Managing Heterogeneous Device Memory using C++17 Memory Resources

Attila Krasznahorkay\textsuperscript{1}  Stephen Nicholas Swatman\textsuperscript{1,2}  Paul Gessinger-Befurt\textsuperscript{1}

29 November 2021

\textsuperscript{1}CERN  \textsuperscript{2}University of Amsterdam
Introduction

• Writing heterogeneous code is difficult! Traditional CPU programming skills do not always carry over.
  • Often requires fundamentally different algorithm design.
  • Many considerations that are not necessary in “traditional” CPU programming.
• Heterogeneous platforms, by and large, use a memory management paradigm that is not compatible with what we teach domain scientists.
  • Heterogeneous memory management is often highly explicit.
  • Heterogeneous memory management is often slow!
• Together with much longer allocation times, this creates a mine field of bugs (functional and non-functional).
In this talk: vecmem, a C++ library that brings performance and ergonomics to heterogeneous memory.

We allow users to use device memory...

- ...with the same easy-to-use semantics as STL containers;
- ...with the safety of modern RAII;
- ...with the performance of host memory.
We exist at the intersection of three areas of development:

- Recent additions to the C++ standard (specifically memory resources);
- Unified shared memory, accessible transparently from both host and device;
- Memory management methods, including sub-allocation schemes.
Memory resources

- Polymorphic memory resources are a C++17 standard library feature.
  - However, adoption has been slow: libstdc++ added support in 9.1 (2019), libc++ is still lacking full support.
- These resources allow us to make run-time decisions about the allocation schemes used by STL containers.
- We do not claim novelty on the design of memory resources – this is part of the library standard.

```cpp
int main(void) {
    std::vector<int> v;
    // Resizing of this vector is handled by the internal logic of the vector class.
    v.push_back(5);
    v.push_back(10);
    v.push_back(2);
}
```

```cpp
int main(void) {
    std::pmr::memory_resource *res = ...;
    std::vector<int, ...> v(res);
    // Here, all allocations and deallocations are handled by the memory resource.
    v.push_back(5);
    v.push_back(10);
    v.push_back(2);
}
```
• One of the core ideas: “hijack” the functionality of memory resources to return device(-accessible) memory.

• This allows us to ergonomically put data into device memory without necessarily having to worry about transferring it.

```cpp
int main(void) {
    vecmem::cuda::managed_memory_resource mem;
    vecmem::vector<int> vec(&mem);

    // All data that we insert into this vector is transparently accessible on the device!
    v.push_back(5);
    v.push_back(10);
    v.push_back(2);

    my_kernel<<<...>>>(vecmem::get_data(vec));
}
```
Heterogeneous memory resources

- At this time, vecmem supports nine memory resources across four platforms:
  - Host memory
  - CUDA
    - Device memory\(^1\)
    - Managed memory
    - Pinned host memory
  - HIP / ROCm
    - Device memory
    - Pinned host memory
  - SYCL
    - Device memory\(^1\)
    - Host memory
    - Shared memory

\(^1\)Device-only memory is supported, but with a smaller feature set.
Performance considerations

- This gives us the tools to ergonomically manage memory on a bunch of platforms!
- But doing this naively is a performance death trap...
- C++ allocation and deallocation patterns assume very low overhead.
- We need a translation layer to platform-friendly allocation patterns.
- For example, the CUB caching allocator, but this has some issues:
  - Only works for CUDA device memory.
  - Only supports one caching scheme.

![Allocation time (ns) vs Allocation size (bytes) graph]

- CUDA 11.4, Linux 5.14, NVIDIA GTX 1660 Ti

Legend:
- Host
- CUDA managed
- CUDA pinned

Graph shows the allocation time in nanoseconds for different allocation sizes (bytes) on various platforms and memory types.
• \textit{vecmem} solves the same problem by taking a compositional approach.
• Compositionality is \textit{the} way to control complexity in computation, so why not in memory?
Downstream resources

- With vecmem, we can construct a caching allocator for CUDA from its components:
  - An upstream CUDA resource;
  - A downstream caching allocator.
- In this example, we use a buddy allocator.
- More caching resources available, such as an arena allocator.

```cpp
int main(void) {
  vecmem::cuda::device_mr up;
  vecmem::buddy_mr dn(up);
  vecmem::vector<int> v(&dn);

  // The first allocation hits the device.
  v.reserve(1024);

  // The following allocations use the cached memory!
  v.shrink_to_fit();
  v.reserve(1024);
  v.shrink_to_fit();
  v.reserve(512);
}
```
Downstream resources

- Downstream resources form a monoid \( \langle \mathcal{M}^\downarrow, \circ \rangle \) under composition\(^1\).
- This means that we can create arbitrarily complex allocation schemes out of simple components.
  - Caching memory resources
  - Conditional memory resources
  - Side-effecting memory resources
- Also, we can do this at runtime, allowing us to treat memory management as a tunable hyperparameter.
- This compositional design delivers high potential for code re-use.
  - New upstream resources can immediately be used with all of our downstream resources.
  - New downstream resources can immediately be used with all of our upstream resources.

\(^1\)This follows from the definition of \( \mathcal{M}^\downarrow \) as an endomorphism: \( \mathcal{M} \rightarrow \mathcal{M} \).
Device code

- *vecmem* makes managing memory on the *host* easier.
- We also provide STL-like containers on the *device*!
  - “Real” STL containers are not supported in device code, regardless of where they are allocated.
- This provides *source-level* portability between device code and host code (often without templating).
- Many modern C++ concepts and semantics are supported on the device.

```cpp
__global__
void kernel(vecmem::vector_view<float> a_data) {
    vecmem::device_vector<float> a_device(a_data);
    float a1 = a_device.at(1);
    ...
}

int main() {
    vecmem::cuda::managed_memory_resource mr;
    vecmem::vector<float> a_host(&mr);
    ...
    kernel<<<...>>>(vecmem::get_data(a_host));
}```
Device code

- Device containers are created from convenient data-management objects.
- Container data can be elegantly moved to and from the device using:
  - Owning buffer types;
  - Non-owning view types.
- Light-weight types that can be passed by value to device kernels.
- Provide the tools necessary to work with host-inaccessible memory.
Device code

- `vecmem` vectors provide support for limited insertion on the device.
  - Requires sufficient capacity allocated before kernel launch.
- We also support a variety of other datatypes on the device side:
  - Jagged vectors, vectors-of-vectors of different sizes.
  - Static vectors, with compile-time fixed capacity.
  - Arrays of compile-time static size, for environments where `constexpr` relaxation is not available.
Conclusions

- Memory management in C++ and heterogeneous platforms has diverged significantly.
- Clever use of *memory resources* allow us to use them for heterogeneous allocation.
- *vecmem* delivers the tools necessary to do this, as well as device-side containers.
- Available under the MPL 2.0 license at https://github.com/acts-project/vecmem.