Umpire Documentation

Release 5.0.1

David Beckingsale

Jun 14, 2021

BASICS

Umpire is a resource management library that allows the discovery, provision, and management of memory on next-generation hardware architectures with NUMA memory hierarchies.

- Take a look at our Getting Started guide for all you need to get up and running with Umpire.
- If you are looking for developer documentation on a particular function, check out the code documentation.
- Want to contribute? Take a look at our developer and contribution guides.

Any questions? File an issue on GitHub, or email umpire-dev@llnl.gov

CHAPTER

ONE

GETTING STARTED

This page provides information on how to quickly get up and running with Umpire.

1.1 Installation

Umpire is hosted on GitHub here. To clone the repo into your local working space, type:

```
$ git clone --recursive https://github.com/LLNL/Umpire.git
```

The `--recursive` argument is required to ensure that the *BLT* submodule is also checked out. BLT is the build system we use for Umpire.

1.1.1 Building Umpire

Umpire uses CMake and BLT to handle builds. Make sure that you have a modern compiler loaded and the configuration is as simple as:

```
$ mkdir build && cd build
$ cmake ../
```

By default, Umpire will only support host memory. Additional backends for device support can be enabled using the options detailed in *Advanced Configuration*. CMake will provide output about which compiler is being used and the values of other options. Once CMake has completed, Umpire can be built with Make:

```
$ make
```

For more advanced configuration options, see *Advanced Configuration*.

1.1.2 Installing Umpire

To install Umpire, just run:

```
$ make install
```

Umpire install files to the `lib`, `include` and `bin` directories of the `CMAKE_INSTALL_PREFIX`. Additionally, Umpire installs a CMake configuration file that can help you use Umpire in other projects. By setting *umpire_DIR* to point to the root of your Umpire installation, you can call `find_package(umpire)` inside your CMake project and Umpire will be automatically detected and available for use.

1.2 Basic Usage

Let's take a quick tour through Umpire's most important features. A complete listing you can compile is included at the bottom of the page. First, let's grab an Allocator and allocate some memory. This is the interface through which you will want to access data:

```
auto& rm = umpire::ResourceManager::getInstance();
umpire::Allocator allocator = rm.getAllocator("HOST");

float* my_data = static_cast<float*>(allocator.allocate(100*sizeof(float));
```

This code grabs the default allocator for the host memory, and uses it to allocate an array of 100 floats. We can ask for different Allocators to allocate memory in different places. Let's ask for a device allocator:

```
umpire::Allocator device_allocator = rm.getAllocator("DEVICE");

float* my_data_device = static_cast<float*>(device_allocator.allocate(100*sizeof(float));
```

This code gets the default device allocator, and uses it to allocate an array of 100 floats. Remember, since this is a device pointer, there is no guarantee you will be able to access it on the host. Luckily, Umpire's ResourceManager can copy one pointer to another transparently. Let's copy the data from our first pointer to the DEVICE-allocated pointer.

```
rm.copy(my_data, my_data_device);
```

To free any memory allocated, you can use the deallocate function of the Allocator, or the ResourceManager. Asking the ResourceManager to deallocate memory is slower, but useful if you don't know how or where an allocation was made:

```
allocator.deallocate(my_data); // deallocate using Allocator
rm.deallocate(my_data_device); // deallocate using ResourceManager
```

CHAPTER

TWO

UMPIRE TUTORIAL

This section is a tutorial introduction to Umpire. We start with the most basic memory allocation, and move through topics like allocating on different resources, using allocation strategies to change how memory is allocated, using operations to move and modify data, and how to use Umpire introspection capability to find out information about Allocators and allocations.

These examples are all built as part of Umpire, and you can find the files in the examples directory at the root of the Umpire repository. Feel free to play around and modify these examples to experiment with all of Umpire's functionality.

The following tutorial examples assume a working knowledge of C++ and a general understanding of how memory is laid out in modern heterogeneous computers. The main thing to remember is that in many systems, memory on other execution devices (like GPUs) might not be directly accessible from the CPU. If you try and access this memory your program will error! Luckily, Umpire makes it easy to move data around, and check where it is, as you will see in the following sections.

2.1 Allocators

The fundamental concept for accessing memory through Umpire is the `umpire::Allocator`. An `umpire::Allocator` is a C++ object that can be used to allocate and deallocate memory, as well as query a pointer to get some extra information about it.

All `umpire::Allocator` s are created and managed by Umpire's `umpire::ResourceManager`. To get an Allocator, you need to ask for one:

```
umpire::Allocator allocator = rm.getAllocator("HOST");
```

You can also use an existing allocator to build a new allocator from it:

```
auto addon_allocator = rm.getAllocator(allocator.getName());
```

This new allocator will also be built with the same memory resource. More information on memory resources is provided in the next section. Additionally, once you have an `umpire::Allocator` you can use it to allocate and deallocate memory:

```
double* data =
    static_cast<double*>(allocator.allocate(SIZE * sizeof(double)));
```

```
allocator.deallocate(data);
```

In the next section, we will see how to allocate memory using different resources.

```cpp
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

int main(int, char**)
{
  auto& rm = umpire::ResourceManager::getInstance();

  // _sphinx_tag_tut_get_allocator_start
  umpire::Allocator allocator = rm.getAllocator("HOST");
  // _sphinx_tag_tut_get_allocator_end

  constexpr std::size_t SIZE = 1024;

  // _sphinx_tag_tut_allocate_start
  double* data =
      static_cast<double*>(allocator.allocate(SIZE * sizeof(double)));
  // _sphinx_tag_tut_allocate_end

  std::cout << "Allocated " << (SIZE * sizeof(double)) << " bytes using the "
            << allocator.getName() << " allocator." << std::endl;

  //_sphinx_tag_tut_getAllocator_start
  auto addon_allocator = rm.getAllocator(allocator.getName());
  //_sphinx_tag_tut_getAllocator_end

  std::cout << "Created an add-on allocator of size " << addon_allocator.getCurrentSize()
            << " using the " << allocator.getName() << " allocator." << std::endl;

  // _sphinx_tag_tut_deallocate_start
  allocator.deallocate(data);
  // _sphinx_tag_tut_deallocate_end

  std::cout << "...Memory deallocated." << std::endl;

  return 0;
}
```

2.2 Resources

Each computer system will have a number of distinct places in which the system will allow you to allocate memory. In Umpire's world, these are *memory resources*. A memory resource can correspond to a hardware resource, but can also be used to identify memory with a particular characteristic, like "pinned" memory in a GPU system.

When you configure Umpire, it will create `umpire::resource::MemoryResource` s according to what is available on the system you are building for. For each resource (defined by `MemoryResourceTraits::resource_type`), Umpire will create a default `umpire::Allocator` that you can use. In the previous example, we were actually using an `umpire::Allocator` created for the memory resource corresponding to the CPU memory.

The easiest way to identify resources is by name. The "HOST" resource is always available. We also have resources that represent global GPU memory ("DEVICE"), constant GPU memory ("DEVICE_CONST"), unified memory that can be accessed by the CPU or GPU ("UM"), host memory that can be accessed by the GPU ("PINNED"), and mmapped file memory ("FILE"). If an incorrect name is used or if the allocator was not set up correctly, the "UNKNOWN" resource name is returned.

Umpire will create an `umpire::Allocator` for each of these resources, and you can get them using the same `umpire::ResourceManager::getAllocator()` call you saw in the previous example:

```
  umpire::Allocator allocator = rm.getAllocator(resource);
```

Note that since every allocator supports the same calls, no matter which resource it is for, this means we can run the same code for all the resources available in the system.

While using Umpire memory resources, it may be useful to query the memory resource currently associated with a particular allocator. For example, if we wanted to double check that our allocator is using the `device` resource, we can assert that `MemoryResourceTraits::resource_type::device` is equal to the return value of `allocator.getAllocationStrategy()->getTraits().resource`. The test code provided in `memory_resource_traits_tests.cpp` shows a complete example of how to query this information.

Note: In order to test some memory resources, you may need to configure your Umpire build to use a particular platform (a member of the `umpire::Allocator`, defined by `Platform.hpp`) that has access to that resource. See the Developer's Guide for more information.

Next, we will see an example of how to move data between resources using operations.

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

void allocate_and_deallocate(const std::string& resource)
{
  auto& rm = umpire::ResourceManager::getInstance();

  // _sphinx_tag_tut_get_allocator_start
  umpire::Allocator allocator = rm.getAllocator(resource);
  // _sphinx_tag_tut_get_allocator_end
```

(continues on next page)

(continued from previous page)

```
  constexpr std::size_t SIZE = 1024;

  double* data =
      static_cast<double*>(allocator.allocate(SIZE * sizeof(double)));

  std::cout << "Allocated " << (SIZE * sizeof(double)) << " bytes using the "
            << allocator.getName() << " allocator...";

  allocator.deallocate(data);

  std::cout << " deallocated." << std::endl;
}

int main(int, char**)
{
  allocate_and_deallocate("HOST");

#if defined(UMPIRE_ENABLE_DEVICE)
  allocate_and_deallocate("DEVICE");
#endif
#if defined(UMPIRE_ENABLE_UM)
  allocate_and_deallocate("UM");
#endif
#if defined(UMPIRE_ENABLE_PINNED)
  allocate_and_deallocate("PINNED");
#endif

  return 0;
}
```

2.3 Operations

Moving and modifying data in a heterogenous memory system can be annoying. You have to keep track of the source and destination, and often use vendor-specific APIs to perform the modifications. In Umpire, all data modification and movement is wrapped up in a concept we call *operations*. Full documentation for all of these is available here. The full code listing for each example is include at the bottom of the page.

2.3.1 Copy

Let's start by looking at how we copy data around. The `umpire::ResourceManager` provides an interface to copy that handles figuring out where the source and destination pointers were allocated, and selects the correct implementation to copy the data:

```
  rm.copy(dest_data, source_data);
```

This example allocates the destination data using any valid Allocator.

2.3.2 Move

If you want to move data to a new Allocator and deallocate the old copy, Umpire provides a `umpire::ResourceManager::move()` operation.

```
  double* dest_data =
      static_cast<double*>(rm.move(source_data, dest_allocator));
```

The move operation combines an allocation, a copy, and a deallocate into one function call, allowing you to move data without having to have the destination data allocated. As always, this operation will work with any valid destination Allocator.

2.3.3 Memset

Setting a whole block of memory to a value (like 0) is a common operation, that most people know as a memset. Umpire provides a `umpire::ResourceManager::memset()` implementation that can be applied to any allocation, regardless of where it came from:

```
    rm.memset(data, 0);
```

2.3.4 Reallocate

Reallocating CPU memory is easy, there is a function designed specifically to do it: `realloc`. When the original allocation was made in a different memory however, you can be out of luck. Umpire provides a `umpire::ResourceManager::reallocate()` operation:

```
    data = static_cast<double*>(rm.reallocate(data, REALLOCATED_SIZE));
```

This method returns a pointer to the reallocated data. Like all operations, this can be used regardless of the Allocator used for the source data.

2.3.5 Listings

Copy Example Listing

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

void copy_data(double* source_data, std::size_t size,
               const std::string& destination)
{
  auto& rm = umpire::ResourceManager::getInstance();
  auto dest_allocator = rm.getAllocator(destination);

  double* dest_data =
```

(continues on next page)

(continued from previous page)

```
      static_cast<double*>(dest_allocator.allocate(size * sizeof(double)));

  // _sphinx_tag_tut_copy_start
  rm.copy(dest_data, source_data);
  // _sphinx_tag_tut_copy_end

  std::cout << "Copied source data (" << source_data << ") to destination "
            << destination << " (" << dest_data << ")" << std::endl;

  dest_allocator.deallocate(dest_data);
}

int main(int, char**)
{
  constexpr std::size_t SIZE = 1024;

  auto& rm = umpire::ResourceManager::getInstance();

  auto allocator = rm.getAllocator("HOST");

  double* data =
      static_cast<double*>(allocator.allocate(SIZE * sizeof(double)));

  std::cout << "Allocated " << (SIZE * sizeof(double)) << " bytes using the "
            << allocator.getName() << " allocator." << std::endl;

  std::cout << "Filling with 0.0...";

  for (std::size_t i = 0; i < SIZE; i++) {
    data[i] = 0.0;
  }

  std::cout << "done." << std::endl;

  copy_data(data, SIZE, "HOST");
#if defined(UMPIRE_ENABLE_DEVICE)
  copy_data(data, SIZE, "DEVICE");
#endif
#if defined(UMPIRE_ENABLE_UM)
  copy_data(data, SIZE, "UM");
#endif
#if defined(UMPIRE_ENABLE_PINNED)
  copy_data(data, SIZE, "PINNED");
#endif

  allocator.deallocate(data);

  return 0;
}
```

Move Example Listing

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

double* move_data(double* source_data, const std::string& destination)
{
  auto& rm = umpire::ResourceManager::getInstance();
  auto dest_allocator = rm.getAllocator(destination);

  std::cout << "Moved source data (" << source_data << ") to destination ";

  // _sphinx_tag_tut_move_start
  double* dest_data =
      static_cast<double*>(rm.move(source_data, dest_allocator));
  // _sphinx_tag_tut_move_end

  std::cout << destination << " (" << dest_data << ")" << std::endl;

  return dest_data;
}

int main(int, char**)
{
  constexpr std::size_t SIZE = 1024;

  auto& rm = umpire::ResourceManager::getInstance();

  auto allocator = rm.getAllocator("HOST");

  double* data =
      static_cast<double*>(allocator.allocate(SIZE * sizeof(double)));

  std::cout << "Allocated " << (SIZE * sizeof(double)) << " bytes using the "
            << allocator.getName() << " allocator." << std::endl;

  std::cout << "Filling with 0.0...";

  for (std::size_t i = 0; i < SIZE; i++) {
    data[i] = 0.0;
  }

  std::cout << "done." << std::endl;

  data = move_data(data, "HOST");
#if defined(UMPIRE_ENABLE_DEVICE)
  data = move_data(data, "DEVICE");
#endif
#if defined(UMPIRE_ENABLE_UM)
```

(continues on next page)

(continued from previous page)

```
  data = move_data(data, "UM");
#endif
#if defined(UMPIRE_ENABLE_PINNED)
  data = move_data(data, "PINNED");
#endif

  rm.deallocate(data);

  return 0;
}
```

Memset Example Listing

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

int main(int, char**)
{
  constexpr std::size_t SIZE = 1024;

  auto& rm = umpire::ResourceManager::getInstance();

  const std::string destinations[] = {
    "HOST"
#if defined(UMPIRE_ENABLE_DEVICE)
    ,
    "DEVICE"
#endif
#if defined(UMPIRE_ENABLE_UM)
    ,
    "UM"
#endif
#if defined(UMPIRE_ENABLE_PINNED)
    ,
    "PINNED"
#endif
  };

  for (auto& destination : destinations) {
    auto allocator = rm.getAllocator(destination);
    double* data =
        static_cast<double*>(allocator.allocate(SIZE * sizeof(double)));

    std::cout << "Allocated " << (SIZE * sizeof(double)) << " bytes using the "
              << allocator.getName() << " allocator." << std::endl;
```

(continues on next page)

(continued from previous page)

```
    // _sphinx_tag_tut_memset_start
    rm.memset(data, 0);
    // _sphinx_tag_tut_memset_end

    std::cout << "Set data from " << destination << " (" << data << ") to 0."
              << std::endl;

    allocator.deallocate(data);
  }

  return 0;
}
```

Reallocate Example Listing

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

int main(int, char**)
{
  constexpr std::size_t SIZE = 1024;
  constexpr std::size_t REALLOCATED_SIZE = 256;

  auto& rm = umpire::ResourceManager::getInstance();

  const std::string destinations[] = {
    "HOST"
#if defined(UMPIRE_ENABLE_DEVICE)
    ,
    "DEVICE"
#endif
#if defined(UMPIRE_ENABLE_UM)
    ,
    "UM"
#endif
#if defined(UMPIRE_ENABLE_PINNED)
    ,
    "PINNED"
#endif
  };

  for (auto& destination : destinations) {
    auto allocator = rm.getAllocator(destination);
    double* data =
        static_cast<double*>(allocator.allocate(SIZE * sizeof(double)));
```

(continues on next page)

(continued from previous page)

```
    std::cout << "Allocated " << (SIZE * sizeof(double)) << " bytes using the "
              << allocator.getName() << " allocator." << std::endl;

    std::cout << "Reallocating data (" << data << ") to size "
              << REALLOCATED_SIZE << "...";

    // _sphinx_tag_tut_realloc_start
    data = static_cast<double*>(rm.reallocate(data, REALLOCATED_SIZE));
    // _sphinx_tag_tut_realloc_end

    std::cout << "done.  Reallocated data (" << data << ")" << std::endl;

    allocator.deallocate(data);
  }

  return 0;
}
```

2.4 Dynamic Pools

Frequently allocating and deallocating memory can be quite costly, especially when you are making large allocations or allocating on different memory resources. To mitigate this, Umpire provides allocation strategies that can be used to customize how data is obtained from the system.

In this example, we will look at the `umpire::strategy::DynamicPoolList` strategy. This is a simple pooling algorithm that can fulfill requests for allocations of any size. To create a new `Allocator` using the `umpire::strategy::DynamicPoolList` strategy:

```
  auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPoolList>(
      resource + "_pool", allocator);
```

We have to provide a new name for the Allocator, as well as the underlying Allocator we wish to use to grab memory.

Additionally, in the previous section on Allocators, we mentioned that you could build a new allocator off of an existing one using the `getAllocator` function. Here is another example of this, but using a strategy:

```
umpire::Allocator addon_allocator = rm.makeAllocator<umpire::strategy::SizeLimiter>(
resource + "_addon_pool", rm.getAllocator(pooled_allocator.getName()), 2098);
```

The purpose of this example is to show that the `getAllocator` function can be used more than just to get an initial allocator. The `addon_allocator` will be a dynamic pool allocator that is limited to 2098 bytes. Another good use case for the `getAllocator` function is grabbing each available allocator in a loop and querying some property. (Note that `addon_allocator` in the above example will be created with the same memory resource as `pooled_allocator` was.)

Once you have an `Allocator`, you can allocate and deallocate memory as before, without needing to worry about the underlying algorithm used for the allocations:

```
  double* data =
      static_cast<double*>(pooled_allocator.allocate(SIZE * sizeof(double)));
```

```
  pooled_allocator.deallocate(data);
```

Don't forget, these strategies can be created on top of any valid Allocator:

```
  allocate_and_deallocate_pool("HOST");

#if defined(UMPIRE_ENABLE_DEVICE)
  allocate_and_deallocate_pool("DEVICE");
#endif
#if defined(UMPIRE_ENABLE_UM)
  allocate_and_deallocate_pool("UM");
#endif
#if defined(UMPIRE_ENABLE_PINNED)
  allocate_and_deallocate_pool("PINNED");
#endif
```

Most Umpire users will make allocations that use the GPU via the `umpire::strategy::DynamicPoolList`, to help mitigate the cost of allocating memory on these devices.

You can tune the way that `umpire::strategy::DynamicPoolList` allocates memory using two parameters: the initial size, and the minimum size. The initial size controls how large the first underly allocation made will be, regardless of the requested size. The minimum size controls the minimum size of any future underlying allocations. These two parameters can be passed when constructing a pool:

```
  auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPoolList>(
      resource + "_pool", allocator, initial_size, /* default = 512Mb*/
      min_block_size /* default = 1Mb */);
```

Depending on where you are allocating data, you might want to use different sizes. It's easy to construct multiple pools with different configurations:

```
  allocate_and_deallocate_pool("HOST", 65536, 512);
#if defined(UMPIRE_ENABLE_DEVICE)
  allocate_and_deallocate_pool("DEVICE", (1024 * 1024 * 1024), (1024 * 1024));
#endif
#if defined(UMPIRE_ENABLE_UM)
  allocate_and_deallocate_pool("UM", (1024 * 64), 1024);
#endif
#if defined(UMPIRE_ENABLE_PINNED)
  allocate_and_deallocate_pool("PINNED", (1024 * 16), 1024);
#endif
```

There are lots of different strategies that you can use, we will look at some of them in this tutorial. A complete list of strategies can be found here.

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/DynamicPoolList.hpp"
```

(continues on next page)

(continued from previous page)

```
void allocate_and_deallocate_pool(const std::string& resource)
{
  auto& rm = umpire::ResourceManager::getInstance();

  auto allocator = rm.getAllocator(resource);

  // _sphinx_tag_tut_makepool_start
  auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPoolList>(
      resource + "_pool", allocator);
  // _sphinx_tag_tut_makepool_end

  constexpr std::size_t SIZE = 1024;

  // _sphinx_tag_tut_allocate_start
  double* data =
      static_cast<double*>(pooled_allocator.allocate(SIZE * sizeof(double)));
  // _sphinx_tag_tut_allocate_end

  std::cout << "Allocated " << (SIZE * sizeof(double)) << " bytes using the "
            << pooled_allocator.getName() << " allocator...";

  // _sphinx_tag_tut_deallocate_start
  pooled_allocator.deallocate(data);
  // _sphinx_tag_tut_deallocate_end

  std::cout << " deallocated." << std::endl;
}

int main(int, char**)
{
  // _sphinx_tag_tut_anyallocator_start
  allocate_and_deallocate_pool("HOST");

#if defined(UMPIRE_ENABLE_DEVICE)
  allocate_and_deallocate_pool("DEVICE");
#endif
#if defined(UMPIRE_ENABLE_UM)
  allocate_and_deallocate_pool("UM");
#endif
#if defined(UMPIRE_ENABLE_PINNED)
  allocate_and_deallocate_pool("PINNED");
#endif
  // _sphinx_tag_tut_anyallocator_end

  return 0;
}
```

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
```

(continues on next page)

(continued from previous page)

```cpp
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/DynamicPoolList.hpp"

void allocate_and_deallocate_pool(const std::string& resource,
                                  std::size_t initial_size,
                                  std::size_t min_block_size)
{
  constexpr std::size_t SIZE = 1024;

  auto& rm = umpire::ResourceManager::getInstance();

  auto allocator = rm.getAllocator(resource);

  // _sphinx_tag_tut_allocator_tuning_start
  auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPoolList>(
      resource + "_pool", allocator, initial_size, /* default = 512Mb*/
      min_block_size /* default = 1Mb */);
  // _sphinx_tag_tut_allocator_tuning_end

  double* data =
      static_cast<double*>(pooled_allocator.allocate(SIZE * sizeof(double)));

  std::cout << "Allocated " << (SIZE * sizeof(double)) << " bytes using the "
            << pooled_allocator.getName() << " allocator...";

  pooled_allocator.deallocate(data);

  std::cout << " deallocated." << std::endl;
}

int main(int, char**)
{
  // _sphinx_tag_tut_device_sized_pool_start
  allocate_and_deallocate_pool("HOST", 65536, 512);
#if defined(UMPIRE_ENABLE_DEVICE)
  allocate_and_deallocate_pool("DEVICE", (1024 * 1024 * 1024), (1024 * 1024));
#endif
#if defined(UMPIRE_ENABLE_UM)
  allocate_and_deallocate_pool("UM", (1024 * 64), 1024);
#endif
#if defined(UMPIRE_ENABLE_PINNED)
  allocate_and_deallocate_pool("PINNED", (1024 * 16), 1024);
#endif
  // _sphinx_tag_tut_device_sized_pool_end

  return 0;
}
```

2.5 Introspection

When writing code to run on computers with a complex memory hierarchy, one of the most difficult things can be keeping track of where each pointer has been allocated. Umpire's instrospection capability keeps track of this information, as well as other useful bits and pieces you might want to know.

The `umpire::ResourceManager` can be used to find the allocator associated with an address:

```
    auto found_allocator = rm.getAllocator(data);
```

Once you have this, it's easy to query things like the name of the Allocator or find out the associated `umpire::Platform`, which can help you decide where to operate on this data:

```
    std::cout << "According to the ResourceManager, the Allocator used is "
              << found_allocator.getName() << ", which has the Platform "
              << static_cast<int>(found_allocator.getPlatform()) << std::endl;
```

You can also find out how big the allocation is, in case you forgot:

```
    std::cout << "The size of the allocation is << "
              << found_allocator.getSize(data) << std::endl;
```

Remember that these functions will work on any allocation made using an Allocator or `umpire::TypedAllocator`.

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

int main(int, char**)
{
  constexpr std::size_t SIZE = 1024;

  auto& rm = umpire::ResourceManager::getInstance();

  const std::string destinations[] = {
    "HOST"
#if defined(UMPIRE_ENABLE_DEVICE)
    ,
    "DEVICE"
#endif
#if defined(UMPIRE_ENABLE_UM)
    ,
    "UM"
#endif
#if defined(UMPIRE_ENABLE_PINNED)
    ,
    "PINNED"
#endif
  };
```

(continues on next page)

(continued from previous page)

```
  for (auto& destination : destinations) {
    auto allocator = rm.getAllocator(destination);
    double* data =
        static_cast<double*>(allocator.allocate(SIZE * sizeof(double)));

    std::cout << "Allocated " << (SIZE * sizeof(double)) << " bytes using the "
              << allocator.getName() << " allocator." << std::endl;

    // _sphinx_tag_tut_getallocator_start
    auto found_allocator = rm.getAllocator(data);
    // _sphinx_tag_tut_getallocator_end

    // _sphinx_tag_tut_getinfo_start
    std::cout << "According to the ResourceManager, the Allocator used is "
              << found_allocator.getName() << ", which has the Platform "
              << static_cast<int>(found_allocator.getPlatform()) << std::endl;
    // _sphinx_tag_tut_getinfo_end

    // _sphinx_tag_tut_getsize_start
    std::cout << "The size of the allocation is << "
              << found_allocator.getSize(data) << std::endl;
    // _sphinx_tag_tut_getsize_end

    allocator.deallocate(data);
  }

  return 0;
}
```

2.6 Typed Allocators

Sometimes, you might want to construct an allocator that allocates objects of a specific type. Umpire provides a `umpire::TypedAllocator` for this purpose. It can also be used with STL objects like `std::vector`.

A `umpire::TypedAllocator` is constructed from any existing Allocator, and provides the same interface as the normal `umpire::Allocator`. However, when you call allocate, this argument is the number of objects you want to allocate, no the total number of bytes:

```
  umpire::TypedAllocator<double> double_allocator{alloc};

  double* my_doubles = double_allocator.allocate(1024);

  double_allocator.deallocate(my_doubles, 1024);
```

To use this allocator with an STL object like a vector, you need to pass the type as a template parameter for the vector, and also pass the allocator to the vector when you construct it:

```
  std::vector<double, umpire::TypedAllocator<double>> my_vector{
      double_allocator};
```

One thing to remember is that whatever allocator you use with an STL object, it must be compatible with the inner workings of that object. For example, if you try and use a "DEVICE"-based allocator it will fail, since the vector will try and construct each element. The CPU cannot access DEVICE memory in most systems, thus causing a segfault. Be careful!

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/TypedAllocator.hpp"

int main(int, char**)
{
  auto& rm = umpire::ResourceManager::getInstance();
  auto alloc = rm.getAllocator("HOST");

  // _sphinx_tag_tut_typed_alloc_start
  umpire::TypedAllocator<double> double_allocator{alloc};

  double* my_doubles = double_allocator.allocate(1024);

  double_allocator.deallocate(my_doubles, 1024);
  // _sphinx_tag_tut_typed_alloc_end

  // _sphinx_tag_tut_vector_alloc_start
  std::vector<double, umpire::TypedAllocator<double>> my_vector{
      double_allocator};
  // _sphinx_tag_tut_vector_alloc_end

  my_vector.resize(100);

  return 0;
}
```

2.7 Replay

Umpire provides a lightweight replay capability that can be used to investigate performance of particular allocation patterns and reproduce bugs.

2.7.1 Input Example

A log can be captured and stored as a JSON file, then used as input to the `replay` application (available under the `bin` directory). The `replay` program will read the replay log, and recreate the events that occured as part of the run that generated the log.

The file `tut_replay.cpp` makes a `umpire::strategy::QuickPool`:

```
  auto allocator = rm.getAllocator("HOST");
  auto pool =
      rm.makeAllocator<umpire::strategy::QuickPool>("pool", allocator);
```

This allocator is used to perform some randomly sized allocations, and later free them:

```
  std::generate(allocations.begin(), allocations.end(),
                [&]() { return pool.allocate(random_number()); });
```

```
  for (auto& ptr : allocations)
    pool.deallocate(ptr);
```

2.7.2 Running the Example

Running this program:

```
UMPIRE_REPLAY="On" ./bin/examples/tutorial/tut_replay > tut_replay_log.json
```

will write Umpire replay events to the file `tut_replay_log.json`. You can see that this file contains JSON formatted lines.

2.7.3 Replaying the session

Loading this file with the `replay` program will replay this sequence of `umpire::Allocator` creation, allocations, and deallocations:

```
./bin/replay -i ../tutorial/examples/tut_replay_log.json
```

We also have a tutorial for the C interface to Umpire. Complete example listings are available, and will be compiled if you have configured Umpire with `-DENABLE_C=On`.

The C tutorial assumes an understanding of C, and it would be useful to have some knowledge of C++ to understand how the C API maps to the native C++ classes that Umpire provides.

2.8 C API: Allocators

The fundamental concept for accessing memory through Umpire is an `umpire:Allocator`. In C, this means using the type `umpire_allocator`. There are corresponding functions that take an `umpire_allocator` and let you allocate and deallocate memory.

As with the native C++ interface, all allocators are accessed via the `umpire::ResourceManager`. In the C API, there is a corresponding `umpire_resourcemanager` type. To get an `umpire_allocator`:

```
umpire_resourcemanager rm;
umpire_resourcemanager_get_instance(&rm);

umpire_allocator allocator;
umpire_resourcemanager_get_allocator_by_name(&rm, "HOST", &allocator);
```

Once you have an `umpire_allocator`, you can use it to allocate and deallocate memory:

```
double* data = (double*) umpire_allocator_allocate(&allocator, SIZE*sizeof(double));

printf("Allocated %lu bytes using the %s allocator...", (SIZE*sizeof(double)), umpire_
↪allocator_get_name(&allocator));

umpire_allocator_deallocate(&allocator, data);
```

In the next section, we will see how to allocate memory in different places.

2.9 C API: Resources

Each computer system will have a number of distinct places in which the system will allow you to allocate memory. In Umpire's world, these are *memory resources*. A memory resource can correspond to a hardware resource, but can also be used to identify memory with a particular characteristic, like "pinned" memory in a GPU system.

When you configure Umpire, it will create `umpire::resource::MemoryResource` s according to what is available on the system you are building for. For each resource, Umpire will create a default `umpire_allocator` that you can use. In the previous example, we were actually using an `umpire_allocator` created for the memory resource corresponding to the CPU memory.

The easiest way to identify resources is by name. The "HOST" resource is always available. In a system configured with NVIDIA GPUs, we also have resources that represent global GPU memory ("DEVICE"), unified memory that can be accessed by the CPU or GPU ("UM") and host memory that can be accessed by the GPU ("PINNED");

Umpire will create an `umpire_allocator` for each of these resources, and you can get them using the same `umpire_resourcemanager_get_allocator_by_name` call you saw in the previous example:

Note that every allocator supports the same calls, no matter which resource it is for, this means we can run the same code for all the resources available in the system:

As you can see, we can call this function with any valid resource name:

In the next example, we will learn how to move data between resources using operations.

2.10 C API: Pools

Frequently allocating and deallocating memory can be quite costly, especially when you are making large allocations or allocating on different memory resources. To mitigate this, Umpire provides allocation strategies that can be used to customize how data is obtained from the system.

In this example, we will look at creating a pool that can fulfill requests for allocations of any size. To create a new `umpire_allocator` using the pooling algorithm:

The two arguments are the size of the initial block that is allocated, and the minimum size of any future blocks. We have to provide a new name for the allocator, as well as the underlying `umpire_allocator` we wish to use to grab memory.

Once you have the allocator, you can allocate and deallocate memory as before, without needing to worry about the underlying algorithm used for the allocations:

This pool can be created with any valid underlying `umpire_allocator`.

Finally, we have a tutorial for Umpire's FORTRAN API. These examples will be compiled when configuring with `-DENABLE_FORTRAN=On`. The FORTRAN tutorial assumes an understanding of FORTRAN. Familiarity with the FORTRAN's ISO C bindings can be useful for understanding why the interface looks the way it does.

2.11 FORTRAN API: Allocators

The fundamental concept for accessing memory through Umpire is an `umpire:Allocator`. In FORTRAN, this means using the type `UmpireAllocator`. This type provides an `allocate_pointer` function to allocate raw memory, and a generic `allocate` procedure that takes an array pointer and an array of dimensions and will allocate the correct amount of memory.

As with the native C++ interface, all allocators are accessed via the `umpire::ResourceManager`. In the FORTRAN API, there is a corresponding `UmpireResourceManager` type. To get an `UmpireAllocator`:

In this example we fetch the allocator by id, using 0 means you will always get a host allocator. Once you have an `UmpireAllocator`, you can use it to allocate and deallocate memory:

In this case, we allocate a one-dimensional array using the generic `allocate` function.

CHAPTER

THREE

ADVANCED CONFIGURATION

In addition to the normal options provided by CMake, Umpire uses some additional configuration arguments to control optional features and behavior. Each argument is a boolean option, and can be turned on or off:

```
-DENABLE_CUDA=Off
```

Here is a summary of the configuration options, their default value, and meaning:

Variable	Default	Meaning
ENABLE_CUDA	Off	Enable CUDA support
ENABLE_HIP	Off	Enable HIP support
ENABLE_NUMA	Off	Enable NUMA support
ENABLE_FILE_RESOURCE	Off	Enable FILE support
ENABLE_TESTS	On	Build test executables
ENABLE_BENCHMARKS	On	Build benchmark programs
ENABLE_LOGGING	On	Enable Logging within Umpire
ENABLE_SLIC	Off	Enable SLIC logging
ENABLE_BACKTRACE	Off	Enable backtraces for allocations
ENABLE_BACKTRACE_SYMBOLS	Off	Enable symbol lookup for backtraces
ENABLE_TOOLS	Off	Enable tools like replay
ENABLE_DOCS	Off	Build documentation (requires Sphinx and/or Doxygen)
ENABLE_C	Off	Build the C API
ENABLE_FORTRAN	Off	Build the Fortran API
ENABLE_PERFORMANCE_TESTS	Off	Build and run performance tests
ENABLE_IPC_SHARED_MEMORY	ENABLE_MPI	Enable Shared Memory support
ENABLE_ASAN	Off	Enable ASAN support

These arguments are explained in more detail below:

- ENABLE_CUDA This option enables support for NVIDIA GPUs using the CUDA programming model. If Umpire is built without CUDA or HIP support, then only the HOST allocator is available for use.
- ENABLE_HIP This option enables support for AMD GPUs using the ROCm stack and HIP programming model. If Umpire is built without CUDA or HIP support, then only the HOST allocator is available for use.
- ENABLE_NUMA This option enables support for NUMA. The umpire::strategy::NumaPolicy is available when built with this option, which may be used to locate the allocation to a specific node.
- ENABLE_FILE_RESOURCE This option will allow the build to make all File Memory Allocation files. If Umpire is built without FILE, CUDA or HIP support, then only the HOST allocator is available for use.

- ENABLE_TESTS This option controls whether or not test executables will be built.
- ENABLE_BENCHMARKS This option will build the benchmark programs used to test performance.
- ENABLE_LOGGING This option enables usage of Logging services for Umpire
- ENABLE_SLIC This option enables usage of logging services provided by SLIC.
- ENABLE_BACKTRACE This option enables collection of backtrace information for each allocation.
- ENABLE_BACKTRACE_SYMBOLS This option enables symbol information to be provided with backtraces. This requires -ldl to be specified for using programs.
- ENABLE_TOOLS Enable development tools for Umpire (replay, etc.)
- ENABLE_DOCS Build user documentation (with Sphinx) and code documentation (with Doxygen)
- ENABLE_C Build the C API, this allows accessing Umpire Allocators and the ResourceManager through a C interface.
- ENABLE_FORTRAN Build the Fortran API.
- ENABLE_PERFORMANCE_TESTS Build and run performance tests
- ENABLE_IPC_SHARED_MEMORY This option enables support for interprocess shared memory. Currently, this feature only exists for for HOST memory.
- ENABLE_ASAN This option enables address sanitization checks within Umpire by compilers that support options like `-fsanitize=address`

CHAPTER

FOUR

UMPIRE COOKBOOK

This section provides a set of recipes that show you how to accomplish specific tasks using Umpire. The main focus is things that can be done by composing different parts of Umpire to achieve a particular use case.

Examples include being able to grow and shrink a pool, constructing Allocators that have introspection disabled for improved performance, and applying CUDA "memory advise" to all the allocations in a particular pool.

4.1 Growing and Shrinking a Pool

When sharing a pool between different parts of your application, or even between co-ordinating libraries in the same application, you might want to grow and shrink a pool on demand. By limiting the size of a pool using device memory, you leave more space on the GPU for "unified memory" to move data there.

The basic idea is to create a pool that allocates a block of your minimum size, and then allocate a single word from this pool to ensure the initial block is never freed:

```
auto pooled_allocator = rm.makeAllocator<umpire::strategy::QuickPool>(
    "GPU_POOL", allocator, 4ul * 1024ul * 1024ul * 1024ul + 1);
```

To increase the pool size you can preallocate a large chunk and then immediately free it. The pool will retain this memory for use by later allocations:

```
void* grow = pooled_allocator.allocate(8ul * 1024ul * 1024ul * 1024ul);
pooled_allocator.deallocate(grow);

std::cout << "Pool has allocated " << pooled_allocator.getActualSize()
          << " bytes of memory. " << pooled_allocator.getCurrentSize()
          << " bytes are used" << std::endl;
```

Assuming that there are no allocations left in the larger "chunk" of the pool, you can shrink the pool back down to the initial size by calling `umpire::Allocator::release()`:

```
pooled_allocator.release();
std::cout << "Pool has allocated " << pooled_allocator.getActualSize()
          << " bytes of memory. " << pooled_allocator.getCurrentSize()
          << " bytes are used" << std::endl;
```

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
```

(continues on next page)

(continued from previous page)

```
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/QuickPool.hpp"
#include "umpire/util/Macros.hpp"

int main(int, char**)
{
  auto& rm = umpire::ResourceManager::getInstance();

  auto allocator = rm.getAllocator("DEVICE");

  //
  // Create a 4 Gb pool and reserve one word (to maintain aligment)
  //
  // _sphinx_tag_tut_create_pool_start
  auto pooled_allocator = rm.makeAllocator<umpire::strategy::QuickPool>(
      "GPU_POOL", allocator, 4ul * 1024ul * 1024ul * 1024ul + 1);
  // _sphinx_tag_tut_create_pool_end

  void* hold = pooled_allocator.allocate(64);
  UMPIRE_USE_VAR(hold);

  std::cout << "Pool has allocated " << pooled_allocator.getActualSize()
            << " bytes of memory. " << pooled_allocator.getCurrentSize()
            << " bytes are used" << std::endl;

  //
  // Grow pool to ~12 by grabbing a 8Gb chunk
  //
  // _sphinx_tag_tut_grow_pool_start
  void* grow = pooled_allocator.allocate(8ul * 1024ul * 1024ul * 1024ul);
  pooled_allocator.deallocate(grow);

  std::cout << "Pool has allocated " << pooled_allocator.getActualSize()
            << " bytes of memory. " << pooled_allocator.getCurrentSize()
            << " bytes are used" << std::endl;
  // _sphinx_tag_tut_grow_pool_end

  //
  // Shrink pool back to ~4Gb
  //
  // _sphinx_tag_tut_shrink_pool_back_start
  pooled_allocator.release();
  std::cout << "Pool has allocated " << pooled_allocator.getActualSize()
            << " bytes of memory. " << pooled_allocator.getCurrentSize()
            << " bytes are used" << std::endl;
```

(continues on next page)

(continued from previous page)

```
  // _sphinx_tag_tut_shrink_pool_back_end

  return 0;
}
```

4.2 Disable Introspection

If you know that you won't be using any of Umpire's introspection capabalities for allocations that come from a particular `umpire::Allocator`, you can turn off the introspection and avoid the overhead of tracking the associated metadata.

Warning: Disabling introspection means that allocations from this Allocator cannot be used for operations, or size and location queries.

In this recipe, we look at disabling introspection for a pool. To turn off introspection, you pass a boolean as the second template parameter to the `umpire::ResourceManager::makeAllocator()` method:

```
  auto pooled_allocator =
      rm.makeAllocator<umpire::strategy::QuickPool, false>(
          "NO_INTROSPECTION_POOL", allocator);
```

Remember that disabling introspection will stop tracking the size of allocations made from the pool, so the `umpire::Allocator::getCurrentSize()` method will return 0:

```
  std::cout << "Pool has allocated " << pooled_allocator.getActualSize()
            << " bytes of memory. " << pooled_allocator.getCurrentSize()
            << " bytes are used" << std::endl;
```

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/QuickPool.hpp"
#include "umpire/util/Macros.hpp"

int main(int, char**)
{
  auto& rm = umpire::ResourceManager::getInstance();

  auto allocator = rm.getAllocator("HOST");
```

(continues on next page)

(continued from previous page)

```
  //
  // Create a pool with introspection disabled (can improve performance)
  //
  // _sphinx_tag_tut_nointro_start
  auto pooled_allocator =
      rm.makeAllocator<umpire::strategy::QuickPool, false>(
          "NO_INTROSPECTION_POOL", allocator);
  // _sphinx_tag_tut_nointro_end

  void* data = pooled_allocator.allocate(1024);

  // _sphinx_tag_tut_getsize_start
  std::cout << "Pool has allocated " << pooled_allocator.getActualSize()
            << " bytes of memory. " << pooled_allocator.getCurrentSize()
            << " bytes are used" << std::endl;
  // _sphinx_tag_tut_getsize_end

  pooled_allocator.deallocate(data);

  return 0;
}
```

4.3 Apply Memory Advice to a Pool

When using unified memory on systems with CUDA GPUs, various types of memory advice can be applied to modify how the CUDA runtime moves this memory around between the CPU and GPU. One type of advice that can be applied is "preferred location", and you can specificy where you want the preferred location of the memory to be. This can be useful for ensuring that the memory is kept on the GPU.

By creating a pool on top of an `umpire::strategy::AllocationAdvisor`, you can amortize the cost of applying memory advice:

```
  //
  // Create an allocator that applied "PREFFERED_LOCATION" advice to set the
  // GPU as the preferred location.
  //
  auto preferred_location_allocator =
      rm.makeAllocator<umpire::strategy::AllocationAdvisor>(
          "preferred_location_device", allocator, "PREFERRED_LOCATION");

  //
  // Create a pool using the preferred_location_allocator. This makes all
  // allocations in the pool have the same preferred location, the GPU.
  //
  auto pooled_allocator = rm.makeAllocator<umpire::strategy::QuickPool>(
      "GPU_POOL", preferred_location_allocator);
```

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
```

(continues on next page)

(continued from previous page)

```
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/AllocationAdvisor.hpp"
#include "umpire/strategy/QuickPool.hpp"
#include "umpire/util/Macros.hpp"

int main(int, char**)
{
  auto& rm = umpire::ResourceManager::getInstance();

  auto allocator = rm.getAllocator("UM");

  // _sphinx_tag_tut_pool_advice_start
  //
  // Create an allocator that applied "PREFFERED_LOCATION" advice to set the
  // GPU as the preferred location.
  //
  auto preferred_location_allocator =
      rm.makeAllocator<umpire::strategy::AllocationAdvisor>(
          "preferred_location_device", allocator, "PREFERRED_LOCATION");

  //
  // Create a pool using the preferred_location_allocator. This makes all
  // allocations in the pool have the same preferred location, the GPU.
  //
  auto pooled_allocator = rm.makeAllocator<umpire::strategy::QuickPool>(
      "GPU_POOL", preferred_location_allocator);
  // _sphinx_tag_tut_pool_advice_end

  UMPIRE_USE_VAR(pooled_allocator);

  return 0;
}
```

4.4 Apply Memory Advice with a Specific Device ID

When using unified memory on systems with CUDA GPUs, various types of memory advice can be applied to modify how the CUDA runtime moves this memory around between the CPU and GPU. When applying memory advice, a device ID can be used to specific which device the advice relates to. One type of advice that can be applied is "preferred location", and you can specificy where you want the preferred location of the memory to be. This can be useful for ensuring that the memory is kept on the GPU.

By passing a specific device id when constructing an `umpire::strategy::AllocationAdvisor`, you can ensure that the advice will be applied with respect to that device

```
  //
  // Create an allocator that applied "PREFFERED_LOCATION" advice to set a
  // specific GPU device as the preferred location.
  //
  // In this case, device #2.
  //
  const int device_id = 2;

  try {
    auto preferred_location_allocator =
        rm.makeAllocator<umpire::strategy::AllocationAdvisor>(
            "preferred_location_device_2", allocator, "PREFERRED_LOCATION",
            device_id);
```

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/AllocationAdvisor.hpp"
#include "umpire/util/Exception.hpp"

int main(int, char**)
{
  auto& rm = umpire::ResourceManager::getInstance();

  auto allocator = rm.getAllocator("UM");

  // _sphinx_tag_tut_device_advice_start
  //
  // Create an allocator that applied "PREFFERED_LOCATION" advice to set a
  // specific GPU device as the preferred location.
  //
  // In this case, device #2.
  //
  const int device_id = 2;

  try {
    auto preferred_location_allocator =
        rm.makeAllocator<umpire::strategy::AllocationAdvisor>(
            "preferred_location_device_2", allocator, "PREFERRED_LOCATION",
            device_id);

    // _sphinx_tag_tut_device_advice_end
    void* data = preferred_location_allocator.allocate(1024);
```

(continues on next page)

(continued from previous page)

```
    preferred_location_allocator.deallocate(data);
  } catch (umpire::util::Exception& e) {
    std::cout << "Couldn't create Allocator with device_id = " << device_id
              << std::endl;

    std::cout << e.message() << std::endl;
  }

  return 0;
}
```

4.5 Moving Host Data to Managed Memory

When using a system with NVIDIA GPUs, you may realize that some host data should be moved to unified memory in order to make it accessible by the GPU. You can do this with the `umpire::ResourceManager::move()` operation:

```
  double* um_data = static_cast<double*>(rm.move(host_data, um_allocator));
```

The move operation will copy the data from host memory to unified memory, allocated using the provided `um_allocator`. The original allocation in host memory will be deallocated. The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

int main(int, char**)
{
  constexpr std::size_t SIZE = 1024;

  auto& rm = umpire::ResourceManager::getInstance();
  auto allocator = rm.getAllocator("HOST");

  //
  // Allocate host data
  //
  double* host_data =
      static_cast<double*>(allocator.allocate(SIZE * sizeof(double)));

  //
  // Move data to unified memory
  //
  auto um_allocator = rm.getAllocator("UM");
  // _sphinx_tag_tut_move_host_to_managed_start
  double* um_data = static_cast<double*>(rm.move(host_data, um_allocator));
  // _sphinx_tag_tut_move_host_to_managed_end
```

(continues on next page)

(continued from previous page)

```
  //
  // Deallocate um_data, host_data is already deallocated by move operation.
  //
  rm.deallocate(um_data);

  return 0;
}
```

4.6 Improving DynamicPoolList Performance with a Coalesce Heuristic

As needed, the `umpire::strategy::DynamicPoolList` will continue to allocate blocks to satisfy allocation requests that cannot be satisfied by blocks currently in the pool it is managing. Under certain application-specific memory allocation patterns, fragmentation within the blocks or allocations that are for sizes greater than the size of the largest available block can cause the pool to grow too large. For example, a problematic allocation pattern is when an application makes several allocations of incrementing size where each allocation is larger than the previous block size allocated.

The `umpire::strategy::DynamicPoolList::coalesce()` method may be used to cause the `umpire::strategy::DynamicPoolList` to coalesce the releasable blocks into a single larger block. This is accomplished by: tallying the size of all blocks without allocations against them, releasing those blocks back to the memory resource, and creating a new block of the previously tallied size.

Applications may offer a heuristic function to the `umpire::strategy::DynamicPoolList` during instantiation that will return true whenever a pool reaches a specific threshold of releasable bytes (represented by completely free blocks) to the total size of the pool. The DynamicPoolList will call this heuristic function just before it returns from its `umpire::strategy::DynamicPoolList::deallocate()` method and when the function returns true, the DynamicPoolList will call the `umpire::strategy::DynamicPoolList::coalesce()` method.

The default heuristic of 100 will cause the DynamicPoolList to automatically coalesce when all of the bytes in the pool are releasable and there is more than one block in the pool.

A heuristic of 0 will cause the DynamicPoolList to never automatically coalesce.

Creation of the heuristic function is accomplished by:

```
  //
  // Create a heuristic function that will return true to the DynamicPoolList
  // object when the threshold of releasable size to total size is 75%.
  //
  auto heuristic_function =
      umpire::strategy::DynamicPoolList::percent_releasable(75);
```

The heuristic function is then provided as a parameter when the object is instantiated:

```
  //
  // Create a pool with an initial block size of 1 Kb and 1 Kb block size for
  // all subsequent allocations and with our previously created heuristic
  // function.
  //
  auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPoolList>(
      "HOST_POOL", allocator, 1024ul, 1024ul, 16, heuristic_function);
```

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/DynamicPoolList.hpp"
#include "umpire/util/Macros.hpp"
#include "umpire/util/wrap_allocator.hpp"

int main(int, char**)
{
  auto& rm = umpire::ResourceManager::getInstance();

  auto allocator = rm.getAllocator("HOST");

  // _sphinx_tag_tut_creat_heuristic_fun_start
  //
  // Create a heuristic function that will return true to the DynamicPoolList
  // object when the threshold of releasable size to total size is 75%.
  //
  auto heuristic_function =
      umpire::strategy::DynamicPoolList::percent_releasable(75);
  // _sphinx_tag_tut_creat_heuristic_fun_end

  // _sphinx_tag_tut_use_heuristic_fun_start
  //
  // Create a pool with an initial block size of 1 Kb and 1 Kb block size for
  // all subsequent allocations and with our previously created heuristic
  // function.
  //
  auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPoolList>(
      "HOST_POOL", allocator, 1024ul, 1024ul, 16, heuristic_function);
  // _sphinx_tag_tut_use_heuristic_fun_end

  //
  // Obtain a pointer to our specific DynamicPoolList instance in order to see the
  // DynamicPoolList-specific statistics
  //
  auto dynamic_pool =
      umpire::util::unwrap_allocator<umpire::strategy::DynamicPoolList>(
          pooled_allocator);

  void* a[4];
  for (int i = 0; i < 4; ++i)
    a[i] = pooled_allocator.allocate(1024);

  for (int i = 0; i < 4; ++i) {
    pooled_allocator.deallocate(a[i]);
```

(continues on next page)

(continued from previous page)

```
    std::cout << "Pool has " << pooled_allocator.getActualSize()
              << " bytes of memory. " << pooled_allocator.getCurrentSize()
              << " bytes are used. " << dynamic_pool->getBlocksInPool()
              << " blocks are in the pool. "
              << dynamic_pool->getReleasableSize() << " bytes are releaseable. "
              << std::endl;
  }

  return 0;
}
```

4.7 Move Allocations Between NUMA Nodes

When using NUMA (cache coherent or non uniform memory access) systems, there are different latencies to parts of the memory. From an application perspective, the memory looks the same, yet especially for high-performance computing it is advantageous to have finer control. *malloc()* attempts to allocate memory close to your node, but it can make no guarantees. Therefore, Linux provides both a process-level interface for setting NUMA policies with the system utility *numactl*, and a fine-grained interface with *libnuma*. These interfaces work on ranges of memory in multiples of the page size, which is the length or unit of address space loaded into a processor cache at once.

A page range may be bound to a NUMA node using the `umpire::strategy::NumaPolicy`. It can therefore also be moved between NUMA nodes using the `umpire::ResourceManager::move()` with a different allocator. The power of using such an abstraction is that the NUMA node can be associated with a device, in which case the memory is moved to, for example, GPU memory.

In this recipe we create an allocation bound to a NUMA node, and move it to another NUMA node.

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/NumaPolicy.hpp"
#include "umpire/util/Macros.hpp"
#include "umpire/util/numa.hpp"

#if defined(UMPIRE_ENABLE_CUDA)
#include <cuda_runtime_api.h>
#endif

int main(int, char**)
{
  auto& rm = umpire::ResourceManager::getInstance();
```

(continues on next page)

(continued from previous page)

```
  const std::size_t alloc_size = 5 * umpire::get_page_size();

  // Get a list of the host NUMA nodes (e.g. one per socket)
  auto host_nodes = umpire::numa::get_host_nodes();

  if (host_nodes.size() < 1) {
    UMPIRE_ERROR("No NUMA nodes detected");
  }

  // Create an allocator on the first NUMA node
  auto host_src_alloc = rm.makeAllocator<umpire::strategy::NumaPolicy>(
      "host_numa_src_alloc", rm.getAllocator("HOST"), host_nodes[0]);

  // Create an allocation on that node
  void* src_ptr = host_src_alloc.allocate(alloc_size);

  if (host_nodes.size() > 1) {
    // Create an allocator on another host NUMA node.
    auto host_dst_alloc = rm.makeAllocator<umpire::strategy::NumaPolicy>(
        "host_numa_dst_alloc", rm.getAllocator("HOST"), host_nodes[1]);

    // Move the memory
    void* dst_ptr = rm.move(src_ptr, host_dst_alloc);

    // The pointer shouldn't change even though the memory location changes
    if (dst_ptr != src_ptr) {
      UMPIRE_ERROR("Pointers should match");
    }

    // Touch it
    rm.memset(dst_ptr, 0);

    // Verify NUMA node
    if (umpire::numa::get_location(dst_ptr) != host_nodes[1]) {
      UMPIRE_ERROR("Move was unsuccessful");
    }
  }

#if defined(UMPIRE_ENABLE_DEVICE)
  // Get a list of the device nodes
  auto device_nodes = umpire::numa::get_device_nodes();

  if (device_nodes.size() > 0) {
    // Create an allocator on the first device NUMA node. Note that
    // this still requires using the "HOST" allocator. The allocations
    // are moved after the address space is reserved.
    auto device_alloc = rm.makeAllocator<umpire::strategy::NumaPolicy>(
        "device_numa_src_alloc", rm.getAllocator("HOST"), device_nodes[0]);

    // Move the memory
    void* dst_ptr = rm.move(src_ptr, device_alloc);
```

(continues on next page)

(continued from previous page)

```
    // The pointer shouldn't change even though the memory location changes
    if (dst_ptr != src_ptr) {
      UMPIRE_ERROR("Pointers should match");
    }

    // Touch it -- this currently uses the host memset operation (thus, copying
    // the memory back)
    rm.memset(dst_ptr, 0);

    // Verify NUMA node
    if (umpire::numa::get_location(dst_ptr) != device_nodes[0]) {
      UMPIRE_ERROR("Move was unsuccessful");
    }
  }
#endif

  // Clean up by deallocating from the original allocator, since the
  // allocation record is still associated with that allocator
  host_src_alloc.deallocate(src_ptr);

  return 0;
}
```

4.8 Determining the Largest Block of Available Memory in Pool

The `umpire::strategy::QuickPool` provides a `umpire::strategy::QuickPool::getLargestAvailableBlock()` that may be used to determine the size of the largest block currently available for allocation within the pool. To call this function, you must get the pointer to the `umpire::strategy::AllocationStrategy` from the `umpire::Allocator`:

```
  auto pool = rm.makeAllocator<umpire::strategy::QuickPool>(
      "pool", rm.getAllocator("HOST"));

  auto quick_pool =
      umpire::util::unwrap_allocator<umpire::strategy::QuickPool>(pool);
```

Once you have the pointer to the appropriate strategy, you can call the function:

```
  std::cout << "Largest available block in pool is "
            << quick_pool->getLargestAvailableBlock() << " bytes in size"
            << std::endl;
```

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/QuickPool.hpp"
#include "umpire/util/Exception.hpp"
#include "umpire/util/wrap_allocator.hpp"

int main(int, char**)
{
  auto& rm = umpire::ResourceManager::getInstance();

  // _sphinx_tag_tut_unwrap_start
  auto pool = rm.makeAllocator<umpire::strategy::QuickPool>(
      "pool", rm.getAllocator("HOST"));

  auto quick_pool =
      umpire::util::unwrap_allocator<umpire::strategy::QuickPool>(pool);
  // _sphinx_tag_tut_unwrap_end

  if (quick_pool == nullptr) {
    UMPIRE_ERROR(pool.getName() << " is not a QuickPool");
  }

  auto ptr = pool.allocate(1024);

  // _sphinx_tag_tut_get_info_start
  std::cout << "Largest available block in pool is "
            << quick_pool->getLargestAvailableBlock() << " bytes in size"
            << std::endl;
  // _sphinx_tag_tut_get_info_end

  pool.deallocate(ptr);

  return 0;
}
```

4.9 Coalescing Pool Memory

The `umpire::strategy::QuickPool` provides a `umpire::strategy::QuickPool::coalesce()` that can be used to release unused memory and allocate a single large block that will be able to satisfy allocations up to the previously observed high-watermark. To call this function, you must get the pointer to the `umpire::strategy::AllocationStrategy` from the `umpire::Allocator`:

```
  auto quick_pool =
      umpire::util::unwrap_allocator<umpire::strategy::QuickPool>(pool);
```

Once you have the pointer to the appropriate strategy, you can call the function:

```
    quick_pool->coalesce();
```

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/QuickPool.hpp"
#include "umpire/util/Exception.hpp"
#include "umpire/util/wrap_allocator.hpp"

int main(int, char**)
{
  auto& rm = umpire::ResourceManager::getInstance();

  auto pool = rm.makeAllocator<umpire::strategy::QuickPool>(
      "pool", rm.getAllocator("HOST"));

  // _sphinx_tag_tut_unwrap_strategy_start
  auto quick_pool =
      umpire::util::unwrap_allocator<umpire::strategy::QuickPool>(pool);
  // _sphinx_tag_tut_unwrap_strategy_end

  if (quick_pool) {
    // _sphinx_tag_tut_call_coalesce_start
    quick_pool->coalesce();
    // _sphinx_tag_tut_call_coalesce_end
  } else {
    UMPIRE_ERROR(pool.getName() << " is not a QuickPool, cannot coalesce!");
  }

  return 0;
}
```

4.10 Building a Pinned Memory Pool in FORTRAN

In this recipe, we show you how to build a pool in pinned memory using Umpire's FORTRAN API. These kinds of pools can be useful for allocating buffers to be used in communication routines in various scientific applications.

Building the pool takes two steps: 1) getting a base "PINNED" allocator, and 2) creating the pool:

```
      rm = rm%get_instance()
      base_allocator = rm%get_allocator_by_name("PINNED")

      pinned_pool = rm%make_allocator_pool("PINNED_POOL", &
                                           base_allocator, &
                                           512_8*1024_8, &
```

(continues on next page)

(continued from previous page)

```
                                        1024_8)
```

The complete example is included below:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
! project contributors. See the COPYRIGHT file for details.
!
! SPDX-License-Identifier: (MIT)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

program umpire_f_pinned_pool
      use umpire_mod
      implicit none
      logical ok

      integer(C_INT), pointer, dimension(:) :: array(:)
      type(UmpireAllocator) base_allocator
      type(UmpireAllocator) pinned_pool
      type(UmpireResourceManager) rm

      ! _sphinx_tag_tut_pinned_fortran_start
      rm = rm%get_instance()
      base_allocator = rm%get_allocator_by_name("PINNED")

      pinned_pool = rm%make_allocator_pool("PINNED_POOL", &
                                           base_allocator, &
                                           512_8*1024_8, &
                                           1024_8)
      ! _sphinx_tag_tut_pinned_fortran_end

      call pinned_pool%allocate(array, [10])
end program umpire_f_pinned_pool
```

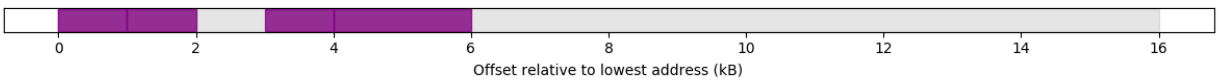
4.11 Visualizing Allocators

The python script *plot_allocations.py* is included with Umpire to plot allocations. This script uses series of three arguments: an output file with allocation records, a color, and an alpha (transparency) value *0.0-1.0*. Although these could be used to plot records from a single allocator, 3 arguments, it can also be used to overlay multiple allocators, by passing 3n arguments after the script name. In this cookbook we use this feature to visualize a pooled allocator.

The cookbook generates two files, *allocator.log* and *pooled_allocator.log*, that contain the allocation records from the underlying allocator and the pool. These can then be plotted using a command similar to the following:

```
tools/plot_allocations allocator.log gray 0.2 pooled_allocator.log purple 0.8
```

That script uses Python and Matplotlib to generate the following image



The complete example is included below:

```cpp
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <fstream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/Umpire.hpp"
#include "umpire/strategy/QuickPool.hpp"

int main(int, char**)
{
  auto& rm = umpire::ResourceManager::getInstance();

  auto allocator = rm.getAllocator("HOST");
  auto pooled_allocator = rm.makeAllocator<umpire::strategy::QuickPool>(
      "HOST_POOL", allocator, 1024 * 16);

  void* a[4];
  for (int i = 0; i < 4; ++i)
    a[i] = pooled_allocator.allocate(1024);

  // Create fragmentation
  pooled_allocator.deallocate(a[2]);
  a[2] = pooled_allocator.allocate(1024 * 2);

  // Output the records from the underlying host allocator
  {
    std::ofstream out("allocator.log");
    umpire::print_allocator_records(allocator, out);
    out.close();
  }

  // Output the records from the pooled allocator
  {
    std::ofstream out("pooled_allocator.log");
    umpire::print_allocator_records(pooled_allocator, out);
    out.close();
  }

  for (int i = 0; i < 4; ++i)
    pooled_allocator.deallocate(a[i]);

  // Visualize this using the python script. Example usage:
  // tools/analysis/plot_allocations allocator.log gray 0.2 pooled_allocator.log
  // purple 0.8

  return 0;
}
```

4.12 Mixed Pool Creation and Algorithm Basics

This recipe shows how to create a default mixed pool, and one that might be tailored to a specific application's needs. Mixed pools allocate in an array of `umpire::strategy::FixedPool` for small allocations, because these have simpler bookkeeping and are very fast, and a `umpire::strategy::QuickPool` for larger allocations.

Note: Because *DynamicPoolMap* has recently been deprecated, the implementation for *MixedPool* recently changed from using *DynamicPoolMap* to using *QuickPool* instead. Although *DynamicPoolMap* is still supported, *QuickPool* is encouraged.

The class `umpire::strategy::MixedPool` uses a generic choice of `umpire::strategy::FixedPool` of size 256 bytes to 4MB in increments of powers of 2, while `umpire::strategy::MixedPoolImpl` has template arguments that select the first, power of 2 increment, and last fixed pool size.

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/MixedPool.hpp"

int main(int, char **)
{
  auto &rm = umpire::ResourceManager::getInstance();
  auto allocator = rm.getAllocator("HOST");

  /*
   * Create a default mixed pool.
   */
  auto default_mixed_allocator = rm.makeAllocator<umpire::strategy::MixedPool>(
      "default_mixed_pool", allocator);

  UMPIRE_USE_VAR(default_mixed_allocator);

  /*
   * Create a mixed pool using fixed pool bins of size 2^8 = 256 Bytes
   * to 2^14 = 16 kB in increments of 5x, where each individual fixed
   * pool is kept under 4MB in size to begin.
   */
  auto custom_mixed_allocator = rm.makeAllocator<umpire::strategy::MixedPool>(
      "custom_mixed_pool", allocator, 256, 16 * 1024, 4 * 1024 * 1024, 5);

  /*
   * Although this calls for only 4*4=16 bytes, this allocation will
   * come from the smallest fixed pool, thus ptr will actually be the
```

(continues on next page)

(continued from previous page)

```
   * first address in a range of 256 bytes.
   */
  void *ptr1 = custom_mixed_allocator.allocate(4 * sizeof(int));

  /*
   * This is too beyond the range of the fixed pools, and therefore is
   * allocated from a dynamic pool. The range of address space
   * reserved will be exactly what was requested by the allocate()
   * method.
   */
  void *ptr2 = custom_mixed_allocator.allocate(1 << 18);

  /*
   * Clean up
   */
  custom_mixed_allocator.deallocate(ptr1);
  custom_mixed_allocator.deallocate(ptr2);

  return 0;
}
```

4.13 Thread Safe Allocator

If you want thread-safe access to allocations that come from a particular `umpire::Allocator`, you can create an instance of a *umpire::strategy::ThreadSafeAllocator* object that will synchronize access to it.

In this recipe, we look at creating a *umpire::strategy::ThreadSafeAllocator* for an *umpire::strategy::QuickPool* object:

```
  auto& rm = umpire::ResourceManager::getInstance();

  auto pool = rm.makeAllocator<umpire::strategy::QuickPool>(
      "pool", rm.getAllocator("HOST"));

  auto thread_safe_pool =
      rm.makeAllocator<umpire::strategy::ThreadSafeAllocator>(
          "thread_safe_pool", pool);
```

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/QuickPool.hpp"
#include "umpire/strategy/ThreadSafeAllocator.hpp"

int main(int, char**)
```

(continues on next page)

(continued from previous page)

```
{
  // _sphinx_tag_tut_thread_safe_start
  auto& rm = umpire::ResourceManager::getInstance();

  auto pool = rm.makeAllocator<umpire::strategy::QuickPool>(
      "pool", rm.getAllocator("HOST"));

  auto thread_safe_pool =
      rm.makeAllocator<umpire::strategy::ThreadSafeAllocator>(
          "thread_safe_pool", pool);
  // _sphinx_tag_tut_thread_safe_end

  auto allocation = thread_safe_pool.allocate(256);
  thread_safe_pool.deallocate(allocation);

  return 0;
}
```

4.14 Using File System Allocator (FILE)

Umpire supports the use of file based memory allocation. When `ENABLE_FILE_RESOURCE` is enabled, the environment variables `UMPIRE_MEMORY_FILE_DIR` can be used to determine where memory can be allocated from:

Variable	Default	Description
UMPIRE_MEMORY_FILE_DIR	./	Directory to create and allocate file based allocations

Requesting the allocation takes two steps: 1) getting a "FILE" allocator, 2) requesting the amount of memory to allocate.

```
  auto& rm = umpire::ResourceManager::getInstance();
  umpire::Allocator alloc = rm.getAllocator("FILE");

  std::size_t* A = (std::size_t*)alloc.allocate(sizeof(size_t));
```

To deallocate:

```
  alloc.deallocate(A);
```

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

int main(int, char** argv)
```

(continues on next page)

(continued from previous page)

```
{
  // _sphinx_tag_tut_file_allocate_start
  auto& rm = umpire::ResourceManager::getInstance();
  umpire::Allocator alloc = rm.getAllocator("FILE");

  std::size_t* A = (std::size_t*)alloc.allocate(sizeof(size_t));
  // _sphinx_tag_tut_file_allocate_end

  // _sphinx_tag_tut_file_deallocate_start
  alloc.deallocate(A);
  // _sphinx_tag_tut_file_deallocate_end

  return 0;
}
```

4.15 Using Burst Buffers On Lassen

On Lassen, 1) Download the latest version of Umpire 2) request a private node to build:

```
$ git clone --recursive https://github.com/LLNL/Umpire.git
$ lalloc 1 -stage storage=64
```

Note that `-stage storage=64` is needed in order to work with the Burst Buffers. 3) Additionally, the environment variable needs to set to `$BBPATH` :

```
$ export UMPIRE_MEMORY_FILE_DIR=$BBPATH/
```

4.15.1 Running File Resource Benchmarks

Continue building Umpire on 1 node, and set the `-DENABLE_FILE_RESOURCE=On` :

```
$ mkdir build && cd build
$ lrun -n 1 cmake -DENABLE_FILE_RESOURCE=On -DENABLE_OPENMP=On ../ && make
```

To run the built-in benchmarks in Umpire from the build run:

```
$ lrun -n 1 --threads=** ./bin/file_resource_benchmarks ##
```

** is a placeholder for the amount of threads wanted to run the benchmark on. ## stands for the number of array elements wanted to be passed through the benchmark. This number can range from 1-100,000,000,000.

Results should appear like:

```
Array Size:   1        Memory Size: 8e-06 MB
Total Arrays: 3               Total Memory Size: 2.4e-05 MB

HOST
  Initialization:      0.0247461 MB/sec
  Initialization Time: 0.000969849 sec
  ---------------------------------------
```

(continues on next page)

(continued from previous page)

```
  Copy:                0.890918 MB/sec
  Copy Time:           1.7959e-05 sec
  ---------------------------------------
  Scale:               0.928074 MB/sec
  Scale Time:          1.724e-05 sec
  ---------------------------------------
  Add:                 1.321 MB/sec
  Add Time:            1.8168e-05 sec
  ---------------------------------------
  Triad:               1.24102 MB/sec
  Triad Time:          1.9339e-05 sec
  ---------------------------------------
  Total Time:          0.00104323 sec

FILE
  Initialization:      0.210659 MB/sec
  Initialization Time: 0.000113928 sec
  ---------------------------------------
  Copy:                0.84091 MB/sec
  Copy Time:           1.9027e-05 sec
  ---------------------------------------
  Scale:               0.938086 MB/sec
  Scale Time:          1.7056e-05 sec
  ---------------------------------------
  Add:                 1.28542 MB/sec
  Add Time:            1.8671e-05 sec
  ---------------------------------------
  Triad:               1.54689 MB/sec
  Triad Time:          1.5515e-05 sec
  ---------------------------------------
  Total Time:          0.000184726 sec
```

This benchmark run similar to the STREAM Benchmark test and can also run a benchmark for the additional allocators like `UM` for CUDA and `DEVICE` for HIP.

4.16 Getting the Strategy Name

Since every Allocator is represented by the same type after it's been created, it can be difficult to determine exactly what kind of strategy the allocator is using. The name of the strategy can be accessed using the *Allocator::getStrategyName()* method:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
```

(continues on next page)

(continued from previous page)

```cpp
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/QuickPool.hpp"

int main(int, char**)
{
  auto& rm = umpire::ResourceManager::getInstance();
  auto allocator = rm.getAllocator("HOST");

  // _sphinx_tag_tut_strategy_name_start
  //
  auto pool = rm.makeAllocator<umpire::strategy::QuickPool>(
      "POOL", allocator);
  std::cout << pool.getStrategyName() << std::endl;
  // _sphinx_tag_tut_strategy_name_end

  UMPIRE_USE_VAR(pool);

  return 0;
}
```

CHAPTER

FIVE

FEATURES

5.1 Allocators

Allocators are the fundamental object used to allocate and deallocate memory using Umpire.

5.2 Allocator Accessibility

The Umpire library provides a variety of `umpire::resource::MemoryResource` s which can be used to create `umpire::Allocator` s depending on what's available on your system. The resources are explained more on the Resources page.

Additionally, the platforms that Umpire supports is defined by the CAMP library. This means that there is also a selection of platforms for which an allocator can be associated with as well. For example, an Allocator created with the pinned memory resource can be used with the host, cuda, hip, or sycl platforms.

Because of these options, it can be difficult to trace not only which memory resource an allocator has been created with but also which allocators can be accessed by which platforms. Umpire has the memory resource trait, `resource_type`, to provide the ability to query which memory resource is associated with a particular allocator (See example here).

Additionally, Umpire has a function, `is_accessible(Platform p, Allocator a)`, that determines if a particular allocator is accessible by a particular platform (See example here). The `allocator_accessibility.cpp` test checks what platforms are available and confirms that all memory resources which should be accessible to that platform can actually be accessed and used.

For example, if a `umpire::Allocator`, `alloc`, is created with the host memory resource and we want to know if it should be accessible from the `omp_target` CAMP platform, then we can use the `is_accessible(Platform::omp_target, alloc)` function and find that it should be accessible. The `allocator_access.cpp` file demonstrates this functionality for the *host* platform specifically.

5.2.1 Allocator Inaccessibility Configuration

On a different note, for those allocators that are deemed inaccessible, it may be useful to double check or confirm that the allocator can in fact NOT access memory on that given platform. In this case, the cmake flag, `ENABLE_INACCESSIBILITY_TESTS`, will need to be turned on.

5.2.2 Build and Run Configuration

To build and run these files, either use uberenv or the appropriate cmake flags for the desired platform and then run `ctest -T test -R allocator_accessibility_tests --output-on-failure` for the test code and `./bin/alloc_access` for the example code.

Note: The Developer's Guide shows how to configure Umpire with uberenv to build with different CAMP platforms.

Below, the `allocator_access.cpp` code is shown to demonstrate how this functionality can be used during development.

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>
#include <string>
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/Umpire.hpp"

bool is_accessible_from_host(umpire::Allocator a)
{
  if(umpire::is_accessible(umpire::Platform::host, a)) {
    std::cout << "The allocator, " << a.getName()
              << ", is accessible." << std::endl;
    return true;
  } else {
    std::cout << "The allocator, " << a.getName()
              << ", is _not_ accessible." << std::endl << std::endl;
    return false;
  }
}

//////////////////////////////////////////////////////////////////////////////////////
//Depending on how Umpire has been set up, several different allocators could be␣
↪accessible
//from the host CAMP platform. This example will create a list of all currently␣
↪available
//allocators and then determine whether each can be accessed from the host platform.
//(To test other platforms, see allocator accessibility test.)
//////////////////////////////////////////////////////////////////////////////////////
int main()
{
  auto& rm = umpire::ResourceManager::getInstance();

  std::vector<std::string> allNames = rm.getResourceNames();
  std::vector<umpire::Allocator> alloc;

  ////////////////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```
  //Create an allocator for each available type
  ////////////////////////////////////////////////
  std::cout << "Available allocators: ";
  for(auto a : allNames) {
    if (a.find("::") == std::string::npos) {
      alloc.push_back(rm.getAllocator(a));
      std::cout << a << " ";
    }
  }
  std::cout << std::endl;

  ////////////////////////////////////////////////
  //Test accessibility
  ////////////////////////////////////////////////
  std::cout << "Testing the available allocators for accessibility from the CAMP host␣
↪platform:" << std::endl;
  const int size = 100;
  for(auto a : alloc) {
    if(is_accessible_from_host(a)) {
      int* data = static_cast<int*>(a.allocate(size*sizeof(int)));
      for(int i = 0; i < size; i++) {
        data[i] = i * i;
      }
      UMPIRE_ASSERT(data[size-1] == (size-1) * (size-1) && "Inequality found in array␣
↪that should be accessible");
    }
  }

  return 0;
}
```

5.3 Backtrace

The Umpire library may be configured to provide using programs with backtrace information as part of Umpire thrown exception description strings.

Umpire may also be configured to collect and provide backtrace information for each Umpire provided memory allocation performed.

5.3.1 Build Configuration

Backtrace is enabled in Umpire builds with the following:

- **``cmake … -DENABLE_BACKTRACE=On …``** to backtrace capability in Umpire.
- **``cmake -DENABLE_BACKTRACE=On -DENABLE_BACKTRACE_SYMBOLS=On …``** to enable Umpire to display symbol information with backtrace. **Note:** Using programs will need to add the `-rdyanmic` and `-ldl` linker flags in order to properly link with this configuration of the Umpire library.

5.3.2 Runtime Configuration

For versions of the Umpire library that are backtrace enabled (from flags above), the user may expect the following.

Backtrace information will always be provided in the description strings of umpire generated exception throws.

Setting the environment variable `UMPIRE_BACKTRACE=On` will cause Umpire to record backtrace information for each memory allocation it provides.

Setting the environment variable `UMPIRE_LOG_LEVEL=Error` will cause to Umpire to log backtrace information for each of the leaked Umpire allocations found during application exit.

A programatic interface is also availble via the func::*umpire::print_allocator_records* free function.

An example for checking and displaying the information this information logged above may be found here:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>
#include <sstream>

#include "umpire/ResourceManager.hpp"
#include "umpire/Umpire.hpp"
#include "umpire/strategy/QuickPool.hpp"

int main(int, char**)
{
  auto& rm = umpire::ResourceManager::getInstance();
  auto allocator = rm.getAllocator("HOST");
  auto pool_allocator = rm.makeAllocator<umpire::strategy::QuickPool>(
      "host_quick_pool", allocator);

  allocator.allocate(16);
  allocator.allocate(32);
  allocator.allocate(64);

  pool_allocator.allocate(128);
  pool_allocator.allocate(256);
  pool_allocator.allocate(512);

  std::stringstream ss;
  umpire::print_allocator_records(allocator, ss);
  umpire::print_allocator_records(pool_allocator, ss);

  // Example #1 of 3 - Leaked allocations
  //
  // If Umpire compiled with -DENABLE_BACKTRACE=On, then backtrace
  // information will be printed for each of the allocations made above.
  //
  // Otherwise, if Umpire was not compiled with -DENABLE_BACKTRACE=On,
  // then only the addresses and size information for each allocation will be
  // printed.
```

(continues on next page)

(continued from previous page)

```
  //
  if (!ss.str().empty())
    std::cout << ss.str();

  // Example #2 of 3 - Umpire error exceptions
  //
  // When umpire throws an exception, a backtrace to the offending call will
  // be provided in the exception string.
  //
  void* bad_ptr = (void*)0xBADBADBAD;

  try {
    allocator.deallocate(bad_ptr); // Will cause a throw from umpire
  } catch (const std::exception& exc) {
    //
    // exc.what() string will also contain a backtrace
    //
    std::cout << "Exception thrown from Umpire:" << std::endl << exc.what();
  }

  // Example #3 of 3 - Leak detection
  //
  // When the program terminates, Umpire's resource manager will be
  // deconstructed.  During deconstruction, Umpire will log the size and
  // address, of each leaked allocation in each allocator.
  //
  // If Umpire was compiled with -DENABLE_BACKTRACE=On, backtrace
  // information will also be logged for each leaked allocation in each
  // allocator.
  //
  // To enable (and see) the umpire logs, set the environment variable
  // UMPIRE_LOG_LEVEL=Error.
  //
  return 0;
}
```

5.4 File I/O

Umpire provides support for writing files containing log and replay data, rather than directing this output to stdout. When logging or replay are enabled, the following environment variables can be used to determine where the output is written:

> `UMPIRE_OUTPUT_DIR` . Directory to write log and replay files `UMPIRE_OUTPUT_BASENAME` umpire Basename of logging and relpay files

The values of these variables are used to construct unique filenames for output. The extension `.log` is used for logging output, and `.replay` for replay output. The filenames additionally contain three integers, one corresponding to the rank of the process, one corresponding to the process ID, and one that is used to make multiple files with the same basename and rank unique. This ensures that multiple runs with the same IO configuration do not overwrite files.

The format of the filenames is:

```
<UMPIRE_OUTPUT_BASENAME>.<RANK>.<PID>.<UID>.<log|replay>
```

If Umpire is compiled without MPI support, then rank will always be 0.

5.5 Logging and Replay of Umpire Events

5.5.1 Logging

When debugging memory operation problems, it is sometimes helpful to enable Umpire's logging facility. The logging functionality is enabled for default builds unless -DENABLE_LOGGING='Off' has been specified in which case it is disabled.

If Umpire logging is enabled, it may be controlled by setting the `UMPIRE_LOG_LEVEL` environment variable to `Error`, `Warning`, `Info`, or `Debug`. The `Debug` value is the most verbose.

When `UMPIRE_LOG_LEVEL` has been set, events will be logged to the standard output.

5.5.2 Replay

Umpire provides a lightweight replay capability that can be used to investigate performance of particular allocation patterns and reproduce bugs. By running an executable that uses Umpire with the environment variable `UMPIRE_REPLAY` set to `On`, Umpire will emit information for the following Umpire events:

- **version** `umpire::get_major_version()`, `umpire::get_minor_version()`, and `umpire::get_patch_version()`
- **makeMemoryResource** `umpire::resource::MemoryResourceRegistry::makeMemoryResource()`
- **makeAllocator** `umpire::ResourceManager::makeAllocator()`
- **allocate** `umpire::Allocator::allocate()`
- **deallocate** `umpire::Allocator::deallocate()`

5.5.3 Running with Replay

To enable Umpire replay, one may execute as follows:

```
UMPIRE_REPLAY="On" ./my_umpire_using_program > replay_log.json
```

will write Umpire replay events to the file `replay_log.json` that will contain the following kinds of information:

5.5.4 Interpretting Results - Version Event

The first event captured is the **version** event which shows the version information as follows:

```
{ "kind":"replay", "uid":27494, "timestamp":1558388052211435757, "event": "version",
↪"payload": { "major":0, "minor":3, "patch":3 } }
```

Each line contains the following set of common elements:

kind Always set to `replay`

uid This is the MPI rank of the process generating the event for mpi programs or the PID for non-mpi.

timestamp Set to the time when the event occurred.

event Set to one of: `version`, `makeMemoryResource`, `makeAllocator`, `allocate`, or `deallocate`

payload Optional and varies upon event type

result Optional and varies upon event type

As can be seen, the *major*, *minor*, and *patch* version numbers are captured within the *payload* for this event.

5.5.5 makeMemoryResource Event

Next you will see events for the creation of the default memory resources provided by Umpire with the **makeMemoryResource** event:

```
{ "kind":"replay", "uid":27494, "timestamp":1558388052211477678, "event":
↪"makeMemoryResource", "payload": { "name": "HOST" }, "result": "0x101626b0" }
{ "kind":"replay", "uid":27494, "timestamp":1558388052471684134, "event":
↪"makeMemoryResource", "payload": { "name": "DEVICE" }, "result": "0x101d79a0" }
{ "kind":"replay", "uid":27494, "timestamp":1558388052471698804, "event":
↪"makeMemoryResource", "payload": { "name": "PINNED" }, "result": "0x101d7a50" }
{ "kind":"replay", "uid":27494, "timestamp":1558388052472972935, "event":
↪"makeMemoryResource", "payload": { "name": "UM" }, "result": "0x101d7b00" }
{ "kind":"replay", "uid":27494, "timestamp":1558388052595814979, "event":
↪"makeMemoryResource", "payload": { "name": "DEVICE_CONST" }, "result": "0x101d7bb0" }
```

The *payload* shows that a memory resource was created for *HOST*, *DEVICE*, *PINNED*, *UM*, and *DEVICE_CONST* respectively. Note that this could also be done with the *FILE* memory resource. The *result* is a reference to the object that was created within Umpire for that resource.

5.5.6 makeAllocator Event

The **makeAllocator** event occurs whenever a new allocator instance is being created. Each call to *makeAllocator* will generate a pair of JSON lines. The first line will show the intent of the call and the second line will show both the intent and the result. This is because the makeAllocator call can fail and keeping both the intent and result allows us to reproduce this failure later.

`umpire::Allocator`:

```
{ "kind":"replay", "uid":27494, "timestamp":1558388052595864262, "event": "makeAllocator
↪", "payload": { "type":"umpire::strategy::QuickPool", "with_introspection":true,
↪"allocator_name":"pool", "args": [ "HOST" ] } }
{ "kind":"replay", "uid":27494, "timestamp":1558388052595903505, "event": "makeAllocator
↪", "payload": { "type":"umpire::strategy::QuickPool", "with_introspection":true,
↪"allocator_name":"pool", "args": [ "HOST" ] }, "result": { "allocator_ref":"0x108a8730
↪" } }
```

The *payload* shows how the allocator was constructed. The *result* shows the reference to the allocated object.

5.5.7 allocate Event

Like the **makeAllocator** event, the **allocate** event is captured as an intention/result pair so that an error may be replayed in the event that there is an allocation failure.

```
{ "kind":"replay", "uid":27494, "timestamp":1558388052595911583, "event": "allocate",
↪"payload": { "allocator_ref": "0x108a8730", "size": 0 } }
{ "kind":"replay", "uid":27494, "timestamp":1558388052595934822, "event": "allocate",
↪"payload": { "allocator_ref": "0x108a8730", "size": 0 }, "result": { "memory_ptr":
↪"0x200040000010" } }
```

The *payload* shows the object reference of the allocator and the size of the allocation request. The *result* shows the pointer to the memory allocated.

5.5.8 deallocate Event

```
{ "kind":"replay", "uid":27494, "timestamp":1558388052596358577, "event": "deallocate",
↪"payload": { "allocator_ref": "0x108a8730", "memory_ptr": "0x200040000010" } }
```

The *payload* shows the reference to the allocator object and the pointer to the allocated memory that is to be freed.

5.5.9 Replaying the session

Loading this file with the `replay` program will replay this sequence of `umpire::Allocator` creation, allocations, and deallocations:

```
./bin/replay -i replay_log.json
```

5.6 Operations

Operations provide an abstract interface to modifying and moving data between Umpire :class:`umpire::Allocator`s.

5.6.1 Provided Operations

5.7 Strategies

Strategies are used in Umpire to allow custom algorithms to be applied when allocating memory. These strategies can do anything, from providing different pooling methods to speed up allocations to applying different operations to every alloctaion. Strategies can be composed to combine their functionality, allowing flexible and reusable implementations of different components.

5.7.1 Provided Strategies

CHAPTER

SIX

CONTRIBUTION GUIDE

This document is intented for developers who want to add new features or bugfixes to Umpire. It assumes you have some familiarity with git and GitHub. It will discuss what a good pull request (PR) looks like, and the tests that your PR must pass before it can be merged into Umpire.

6.1 Forking Umpire

If you aren't an Umpire deveolper at LLNL, then you won't have permission to push new branches to the repository. First, you should create a fork. This will create a copy of the Umpire repository that you own, and will ensure you can push your changes up to GitHub and create pull requests.

6.1.1 Developing a New Feature

New features should be based on the `develop` branch. When you want to create a new feature, first ensure you have an up-to-date copy of the `develop` branch:

```
$ git checkout develop
$ git pull origin develop
```

You can now create a new branch to develop your feature on:

```
$ git checkout -b feature/<name-of-feature>
```

Proceed to develop your feature on this branch, and add tests that will exercise your new code. If you are creating new methods or classes, please add Doxygen documentation.

Once your feature is complete and your tests are passing, you can push your branch to GitHub and create a PR.

6.1.2 Developing a Bug Fix

First, check if the change you want to make has been fixed in `develop`. If so, we suggest you either start using the `develop` branch, or temporarily apply the fix to whichever version of Umpire you are using.

If the bug is still unfixed, first make sure you have an up-to-date copy of the develop branch:

```
$ git checkout develop
$ git pull origin develop
```

Then create a new branch for your bugfix:

```
$ git checkout -b bugfix/<name-of-bug>
```

First, add a test that reproduces the bug you have found. Then develop your bugfix as normal, and ensure to `make test` to check your changes actually fix the bug.

Once you are finished, you can push your branch to GitHub, then create a PR.

6.1.3 Creating a Pull Request

You can create a new PR here. Ensure that your PR base is the `develop` branch of Umpire.

Add a descriptive title explaining the bug you fixed or the feature you have added, and put a longer description of the changes you have made in the comment box.

Once your PR has been created, it will be run through our automated tests and also be reviewed by Umpire team members. Providing the branch passes both the tests and reviews, it will be merged into Umpire.

6.1.4 Tests

Umpire uses Bamboo and Gitlab for continuous integration tests. Our tests are automatically run against every new pull request, and passing all tests is a requirement for merging your PR. If you are developing a bugfix or a new feature, please add a test that checks the correctness of your new code. Umpire is used on a wide variety of systems with a number of configurations, and adding new tests helps ensure that all features work as expected across these environments.

Umpire's tests are all in the `test` directory and are split up by component.

CHAPTER

SEVEN

DEVELOPER GUIDE

This section provides documentation on Umpire's Continuous Integration, how to configure Umpire using Uberenv, and the process for analyzing Umpire applications using HPCToolKit.

7.1 Continuous Integration

7.1.1 Gitlab CI

Umpire uses continuous integration to ensure that changes added to the repository are well integrated and tested for compatability with the rest of the existing code base. Our CI tests incude a variety of vetted configurations that run on different LC machines.

Umpire shares its Gitlab CI workflow with other projects. The documentation is therefore shared.

7.2 Uberenv

Umpire shares its Uberenv workflow with other projects. The documentation is therefore shared.

This page will provides some Umpire specific examples to illustrate the workflow described in the documentation.

7.2.1 Before to start

First of all, it is worth noting that Umpire does not have dependencies, except for CMake, which is most of the time installed externally.

That does not make the workflow useless: Uberenv will drive Spack which will generate a host-config file with the toolchain (including cuda if activated) and the options or variants pre-configured.

Machine specific configuration

```
$ ls -c1 scripts/uberenv/spack_configs
blueos_3_ppc64le_ib
darwin
toss_3_x86_64_ib
blueos_3_ppc64le_ib_p9
config.yaml
```

Umpire has been configured for `toss_3_x86_64_ib` and other systems.

Vetted specs

```
$ ls -c1 .gitlab/*jobs.yml
.gitlab/lassen-jobs.yml
.gitlab/ruby-jobs.yml
```

CI contains jobs for ruby.

```
$ git grep -h "SPEC" .gitlab/ruby-jobs.yml | grep "gcc"
    SPEC: "%gcc@4.9.3"
    SPEC: "%gcc@6.1.0"
    SPEC: "%gcc@7.1.0"
    SPEC: "%gcc@7.3.0"
    SPEC: "%gcc@8.1.0"
```

We now have a list of the specs vetted on `ruby/toss_3_x86_64_ib`.

Note: In practice, one should check if the job is not *allowed to fail*, or even deactivated.

MacOS case

In Umpire, the Spack configuration for MacOS contains the default compilers depending on the OS version (*compilers.yaml*), and a commented section to illustrate how to add *CMake* as an external package. You may install CMake with homebrew, for example.

7.2.2 Using Uberenv to generate the host-config file

We have seen that we can safely use *gcc@8.1.0* on ruby. Let us ask for the default configuration first, and then produce static libs, have OpenMP support and run the benchmarks:

```
$ python scripts/uberenv/uberenv.py --spec="%gcc@8.1.0"
$ python scripts/uberenv/uberenv.py --spec="%gcc@8.1.0~shared+openmp tests=benchmarks"
```

Each will generate a CMake cache file, e.g.:

```
hc-ruby-toss_3_x86_64_ib-gcc@8.1.0-fjcjwd6ec3uen5rh6msdqujydsj74ubf.cmake
```

7.2.3 Using host-config files to build Umpire

```
$ mkdir build && cd build
$ cmake -C <path_to>/<host-config>.cmake ..
$ cmake --build -j .
$ ctest --output-on-failure -T test
```

It is also possible to use this configuration with the CI script outside of CI:

```
$ HOST_CONFIG=<path_to>/<host-config>.cmake scripts/gitlab/build_and_test.sh
```

7.2.4 Using Uberenv to configure and run Leak Sanitizer

During development, it may be beneficial to regularly check for memory leaks. This will help avoid the possibility of having many memory leaks showing up all at once during the CI tests later on. The Leak Sanitizer can easily be configured from the root directory with:

```
$ srun -ppdebug -N1 --exclusive python scripts/uberenv/uberenv.py --spec="%clang@9.0.0␣
↪cxxflags=-fsanitize=address"
$ cd build
$ cmake -C <path_to>/hc-ruby-toss_3_x86_64_ib-clang@9.0.0.cmake ..
$ cmake --build -j
$ ASAN_OPTIONS=detect_leaks=1 make test
```

Note: The host config file (i.e., `hc-ruby-...cmake`) can be reused in order to rebuild with the same configuration if needed.

This will configure a build with Clang 9.0.0 and the Leak Sanitizer. If there is a leak in one of the tests, it can be useful to gather more information about what happened and more details about where it happened. One way to do this is to run:

```
$ ASAN_OPTIONS=detect_leaks=1 ctest -T test --output-on-failure
```

Additionally, the Leak Sanitizer can be run on one specific test (in this example, the "replay" tests) with:

```
$ ASAN_OPTIONS=detect_leaks=1 ctest -T test -R replay --output-on-failure
```

Depending on the output given when running the test with the Leak Sanitizer, it may be useful to use `addr2line -e <./path_to/executable> <address_of_leak>` to see the exact line the output is referring to.

7.3 HPCToolKit

This page will describes the process and series of steps to analyze Umpire specific applications with HPCToolKit.

7.3.1 Using HPCToolKit

LLNL's documentation for using HPCToolKit for general analysis is a great starting resource and can be found here. The HPCToolKit manual can be found here.

The LC machines have `hpctoolkit` installed as a module which can be loaded with `module load hpctoolkit`. The rest of this page will describe the steps for specific analysis examples with Umpire.

Getting Started

Below is the basic (Umpire-specific) set up to load, build with, and run with HPCToolKit:

```
$ ssh lassen
$ module load hpctoolkit
$ cmake -DENABLE_CUDA=On -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_FLAGS="-O3 -g" -DCMAKE_C_
↪FLAGS="-O3 -g"
$ make -j
$ hpcrun -e CPUTIME ./bin/executable
$ hpcstruct ./bin/executable
$ hpcprof -S executable.hpcstruct hpctoolkit-executable-measurements-<job_id>/
```

Note: The HPCToolKit manual recommends building with a fully optimized version of the executable (hence, the added flags in the `cmake` command).

Note: The `hpcrun` command can measure certain "events". Those events are added with a `-e` argument and include things like `CPUTIME`, `gpu=nvidia`, `gpu=nvidia,pc`, `IO`, `MEMLEAK`, etc. A full list of the possible events can be found by running `hpcrun -L`.

After running the `hpcrun` command, a `hpctoolkit-executable-measurements-<job_id>/` folder will be generated. After running the `hpcstruct` command, a `executable.hpcstruct` file will be generated. These two generated items will then become input used with `hpcprof`, as shown above. If the file that you are analyzing is large or is using a lot of resources, then `hpcprof-mpi` could be better to use. The `hpcprof-mpi` command looks the same otherwise. The result of either `hpcprof` command is a generated "database" folder.

At this point, we need the HPCViewer program to view the resulting database. The easiest way to engage with the HPCViewer is to do it locally. Therefore, we can tar up the generated database folder and use `scp` to send it to a local machine. For example:

```
$ tar -czvf database.tar hpctoolkit-executable-database-<job_id>/
$ scp username@lassen.llnl.gov:path/to/database.tar .
$ tar -xzvf database.tar
```

From here, you can open the HPCViewer and select the untarred database folder we just sent over to be viewed. More information on how to use HPCViewer will be provided in the next section.

Otherwise, using HPCViewer from the command line can be tricky since we need X11 forwarding. In order to have X11 forwarding available, we have to ssh into the LC machine and compute node a little differently:

```
$ ssh -YC lassen
$ bsub -XF -W 60 -nnodes 1 -Is /bin/bash
```

Note: If that doesn't work, you can also try ssh'ing into the LC machine with `ssh -X -S none lassen`.

From here we can run the same steps as before (listed at the top of this section). When we have generated the database folder, we will just call the hpcviewer program with `hpcviewer hpctoolkit-executable-database-<job_id>/`. LLNL's documentation for HPCToolKit also provides an example command to use the `hpctraceviewer` tool.

Using HPCViewer

Once you have your own version of HPCViewer locally, it is very easy to launch and open up the database folder generated earlier. You can do this with just `./hpcviewer` and selecting the right database folder.

For our use cases, we mostly used the "Hot Child" feature, but that is by no means the most valuable or most important feature that HPCViewer offers. To learn more about what HPCViewer can do, the instruction manual is here.

Note: Depending on what's available on your local machine, you may have to download or update Java in order to run hpcviewer. There are instructions here for hpcviewer. You can get Java 8 from here.

Running with Hatchet

Hatchet is a tool that can better analyze performance metrics given from a variety of tools, including HPCToolKit. Using Hatchet to analyze the output from HPCToolKit can help visualize the performance of different parts of the same program.

To use Hatchet, we create a HPCToolKit analysis, just as before, but this time there is a specialized `hpcprof-mpi` command needed when generating the database folder. Below is an example:

```
$ module load hpctoolkit
$ cmake -DENABLE_CUDA=On -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_FLAGS="-O3 -g" -DCMAKE_C_
↪FLAGS="-O3 -g"
$ make -j
$ hpcrun -e CPUTIME ./bin/executable
$ hpcstruct ./bin/executable
$ hpcprof-mpi --metric-db yes -S executable.hpcstruct hpctoolkit-executable-measurements-
↪<job_id>/
```

The flag, `--metric-db yes`, is an optional argument to `hpcprof-mpi` that allows Hatchet to better interpret information given from HPCToolKit. Without it, it will be very hard to get Hatchet to understand the HPCToolKit output.

We've now generated a HPCToolKit database folder which Hatchet can read. Now we need to launch Hatchet and get started with some analysis. Below is a Python3 interpreter mode example:

```
$ python3 #start the python interpreter
$ import hatchet as ht #import hatchet
$ dirname = "hpctoolkit-executable-database-<job_id>" #set var to hpctoolkit database
$ gf = ht.GraphFrame.from_hpctoolkit(dirname) #set up the graphframe for hatchet that␣
↪uses database

$ print(gf.tree(depth=3)) #This is to check briefly that I recognize my tree by checking␣
↪the root node + a couple sub-nodes
$ print(len(gf.graph)) #I can also verify the tree by checking the length of the␣
↪graphframe
$ print(gf.dataframe.shape) #I can also print out the 'shape' of the tree (depth x column_
↪metrics)
$ print(list(gf.dataframe.columns)) #I can print out all the column_metrics (e.g. "time",
↪ "nid", etc.)
$ print(gf.dataframe.index.names) #I can also print the node names (may be kind of␣
↪confusing unless you know what you're looking for)

$ query1 = [{"name": "119:same_order\(umpire::Allocator\)"}, "*"] #Set up a query method␣
↪to filter for the "same_order" sub tree
```

(continues on next page)

(continued from previous page)

```
$ filtered_gf = gf.filter(query1) #apply the query method as a filter on the original␣
↪tree
$ print(len(filtered_gf.graph)) #verifying that I now have a subtree (length will be␣
↪smaller)
$ print(filtered_gf.tree(metric_column="time (inc)")) #printing the new filtered subtree␣
↪by inclusive time metric
$ print(filtered_gf.tree()) #printing the whole filtered tree as is

$ query2 = [{"name": "120:reverse_order\(umpire::Allocator\)"}, "*"] #Set up a query␣
↪method to filter for the "reverse_order" sub tree
$ filtered_gf_rev = gf.filter(query2) #apply the query method as a filter on the␣
↪original tree
$ print(len(filtered_gf_rev.graph)) #verifying that I now have a subtree (length will be␣
↪smaller)
$ print(filtered_gf_rev.tree(metric_column = "time (inc)")) #printing the new filtered␣
↪subtree by inclusive time metric

$ filtered_gf.drop_index_levels() #As-is, the tree will include info for ranks - if that␣
↪isn't needed, this function drops that info
$ filtered_gf.dataframe #this provides a spreadsheet of the data that is populating the␣
↪graphframe (what the tree shows)
$ filtered_gf.dataframe.iloc[0] #gives the first entry of the spreadsheet, here that is␣
↪the root node of the filtered tree
$ filtered_gf.dataframe.iloc[0,0] #gives the first part of the first entry of the␣
↪spreadsheet (here, it's the inclusive time)

$ gf3 = filtered_gf - filtered_gf_rev #Stores the diff between two (comparable) trees in␣
↪gf3
$ print(gf3.tree()) #prints the diff tree
$ gf3.dataframe #outputs the spreadsheet of data that populates the diff tree
```

This example was set up to analyze the performance of the `no-op_stress_test.cpp` benchmark file from the Umpire repo. It compares the performance from one part of the program (i.e., the part that measure the performance when doing deallocations in the "same order" as they were allocated) versus another part of the same program (i.e., the part that measures the performance when doing deallocations in the "reverse order" as they were allocated).

In Hatchet, these two parts show up as subtrees within the entire call path tree of my example program. Therefore, I can compare one subtree to another in terms of performance (in my case, I compared in terms of inclusive time).

7.3.2 Analyzing results

After opening up a database folder in HPCViewer or analyzing the call paths in Hatchet, we can compare the performance (or whatever measurement we are looking at) of different parts of the program against other parts and try to find performance trends. In Umpire's use case, we plan to use Hatchet as part of our CI to find out if integrating a new commit into our repository increases the performance by a certain threshold or more. If so, our CI test will fail. Our process looks something like:

- Grab the example program's database from the develop branch
- Grab the example program's database from a different branch I want to compare against
- Create a graphframe for each database
- Create a filtered graphframe for each that focuses on the specific part of the program I want to measure against

- Compare the inclusive time for each filtered graphframe (or whatever metric I want to analyze)
- If the metric (e.g., inclusive time) of the new branch's filtered graphframe is more than `threshold` more than that of develop's, then fail the test!