Beyond Explicit Transfers: Shared and Managed Memory in OpenMP

Brandon Neth[1], Thomas R.W. Scogland[2], Alejandro Duran[3], and Bronis R. de Supinski[2]

[1] University of Arizona, Tucson AZ 85721, USA
[2] Lawrence Livermore National Lab, Livermore CA 94550, USA
[3] Intel Corporation, Iberia, Spain

Abstract. OpenMP began supporting offloading in version 4.0, almost 10 years ago. It introduced the programming model for offload to GPUs or other accelerators that was common at the time, requiring users to explicitly transfer data between host and devices. But advances in heterogeneous computing and programming systems have created a new environment. No longer are programmers required to manage tracking and moving their data on their own. Now, for those who want it, inter-device address mapping and other runtime systems push these data management tasks behind a veil of abstraction. In the context of this progress, OpenMP offloading support shows signs of its age. However, because of its ubiquity as a standard for portable, parallel code, OpenMP is well positioned to provide a similar standard for heterogeneous programming. Towards this goal, we review the features available in other programming systems and argue that OpenMP expand its offloading support to better meet the expectations of modern programmers. The first step, detailed here, augments OpenMP's existing memory space abstraction with device awareness and a concept of shared and managed memory. Thus, users can allocate memory accessible to different combinations of devices that do not require explicit memory transfers. We show the potential performance impact of this feature and discuss the possible downsides.

1 Introduction

Heterogeneous systems are becoming the new norm in computing, from the smartphones in our hands to the largest computers in the world. GPUs, and the introduction of general purpose GPU (GPGPU) programming, have been the stars of this growth in popularity. Even so, leveraging the performance benefits of a heterogeneous system can be a time intensive process. One cause of this complexity is the variety of GPU programming systems available to programmers. Compounding this problem is the desire to write single-source code that will perform well across a variety of node architectures. While OpenMP* has supported heterogeneous programming to some extent since v4.0, managing memory among the execution units has been limited.

OpenMP's current support for memory allocation includes allocating host memory and allocating device memory. In contrast to this binary model, CUDA* supports a much wider variety of memory allocations, including device-accessible host memory, managed memory accessible by all execution units, and different levels of device-only memory, such as thread-private and thread-group-private memory. Similarly, OpenCL* 3.0 supports shared memory and different device memory allocations, while oneAPI Level Zero(Level Zero) supports a more relaxed model of memory ownership, where host memory allocations are device-accessible, and data movement does not need to be explicit.

If OpenMP is to stay competitive in this space, a more nuanced system of memory allocation is necessary. This paper presents such a system. The contributions of this work include:

- a survey of memory allocation and management features in existing heterogeneous programming systems,
- a proposed improvement to OpenMP's current allocation and management features, and
- an evaluation of the potential performance impact of such a change.

Section 2 summarizes OpenMP's current support for memory allocation. Section 3 surveys the approach to memory allocation and management in existing programming systems. Section 4 proposes new memory allocation features for OpenMP. Section 5 compares the performance of OpenMP's current capabilities with the capabilities of other programming systems. Section 6 concludes.

2 Current Support in OpenMP

2.1 Allocators

Because it was originally designed for shared-memory multi-core parallelism, OpenMP's memory model does not easily line up with those commonly used in offloading. One specific concept that is missing is accessibility of memories from more than one device. Instead, OpenMP supports two completely separate interfaces for memory on the host and memory on target devices.

2.2 Host Memory

The OpenMP allocator APIs support only memory that is accessible by the host. The allocator clauses can support device memory when executing a construct on that device, but only in certain circumstances. Rather than allocating for other devices, it allows a programmer to specify the properties they want for an allocation to allow allocation in lower latency, larger or higher bandwidth memories as appropriate:

- `omp_default_mem_space`: Default system storage
- `omp_large_cap_mem_space`: Storage with large capacity

- omp_const_mem_space: Storage optimized for reading rather than writing
- omp_high_bw_mem_space: Storage with high bandwidth
- omp_low_lat_mem_space: Storage with low latency.

Note that within these five predefined memory spaces, no clarification is made about the location of the memory in terms of device accessibility [5](2.13.1). Also they are all properties that change based on where code is running, so the context is important. The implicit context for all of these properties is that of the thread doing the allocation, not the one that creates the allocator.

Memory spaces are used by OpenMP memory allocators to request memory from the system [5](2.13.2). There are two ways to use an allocator within a target region. First, if the target region contains the `uses_allocators` clause, the target region can contain `allocate` directives that use the specified allocators. Second, default allocators and allocator routines can be used within a target region if the `requires(dynamic_allocators)` clause is present [5](2.5.1).

2.3 Device Memory

OpenMP also supports directive-based data movement between the host and device for specific code regions. When a `target` region begins, the `map` clause indicates what data to move when, either host to device (`to`), from device back to host (`from`), or host to device and back (`tofrom`) [5](2.21.7.1). The particular nature of how this data is moved, copied, mapped, registered or even if anything is done at all, is implementation defined. So the programmer does not have control over how the memory becomes accessible to the device, only control over whether it needs to be made accessible.

Although it is not supported using directives, OpenMP has runtime functions for allocation and management of device memory. For allocation and freeing of device memory, programmers use `omp_target_alloc` and `omp_target_free`. Memory from the device is considered inaccessible from the host, and from other devices. Beyond that, the pointer returned by `omp_target_alloc()` isn't even considered a valid pointer (no pointer arithmetic allowed, can not be dereferenced, may not be unique) until it has been passed, and possibly fixed up, by passing it to the device it was allocated for with an `is_device_ptr()` clause. There is also support for queries about device memory: `omp_target_is_present` checks whether a pointer refers to memory that has been mapped with one of the mapping constructs, and `omp_target_is_accessible` checks whether memory can be accessed by a device. Copying is supported using `omp_target_memcpy` and `omp_target_memcpy_rect`, asynchronously by `omp_target_memcpy_async` and `omp_target_memcpy_rect_async`. Finally, `omp_target_associate_ptr` associates a device pointer to a host pointer so map clauses move the host pointer's data to the device pointer instead of potentially allocating new device memory [5](3.8). LLVM specific extensions also exist for allocating host-accessible, device-accessible, and migratable memory [1].

3 Survey

Multiple different programming systems are in use for heterogeneous computing. We survey those systems here, focusing on their approach to memory allocation and management.

3.1 OpenCL

Within an OpenCL program, the highest level scope is the *context*, which contains a host, some number of devices, command-queues, and memory. The memory model describes how the other elements of the context access and modify data values. It is broken into four parts: memory regions, memory objects, shared virtual memory, and consistency model [3](3.3).

Memory regions, named address spaces between which memory objects are moved, are divided into two top-level types. Host memory is the memory available to the host device. This is "normal" memory recognizable to homogeneous system programmers. Device memory is the memory available to the devices executing OpenCL kernels. Device memory is further divided into four address spaces. Global (device) memory is memory accessible to all parts of all devices. Constant (device) memory is similar to global memory, but it is initialized by the host and can not be changed by the devices. Local (device) memory is a memory region accessible by a single work-group. All work-items in the group can access local memory. Finally, private (device) memory is only accessible by individual work items.

Memory objects manage the transfer and manipulation of pieces of data. OpenCL contains three types of memory objects, all of which are part of global memory. The simplest memory object is the buffer. A buffer is a block of contiguous memory that can hold any data and is manipulable through pointers. Second is the image, which holds one, two, or three dimensional images. While as a data structure the image is more complex than the buffer, it is moved between host and device in the same manner as the buffer. The last memory object is the pipe, similar to the queue data structure. The pipe has two endpoints that kernels can connect to. One kernel connects to the write endpoint, where it produces values and writes them into the pipe. The second kernel connects to the read endpoint, where it consumes the values for its own computation [3](3.3.2).

Shared virtual memory (SVM), introduced in OpenCL 2.0, combines parts of the global and host memory regions to create a region accessible by all computing elements. OpenCL has three types of SVM. Coarse-grained buffer SVM works at the level of buffer memory objects. Explicit synchronization drives updates between the host and devices, so coarse-grained SVM mimics non-SVM in code design. However, because coarse-grained SVM does not move buffers between devices, pointer-based structures like trees can be used by devices. Fine-grained buffer SVM works at the level of individual accesses to the bytes within buffer objects, while fine-grained system SVM works at the level of individual accesses to the bytes of the entire host memory. For fine-grained SVM, consistency is guaranteed using explicit synchronization between devices [3](3.3.3).

The last piece of the OpenCL memory model is the consistency model, consisting of rules about the behavior of data manipulation. OpenCL's consistency model is based on that of ISO C11, using a relaxed memory consistency model. Depending on their needs, a programmer can specify what ordering they need for atomic operations, ranging from the less restrictive `memory_order_relaxed` to the more restrictive `memory_order_acq_rel`. Also, different memory scopes for the ordering constraints enable potential optimizations. For example, atomic operations at the work-item (thread) scope require much less synchronization than those at the global scope [3](3.3.5).

3.2 Level Zero

Unlike OpenCL, which has different named address spaces, Level Zero uses a unified memory design with a single address space. While it uses a single address space, memory can still be managed at a finer granularity. For devices, there is device-wide local memory (global memory in OpenCL terms) and two controllable cache levels (roughly local and private memory in OpenCL terms).

For allocations, three types are supported, based on the ownership of the memory. Host allocations are made out of system memory, but can still be accessed by devices. Because Level Zero uses a unified virtual address space, the same pointer is used on host and device. While host allocations can be accessed on device, they are not meant to be transferred from host memory to device memory, so all accesses must occur over interconnects. Device allocations are made and owned by a specific device. These allocations are not meant to be accessed by any device other than the one they are made on, but have high access speed. Device allocations can be explicitly copied to the host or another device if those devices need access. Last, shared allocations are intended to migrate between the host and devices. These allocations are comparable to CUDA's managed memory [2].

3.3 CUDA

The CUDA programming model, like OpenCL, has a different execution and memory model than OpenMP. Like OpenMP, individual execution units are called threads. Groups of threads are called blocks, and groups of blocks are called grids [4](2.2). This creates a 3 level hierarchy reflected in the memory model. Global memory is accessible by all threads, shared memory (local in OpenCL terms) by threads within a block (work group), and local memory (private) by individual threads [4](2.3). Furthermore, constant memory is a read-only section of global memory.

CUDA supports a variety of approaches to sharing data between host and device. The simplest is explicit transfers to and from the device. Before a kernel begins, the host allocates memory on the device using `cudaMalloc` and copies the host data to the device using `cudaMemcpy`. There are also functions to allocate and copy 2D and 3D arrays [4](3.2.2).

Another sharing technique uses pinned host memory to improve transfer speed and enable concurrent copying. The programmer can use `cudaHostAlloc` and `cudaFreeHost` to allocate and free pinned memory, and `cudaHostRegister` to pin pageable memory [4](3.2.5). Because pinned memory can be copied directly between devices using DMAs, it reduces the transfer cost. Furthermore, transfers can be overlapped with kernel execution. Finally, pinned memory can be mapped into the device address space. This technique is called mapped memory or zero-copy memory. While computation using mapped memory has lower bandwidth, it removes the need to explicitly allocate and transfer data to the device, and the computation and data transfer are automatically overlapped [4](3.2.5.3).

The final data sharing technique in CUDA is Unified Memory Programming. The fundamental component of Unified Memory is the managed memory space: a single memory space, visible and accessible by all devices, with a common address space. For this reason, it is used interchangeably with "managed memory". Like with mapped memory, managed memory removes the need for explicit allocations and transfers between devices. However, with mapped memory, the physical location of the memory is always with the host. With managed memory, data is moved towards where it is being used, so it may reside on any of the devices in the system [4](M.1). Programmers allocate managed memory using `cudaMallocManaged`, and on some systems, using system allocators [4](M.1.1).

By removing the explicit transfers between devices, Unified Memory also removes the synchronization inherent in the transfers. Thus, programmers need to use `cudaDeviceSynchronize` before the CPU uses results from a GPU kernel. This is another distinction between mapped memory and managed memory, because with mapped memory the CPU can access the memory while the GPU is active. Even so, some systems (those with the `concurrentManagedAccess` property), support simultaneous access of managed memory [4](M.2.2.2).

3.4 HIP

HIP is a runtime API and kernel language for creating single-source applications for both AMD and NVIDIA GPUs. Thus, much of HIP's memory and execution model is based on CUDA's, and its API is described as a "strong subset" of CUDA [7]. Of CUDA's memory sharing techniques described above, HIP supports explicit transfer and pinned/mapped host memory [6](3.3.1). HIP does not support managed memory [6](3.4.2.3).

4 Proposed OpenMP Extension

Detailed further in Figure 1, across all of the models we've surveyed, we found support for:

- Device memory accessible only from that device, this OpenMP already supports.

	OpenCL	**Level Zero**	**CUDA**	**HIP**	**OpenMP**
Device-exclusive memory	Yes	Yes	Yes	Yes	Yes
Explicit transfers	Yes	Yes	Yes	Yes	Yes
Device-accessible host memory	Mapped buffers	Host allocations	Mapped memory	Mapped memory	No
Managed memory	Coarse-grained SVM	Shared allocations	Unified memory	In development	No
Shared memory	Fine-grained SVM	Shared allocations	Mapped memory, unified memory on some systems	Mapped memory	No
Unformatted allocations	Buffers	Memory	Linear memory	Linear memory	Arrays
Formatted allocations	Images, Pipes	Images	CUDA arrays	None	Rectangular subvolumes
Language-level variable qualifiers	__global, __constant, __local, __private	None	__device__, __constant__, __shared__, __managed__	_constant__, __shared__	None

Fig. 1. Summary of feature support across programming systems. Table entries are the internal feature name.

- Host memory accessible from other devices: this memory usually also provides faster transfers due to being pinned to a specific physical memory resource.
- Managed memory, accessible from the host and at least one other device but may only be valid to access from one of them at any given time, requiring synchronization *even when concurrent access would not otherwise be a race condition.*
- Shared, shared virtual, or unified memory that can be accessed by all relevant devices simultaneously and provides some mechanism for finer grained coherence than managed memory.

There are several factors at play in this set of interfaces. If we leave off the memory that is only accessible to the device it was allocated for, which is already supported, then we are left with three forms of memory which are all accessible by the host and some set of other devices. Conventionally they map to three, or perhaps four, functions, but amount to three separate axes of memory properties: what devices can access the memory, what synchronization is required for accesses to be correct, and where is the memory allocated and initially resident.

To better support these features, we propose the following extensions to the OpenMP specification. First, we expand the memory space concept to include information about what devices need access to the memory space. Second, we introduce support for memory accessible from multiple devices in the form of shared and managed memory. Last, we'll discuss options for controlling where the memory is allocated, and possibly how it is allowed to migrate after allocation.

4.1 Memory Space Accessibility

In order to expand allocators to apply to multiple devices, we propose to add a way to request new memory spaces, in addition to those provided by default by OpenMP. The new `omp_get_target_memspace` function would accept an array of devices, and a default memory space, and return a new memory space that provides memory accessible from all the devices listed in the array. This method includes the host, whose device number is accessible using `omp_get_initial_device()` [5](3.7.7).

```
omp_memspace_handle_t * omp_get_target_memspace(
  int count,
  int *dev_nums,
  omp_memspace_handle_t existing_memory_space);
```

Allocators created from the new memory space with `omp_init_allocator` will allocate memory accessible from all relevant devices. This interface is currently slightly different from the other allocator interfaces in that it uses a *pointer* to an `omp_memspace_handle_t` object. We decided to do that so that a `NULL` return value could indicate that the requested memory space couldn't be constructed. The `existing_memory_space` argument serves to provide a hint to the runtime about what should be prioritized for memory allocated from this space as well.

4.2 Shared and Managed Memory

Ideally OpenMP should provide a way to request all the relevant allocation types discussed above, but as always OpenMP also needs to remain portable to a variety of hardware and foreign runtimes. The portability is a concern because while the percentage of runtimes that support shared or unified memory (henceforth shared memory) rather than just managed memory is growing, it is still not universal. If we look at these properties across each type however, the coherence and accessibility properties of host pinned memory and shared memory are reasonably equivalent. Further, since managed memory requires more and more stringent synchronization, using either in place of managed memory results in a correct program. It is their performance characteristics that differ.

Since that is the case, any platform that can provide at least host pinned memory could in principle provide semantically correct execution for programs requesting managed or even shared memory, if at a low quality of implementation. That opens the way to use allocator traits to request the appropriate synchronization model, where managed memory could be allocated as managed, shared, or if necessary host pinned and shared could fall back to host pinned as well. Thus, we propose to add a new allocator trait `memory_access` with values of `managed` or `coherent`. For coherent mode, the memory acts like common shared memory, allowing (with other extensions) atomics and other fine-grained synchronization. In managed mode, we would further require that all work on the device had been synced to the host before the host is allowed to access the

memory. That roughly matches the requirement used by HIP and CUDA, but currently lacks a method of tying a specific managed allocation to a stream. It would require the "no mapping" mode of OpenCL SVM either to be available or to be emulated by calls from the OpenMP runtime to perform the appropriate mappings. Since OpenMP lacks an equivalent concept, we will need to consider alternate mechanisms. An interface using a depend object may be an option, but there's no clean mapping for that functionality.

One main downside to relying on this fallback behavior for portability is that it could pose a significant performance problem for code that relies on memory migrating automatically. While we could make it unspecified behavior to use these modes on systems where a fallback is necessary, or add a `requires` clause for it, it would reduce the portability of such code substantially and further subdivide implementations. We discuss some options for detecting and mitigating that in Section 4.3.

4.3 Memory Location Control

Much like selecting between `managed` and `coherent`, selecting the location of an allocation is better done at a finer granularity than the memory space. OpenMP needs a way to specify the context to use when allocating memory from an allocator, in a way that it can be used for both device selection and locality with other places on the host.

We propose that OpenMP adds two new allocator property keys to the allocator traits. `preferred_location` will take a device ID and `preferred_place` a place ID, indicating the users preference for allocations on that device and near that place.

```
1 omp_alloctrait_t traits[] = {{.key = preferred_location, .value = 0}, {.
    key = preferred_place, .value = 1}};
```

Specifying the preferred location this way expands the flexibility of existing traits to allow requesting things like lowest latency with respect to a place, but makes allocating with different preferences across an application cumbersome. Having a way to specify an allocator property as part of an actual allocation, as an extension to the existing allocator API, could help mitigate that but there are several routines and directives involved which would make it a wide-ranging change. Adding a function to produce an allocator from an existing allocator with one property adjusted or added could also help with the proliferation of allocator objects, but exploring that is left for future work.

To offer control over whether memory should be allowed to migrate or not, we propose to leverage the existing `pinned` allocator trait as well. While that makes it partly overloaded, using it in this way avoids adding another trait, and makes it clear that the memory should not migrate even within the device.

Allowing the location and the memory access characteristics to be traits, and effectively hints, allows the API to remain consistent but adds to the challenge for programmers to understand the behavior of their code. If these interfaces become

part of OpenMP, it will be important to expand the allocator API with a way to query or guarantee what access behavior, location and other trait properties are actually provided by a given allocation, or guaranteed by an allocator.

5 Evaluation

To evaluate the potential benefit of our proposal, we compare an OpenMP microbenchmark implementation with two CUDA implementations. The system we used for our evaluation uses two 24-core IBM Power9* CPUs and 4 NVIDIA* V100 Volta GPUs, however our evaluation only uses one GPU. For the OpenMP variant we used IBM's XL compiler, version 16.1.1. For the CUDA variants we used NVIDIA's nvcc compiler, version 10.1.243. We evaluated using a single node.

In our microbenchmark, we execute the `daxpy` (double precision `a*x + y`) kernel a number of times. In the OpenMP variant, we utilize a naive data movement strategy. Each time the kernel is executed, `map()` clauses move the data back and forth between the host and device. The kernel is extracted into the function shown in Listing 1.1. This microbenchmark simulates the potential data reuse between consecutive kernels in a larger application that does not use shared memory. Using this naive approach to improve the code's modularity comes at the cost of repeated data movement.

```
1 void omp_daxpy(int n, double a, double * x, double * y, double * z) {
2     #pragma omp target map(to: x[0:n], y[0:n], n, a) map(tofrom: z[0:n])
3     #pragma omp teams distribute parallel for simd
4     for(int i = 0; i < n; i++) {
5      z[i] += a * x[i] + y[i];
6     }
7 }
```

Listing 1.1. OpenMP microbenchmark kernel implementation.

In contrast, the two CUDA variants use either mapped (zero-copy) or managed memory. With these implementations, the kernel implementation remains modularized, but the data movement is not always performed and when it is it can happen asynchronously. The CUDA variants show the potential for OpenMP codes to maintain modularity and reduce unnecessary data movement without requiring modifying outer scope code as with using `target data`. Performance results for 1, 10, and 100 kernel repetitions are shown in Figure 2. We report the average of three runs.

Figure 2 shows the high costs of offloading data transfers in OpenMP compared to mapped or managed memory. For the managed memory variant, the increase in execution time is minimal, driven by the increase in the size of the computation. In fact, between 1 and 10 repetitions, execution time increases only

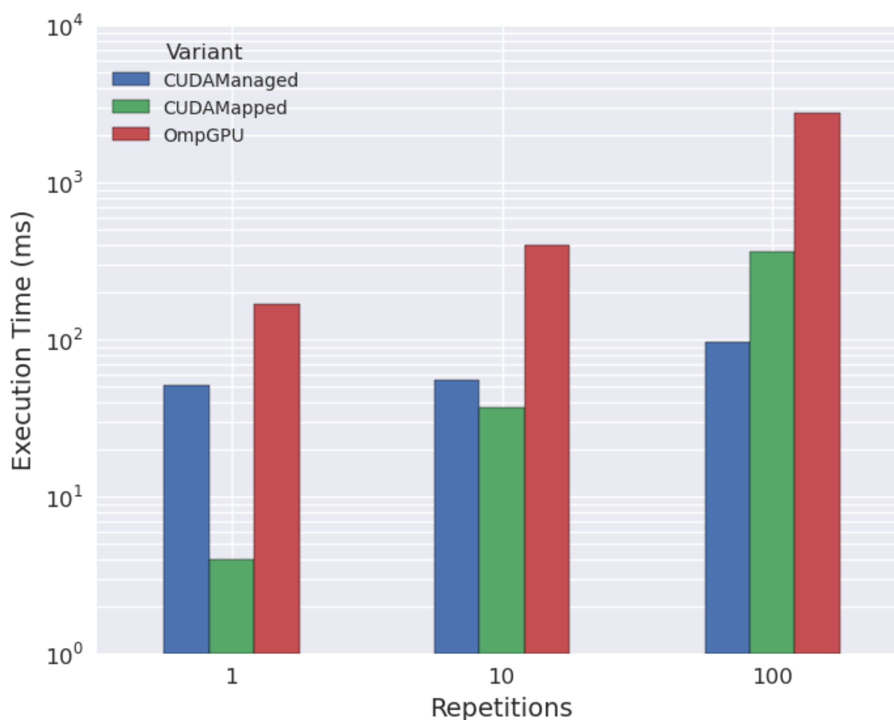

Fig. 2. Execution time for OpenMP offloading, CUDA mapped (zero-copy) memory, and CUDA managed memory variants (lower is better).

7%. In contrast, for the OpenMP variant, execution time increases by 250% from 1 rep to 10 reps and by almost 700% from 10 to 100 due to the transfers back and forth on every iteration. The mapped memory variant increases 12× between 1 rep and 10 reps and 10× between 10 and 100. However, the mapped memory variant still outperforms the OpenMP variant at all tested repetitions. These results demonstrate the necessity for OpenMP to support other types of data sharing among devices if it is to remain competitive not only for functionality but in some cases for performance as well.

6 Conclusion

OpenMP endeavors to provide a comprehensive API for parallel programming, including of heterogeneous nodes since the release of OpenMP 4.0. While the flexibility and power of many of the interfaces for offload have been refined and extended since, support for memories accessible by multiple devices has largely stayed the same. The addition of `requires unified_shared_memory` helps for cases where true shared memory is available, but it's becoming ever more clear

that enforcing an all-or-nothing switch on unified memory for an application is not sufficient to cover common uses across platforms anymore. Therefore, we present a survey of the state of memory accessibility and memory allocation interfaces across several APIs along with a proposal to extend OpenMP to cover a wider variety of commonly available memory types. Further we present a test case where the availability of a managed, or even pinned, allocation in an otherwise manually managed memory application allows for far greater efficiency and performance, 5× faster for this particular microbenchmark in the 100-run case.

Beyond what we have explicitly proposed here, bringing allocations supporting access from multiple devices into OpenMP opens the door to many more use-cases in the future. Fine-grained synchronization between host and target devices with cross-device atomics along with scoped atomics become a possibility in portable applications. Incorporating the notion of places in devices raises the possibility of supporting subsections of devices, and more closely incorporating offload devices into the places list as well.

Disclaimers

*Brands and names are the property of their respective owners.

References

1. grokos. [libomptarget] add allocator support for target memory, March 2021.
2. Intel. oneAPI Level Zero: 1.1.2, 2020.
3. Khronos OpenCL Working Group. OpenMP application program interface version 5.1, April 2021.
4. NVIDIA. Cuda C++ programming guide v11.3.1, May 2021.
5. OpenMP Architecture Review Board. OpenMP application program interface version 5.1, November 2020.
6. Radeon Open Compute. HIP API, May 2021.
7. Radeon Open Compute. Hip-faq, 2021.