Developments in memory management in OpenMP

Jason Sewall*, S. John Pennycook and Alejandro Duran

Intel Corporation,
3600 Mission College Boulevard,
Santa Clara, California 95050, USA
Email: jason.sewall@intel.com
Email: john.pennycook@intel.com
Email: alejandro.duran@intel.com
*Corresponding author

Christian Terboven

High Performance Computing (I12),
RWTH Aachen University,
Seffenter Weg 23 52074 Aachen, Germany
Email: terboven@itc.rwth-aachen.de

Xinmin Tian and Ravi Narayanaswamy

Intel Corporation,
3600 Mission College Boulevard,
Santa Clara, California 95050, USA
Email: xinmin.tian@intel.com
Email: ravi.narayanaswamy@intel.com

Abstract: Modern computers with multi-/many-core processors and accelerators feature a sophisticated and deep memory hierarchy, potentially including distinct main memory, high-bandwidth memory, texture memory and scratchpad memory. The performance characteristics of these memories are varied, and studies have demonstrated the importance of using them effectively. In this article, we explore some of the major issues in developing software to effectively and portably implement these technologies and describe enhancements being added to the OpenMP language to bridge this software-hardware gap. Our proposal separately exposes the characteristics of memory resources (such as kind) and the characteristics of allocations (such as alignment), and is fully compatible with existing OpenMP constructs.

Keywords: memory management; programming languages; compiler directives; heap allocation.

Reference to this paper should be made as follows: Sewall, J., Pennycook, S.J., Duran, A., Terboven, C., Tian, X. and Narayanaswamy, R. (2019) 'Developments in memory management in OpenMP', *Int. J. High Performance Computing and Networking*, Vol. 13, No. 1, pp.70–85.

Biographical notes: Jason Sewall is a Software Engineer in the HPC Ecosystem and Application Team (HEAT) interested in optimisation, numerical techniques, and parallel algorithms. He holds a PhD in Computer Science from UNC-Chapel Hill.

S. John Pennycook is an HPC Application Engineer at Intel Corporation, focused on enabling developers to fully utilise the parallelism available in Intel Xeon Phi processors and coprocessors. He holds a PhD in Computer Science from the University of Warwick.

Alejandro Duran is an Application Engineer in Intel's Data Center Group. He holds a PhD in Computer Engineering from the Technical University of Catalonia. He has been part of the OpenMP language committee since 2006.

Christian Terboven is a Senior Scientist and group leader of the HPC group at RWTH Aachen University. Since 2006, he is a member of the OpenMP language committee, where he leads the affinity subcommittee. He holds a Doctoral degree in Computer Science from RWTH Aachen University, Germany.

Xinmin Tian is a Senior Principal Engineer and Compiler Architect in the Intel Compilers and Languages organisation. His primary responsibility is compiler vectorisation, parallelisation and OpenMP for Intel Compilers and Intel LLVM-based compilers for IA based platforms.

Ravi Narayanaswamy is a Senior Staff Engineer in the Intel Compiler Lab. He is currently working on offloading support in the compiler and library. Since he joined Intel, he has worked on compilers for various platforms. His areas of expertise are scalar optimisation, exception handling, transaction memory and offloading. He holds an MS degree in Environmental Engineering and in Computer Science, both from Southern Illinois University, Carbondale.

This paper is a revised and expanded version of a paper entitled 'A modern memory management system for OpenMP' presented at WACCPD 2016, Salt Lake City, 14 November 2016.

1 Introduction

Although memory capacity has been steadily increasing over the history of computing, memory bandwidth has not kept pace with processor throughput (Borkar and Chien, 2011), particularly in the face of growing amounts of parallelism. As a result, the memory subsystems of modern computing devices – socketable processors and coprocessors/accelerators alike – have a deep hierarchy employing a variety of memory technologies (Handy, 2014; Smith, 2015, 2016; Valich, 2015): main memory (DDR), high-bandwidth memory (HBM) (Sodani et al., 2016), texture memory (Akeley, 1993), scratchpad memory (SPM) (Banakar et al., 2002), and multiple levels of cache. The hierarchy found in future systems is expected to be deeper still following the introduction of non-volatile RAM (NVRAM) and other developments.

Each of these technologies has distinct performance characteristics (e.g., latency, bandwidth) and semantic characteristics (e.g., access permissions, volatility). The variation in these characteristics can be significant, and addressing them in software is becoming increasingly critical for high-performance computing (HPC); considering memory bandwidth alone, today's DDR maxes out at 25.6 GB/s, HBM (Handy, 2014; Kim and Kim, 2014) supports a 1,024 bit-wide bus at 125 GB/s, and the hybrid memory cube consortium (Handy, 2014) will likely reach 400 GB/s based on stacked memory chips (Handy, 2014; Smith, 2016; Valich, 2015; Kim and Kim, 2014).

The existing methods of utilising the available varieties of memory are generally proprietary, narrow in scope, or both; in this paper, we propose extensions to OpenMP* that enable the use of new memory technologies by user software in a platform-agnostic, future-proof and portable fashion.

Our OpenMP extensions consist of:

- New directives, clauses and runtime APIs that replace non-portable and vendor-specific solutions for memory management, and are fully compatible with existing OpenMP parallel, SIMD and offloading constructs.
- A mechanism for vendor extensions that allow growth and adaptation to the newest hardware features, minimising the need for additional non-portable and vendor-specific APIs in the future.
- A method of passing additional information about memory (e.g., kind, size, alignment) to compilers and OpenMP runtime libraries, allowing for better code generation and improved runtime performance.

To the best of our knowledge, ours is the first attempt to standardise program constructs and directives for effective utilisation of existing and future memory in software.

Table 1 A comparison of the different memory kinds available in modern hardware

Platform	*Memory kind*							
	Constant	*Texture*	*SPM*	*DDR*	*eDRAM*	*GDDR*	*HBM*	*NVRAM*
Intel® Xeon® processor	–	–	–	✓	–	–	–	–
1st Gen. Intel® Xeon PhiTM coprocessor	–	–	–		–	✓	–	–
2nd Gen. Intel® Xeon PhiTM processor	–	–	–	✓	–	–	✓	
Future system with 3D XPoint™	–	–	–	✓	–	–	–	✓
Intel® HD Graphics	–	–	✓	✓	–	–	–	–
Intel® Iris™ Graphics	–	–	✓	✓	✓	–	–	–
pre-'Pascal' NVIDIA GPU	✓	✓	✓			✓	–	–
'Pascal' NVIDIA GPU	✓	✓	✓			✓	✓	

2 Motivation

Table 1 highlights the wide variety of different memory kinds that are available in a selection of modern hardware platforms. In cases where the only difference between platforms is the default memory kind (e.g., DDR, GDDR and HBM in certain configurations) or an application developer is otherwise content to use only one of the available memory kinds, it may be possible to make platform differences transparent to software. However, as demonstrated by our table, there are often more significant differences between platforms than the default memory kind; furthermore, each distinct memory in a system may require some special (and often proprietary) consideration in applications.

Application optimisation studies have repeatedly demonstrated that the performance impact of placing specific data in certain memories can be significant (Doerfler et al., 2016; Heinecke et al., 2016; Ruetsch and Micikevicius, 2009; Friedrichs et al., 2009). Unfortunately, it is highly unlikely that compilers (or other tools) with no hints about intent from the user will be able to reliably determine the optimal memory configuration for data in applications. There is a clear need for a procedural method for developers to manually select and use different memory kinds for individual allocations, and to make this selection on a platform-by-platform basis.

As a simple motivating example, we consider a hypothetical future processor with a deep memory hierarchy: a scratchpad (local) memory per core; a HBM of limited capacity; a low-bandwidth capacity memory; and a non-volatile (persistent) memory. From a correctness standpoint, whether dynamic allocations are small enough to fit entirely within scratchpad or HBMs is something that can only be known at runtime (or asserted at compile-time by the developer). From a performance standpoint, even the simplest and most obvious mappings of application logic to this hierarchy are complicated to express: that threads on the same core should collaborate within the scratchpad; that bandwidth-critical data should be placed in the HBM; that other data (including bandwidth-critical data that exceeds the capacity of the HBM) should be stored in the capacity memory; and that some subset of the data (a checkpoint) should be written to the persistent memory at some application-specific frequency.

This paper describes directives and a procedural interface as an extension to OpenMP that will deliver enhanced performance and flexibility to programs. Furthermore, these additions are extensible with respect to both hardware and software evolution: as operating systems and runtimes are enhanced with improved memory management interfaces, and as memory technologies continue to advance, the flexible interface proposed here will make their functionality available to application developers.

As application developers ourselves, it is our hope that that these enhancements will see widespread adoption, as they provide portable and extensible methods for memory management that is lacking in most current systems.

3 State-of-the-art

While the term ‘compute’ is used to refer to the broad notion of using computers in any fashion, it is also used to indicate the literal execution of instructions, generally in contrast to memory usage (e.g., compute/memory ratios). Despite this distinction, memory and memory management have always been central parts of computing – in the former sense – and a tremendous amount of work has gone into the development of both storage technologies and methods for exposing and utilising them efficiently. Below, we review how these have evolved over the years.

3.1 Memories past, present, and future

3.1.1 Virtual memory

With the advent of multi-user operating systems, it became necessary to protect, allocate, and share system memory among multiple executing processes. Multiple solutions were developed, from simply dividing memory between programs [known as *base and bounds* (Saltzer and Schroeder, 1975)], to segmentation, to virtual memory. Virtual memory simply entails the separation of physical memory addresses from logical ones exposed to software; hardware support for this practice is extremely useful, and all but the most specialised/low-power processors have a memory management unit (MMU) with features to expedite virtual memory processing (Hennessy and Patterson, 2011).

3.1.2 Page size

Virtual memory is almost always implemented by dividing memory into *pages*: contiguous ranges of memory that embody the physical/logical mapping. Hardware has long supported paging through a *translation lookaside buffer* (TLB) that caches virtual-to-physical mappings and which supports a limited number of page sizes. In modern systems, the default has been four-kilobyte pages – but especially performance-sensitive applications may benefit from other sizes. In particular, two-megabyte and one-gigabyte page sizes are supported by some processors and operating systems, and are often used in HPC to improve the virtual memory performance by reducing turnover in the TLB.

3.1.3 NUMA

Processors may be able to access some memories (e.g., those which are physically closer) faster than others; this arrangement is referred to as non-uniform memory access (NUMA) (Cox and Fowler, 1989). Many NUMA systems strive to transparently deliver functionality and performance to software, but nearly all can be more effectively used with some level of application awareness; for example, some HBM and SPM are reconfigurable at boot time to change coherence, or behave like caches (Sodani et al., 2016), which will affect the NUMA topology of the system.

3.1.4 Further diversification

System memory continues to evolve in capacity and performance as well as in functionality. Error-correcting code (ECC) memory has been in use for decades, as has registered or buffered memory, and these will continue to develop. NVRAM is being made available in a form that resembles the DRAM that serves as main memory in most modern systems (Coughlin, 2016).

The topic of memory has become much more complex since the early days of computing. As physical barriers to capacity and performance are only overcome slowly and user needs diversify, further specialisation is likely: complexity is likely to continue growing, and it is critical that operating systems and programmer interfaces advance to expose these developments.

3.2 Software interfaces to memory

Contemporary systems support the simultaneous execution of multiple processes; naturally this requires arbitration of the system's resources, which is done at an operating system or driver level that carries some elevated privilege. For memory, this privileged layer is responsible for creating and maintaining the page tables on the system, awarding and reclaiming pages on behalf of user-level processes, and enforcing administrative policies on the system.

Atop this privileged layer, user-space libraries and language facilities provide fine-grained allocation/de-allocation routines for use by application developers. This user-space layer brokers with the privileged layer for available memory resources and deals with other user requirements (e.g., thread safety, alignment).

From a programmer's perspective, the manner in which storage is specified is relevant to how it is allocated and where it ultimately comes from. Binary executable formats have allowed for *data segments* where *static* storage – tables known at compile-time, or global variables with known, fixed size – may be placed; these are given backing in memory by the program loader (ld in Linux).

In systems with memories not accessible from the program loader (e.g., SPMs), allocation of *static* storage into those memories can be distinguished between compile-time and runtime methods (Angiolini et al., 2004; Avissar et al., 2001; Avissar et al., 2002; Kandemir et al., 2001; Steinke et al., 2002; Udayakumaran and Barua, 2003). Compile-time/static allocation cannot be changed after initialisation. To find an efficient solution, it may be modelled as a knapsack problem and approximate greedy algorithms can be employed to find good arrangements in modest amounts of time. Dynamic methods are those which can change scratchpad allocations during run-time (Steinke et al., 2002; Udayakumaran and Barua, 2003). For example, the method in Udayakumaran and Barua (2003) allocates global and stack data to SPMs dynamically, with explicit compiler-inserted copying code that copies data between slow memory and SPM when profitable. All dynamic data movement decisions are made at compile-time based on profile information. However, profiling-based compiler analysis methods for the utilisation of such memories are limited in general by many unknown program factors, such as: the input dataset for profiling, unknown dataset size, and memory access patterns.

Additionally, most high-level programming languages use a *stack* to pass arguments to and receive return values from function calls, and for temporary storage of *automatic* variables. Stacks take advantage of known usage scopes to use simple last-in, first-out (LIFO) ordering. Finally, *heap* allocation allows for flexible (but more costly) storage without assuming any size or ordering. Due to performance considerations like cache-line granularity and vector instruction sets, programmers are often interested in specifying how some storage is aligned in memory, which requires special attention during allocation.

Numerous languages, particularly high-level ones (e.g., C#, Python, R, MATLAB, Scheme, Ruby) do not expose the details of allocation and memory references to the user and instead rely on a fully automatic allocation/reclamation mechanism known as *garbage collection*. The opaque nature of these systems makes them difficult to customise in the face of multiple memory types, but there has been extensive research on how to improve their performance, particularly with concurrent execution.

3.2.1 System-level memory management

Operating systems expose and arbitrate hardware resources, like memory, to users in a safe and efficient fashion. To do so, an operating system must be able to discover the types of memory present and their arrangement in the system; a common mechanism is to use the advanced configuration and power interface (ACPI) (Unified EFI, 2016) standard, which defines a manner for discovery of memory types and configurations. Some aspects of memory systems are configurable; for example, the Linux kernel allows available page sizes to be set at boot time or at runtime (by privileged users) (hugetlb, 2016); the same is true for other policies such as process memory limits and page reclamation strategies. In an example of a system call that the user might call directly to get storage, Linux-specific extensions to `mmap` allow the user to specify backings of a particular page size.

With finite resources and multiple competing processes, operating systems must also grapple with issues of resource exhaustion and oversubscription. Some of this is addressable by system policies, but the general rule has been first-come, first-serve; in any case, the potential for failure to acquire a specific type of memory highlights the need for robust fallback mechanisms. For accelerators like GPUs, a device driver typically handles these same issues, although the specialised nature of these components allows some simplification compared to general-purpose operating systems.

3.2.2 Language-level interfaces

3.2.2.1 Language support

Many programming languages require that the user perform memory management with facilities provided by the language: C++'s `new/delete` operators and Fortran '90s `ALLOCATE/DEALLOCATE` statements are familiar examples of this. Notably, C's `malloc` function (see Figure 1) is not part of the language itself, but provided by the Standard Library. These built-in memory facilities were designed decades ago and are not prepared to express the variety of configurations possible today; vendor-specific extensions such as Intel's `_mm_malloc` and newer language standards (e.g., C11, C++17) overcome some of these limitations, but are focused on providing aligned allocations (see Figure 2).

Figure 1 Example usage of C's malloc function (see online version for colours)

```
double *a = (double *) malloc(N*sizeof(double));
```

Figure 2 64-byte-aligned allocation of *N* doubles in C (top) and C++ (bottom) (see online version for colours)

```
double *a = (double *) aligned_alloc(64, N*sizeof(double));
double *b = new (std::align_val_t(64)) double[N];
```

3.2.2.2 Directives and accelerator languages

Directive-based programming with languages like OpenMP (OpenMP Architecture Review Board, 2015) and OpenACC (OpenACC Working Group et al., 2015) has become a popular method for augmenting traditional languages to take advantage of hardware developments, from multicore systems and new vector instructions to compute accelerators. Other languages like OpenCL* (Stone et al., 2010) and CUDA* (Nickolls et al., 2008) fulfil similar roles.

Both directives and languages typically provide host-based functions for allocating space on the device, and some also allow for allocation from within kernels (e.g., offload to Intel® Xeon Phi™ coprocessors, recent CUDA devices). A mechanism is provided for indicating which memory space (i.e., host or device) variables are bound to and for transferring data to and from the host, and some systems and programming environments [such as MYO (Jeffers and Reinders, 2013) and NVIDIA* Unified Memory (Harris, 2013)] provide a unified virtual address space across devices.

Some of these languages offer support for how memory is used/allocated; CUDA supports the `__shared__` keyword to declare that storage should be placed in the SPM (OpenCL uses the `__local` keyword for the same purpose). CUDA and OpenCL also allow the user to designate storage as belonging to a GPU's *texture* memory through a special templated `texture` data type, with separate allocation, access, and binding procedures.

The Kokkos library (Edwards et al., 2014) defines *memory spaces* that are associated with *execution spaces*. The relationship between a memory space and an execution space aid the library in determining memory layout and how to transfer data between devices (if necessary), and memory spaces can be given attributes that help direct the type of underlying memory (e.g., DDR, HBM) that Kokkos should allocate.

3.2.3 Library interfaces

There have been numerous efforts to produce software libraries to expose functionality and provide specialised behaviour for memory systems. libhugetlbfs (Gibson and Litke, 2007) is a Linux library that provides user-level access (in both library and runtime-utility capacities) to non-standard page sizes; libnuma (Kleen, 2005) similarly provides a mechanism for querying memory locations (*NUMA nodes*) and for specifying user-level preferences about page placement. The memkind library (Cantalupo et al., 2015) builds upon NUMA node functionality to enable allocations of HBM within C code as a supplement to the traditional `malloc`, and additionally supports selection of page size and alignment (see Figure 3).

3.2.4 Heap allocation

In addition to libraries that give the user access to non-standard memory, there have been countless efforts to build specialised user-space heaps. Hoard (Berger et al., 2000), jemalloc (Evans, 2006), and TCmalloc (Ghemawat and Menage, 2009) are all drop-in replacements for libc's `malloc` that aim to improve performance for a range of applications. The Boost family of C++ libraries has specialised allocators (Boost Community, http://www.boost.org), and the Intel® Thread Building Blocks package includes heap allocation utilities designed for performance under concurrency (Reinders, 2007). The POSIX standard provides the `posix_memalign` function for aligned heap allocations.

Figure 3 Example usage of the memkind library to request a 64-byte aligned allocation of *N* doubles backed by 2M pages in MCDRAM, with a silent fallback to DDR (see online version for colours)

```
double* a;
int error = memkind_posix_memalign(MEMKIND_HBW_PREFERRED_HUGETLB,
                                   &a,
                                   64,
                                   N*sizeof(double));
```

3.3 Memory resource sharing

Just as operating systems arose to manage increasingly complex hardware for a growing set of users and uses, so do libraries present new challenges in the face of more parallelism and memory types. When applications delegate certain roles to libraries, the question about what memory that library should use for intermediate storage arises. Some libraries provide a global interface that allows for a user-defined allocation routine [for example, SQLite may be configured via `sqlite3_config` (Owens and Allen, 2010)], and others allow POSIX-shell based environment variables to control allocation behaviour (as in the case of `MKL_FAST_MEMORY_LIMIT`, from the Intel® Math Kernel Library). These interfaces are far from uniform, and many libraries provide no mechanism for customising allocation behaviour at all.

4 Design principles

As shown previously, the state of the art in memory management is decades-old standard utilities augmented by a combination of platform-specific libraries and proprietary language extensions. As a result, maintaining separate code paths for multiple platforms – and for each kind of memory available on each of those platforms – is becoming increasingly more burdensome for developers; there is a clear need for a platform-agnostic interface for managing the increasing complexity in memory subsystems.

4.1 Requirements

The platform-agnostic interface presented in this paper is designed around four main requirements:

1 As many existing memory kinds and platform configurations as possible (ideally, all of them) should be supported.

2 Extensions to future memories and platforms should not require significant modification (of either the interface itself or codes which use it).

3 Each of the different types of memory that a programmer can declare (e.g., static, stack and heap memory) should be supported.

4 The needs of application developers with different performance, portability and productivity trade-offs should be met, by providing both high- and low-level interfaces where appropriate.

In the remainder of this section, we detail how these requirements have influenced our design decisions; the proposed syntax and semantics of the interface itself are discussed in Sections 5 and 6.

4.2 Design

4.2.1 Platform support

The most user-friendly way to support multiple platforms is to provide a run-time mechanism for querying the kinds of memory that are available, along with well-defined error and fallback behaviours when memory of a particular kind cannot be allocated (e.g., because no such resource exists, or because the resource is exhausted). However, a pure run-time mechanism cannot handle the reality that several of today's platforms require special instructions to access or modify data in certain memories (e.g., because they are a separate address space). In order to ensure support for such memories, our proposal includes a combination of run-time queries (to direct allocations) and compile-time hints (to direct code generation).

With regards to support for future memories, we fully anticipate that hardware will develop at a faster rate than any software standard can match, and believe that any attempt to pre-empt the behaviours and properties of new memories would undoubtedly fail to capture everything. As such, we have tried not to unnecessarily restrict our interface based on the properties of today's platforms – our proposal permits vendor extensions to the standard (as in OpenGL/OpenCL) to support functionality that is too new (or too proprietary) to be reflected by the standard.

4.2.2 Usability

A mixture of run-time and compile-time information is required to handle the different types of memory allocations that already exist in applications: we propose a familiar (i.e., `malloc-like`) interface to handle dynamic (heap) memory allocations, and compiler directives to handle static and stack allocations. This aligns well with our platform support strategy, as static and stack allocations are among the most likely candidates to be allocated in memories requiring special instructions (e.g., constant and SPMs on GPUs).

Our proposal also separates the concepts of *where* an allocation happens (i.e., the kind of memory it is) from *how* an allocation happens (e.g., whether the memory comes from a pool or from the OS heap). This separation allows our proposal to provide a standard interface to two different kinds of memory management optimisation: allocation of fast memory (e.g., MCDRAM); and fast allocation of memory [e.g., jemalloc (Evans, 2006)].

4.2.3 Legacy considerations

It is tempting to assume that all applications will be re-written to our new interface. However, we acknowledge that this is unrealistic: the maintainers of legacy applications are unlikely to want to change much of their code, and the maintainers of libraries are unlikely to want to change their public interfaces. Our proposal makes a number of allowances for such situations, but these are not intended to be as expressive or to provide the same level of performance as the rest of the interface.

4.2.4 Extending OpenMP

The OpenMP specification has grown from an abstraction of thread-level parallelism to incorporate tasking, SIMD (vector) parallelism, and offloading code regions to remote targets. Many of these features introduce additional (and often implicit) memory allocations, including: private copies of variables allocated for each thread/task/SIMD lane; temporary variables used to implement reductions; and a combination of local and remote variables used to facilitate offload.

How and where such additional variables are allocated can have a significant impact upon application performance, but this is not something that is currently exposed to the programmer at a source code level – the only way in which we can hope to provide a memory interface that that affects these allocations is by extending the existing OpenMP constructs. Designing our interface as an OpenMP extension has also compelled us to consider platforms and corner-cases that we may otherwise have overlooked, increasing our confidence that we have covered all important aspects of modern memory management.

5 Basic syntax and semantics

Our proposal is based on two new concepts: *memory spaces* and *allocators*. A memory space refers to a memory resource available in the system at the time the OpenMP program is executed, such that each space has certain characteristics depending on the kind of physical memory. An allocator is an object that allocates (and frees) memory from an associated memory space, which is bound to the allocator during its creation. The allocation mechanisms and the other functionality of this proposal are built around these two concepts.

This section presents what we consider the base concepts and functionality of our proposal. We have presented this to the OpenMP Language Committee, which has led to the publication of Technical Report 5 (TR5) (Duran et al., 2017) as a statement on the future direction of the development of OpenMP. Functionality beyond this initial proposal is the subject of technical discussions; we present this material in Section 6. These future features are desirable to improve programmer productivity, but most could be built on top of the minimum set presented here.

Although our full proposal as outlined in the TR5 document contains support for both C/C++ and Fortran, we show only C/C++ syntax throughout this section because of space constraints. Likewise, we omit proper checking of return values from our code examples to keep them simple; the reader should assume that most API calls return `NULL` when an operation cannot be completed.

5.1 Key concepts

In this section, we describe the concepts of memory spaces and allocators in detail, along with their configuration options (traits).

5.1.1 Memory spaces

Memory spaces, represented by the `omp_memspace_t` (C/C++) datatype, encapsulate a storage resource that is available in the system. Contemporary HPC server systems are usually equipped with some DDR-based main memory, which could be the only memory space in a system. Memories with enhanced performance (e.g., HBM) or functionality (e.g., non-volatile memory) appear as additional memory spaces.

Memory traits, represented by the `omp_memtrait_t` (C/C++) datatype, define the characteristics of memory spaces. Memory traits may be combined into a set to allow for queries, identification and description of the different memory spaces of a system. Memory traits can either be *prescriptive*, meaning an exact match is required, or *descriptive*, meaning the runtime is requested to select the optimal type of memory based on the requested properties. Traits with the matching rule = (or ≥) match if the value of the trait in the memory space equals (or exceeds) the value in the list, while traits with the matching rule ≈ match if the value of the trait in the memory space is closest to the value in the list (compared to all available memory spaces). The matching process first considers traits with the = and ≥ rules, and from the candidates those that match the ≈ rule.

A single system resource may appear as multiple memory spaces, but with different characteristics. For instance, the DDR-based main memory may appear as one memory space with 4K page size and another with 2M page size. In every system, at least one memory space which maps to the default memory resource.

The list of traits in Table 2 might not be sufficient to describe the full variety of memories, but can be extended in the future to provide additional functionality or to accommodate vendor-specific traits and values. For use of the traits and values in the API calls, TR5 defines corresponding constants with the prefix `OMP_MTK_` (memory trait keys) and `OMP_MTV_` (memory trait values).

5.1.2 Allocators

An *allocator*, represented by the `omp_allocator_t` (C/C++) datatype, is an object that performs allocations of contiguous memory chunks from a given memory space. *Allocator traits*, represented by the `omp_alloctrait_t` (C/C++) datatype, are analogous to memory traits and can be employed to specify different aspects of an allocator's behaviour. Table 3 lists the base set of allocator traits that are part of this proposal.

Table 2 Set of proposed memory traits; = requires strict matches, while ≥ matches greater-than-or-equal-to (numerical values only)

Trait	*Matching*	*Values*	*Description*
`distance`	≈	`near`, `far`	The relative physical distance of the memory space with respect to the task the request binds to.
`bandwidth`	≈	`highest`, `lowest`	The relative bandwidth of the memory space with respect to other memories in the system.
`latency`	≈	`highest`, `lowest`	The relative latency of the memory space with respect to other memories in the system.
`location`	≈	`core`, `socket`, `device`	The physical location of the memory space.
`optimised`	=	`bandwidth`, `latency`, `capacity`, `none`	Specifies if the memory space's underlying technology is optimised to maximise a certain characteristic. The exact mapping to actual technologies is implementation defined.
`pagesize`	=	`positive integer`	The size of the pages used by the memory space.
`permission`	=	`r`, `w`, `rw`	Specifies if read (r), write (w) or both (rw) operations are supported by the memory space.
`capacity`	≥	`positive integer`	The physical capacity [bytes] of the memory space.
`available`	≥	`positive integer`	The current available capacity [bytes] for new allocations in the memory space.

Notes: ≈ performs a nearest match. = and ≥ traits are always satisfied before ≥.

Figure 4 Example of memory space and allocator creation (see online version for colours)

```c
void *omp_alloc(size_t size, omp_allocator_t *allocator);
void *omp_free(void *ptr, omp_allocator_t *allocator);
omp_memtrait_t ddr_2m[2] = {{OMP_MTK_OPTIMIZED, OMP_MTV_NONE},
                            {OMP_MTK_PAGESIZE, 2*1024*1024}};
omp_memtrait_t hbm_2m[2] = {{OMP_MTK_BANDWIDTH, OMP_MTV_HIGHEST},
                            {OMP_MTK_PAGESIZE, 2*1024*1024}};

omp_memtrait_set_t mtraits_ddr_2m;
omp_memtrait_set_t mtraits_hbm_2m;

omp_init_memtrait_set(&mtraits_ddr_2m, 2, ddr_2m);
omp_init_memtrait_set(&mtraits_hbm_2m, 2, hbm_2m);

omp_memspace_t *memspace_DDR_2M = omp_init_memspace(2, mtraits_ddr_2m);
omp_memspace_t *memspace_HBM_2M = omp_init_memspace(2, mtraits_hbm_2m);

omp_alloctrait_t atrait_list1[2] = {{OMP_ATK_ALIGNMENT, 64},
                                    {OMP_ATK_FALLBACK, OMP_ATV_ABORT}};
omp_alloctrait_set_t atraits_ddr;
omp_init_alloctrait_set(&atraits_ddr, 2, atrait_list1);
omp_allocator_t *alloc_DDR_2M = omp_init_allocator(memspace_DDR_2M, &atraits_ddr);

omp_alloctrait_t atrait_list2[3] = {{OMP_ATK_ALIGNMENT, 64},
                                    {OMP_ATK_FALLBACK, OMP_ATV_ALLOCATOR},
                                    {OMP_ATK_FBDATA, alloc_DDR_2M}};
omp_alloctrait_set_t atraits_hbm;
omp_init_alloctrait_set(&atraits_hbm, 3, atrait_list2);
omp_allocator_t *alloc_HBM_2M = omp_init_allocator(memspace_DDR_2M, &atraits_hbm);
```

Table 3 Set of proposed allocator traits

Trait	*Values*	*Default*
`threadmodel`	`shared`, `exclusive`	`shared`
`alignment`	integer: 0, or power of 2	0
`pinned`	`true`, `false`	`false`
`fallback`	`null_fb`, `abort_fb`, `allocator_fb`, `default_fb`	`default_fb`
`fb_data`	an allocator handle	-

The `threadmodel` and `fallback` traits require some additional explanation. Setting the threadmodel trait to `exclusive` guarantees that the allocator will never be called by more than one thread at a time, which an implementation could exploit to optimise internal operations (e.g., with reduced locking). The fallback behaviour becomes relevant if the allocation cannot be performed successfully from the given memory space – for instance, if an allocation requests an amount of high bandwidth memory that exceeds the amount of available memory in the corresponding memory space. Our proposal defines four different fallback behaviours for when an allocator fails to allocate memory:

- `null_fb`: The allocator returns the value 0 (i.e., `NULL`). This is similar to the behaviour of `malloc`.
- `abort_fb`: The allocator aborts program execution.
- `allocator_fb`: The allocator forwards the allocation request to another allocator, specified by `fb_data`.
- `default_fb`: The allocator attempts to allocate memory in the memory space that maps to default memory.

As with memory traits, the list of allocator traits in Table 3 is intended to be extensible to meet future needs. For use of the traits and values in the API calls, TR5 defines corresponding constants with the prefix `OMP_ATK_` (allocator trait keys) and `OMP_ATV_` (allocator trait values).

5.2 Creating/destroying allocators and memory spaces

A set of APIs to create and destroy memory spaces and allocators is part of the proposal. The corresponding API functions follow the naming conventions of other OpenMP types and API parts (e.g., locks); the data structure must be initialised before first use with an `omp_init_` function and must be destroyed after last use with an `omp_destroy_` function.

The example in Figure 4 shows how this API can be used to create an allocator that performs aligned allocations backed by 2M pages from high bandwidth memory if possible (`allocator_HBM_2M`). If not, the request is forwarded to an allocator that performs aligned allocations backed by 2M pages from DDR memory (`allocator_DDR_2M`). If the allocation cannot be served by either allocator, program execution is aborted.

5.3 Allocation API

Two new API calls are provided to allocate and deallocate memory using an allocator: `omp_alloc` and `omp_free`. These functions have the following prototypes.

Users request an allocation of a certain size from a given allocator by means of `omp_alloc`. The memory must be deallocated with `omp_free`, and both the pointer and the allocator must match those provided to `omp_allocate` previously. Figure 5 shows an example of this API, illustrating that our proposal allows to pass allocators as arguments throughout the program.

Note that in this example code, the memory in which the allocations will be performed is not exposed to the function; which memory these allocators will use, or even if the two allocators will be the same, can be decided by the caller of the function. This is enabled by the separation of the definition of memory spaces, allocators, and the actual allocation. Moving from one system to another only requires, at most, modification of the allocator definitions and not the allocation sites themselves.

Figure 5 Example of the proposed allocation API (see online version for colours)

```
void foo(omp_allocator_t *bigobj_alloc,
         omp_allocator_t *smallobj_alloc)
{
    double *pd = (double *) omp_alloc(sizeof(double) * N, bigobj_alloc);
    float *pf  = (float  *) omp_alloc(sizeof(float)  * N, smallobj_alloc);
    ...
    omp_free(pf, smallobj_alloc);
    omp_free(pd, bigobj_alloc);
}
```

Figure 6 Example usage of the allocate directive (see online version for colours)

```
MyType a[M];
#pragma omp allocate(a) memtraits(optimized=none, pagesize=1G)

void foo(omp_allocator_t *allocator)
{
    double b[M],c[M];
    static AnotherType d;

    #pragma omp allocate(b) allocator(allocator)
    #pragma omp allocate(c) memtraits(bandwidth=highest, pagesize=2M)
    #pragma omp allocate(d) alloctraits(alignment=64)

    ... // use of a,b,c & d

    // b and c are automatically deallocated here
}
```

5.4 The `allocate` directive

The new `allocate` directive enables developers to change the allocation process of variables without having to use the API. This is particularly useful for variables not allocated on the heap (e.g., stack variables). In Fortran, this directive provides the additional functionality to work on variables declared as `ALLOCATABLE`.

The syntax for the allocate directive is as follows:

(see online version for colours)

```
#pragma omp allocate (list)[clauses]
```

with clauses being:

- `allocator` (user-constructed-allocator)
- `memspace` (user-constructed-memspace)
- `memtraits` (memory-traits-list)
- `alloctraits` (allocator-traits-list).

The storage locations of all variables specified in the associated list will be overridden by memory returned by an allocator. If this allocator is *explicit* (i.e., one constructed by the programmer), it is specified by means of the allocator clause. Otherwise, an *implicit* allocator is constructed using the specified memory traits, memory space or allocator traits. At least one of the clauses must be provided by the user.

The `allocate` directive is a *lexically scoped* directive – the scope of the allocation is that of the variable that it affects, and the implementation will deallocate the storage that the allocate directive provided when the variable would normally go out of scope. Figure 6 shows examples of the usage of the `allocate` directive, where a global variable (`a`) and three local variables (`b`, `c`, `d`) are affected. The variables `b` and `c` have automatic storage and are therefore deallocated at the end of the function scope. Variable `b` is allocated through an explicit *allocator*, while all others are allocated using implicit allocators based on the specified `memtraits` and `alloctraits`. The application of the allocate directive to a variable may imply that a variable that would otherwise not been allocated from a heap be allocated from one; this depends on what the underlying implementation supports (e.g., linker support) and the user should not assume any particular behaviour.

An additional form of this directive is provided for Fortran programs, allowing the programmer to modify the implicit behaviour of `ALLOCATE` statements and redirect the internal allocation to an allocator-based one.

5.5 The allocate clause

OpenMP specifies that additional storage needs to be provided for data-sharing clauses that involve privatisation (e.g., `private`, `reduction`, `linear`) and the `threadprivate` directive. This storage cannot be directly referred to by the programmer, so the two previous mechanisms cannot be used to affect which storage is used. The aim of the allocate clause is to provide that functionality. The syntax for the allocate clause is as follows:

```
allocate([modifiers:]list)
```

where *modifiers* is a comma separated list of elements that are either:

- `allocator` (*user-constructed-allocator*)
- `memspace` (*user-constructed-memspace*)
- `memtraits` (*memory-traits-list*)
- `alloctraits` (*allocator-traits-list*)
- `safe_align` (*alignment*).

Figure 7 Example usage of the *allocate* clause (see online version for colours)

```
void f(double a, omp_allocator_t *myalloc)
{
    double b[M];
    some_type_t c;

    #pragma omp parallel \
            private(a,b) allocate(memtraits(bandwidth=highest,pagesize=2M):a,b) \
            firstprivate(c) allocate(allocator(myalloc):c)
    {
        // Here the private copies of 'a' and 'b' are
        // allocated in 2M pages in HBM memory
        // The private copy of 'c' is allocated
        // by means of the 'myalloc' allocator
    }
}
```

Figure 8 Default allocator API prototype (see online version for colours)

```
void  omp_set_default_allocator ( omp_allocator_t *allocator );
```

When the allocate clause is specified, any additional storage required by OpenMP will be obtained by means of an explicit or implicit allocator (depending on *modifiers*). The storage will automatically be deallocated when the additional storage would have been normally deallocated as specified by OpenMP. Figure 7 shows an example using the `allocate` clause.

5.6 Default allocators

For cases in which a single allocator is sufficient for the whole program or a long program section, we introduce a new OpenMP per-thread *internal control variable* (which is inherited by spawned teams) that points to an allocator. The API call `omp_set_default_allocator` is used to set the thread default allocator. When the allocator argument of the `omp_alloc` and `omp_free` routines is set to `NULL` then the default allocator in the internal control variable of the calling thread is used. Figure 8 shows how the default allocator can be set with the API call and used from the allocation routines.

Figure 9 Example usage of default allocators (see online version for colours)

```
void fixed_interface()
{
    float *ptr = omp_alloc(N*sizeof(float),NULL);
    ...
    omp_free(ptr,NULL);
}

void foo()
{
    omp_allocator_t *hbm_allocator = ...;
    omp_allocator_t *ddr_allocator = ...;

    // allocation happens in HBM memory
    omp_set_default_allocator(hbm_allocator);
    fixed_interface();
    // allocation happens in DDR memory
    omp_set_default_allocator(ddr_allocator);
    fixed_interface();
}
```

The default allocator is also useful to overcome the situation where for practical reasons (e.g., backward compatibility) an existing library interface cannot be extended to support an allocator argument. In these cases, the *internal control variable* can be used as a *hidden* argument, as shown in Figure 9 where the first call to `fixed_interface` allocates in HBM memory while the second call allocates in DDR memory.

Additionally, our proposal contains a new environment variable, `OMP_ALLOCATOR` that allows the programmer to use memtraits and allocator traits to define the *default allocator* of the initial thread of an OpenMP application.

6 Advanced features

This section presents functionality that furthers memory management support in OpenMP, but which goes beyond the set of features published in TR5.

6.1 Named allocators

TR5 supports a single allocator that can be externally modified (see Section 5.6); we believe it would be very useful to support multiple such allocators that could be individually modified to adapt (or experiment with) different parts of an application.

We propose to extend the syntax for `OMP_ALLOCATOR` to enable users to define new *named allocators*, a pointer to which can be obtained via a call to `omp_get_allocator`. The call `omp_set_allocator_default` can be used to set a default overridable value.

An example of how this API would be used is shown in Figure 10, where an allocator named `MATRIX_ALLOCATOR` is used. Figure 11 shows how the policy of the allocator can be changed by means of the `OMP_ALLOCATOR` environment variable.

6.2 Pre-defined allocators

The memory spaces and allocators API proposed in TR5 allows advanced users to create complex memory management policies, but may be considered too cumbersome an interface for average users. To ease this, we propose to provide a list of standard predefined allocators that an OpenMP implementation can map to different allocation strategies on different systems, providing users with a set of allocator behaviours that they can rely on across different runtimes. In cases where the user believes that they can provide a better allocation strategy than a runtime's default mapping, pre-defined allocators can be overridden in the same way as *named* allocators (i.e., using `OMP_ALLOCATOR`).

A possible – by no means exhaustive – list of pre-defined allocators is provided in Table 4, along with a description of the expected behaviour of each allocator. These allocators could then be used directly with the `omp_alloc` call.

Table 4 Proposed list of pre-defined allocators

Name	*Description*
`default_mem`	Returns an allocation from the default memory of the system.
`hbw_mem`	Returns an allocation from high-bandwidth memory (if available).
`large_mem`	Returns an allocation from a large capacity memory.
`local_mem`	Returns an allocation from a memory that is close (or even private) to the thread.

Figure 10 Example of the API for *named allocators* (see online version for colours)

```
some_allocator = ...
omp_set_allocator_default("MATRIX_ALLOCATOR", some_allocator);
...
omp_allocator_t *matrix_allocator = omp_get_allocator("MATRIX_ALLOCATOR");
void *p = omp_alloc(N, matrix_allocator);
```

Figure 11 Example definitions of *named allocators* using `OMP_ALLOCATOR` (see online version for colours)

```
$ export OMP_ALLOCATOR="MATRIX_ALLOCATOR:bandwidth=highest,pagesize=2M,alignment=4K"
$ export OMP_ALLOCATOR="MATRIX_ALLOCATOR:optimized=latency,distance=near,alignment=64"
```

Figure 12 Example of the *distribution* trait and *merged memory spaces* (see online version for colours)

```
hbw_ms = ...
def_ms = ...

aggr_ms = omp_merge_memspaces(hbw_ms,def_ms);

omp_alloctrait_t alist[2] = {{OMP_ATK_DISTRIBUTION,        OMP_ATV_INTERLEAVED},
                             {OMP_ATK_DISTRIBUTION_FACTOR, 4}};
omp_alloctrait_set_t atraits;
omp_init_alloctrait_set(&atraits, 2, alist);
omp_allocator_t *aggr_allocator = omp_init_allocator(aggr_ms, &atraits);

void *p = omp_alloc(N,aggr_allocator);
```

Figure 13 Example usage of the proposed support for OpenMP devices (see online version for colours)

```
int              device       = 0;
omp_memtrait_t   mtrait       = {OMP_MTK_BANDWIDTH, OMP_MTV_HIGHEST};
omp_memspace_t  *device_ms    = omp_target_init_memspace(device, 1, &mtrait);
omp_allocator_t *device_alloc = omp_target_init_allocator(device, device_ms, 0, 0);

// a is allocated on the device using the device_alloc allocator
double *a = omp_target_ext_allocate(device_alloc, N*sizeof(double));
double b[N];

// allocate modifier changes allocation of b in the device
#pragma omp target map(from:b) allocate(allocator(device_alloc))
{
    ...
}
```

6.3 Distributed allocations

In many modern systems, users may desire an allocation that spans multiple resources. This scenario is most commonly encountered on NUMA systems, where the same kind of memory is present in several different physical locations, but other cases are arising – for example, a programmer may want an object to span multiple different kinds of memories to obtain higher aggregate bandwidth. In order to support these scenarios, we propose two new features:

1 a distribution allocator trait

2 merged memory spaces.

The `distribution` allocator trait specifies how allocations should be distributed across the different storage resources that a memory space represents. Table 5 defines the different values that the trait can take; the `distribution-factor` auxiliary trait is used for distribution policies that need an additional argument. The runtime should make a best effort attempt to match the requested distribution policy, but the underlying memory resources might not support the requested distribution.

Table 5 Possible values of the `distribution` allocator trait

Value	*Description*
`local`	Allocations should be in a resource close to the thread.
`static`	Allocations should be partitioned into pieces of approximately the same size, one for each resource.
`interleaved`	Allocations should be partitioned into pieces of `distribution-factor` pages and these pieces should be interleaved across the different resources.
`remote`	Allocations should be in a resource far from the thread.

Merged memory spaces represent a combination of two or more memory spaces. To support the creation of merged memory spaces, we propose a new API call, `omp_merge_memspaces`, that merges a set of obtained memory spaces into a single new memory space. Figure 12 shows an example of how this API call can be combined with the `distribution` trait to specify an allocator that distributes objects between HBM and the default memory space.

Whether the memory spaces in NUMA systems will appear initially as merged memory spaces that users must specify distribution policies for, or as separate memory spaces that users can choose to merge together (or not), are both valid and desirable runtime behaviours. The syntax proposed here is compatible with either option, and we expect that OpenMP runtimes may provide some high-level switch (e.g., an environment variable) to toggle between different handling of NUMA memory spaces.

6.4 Device support

Starting with Version 4.0 of its specification, OpenMP supports devices (e.g., an accelerator or coprocessor) attached to the host. This support includes data-mapping clauses that create copies of host variables on attached devices, and API calls (e.g., `omp_target_alloc`) that allow allocation of device memory directly from host threads. TR5 does not yet support mechanisms that allow the use of *allocators* in either case.

In order to support the data-mapping clauses, we propose to allow the allocate modifier in the map clause, with the effect of using an *allocator* on the device instead of on the host. To support allocations from host threads, we require a second set of APIs; we propose an API identical to that described for the host, except having the name extended with the target key, and with an additional parameter specifying the *device number* where the operation should happen. For example, `omp_init_memspace` becomes `omp_target_init_memspace` with an additional device number parameter. The only exceptions to this rule are `omp_allocate` and `omp_free`, which would collide with existing APIs – for this reason, we have named their target versions `omp_target_ext_allocate` and `omp_target_ext_free`. While we could have provided a single combined API across host and device, we felt that an extra (host/device) parameter would be a burden to the many OpenMP users that do not use the offloading constructs of OpenMP.

Figure 13 shows example usage of the proposed device support, where a remote *allocator* is initialised and then used both through the `omp_target_ext_allocate` call and the allocate modifier in the clause to perform allocations in the memory with the highest bandwidth of the device.

6.5 Support for specialised instructions

Some memories require special handling by the compiler during code generation: special instructions may be needed for read/write operations to the memory (e.g., constant and texture memory) or to access some special capability (e.g., to commit memory to persistent storage).

So far, we have focused on support for dynamic decisions regarding choice of memory resource. Supporting memories that require special handling requires some amount of static knowledge, as the compiler needs to know which memories might be used – however, it is still desirable to support a reasonable amount of dynamic behaviour, in order to avoid forcing programmers to compile separate binaries for each memory configuration of interest. To achieve this, we have divided our proposed support into two new directives:

1 a `declare version` directive that handles the static portion (related to code generation)

2 a `dispatch` directive that handles dynamic behaviour.

6.5.1 The declare version directive

OpenMP already has a directive that directs compilers on how to generate specific code in the form of the `declare simd` directive. We believe that instead of adding a specific directive for the memory code generation (e.g., a `declare allocation` directive) it is better to introduce a new directive that allows composition of current and future properties that can affect code generation. Our proposed `declare version` directive is applied to a function declaration and, for each directive applied to a given function, a specialised version of the function might be generated by the compiler. The syntax of the declare version directive is as follows:

(see online version for colours)

```
#pragma omp declare version[(version-name)][clauses]
```

where supported *clauses* are:

- `alloc_def` (*allocate-specification:argument-list*)
- `simd_def` (*simd-specification:argument-list*)
- `implements` (*orig-name*).

The `alloc_def` clause instructs the compiler to assume that the specified *arguments* are allocated through an allocator that matches the *allocate-specification*. The `simd_def` clause allows the programmer to specify SIMD properties of the arguments with the same level of support as the existing declare `simd` directive (e.g., `uniform`, `linear`). Additionally, the arguments in both lists can be modified with the existing ref and val modifiers, to specify whether the specification refers to pointer or pointee for pointers and references.

Figure 14 Usage example of the version and dispatch directives (see online version for colours)

```
#pragma omp declare version(foo_lat) alloc_def(optimized=latency:v)
#pragma omp declare version(foo_hbw) alloc_def(optimized=bandwidth:v)
void foo(double *v) { ... }

void bar(double *a)
{
    double b[N];
    #pragma omp allocate(b) memtraits(optimized=bandwidth)

    #pragma omp dispatch(b)
        foo(b); // compiler calls foo_hbw

    #pragma omp dispatch(a)
        oo(a); // compiler has no information
                // about 'a':uses dynamic dispatch
}
```

With this information the compiler can generate a specialised version of the function if necessary (in a given platform there may be no need of specialised code); this version of the function will be called *version-name* if specified, and will have an anonymous name otherwise. If the implements clause is specified, then the user is asserting that the annotated function implements a specialised version of the *orig-name* function that complies with the `alloc_def` and `simd_def` specifications. Multiple `declare version` directives can be applied to a single function.

6.5.2 *The `dispatch` directive*

In a program where multiple versions of a function are present, the correct one must be called depending on the memory kind of its arguments. In some cases, the compiler is able to statically generate the call to the proper function, such as when the call happens in the same scope as a static allocation. In other cases, run-time (dynamic) dispatch is necessary.

For dynamic dispatch to work, memory reflection through a pointer must be supported (i.e., the implementation needs to be able to discover the allocation information of a given pointer). One possible implementation allocates a descriptor at a fixed offset from the base pointer (e.g., in front of the allocated chunk) that contains the information (or a pointer to it). For many cases, the amount of extra memory required by such a descriptor is small and the pointer reflection operation is quite fast. However, there are some corner cases where the amount of wasted memory space is too large (e.g., requesting page-aligned memory in 1G pages or requesting many small objects). An alternative implementation that is based on a hash table may be preferable for such cases, even if it is slower. We envision an environment variable that would allow users to specify if they want the implementation to optimise memory reflection in time or space to cope with this problem. Memory reflection can only be requested on the base pointers of the allocations, as otherwise the implementation of reflection is prohibitively expensive.

Figure 14 shows how the `dispatch` directive can be used to instruct the compiler to call the correct version of the `foo` function. For `foo(b)`, the compiler can deduce from the `allocate` directive that `b` will be in a memory of optimised for bandwidth and replace the call to `foo` with a call to `foo_hbw`. For `foo(a)` the compiler has no knowledge about the allocation of a and therefore must generate dynamic dispatch code that will first lookup the memory traits of a before calling the correct version of `foo`.

7 Summary

In modern multi-/many-core processors and accelerators, effective utilisation of all kinds of memory at different levels in the memory subsystem hierarchy has become even more crucial for achieving close-to-the-metal performance and power efficiency. This paper addresses this challenge through the introduction a new set of directives and runtime APIs to OpenMP, a widely-accepted industry standard for exploiting parallel hardware.

The main contributions of this paper are three-fold:

1 We provide a comprehensive study and analysis of state-of-the-art memory subsystems and memory management software solutions.

2 We introduce two new concepts – *memory spaces* and *allocators* – to represent the storage resources available in a system and the objects that perform allocations from a given memory space. Based on these new concepts, we propose a set of new language extensions and runtime APIs that provide a platform-agnostic interface for managing the increased complexity of modern memory subsystems.

3 We propose additional language extensions that provide compilers and runtimes sufficient information (e.g., memory kind, size, availability, alignment) to allow for optimal code generation and dispatching. These language extensions blend well with the existing OpenMP parallel, SIMD and offloading constructs.

We feel that this proposal is an important step to expanding the utility and flexibility of OpenMP and helps to affirm its status as a platform-independent yet powerful tool for programmers.

We have endeavored to propose a truly novel memory abstraction and API: something that is expressive, extensible, and flexible that nonetheless does not compromise on allowing programmers to express specific needs critical to high-performance applications. This is the first such memory proposal to embrace this vision and we look forward to its adoption and evolution.

Disclaimers

Intel, Xeon, Xeon Phi, 3D XPoint and Iris are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the USA and other countries.

*Other brands and names are the property of their respective owners.

References

Akeley, K. (1993) 'Reality engine graphics', in *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, pp.109–116.

Angiolini, F., Menichelli, F., Ferrero, A., Benini, L. and Olivieri, M. (2004) 'A post-compiler approach to scratchpad mapping of code', in *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ACM, pp.259–267.

Avissar, O., Barua, R. and Stewart, D. (2001) 'Heterogeneous memory management for embedded systems', in *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ACM, pp.34–43.

Avissar, O., Barua, R. and Stewart, D. (2002) 'An optimal memory allocation scheme for scratch-pad-based embedded systems', *ACM Transactions on Embedded Computing Systems (TECS)*, Vol. 1, No. 1, pp.6–26.

Banakar, R., Steinke, S., Lee, B-S., Balakrishnan, M. and Marwedel, P. (2002) 'Scratchpad memory: design alternative for cache on-chip memory in embedded systems', in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, ACM, pp.73–78.

Berger, E.D., McKinley, K.S., Blumofe, R.D. and Wilson, P.R. (2000) 'Hoard: a scalable memory allocator for multithreaded applications', *ACM Sigplan Notices*, Vol. 35, No. 11, pp.117–128.

Boost Community, *Boost C++ Libraries* [online] http://www.boost.org (accessed July 2017).

Borkar, S. and Chien, A.A. (2011) 'The future of microprocessors', *Commun. ACM*, May, Vol. 54, No. 5, pp.67–77, ISSN: 0001-0782, DOI: 10.1145/1941487.1941507.

Cantalupo, C., Venkatesan, V., Hammond, J., Czurlyo, K. and Hammond, S.D. (2015) *memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies*, Technical Report, Sandia National Laboratories (SNL-NM), Albuquerque, NM, USA.

Coughlin, T. (2016) 'Crossing the chasm to new solid-state storage architectures [the art of storage]', *IEEE Consumer Electronics Magazine*, Vol. 5, No. 1, pp.133–142.

Cox, A. and Fowler, R. (1989) 'The implementation of a coherent memory abstraction on a NUMA multiprocessor: experiences with platinum', in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, SOSP '89*, ACM, New York, NY, USA, pp.32–44, ISBN: 0-89791-338-8, DOI: 10.1145/74850.74855.

Doerfler, D., Deslippe, J., Williams, S., Oliker, L., Cook, B., Kurth, T., Lobet, M., Malas, T., Vay, J-L. and Vincenti, H. (2016) 'Applying the roofline performance model to the Intel Xeon Phi Knights landing processor', in *Proceedings of the ISC 2016 IXPUG Workshop*, June.

Duran, A., Terboven, C., Beard, J., de Supinski, B., Eachempati, D., Eichenberger, A., Karlin, I., Li, K., Olivier, S., Narayanaswamy, R., Pennycook, J., Rico, A., Sandoval, J., Scogland, T., Sewall, J. and Tian, X. (2017) *OpenMP Technical Report 5: Memory Management Support for OpenMP 5.0*, Technical Report, OpenMP Architecture Review Board (ARB), January [online] http://www.openmp.org/wpcontent/uploads/openmp-TR5-final.pdf (accessed July 2017).

Edwards, H.C., Trott, C.R. and Sunderland, D. (2014) 'Kokkos: enabling manycore performance portability through polymorphic memory access patterns', *Journal of Parallel and Distributed Computing*, Vol. 74, No. 12, pp.3202–3216.

Evans, J. (2006) 'A scalable concurrent malloc (3) implementation for FreeBSD', in *Proceedings of the BSDCan Conference*, Ottawa, Canada.

Friedrichs, M.S., Eastman, P., Vaidyanathan, V., Houston, M., Legrand, S., Beberg, A.L., Ensign, D.L., Bruns, C.M. and Pande, V.S. (2009) 'Accelerating molecular dynamic simulation on graphics processing units', *Journal of Computational Chemistry*, Vol. 30, No. 6, pp.864–872, ISSN: 1096-987X, DOI: 10.1002/jcc.21209.

Ghemawat, S. and Menage, P. (2009) *TCmalloc: Thread-caching malloc* [online] http://goog-perftools.sourceforge.net/doc/tcmalloc.html (accessed July 2017).

Gibson, D. and Litke, A. (2007) *libhugetlbfs* [online] http://libhugetlbfs.sourceforge.net (accessed July 2017).

Handy, J. (2014) *Where are DRAM Interfaces Headed?*, April [online] http://www.eetimes.com/author.asp?section_id=36&doc_id=1321783 (accessed 8 September 2016).

Harris, M. (2013) *Unified Memory in CUDA 6*, November [online] https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/ (accessed 8 September 2016).

Heinecke, A., Breuer, A., Bader, M. and Dubey, P. (2016) 'High order seismic simulations on the Intel Xeon Phi Processor (Knights Landing)', in Kunkel, M.J., Balaji, P. and Dongarra, J. (Eds.): *High Performance Computing: 31st International Conference, ISC High Performance 2016, Proceedings*, Springer International Publishing, Cham, Frankfurt, Germany, 19–23 June, pp.343–362, ISBN: 978-3-319-41321-1, DOI: 10.1007/978-3-319-41321-1_18.

Hennessy, J.L. and Patterson, D.A. (2011) *Computer Architecture: A Quantitative Approach*, Elsevier, Burlington, MA.

hugetlb (2016) *Linux Kernel HugeTLB Documentation*, Burlington, MA [online] http://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt (accessed 9 September 2016).

Jeffers, J. and Reinders, J. (2013) *Intel Xeon Phi Coprocessor High-Performance Programming*, Elsevier Science, ISBN: 9780124104945.

Kandemir, M., Ramanujam, J., Irwin, J., Vijaykrishnan, N., Kadayif, I. and Parikh, A. (2001) 'Dynamic management of scratch-pad memory space', in *Proceedings of the 38th annual Design Automation Conference*, ACM, pp.690–695.

Kim, J. and Kim, Y. (2014) 'HBM: memory solution for bandwidth-hungry processors', in *Hot Chips*, Vol. 26.

Kleen, A. (2005) *A NUMA API for Linux*, Novel Inc, Nuremberg, Germany.

Nickolls, J., Buck, I., Garland, M. and Skadron, K. (2008) 'Scalable parallel programming with CUDA', *Queue*, March, Vol. 6, No. 2, pp.40–53, ISSN: 1542-7730, DOI: 10.1145/1365490.1365500.

OpenACC Working Group et al. (2015) *The OpenACC Application Programming Interface*, October [online] http://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5.pdf (accessed July 2017).

OpenMP Architecture Review Board (2015) *OpenMP Application Program Interface Version 4.5*, November [online] http://www.openmp.org/mpdocuments/spec45.pdf (accessed July 2017).

Owens, M. and Allen, G. (2010) *SQLite*, Springer, Berlin, Germany.

Reinders, J. (2007) *Intel® Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*, O'Reilly Media, Sebastapol, CA.

Ruetsch, G. and Micikevicius, P. (2009) *Optimizing Matrix Transpose in CUDA*, Technical Report, NVIDIA.

Saltzer, J.H. and Schroeder, M.D. (1975) 'The protection of information in computer systems', *Proceedings of the IEEE*, Vol. 63, No. 9, pp.1278–1308.

Smith, R. (2015) *AMD HBM Deep Dive*, May [online] http://www.anandtech.com/show/9266/amd-hbm-deep-dive (accessed 8 September 2016).

Smith, R. (2016) *Hot Chips 2016: Memory Vendors Discuss Ideas for Future Memory Tech – DDR5, Cheap HBM, & More*, August [online] http://www.anandtech.com/show/10589/hot-chips-2016-memory-vendorsdiscuss-ideas-for-future-memory-tech-ddr5-cheap-hbm-more (accessed 8 September 2016).

Sodani, A., Gramunt, R., Corbal, J., Kim, H-S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R. and Liu, Y-C. (2016) 'Knights Landing: Second-Generation Intel® Xeon PhiTM Product', *IEEE Micro*, Vol. 36, No. 2, pp.34–46.

Steinke, S., Wehmeyer, L., Lee, B-S. and Marwedel, P. (2002) 'Assigning program and data objects to scratchpad for energy reduction', in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, IEEE, pp.409–415.

Stone, J.E., Gohara, D. and Shi, G. (2010) 'OpenCL: a parallel programming standard for heterogeneous computing systems', *IEEE Des. Test*, May, Vol. 12, No. 3, pp.66–73, ISSN: 0740-7475, DOI: 10.1109/MCSE.2010.69.

Udayakumaran, S. and Barua, R. (2003) 'Compiler-decided dynamic memory allocation for scratch-pad based embedded systems', in *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ACM, pp.276–286.

Unified EFI (2016) *Advanced Configuration and Power Interface Specification Version 6.1*, January [online] http://www.uefi.org/sites/default/files/resources/ACPI_6_1.pdf (accessed July 2017).

Valich, T. (2015) *NVIDIA Unveils Pascal GPU: 16GB of Memory, 1TB/s Bandwidth*, November [online] http://vrworld.com/2015/11/16/nvidia-unveilspascal-gpu-16gb-of-memory-1tbs-bandwidth/ (accessed 8 September 2016).