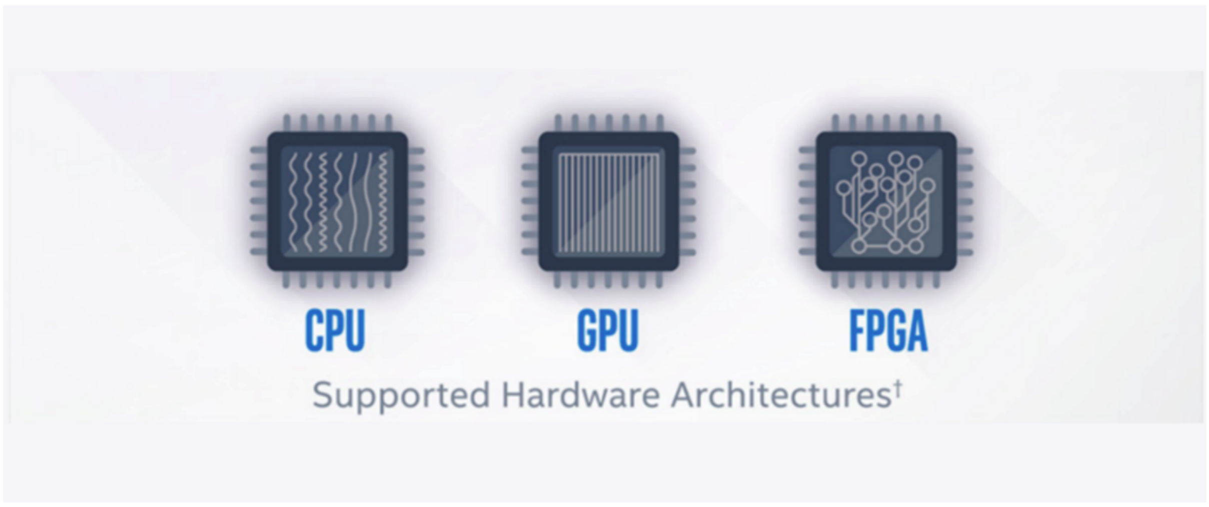
PRESENTATION NOTES

DPC++

What is DPC?

Data parallel C++ is a new language, which is the ISO C++ specification with Khronos Group SYCL. It is an open source implementation of the oneAPI specification to make heterogeneous computation more accessible and widespread.

The basic idea is that a single C++ application can use a wide range of devices on a heterogeneous system, specifically, It allows for the creation of explicit parallelism on a diverse range of compute architectures. Devices such as CPUs, GPUs, FPGAs, AI accelerators or any other custom ASIC accelerator can have DPC++ code run on them to do any form of parallel computation, all coming from that one single source, C++ application.



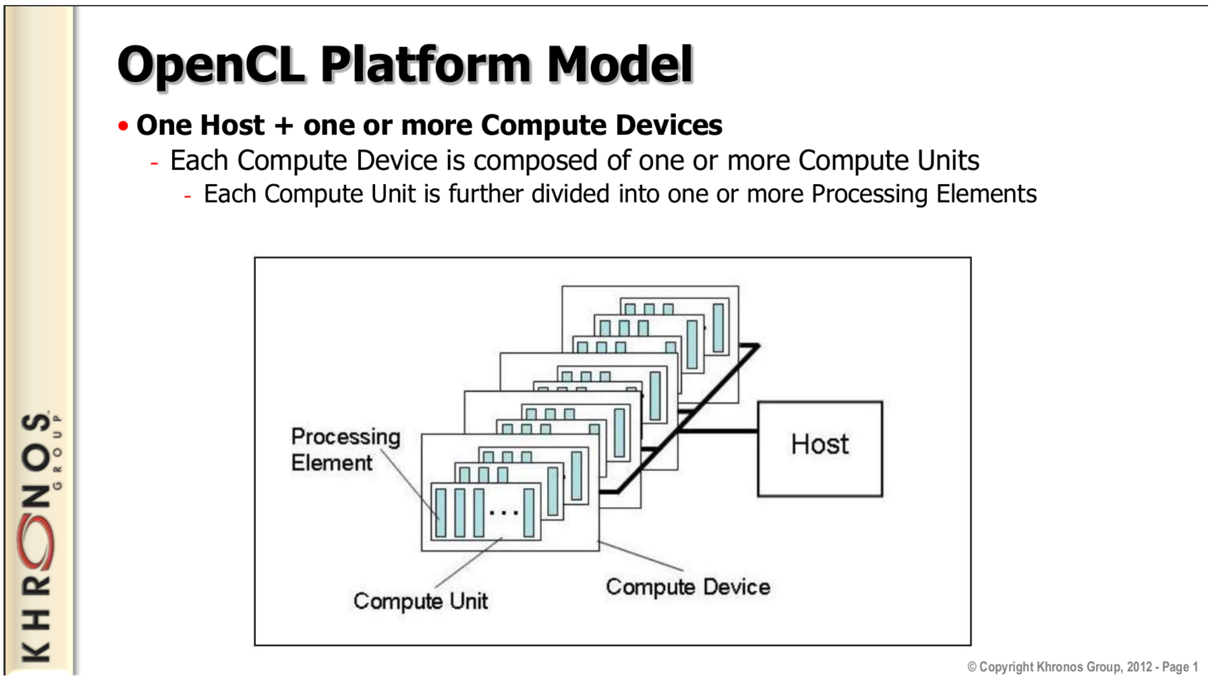
How does it work?

DPC++ is basically an API extension of SYCL. SYCL is the base which allows for being able to program for various hardware targets. These API extensions are often then implemented in the SYCL standard if the implementation in DPC++ or any other community extension is proven.

SYCL™

The idea behind SYCL is to have a single source file, which contains a host and device code, and to be able to code for both. SYCL was made for openCL but has since moved away to be made more generic. There are some similarities between openCL and SYCL (such as the host/device model.)

That's why when pulling LLVM for DPC++, it is under the SYCL branch of LLVM.



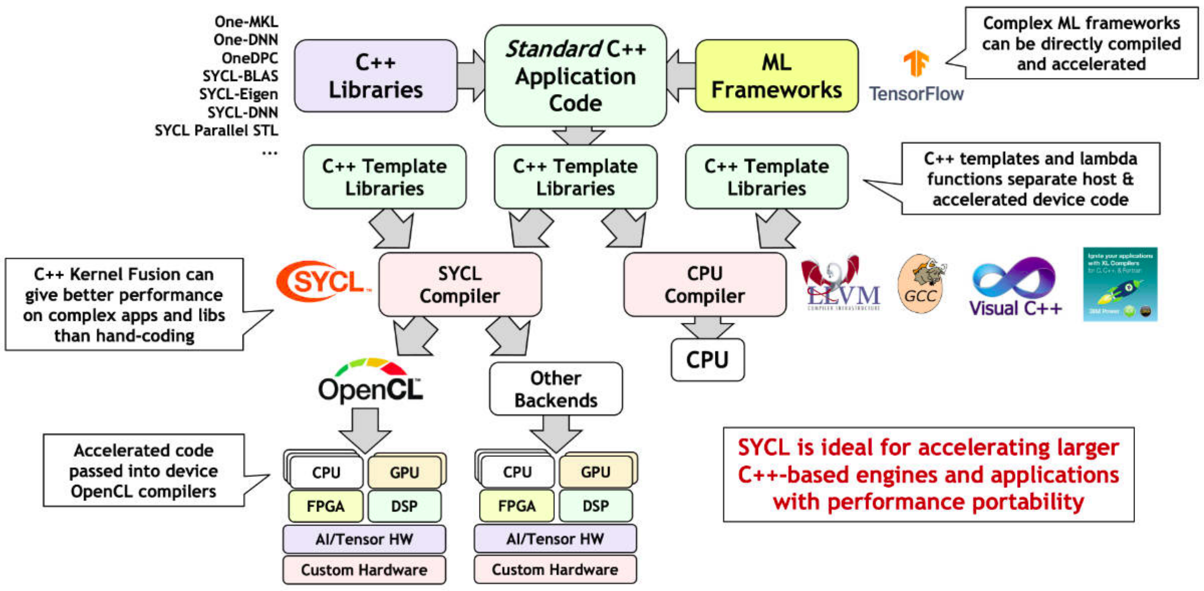
OpenCL Platform Model

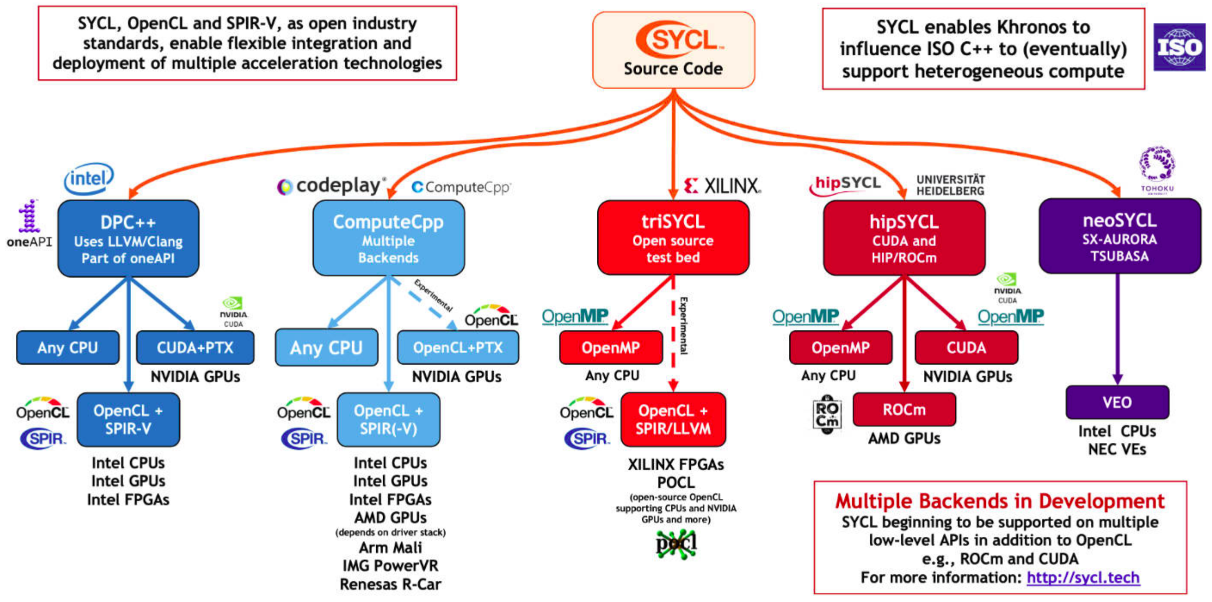
- **One Host + one or more Compute Devices**
  - Each Compute Device is composed of one or more Compute Units
    - Each Compute Unit is further divided into one or more Processing Elements

Processing Element
Host
Compute Unit
Compute Device
KHRONOS GROUP

© Copyright Khronos Group, 2012 - Page 1

;


One-MKL
One-DNN
OneDPC
SYCL-BLAS
SYCL-Eigen
SYCL-DNN
SYCL Parallel STL
...
C++ Libraries
Standard C++ Application Code
ML Frameworks
TensorFlow
Complex ML frameworks can be directly compiled and accelerated
C++ Template Libraries
C++ Template Libraries
C++ Template Libraries
C++ templates and lambda functions separate host & accelerated device code
C++ Kernel Fusion can give better performance on complex apps and libs than hand-coding
SYCL
SYCL Compiler
CPU Compiler
LLVM
GCC
Visual C++
CPU
OpenCL
Other Backends
Accelerated code passed into device OpenCL compilers
CPU
GPU
FPGA
DSP
AI/Tensor HW
Custom Hardware
CPU
GPU
FPGA
DSP
AI/Tensor HW
Custom Hardware
SYCL is ideal for accelerating larger C++-based engines and applications with performance portability


SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies
SYCL Source Code
SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute
ISO
intel
oneAPI
DPC++ Uses LLVM/Clang Part of oneAPI
Any CPU
CUDA+PTX
NVIDIA GPUs
OpenCL + SPIR-V
Intel CPUs
Intel GPUs
Intel FPGAs
codeplay
ComputeCpp
ComputeCpp Multiple Backends
Experimental
Any CPU
OpenCL+PTX
NVIDIA GPUs
OpenCL + SPIR(-V)
Intel CPUs
Intel GPUs
Intel FPGAs
AMD GPUs
(depends on driver stack)
Arm Mali
IMG PowerVR
Renesas R-Car
XILINX
triSYCL Open source test bed
Experimental
OpenMP
Any CPU
OpenCL + SPIR/LLVM
XILINX FPGAs
POCL
(open-source OpenCL supporting CPUs and NVIDIA GPUs and more)
hipSYCL
UNIVERSITÄT HEIDELBERG
hipSYCL CUDA and HIP/ROCm
OpenMP
Any CPU
CUDA
NVIDIA GPUs
ROCm
AMD GPUs
TOHOKU
neoSYCL SX-AURORA TSUBASA
VEO
Intel CPUs
NEC VEs
Multiple Backends in Development
SYCL beginning to be supported on multiple low-level APIs in addition to OpenCL
e.g., ROCm and CUDA
For more information: http://sycl.tech

Why?

DPC++/SYCL are similar in nature to OpenMP and share the goal of making C++ more heterogeneous. Making heterogeneous computation more accessible means that more programs can start to take advantage of an ever growing number of heterogeneous systems. SYCL is part of the oneAPI dream, one unified programming model that could be used to serve both the HPC and the embedded industries.

Good for us as we can take advantage of simpler parallelism to make operations faster

Key features:

Key Features: Programming Concepts

Single Source
As explained above, a single C++ application made in a single file, can be the only thing used to create and compile these multiprocessor applications. All of which is done by writing C++ and no other language (Eww VHDL???)

The resulting executables are multi architecture binaries, fat binaries.

Selecting and running for many different systems and devices

Programming and compilation targets

Host Device model
    Hosts

Often the CPU in the system (does not have to be, however must implement C++17) is considered the host of the system, where most of the code is run. The Host can be considered the conductor of the entire application

Devices

Devices are the target for the acceleration offload. Meaning that the computation work is offloaded from the host to the device to then accelerate the computation of that piece of work.

Code for these devices are considered kernels, same as in openCL or CUDA. Kernel code has some restrictions but allows for massive parallelism
Restrictions:

- Dynamic polymorphism
- Dynamic memory allocation (new or delete)
- Static variables
- Function pointers
- Exception handling

Asynchronous Execution

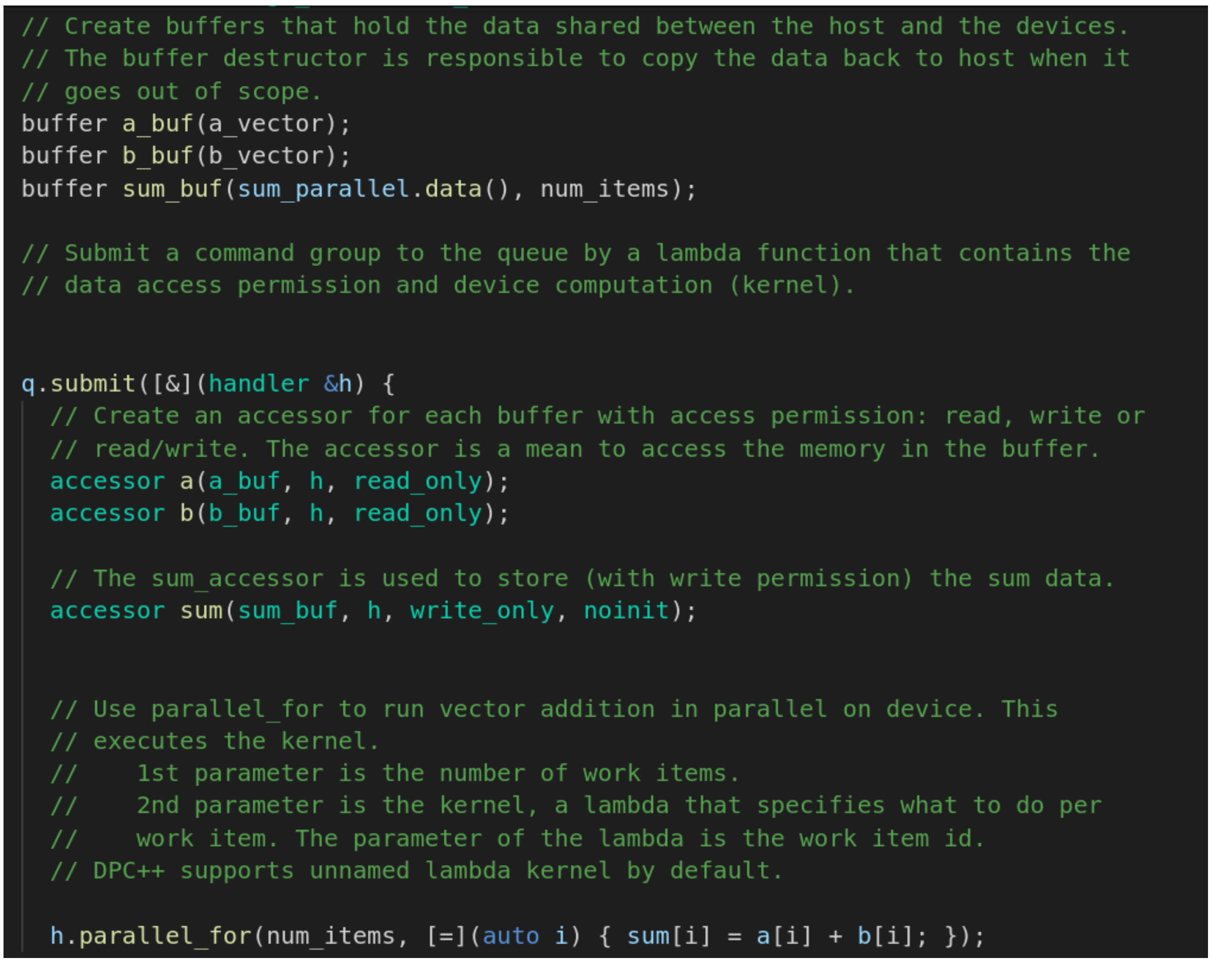
When a piece of work gets sent to the device to be computed, the requests are done so asynchronously. This is a really important feature, as once the device has received the work, the host will move onto the next section in code. If this was not the case, then we would find the host and/or other devices in the system waiting for a single device, and as we know with amdahl's law, we are penalized for having sequential sections. There should be no busy waiting.

If this is done right, then your application can be very scalable, however, done wrong and you come across the issues with parallel programming, mainly race conditions.

Key Features: Language:

Basic C++ :

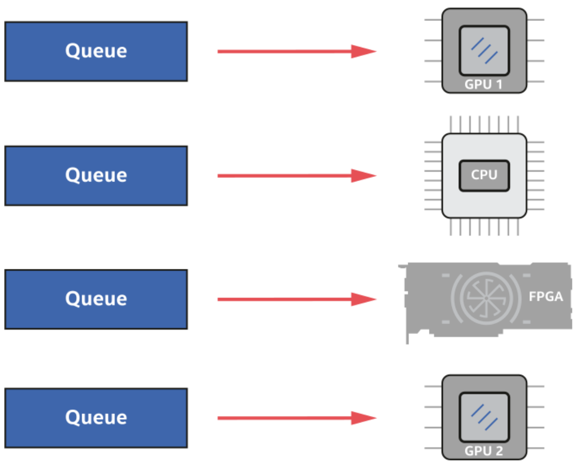
Many of you would already know C++. DPC++ is very familiar, however there are the DPC/SYCL specific parts which allow for explicitly running on accelerator devices with parallelism in mind.

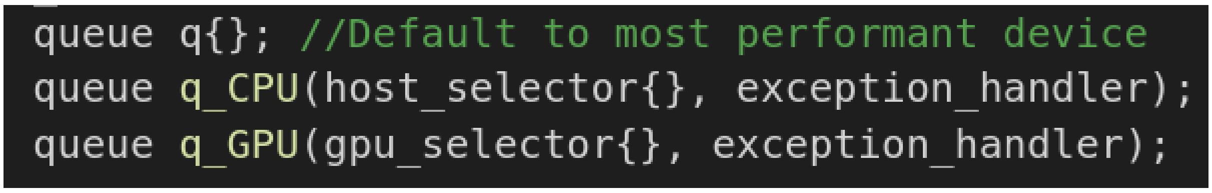


```
// Create buffers that hold the data shared between the host and the devices.
// The buffer destructor is responsible to copy the data back to host when it
// goes out of scope.
buffer a_buf(a_vector);
buffer b_buf(b_vector);
buffer sum_buf(sum_parallel.data(), num_items);

// 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]; });
```

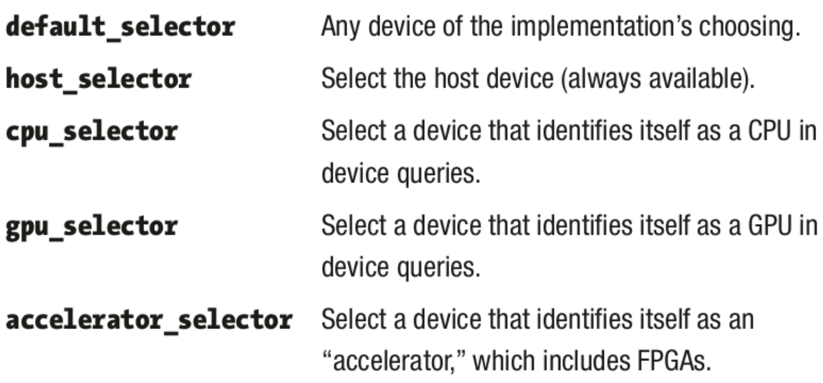
Queues
Queues are an abstraction which allows for sending and running work on a device.
A queue is generally bound to a single device at all times. In this way, we can say it is the main way other devices are exposed to the C++ Application.

Multiple queues can be used for the same device. However a queue cant be mapped to multiple devices, usually it is a one to one pairing.





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



default_selector	Any device of the implementation’s choosing.
host_selector	Select the host device (always available).
cpu_selector	Select a device that identifies itself as a CPU in device queries.
gpu_selector	Select a device that identifies itself as a GPU in device queries.
accelerator_selector	Select a device that identifies itself as an “accelerator,” which includes FPGAs.

The Queue has a submit method in which work is sent to the device to be computed. This block is known as a command group, which contains the device code and any thing else required to run , such as accessors for buffers

Memory management:

Buffers Accessors
Buffers act a high level abstraction for data, and the underlying data can have many different locations in memory. Memory can be shared between the hosts and the devices. The buffer simply represents the data in a consistent way. As such an accessor must be used on a buffer to safely access data.

Unified Shared Memory
Memory can also be represented as shared memory with pointers, with device, host or shared memory. It’s a lower level abstraction of Buffers, as it requires more memory management, but it's more familiar to C++ programmers.

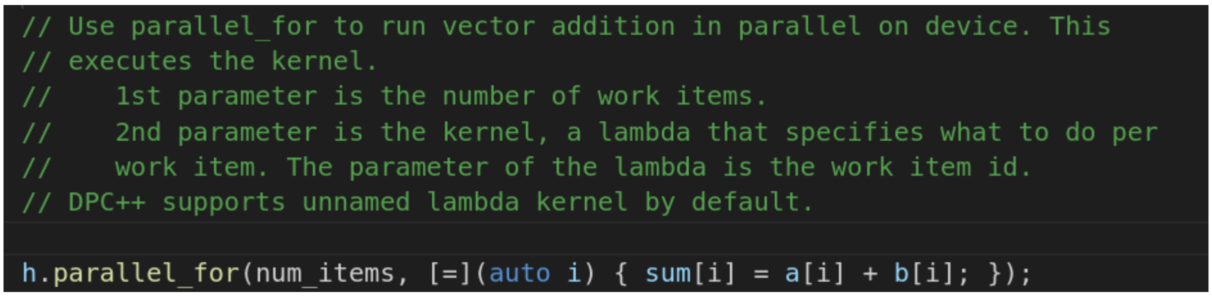
Kernels

A kernel in DPC++ refers to code which will run on the device. Kernels are a portable piece of code which relate to the kind of parallel operation being performed, rather than having any specific details about threads, SIMD instructions or anything else, instead the described parallelism is mapped to the hardware parallelism elements available on the device.

You can see how this is somewhat similar to OpenMP, as rather than specifying any hardware specific details, the parallelism is described in abstract and then implemented in on the device.

These kernels are commonly represented as a lambda function.

C++ Lambdas



```
// 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]; });
```

General form is basically an anonymous function, where parallel_for takes in a range it needs to work on, and then a lambda function.

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

The lambda function requires a capture list, what values are captured, either by reference or value. In SYCL this is usually always [=] captured automatically by value.

Parameters are passed in, like arguments in a function. The idea is that the indexes are passed through this parameter input.

Then there is the lamda body, which contains the code required of the kernel.

Kernels can be function objects, but outputs are not automatically captured, and it is recommended to be lambda expressions.

ND-range as in Nth Dimension

Can be used for explicit execution of a specific range of items, rather than executing everything all at once together. Allows us to prescribe how to map work to a group.

This allows for

- Taking advantage of Memory locality

- Memory barriers for synchronization

Example Program

Parallel Graph Algorithms

So know that we know what DPC++ is, let’s look at graphs. Firstly, what is a graph?

What are Graphs
A graph is ‘a structure amounting to a set of objects in which some pairs of the objects are "related"’. Objects are usually referred to as vertices or nodes (**image in powerpoint**) and relations are typically referred to as ‘edges’. Edges can be given weights, indicating an attribute of the relation, such as the amount of time it takes to travel from node A to node B on a map. Edges can also be directional, such as in a decision tree. An example of a directional graph could be the owing of money, with weights being the amount of money which one person owes another. With this small set of features, graphs can be expanded to various applications, ~~with one of the most notable being Google Maps.~~

A subset of graphs which we will be considering in some algorithms are trees, which are graphs which do not contain any cycles. Visually, this can be expressed as a graph where the edges never make a loop, but more importantly this signifies that for any two vertices in the graph, there is only one path which connects them. Following the same logic, a forest is a graph which is a collection of trees.

Binary Search

Binary search is an algorithm which searches for a result (usually in an array, list, or a node in a graph). Although binary search is highly efficient, outperforming linear search (checking if each node is the destination node) by greater and greater margins as the array increases in size, the caveat of binary search is that it only works on sorted data. Binary search is a method which most people use without realising, with a commonly used example being searching for a word in the dictionary. It operates by taking the middle value of a sorted array, comparing the destination to the middle value, and then discarding half of the remaining array. **Explain example here**

Generalization to graphs
Binary search has been generalized to work on certain types of graphs, where the target value is stored in a vertex instead of an array element. Binary search trees are one such generalization—when a vertex (node) in the tree is queried, the algorithm either learns that the vertex is the target, or otherwise which subtree the target would be located in.

Parallel Binary Search:

The naive method of parallelising binary search is to first check that none of the child nodes of the starting node are the target destination (as well as the starting node itself), then allocating each subtree to a processor. While this method will perform slightly faster than a sequential binary search, as there are less comparisons for each processor to perform, this should not be confused with increasing efficiency. Since most subtrees will not contain the target destination, they will be performing comparisons which will never reach the destination.

Another method which can be implemented is splitting the remaining nodes into p groups (where p is the number of processors) instead of only 2 groups. The reasoning behind this method is that the performance boosts generated by binary search over linear search come from the algorithm's ability to dismiss large amounts of subtrees (hence nodes) which may contain the target node. Whether this method produces a performance benefit depends greatly on the graph and the implementation. If the graph is not connected in a form which allows the algorithm to easily divide the subtrees for comparisons, the performance of this method will greatly decrease. Another factor which greatly affects the viability of this implementation is the speed of writing and synchronization. Since the processor with the 'winning' subtree would have to write some kind of indication of their subtree being the winning tree for the algorithm to move onto the next iteration on top of the usual comparison to figure out if they possess the winning subtree. The synchronisation between each iteration will decide whether this method provides any form of benefit to execution time.

The final method of parallelization for a binary search tree would be the parallelisation of multiple searches. This may be useful in scenarios where

What Graph algorithms are there currently / Are parallelizable:

Graphs

In sequential execution,

Depth-first search has been called 'a nightmare for parallel programming' due to it's largely sequential nature, as it is difficult to know

How it works in seq
How to parallel

Breadth first search

- Can be parallelized

Depth first search

- 'A nightmare for parallel programming' so probs avoid it

Minimum Spanning Tree
- Prims algorithm - Parallelizable

Divide and conquer algorithm

Binary Search

Ones we have chosen algorithms:
- BFS - https://en.wikipedia.org/wiki/Parallel_breadth-first_search
- Prims Algorithm - https://en.wikipedia.org/wiki/Prim%27s_algorithm
- Parallel Binary Search
    - Binary search is more of a array algorithm rather than a graph one, but it can work on graphs
- Delta stepping(?) - seems kinda hard

Breadth first Search

Background

Breadth first Search is an Algorithm used for traversing a graph. Graph traversal refers to the process of starting at a node and then visiting every other node in the graph. The algorithm explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

The functionality revolves around the use of queues following the first in first out policy (data inserted first will be accessed first). The queue is an abstract data structure which works like a real world queue would.

How it works:

BFS is an Algorithm for traversing a graph. Graph traversal refers to the process of starting at a node and then visiting every other node in the graph. The algorithm starts at a “tree” node and then moves from there. From the starting node, BFS investigates all the adjacent nodes to the starting node marking them as found. The algorithm then goes one level deeper, and finds all of the adjacent nodes to the previously found nodes. As we can see, the level 2 nodes are found after all the level one nodes, which are connected to the ‘origin’ node have been marked as found, hence breadth first. This allows BFS to be used in many contexts, such as finding friends on social networks, peer-to-peer networks such as torrent applications and searching for nearby locations on Google Maps if you want to find the closest restaurant or petrol station.

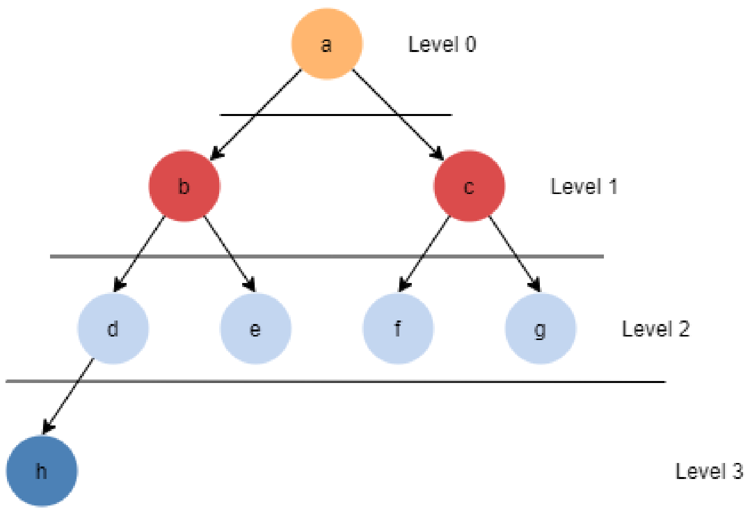
1. Begin with an empty queue
2. Select a starting node and insert it into the queue
3. Extract the node from the queue and insert its child nodes
4. Repeat until all nodes have been visited

Use-Case

Breadth first Search uses the opposite strategy of depth-first search, so instead of exploring the node branch as far as possible before backtracking and expanding to other nodes; it searches all the nodes at the same level of depth before moving on to the next level.

This feature, paired with its simplistic design, makes Breadth first Search useful in many applications. Commonly, it is used in search engine crawlers, gps navigation systems and peer-to-peer networking.

Breadth first Search is also used as a foundation for more complex algorithms such as in an implementation of Djikstra’s algorithm.



Parallelisation

It is possible to process each depth level in parallel since you are essentially performing the same operation on each node; each node can be assigned to a processor and that processor searches for the children of its assigned node. Each processor needs to then queue the children it found.

To avoid assigning multiple processors the same node, and to avoid queuing nodes in the wrong order, we require a critical section when queuing and dequeuing.

Shown here is the openmp pseudocode of a parallel implementation of Breadth first search. When we translate to DPC++, the logic will remain mostly the same, except we will use the DPC++ programming model.

Prim’s Algorithm

Background

Prim’s algorithm is an algorithm that finds a minimum spanning tree based on a particular starting point on a graph. A minimum spanning tree is the minimum path which connects all the nodes on a graph.

It is a greedy algorithm because it makes the local optimal-choice in each iteration, the disadvantage of this is that it may not find the global-optimal solution. In the context of minimum spanning trees, the global-optimal solution is the minimum spanning tree with the shortest *total* path length. This is because there can be multiple minimum spanning trees depending on the starting location.

~~The time complexity of Prim's algorithm varies depending on the data structures used for the graph and paths. We will need to take this into account and test the effect of different data structures in our DPC++ implementation of the algorithm.~~

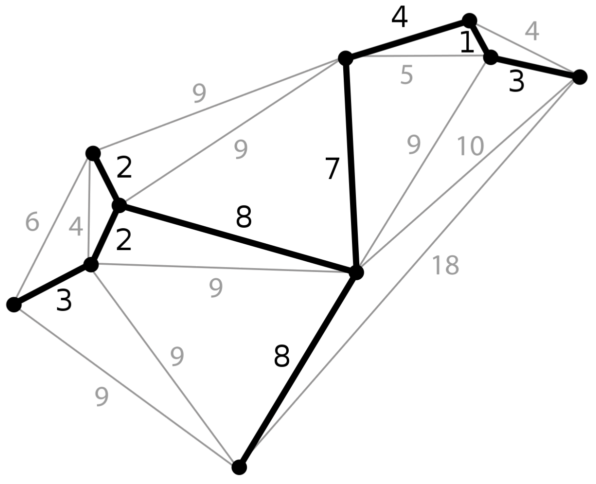
How it works

1. The algorithm begins with a random node
2. The lowest-weighted path to the next available node is selected. In the event of a tie, a random path is chosen
3. This is repeated until there are no more available nodes

Use-Case

Minimum spanning trees are very useful for network designs, such as in telecommunications. With a minimum spanning tree, the telecommunications company can connect the same amount of nodes using less cable, saving them money.

They are also used to find approximate solutions for complex mathematical problems such as the Traveling Salesman Problem. (Finding the shortest yet most efficient route for a person to take given a list of specific destinations).



Parallelisation

The main loop of the algorithm, which refers to the loop that determines which paths are available at each stage, is not parallelizable because it is inherently sequential — You need to build the graph one node at a time. The inner loop however, is parallelizable. This is because the inner loop simply determines which, out of the available paths, has the minimum weight.

A major drawback of Prim's algorithm is that the time taken to check for the smallest weight arc is slow for a large number of nodes; this is because, traditionally, each path weight would have to be checked sequentially. Utilising parallel computing, a large amount of the path weights can be ruled out in parallel as the program converges on the minimum weighted path.

The example shown here is an openmp implementation of the inner loop of the algorithm. One of the ways we can translate this to DPC++ is by using the parallel_for directive to parallelise the for loop and the barrier directive to avoid the race condition when writing to the minimum distance variable.