AVT006: Heterogeneous Programming: Distributed Data Structures, Algorithms, and Views in C++

Benjamin Brock, Intel Labs

Notices and Disclaimers

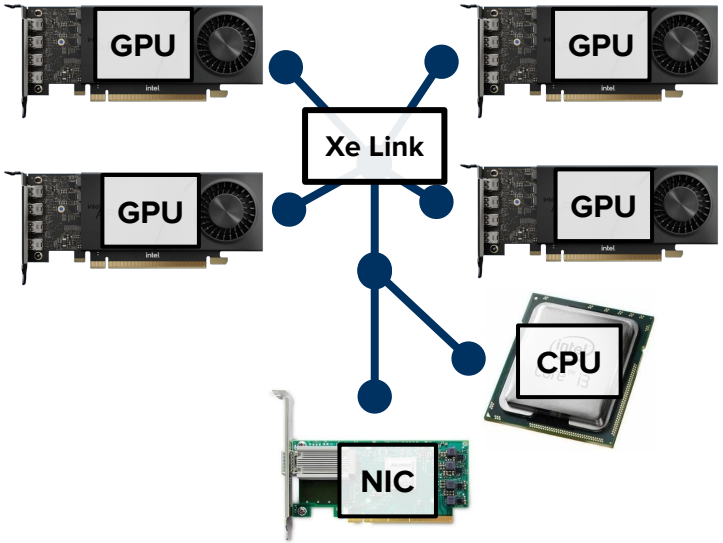
For notices, disclaimers, and details about performance claims, visit www.intel.com/PerformanceIndex or scan the QR code:

Problem: writing **parallel programs** is hard

- **Multi-GPU**, **multi-CPU** systems require **partitioning** data

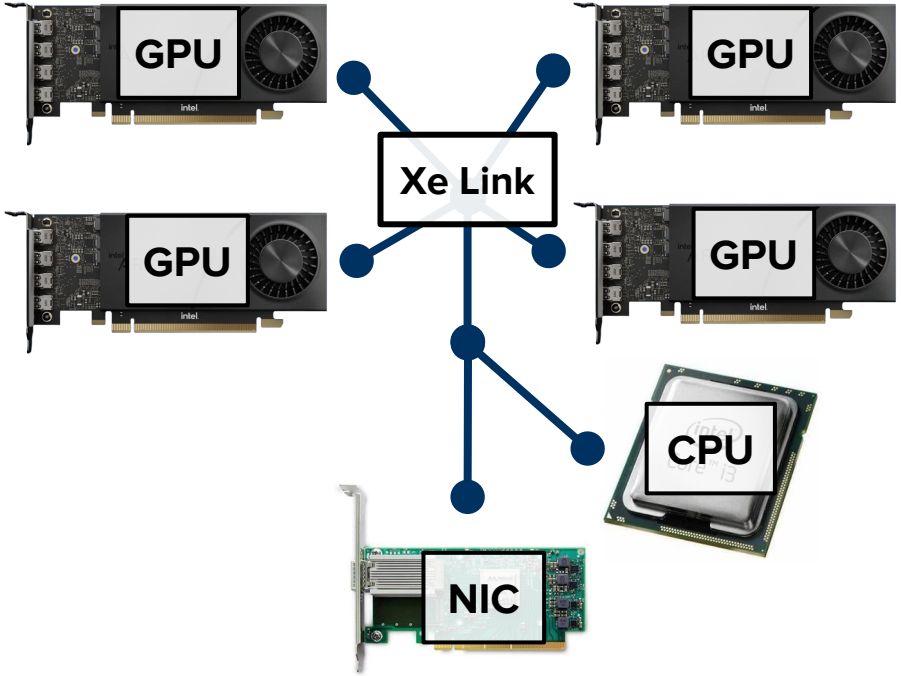
- Users must **manually split up data** amongst GPUs / nodes

- High-level mechanisms for **data distribution** / **execution** necessary.



Multi-GPU Systems

- NUMA regions:
  - **4+ GPUs**
  - **2+ CPUs**

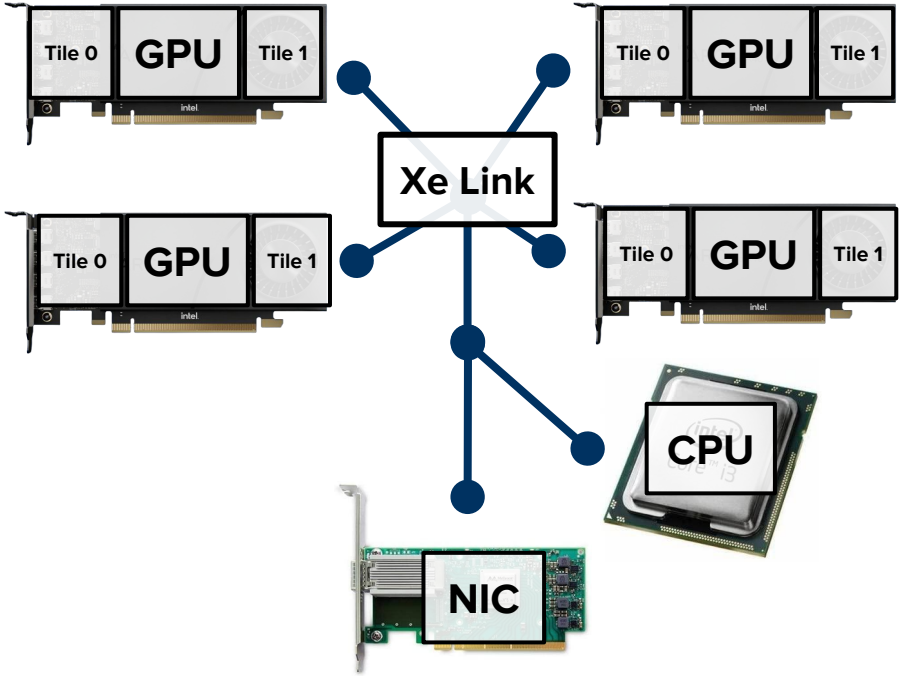


Multi-GPU Systems

- NUMA regions:
  - **4+ GPUs**
  - **2+ CPUs**

- Systems becoming more **hierarchical**: even more **memory domains**

- Software needed to **reduce complexity**



Project Goals

- Offer high-level, standard C++ **distributed data structures**
- Support **distributed algorithms**
- Achieve **high performance** for both **multi-GPU, NUMA,** and **multi-node** execution

```
float dot_product(vector<float>& x,
                  vector<float>& y) {

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
              auto [a, b] = element;
              return a * b;
            });

  return reduce(par_unseq, z, 0, std::plus());
}
```

Agenda

1. C++ standard parallelism
2. Distributed data structures
3. Code Demo
4. Performance
5. Complex data structures

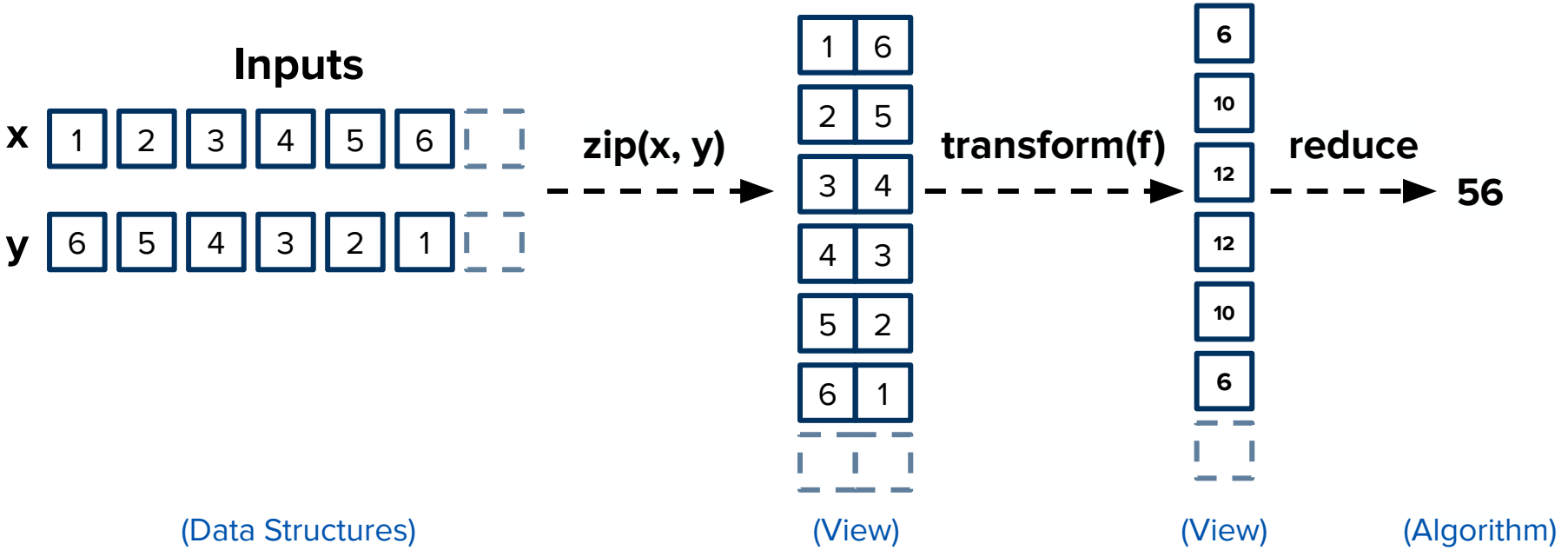
C++ Standard Parallelism

C++ Parallelism

- **Data structures**
  - Organize data
- **Views**
  - Provide modified views of data
- **Algorithms**
  - Operate on and modify data

```
using namespace std;
using namespace std::ranges;
using namespace std::execution;


float dot_product(vector<float>& x,
                  vector<float>& y) {

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
           });

  return reduce(par_unseq, z, 0, std::plus());
}
```

Dot Product Algorithm



Standard C++ Parallelism

- **Data structures**
  - Organize data
- **Views**
  - Lightweight, modified views of data
- **Algorithms**
  - Operate on and modify data

```
using namespace std;
using namespace std::ranges;
using namespace std::execution;


float dot_product(vector<float>& x,
                  vector<float>& y) {

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
           });

  return reduce(par_unseq, z, 0, std::plus());
}
```

Standard C++ Parallelism

- **Extensible:** with extensions, can automatically run on GPU
- All depends on **ranges**, concept for **iterating over data**

```
using namespace std;
using namespace std::ranges;
using namespace std::execution;


float dot_product(vector<float>& x,
                  vector<float>& y) {

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
           });

  return reduce(par_unseq, z, 0, std::plus());
}
```

Standard C++ Parallelism

- **Extensible:** with extensions, can automatically run on GPU

- All depends on **ranges**, concept for **iterating over data**

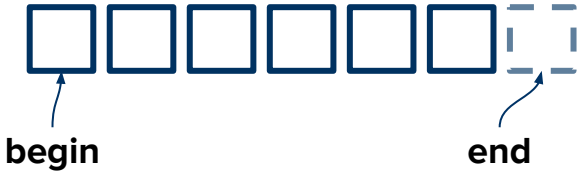
```
using namespace std;
using namespace std::ranges;
using namespace std::execution;
using namespace oneapi;

float dot_product(device_vector<float>& x,
                  device_vector<float>& y) {

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
              auto [a, b] = element;
              return a * b;
            });

  auto dpl_policy = ...;

  return dpl::reduce(dpl_policy, z, 0, std::plus());
}
```

Ranges

C++ 20 introduced **ranges**

A **range** is a **collection of values**

Range concepts provide a **standard way** to iterate over values



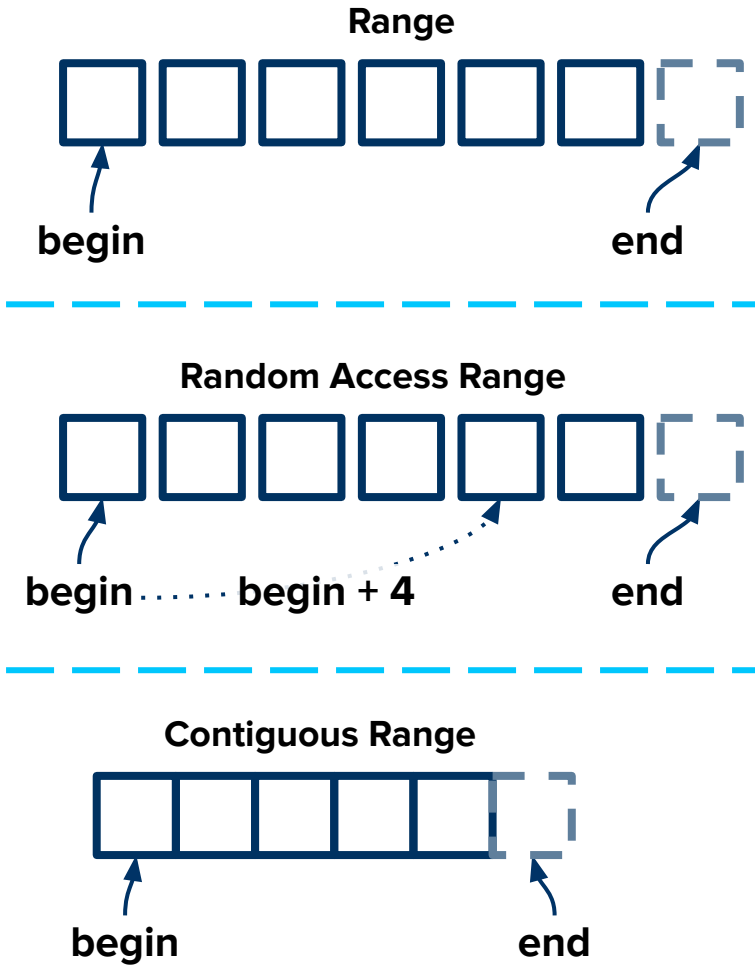

```
// Iteration
for (auto&& value : range) {
  printf("%d\n", value);
}
```

```
// Algorithms
auto r = std::ranges::reduce(range);
auto r = std::ranges::partial_sum(range);
```

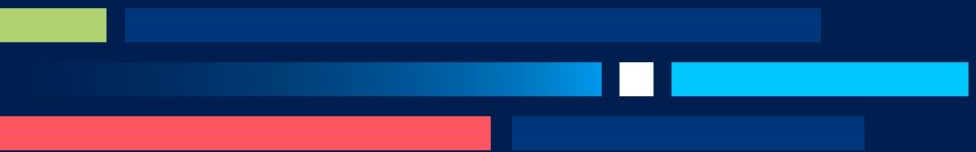
```
// Views
auto add_two = [](auto v) { return v + 2; };
auto view =
     std::ranges::transform_view(range, add_two);
```

Ranges API

- Have a **`begin()`** and **`end()`**
- Have a **`size()`** (usually)
- **Random access**: can access any element at random in **constant time**
- **Contiguous**: represents a **contiguous block of memory**



Distributed Data Structures



Distributed Data Structures

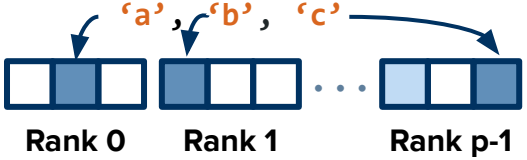
Distributed data structures **split up** data across multiple **segments**

Segments may be stored in **different memory regions**

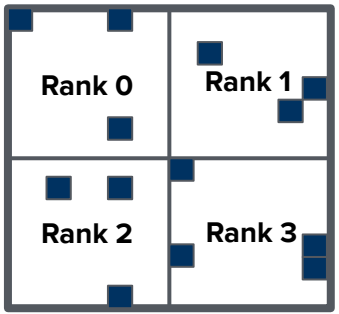
We need a unified API for accessing these distributed data structures!

Distributed Array

Distributed Hash Table



Distributed Matrix



Distributed Data Structures

Data is typically **partitioned** amongst processors into **segments**

Segments are **remotely accessible**, and are located on a **single rank**

Distributed Data Structures

Data is typically **partitioned** amongst processors into **segments**

Segments are **remotely accessible**, and are located on a **single rank**

Distributed Range Concept

R needs two things to be a **distributed range:**

1. **R** is a **standard range**
2. **R** has **segments()**

Distributed Range Concept

R needs two things to be a **distributed range:**

1. **R** is a **standard range**
2. **R** has **segments()**

Distributed Range Concept

R needs two things to be a **distributed range:**

1. **R** is a **standard range**
2. **R** has **segments()**

Distributed Range Concept

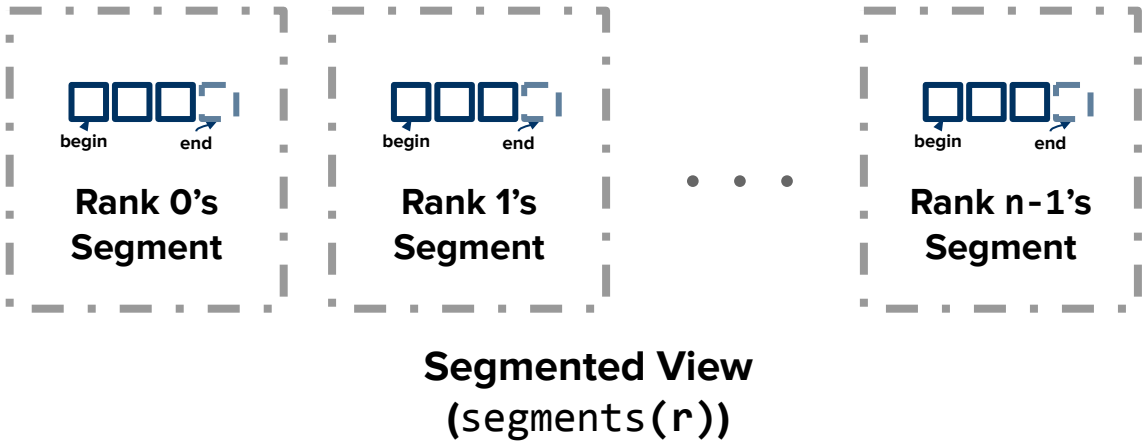
R needs two things to be a **distributed range:**

1. **R** is a **standard range**
2. **R** has **segments()**

Distributed Range Concept

R needs two things to be a **distributed range:**

1. **R** is a **standard range**
2. **R** has **segments()**

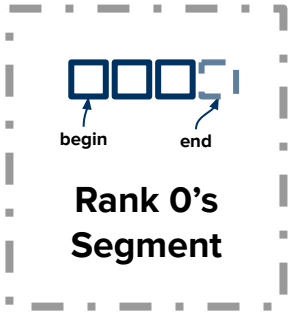


Segments (Remote Range)

Each of the segments in a distributed range is a **remote range**

A remote range is a **standard range**

—Plus it has a **rank**

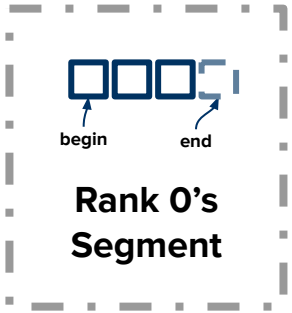


Segments (Remote Range)

Each of the segments in a distributed range is a **remote range**

A remote range is a **standard range**

—Plus it has a **rank**



Algorithms can be implemented hierarchically.

Distributed Algorithms

- Algorithms use the **distributed range concept (segments())**
- Written **hierarchically** using **oneDPL algorithms**

```
using namespace dr::shp;
using namespace oneapi;

float reduce(auto policy,
             distributed_vector<float>& v) {

  float init = 0.0f;
  for (auto&& segment : v.segments()) {
    auto device = devices()[segment.rank()];

    init += dpl::reduce(device, segment);
  }
  return init;
}
```

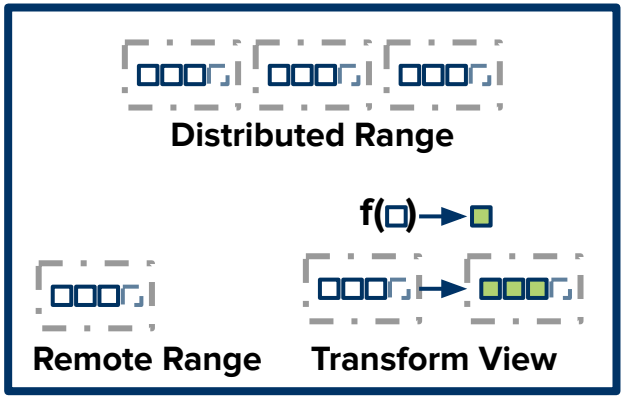
Distributed Views

- Views implement **segments()** by applying transformation to parents' segments
- Views can be built **hierarchically**

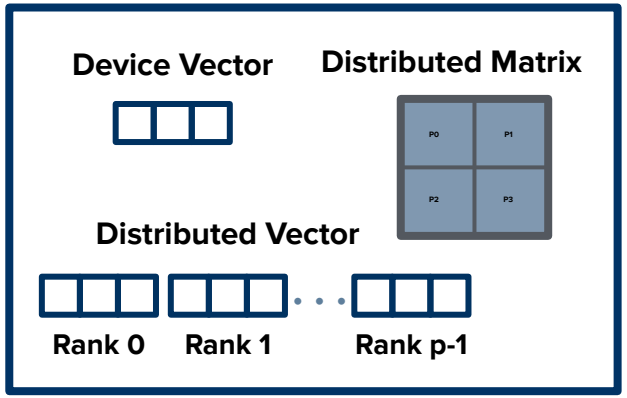
```
template <typename Range,
          typename Fn>
class transform_view {
  . . .

  auto segments() {
    return  base.segments()
          | views::transform(
            [](auto&& segment) {
              return segment
                   | views::transform(fn);
            });
  }

  . . .
};
```

Distributed Ranges Project

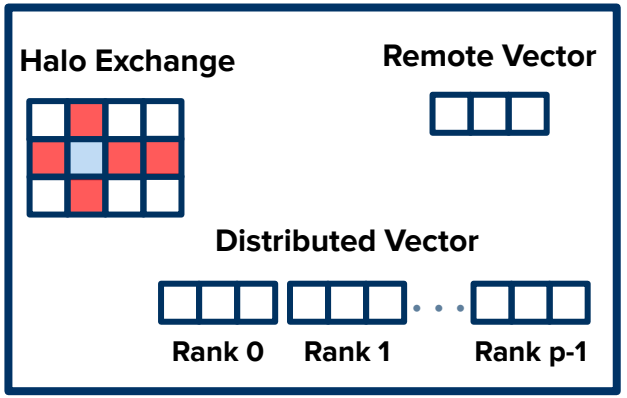
Shared Concepts and Views



GPU Data Structures and Algorithms (“shp”)



MPI Data Structures and Algorithms (“mhp”)



SYCL Codebase (shp)

- Data **automatically distributed** amongst **multiple GPUs**

- **Distributed algorithms**: each GPU calls into **oneDPL algorithms**

```
using namespace dr::shp;


float dot_product(distributed_vector<float>& x,
                  distributed_vector<float>& y) {

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
            });

  return reduce(par_unseq, z, 0, std::plus());
}
```

SYCL Codebase (shp)

- Data **automatically distributed** amongst **multiple GPUs**

- **Distributed algorithms**: each GPU calls into **oneDPL algorithms**

```
using namespace dr::shp;
```

```
float dot_product(distributed_vector<float>& x,
                  distributed_vector<float>& y) {

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
              auto [a, b] = element;
              return a * b;
            });

  return reduce(par_unseq, z, 0, std::plus());
}
```

Multi-Node Codebase (mhp)

- **Multi-process, SPMD** program
- Data structures **automatically distributed** on **multiple nodes** using MPI
- Data structure **constructors** and **algorithms** are collective

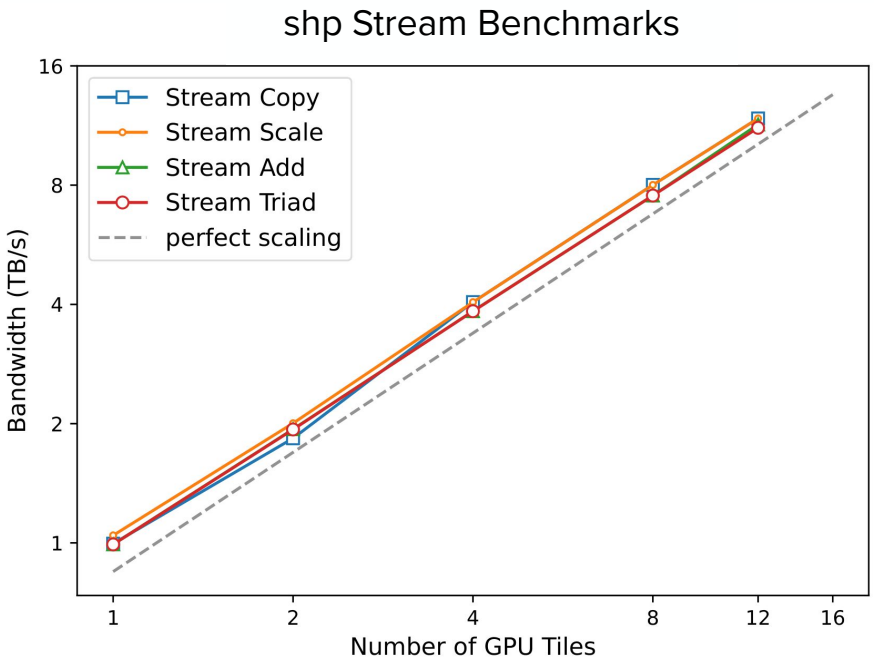
```
using namespace dr::mhp;
```

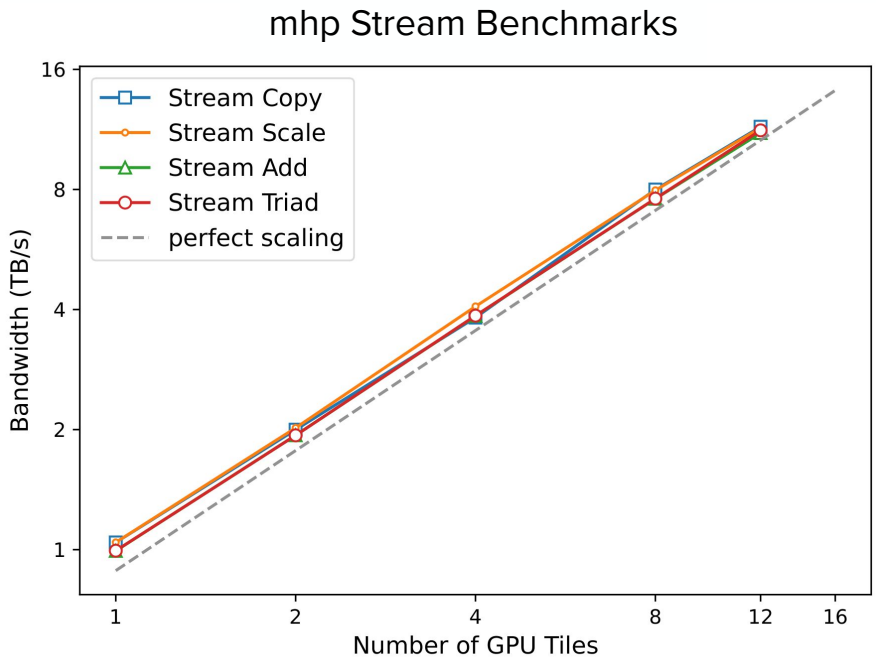
```
float dot_product(distributed_vector<float>& x,
                  distributed_vector<float>& y) {

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
              auto [a, b] = element;
              return a * b;
            });

  return reduce(par_unseq, z, 0, std::plus());
}
```

Data Structure and Algorithms Demo

Performance

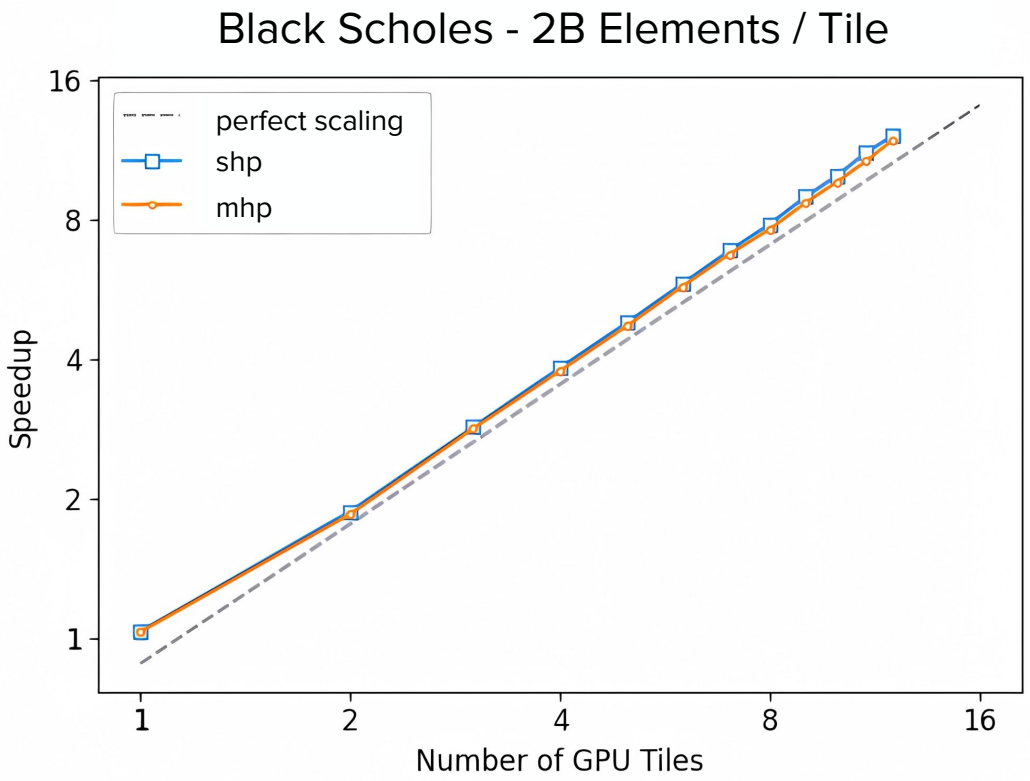
Stream Benchmarks





intel labs

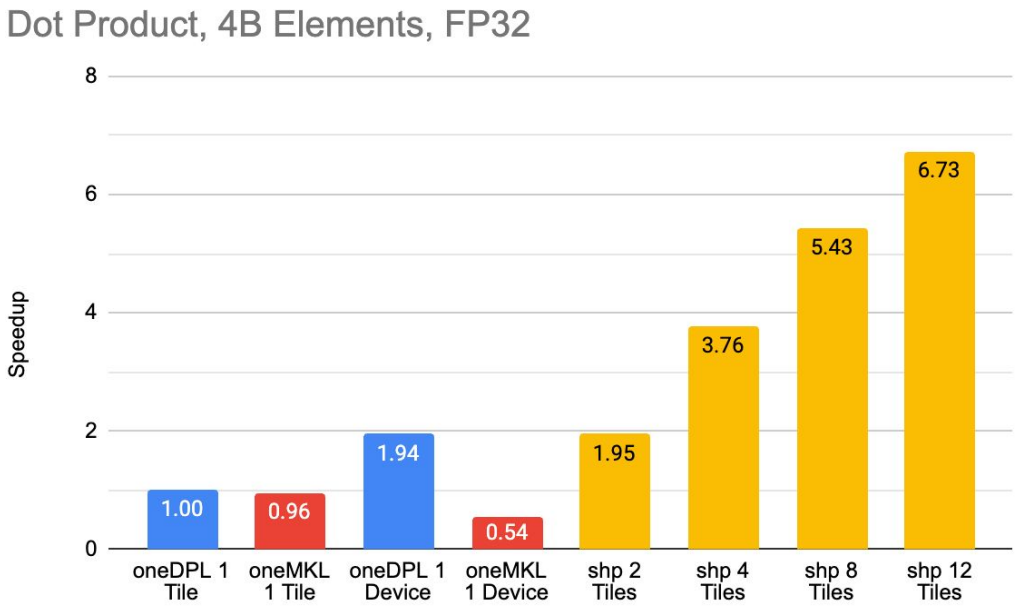
Black Scholes




```
auto black_scholes_kernel = [=](auto &&e) {
  auto &&[s0, x, t, vcall, vput] = e;
  T d1 = (std::log(s0 / x) + (r + T(0.5) * sig * sig) * t) /
         (sig * std::sqrt(t));
  T d2 = (std::log(s0 / x) + (r - T(0.5) * sig * sig) * t) /
         (sig * std::sqrt(t));
  vcall = s0 * normalCDF(d1) - std::exp(-r * t) * x *
          normalCDF(d2);
  vput = std::exp(-r * t) * x * normalCDF(-d2) - s0 *
         normalCDF(-d1);
};


void black_scholes(auto&& s0, auto&& x, auto&& t,
                   auto&& vcall, auto&& vput) {
  for_each(zip(s0, x, t, vcall, vput),
           black_scholes_kernel);
}
```

intel labs

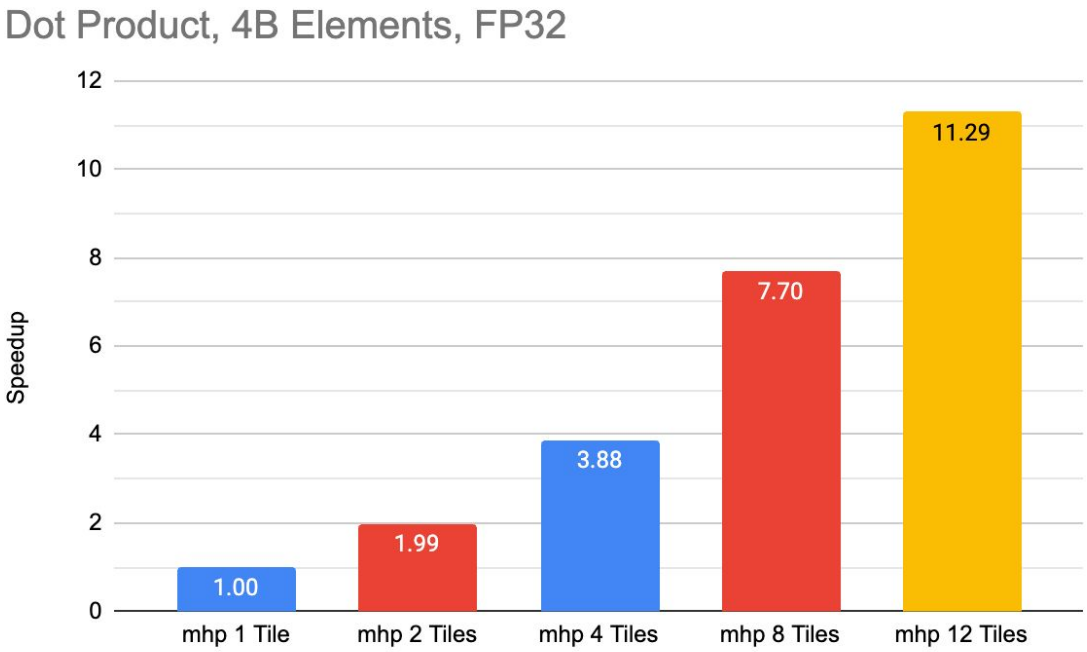
Dot Product - shp




```
float dot_product(vector<float>& x,
                  vector<float>& y) {

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
           });

  return reduce(par_unseq, z, 0, std::plus());
}
```

intel labs

Dot Product - mhp




```
float dot_product(vector<float>& x,
                  vector<float>& y) {

  auto z =  views::zip(x, y)
          | views::transform([](auto element) {
             auto [a, b] = element;
             return a * b;
           });

  return reduce(par_unseq, z, 0, std::plus());
}
```

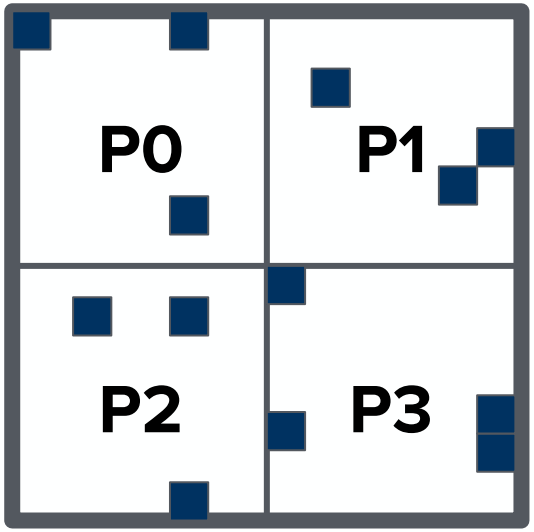
intel labs

Beyond Standard Data Structures

Beyond Standard Data Structures - Matrices

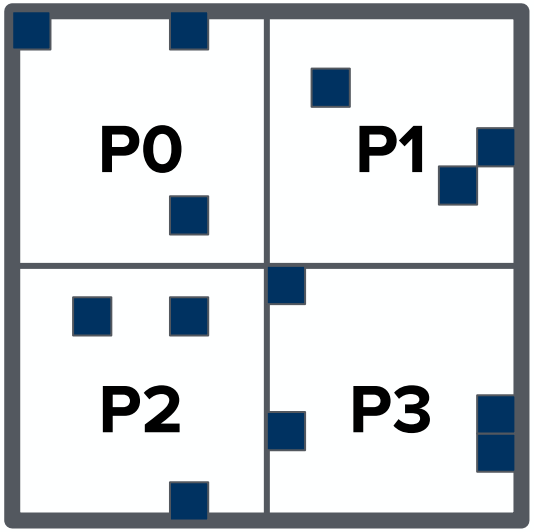
- Can implement **more complex data structures** using distributed range abstraction

- Distributed matrix data structure **splits up matrix**



Beyond Standard Data Structures - Matrices

- Each tile is a **remote range** representing the submatrix
- All of these tiles together constitute the matrix
- Tiles can be **sparse** or **dense**



GraphBLAS C++ Matrix Concept

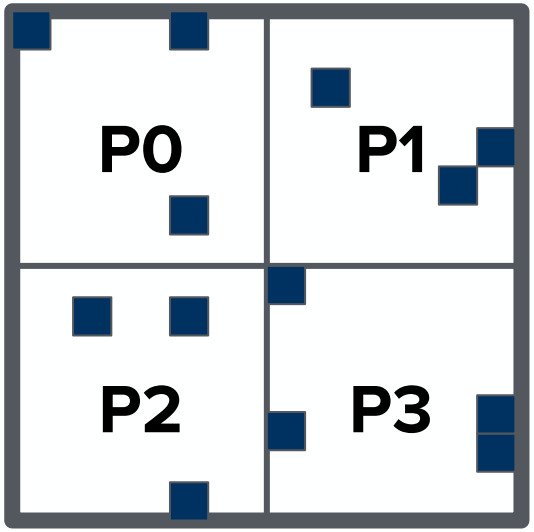
- When iterating through a matrix, observe an **unordered sequence of tuples**

- This works for all varieties of **sparse matrices**

- Can access other, **data structure-specific** iteration methods using **customization points**

Matrices - Can Also Access Tiles Individually

- **`tile()`** - get remote view of tile
- **`get_tile()`** - get copy of tile
- **`get_tile_async()`** - get copy of tile, asynchronously

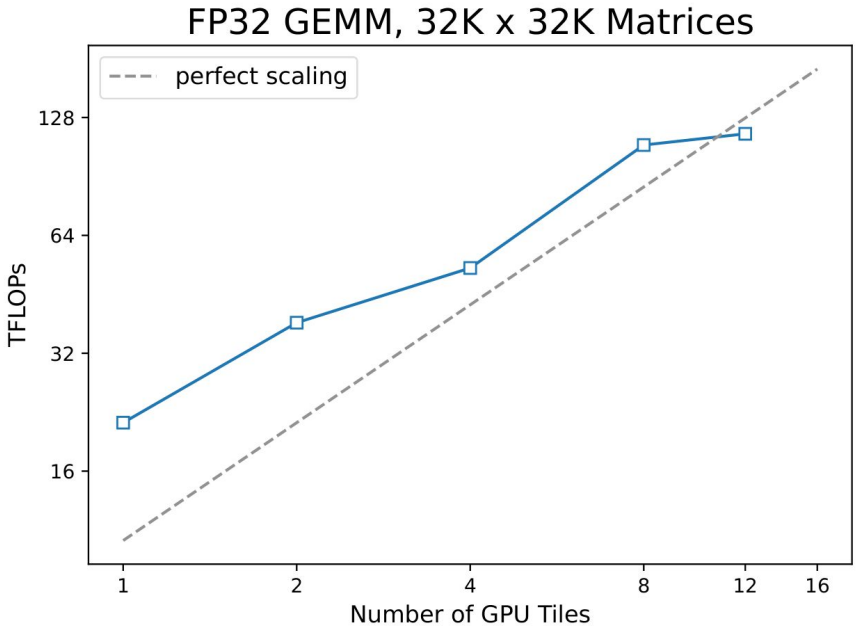


intel labs

Matrix Multiply

- Implement an **DMA-based, multi-GPU** matrix multiply

- GPUs **copy the tiles** they need for the multiply



intel labs

Call to Action

- Standard C++: **Jump in, the water's fine!**
- Our work is **open-source**: https://github.com/oneapi-src/distributed-ranges

Questions?

https://github.com/oneapi-src/distributed-ranges


intel