Data Parallel C++ Essentials

Graphs and Data Dependences

Find out how to use Graphs and Data Dependences in DPC++

Graphs and Dependences

- Agenda
  - Implicit and Explicit memory management
  - Execution Graph Scheduling
  - Graphs and Dependences
    - RAW- Read after Write, WAR- Write after Read and WAW- Write after Write
  - Dependency in Linear dependency chain graphs and Y pattern Graphs
- Hands On
  - Types of Dependences (RAW, WAR and WAW)
  - Linear chain graphs and Y pattern Graphs using USM and Buffers

Learning Objectives

Utilize USM and Buffers and Accessors to apply Memory management and take control over data movement implicitly and explicitly

Utilize different types of data dependences that are important for ensuring execution of graph scheduling

Select the correct modes of dependences in Graphs scheduling.

Memory Management

- Explicitly by the programmer:
  - Explicitly copy data between host and the device.
  - Programmer has full control over when data is transferred between the device and the host and back to host from the device
- Implicitly by the runtime:
  - Controlled by the runtime or driver
  - Less effort on the programmer's part but has less or no control over the behavior of the runtime's implicit mechanisms

Asynchronous Execution

Host

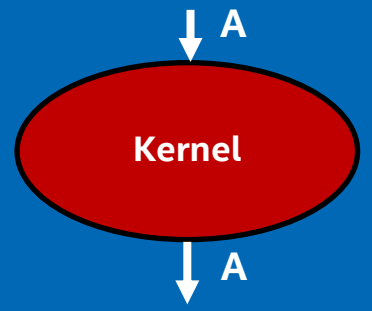
Host code execution

Enqueues kernel to graph, and keeps going

```
#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main() {
  std::vector<int> data(N);
  {
    buffer A(data);
    queue q;
    q.submit([&](handler& h) {
      accessor out(A, h, write_only);
      h.parallel_for(N, [=](auto i) {
        out[i] = i;
      });
    });
  }
  for (int i=0; i<N; ++i) std::cout << data[i];
}
```

Graph

Graph executes asynchronously to host program



Asynchronous Execution

```
int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R }, B{ R };
  queue q;

  q.submit([&](handler& h) {
    accessor out(A, h, write_only);
    h.parallel_for(R, [=](id<1> i) {
      out[i] = i; }); });

  q.submit([&](handler& h) {
    accessor out(A, h, write_only);
    h.parallel_for(R, [=](id<1> i) {
      out[i] = i; }); });

  q.submit([&](handler& h) {
    accessor out(B, h, write_only);
    h.parallel_for(R, [=](id<1> i) {
      out[i] = i; }); });

  q.submit([&](handler& h) {
    accessor in(A, h, read_only);
    accessor inout(B, h);
    h.parallel_for(R, [=](id<1> i) {
      inout[i] *= in[i]; }); });
}
```

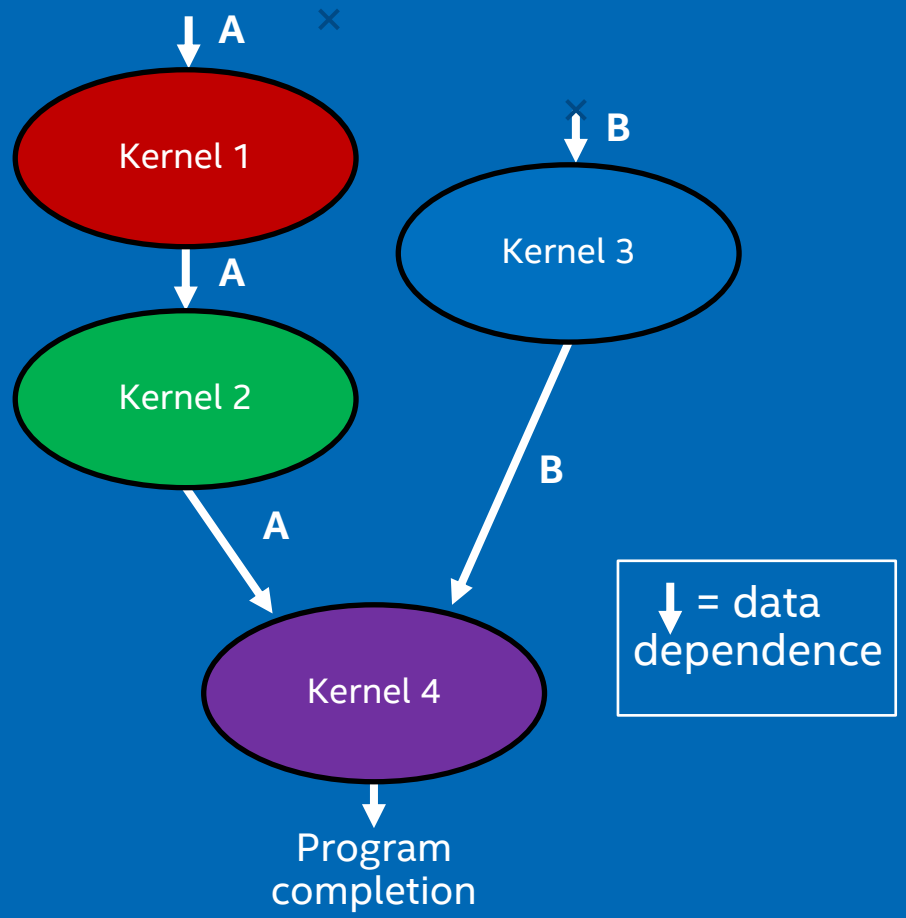
Kernel 1

Kernel 2

Kernel 3

Kernel 4

Automatic data and control dependence resolution!



Execution Graph Scheduling

Mechanism to achieve proper sequencing of kernels, and data movement in a DPC++ application.

- Read-after-Write (RAW) : Occurs when one task needs to read data produced by a different task.
- Write-after-Read (WAR) : Occurs when one task needs to update data after another task has read it.
- Write-after-Write (WAW) : Occurs when two tasks try to write the same data.

Read after Write (RAW)

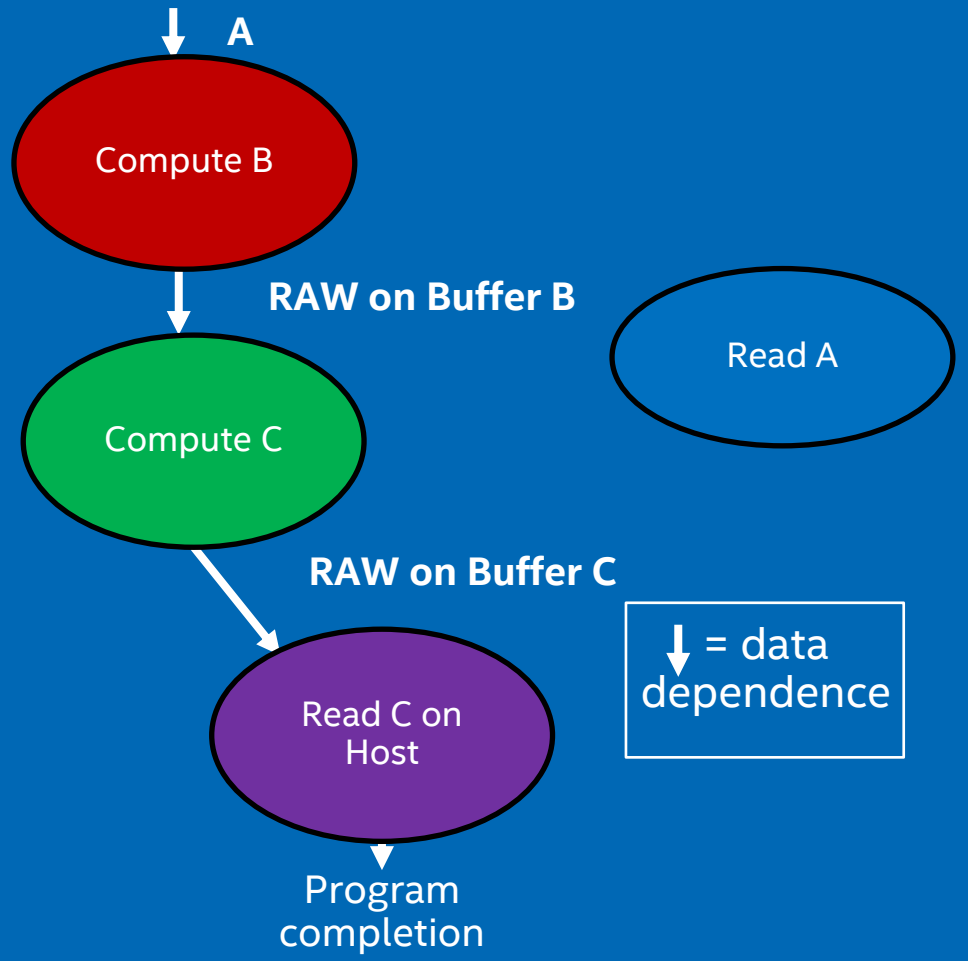
```
int main() {
  queue Q;
  //Create Buffers
  buffer A{a};
  buffer B{b};
  buffer C{c};
  Q.submit([&](handler &h) {
      accessor accA(A, h, read_only);
      accessor accB(B, h, write_only);
      h.parallel_for( // computeB
        N, [=](id<1> i) { accB[i] = accA[i] + 1; });
    });
  Q.submit([&](handler &h) {
      accessor accA(A, h, read_only);
      h.parallel_for( // readA
        N, [=](id<1> i) {
          // Useful only as an example
          int data = accA[i];
        });
    });
  Q.submit([&](handler &h) {
      // RAW of buffer B
      accessor accB(B, h, read_only);
      accessor accC(C, h, write_only);
      h.parallel_for( // computeC
        N,  [=](id<1> i) { accC[i] = accB[i] + 2; });
    });
  // read C on host
  host_accessor host_accC(C, read_only);
    std::cout << "\n";
  return 0;
}
```

Kernel 1

Kernel 2

Kernel 3

Automatic data and control dependence resolution!



Write After Read and Write after Write

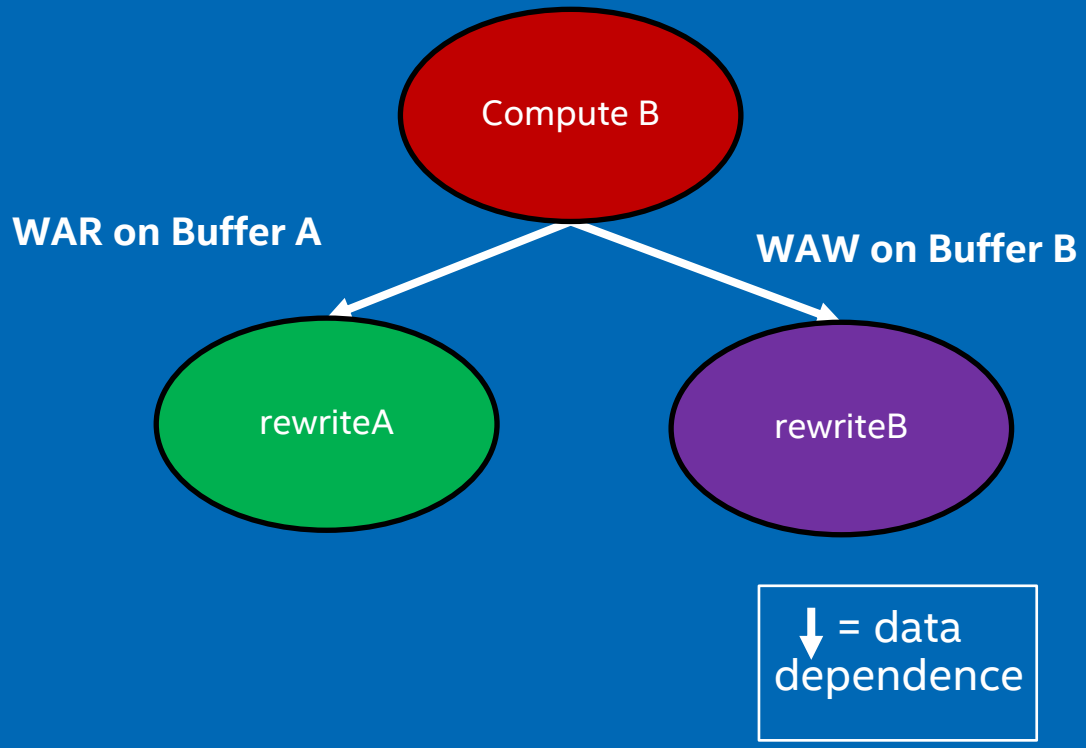
```
queue Q;
buffer A{a};
buffer B{b};
Q.submit([&](handler &h) {
    accessor accA(A, h, read_only);
    accessor accB(B, h, write_only);
    h.parallel_for( // computeB
        N, [=](id<1> i) {
        accB[i] = accA[i] + 1;
        });
    });
Q.submit([&](handler &h) {
    // WAR of buffer A
    accessor accA(A, h, write_only);
    h.parallel_for( // rewriteA
        N, [=](id<1> i) {
        accA[i] = 21 + 21;
        });
    });
Q.submit([&](handler &h) {
    // WAW of buffer B
    accessor accB(B, h, write_only);
    h.parallel_for( // rewriteB
        N, [=](id<1> i) {
        accB[i] = 30 + 12;
        });
    });
host_accessor host_accA(A, read_only);
host_accessor host_accB(B, read_only);
```

Kernel 1

Kernel 2

Kernel 3

Automatic data and control dependence resolution!



Linear dependency chain graphs and Y pattern Graphs

- Linear dependence chains where one task executes after another
  - First node represents the initialization of data.
  - Second node presents the reduction operation that will accumulate the data.
- “Y” pattern we independently initialize two different pieces of data.
  - An addition kernel will sum the two vectors together.
  - Finally, the last node in the graph accumulates the result into a single value.

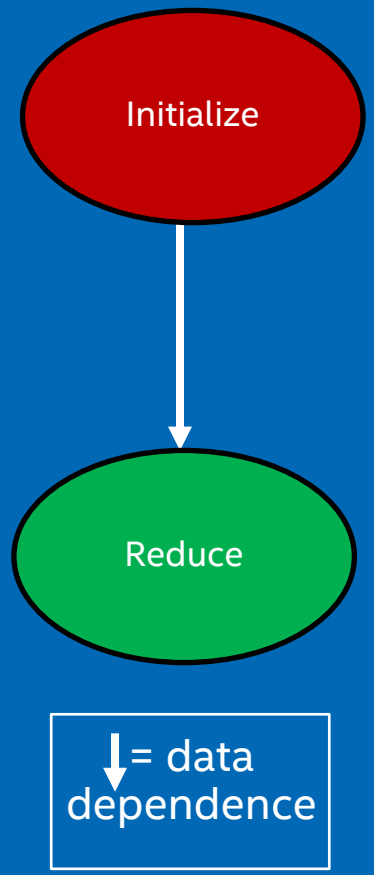
Linear Dependence Using In-order queue

Create In-order queue

Initialize the data in Kernel 1

Kernel 2 sums up the elements

```
constexpr int N = 42;

int main() {
  queue Q{property::queue::in_order()};

  int *data = malloc_shared<int>(N, Q);

  Q.parallel_for(N, [=](id<1> i) { data[i] = 1; });

  Q.single_task([=]() {
      for (int i = 1; i < N; i++)
        data[0] += data[i];
    });
  Q.wait();

  assert(data[0] == N);
  return 0;
}
```

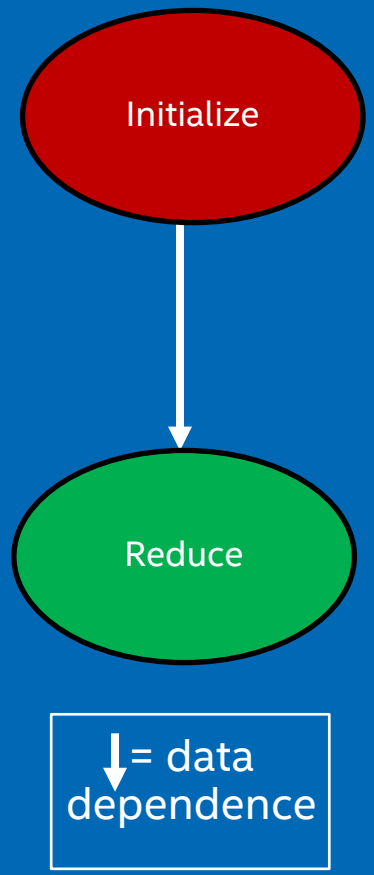


Linear Dependence Using Events

Create event to Initialize the data in Kernel 1

Kernel 2 sums up the elements

```
constexpr int N = 42;

int main() {
  queue Q;

  int *data = malloc_shared<int>(N, Q);

  auto e = Q.parallel_for(N, [=](id<1> i) { data[i] = 1;
});

  Q.submit([&](handler &h) {
      h.depends_on(e);
      h.single_task([=]() {
          for (int i = 1; i < N; i++)
            data[0] += data[i];
        });
    });
  Q.wait();

  assert(data[0] == N);
  return 0;
}
```

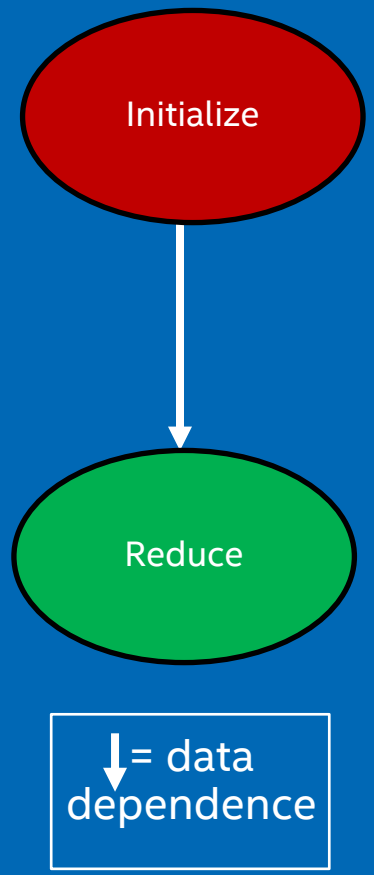


Linear Dependence using Buffers and Accessors

Use Buffers and Accessors to Initialize the data in Kernel 1

Kernel 2 sums up the elements

```
constexpr int N = 42;

int main() {
  queue Q;

  buffer<int> data{range{N}};

  Q.submit([&](handler &h) {
      accessor a{data, h};
      h.parallel_for(N, [=](id<1> i) { a[i] = 1; });
    });

  Q.submit([&](handler &h) {
      accessor a{data, h};
      h.single_task([=]() {
          for (int i = 1; i < N; i++)
            a[0] += a[i];
        });
    });

  host_accessor h_a{data};
  return 0;
}
```



Y Pattern using in-order queues

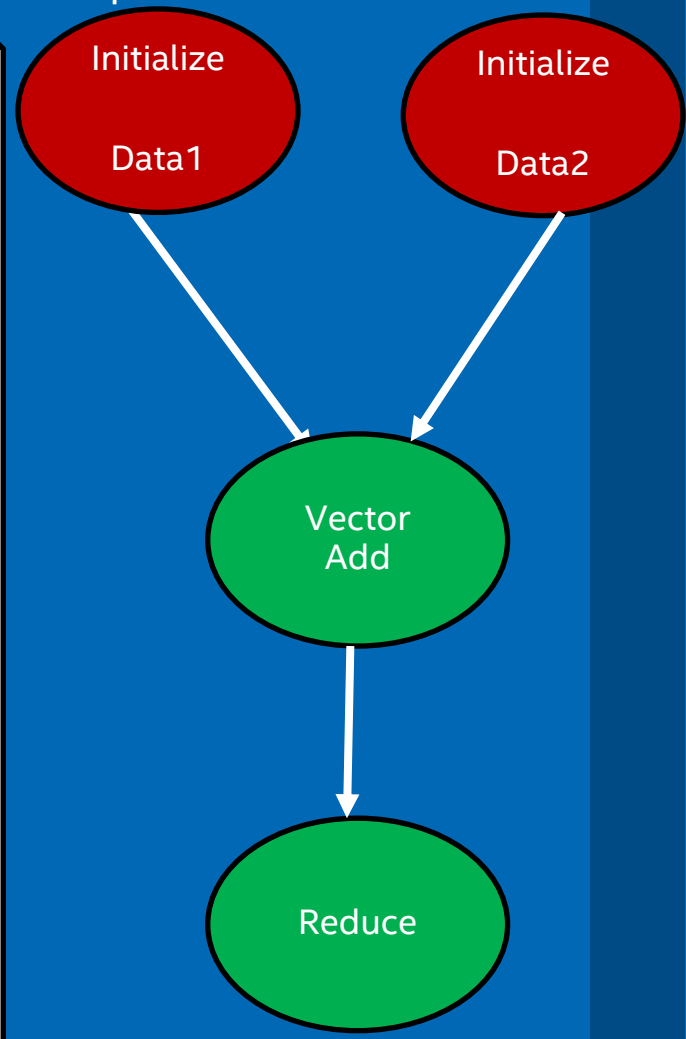
We can see a “Y” pattern using in-order queues in the below example

In-Order Queue

Kernel 3 is dependant on Kernel1 and Kernel2

The final kernel sums up the elements of the first array

```
constexpr int N = 42;

int main() {
  queue Q{property::queue::in_order()};

  int *data1 = malloc_shared<int>(N, Q);
  int *data2 = malloc_shared<int>(N, Q);

  Q.parallel_for(N, [=](id<1> i) { data1[i] = 1; });

  Q.parallel_for(N, [=](id<1> i) { data2[i] = 2; });

  Q.parallel_for(N, [=](id<1> i) { data1[i] += data2[i]; });

  Q.single_task([=]() {
      for (int i = 1; i < N; i++)
        data1[0] += data1[i];

      data1[0] /= 3;
    });
  Q.wait();

  assert(data1[0] == N);
  return 0;
}
```



Y Pattern using Events

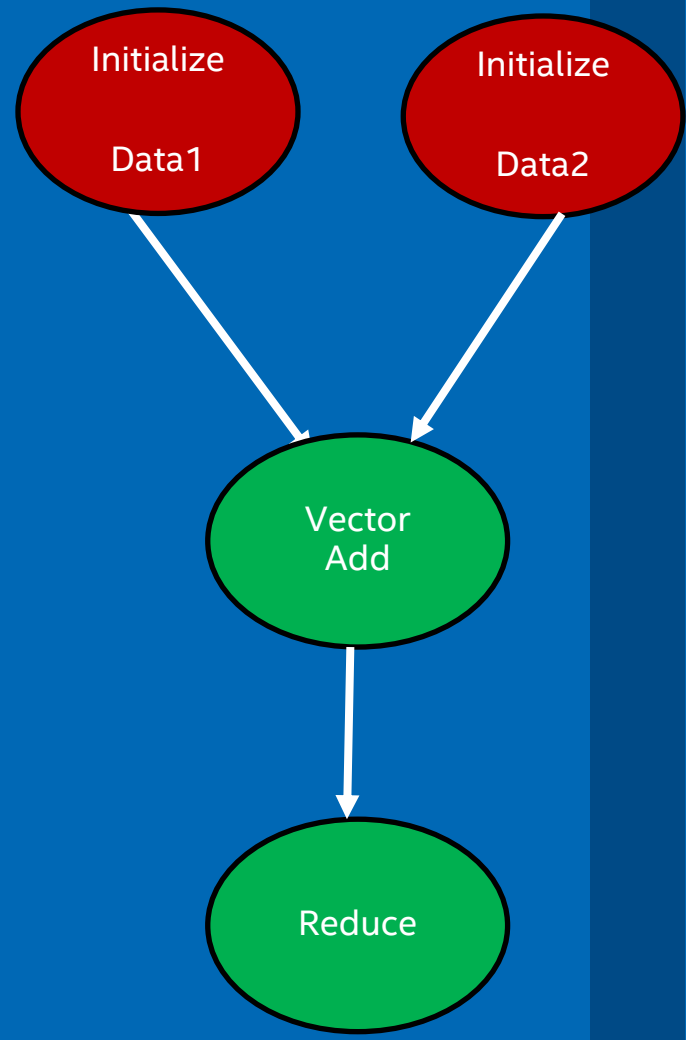
We can see a "Y" pattern using events in the below example

Out of order Queue

Create events for three different kernels. Kernel 3 is dependent on Kernel1 and Kernel2

The final kernel sums up the elements of the first array

```
constexpr int N = 42;

int main() {
  queue Q;

  int *data1 = malloc_shared<int>(N, Q);
  int *data2 = malloc_shared<int>(N, Q);

  auto e1 = Q.parallel_for(N, [=](id<1> i) { data1[i] = 1; });

  auto e2 = Q.parallel_for(N, [=](id<1> i) { data2[i] = 2; });

  auto e3 = Q.parallel_for(range{N}, {e1, e2},
                           [=](id<1> i) { data1[i] += data2[i]; });

  Q.single_task(e3, [=]() {
      for (int i = 1; i < N; i++)
        data1[0] += data1[i];

      data1[0] /= 3;
    });
  Q.wait();

  assert(data1[0] == N);
  return 0;
```



Y Pattern using Buffers and Accessors

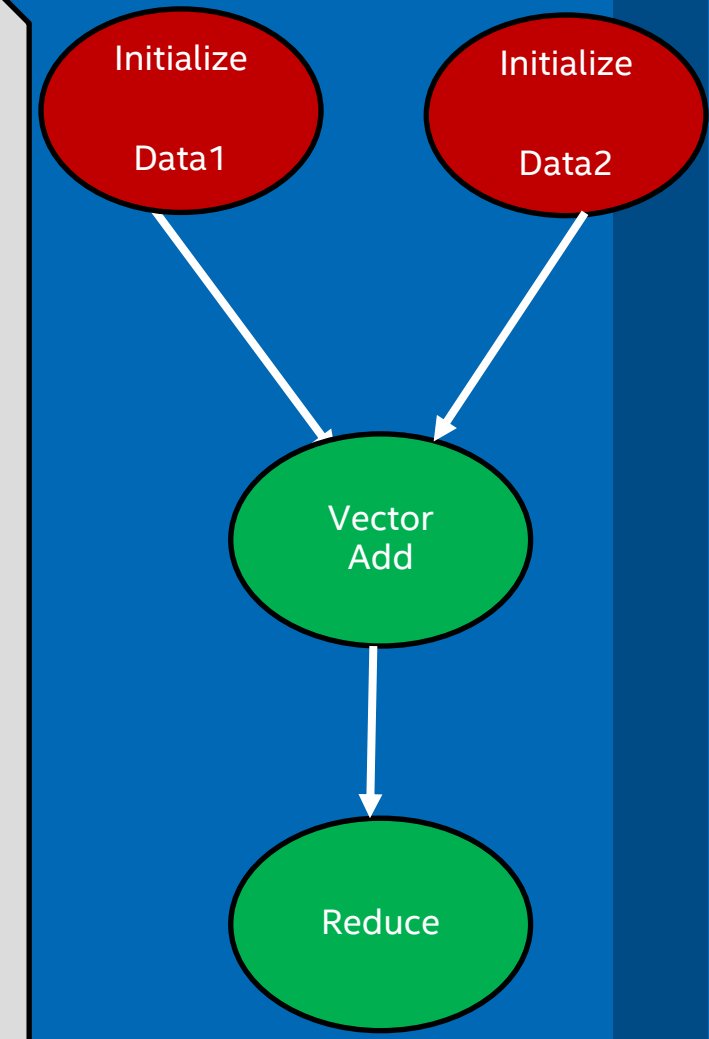
We can see a "Y" pattern using Buffers and Accessors as follows

Kernel 1 and Kernel 2 Initializes data

Kernel 3 waits for Kernel 1 and Kernel 2 to complete.

The final kernel sums up the elements of the first array

```
constexpr int N = 42;
int main() {
  queue Q;
  buffer<int> data1{range{N}};
  buffer<int> data2{range{N}};
  Q.submit([&](handler &h) {
      accessor a{data1, h};
      h.parallel_for(N, [=](id<1> i) { a[i] = 1; });
    });
  Q.submit([&](handler &h) {
      accessor b{data2, h};
      h.parallel_for(N, [=](id<1> i) { b[i] = 2; });
    });
  Q.submit([&](handler &h) {
      accessor a{data1, h};
      accessor b{data2, h, read_only};
      h.parallel_for(N, [=](id<1> i) { a[i] += b[i]; });
    });
  Q.submit([&](handler &h) {
      accessor a{data1, h};
      h.single_task([=]() {
          for (int i = 1; i < N; i++)
            a[0] += a[i];
          a[0] /= 3;
        });
    });
  host_accessor h_a{data1};
  return 0;
}
```



Hands-on Coding on Intel DevCloud

Graphs and Dependences

Summary

In this module you learned:

Utilize different types of data dependences that are important for ensuring execution of graph scheduling

Select the correct modes of dependences in Graphs scheduling.


intel