly used along with STL
  - functors are called using the same old function call syntax
  - to create a functor, we create an object that overloads the operator().
- Classical approach would use a unary function (one argument)
  - unlike regular functions, they can contain state
  - functor's state can save additional values

```
#include <algorithm>
#include <vector>
#include <iostream>

// functors
class increment {
private:
  int num;
public:
  increment(int n) : num(n) {  }
  int operator () (int arr_num) const {
    return num + arr_num;
  }
};
// functions
int incr(int val){return val + 1; }

int main() {
  std::vector<int> vec{ 10, 20, 30 };
  std::transform(vec.begin(), vec.end(),
    vec.begin(), incr);
  for(int i : vec) std::cout << i << " ";

  int to_add = 5;
  std::transform(vec.begin(), vec.end(),
    vec.begin(), increment(to_add));
  for(int i : vec) std::cout << i << " ";
}
```

C++ Lambdas

- Unnamed function object capable of capturing variables in scope (closure)
  - function pointers have minimal syntactic overhead but do not retain state within a scope
  - function objects can maintain state but require the syntactic overhead of a class definition
- Lambda Expression
  1. capture clause
     - specifies which variables are captured, and whether the capture is by value or by reference
     - `[ ]` body of the lambda expression accesses no variables in the enclosing scope
     - `[&]` all variables you refer are captured by reference
     - `[=]` means they are captured by value
  2. parameter list (optional)
  3. mutable specification (optional)
  4. exception specification (optional)
  5. trailing return type (optional)
  6. lambda body

```
[=] () mutable
throw() -> int
{
  int n = x + y;
  x = y;
  y = n;
  return n;
}
```

```
void abssort(float* x, unsigned n) {
  std::sort(x, x + n,
    // Lambda expression begins
    [](float a, float b) {
      return (std::abs(a)<std::abs(b));
    } // end of lambda expression
  );
}
```

Lambda Capture Examples

- You can use a default capture mode, and then specify the opposite mode explicitly for specific variables
  - e.g., the following capture clauses are equivalent:

```
[&total, factor]
[factor, &total]
[&, factor]
[factor, &]
[=, &total]
[&total, =]
```

- Also: high-order lambda functions
  - lambda in lambda

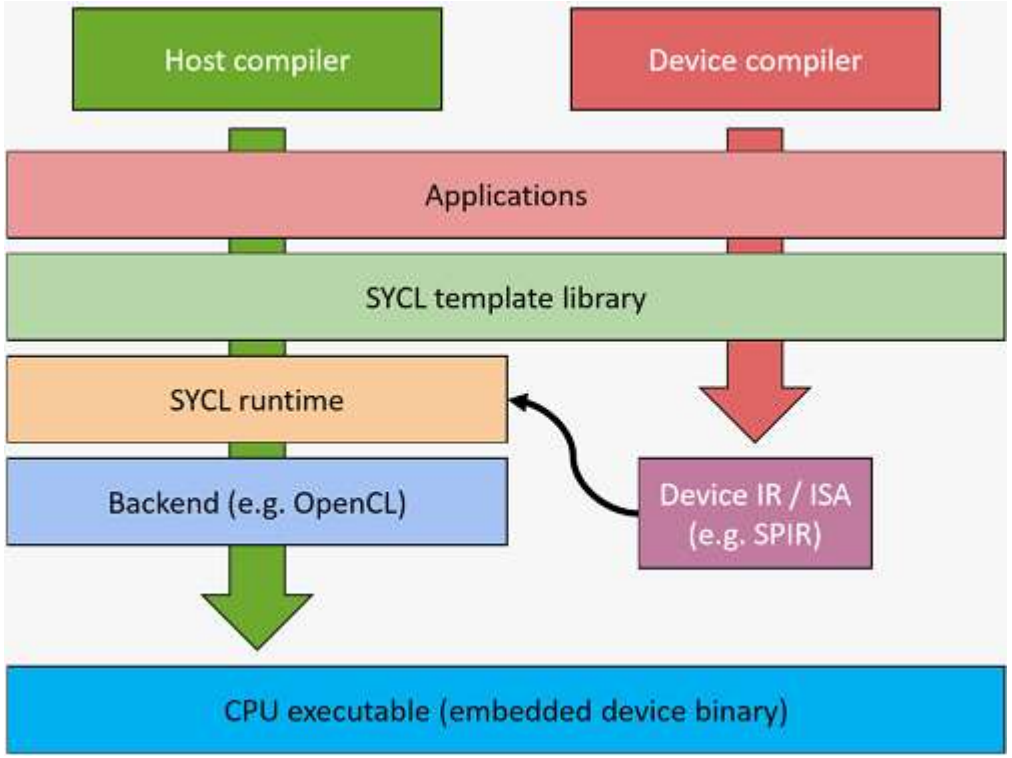
```
#include <functional>
#include <iostream>
int main(){
  using namespace std;
  // simple lambda expression
  auto f1 = [](int x, int y) { return x + y; };
  cout << f1(2, 3) << endl; // print 5
  // lambda expression assigned to a function object
  function<int(int,int)> f2 = [](int x, int y) {return x+y;};
  cout << f2(3, 4) << endl; // print 7
  // lambda expr. captures i by value and j by reference
  int i = 3;
  int j = 5;
  function<int (void)> f = [i, &j] { return i + j; };
  i = 30;
  j = 50;
  cout << f() << endl; // print 53
  // calling lambda with two input variable
  int n = [] (int x, int y) { return x + y; }(5, 4);
  cout << n << endl; // print 9
  // lambda capture m1 by reference and m2 by value
  // mutable allows m1 to be modified within the lambda
  int m1 = 0;
  int m2 = 0;
  [&, m2] (int a) mutable { m2 = ++m1 + a; }(4);
  cout << m1 << " " << m2 << endl; // print 1 0
}
```

SYCL

- SYCL is a single source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms
  - Pronunciation: sickle ["si-kel]
  - SYCL and SYCL logo are trademarks of the Khronos Group Inc.
- Programming interface based on abstraction of OpenCL components
- Most modern C++ features available for OpenCL
  - Enabling the creation of higher-level programming models
  - C++ templated libraries based on OpenCL
  - Exceptions for error handling
- Portability across platforms and compilers

SYCL: Single-source

- SYCL is a single source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms
- SYCL allows you to write both host CPU and device code in the same C++ source file
- This requires two compilation passes
  - one for the host code
  - one for the device code



SYCL: High-level

- SYCL is a single source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms
- SYCL provides high-level abstractions over common boilerplate code
  - Platform/device selection
  - Buffer creation
  - Kernel compilation
  - Dependency management and scheduling

Is SYCL a high-level language? SYCL vs OpenCL

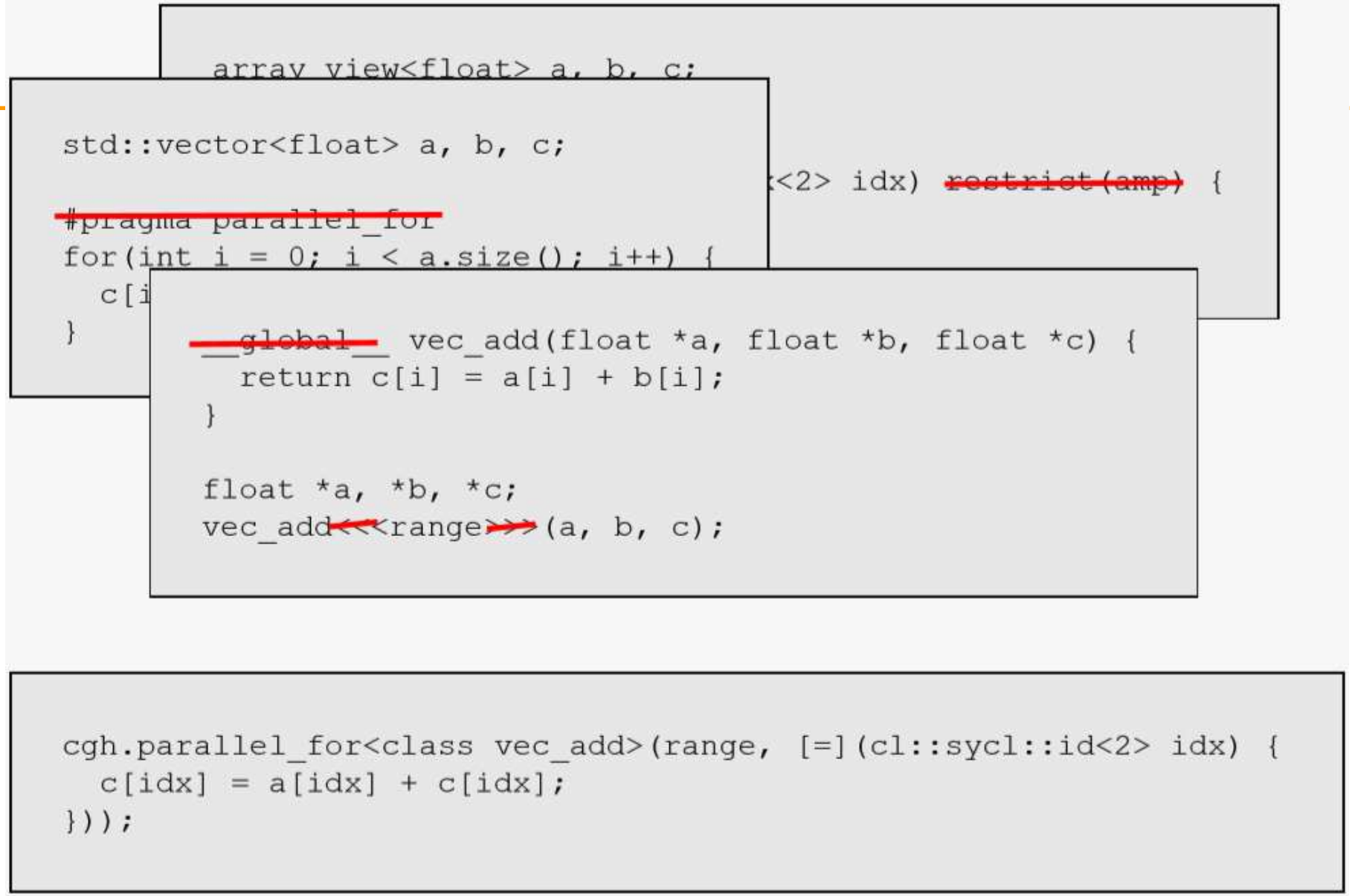
Typical OpenCL hello world application

Typical SYCL hello world application

SYCL and the C++ Standard

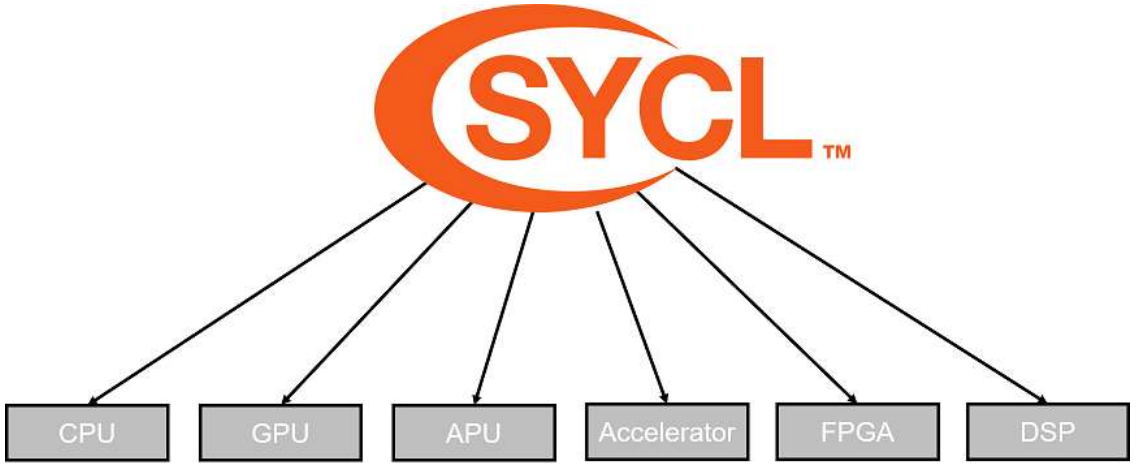
- SYCL is a single source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms
- SYCL allows you to write standard C++
- Unlike the other implementations shown on the left there are
  - no language extensions
  - no pragmas
  - no attributes
- Also
  - C++ memory model, atomics, ect

SYCL is C++


array_view<float> a, b, c;
<2> idx) restrict(amp) {
std::vector<float> a, b, c;
#pragma parallel_for
for(int i = 0; i < a.size(); i++) {
c[i
}
__global__ vec_add(float *a, float *b, float *c) {
return c[i] = a[i] + b[i];
}
float *a, *b, *c;
vec_add<<<range>>>(a, b, c);
cgh.parallel_for<class vec_add>(range, [=](cl::sycl::id<2> idx) {
c[idx] = a[idx] + c[idx];
}));

SYCL for Heterogenous Programming

- SYCL is a single source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms
  - SYCL can target any device supported by its back-end
  - SYCL can target a number of different backends
  - While the current specification is limited to OpenCL, some implementations are already supporting other non-OpenCL back-ends



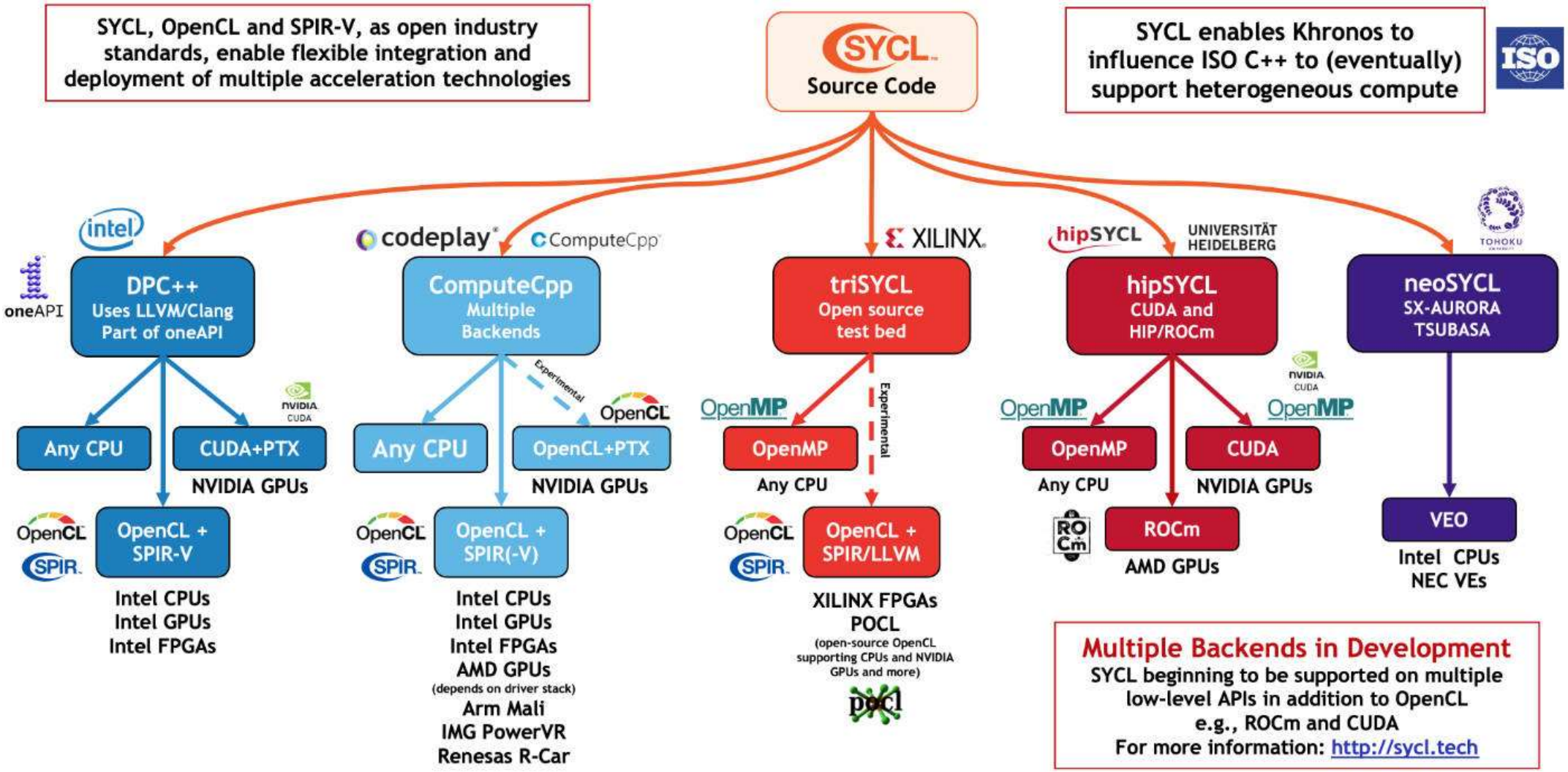
SYCL is Heterogenous

- SYCL is a single source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms
- Implementations:
  - ComputeCpp https://developer.codeplay.com
  - HipSYCL https://github.com/illuhad/hipSYCL
  - triSYCL https://github.com/triSYCL/triSYCL
  - Intel LLVM SYCL https://github.com/intel/llvm
- Code examples in this section are courtesy of CodePlay
- References to specification Ver. 1.2.1, Rev. 6
  - Warning: SYCL 2020 introduces several changes

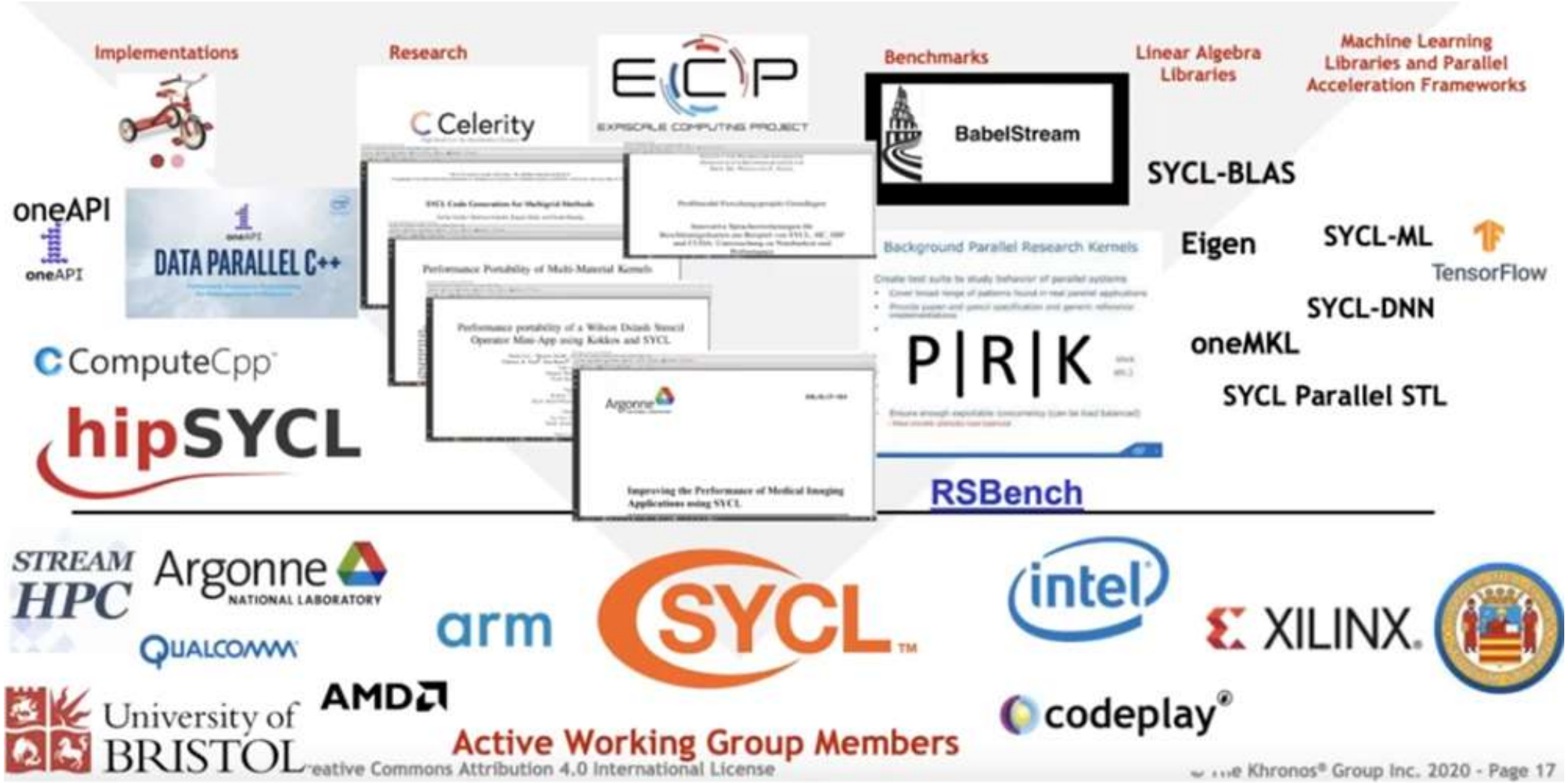
SYCL Academy

References to the specification
Ver 1.2.1, Revision 6

Available SYCL Implementations

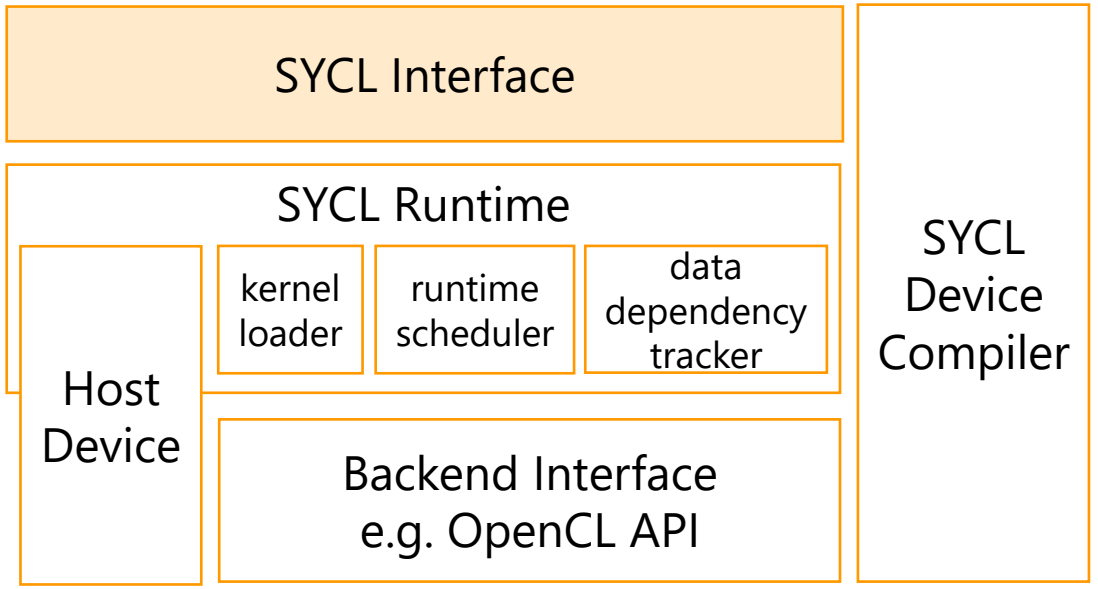


SYCL Ecosystem, Research and Benchmarks



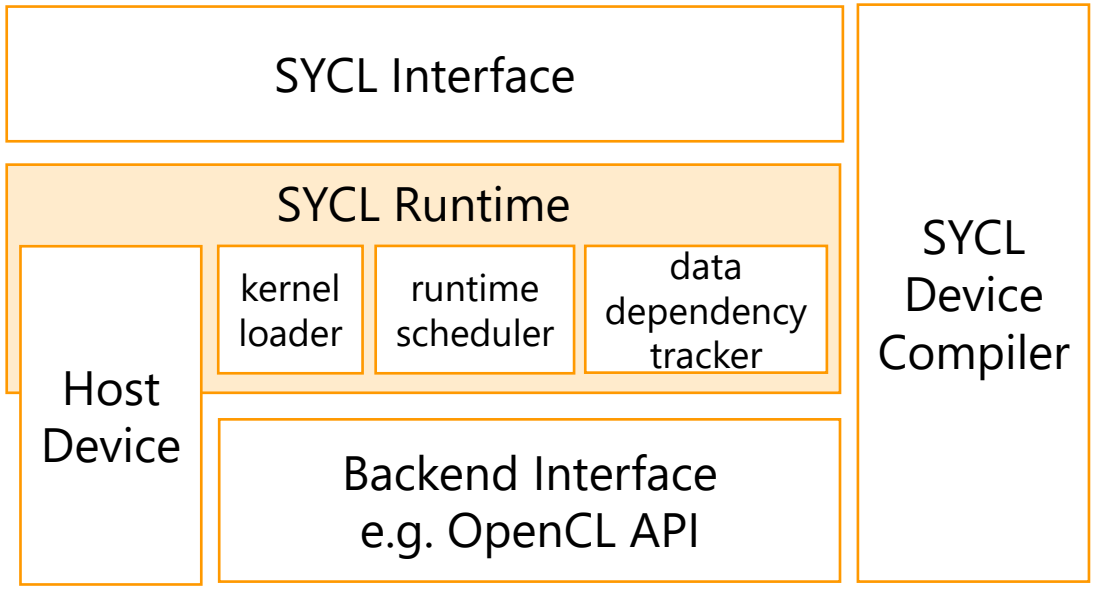
SYCL Interface

- The SYCL interface is a C++ template library that developers can use to access the features of SYCL
- The same interface is used for both the host and device code
  - The host is generally the CPU and is used to dispatch the parallel execution of kernels
  - The device is the parallel unit used to execute the kernels, such as a GPU



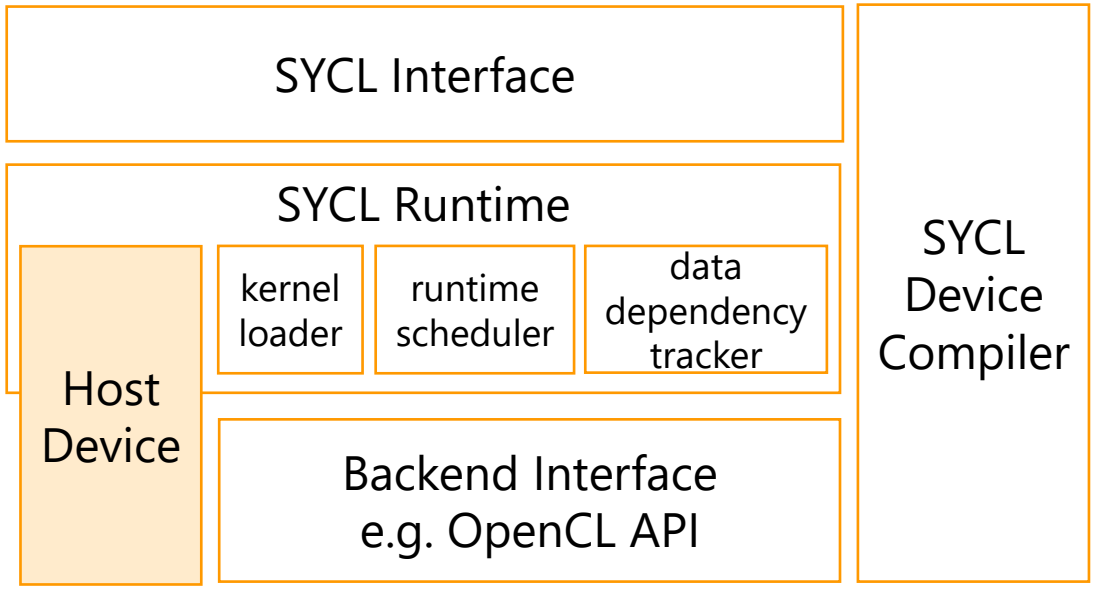
SYCL Runtime

- The SYCL runtime is a library that schedules and executes work
  - It loads kernels, tracks data dependencies and schedules commands



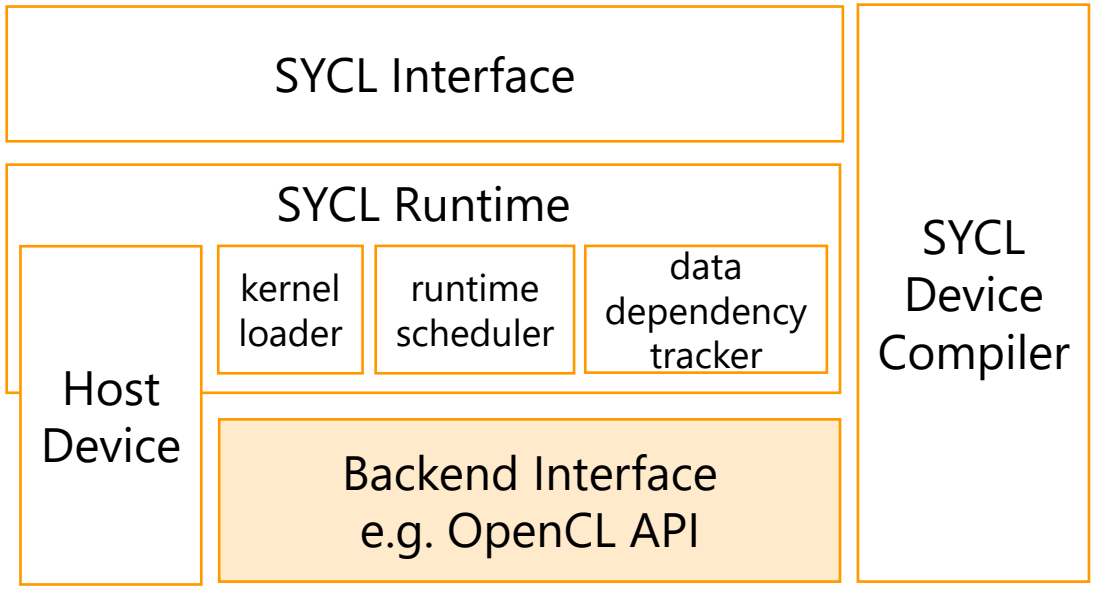
SYCL Host Device

- The host device is an emulated backend that is executed as native C++ code and emulates the SYCL execution and memory model
- The host device can be used to execute kernels without backend drivers and for debugging purposes



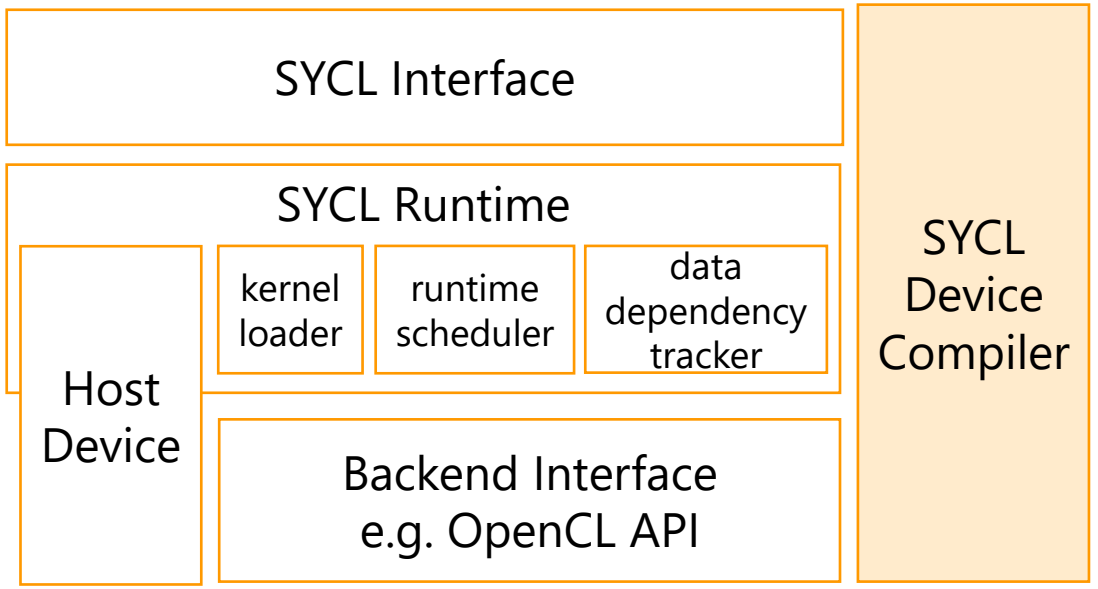
SYCL Back-end Interface

- The back-end interface is where the SYCL runtime calls down into a back-end in order to execute on a particular device
- The standard back-end is OpenCL but some implementations support other interfaces



SYCL Interface

- The SYCL device compiler is a C++ compiler which can identify SYCL kernels and compile them down to an IR or ISA
  - This can be SPIR, SPIR-V, GCN, PTX or any proprietary vendor ISA
  - IR = Intermediate Representation
  - ISA = Instruction Set Architecture



First SYCL Example

SYCL Academy codeplay

- First, we include the SYCL header which contains the runtime API
- We also import the `cl::sycl` namespace here, this reduces the amount of code we need to write
  - Warning: in SYCL 2020 is `sycl`

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;
int main(int argc, char *argv[]) {




}
```

First SYCL Example

- Device selectors allow you to choose a device based on a custom configuration
- The queue default constructor uses the `default_selector`, which allows the runtime to select a device for you

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;
int main(int argc, char *argv[]) {


  queue gpuQueue{gpu_selector{}};





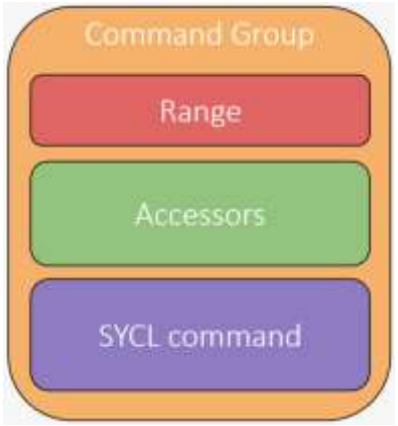




}
```

First SYCL Example

- We can submit a command group to a queue
- A command group contains
  - a SYCL command, e.g., kernel function
  - execution range
  - accessors




```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;
int main(int argc, char *argv[]) {


  queue gpuQueue{gpu_selector{}};



  gpuQueue.submit([&](handler &cgh){




  });


}
```

First SYCL Example

- Buffer creation
- We create a buffer for each vector to manage the data across host and device

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;
int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  queue gpuQueue{gpu_selector{}};
  buffer<float,1> bufA(dA.data(),range<1>(dA.size()));
  buffer<float,1> bufB(dB.data(),range<1>(dB.size()));
  buffer<float,1> bufO(dO.data(),range<1>(dO.size()));
  gpuQueue.submit([&](handler &cgh){









  });


}
```

First SYCL Example

- Buffers on leaving scope
- Buffers synchronize on destruction via RAII
  - adding this scope means that all kernels writing to the buffers will wait
  - …and the data will be copied back to the vectors on leaving this scope

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;
int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  queue gpuQueue{gpu_selector{}};
  buffer<float,1> bufA(dA.data(),range<1>(dA.size()));
  buffer<float,1> bufB(dB.data(),range<1>(dB.size()));
  buffer<float,1> bufO(dO.data(),range<1>(dO.size()));
  gpuQueue.submit([&](handler &cgh){









  });


}
```

First SYCL Example

- Accessors creation
- We create an accessor for each of the buffers
  - read access for the two input buffers
  - write access for the output buffer

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;
int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  queue gpuQueue{gpu_selector{}};
  buffer<float,1> bufA(dA.data(),range<1>(dA.size()));
  buffer<float,1> bufB(dB.data(),range<1>(dB.size()));
  buffer<float,1> bufO(dO.data(),range<1>(dO.size()));
  gpuQueue.submit([&](handler &cgh){
    auto inA = bufA.get_access(cgh);
    auto inB = bufB.get_access(cgh);
    auto out = bufO.get_access(cgh);


  });


}
```

First SYCL Example

- We define a SYCL kernel function for the command group using the `parallel_for` API
  - 1st argument is a range, specifying the iteration space
  - 2nd argument is a lambda function that represents the entry point for the SYCL kernel
  - This lambda takes an id parameter that describes the current iteration being executed

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;
int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  queue gpuQueue{gpu_selector{}};
  buffer<float,1> bufA(dA.data(),range<1>(dA.size()));
  buffer<float,1> bufB(dB.data(),range<1>(dB.size()));
  buffer<float,1> bufO(dO.data(),range<1>(dO.size()));
  gpuQueue.submit([&](handler &cgh){
    auto inA = bufA.get_access(cgh);
    auto inB = bufB.get_access(cgh);
    auto out = bufO.get_access(cgh);
    cgh.parallel_for<add>(range<1>(dA.size()),
      [=](id<1> i){ out[i] = inA[i] + inB[i]; });
  });


}
```

First SYCL Example

- The template parameter to `parallel_for` is used to name the lambda
  - C++ does not have a standard ABI for lambdas so they are represented differently across the host and device compiler
- SYCL kernel functions follow C++ ODR rules:
  - if a SYCL kernel is in a template context, the kernel name needs to reflect that context, so must contain the same template arguments

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;
int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  queue gpuQueue{gpu_selector{}};
  buffer<float,1> bufA(dA.data(),range<1>(dA.size()));
  buffer<float,1> bufB(dB.data(),range<1>(dB.size()));
  buffer<float,1> bufO(dO.data(),range<1>(dO.size()));
  gpuQueue.submit([&](handler &cgh){
    auto inA = bufA.get_access(cgh);
    auto inB = bufB.get_access(cgh);
    auto out = bufO.get_access(cgh);
    cgh.parallel_for<add>(range<1>(dA.size()),
      [=](id<1> i){ out[i] = inA[i] + inB[i]; });
  });


}
```

First SYCL Example

- Error handling
  - in SYCL errors are handled using exception handling
  - you should always wrap SYCL code in a try-catch block
- Exceptions are either
  - thrown synchronously at the point of using a SYCL API
  - asynchronous, stored by the runtime, and passed to an async handler when the queue is told to throw

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;
int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };
  try{
    queue gpuQueue{gpu_selector{}, async_handler{}};
    buffer<float,1> bufA(dA.data(),range<1>(dA.size()));
    buffer<float,1> bufB(dB.data(),range<1>(dB.size()));
    buffer<float,1> bufO(dO.data(),range<1>(dO.size()));
    gpuQueue.submit([&](handler &cgh){
      auto inA = bufA.get_access(cgh);
      auto inB = bufB.get_access(cgh);
      auto out = bufO.get_access(cgh);
      cgh.parallel_for<add>(range<1>(dA.size()),
        [=](id<1> i){ out[i] = inA[i] + inB[i]; });
    });
    gpuQueue.wait_and_throw();
  } catch (...) { /* handle errors */ }
}
```

SYCL 2020 Features

- Specifications: https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html
- (Some) SYCL 2020 features
  - simplification of accessors, class template argument deduction (CTAD) and deduction guides to enable simpler class template instantiation
  - work group and subgroup algorithms, enabling efficient operations between work items
  - reduction
  - unified shared memory (USM)

SYCL 2020

- Accessor semantic
  - CTAD

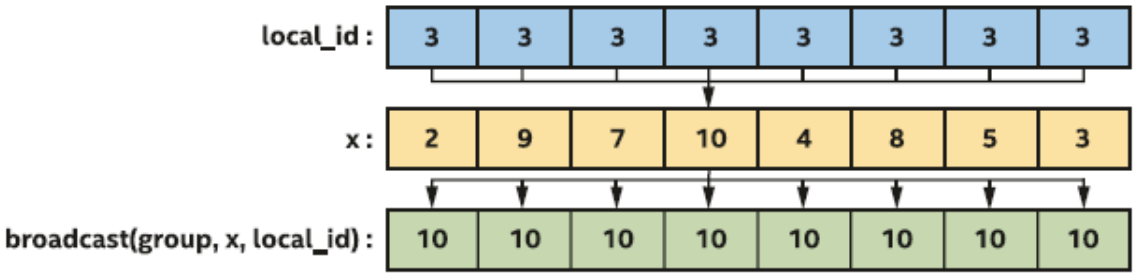
```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

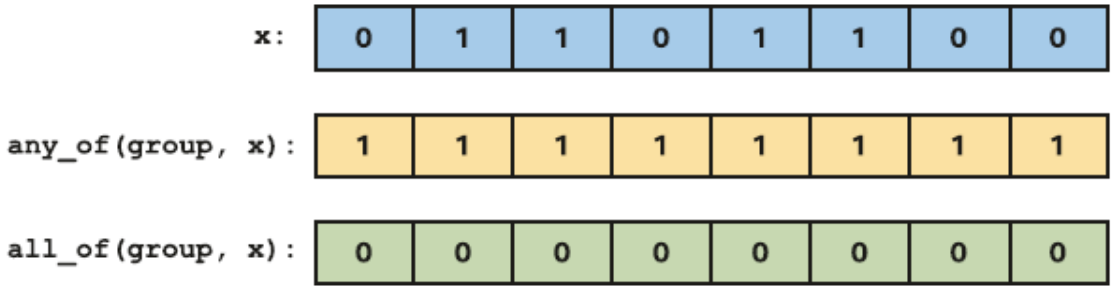
int main() {
  int data[1024];  // Allocate data to be worked on
  // Create a default queue to enqueue work to the default device
  queue myQueue;
  // By wrapping all the SYCL work in a {} block, we ensure all SYCL tasks mus
  // before exiting the block, because the destructor of resultBuf will wait
  {
    // Wrap our data variable in a buffer
    buffer<int, 1> resultBuf { data, range<1> { 1024 } };

    // Create a command group to issue commands to the queue
    myQueue.submit([&](handler& cgh) {
      // Request write access to the buffer without initialization
      accessor writeResult { resultBuf, cgh, write_only, no_init };
      // Enqueue a parallel_for task with 1024 work-items
      cgh.parallel_for(1024, [=](id<1> idx) {
        // Initialize each buffer element with its own rank number starting at
        writeResult[idx] = idx;
      }); // End of the kernel function
    }); // End of our commands for this queue
  } // End of scope, so we wait for work producing resultBuf to complete
  // Print result
  for (int i = 0; i < 1024; i++)
    std::cout << "data[" << i << "] = " << data[i] << std::endl;
  return 0;
}
```

SYCL 2020 Subgroup

- A sub-group is an implementation-defined subset of work-items in a work-group that execute together on the same hardware resources or with additional scheduling guarantees
  - because the implementation decides how to group work-items into sub-groups
  - work-items in a sub-group may be able to communicate or synchronize more efficiently than the work-items in an arbitrary work-group
  - mapped on: SIMD (Intel/AMD CPU), warp (NVIDIA GPU), wavefront (AMD GPU)
- Example of subgroup operation: `broadcast, any_of, all_of`





SYCL 2020 Reduction

- All functionality related to reductions is captured by the reducer class and the reduction function

```
buffer<int> valuesBuf { 1024 };
{
  // Initialize buffer on the host with 0, 1, 2, 3, ..., 1023
  host_accessor a { valuesBuf };
  std::iota(a.begin(), a.end(), 0);
}

// Buffers with just 1 element to get the reduction results
int sumResult = 0;
buffer<int> sumBuf { &sumResult, 1 };
int maxResult = 0;
buffer<int> maxBuf { &maxResult, 1 };

myQueue.submit([&](handler& cgh) {
  // Input values to reductions are standard accessors
  auto inputValues = valuesBuf.get_access<access_mode::read>(cgh);

  // Create temporary objects describing variables with reduction semantics
  auto sumReduction = reduction(sumBuf, cgh, plus<>());
  auto maxReduction = reduction(maxBuf, cgh, maximum<>());

  // parallel_for performs two reduction operations, for each reduction variable:
  // - Creates a corresponding reducer
  // - Passes a reference to the reducer to the lambda as a parameter
  cgh.parallel_for(range<1>{1024}, sumReduction, maxReduction,
    [=](id<1> idx, auto& sum, auto& max) {
      // plus<>() corresponds to += operator
      sum += inputValues[idx];
      // maximum<>() has no shorthand operator
      max.combine(inputValues[idx]);
  });
});

// sumBuf and maxBuf contain the reduction results once the kernel completes
assert(maxBuf.get_host_access()[0] == 1023 && sumBuf.get_host_access()[0] == 523776);
```

Lab

- Setup for OpenCL or SYCL on a heterogenous platform, e.g., CPU and GPU
- Execute a kernel either on a CPU or a GPU
  - vector addition
  - saxpy
- You can also access to
  - Google Codelabs: https://codelabs.developers.google.com
  - Intel DevCloud: https://intelsoftwaresites.secure.force.com/devcloud/oneapi
    - include Intel Arria and Stratix FPGAs
    - OneAPI Data Parallel C++ (include SYCL)