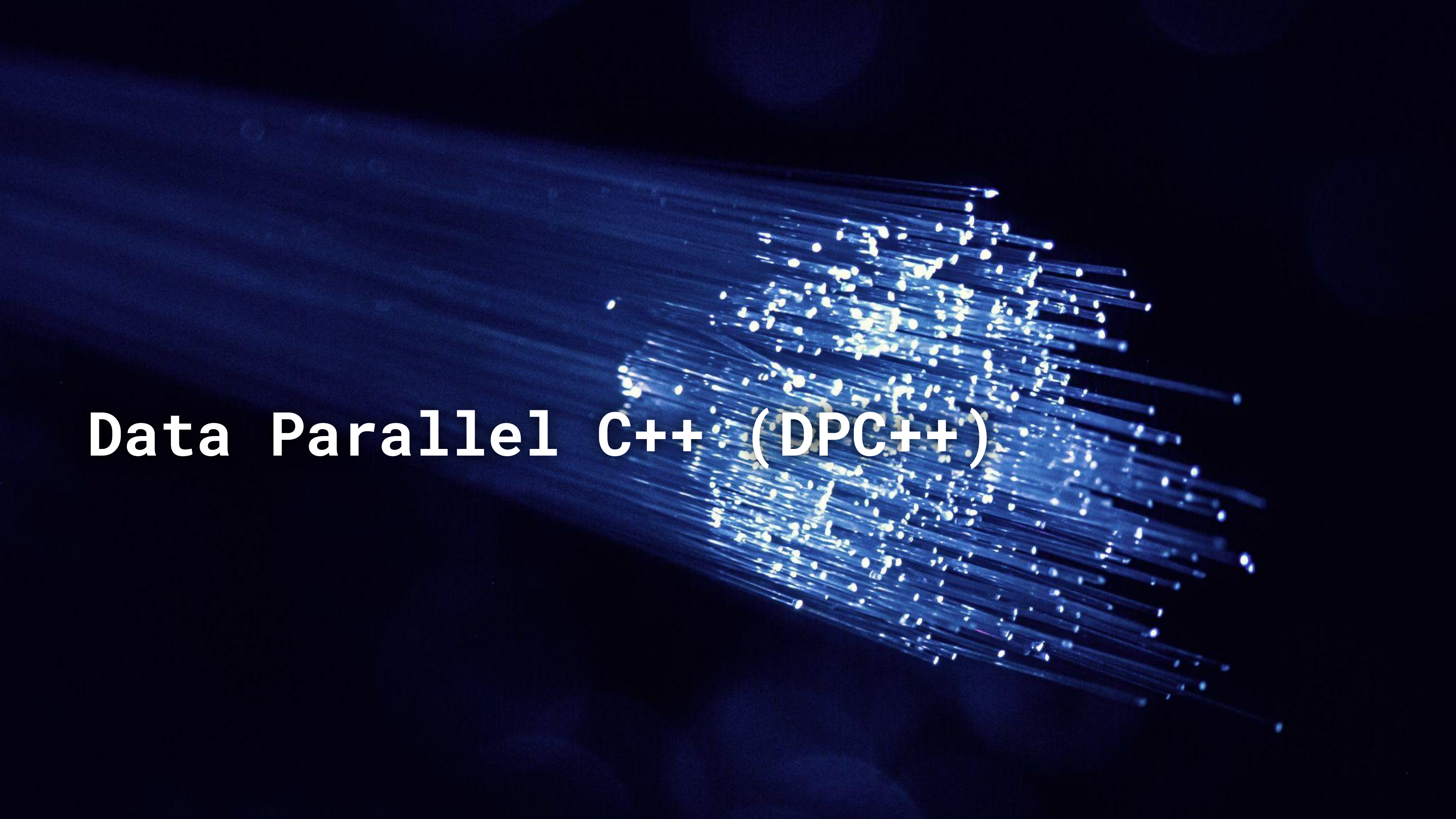
Parallel Graph Algorithms in Intel Data Parallel C++ (DPC++)

Group 1


Data Parallel C++ (DPC++)

DPC++

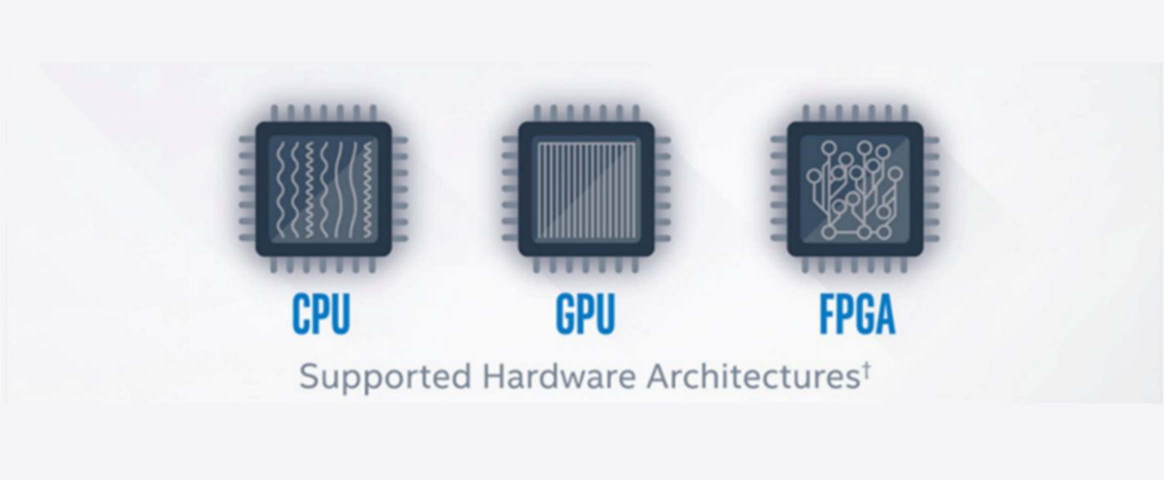
- Data Parallel C++
- New open source cross-architecture language
- Built upon ISO C++ and Khronos Group SYCL standards
- Part of a bigger oneAPI toolkit

KHRONOS® GROUP

Presenter: Rodger Chen

DPC++ Platforms

- DPC can be run on a variety of devices
  - CPU
  - GPU
  - FPGA
  - Accelerators



Presenter: Rodger Chen

How does DPC work?

- An extension of SYCL
- SYCL is for portable heterogeneous C++ applications
- SYCL intended for openCL but is now more generic
    - Similarities between the two languages such as the same Host Kernel model.

Presenter: Rodger Chen

Key Features:

- Single Source
- Host/Device model
- Asynchronous Execution

Single Source

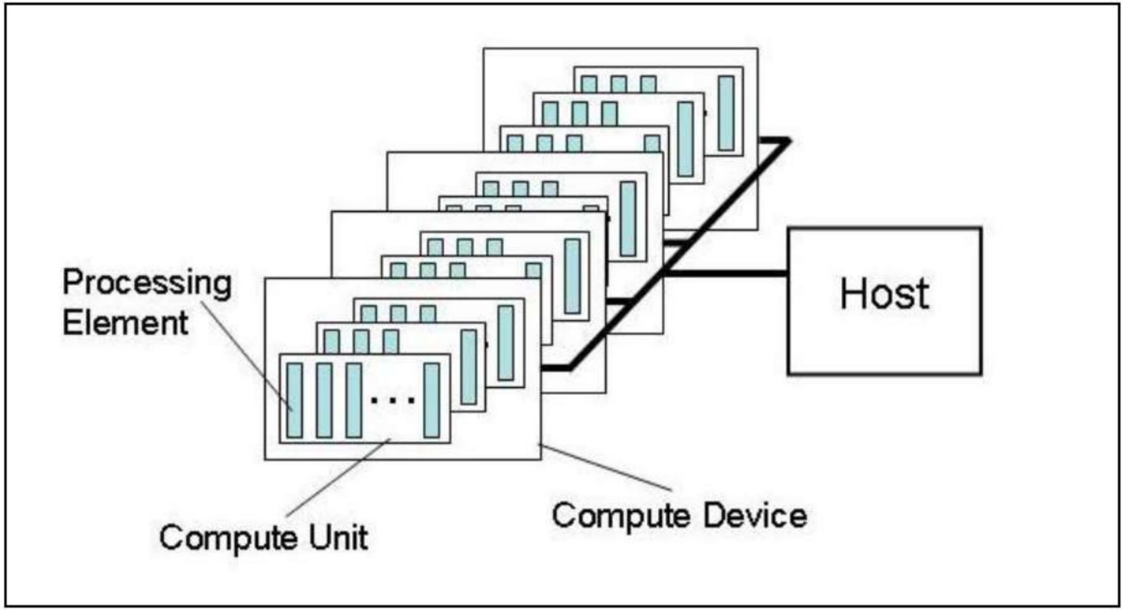
- SIngle source file which contains both the host code and kernel code
- C++ based
  - No VHDL
- Easy to read and write - even for people that have basic coding knowledge

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

static const int N = 8;

int main(){
  queue q;
  std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl;

  int *data = malloc_shared<int>(N, q);
  for(int i=0; i<N; i++) data[i] = i;

  q.parallel_for(range<1>(N), [=] (id<1> i){
    data[i] *= 2;
  }).wait();

  for(int i=0; i<N; i++) std::cout << data[i] << std::endl;
  free(data, q);
  return 0;
}
```

Presenter: Rodger Chen

Host / Kernel Model

- Host
    - Controls the execution of the program (“The brain”)
    - Usually the CPU, but does not have to be
- Kernel
    - Refers to the code that will be run on the device
    - Can be run on a number of platforms

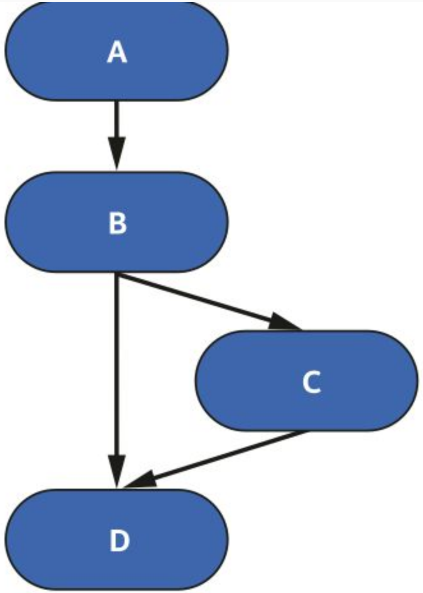


Presenter: Rodger Chen

Asynchronous Execution

- Any work sent to a device is done asynchronously from the Host.
- Amdahl's law!
- If implemented correctly: scalable
- If implemented incorrectly: Race conditions
- Data dependencies can be satisfied in a multitude of ways in DPC++

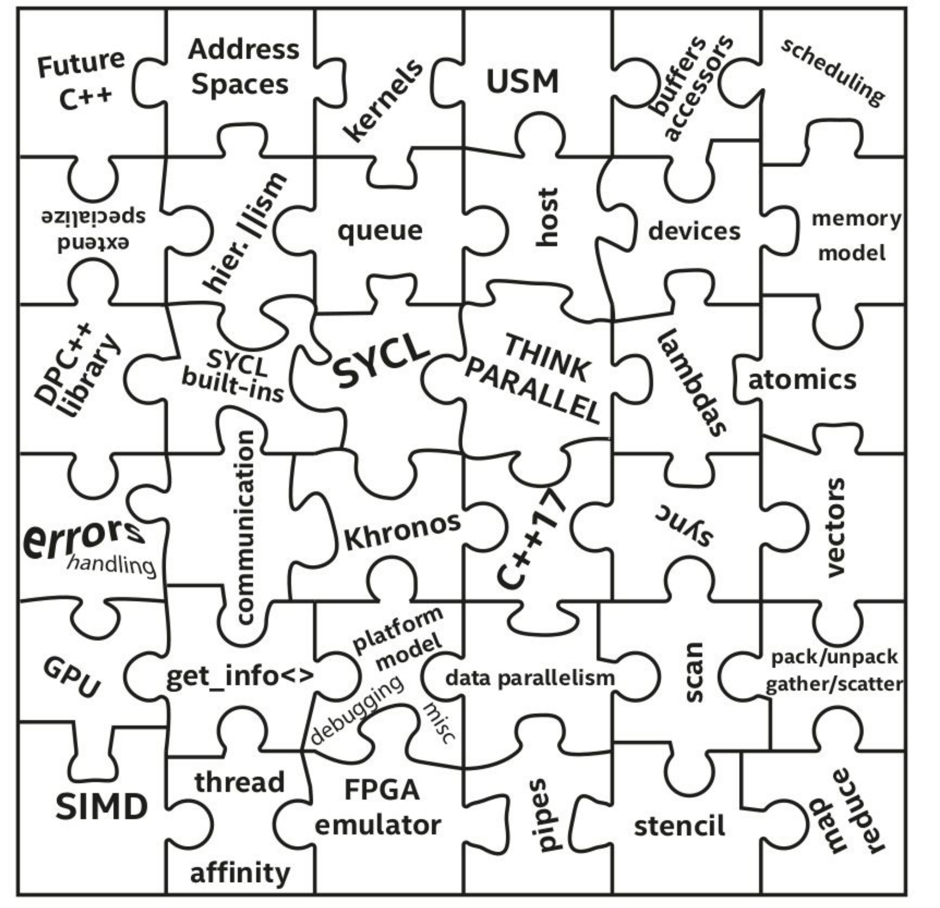
```
// Task A
auto eA = Q.submit([&](handler &h) {
  h.parallel_for(N, [=](id<1> i) { /*...*/ });
});
eA.wait();

// Task B
auto eB = Q.submit([&](handler &h) {
  h.parallel_for(N, [=](id<1> i) { /*...*/ });
});

// Task C
auto eC = Q.submit([&](handler &h) {
  h.depends_on(eB);
  h.parallel_for(N, [=](id<1> i) { /*...*/ });
});

// Task D
auto eD = Q.submit([&](handler &h) {
  h.depends_on({eB, eC});
  h.parallel_for(N, [=](id<1> i) { /*...*/ });
});

return 0;
```



Presenter: Rodger Chen

DPC++ Language Features

- Similar to OpenCL
- Queues
- Buffers
- Kernels



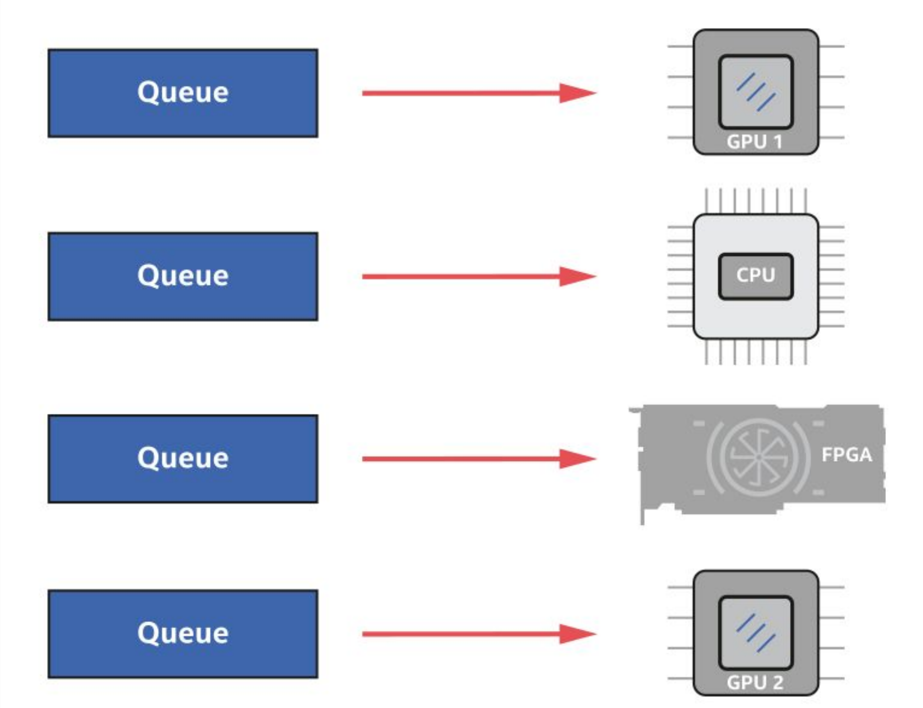
Presenter: Nikhil Kumar

```cpp
// Create buffers that hold the data shared between the host and the devices.
// The buffer destructor is responsible to copy the data back to host when it
// goes out of scope.
buffer a_buf(a_vector);
buffer b_buf(b_vector);
buffer sum_buf(sum_parallel.data(), num_items);

// Submit a command group to the queue by a lambda function that contains the
// data access permission and device computation (kernel).


q.submit([&](handler &h) {
  // Create an accessor for each buffer with access permission: read, write or
  // read/write. The accessor is a mean to access the memory in the buffer.
  accessor a(a_buf, h, read_only);
  accessor b(b_buf, h, read_only);

  // The sum_accessor is used to store (with write permission) the sum data.
  accessor sum(sum_buf, h, write_only, noinit);


  // Use parallel_for to run vector addition in parallel on device. This
  // executes the kernel.
  //    1st parameter is the number of work items.
  //    2nd parameter is the kernel, a lambda that specifies what to do per
  //    work item. The parameter of the lambda is the work item id.
  // DPC++ supports unnamed lambda kernel by default.

  h.parallel_for(num_items, [=](auto i) { sum[i] = a[i] + b[i]; });
});
```

Presenter: Full Name

Queues

- Abstraction for connecting to a device
- Bound to a single device at a time
- Queue has a submit method which takes a command group
- Work is transferred to the Device




```
queue q{}; //Default to most performant device
queue q_CPU(host_selector{}, exception_handler);
queue q_GPU(gpu_selector{}, exception_handler);
```

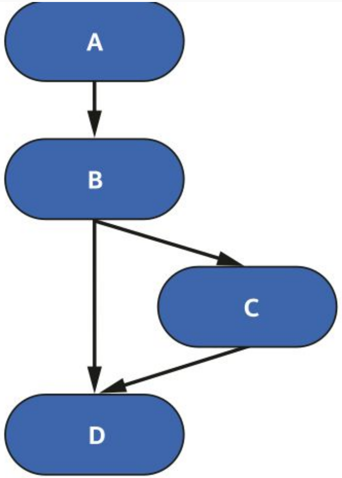
Presenter: Nikhil Kumar

Queues

- Abstraction for connecting to a device
- Bound to a single device at a time
- Queue has a submit method which takes a command group
- Work is transferred to the Device

```cpp
q.submit([&](handler &h) {
  // Create an accessor for each buffer with access permission: read, write or
  // read/write. The accessor is a mean to access the memory in the buffer.
  accessor a(a_buf, h, read_only);
  accessor b(b_buf, h, read_only);

  // The sum_accessor is used to store (with write permission) the sum data.
  accessor sum(sum_buf, h, write_only, noinit);

  h.parallel_for(num_items, [=](auto i) { sum[i] = a[i] + b[i]; });
```

```cpp
// Task A
auto eA = Q.submit([&](handler &h) {
  h.parallel_for(N, [=](id<1> i) { /*...*/ });
});
eA.wait();

// Task B
auto eB = Q.submit([&](handler &h) {
  h.parallel_for(N, [=](id<1> i) { /*...*/ });
});

// Task C
auto eC = Q.submit([&](handler &h) {
  h.depends_on(eB);
  h.parallel_for(N, [=](id<1> i) { /*...*/ });
});

// Task D
auto eD = Q.submit([&](handler &h) {
  h.depends_on({eB, eC});
  h.parallel_for(N, [=](id<1> i) { /*...*/ });
});

return 0;
```



Presenter: Nikhil Kumar

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

int main() {
  constexpr int size=16;
  std::array<int, size> data;

  // Create queue on implementation-chosen default device
  queue Q;

  // Create buffer using host allocated "data" array
  buffer B { data };

  Q.submit([&](handler& h) {
    accessor A{B, h};
    h.parallel_for(size , [=](auto& idx) {
        A[idx] = idx;
        });
    });

  // Obtain access to buffer on the host
  // Will wait for device kernel to execute to generate data
  host_accessor A{B};
  for (int i = 0; i < size; i++)
    std::cout << "data[" << i << "] = " << A[i] << "\n";

  return 0;
}
```

Host code

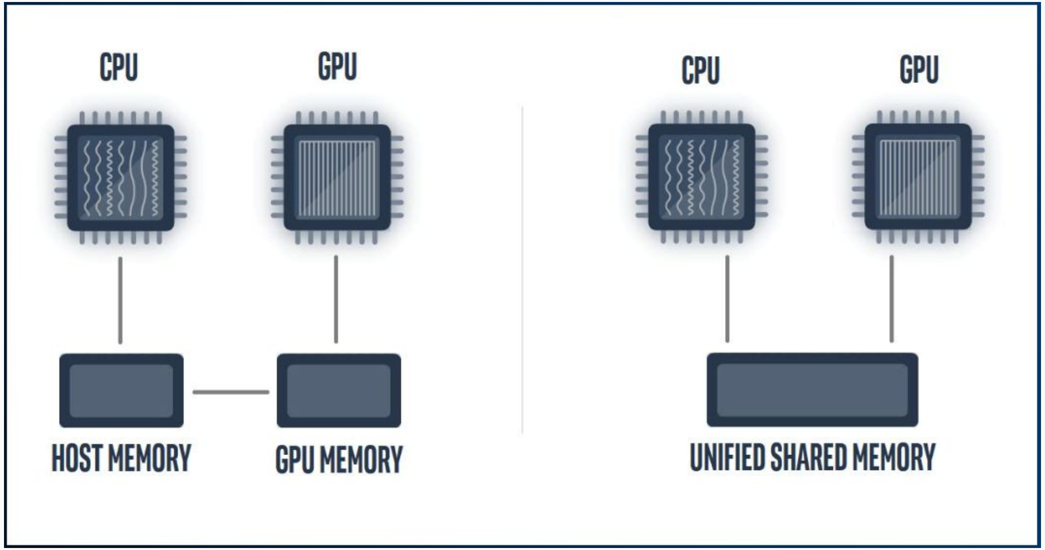
Device code

Host code

Presenter: Nikhil Kumar

Memory Management

- Buffers and Accessors
  - Buffers for holding data
    - Memory can be located anywhere
  - Accessors for accessing
- Unified Shared Memory
  - Familiar C++ Pointer Style
  - Implicit or explicit data transfer



Allocation Type	Description	Accessible on host?	Accessible on device?	Located on
device	Allocations in device memory	×	✓	device
host	Allocations in host memory	✓	✓	host
shared	Allocations shared between host and device	✓	✓	can migrate back and forth

Presenter: Nikhil Kumar

Memory Management

- Buffers and Accessors
  - Buffers for holding data
    - Memory can be located anywhere
  - Accessors for accessing
- Unified Shared Memory
  - Familiar C++ Pointer Style
  - Implicit or explicit data transfer

```
// Create buffers that hold the data shared between the host and the devices.
// The buffer destructor is responsible to copy the data back to host when it
// goes out of scope.
buffer a_buf(a_vector);
buffer b_buf(b_vector);
buffer sum_buf(sum_parallel.data(), num_items);
```

```
// Create an accessor for each buffer with access permission: read, write or
// read/write. The accessor is a mean to access the memory in the buffer.
accessor a(a_buf, h, read_only);
accessor b(b_buf, h, read_only);

// The sum_accessor is used to store (with write permission) the sum data.
accessor sum(sum_buf, h, write_only, noinit);
```

Explicit memory operation	copy	Copy data between locations specified by accessor, pointer, and/or shared_ptr. The copy occurs as part of the DAG, including dependence tracking.
	update_host	Trigger update of host data backing of a buffer object.
	fill	Initialize data in a buffer to a specified value.

Presenter: Nikhil Kumar

Kernels

- Code which runs on the device
- Platform Agnostic
  - Describes parallelism
- Lambda Function
  - Capture
  - Parameters
  - Lamda Body

```
// Use parallel_for to run vector addition in parallel on device. This
// executes the kernel.
//    1st parameter is the number of work items.
//    2nd parameter is the kernel, a lambda that specifies what to do per
//    work item. The parameter of the lambda is the work item id.
// DPC++ supports unnamed lambda kernel by default.

h.parallel_for(num_items, [=](auto i) { sum[i] = a[i] + b[i]; });
```

```
[ capture-list ] ( params ) -> ret { body }
```

Presenter: Nikhil Kumar

Actions Types

- Single task
- Parallel for
  - Basic Kernel
- Parallel for work group
  - Thread of execution
  - ND-Kernel
  - Hierarchical kernel

Work Type	Actions (handler class methods)	Summary
Device code execution	single_task	Execute a single instance of a device function.
	parallel_for	Multiple forms are available to launch device code with different combinations of work sizes.
	parallel_for_work_group	Launch a kernel using hierarchical parallelism, described in Chapter 4.
Explicit memory operation	copy	Copy data between locations specified by accessor, pointer, and/or shared_ptr. The copy occurs as part of the DAG, including dependence tracking.
	update_host	Trigger update of host data backing of a buffer object.
	fill	Initialize data in a buffer to a specified value.

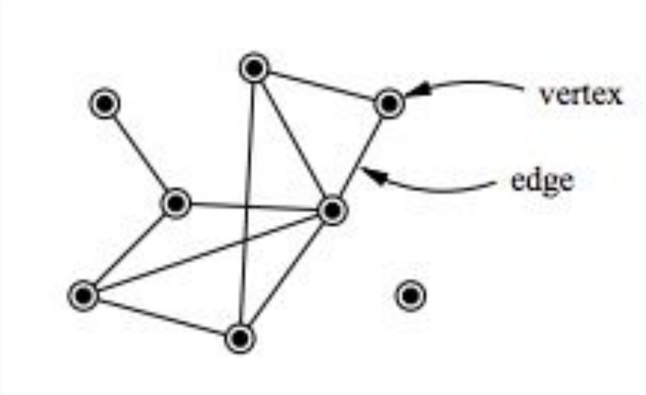
Parallel_for	#pragma omp parallel for
Single_task ~~	#pragma omp critical

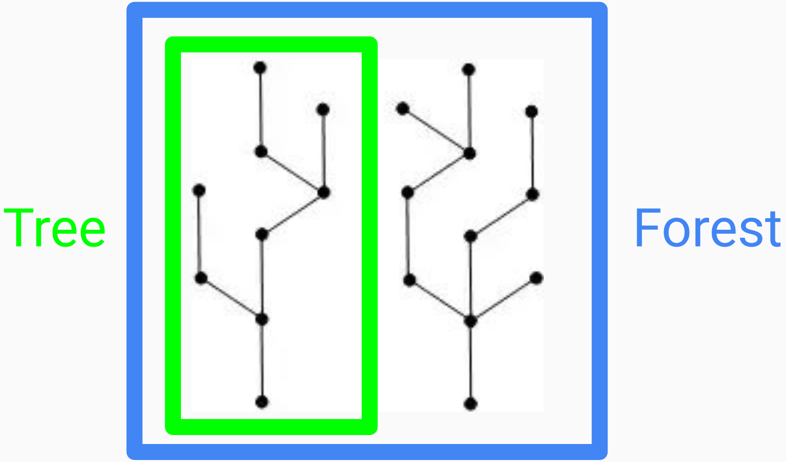
Presenter: Nikhil Kumar

Graph Algorithms

What is a Graph?

- Composed of vertices/nodes and edges
- Edges are used to represent relation between vertices
- Weighted vs Unweighted
- Directed vs Undirected
- Cyclic vs Acyclic





Presenter: Bill Yang

Binary Search

- Used to find (search for) a specific node
- Used on sorted arrays
- Continues to compare middle value with target value until target is found
- Useful for going through sorted databases

Presenter: Bill Yang

Binary Search

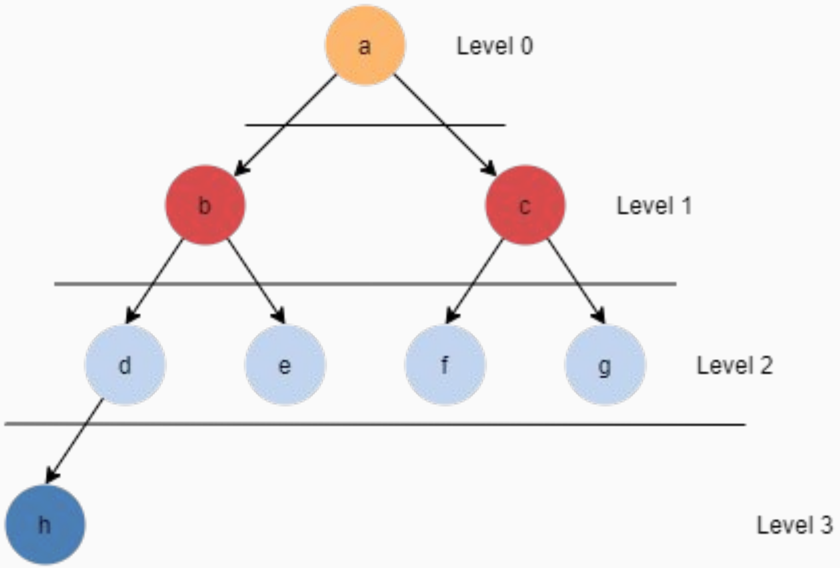
- Naive implementation: Split nodes based on number of processors and perform binary search on each of them
- Parallelize the comparisons by splitting into p groups instead of 2
- Parallelizing multiple binary searches on the same tree

```
parallel for (number of subtrees){
    PerformBinarySearch(subtree,endnode);
}


parallel for (nodes connecting to current node){
    if (the new node is the target node){
        return target node has been found;
    }
    else if (distance from new node to target node is less than the previous closest node) {
        closest node = new node;
    }
    return closest node;
}


parallel for (number of nodes that need to be found){
    PerformBinarySearch(tree,endnode);
}
```

Presenter: Bill Yang

Breadth First Search

- Graph traversal algorithm
- Simple
- Used as the foundation for more complicated algorithms
- Useful in many real-world applications



Presenter: Lenny Schwarz

Breadth First Search

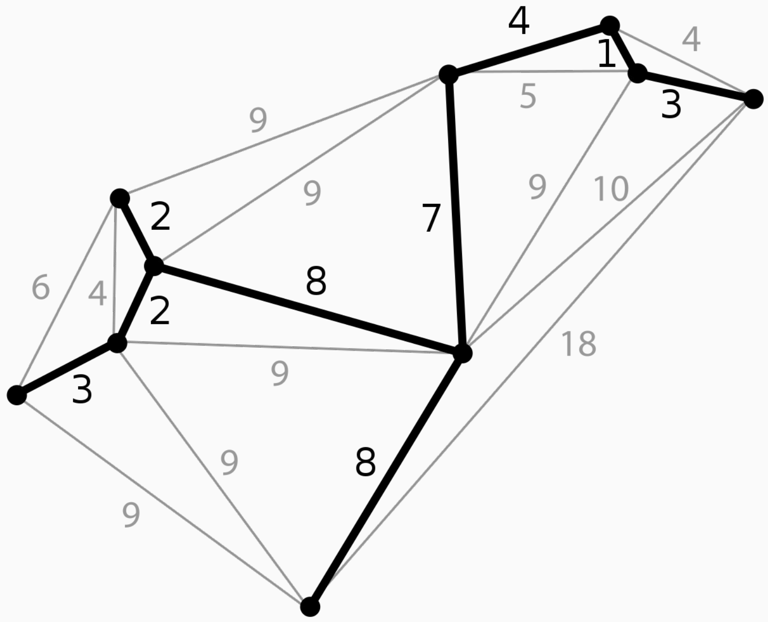
- Process each depth level in parallel
- Each processor assigned a node, searches for children
- Critical sections are needed when queuing and dequeuing
- Pseudocode in OpenMP, we will translate to DPC++

```
while (!q.empty()) {
    qSize = q.size();

    // process entire level in parallel
    #pragma omp parallel for
    for (int i = 0; i < qSize; i++) {
        node* currNode;
        #pragma omp critical
        {
            currNode = q.front();
            q.pop();
        }
        // find children for the current node
        node* childNodes;
        findChildren(currNode);
        #pragma omp critical
        q.push(childNodes);
    }
}
```

Presenter: Lenny Schwarz

Prim’s Algorithm

- An algorithm that finds the minimum spanning tree on a graph
- Greedy algorithm, always makes the local optimal choice
- Simple
- Useful for telecommunications or solving complex mathematical problems



Presenter: Lenny Schwarz

Prim's Algorithm

- Main loop not parallelizable; however, inner loop is
- Slow for large numbers of nodes, we can use parallelisation to overcome this issue
- Current parallel implementation in OpenMP, we will translate to DPC++

```
// process inner loop in parallel
#pragma omp parallel for
for(j = 0; j < prim.dim; j++) {
    // find the minimum weight
    if(prim.edge[prim.U[i]][j] > minDist || prim.edge[prim.U[i]][j]==0) {
        continue;
    } else {
        #pragma omp critical
        {
        }
    }
}
```

```
h.parallel_for(nd_range{{size}, {16}}, [=](nd_item<1> item) {
  auto sg = item.get_sub_group();
  ...
  sg.barrier();
  ...
});
```

Presenter: Lenny Schwarz

Questions?