codeplay®

THE HETEROGENEOUS SYSTEMS EXPERTS

Executors & Data Movement

Gordon Brown – Senior Software Engineer, SYCL

DHPCC++17 – May 2017

Motivation

- P0443R1: A Unified Executors Proposal for C++
  - Brief overview of current proposal
- P0567R0: Asynchronous Managed Pointer for Heterogeneous Computing
  - Motivation
  - Example using managed pointer interface
  - Future work

Disclaimer

The proposals describe here are a work in progress

This may not reflect the final proposals

P0443R1: A Unified Executors Proposal for C+
+

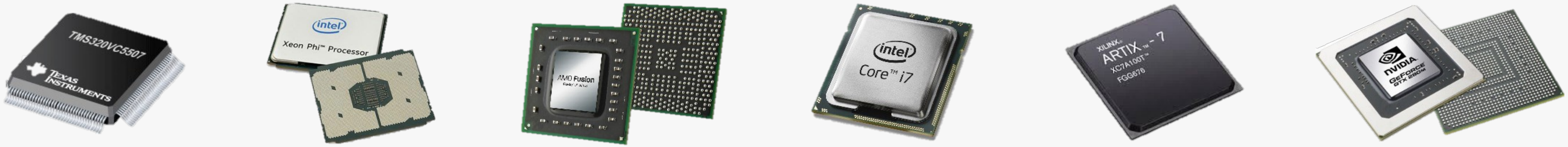
What Are Executors?

invoke   async   parallel algorithms   future::then   post

defer   define_task_block   dispatch   asynchronous operations   strand<>

Unified interface for execution

OpenCL	OpenMP	CUDA	C++ Threads



SAXPY – Computation

```
{
  …

  #pragma omp parallel for
  for (int i = 0; i < SIZE; i++) {
    y[i] = a * x[i] + y[i];
  }

  …
}
```

OpenMP

```
{
  …

  for (int i = 0; i < SIZE; i++) {
    std::thread([=](){
      y[i] = a * x[i] + y[i];
    });
  }

  …
}
```

C++11 Threads

```
{
  …

  cgh.parallel_for(range<1>(SIZE), [=](id<1> idx){
      y[i] = a * x[i] + y[i];
    });
  });

  …
}
```

SYCL

```
__global__ void saxpy(float a, float * restrict x,
                      float * restrict y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  y[i] = a * x[i] + y[i];
}

{
  …

  saxpy<<< SIZE >>>(a, x, y);

  …
}
```

CUDA

SAXPY – Computation

```
{
  …

  executor exec;
  exec.bulk_execute(shape<1>(SIZE), [=](index<1> i) {
    y[i] = a * x[i] + y[i];
  });

  …
}
```

Executors

P0567R0: Asynchronous Managed Pointer for Heterogeneous Computing

Motivation

- Data movement is tightly coupled with computation
  - Describe relationship between computation and data movement
- Current executors proposal does not define data movement
  - Support moving data on platforms that do not have a unified address space
- Interoperability between executor implementations
  - Move data between different executor implementations
- Without a programming model which support this
  - Executors will require extensions for allocation and data movement

SAXPY – Data Movement

```
{
  openmp_executor exec;

  float x[SIZE], y[SIZE], a;

  exec.bulk_execute(shape<1>(SIZE), [=](index<1> i) {
    y[i] = a * x[i] + y[i];
  });
}
```

OpenMP

```
{
  thread_pool_executor exec;

  float x[SIZE], y[SIZE], a;

  exec.bulk_execute(shape<1>(SIZE), [=](index<1> i) {
    y[i] = a * x[i] + y[i];
  });
}
```

C++11 Threads

```
{
  sycl_executor exec;

  auto x = b_x.get_access<access:read>(cgh);
  auto y = b_y.get_access<access:read_write>(cgh);

  exec.bulk_execute(shape<1>(SIZE), [=](index<1> i) {
    y[i] = a * x[i] + y[i];
  });
}
```

SYCL

```
{
  cuda_executor exec;

  cudaMalloc((void **)&x, SIZE * sizeof(float));
  cudaMalloc((void **)&y, SIZE * sizeof(float));
  cudaMemcpy(x, h_x, SIZE, cudaMemcpyHostToDevice);
  cudaMemcpy(y, h_y, SIZE, cudaMemcpyHostToDevice);

  exec.bulk_execute(shape<1>(SIZE), [=](index<1> i) {
    y[i] = a * x[i] + y[i];
  });

  cudaMemcpy(h_y, y, SIZE, cudaMemcpyDeviceToHost);
}
```

CUDA

SAXPY – Data Movement

```
{
  executor exec;

  managed_ptr<float> x, y;
  float a;

  put(exec, x, y);

  exec.bulk_execute(shape<1>(SIZE), [=](index<1> i) {
    y[i] = a * x[i] + y[i];
  });

  get(exec, y);
}
```

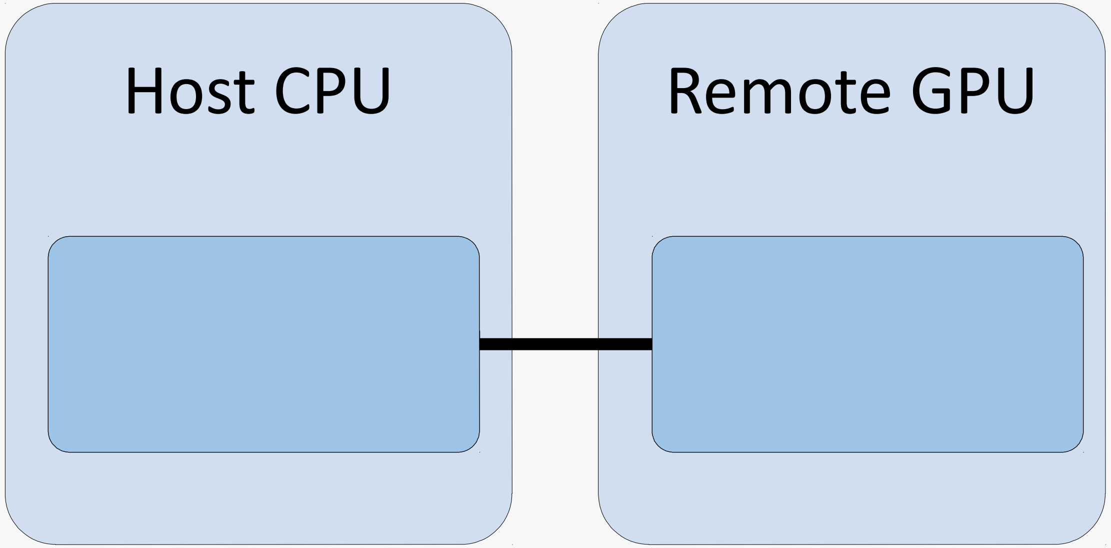
Executors (with Managed Pointer)

Managed Pointer Overview

- Extension to Executors
  - Executors must encapsulate a memory region
- Managed Pointer Class
  - Manages a virtual memory allocation across local and remote memory regions
  - Allocates memory on a memory region when required
  - Can be accessible on a single memory region
- Synchronisation operations
  - Asynchronous implementation defined commands for making a managed pointer accessible on a particular memory region
  - Return a future which can be used to link asynchronous operations

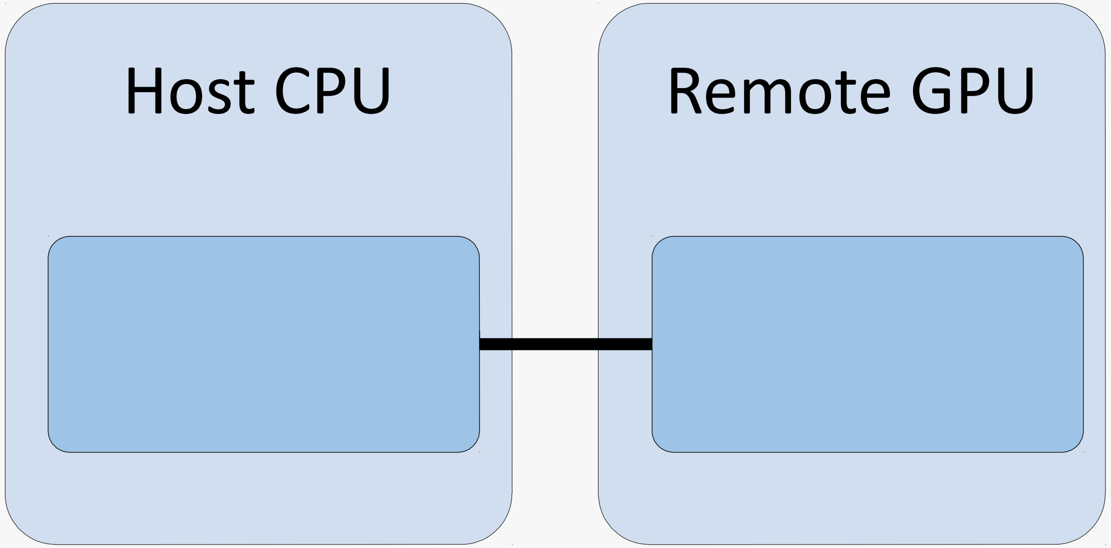
Managed Pointer Example

```
{
  using std::experimental::concurrency_v2;

  struct my_data { /* … */ }

  gpu_execution_context gpuContext;
  auto gpuExec = gpuContext.executor();

  managed_ptr<my_data> ptrA(new my_data());

  managed_ptr<my_data> ptrB(gpuContext.allocator());

  populate(ptrA);

  auto fut =
    put(gpuExec, ptrA)
        .then_async(gpuExec, [=](){ func(ptrA, ptrB); })
            .then_get(gpuExec, ptrB);

  fut.wait();

  print(ptrB);
}
```



Managed Pointer Example

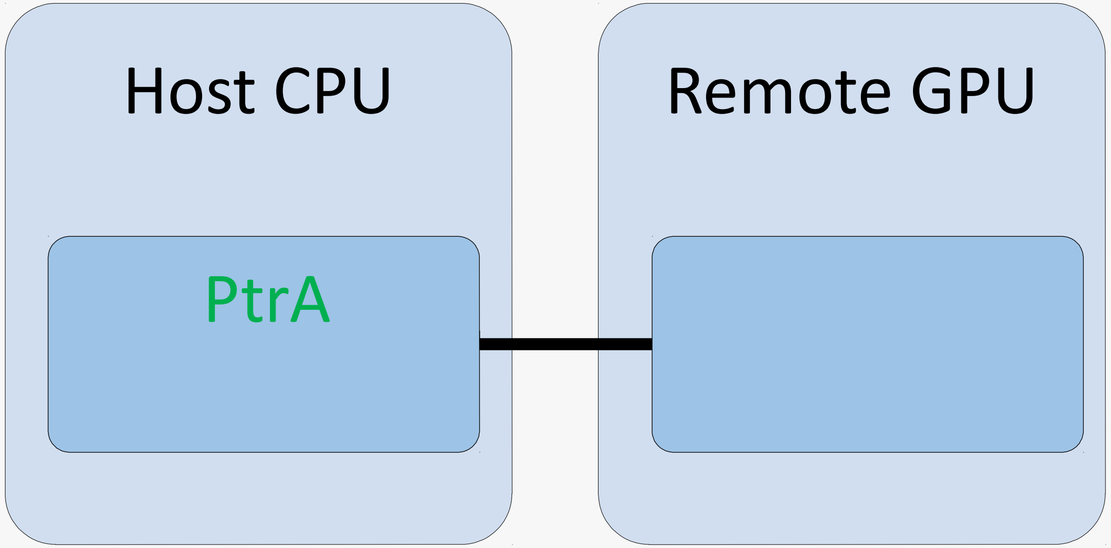
```
{
  using std::experimental::concurrency_v2;

  struct my_data { /* … */ }

  gpu_execution_context gpuContext;
  auto gpuExec = gpuContext.executor();

  managed_ptr<my_data> ptrA(new my_data());

  managed_ptr<my_data> ptrB(gpuContext.allocator());

  populate(ptrA);

  auto fut =
    put(gpuExec, ptrA)
         .then_async(gpuExec, [=](){ func(ptrA, ptrB); })
             .then_get(gpuExec, ptrB);

  fut.wait();

  print(ptrB);
}
```



An executor can be retrieved from an execution context

Managed Pointer Example

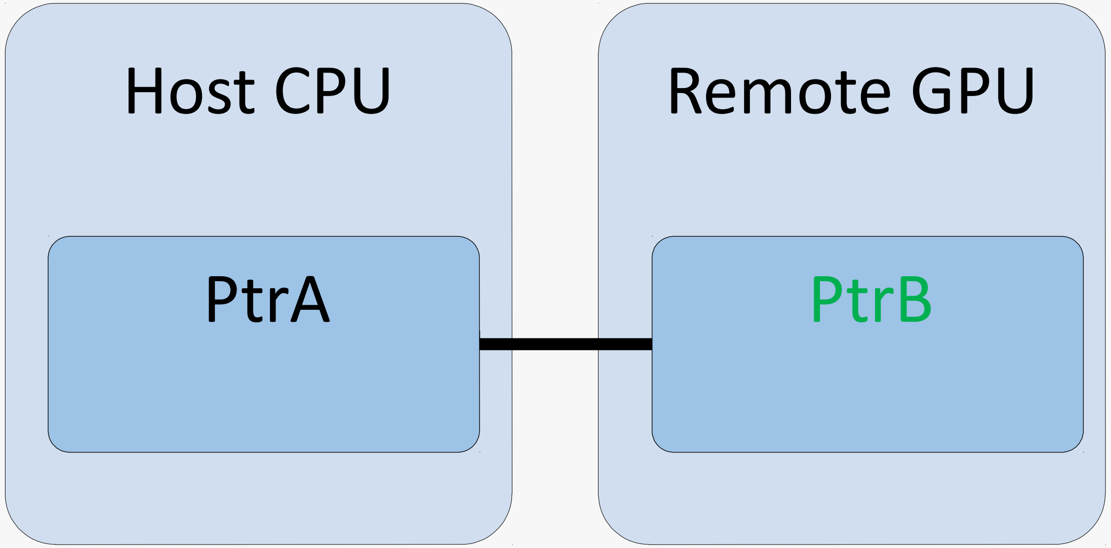
```
{
  using std::experimental::concurrency_v2;

  struct my_data { /* … */ }

  gpu_execution_context gpuContext;
  auto gpuExec = gpuContext.executor();

  managed_ptr<my_data> ptrA(new my_data());

  managed_ptr<my_data> ptrB(gpuContext.allocator());

  populate(ptrA);

  auto fut =
    put(gpuExec, ptrA)
        .then_async(gpuExec, [=](){ func(ptrA, ptrB); })
            .then_get(gpuExec, ptrB);

  fut.wait();

  print(ptrB);
}
```



By default memory is allocated in the host memory region

Managed Pointer Example

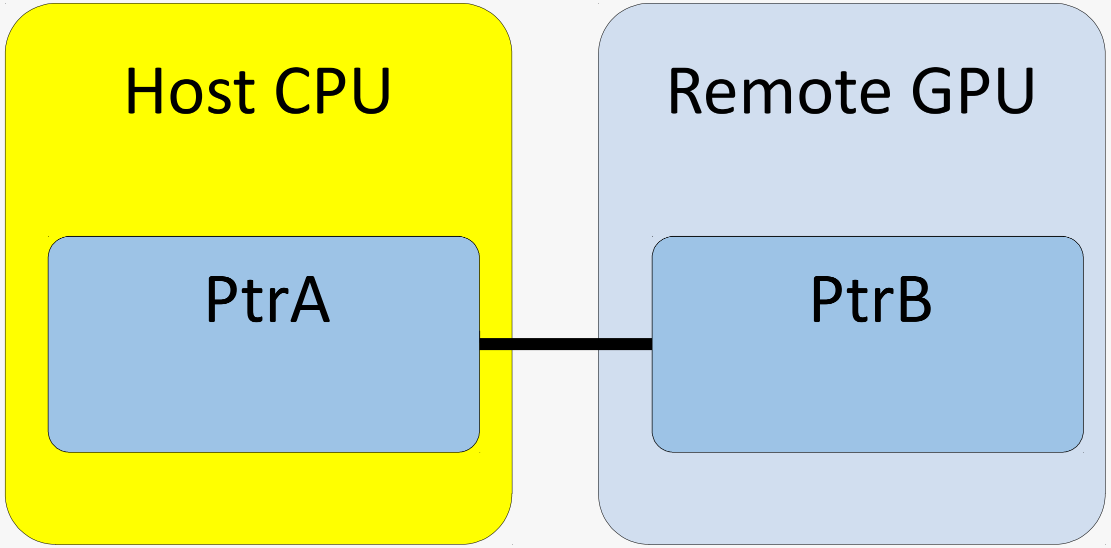
```
{
  using std::experimental::concurrency_v2;

  struct my_data { /* … */ }

  gpu_execution_context gpuContext;
  auto gpuExec = gpuContext.executor();

  managed_ptr<my_data> ptrA(new my_data());

  managed_ptr<my_data> ptrB(gpuContext.allocator());

  populate(ptrA);

  auto fut =
    put(gpuExec, ptrA)
        .then_async(gpuExec, [=](){ func(ptrA, ptrB); })
            .then_get(gpuExec, ptrB);

  fut.wait();

  print(ptrB);
}
```



If an allocator is provided memory can be allocated directly on a remote memory region

Managed Pointer Example

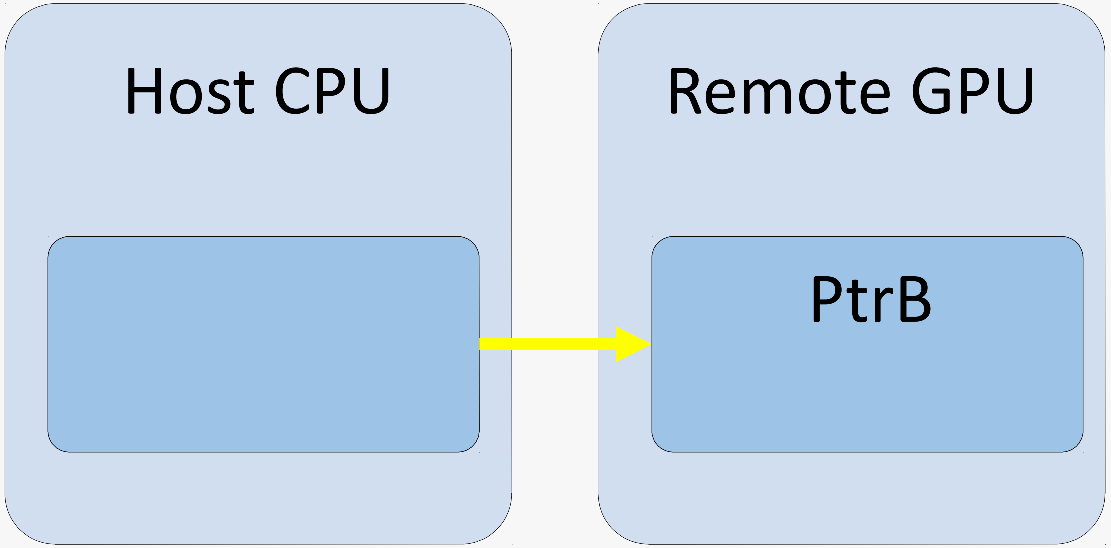
```
{
  using std::experimental::concurrency_v2;

  struct my_data { /* … */ }

  gpu_execution_context gpuContext;
  auto gpuExec = gpuContext.executor();

  managed_ptr<my_data> ptrA(new my_data());

  managed_ptr<my_data> ptrB(gpuContext.allocator());

  populate(ptrA);

  auto fut =
    put(gpuExec, ptrA)
        .then_async(gpuExec, [=](){ func(ptrA, ptrB); })
            .then_get(gpuExec, ptrB);

  fut.wait();

  print(ptrB);
}
```



When accessible the pointer can be accessed

Managed Pointer Example

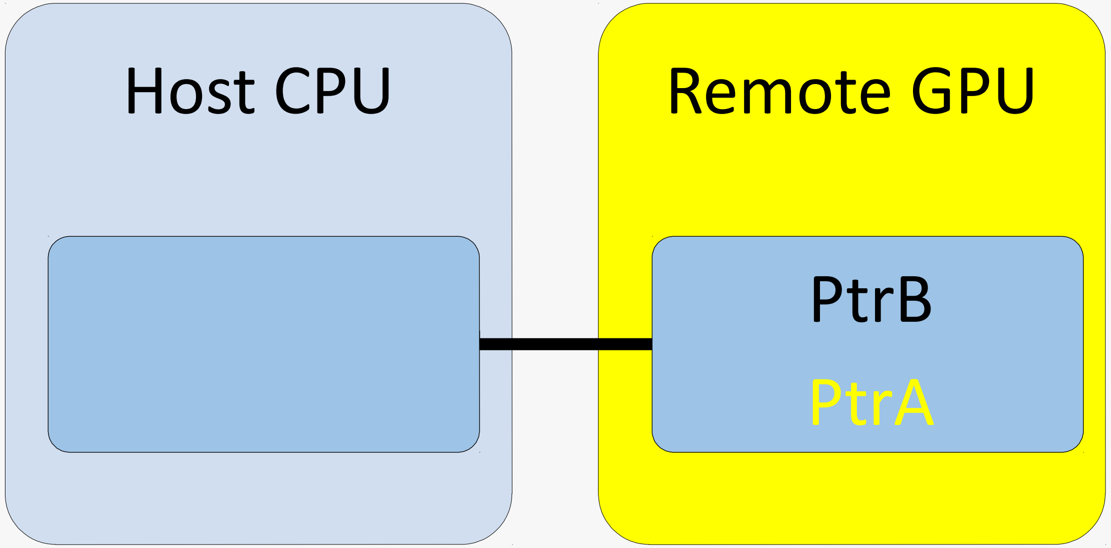
```
{
  using std::experimental::concurrency_v2;

  struct my_data { /* … */ }

  gpu_execution_context gpuContext;
  auto gpuExec = gpuContext.executor();

  managed_ptr<my_data> ptrA(new my_data());

  managed_ptr<my_data> ptrB(gpuContext.allocator());

  populate(ptrA);

  auto fut =
    put(gpuExec, ptrA)
        .then_async(gpuExec, [=](){ func(ptrA, ptrB); })
            .then_get(gpuExec, ptrB);

  fut.wait();

  print(ptrB);
}
```



The put function makes a pointer accessible on another memory region

Managed Pointer Example

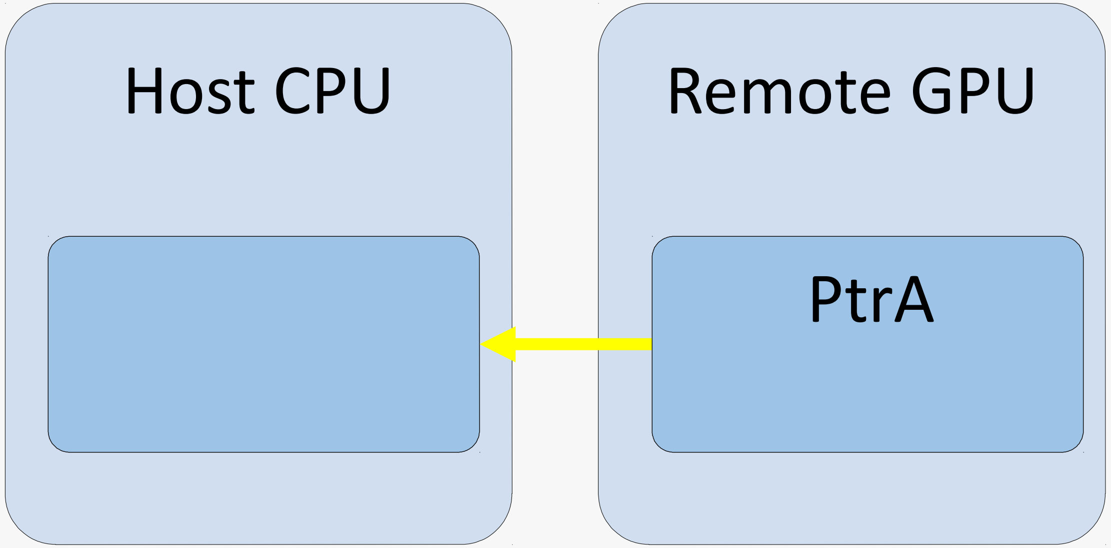
```
{
  using std::experimental::concurrency_v2;

  struct my_data { /* … */ }

  gpu_execution_context gpuContext;
  auto gpuExec = gpuContext.executor();

  managed_ptr<my_data> ptrA(new my_data());

  managed_ptr<my_data> ptrB(gpuContext.allocator());

  populate(ptrA);

  auto fut =
    put(gpuExec, ptrA)
        .then_async(gpuExec, [=](){ func(ptrA, ptrB); })
            .then_get(gpuExec, ptrB);

  fut.wait();

  print(ptrB);
}
```



Execution functions can access any pointers that are accessible on the remote memory region

Managed Pointer Example

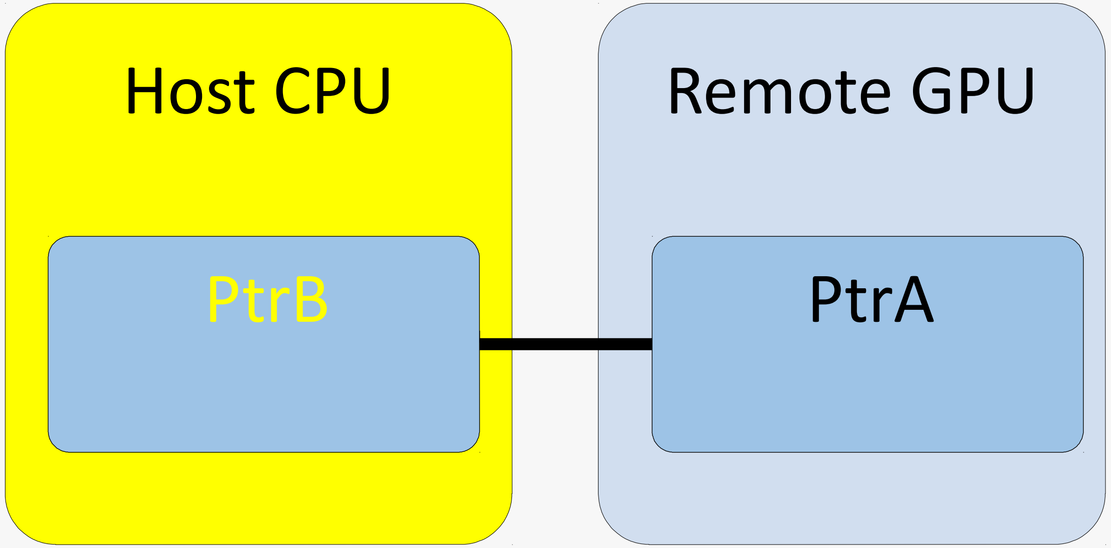
```
{
  using std::experimental::concurrency_v2;

  struct my_data { /* … */ }

  gpu_execution_context gpuContext;
  auto gpuExec = gpuContext.executor();

  managed_ptr<my_data> ptrA(new my_data());

  managed_ptr<my_data> ptrB(gpuContext.allocator());

  populate(ptrA);

  auto fut =
    put(gpuExec, ptrA)
        .then_async(gpuExec, [=](){ func(ptrA, ptrB); })
            .then_get(gpuExec, ptrB);

  fut.wait();

  print(ptrB);
}
```



The put function makes a pointer accessible back on the local memory region

Managed Pointer Example

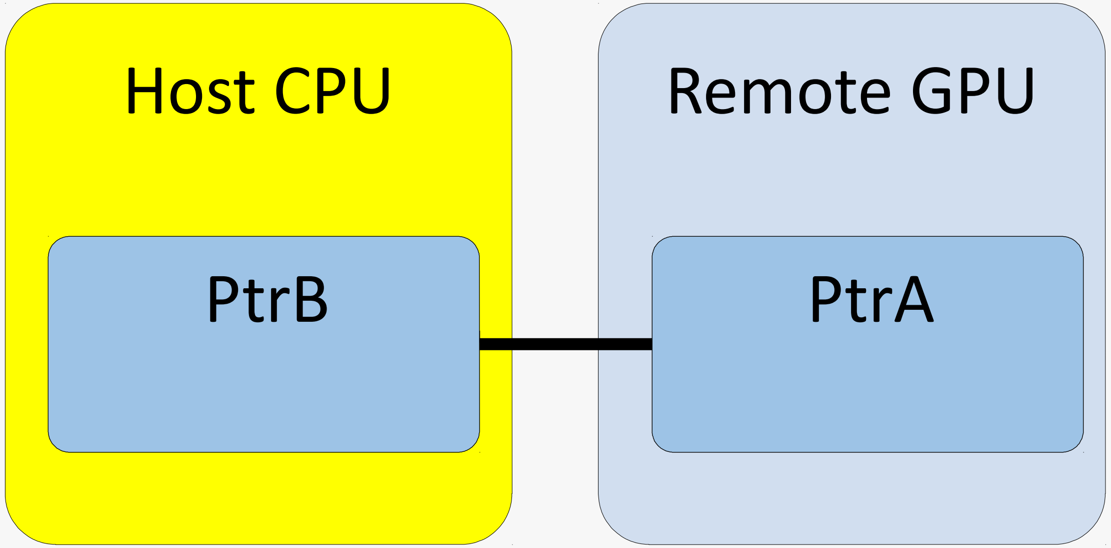
```
{
  using std::experimental::concurrency_v2;

  struct my_data { /* … */ }

  gpu_execution_context gpuContext;
  auto gpuExec = gpuContext.executor();

  managed_ptr<my_data> ptrA(new my_data());

  managed_ptr<my_data> ptrB(gpuContext.allocator());

  populate(ptrA);

  auto fut =
    put(gpuExec, ptrA)
        .then_async(gpuExec, [=](){ func(ptrA, ptrB); })
            .then_get(gpuExec, ptrB);

  fut.wait();

  print(ptrB);
}
```



The future returned from the asynchronous operations can be used to wait for completion

Managed Pointer Example

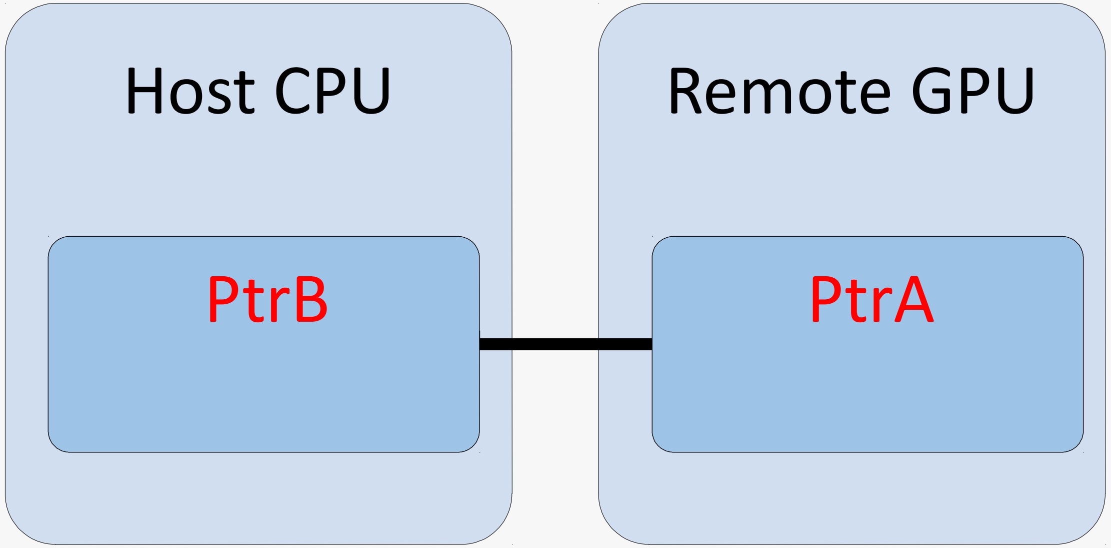
```
{
  using std::experimental::concurrency_v2;

  struct my_data { /* … */ }

  gpu_execution_context gpuContext;
  auto gpuExec = gpuContext.executor();

  managed_ptr<my_data> ptrA(new my_data());

  managed_ptr<my_data> ptrB(gpuContext.allocator());

  populate(ptrA);

  auto fut =
    put(gpuExec, ptrA)
        .then_async(gpuExec, [=](){ func(ptrA, ptrB); })
            .then_get(gpuExec, ptrB);

  fut.wait();

  print(ptrB);
}
```



Once the operations are complete the pointer is accessible in the local memory region it can be accessed

Managed Pointer Example

```
{
  using std::experimental::concurrency_v2;

  struct my_data { /* … */ }

  gpu_execution_context gpuContext;
  auto gpuExec = gpuContext.executor();

  managed_ptr<my_data> ptrA(new my_data());

  managed_ptr<my_data> ptrB(gpuContext.allocator());

  populate(ptrA);

  auto fut =
    put(gpuExec, ptrA)
        .then_async(gpuExec, [=](){ func(ptrA, ptrB); })
            .then_get(gpuExec, ptrB);

  fut.wait();

  print(ptrB);
}
```



All memory allocations of a pointer are deallocated on destruction

Future Work

- Optimal support for both discrete and unified address spaces
  - The current proposal does not yet allow users to take full advantage of unified address spaces
- Allowing optimisation of data movement between contexts
  - The current proposal requires data movement to always go through the host
- Consider other kinds of managed containers
  - The core concepts of the managed pointer could be extended to a managed array or managed stream

We're Hiring!
codeplay.com/careers/

codeplay®
THE HETEROGENEOUS SYSTEMS EXPERTS

Thank you for Listening

@codeplaysoft

info@codeplay.com

codeplay.com