LA-UR-15-27730
Approved for public release; distribution is unlimited.

Title:	Summer 2015 LANL Exit Talk
Author(s):	Usher, Will Canada, Curtis Vincent
Intended for:	Web
Issued:	2015-10-05

Summer 2015 LANL Exit Talk

Will Usher

Summer Work

Worked on writing the OpenMP backend and making general performance improvements and comparisions in VTK-m

Performance Measurement and Improvement

- Added a benchmarking suite to VTK-m to compare backends and changes to backends
- Migrated the default storage type to use an aligned allocator to improve CPU and MIC performance

OpenMP Backend

- Ported Jeff Inman's hand-vectorized MIC scan to a generic version in VTK-m, achieving somewhat comparable performance
- Working on implementing a parallel quick sort for the backend as well, but still some work left to do

Along with some general development work, bug fixes, etc.

VTK-m

VTK-m is a toolkit of visualization algorithms for existing and emerging processor architectures: multi-core CPUs, GPUs, Xeon Phi, …

Provides abstract models for data and execution that can be efficiently implemented in parallel across many architectures and used to implement performant visualization algorithms

- Lower/Upper Bounds: Find first/last index each element can be inserted into a sorted array without changing order
- Reduce, ReduceByKey: Compute an accumulated operation on the data
- Sort, SortByKey
- Scan Inclusive/Exclusive: Inclusive/exclusive prefix sum
- StreamCompact: Remove unwanted elements based on some stencil or predicate
- Unique: Remove adjacent duplicate values in an array
- Schedule: Run instances of a user provided functor on concurrent threads

Benchmarking VTK-m

Main task for the summer was to start writing an OpenMP backend for VTK-m and work on general performance improvements. To measure performance of the backend relative to others and itself between changes we need a way to benchmark the backend

Based on the design of Rust's benchmarking: Take a functor, run it a bunch of times to collect statistics, perform some outlier limiting and report a summary of run times, eg. median, mean, std dev, etc.

User provides a functor that will run the benchmark and return the time what they're interested in took, allowing them to perform per-benchmark setup without including this time in the benchmark time

Benchmarking: Running On Many Types

Some extra (semi-hacky) functionality is included to run benchmark functors on various types by templating the functor on the type being run on, allowing for one-time initialization within the benchmark's constructor

```
template<typename DeviceAdapterTag>
struct SampleBenchmarker {
    typedef vtkm::cont::Timer<DeviceAdapterTag> Timer;
    template<typename Value>
    struct BenchHelloWorld {
        vtkm::Float64 operator()(){
            Timer timer;
            std::cout << "Hello world\n";
            return timer.GetElapsedTime();
        }
        std::string Description() const {
            return "BenchHelloWorld";
        }
    };
    VTKM_MAKE_BENCHMARK(HelloWorld, BenchHelloWorld);
    // Run in some function (eg. main) with:
    // VTKM_RUN_BENCHMARK(HelloWorld,
    //        vtkm::ListTagBase<vtkm::Int32, vtkm::Float32>());
};
```

Benchmarking: Flexibility

If your benchmark doesn't need to run on a bunch of different types you can use the `Benchmarker` directly to benchmark your functor

```
struct HelloWorld {
    const std::string msg;
    // Do some one-time setup for the 'benchmark'
    HelloWorld() : msg("Hello world!\n"){}

    vtkm::Float64 operator()(){
        Timer timer;
        std::cout << msg;
        return timer.GetElapsedTime();
    }
    std::string Description() const {
        return "HelloWorld";
    }
};

int main(int, char**){
    Benchmarker bencher;
    bencher(HelloWorld());
    return 0;
};
```

VTK-m OpenMP Backend

VTK-m provides a set of parallel algorithms that are used to implement various visualization algorithms, allowing a common set of heavily used operations to be specialized and tuned for various architectures instead of re-writing every visualization algorithm for every architecture

A generic backend is also provided which allows for new backends to be created very easily, the generic backend will use the specialized backend's `Schedule` and `Synchronize` methods to implement all other algorithms (although providing more specializations can improve performance quite a bit, eg. of scan and sort)

A backend specialization must also provide a specialization of `ArrayManagerExecution` to transfer data to/from the device

- The OpenMP backend shares memory with the host (control) and execution environments so we specialize `ArrayManagerExecutionShareWithControl`

VTK-m OpenMP Backend: Schedule

Scheduling a user functor in parallel with OpenMP is pretty straightforward, the backend simply executes a parallel for

```
// Within the specialization of DeviceAdapterAlgorithm for OpenMP
template<class Functor>
VTKM_CONT_EXPORT static void Schedule(Functor functor, vtkm::Id numInstances){
    // Some error handling code goes here, removed for brevity
    const ScheduleKernel<Functor> kernel(functor);
#pragma omp parallel for schedule(guided, GRAIN_SIZE)
    for(vtkm::Id i = 0; i < numInstances; ++i){
        kernel(i);
    }
}
```

The specialization of the 3D schedule method is similar

The OpenMP backend follows a fork-join execution model so no asynchronous computation is left running after returning from a method, making `Synchronize` trivial

VTK-m OpenMP Backend: Scan Inclusive

Scan inclusive/exclusive are used to implement many of the algorithms in the general device adapter, providing a faster specialization can help improve performance of a lot of methods making it a good first target for optimization

Started with a hand-vectorized tile based exclusive +scan for Xeon Phi by Jeff Inman to migrate, a few challenges

No Intrinsics

- Our method must be generic on the type being operated on and the operation being performed and should run across all CPU architectures along with KNC and KNL
- Need to work with the compiler to get the code we want generated, Jeff's version makes heavy use of permute intrinsics to shuffle elements in-register, how can we get similar behavior?

No Assumptions on Operators

- The specialized version makes assumptions about the identity value of the operator (eg. for addition identity is 0), we don't know what operator is being applied

VTK-m OpenMP Backend: Scan Inclusive

Need to generate fast code from templated functions and operators, must understand compiler's limitations when generating code

If the compiler naively vectorizes this loop of the upward pass of a tree scan it generates gathers and scatters to handle the non-linear memory accesses of the inner loop, resulting in poor performance

```
vtkm::Id stride;
for (stride = 2; stride - 1 < VectorSize; stride *= 2){
    const vtkm::Id scan_offset = stride / 2 - 1;
    const vtkm::Id scan_distance = stride / 2;
    for (vtkm::Id i = 0; i < VectorSize / stride; ++i){
        const vtkm::Id left = scan_offset + i * stride;
        const vtkm::Id right = left + scan_distance;
        if (right < VectorSize){
            vector[right] = BinaryOperator(vector[left], vector[right]);
        }
    }
}
```

VTK-m OpenMP Backend: Scan Inclusive

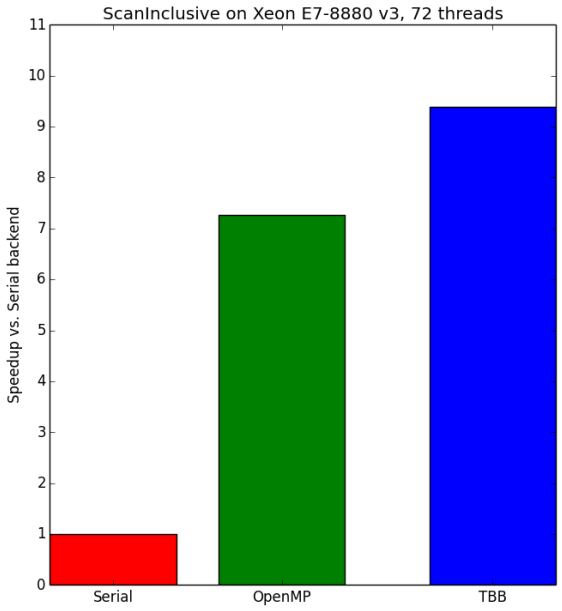
Instead we'll take advantage of loop unrolling and fully unroll a 16-element tree scan, similar to the intrinsics version with permutes but instead perform generic operations

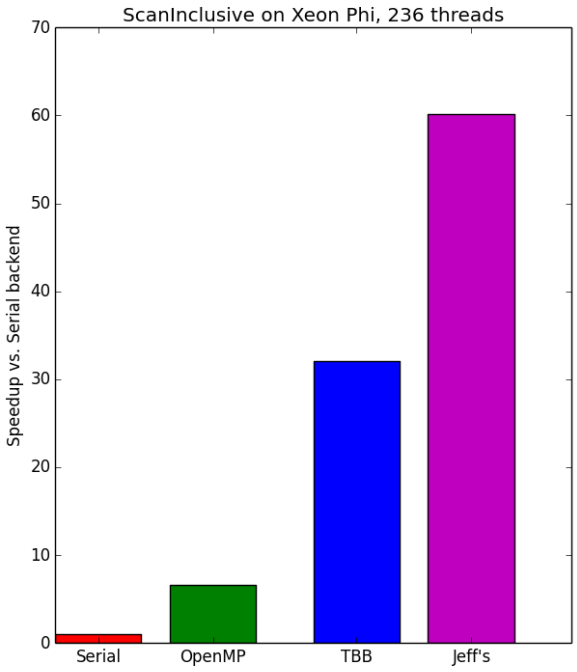
```
// Unrolled version of first iteration of the outer loop
vector[1]  = BinaryOperator(vector[0],  vector[1]);
vector[3]  = BinaryOperator(vector[2],  vector[3]);
vector[5]  = BinaryOperator(vector[4],  vector[5]);
vector[7]  = BinaryOperator(vector[6],  vector[7]);
vector[9]  = BinaryOperator(vector[8],  vector[9]);
vector[11] = BinaryOperator(vector[10], vector[11]);
vector[13] = BinaryOperator(vector[12], vector[13]);
vector[15] = BinaryOperator(vector[14], vector[15]);
```

Get further gains by special casing loads where we have 16 valid elements and unrolling this with a `#pragma unroll`, letting us handle arrays that aren't multiples of 16 long while keeping the unrolled 16 element scan

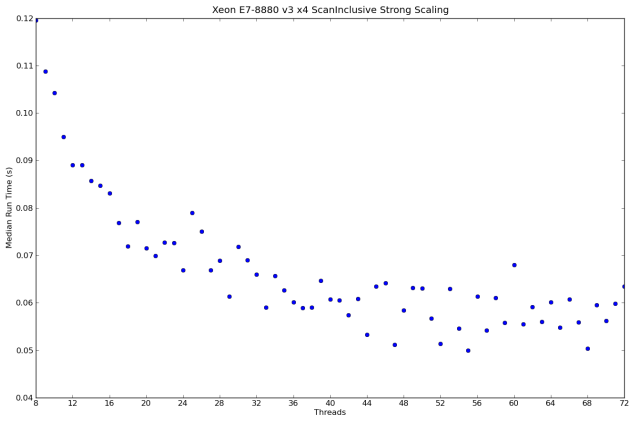
VTK-m OpenMP Backend: Scan Inclusive

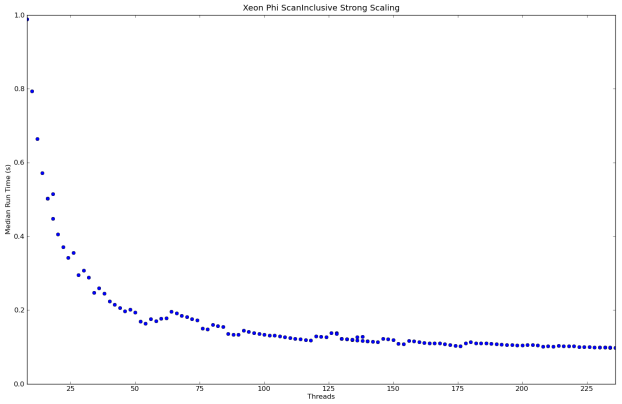
Performance is ok on CPU however on Xeon Phi Intel's TBB scan is *significantly* faster. My current thoughts are that poor work distribution and threading overhead are holding our performance back





VTK-m OpenMP Backend: Scan Inclusive

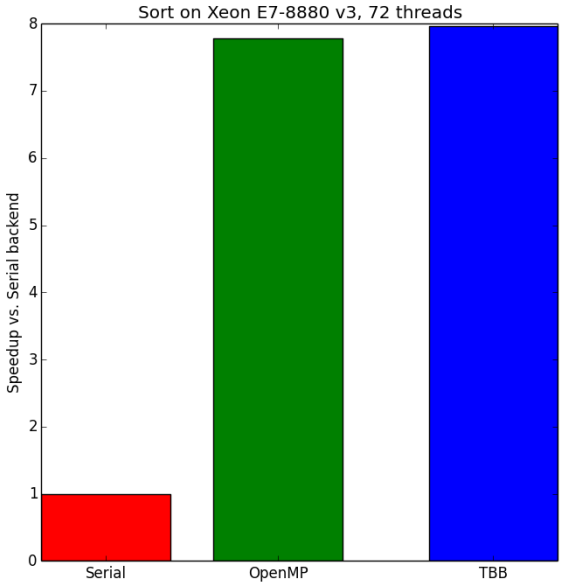


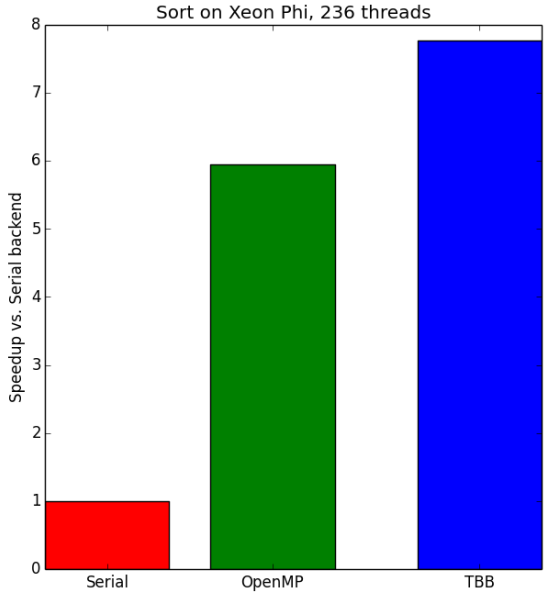


VTK-m OpenMP Backend: Sort (preliminary)

Recently (past few days) implemented a parallel quicksort for the OpenMP backend's specialization of Sort (still some bugs to sort out)

Intel TBB's parallel sort is also a parallel quicksort so I'd expect at least comparable performance





Miscellaneous Performance: Alignment

Aligning data to cache line boundaries can provide a decent bump, as it helps:

- Reduce cache misses and flushes
- Reduce number of bus transactions to read data
- Reduce false sharing
- Reduce unaligned loads into vector registers, aiding vectorization

Switching to aligned allocation isn't too difficult, each platform provides an aligned allocation function

- Any POSIX: `posix_memalign`
- WIN32: `_aligned_alloc`
- CPUs with SSE: `_mm_malloc` (useful as a fallback if you can't identify the platform)

Added an aligned `std::allocator` which is used by VTK-m's default storage type and can be used with STL containers so users can pass aligned memory to VTK-m

Small boost on OpenMP scan inclusive on MIC, gain ~0.01s (~12.5%) for scan on 2^{28} int's

Questions?

Creating a VTK-m Backend

What's Needed for a New Backend

Create a directory under `vtkm/cont/` for your backend. The implementation will go in `internal/` and a header to include your backend will be placed under `vtkm/cont/backend`

You'll need to provide specializations of:

- ArrayManagerExecution (or ArrayManagerExecutionShareWithControl if control and execution environments share memory)
- DeviceAdapterAlgorithmGeneral, you only need to provide specializations for both `Schedule` methods and `Synchronize`

Additionally you'll make use of the `VTKM_CREATE_DEVICE_ADAPTER` macro to create the adapter tag

After this we'll need to add our backend into CMake, the test and benchmark suites and into some header files

Structure of a Backend

```
vtkm/cont/
    tbb/
    cuda/
    demo/
        DeviceAdapterDemo.h
        internal/
            DeviceAdapterAlgorithmDemo.h
            ArrayManagerExecutionDemo.h
            DeviceAdapterTagDemo.h
            ... (anything else you need)
        testing/
            More on testing shortly
```

- DeviceAdapterAlgorithmDemo.h: Specialization of DeviceAdapterAlgorithmGeneral
- DeviceAdapterTagDemo.h: Use `VTKM_CREATE_DEVICE_ADAPTER(Demo)` macro to generate a tag for your adapter
- ArrayManagerExecutionDemo.h: Specialize an array manager for passing data to/from the execution environment
- DeviceAdapterDemo.h: Include the headers from `internal/`

Creating a Device Adapter Tag

The device adapter tag is used to pick the right template specializations to run on your backend

Creating one is done using the `VTKM_CREATE_DEVICE_ADAPTER` macro

File: `DeviceAdapterTagDemo.h`

```
#ifndef vtk_m_cont_demo_internal_DeviceAdapterTagDemo_h
#define vtk_m_cont_demo_internal_DeviceAdapterTagDemo_h
#include <vtkm/cont/internal/DeviceAdapterTag.h>

VTKM_CREATE_DEVICE_ADAPTER(Demo);

#endif
```

Providing a Device Adapter Algorithm

Specialize DeviceAdapterAlgorithmGeneral for your specific backend in DeviceAdapterAlgorithmDemo

It must provide at least both `Schedule` methods and `Synchronize`, however providing tuned specializations for other methods can improve performance (eg `Sort`, `ScanInclusive`)

File: `DeviceAdapterAlgorithmDemo.h`

```
// Include guards + lots of includes here, see existing backends for examples
// or the source for this demo
namespace vtkm {
namespace cont {
template<>
struct DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagDemo> :
    vtkm::cont::internal::DeviceAdapterAlgorithmGeneral<
        DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagDemo>,
        vtkm::cont::DeviceAdapterTagDemo>
{
    ...
};
```

Specializing Schedule for 1D Workloads

The device adapter must provide specializations for scheduling 1D and 3D workloads

Scheduling a 1D workload:

```
template<class Functor>
VTKM_CONT_EXPORT static void Schedule(Functor functor, vtkm::Id numInstances){
    // For error handling
    const vtkm::Id MESSAGE_SIZE = 1024;
    char errorString[MESSAGE_SIZE];
    errorString[0] = '\0';
    vtkm::exec::internal::ErrorMessageBuffer
        errorMessage(errorString, MESSAGE_SIZE);
    // Schedule the functor
    for (vtkm::Id i = 0; i < numInstances; ++i){
        functor(i);
    }
    // Check if something went wrong in the functor
    if (errorMessage.IsErrorRaised()){
        throw vtkm::cont::ErrorExecution(errorString);
    }
}
```

Specializing Schedule for 3D Workloads

```
template<class Functor>
VTKM_CONT_EXPORT static void Schedule(Functor functor, vtkm::Id3 rangeMax){
    // For error handling
    const vtkm::Id MESSAGE_SIZE = 1024;
    char errorString[MESSAGE_SIZE];
    errorString[0] = '\0';
    vtkm::exec::internal::ErrorMessageBuffer
        errorMessage(errorString, MESSAGE_SIZE);
    // Schedule the functor
    for (vtkm::Id i = 0; i < rangeMax[0] * rangeMax[1] * rangeMax[2]; ++i){
        functor(i);
    }
    // Check if something went wrong in the functor
    if (errorMessage.IsErrorRaised()){
        throw vtkm::cont::ErrorExecution(errorString);
    }
}
```

Specializing Synchronize

Synchronize should synchronize the control and execution environments of your backend, eg. wait for any running asynchronous computations to finish

Our demo backend is serial so there's not much to be done here

```
VTKM_CONT_EXPORT static void Synchronize(){}
```

Providing an Array Manager

Your ArrayManagerExecution* will manage moving data between the execution and control environments

If your execution and control share memory (eg. both on CPU) check the Serial and TBB backends for examples

If they don't share memory (eg. run on GPU or MIC offload) check the CUDA backend for an example

This will be placed in: `ArrayManagerExecutionDemo.h`

Providing an Array Manager

Our demo backend is just on the CPU so we can base ours on the Serial backend's array manager

```
// Include guards + includes, etc..
namespace vtkm {
namespace cont {
namespace internal {
template<typename T, class StorageTag>
class ArrayManagerExecution<T, StorageTag, vtkm::cont::DeviceAdapterTagDemo>
    : public vtkm::cont::internal::ArrayManagerExecutionShareWithControl
        <T, StorageTag>
{
public:
    typedef vtkm::cont::internal::ArrayManagerExecutionShareWithControl
        <T, StorageTag> Superclass;
    typedef typename Superclass::ValueType ValueType;
    typedef typename Superclass::PortalType PortalType;
    typedef typename Superclass::PortalConstType PortalConstType;

    VTKM_CONT_EXPORT
    ArrayManagerExecution(typename Superclass:StorageType *storage)
        : Superclass(storage){}
};
}
}
```

Providing a Device Adapter Header

The header `DeviceAdapterDemo.h` just includes your internal headers and is placed in `vtkm/cont/demo/`

```
#ifndef vtk_m_cont_demo_DeviceAdapterDemo_h
#define vtk_m_cont_demo_DeviceAdapterDemo_h

#include <vtkm/cont/demo/internal/DeviceAdapterAlgorithmDemo.h>
#include <vtkm/cont/demo/internal/ArrayManagerExecutionDemo.h>
#include <vtkm/cont/demo/internal/DeviceAdapterTagDemo.h>

#endif
```

Telling VTK-m About Your Backend: CMake

We need to add our backend to CMake and add some conditional includes to get our backend code included in the build

We'll start by adding a CMake option to build our backend. In the main `CMakeLists.txt` we'll add:

```
# After the CUDA and TBB backend options
option(VTKm_ENABLE_DEMO "Enable the demo backend" OFF)
```

Add us to `vtkm/cont/CMakeLists.txt` as well

```
# After the if (VTKm_ENABLE_TBB) and CUDA conditions
if (VTKm_ENABLE_DEMO)
    add_subdirectory(demo)
endif()
```

Telling VTK-m About Your Backend: CMake

Optional: If your backend needs to add compilation flags or find some dependencies you can add a script in the `CMake` folder named `UseVTKm<backend>.cmake`, so here we'll have one that does nothing:

`CMake/UseVTKmDemo.cmake`:

```
if (VTKm_Demo_initialize_complete)
    return()
endif (VTKm_Demo_initialize_complete)

# Find any required libraries
if (NOT VTKm_Demo_FOUND)
    set(VTKm_Demo_FOUND TRUE)
endif (NOT VTKm_Demo_FOUND)

# Setup any dependent packages and compilation flags
if (VTKm_Demo_FOUND)
    set(VTKm_Demo_initialize_complete TRUE)
endif (VTKm_Demo_FOUND)
```

Telling VTK-m About Your Backend: CMake

Optional: To have CMake run our setup script from the previous slide we'll add a call to `vtkm_configure_device` in the main `CMakeLists.txt` file beneath the 'Set up devices selected' comment.

```
# After the if (VTKm_ENABLE_TBB) and CUDA conditions
if (VTKm_ENABLE_DEMO)
    vtkm_configure_device(Demo)
endif()
```

Telling VTK-m About Your Backend: CMake

Now we define CMake files to build our backend

In `vtkm/cont/demo/CMakeLists.txt`:

```
set(headers DeviceAdapterDemo.h)
add_subdirectory(internal)
vtkm_declare_headers(${headers})
# More on testing in a bit
add_subdirectory(testing)
```

In `vtkm/cont/demo/internal/CMakeLists.txt`:

```
set(headers
    DeviceAdapterAlgorithmDemo.h
    DeviceAdapterTagDemo.h
    ArrayManagerExecutionDemo.h
)
vtkm_declare_headers(${headers})
```

Telling VTK-m About Your Backend

vtkm/cont/internal/DeviceAdapterAlgorithm.h:

```
// At the bottom of the file with the other backend conditional includes add:
#elif VTKM_DEVICE_ADAPTER == VTKM_DEVICE_ADAPTER_DEMO
#include <vtkm/cont/demo/internal/DeviceAdapterAlgorithmDemo.h>
#endif
```

vtkm/cont/internal/ArrayManagerExecution.h:

```
// At the bottom of the file with the other backend conditional includes add:
#elif VTKM_DEVICE_ADAPTER == VTKM_DEVICE_ADAPTER_DEMO
#include <vtkm/cont/demo/internal/ArrayManagerExecutionDemo.h>
#endif
```

vtkm/cont/internal/DeviceAdapterTag.h:

```
// At the bottom of the file with the other backend conditional above the
// ADAPTER_ERROR tag add:
#elif VTKM_DEVICE_ADAPTER == VTKM_DEVICE_ADAPTER_DEMO
#include <vtkm/cont/demo/internal/DeviceAdapterTagDemo.h>
#define VTKM_DEFAULT_DEVICE_ADAPTER_TAG ::vtkm::cont::DeviceAdapterTagDemo
```

Testing Your Backend

VTK-m should now recognize and make available your backend, so now we'd like to also build and run the test suite to make sure everything's ok

Under `vtkm/cont/demo/testing` we'll create files to run the unit tests

`CMakeLists.txt` for our unit tests:

```
set(unit_tests
    UnitTestDeviceAdapterDemo.cxx
    UnitTestDemoArrayHandle.cxx
    UnitTestDemoArrayHandleFancy.cxx
)
vtkm_unit_tests(Demo SOURCES ${unit_tests})
```

Device Adapter Unit Tests

Within `UnitTestDeviceAdapterDemo.cxx` we'll use the TestingDeviceAdapter class to generate and run tests for our backend:

```
#define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_ERROR

#include <vtkm/cont/demo/DeviceAdapterDemo.h>
#include <vtkm/cont/testing/TestingDeviceAdapter.h>

int UnitTestDeviceAdapterDemo(int, char *[]){
    return vtkm::cont::testing::TestingDeviceAdapter
        <vtkm::cont::DeviceAdapterTagDemo>::Run();
}
```

Array Handle Unit Tests

Within `UnitTestDemoArrayHandle.cxx` we'll use the TestingDeviceAdapter class to generate and run tests for our backend:

```
#define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_ERROR

#include <vtkm/cont/demo/DeviceAdapterDemo.h>
#include <vtkm/cont/testing/TestingArrayHandles.h>

int UnitTestDemoArrayHandle(int, char *[]){
    return vtkm::cont::testing::TestingArrayHandles
        <vtkm::cont::DeviceAdapterTagDemo>::Run();
}
```

Fancy Array Handle Unit Tests

Within `UnitTestDemoArrayHandleFancy.cxx` we'll use the TestingDeviceAdapter class to generate and run tests for our backend:

```
#define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_ERROR

#include <vtkm/cont/demo/DeviceAdapterDemo.h>
#include <vtkm/cont/testing/TestingFancyArrayHandles.h>

int UnitTestDemoArrayHandleFancy(int, char *[]){
    return vtkm::cont::testing::TestingFancyArrayHandles
        <vtkm::cont::DeviceAdapterTagDemo>::Run();
}
```

Worklet Tests

Fortunately the worklet tests require much less effort, we just add a condition to build them for our backend if it's enabled to `vtkm/worklet/testing/CMakeLists.txt`

```
if (VTKm_ENABLE_DEMO)
    vtkm_worklet_unit_tests(VTKM_DEVICE_ADAPTER_Demo)
endif()
```

Benchmarking our Backend

Adding our backend to the micro benchmarks suite is similar to building the worklet tests, we just add a similar condition to `vtkm/benchmarking/CMakeLists.txt`

```
if (VTKm_ENABLE_DEMO)
    vtkm_benchmarks(VTKM_DEVICE_ADAPTER_Demo)
endif()
```

Building

Now we can run CMake, turn on `VTKm_ENABLE_DEMO` and have our new backend built along with tests and benchmarks for it

Testing: Run tests with just `make test`

Benchmarking: An executable named `Benchmarks_Demo` will be produced, run this and select the benchmark suite to run

An example of the Demo backend is available on Gitlab