Session 2: Kokkos

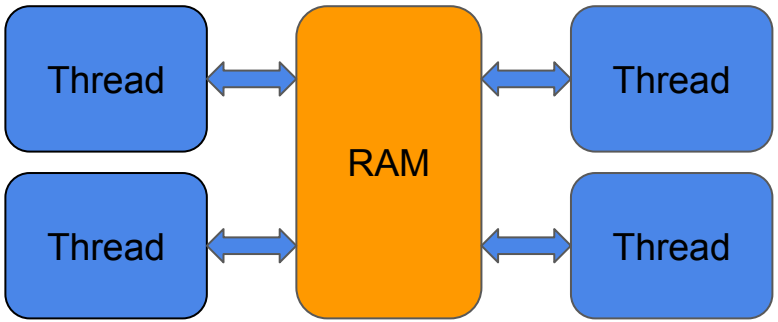
Outline

- Computing architectures, shared- vs. distributed-memory parallelism
- Initialization
- Views
- Parallelizing for loops and reductions
- Random number generation
- ScatterViews
- Virtual functions

Parallelism comes in two flavors

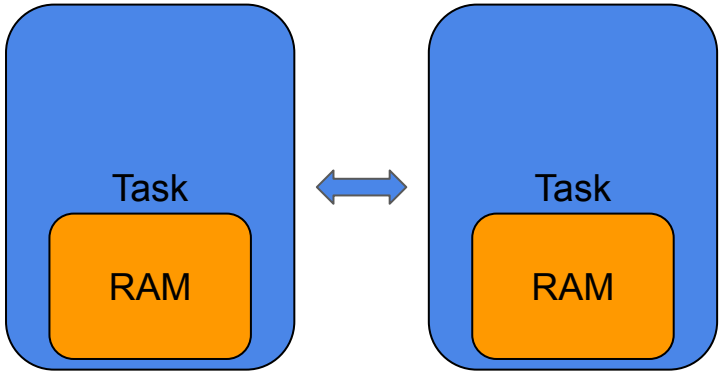
Shared-memory

- Participating **threads** all share a chunk of memory
- Data (variables) accessible to all
- Impossible to scale up to supercomputer size



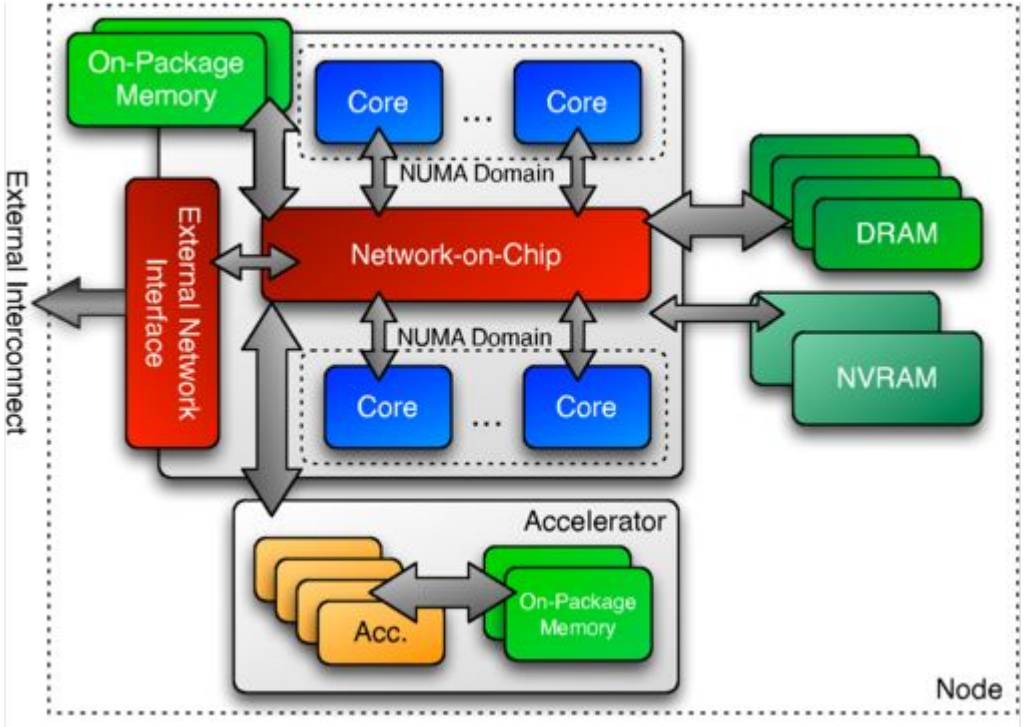
Distributed-memory

- Participating **tasks** or **processes** each have their own chunk of memory
- Data must be manually communicated between tasks



Modern supercomputers require us to use both

- Supercomputers comprise many nodes, which are independent computers in their own right
- Each node has its own memory
- Nodes may contain many processors and even GPUs, which have their own separate memory



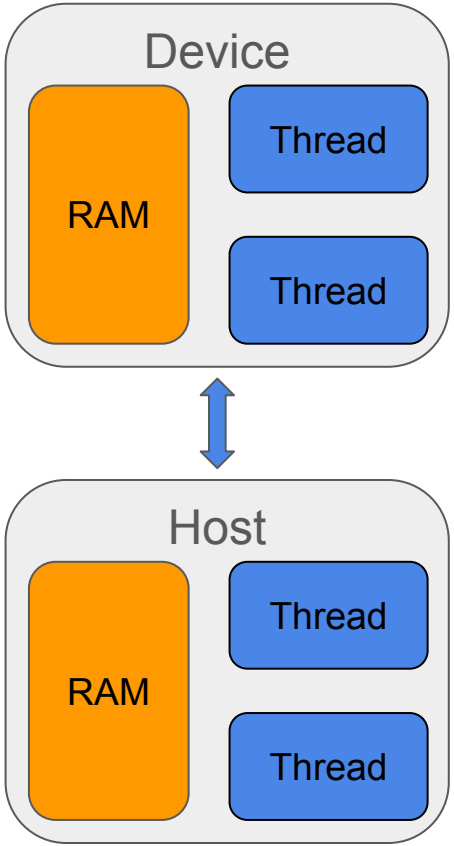
https://kokkos.github.io/kokkos-core-wiki/ProgrammingGuide/Machine-Model.html#figure-2-1-conceptual-model-of-a-future-high-performance-computing-node

Kokkos lets us exploit shared-memory parallelism on any system

- GPU manufacturers each have their own little languages that they prefer
  - NVIDIA uses CUDA
  - AMD uses HIP
  - Intel uses DPC++
- Kokkos basically translates code to each of these, depending on how you install Kokkos
- Rather than writing a bunch of special cases, we can write one code and trust that Kokkos optimizes it for the target architecture (and it does this well)
- There are many alternatives (OpenCL, OpenGL, SYCL, etc.), but we use Kokkos because it *emphasizes scientific computing* on typical *supercomputers in the US*
- Kokkos supports arbitrarily many CPUs on one memory bank and up to one GPU at a time
  - We'll need MPI to get more than one device (next week!)

The host/device model is used by Kokkos

- The set of CPUs available to the program and attached to main RAM are called the “host”
- Attached accelerator (GPU, FPGA) is called the “device”
- Device memory is separate from host memory and must be manually communicated
- Device memory accessible by all device threads, host memory accessible by all host threads



Initialize Kokkos in code before doing anything else

- … and finalize before ending anything!
- Kokkos will request resources from the system, like GPU access
- By feeding in argc and argv, Kokkos will also “trim out” its own command-line options, leaving just ones specific to your program
  - To see the options, run any program built with Kokkos with the –kokkos-help option
- Can also use the ScopeGuard, which is just a class with initialize in its ctor and finalize in its destructor
  - Obviates having to finalize manually, since it’s called when ScopeGuard goes out of scope

```
#include <Kokkos_Core.hpp>

int main(int argc, char *argv[]) {
    Kokkos::initialize(argc, argv);
    {
        // your code here
    }
    Kokkos::finalize();
    return 0;
}

int main(int argc, char *argv[]) {
    Kokkos::ScopeGuard scope_guard(argc, argv);
    {
        // your code here
        // no need to finalize!
    }
    return 0;
}
```

Why do we put our code in curly braces?

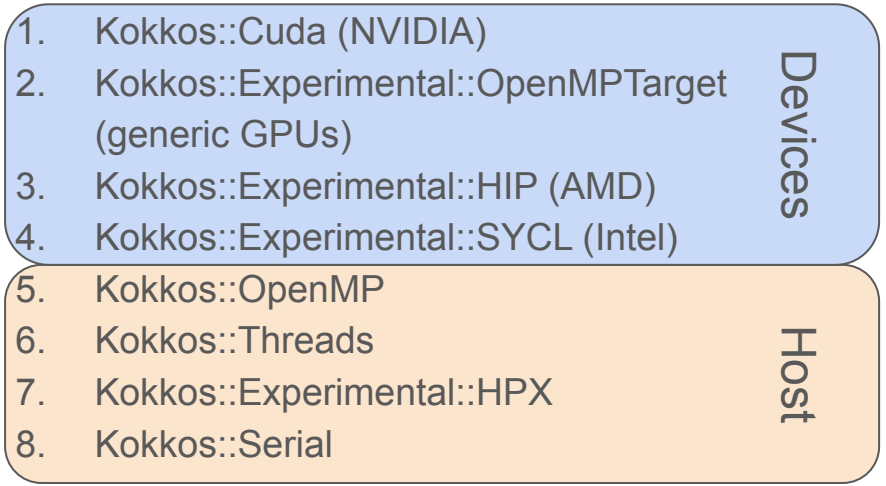
- Kokkos::finalize must be called only after ALL Kokkos data has been deallocated!
- Without curly brace scope guards, data will be destroyed at program termination, AFTER Kokkos::finalize
- The Kokkos::ScopeGuard helps here too
  - If it's constructed first, it'll be destroyed last, ensuring that Kokkos::finalize is called later

```
int main(int argc, char *argv[]) {
    Kokkos::initialize(argc, argv);
    {
        Kokkos::View<double> example;
    }
    // example is now destroyed, all is good
    Kokkos::finalize();
    return 0;
}

int main(int argc, char *argv[]) {
    Kokkos::initialize(argc, argv);
    Kokkos::View<double> example;
    Kokkos::finalize();
    return 0;
    // example isn't destroyed until now. Bad!
}

int main(int argc, char *argv[]) {
    Kokkos::ScopeGuard scope_guard(argc, argv);
    Kokkos::View<double> example;
    return 0;
    // example isn't destroyed until now,
    // but scope_guard is destroyed even later, so all good!
}
```

Kokkos defines “execution spaces” at compile time

- These specify places where work can happen and data can be stored
- Execution spaces are enabled in Kokkos as options when Kokkos is installed
- Kokkos also defines two default execution spaces
  - DefaultExecutionSpace is set to the highest space in the hierarchy to the right
  - DefaultHostExecutionSpace is set to the highest host space to the right
  - Note that, if you do not enable devices in Kokkos, these can be the same!
- Default spaces enable code portability; never use specific spaces!

Hierarchy of execution spaces



A View is a flexible, multidimensional data array

- Layout of data is determined at compile time for optimal performance in parallel
- Label string can be provided to provide more helpful error messages
- Dimensions of array set in datatype template argument
  - Number of asterisks sets number of dimensions known at runtime
  - Numbers in square brackets set dimensions at compile-time
  - Runtime dimensions must precede compile-time dimensions
- Kokkos::deep_copy can be used to copy between Views with the same dimensions and layout, or to copy a single value to all entries of a View

The first arg for the ctor for Views is a label string, and the remaining args are sizes for the runtime-sized dimensions

```
// Creating a 4x5x3 array with two runtime-sized dimensions
// and one compile-time-sized dimension.
Kokkos::View<double**[3]> example("label", 4, 5);

Kokkos::View<double**[3]> to_copy("to_copy", 4, 5);
// This copies the entries of to_copy to example.
Kokkos::deep_copy(example, to_copy);

// Set all the entries of example to 4.0
Kokkos::deep_copy(example, 4.0);
```

Views are not generally accessible from host

- So we can’t just read data from a View
- This is because, by default, the data lives in the DefaultExecutionSpace, which may be on device
- But we can create a host mirror
- Behavior of host mirror differs based on enabled execution spaces
  - If DefaultExecutionSpace is a device space, host mirror is allocated on host and is separate memory from original View
  - If DefaultExecutionSpace is a host space, host mirror is a shallow copy to same data to avoid redundancy
- View data can only be safely accessed on host through host mirrors

Note that the host mirror datatype is just the View you want to mirror followed by ::HostMirror

```
Kokkos::View<double*> example("example", 6);
Kokkos::View<double*>::HostMirror h_example =
    Kokkos::create_mirror_view(example);
// For performance reasons,
// creating a mirror view doesn't copy data,
// so we manually do that now.
Kokkos::deep_copy(h_example, example);
// Now we can access the copy of the data on host!
h_example(1) = 4.0;
```

Typical workflow for filling up a View

1. Construct your View
2. Make a host mirror
3. Fill up your host mirror, e.g. with data from a file
4. Deep copy from your host mirror to the View

```
int main() {
    int num_x_nodes = 100;
    int num_y_nodes = 100;
    Kokkos::View<int**> field("field", num_x_nodes, num_y_nodes);
    Kokkos::View<int**>::HostMirror h_field =
        Kokkos::create_mirror_view(field);
    for (int i=0; i<num_x_nodes; i++) {
        for (int j=0; j<num_y_nodes; j++) {
            h_field(i, j) = i*j;
        }
    }
    Kokkos::deep_copy(field, h_field);
}
```

To parallelize for loops, use Kokkos::parallel_for

- Three arguments for parallel_for
  - The first is a label (just like Views)
  - Second is a range. If it's just a number, the range is 0 to the number, exclusive
  - Third is a **functor**, which is a class or struct with a method that does the body of the desired for loop
- Variables used in the parallel for loop body must be members of the functor
- Note that Views can be accessed inside parallel_fors!
- This is clunky; do we really need to define a class for every single for loop we want to parallelize?

NO. Yay for lambdas!

Here I parallelize the outer for loop of the previous example

```
struct Functor{
// Our functor can have as many member variables as we want,
// but note that these are all copied to device,
// so changes do not persist except for Views.
int num_y_nodes;

Kokkos::View<int**> field;

// Functor operators must be marked with the KOKKOS_INLINE_FUNCTION macro
// and be marked const like this.
KOKKOS_INLINE_FUNCTION void operator() (const int i) const {
    for (int j=0; j<num_y_nodes; j++) {
        field(i, j) = i*j;
    }
}
};

int main() {
    Kokkos::ScopeGuard scope_guard;
    int num_x_nodes = 100;
    int num_y_nodes = 100;
    Kokkos::View<int**> field("field", num_x_nodes, num_y_nodes);
    Kokkos::parallel_for(
        "computeField",
        num_x_nodes,
        Functor{num_y_nodes, field}
    );
}
```

Turns out, lambdas ARE functors!

Same functionality, but MUCH simpler!

- So we can write a lambda in place instead of having to define a functor elsewhere
- Kokkos provides a handy KOKKOS_LAMBDA macro that simplifies defining lambdas
- Variables are automatically captured by lambda *by value*
  - Recall that this means variables are copied – effectively, they're copied up to device
  - Since Views are really just pointers to device data, it's okay that they're shallow copied
- Syntax on right attempts to emulate standard for loop syntax as closely as possible
  - It's not exactly the same, but you'll get used to it

```
int main() {
    Kokkos::ScopeGuard scope_guard;
    int num_x_nodes = 100;
    int num_y_nodes = 100;
    Kokkos::View<int**> field("field", num_x_nodes, num_y_nodes);
    Kokkos::parallel_for(
        "computeField",
        num_x_nodes,
        KOKKOS_LAMBDA (const int i)
    {
        for (int j=0; j<num_y_nodes; j++) {
            field(i, j) = i*j;
        }
    });

    Kokkos::View<int**>::HostMirror h_field =
        Kokkos::create_mirror_view(field);
    // Just to compare, here's the serial version again.
    // Note how similar code above is to this.
    // See, parallelism can be easy!
    for (int i=0; i<num_x_nodes; i++) {
        for (int j=0; j<num_y_nodes; j++) {
            h_field(i, j) = i*j;
        }
    }
    Kokkos::deep_copy(field, h_field);
}
```

We can do multidimensional for loops as well

- We just need to use a special **range policy**: Kokkos::MDRangePolicy
  - MD stands for MultiDimensional
- Template argument is a rank, the number of dimensions in range
- First argument is a list of starts for each of the indices
- Second argument is a list of ends for indices, exclusive
- Also must add extra indices as arguments to the lambda

```
int main() {
    Kokkos::ScopeGuard scope_guard;
    int num_x_nodes = 100;
    int num_y_nodes = 100;
    Kokkos::View<int**> field("field", num_x_nodes, num_y_nodes);
    Kokkos::parallel_for(
        "computeField",
        Kokkos::MDRangePolicy<Kokkos::Rank<2>>(
            {0, 0},
            {num_x_nodes, num_y_nodes}
        ),
        KOKKOS_LAMBDA (const int i, const int j)
    {
        field(i, j) = i*j;
    });
}
```

Functions can be used in parallel regions

- But they must be marked with KOKKOS_INLINE_FUNCTION
- CUDA, HIP, and other GPU languages require functions used on device to be marked with special macros
- Kokkos defines KOKKOS_INLINE_FUNCTION to expand into these macros when those backends are enabled
- Otherwise, it expands to nothing

```
KOKKOS_INLINE_FUNCTION
int product(int i, int j) {
    return i*j;
}

int main() {
    Kokkos::ScopeGuard scope_guard;
    int num_x_nodes = 100;
    int num_y_nodes = 100;
    Kokkos::View<int**> field("field", num_x_nodes, num_y_nodes);
    Kokkos::parallel_for(
        "computeField",
        Kokkos::MDRangePolicy<Kokkos::Rank<2>>(
            {0, 0},
            {num_x_nodes, num_y_nodes}
        ),
        KOKKOS_LAMBDA (const int i, const int j)
    {
        field(i, j) = product(i, j);
    });
}
```

Reductions can be parallelized as well

- A reduction is basically just a sum
- Use Kokkos::parallel_reduce
- Add an “update” reference to lambda signature
  - This holds a thread-local temporary sum value
- Result of reduction is saved to last argument in parallel_reduce

```
int main() {
    Kokkos::View<double*> field(100);
    Kokkos::deep_copy(field, 1.0);
    // Want to find the sum of the entries of field
    double total_sum = 0.0;
    Kokkos::parallel_reduce(
        "findSum",
        field.extent(0), // Size of 0th dimension of field
        KOKKOS_LAMBDA (const int i, double &update)
    {
        update += field(i);
    }, total_sum);
    // total_sum now contains the sum of all entries of field
    return 0;
}
```

With devices, we can exploit **latency hiding**

- This means host code continues to execute *at the same time* as device code
- Only works if a device backend is enabled
- Fences can be added to synchronize device and host, if needed
  - deep_copy has a built-in fence
  - parallel_reduce has a built-in fence at the end
- General workflow is to get the parallel_for running ASAP, and while it's running, set up data structures that'll use the output

```
int main() {
    Kokkos::ScopeGuard scope_guard;

    Kokkos::View<int*> data("data", 1000000);

    Kokkos::parallel_for(
        "fillData",
        data.extent(0),
        KOKKOS_LAMBDA (const int i)
    {
        data(i) = i;
    });

    // If we're using a device, the following code runs while
    // the parallel_for is still running!
    Kokkos::View<int*>::HostMirror h_data =
        Kokkos::create_mirror_view(data);

    // Use a fence to wait for the device to be done.
    Kokkos::fence();

    // deep_copy also has a built in fence, so no need to fence first.
    Kokkos::deep_copy(h_data, data);

    return 0;
}
```

Must be careful with parallel kernels in class methods

- You should make scope-local copies of member variables you intend to use in parallel kernels
- If you naively use a member variable in a parallel kernel, it'll attempt to copy the ENTIRE class up to device, which is usually not what you want
- Copies of Views are typically shallow, so no need to worry about accidentally copying a million data values

```
struct ParallelExample {

Kokkos::View<int*> field;

void computeField() {
    // This is a shallow copy,
    // so I'm basically just making a new pointer
    // to pre-allocated data.
    Kokkos::View<int*> local_field = field;
    Kokkos::parallel_for(
        "computeField",
        100,
        KOKKOS_LAMBDA (const int i)
    {
        // Note that I'm using local_field
        // rather than field.
        // If I used field, the whole class
        // would be copied up,
        // and the compiler would likely complain.
        local_field(i) = i;
    });
}
};
```

Problem: GPU compilers lack linkers

- That means that code that uses a device function MUST have the definition for that function in the same translation unit
- We can design our file structure to emulate the typical structure, but include device function definitions in headers
- Device function declarations are still in the usual header (marked with KOKKOS_FUNCTION), but definitions are in a separate file, _impl.hpp
- This file is included at the *bottom* of the usual header
- Files that include the header now also get the definitions

Contents of "DeviceFunctions.hpp"

```
#ifndef DEVICEFUNCTIONS_HPP
#define DEVICEFUNCTIONS_HPP

int hostAdd(int i, int j);

KOKKOS_FUNCTION
int deviceAdd(int i, int j);

#include "DeviceFunctions_impl.hpp"

#endif
```

Contents of "DeviceFunctions_impl.hpp"

```
#ifndef DEVICEFUNCTIONS_IMPL_HPP
#define DEVICEFUNCTIONS_IMPL_HPP

KOKKOS_INLINE_FUNCTION
int deviceAdd(int i, int j) {
    return i + j;
}

#endif
```

Contents of "DeviceFunctions.cpp"

```
int hostAdd(int i, int j) {
    return i + j;
}
```

Random number generation is tricky in parallel

- If we give each thread a generator with the same seed, they'll just generate the same numbers
- We have to generate a seed for each thread
- Whenever we need a random number on a thread, we have to grab the seed for that thread, generate the number with that seed, and update the seed so that the next generated number is different
- Ideally the thread-local seeds can all be generated from a single seed, so that the user doesn't need to know the number of threads to specify a seed

Kokkos provides an easier way to do all that

- All in Kokkos_Random.hpp
- First create a random pool
  - The ctor for this generates thread seeds
- Within a parallel kernel, grab a thread-local generator from pool
- Use the generator to generate your random numbers
  - There are several distributions available to draw from
- Must manually release the generator so that it could be grabbed by another thread

```cpp
#include <Kokkos_Core.hpp>
#include <Kokkos_Random.hpp>

int main() {
    Kokkos::ScopeGuard scope_guard;

    // construct a pool with a given seed
    int seed = 1;
    Kokkos::Random_XorShift64_Pool<> rand_pool(seed);

    Kokkos::View<double*> uniform_data("uniform_data", 1000);
    Kokkos::View<double*> normal_data("normal_data", 1000);

    Kokkos::parallel_for(
        "fillData",
        uniform_data.extent(0),
        KOKKOS_LAMBDA (const int i)
    {
        // Grab a generator
        auto rgen = rand_pool.get_state();

        // Generate random numbers uniformly between 2 and 5
        uniform_data(i) = rgen.drand(2.0, 5.0);

        // Generate random numbers distributed Gaussian
        // with mean 1 and standard deviation 2
        normal_data(i) = rgen.normal(1.0, 2.0);

        // Release generator
        rand_pool.free_state(rgen);
    });

    return 0;
}
```

ScatterViews help to parallelize histograms

- In the Experimental namespace
- Charge density is basically a histogram
- ScatterView is created based on a standard View “target”
- Behavior depends on DefaultExecutionSpace
  - With devices, ScatterView is just a shallow copy of target with **atomic access** – only one thread is allowed to write at a time
  - Without devices, ScatterView is a deep copy of the target per thread, so each thread has its own array. These need to be combined later on
- Generally, contribute ASAP after histogramming, and reset right before the next histogram

```cpp
#include <Kokkos_ScatterView.hpp>

int main() {
    Kokkos::ScopeGuard scope_guard;
    int num_nodes = 100;

    Kokkos::View<double*> charge("charge", num_nodes);
    Kokkos::Experimental::ScatterView<double*> scatter_charge(charge);

    // Add into the ScatterView
    Kokkos::parallel_for(
        "addIntoScatterView",
        num_nodes,
        KOKKOS_LAMBDA (const int i)
    {
        // Need to get an "access" object to ScatterView
        auto access = scatter_charge.access();

        // Now add into access as if it were a regular array!
        access(i) += 10.0 * i;
    });

    // Need to contribute to the target View to consolidate histogram
    Kokkos::Experimental::contribute(charge, scatter_charge);

    // We can zero out all the arrays with the reset method
    scatter_charge.reset();
}
```

Class methods don’t work easily on device

- When copying class to device, all the member variables get copied as desired
- But when you mark a method KOKKOS_INLINE_FUNCTION, both a host and device version are compiled
- Only the host methods get copied to device if you naively copy a class
- When attempting to call method, function pointer is invalid – CUDA will emit a *very* subtle error!

Example of naive class copy that fails

```
class InvalidMethod {
public:
    double to_add = 3.0;

    KOKKOS_INLINE_FUNCTION
    double increment(double to_increment) const {
        return to_add + to_increment;
    }
};

int main() {
    Kokkos::ScopeGuard scope_guard;

    InvalidMethod invalid_method;
    Kokkos::parallel_for(
        "attemptInvalidMethod",
        100,
        KOKKOS_LAMBDA (const int i)
    {
        // This is okay
        double to_add = invalid_method.to_add;

        // Invalid!
        double incremented = invalid_method.increment(i);
    });

    return 0;
}
```

Placement new lets us sidestep this issue

1. Pre-allocate enough space for the class and store a pointer to it
2. Create a new class instance at that pointer
3. Copy only member variables to new class

- Since the class was created on device, it has the correct device function pointers
- All the member variables are preserved as desired

```
class ValidMethod {
public:
    double to_add = 3.0;

    KOKKOS_INLINE_FUNCTION
    double increment(double to_increment) const {
        return to_add + to_increment;
    }
};

int main() {
    Kokkos::ScopeGuard scope_guard;

    ValidMethod valid_method;

    // Step 1: preallocate space on device and store pointer
    ValidMethod *d_valid_method = static_cast<ValidMethod*>(
        Kokkos::kokkos_malloc(sizeof(ValidMethod))
    );
    // Step 2 and 3: create new class instance at pointer,
    // copy members to new instance.
    // We enter a 1-long parallel_for just to get on device.
    Kokkos::parallel_for(
        "copyClass",
        1,
        KOKKOS_LAMBDA (const char)
    {
        // This line handles creation at pointer and copying,
        // using a copy constructor (which I haven't talked about).
        new (d_valid_method) ValidMethod(valid_method);
    });

    // Now we can use d_valid_method as a pointer to the copy of the class.
    Kokkos::parallel_for(
        "attemptValidMethod",
        100,
        KOKKOS_LAMBDA (const int i)
    {
        // Valid!
        double incremented = d_valid_method->increment(i);
    });
    Kokkos::fence();

    // Must manually free data at pointer.
    // In principle, this can be done automatically with
    // a smart pointer.
    Kokkos::kokkos_free(d_valid_method);

    return 0;
}
```

SYCL does not natively support virtual functions on device

- Specifically it doesn't support **indirect function calls** – basically any call where the function could change
- We can get around this with type enumeration
- Basically we build our own runtime derived-class type determination into the classes themselves
- Base class has a public member variable that determines the type
- Derived classes store the right type in that variable
- Non-member function can read that variable, cast the base class to the appropriate derived class, and call the "virtual" method

An example of type enumeration on host

```
#include <iostream>

enum class type_t {
  CHILD_A,
  CHILD_B,
};

struct Base {
  type_t type;

  Base(type_t type)
    : type(type) {}
};

struct ChildA : public Base {
  ChildA()
    : Base(type_t::CHILD_A) {}

  void pseudo_virtual() {
    std::cout << "A calling." << std::endl;
  }
};

struct ChildB : public Base {
  ChildB()
    : Base(type_t::CHILD_B) {}

  void pseudo_virtual() {
    std::cout << "B calling." << std::endl;
  }
};

void call_pv(Base* base) {
  if (base->type == type_t::CHILD_A) {
    static_cast<ChildA*>(base)->pseudo_virtual();
  } else {
    static_cast<ChildB*>(base)->pseudo_virtual();
  }
}
```

https://developer.codeplay.com/products/computecpp/ce/1.1.0/guides/sycl-guide/limitations

Gotta add some fluff to make it work on device

- Just preceding every function that'll be called on device with KOKKOS_INLINE_FUNCTION
- Combine this with placement new to use pseudo-virtual functions on device!

```cpp
#include <iostream>
#include <Kokkos_Core.hpp>

enum class type_t {
  CHILD_A,
  CHILD_B,
};

You, 44 seconds ago | 2 authors (You and others)
struct Base {
  type_t type;

  Base(type_t type)
    : type(type) {}
};

You, 44 seconds ago | 1 author (You)
struct ChildA : public Base {
  ChildA()
    : Base(type_t::CHILD_A) {}

  KOKKOS_INLINE_FUNCTION
  void pseudo_virtual() {
    std::cout << "A calling." << std::endl;
  }
};

You, 44 seconds ago | 1 author (You)
struct ChildB : public Base {
  ChildB()
    : Base(type_t::CHILD_B) {}

  KOKKOS_INLINE_FUNCTION
  void pseudo_virtual() {
    std::cout << "B calling." << std::endl;
  }
};

KOKKOS_INLINE_FUNCTION
void call_pv(Base* base) {
  if (base->type == type_t::CHILD_A) {
    static_cast<ChildA*>(base)->pseudo_virtual();
  } else {
    static_cast<ChildB*>(base)->pseudo_virtual();
  }
}
```

Exercise: write a 1D particle pusher!

For now, we'll just use a constant force. If you have a working hPIC2 installation, you can just delete the contents of main.cpp and write this here so that you have easy access to Kokkos libraries.

1. Use previous lessons to write a main function. You can just hardcode the parameters (number of particles, particle weight, number of grid elements, grid size, number of timesteps, timestep size, constant acceleration) and make it a main function with no inputs.
2. Use initialize/finalize or a ScopeGuard to set up a Kokkos instance.
3. Make 2 View<double*>, one for particle positions and one for particle velocities. Make an RNG pool from a hardcoded seed, then enter a parallel_for to randomly generate particle positions uniformly throughout domain and particle velocities from a Gaussian distribution.
4. Make a View<double*> for the charge accumulated by particles at each node, and make a ScatterView targeting this.
5. Write an outer for loop over timesteps
6. At each timestep…
    a. Reset the charge ScatterView
    b. Do a parallel_for over all particles
        i. Update velocity, v += a * dt
        ii. Update position, x += v * dt
        iii. If a particle leaves the grid, just wrap it back around to the other side Pacman-style (periodic boundaries)
        iv. Find the two nodes adjacent to the element containing the particle, linearly interpolate particle weight to each node, get access to charge ScatterView, and contribute charge to those nodes in the ScatterView
    c. contribute charge from ScatterView to target View
7. Compile with multiple Kokkos backends to compare speeds between, e.g., OpenMP and serial