Journal of Physics: Conference Series

PAPER • **OPEN ACCESS**

Managing heterogeneous device memory using C++17 memory resources

To cite this article: S N Swatman *et al* 2023 *J. Phys.: Conf. Ser.* **2438** 012050

View the article online for updates and enhancements.

You may also like

- pynucastro: A Python Library for Nuclear Astrophysics
Alexander I. Smith, Eric T. Johnson, Zhi Chen et al.
- Preparing for the new C++11 standard
Axel Naumann
- Migrating large codebases to C++ Modules
Y Takahashi, O Shadura and V Vassilev

This content was downloaded from IP address 216.173.84.159 on 01/10/2023 at 12:33

Managing heterogeneous device memory using C++17 memory resources

S N Swatman[1,2], A Krasznahorkay[1] and P Gessinger[1]

[1] European Organization for Nuclear Research, Meyrin, Switzerland
[2] University of Amsterdam, Amsterdam, The Netherlands

E-mail: stephen.nicholas.swatman@cern.ch attila.krasznahorkay@cern.ch paul.gessinger@cern.ch

Abstract. Programmers using the C++ programming language are increasingly taught to manage memory implicitly through containers provided by the C++ standard library. However, heterogeneous programming platforms often require explicit allocation and deallocation of memory. This discrepancy in memory management strategies can be daunting and problematic for C++ developers who are not already familiar with heterogeneous programming. The C++17 standard introduces the concept of *memory resources*, which allow the user to control how standard library containers allocate memory; we believe that this addition to the C++17 standard is a powerful tool towards the unification of memory management for heterogeneous systems with best-practice C++ development. In this paper, we present *vecmem*, a library of memory resources which allows efficient and user-friendly allocation of memory on CUDA, HIP, and SYCL devices through standard C++ containers. We investigate the design and use cases of such a library, the potential performance gains over naive memory allocation, and the limitations of this memory allocation model.

1. Introduction

Heterogeneous software is increasingly becoming a necessity in scientific computing as it allows tackling data challenges at ever greater scale, and with ever greater efficiency [1]. Unfortunately, heterogeneity brings its own sets of challenges, one of which is increased complexity to developers; developing software for heterogeneous systems is often very different from more traditional homogeneous systems, which can be especially daunting to domain scientists who may not necessarily be experts in computing [2]. As a result, developing and maintaining heterogeneous software comes at the cost of great human effort, and the resulting software may be more error-prone and harder to maintain.

One particularly grating aspect of many heterogeneous development platforms is the management of device memory. Over the last few decades, the C++ language has moved away from explicit management of memory by the programmer: the use of `malloc` and `free` to dynamically allocate and deallocate memory on the heap, as well as the `new` and `delete` keywords, have become increasingly uncommon [3]. Instead, it is often considered good practice to rely on *implicit* memory management, such as through RAII and standard library containers, which hide the management of memory from the user completely [4]. Meanwhile, heterogeneous platforms are often designed to be compatible with the C programming language and are

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

therefore required to be far more *explicit*. In CUDA, for example, the `cudaMalloc` and `cudaFree` methods are pervasive [5].

This leads to the unfortunate state of heterogeneous memory management today: we teach aspiring domain scientists to practice modern C++ on homogeneous systems, and then ask them to work with heterogeneous platforms which do not support those practices. This not only wastes these scientists' valuable time, but also increases the potential for errors in the software; explicit memory management can very easily cause memory safety violations if not used carefully [6].

We believe that this barrier between homogeneous and heterogeneous memory management is not only undesirable, but also unnecessary. In this paper, we present *vecmem* (available at `github.com/acts-project/vecmem`), a library which bridges the gap between the ergonomics of idiomatic C++ and the power of heterogeneous computing. Our library is based around the use of standard C++ library classes such as vectors and smart pointers with device memory through *memory resources*, a C++17 library feature which allows fine-grained control over the allocation schemes of library classes [7]. We extend these memory resources to cover heterogeneous memory, and demonstrate how they can be applied to heterogeneous programming to provide a more comfortable and productive experience to C++ programmers. The project also provides the CMake [8] build infrastructure for making use of the supported heterogeneous programming languages on all platforms that those languages themselves support.

2. Host-side Functionality

The objective of the host-side[1] functionality of *vecmem* is to allow C++ developers to interact with device-accessible memory in ways that: (1) are as similar as possible to standard C++ code; (2) support a diverse range of heterogeneous programming platforms; and (3) are sufficiently performant as to not introduce new bottlenecks. To this end, *vecmem* takes a compositional view of memory management and provides two classes of memory resources; our library provides *upstream* memory resources which interface directly with the underlying programming platform, as well as *downstream* memory resources which wrap additional behaviour around an existing memory resource. For example, an upstream allocator might interface directly with the CUDA runtime library, while a downstream allocator may sub-allocate memory allocated by that upstream allocator in order to avoid costly and unnecessary allocations and deallocations. Examples of interactions between user code and the heterogeneous platform, directly as well as through our library, are given in Figure 1.

2.1. General memory resources

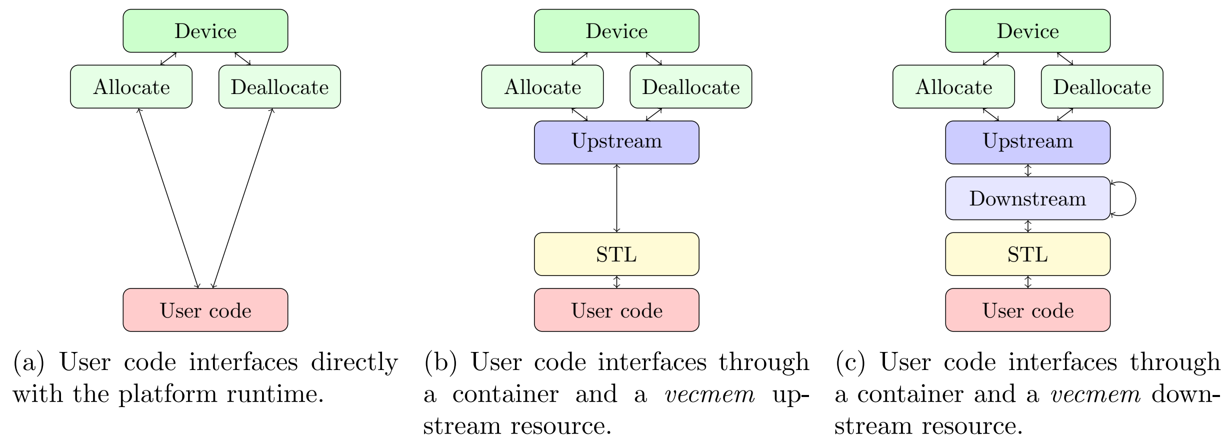
We will first examine memory resources in the general sense, the type of which we shall refer to as $\mathcal{M}$. Memory resources share a common interface which requires the implementation of the following three methods:

Allocation A memory resource must be able to allocate a chunk of memory of a given size with a given alignment[2]. This allocation is allowed to fail; such failures are – in accordance with the C++ standard – modelled as exceptions.

Deallocation A memory resource must be able to deallocate a chunk of memory of a given size, at a given pointer. This method is assumed to always succeed, and as such does not return any indication of success or failure. Notably, invalid deallocation requests (e.g. double deallocations, or deallocations where the size does not match the original allocation) are

[1] In this context, the *host* refers to the processing unit responsible for the core functionality of the system, commonly a CPU. In contrast, a *device* refers to some auxiliary processing unit such as a GPU.

[2] The alignment of an allocation refers to the requirement for that allocation to begin at a memory address which is a multiple of the alignment size.



(a) User code interfaces directly with the platform runtime.

(b) User code interfaces through a container and a *vecmem* upstream resource.

(c) User code interfaces through a container and a *vecmem* downstream resource.

Figure 1: Models of interaction between the programmer, platform, and device.

considered undefined behaviour at the call site, and thus need not be verified by the callee memory resource.

Equivalence Two memory resources are equivalent if and only if one of the resources can deallocate memory allocated by the other, and vice versa. A memory resource must be able to determine whether it is equivalent to another.

In C++, memory resources can be passed as run-time arguments to certain value constructors of standard library containers. For this to be possible, the container has to be configured to support polymorphic allocators at the type level through specific template parameters. For example, `std::vector` is a type with two template parameters, the second of which determines the allocation scheme[3]. In order to construct *vecmem*'s vector type `vecmem::vector` – itself a type with one template parameter – we partially specialise `std::vector` with `std::pmr::polymorphic_allocator` as its second template argument. In addition to containers, *vecmem* also extends standard library *smart pointers* such that they can be used with memory resources. Since we are effectively re-purposing types from the standard library, we inherit all the methods that C++ developers are used to. In addition, we inherit certain C++ semantics; for example, `vecmem::vector` will automatically deallocate its owned (device) memory when leaving scope, and we inherit support for move semantics.

An example of working with memory resources (in this case, one for CUDA managed memory) is given in Listing 1b. Listing 1a shows idiomatic C++ code for performing the same operations on a vector contained in host memory; these examples are included to illustrate how well the semantics of C++ carry over to *vecmem*.

2.2. Upstream memory resources

In our library, an *upstream memory resource* is defined as a memory resource that interfaces directly with a given platform run-time. We denote the type of these resources as $\mathcal{M}^{\uparrow} <: \mathcal{M}$[4]. Note that all upstream memory resources are, in themselves, complete memory resources, and may be used with containers and other *vecmem* features directly. Some upstream memory resources are additionally parameterized. For example, CUDA memory resources may be configured to act on a specific device, if there are multiple devices present in the system. We

[3] This template parameter is usually hidden from developers, as the default argument is sufficient for most common use cases.

[4] All upstream memory resources are memory resources, but not all memory resources are upstream memory resources. As such, $\mathcal{M}^{\uparrow}$ is a subtype of $\mathcal{M}$.

```
std::vector<int> v;

v.push_back(5);
for (size_t i = 0; i < 100; ++i) {
  v.emplace_back(i);
}

v.resize(150);
std::iota(v.begin(), v.end(), 0);
std::sort(v.begin(), v.end());
```

(a) Using idiomatic C++ for host memory.

```
vecmem::cuda::managed_memory_resource m;
vecmem::vector<int> v(&m);

v.push_back(5);
for (size_t i = 0; i < 100; ++i) {
  v.emplace_back(i);
}

v.resize(150);
std::iota(v.begin(), v.end(), 0);
std::sort(v.begin(), v.end());
```

(b) Using *vecmem* for shared CUDA memory.

Listing 1: Inserting and manipulating elements of a vector in various kinds of memory using idiomatic C++ and *vecmem*.

currently provide nine upstream memory resources: one for host memory, three for the NVIDIA CUDA platform (one for pinned host memory, one for shared[5] memory, and one for device memory) [5], two for AMD HIP (one for device memory and one for shared memory) [10], and three for SYCL (one for host memory, one for shared memory, and one for device memory) [11].

2.3. Downstream memory resources

To allow users to add additional logic to allocation schemes, *vecmem* defines *downstream memory resources*, the type of which is defined as $\mathcal{M}^{\downarrow} \equiv \mathcal{M} \to \mathcal{M}$. Crucially, downstream memory resources are not, in themselves, fully fledged memory resources; they cannot be used to allocate or deallocate memory without first applying them – in a functional sense – to an existing memory resource. *vecmem*'s downstream memory resources can be roughly subdivided into the following categories, based on their intended use:

Caching memory resources provide caching of allocated memory such that it is not immediately deallocated when it is no longer needed. This allows future allocations to avoid interactions with the heterogeneous runtime, which are generally orders of magnitude slower than allocations in host memory. We provide memory resources with buddy allocation [12] and arena allocation schemes [13].

Utility memory resources can be used to enforce certain rules on allocation schemes. For example, we provide a memory resource that guarantees that memory is allocated in a contiguous block of memory.

Conditional memory resources provide branching behaviour to allocation schemes, allowing developers to encode complex decision-making processes in their memory management systems. For example, a conditional memory resource may direct small allocations to one memory resource, and large allocations to another.

Instrumentation memory resources allow developers to instrument, profile, and debug their memory allocation schemes by providing useful side-effects, such as providing helpful debug messages, checking whether incoming allocations are valid, and recording the execution times of individual allocations.

Abstract memory resources have properties that are useful for studying memory resources in an abstract sense, but which have little use in practice. For example, the identity memory resource which simply forwards any requests upstream, and the terminal memory resource which always fails.

[5] In this context, *shared* memory refers to memory that is – through a sufficiently abstract lens – accessible to both the host and a device. In CUDA terminology, this is also referred to as *managed* memory [9].

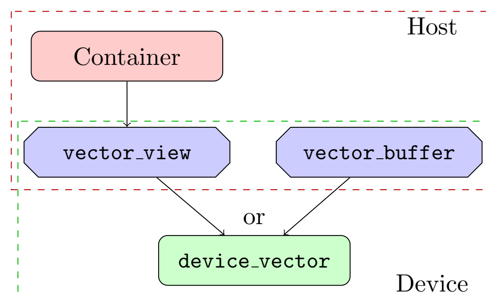


Figure 2: The construction and consumption of buffer and view types enables elegant crossing of the host-device barrier.

The downstream memory resources provided by our library form a monoid $\langle \mathcal{M}^{\downarrow}, \circ \rangle$ under composition[6]. This property allows for the construction of arbitrarily complex downstream allocators – at run-time, if necessary. For example, the configuration of memory allocation may be derived from user-supplied configuration, or it may be decided heuristically at run-time; treating memory management as a hyper-parameter in this fashion can provide an avenue of optimisation in cases where the performance of an application is highly sensitive to the overhead incurred by memory allocation.

We believe the compositional design of memory resources is a powerful one, as any downstream allocator can, in principle, be used with any upstream allocator. This means that the addition of a single upstream memory resource for a new programming platform allows the user to create a large number of composite memory resources with different allocation strategies for different use cases, without any additional development work. Similarly, a new downstream memory resource can immediately be used with any existing upstream allocators; a programmer implementing a new downstream allocation strategy can immediately use that strategy with CUDA, HIP, SYCL, and host memory.

3. Device-side Functionality

In addition to host-side code related to the *allocation* of memory, we also provide device-side functionality for *using* that memory. Like the host-side interfaces, our goal is to provide a software development experience that is as close to the homogeneous C++ experience as possible. Most containers and types defined in the C++ standard library are not available in heterogeneous environments; even if the memory they use is accessible by the device, the code that operates on that memory is not usable on the device. To resolve this problem, we provide device-compatible classes that mimic the behaviour of standard library containers.

In order to cross the barrier between data structures on the host and the device, *vecmem* provides light-weight data types that can be passed directly to the device. We provide *buffer* types which own the memory they point to, as well as *view* types, which do not. These buffer and view types can be generated on the host, and passed to the device. In device code, we can then construct standard library-like containers around *view* type objects, which we can interact with as we would expect to. A schematic view of the different types involved in this host-device interaction is shown in Figure 2. Additionally, these types allow the user to work with host-inaccessible memory in ways that we cannot safely do using standard types.

In order to emulate the ergonomics of C++ library containers, we provide high-level features such as data accesses and iterators. In addition, we provide support for operations which are

[6] In other words, any two downstream memory resources m and n can be chained to produce $m \circ n$, which is guaranteed to be another downstream memory resource. This property follows from our definition of $\mathcal{M}^{\downarrow}$ as an endomorphism.

```
__global__ void kernel(
  vecmem::data::vector_view<float> out,
  vecmem::data::vector_view<float> buf
) {
  vecmem::device_vector<> out_v(out);
  vecmem::device_vector<> buf_v(buf);

  buf_v.push_back(123.f);
  out_v[5] = 1.23f;
}
```

(a) Device-side code.

```
vecmem::cuda::managed_memory_resource mr;

vecmem::vector<float> out{100, 0, &mr};
vecmem::data::vector_buffer<float> buf{
  100, 0, mr};

kernel<<<1, 1>>>(
  vecmem::get_data(out), buf);
```

(b) Host-side code.

Listing 2: Example of using *vecmem* device containers in a CUDA application.

non-trivial on heterogeneous devices, such as the resizing of vectors. As long as there is sufficient capacity in a vector, we allow device-side code to insert elements into it atomically, ensuring that there are no data races in a massively parallel environment such as a GPU. An example of passing containers to a CUDA kernel, one of which would only be used on the device by the kernel, is shown in Listing 2.

4. Conclusion

In this paper, we have presented the design and implementation of the *vecmem* library, which aims to bring the software development ergonomics of C++ to heterogeneous device memory. Our library is built upon the idea of composing upstream and downstream allocators, to reduce overhead while supporting a large variety of platforms. In addition, we provide support for device-side containers which emulate the behaviour of standard library containers.

Acknowledgments

This work was partially funded and supported by the CERN Strategic R&D Programme on Technologies for Future Experiments [14].

References

[1] Owens J D, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn A E and Purcell T J 2007 *Computer Graphics Forum* **26** 80–113
[2] Ujaldón M 2016 CUDA achievements and GPU challenges ahead *International Conference on Articulated Motion and Deformable Objects* (Springer) pp 207–217
[3] Stroustrup B 1996 *A History of C++: 1979–1991* (New York, NY, USA: Association for Computing Machinery) p 699–769 ISBN 0201895021
[4] Murray R B 1993 *C++ strategies and tactics* (Addison Wesley Longman Publishing Co., Inc.)
[5] NVIDIA Corporation 2021 CUDA C++ Programming Guide
[6] De Amorim A A, Hriţcu C and Pierce B C 2018 The meaning of memory safety *International Conference on Principles of Security and Trust* (Springer) pp 79–105
[7] Dawes B and Meredith A 2016 Adopt Library Fundamentals V1 TS Components for C++17 (R1)
[8] Hoffman W and Martin K 2003 *Dr. Dobb's Journal: Software Tools for the Professional Programmer* **28** 40–43
[9] Li W, Jin G, Cui X and See S 2015 An evaluation of unified memory technology on NVIDIA GPUs *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (IEEE) pp 1092–1098
[10] Advanced Micro Devices, Inc 2021 HIP Programming Guide
[11] Reyes R and Lomüller V 2016 SYCL: Single-source C++ accelerator programming *Parallel Computing: On the Road to Exascale* (IOS Press) pp 673–682
[12] Knowlton K C 1965 *Communications of the ACM* **8** 623–624
[13] Hanson D R 1990 *Software: Practice and Experience* **20** 5–12
[14] CERN Experimental Physics Department 2018 Strategic R&D Programme on Technologies for Future Experiments Tech. rep. CERN Geneva URL `https://cds.cern.ch/record/2649646`