Utilizing Heterogeneous Memory Hierarchies in the PGAS Model

Roger Kowalewski, Tobias Fuchs, Karl Fürlinger
Institute of Informatics (MNM Team)
Ludwig-Maximilians-Universität München
Oettingenstr. 67, 80538 Munich, Germany
kowalewski@nm.ifi.lmu.de

***Abstract*—Emerging technologies such as non-volatile or 3D-stacked memory significantly impact the design of future high performance computing systems. To keep up with the increasing core count, relying only on DRAM is inefficient due to its static power consumption. Modern HPC architectures feature a heterogeneous memory hierarchy with different capacities and capabilities. This paper addresses two challenges. First, we need programming models to abstract the complexity of the underlying heterogeneous memory hierarchy, while still giving explicit control to domain experts. We propose the concept of memory spaces to model a heterogeneous memory hierarchy and integrate it into a PGAS-like programming model. Second, we need to understand the impact of different memory capabilities on the performance of scientific applications. An experimental evaluation with a series of benchmarks, conducted on a Intel KNL platform, reveals that proper data placement on specific types of memory achieves significant speedup.**

***Index Terms*—Memory, Heterogeneity, Locality, MPI, PGAS**

I. Introduction

Machine-level hardware design in High Performance Computing is progressing towards heterogeneous computation and memory resources in increasingly complex topologies. In effect, maintaining portable efficiency across today's largest HPC systems is an overwhelming challenge for application developers. While heterogeneity of compute devices is already common, we can currently observe significant changes in modern memory architectures. The increasing core count requires higher memory bandwidth which cannot be delivered only through conventional DRAM. DRAM suffers from both its low density and static power consumption which makes it inefficient for these workloads. Novel 3D stacked memory technologies provide better density and bandwidth [1] but come at disproportionate high cost considering the large capacities which are required to sustain data-intensive workloads.

Systems with heterogeneous memory technologies, working side-by-side of each other, will be the design approach in the near future. This trend is already observable by recently released architectures. As an example, consider the Intel Xeon Phi Knights Landing (KNL) platform which features multi-channel DRAM (MCDRAM) as on-chip high bandwidth memory (HBM) along with a substantially larger DRAM [2]. Another interesting design decision is the next generation HPC system at the Oak Ridge Leadership Computing Facility (Summit) where each node is equipped with 1.3 TBytes of heterogeneous memory, including DRAM, HBM and non-volatile memory (NVRAM) [3]. The question of how to effectively utilize these co-located memory technologies to improve data placement is subject of active research [4]. Approaches exist to utilize these additional memory as a *fast* last-level cache which is transparently managed by the hardware or operating system [5]–[7]. These designs do not require any modification in applications to take advantage of the additional memory. On the other hand, it increases hardware and operating system complexity to manage cache line or page replacement.

We believe that domain scientists or programmers need to have full control over the data distribution to explore algorithmic locality and prefer a *flat* memory space where available memory is exposed in its full potential. The operating system or third party vendor libraries have to provide a `malloc` like interface to explicitly allocate from heterogeneous memories. However, such a design is neither portable across different platforms nor flexible enough to support hierarchical and distributed data placements.

In this paper, we address these challenges and present a locality-aware and portable interface of heterogeneous memory spaces in the Partitioned Global Address Space (PGAS) model. We have established the following main goals:

1) Flexibility: The memory interface has to be flexible enough to support current and future memory technologies. The information about a specific memory space and its characteristics is encoded in compile-time type information.
2) Portability: Existing tools are either architecture-specific to a particular platform (e.g., libmemkind [8]) or not aware of unconventional memory technologies (e.g., `malloc`). The memory space abstraction has to support any memory type regardless of the underlying HPC platform.
3) Capability abstraction: Instead of programming against a particular memory API we need a capability model to express both qualitative and quantitative requirements which can then be specified as user-provided constraints to support proper data placement.

We gratefully acknowledge funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA).

The remainder of this paper is organized as follows. After presenting relevant background in Section II, Section III elaborates the approach to integrate memory spaces into DASH as a PGAS-like C++ template library. Section IV conducts a case study on an Intel KNL cluster, based on benchmarks in the Cowichan problems [9]. It shows that selecting between *fast* and *slow* memory is not simply a matter of choice but depends on application-specific characteristics which may alternate at runtime. Finally, Section V summarizes related work and Section VI concludes.

II. Background about PGAS and DASH

In the Partitioned Global Address Space (PGAS) model globally available memory is spanned over all processors by exploiting a one-sided *put/get* interface. It contrast to the well-known Message Passing Interface (MPI) it semantically decouples communication and synchronization [10], enabling a more data-centric programming model with flexible data placement strategies.

The work presented in this paper is performed in the context of DASH, a PGAS abstraction based on C++ templates which aims at providing distributed data structures and algorithms operating on them in parallel. The set of DASH data structures includes both static n-dimensional arrays and dynamic containers such as lists and maps. We apply the well-known concepts of C++11 to provide data manipulation and iterator primitives. Thus, DASH containers seamlessly work with the sequential C++ standard template library (STL) algorithms. In order to exploit the full performance potential the DASH container concept specifies an explicit model of locality, i.e., all accesses are either local or remote. Domain experts can specify flexible data placement strategies, like blocking or tiling, to match application-specific locality requirements [11]. Communication of shared data is transparently handled by the DASH Runtime (DART) which supports, amongst others, MPI-3 remote memory access (RMA) as the underlying communication substrate.

DASH follows the single program multiple data (SPMD) execution model. A processing entity is called a *unit* and a *team* groups an ordered set of units. Teams encapsulate computation and memory resources which are “owned” by the included units. A heterogeneous machine is represented as a set of hierarchically organized teams in a topological graph model. Primitives to query, traverse or filter this graph enables to obtain information about available memory and computation capacities. [12].

III. Heterogeneous Memory Hierarchies in PGAS

Figure 1 illustrates a high-level overview of a hierarchical memory space model. On the very top are DASH containers to organize the globally shared data. In the vertical direction are several layers of memory spaces, while the horizontal direction visualizes single nodes which contribute to the global memory. The following sections elaborate this design in more detail.

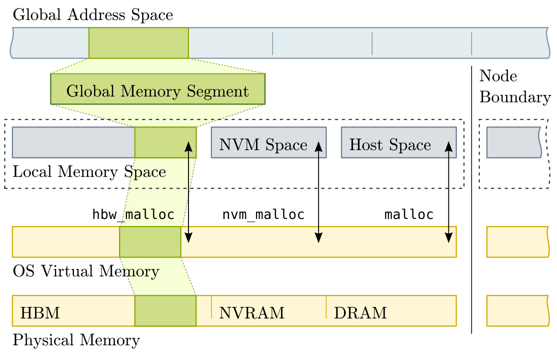


Fig. 1: Architectural overview of heterogeneous memory spaces.

A. Memory Space Abstraction

The purpose of a memory space is to abstract both qualitative and quantitative characteristics of the underlying memory resources, as well as mechanisms to allocate and deallocate chunks of memory. The interface for allocation and deallocation simply encapsulates the complexity in vendor-specific libraries such as Intel’s memkind [8] and PMEM libraries [13] or Nvidia’s CUDA Toolkit.

Qualitative characteristics include the memory type, such as DRAM, HBM or NVRAM. All of them have different performance characteristics. Another significant distinction refers to capabilities like persistency or volatility. Persistent memory spaces have stricter requirements regarding memory consistency, compared to volatile memory spaces [14]. Therefore, memory spaces which operate on NVRAM have to support operations to satisfy the underlying persistency model. We added an explicit flush operation to the memory space concept. For volatile memory spaces, this can be considered as a `NOP` which can be easily optimized away by compilers. Since this information is encoded in C++ templates, qualitative user-defined constraints can be formulated as predicates already at compile-time to guide proper data placement. As an example, domain scientists can leverage their application knowledge to allocate bandwidth-sensitive data on HBM or utilize NVRAM to enable checkpoint restart mechanisms in their applications. Figure 2 illustrates the programming model with two globally allocated DASH arrays, one in the default DRAM memory and other in HBM. Migration from one memory space to the other corresponds to a copy operation.

Another aspect is the scope which is inherent to the PGAS model. A memory space can be in the *global* domain which generally applies to all DASH containers. The global domain obviously requires a team (group of processes), defining the scope of the global memory. The remaining two domains are *local* to represent the system’s on-node memory or *device*, representing the memory space on accelerator devices. This makes it very flexible to integrate other domains. An example may be *NUMA*, enabling to reason about spatial locality in a more fine-grained manner.

```
// type alias for an array in default memory space
using array_t = dash::Array<int>;
// type alias for an array in HBW memory space
using array_hbw_t = dash::Array<int, dash::hbw_tag>;

int main(int argc , char * argv []) {
  dash::init(&argc, &argv)
  size_t  global_capacity = ...;
  array_t arr_dram(global_capacity);
  dash::fill(arr_dram.begin(), arr_dram.end(), ...);
  // Initial processing in DRAM...

  // Migrate data from DRAM to HBM
  array_hbw_t arr_hbw(array_dram.size());
  dash::copy(arr_dram.begin(), arr_dram.end(),
              arr_hbw.begin());
  // Further processing in HBM...
  dash::finalize();
  return 0;
}
```

Fig. 2: DASH example with heterogeneous memory spaces.

B. Memory Allocation Policies

In DASH, we separate the concepts of memory spaces and allocation policies. In order to follow application-specific requirements we provide a set of global and local allocators. Allocation strategies are chosen according to the DASH container semantics. As an example, memory allocation can be collective. Each unit allocates a local portion and collectively attaches it to the global memory domain. This is a good fit for static containers, such as arrays or matrices since capacity and size of the underlying memory are known prior to container construction. In contrast, non-collective means that units within a team can independently attach local to global memory. This approach is required for dynamic containers (lists, maps) as the capacity increases or decreases during the container's life time.

Global memory allocators rely on local allocators to acquire and release memory resources. Local allocators specify allocation strategies to match application-specific requirements. As an example, frequent allocation and deallocation may lead to memory fragmentation, both in the local and global memory space. This in turn may result in non-negligible overhead as the memory footprints increases. In order to mitigate these potential challenges we provide a set of strategies, such as the well-known arena or pool allocators to satisfy more demanding use cases in DASH lists or maps. Another purpose of local allocators is to compose various allocation strategies. Consider a situation where memory allocation is requested from a particular memory space, however, the requested capacity cannot be completely or at most partially serviced. Depending on the user requirements we can throw a runtime error or provide a fallback policy to allocate from another memory space.

IV. Experimental Evaluation

This evaluation studies two questions. First, we show the feasibility of our proposed memory space concept with current state of the art architectures. Second, we study the impact of various memory allocation strategies in scientific applications.

We conduct the experiments on the second phase NERSC Cori system which consists of 9 668 Xeon Phi KNL 7250 nodes, all configured in quadrant mode. Each node features 16GB MCDRAM and 96GB DRAM.

A. The Cowichan Problems

The Cowichan benchmarks represent common problems in scientific applications [9] to study the impact of various allocation strategies in heterogeneous memory systems. The following provides a brief overview of our selected kernels to conduct the experiments.

randmat: Generate a 2-dimensional array mat of random integers using a deterministic pseudo-random-number generator.

thresh: Given a percental threshold p, calculate a 2-dimensional bitmask $mask$ such that applying $mask$ to mat selects p percent of the highest values in mat.

winnow: Apply $mask$ to mat and store all masked values along with their 2D indices in (row, col, val)-tuples. Sort these tuples in ascending order by val and select n evenly strided elements, resulting in a vector of (row, col)-coordinates, called $points$.

outer: Given a matrix $omat$ with dimensions identical to mat. For all elements with coordinates (i, j) such that $i \neq j$, compute $omat[i][j]$ as the Euclidean distance of $points[i]$ and $points[j]$. .

product: Compute the inner matrix-vector product of $omat$ and vec.

We have implemented these benchmarks by using DASH containers and algorithms. The source code is publicly accessible[1]. *Winnow* and *outer* cause significantly more overhead compared to the other kernels due to higher memory footprint and algorithmic complexity. *Winnow* requires sorting a global array which is a non-trivial operation, particularly in distributed memory. *Outer* is memory-bound because we iterate with a unit-stride over the distributed array and compute element by element.

Optimizing memory placement requires considering the underlying memory access patterns. As an example, mat has a lifetime over 3 chained kernels (randmat, thresh, winnow) and is written to only in the first one. Moreover, its elements are accessed in a unit-stride manner both in *thresh* and *winnow*, resulting in good spatial and temporal locality. We implement our algorithm to allocate mat first in DRAM and subsequently migrate it to HBM to utilize the available bandwidth.

The same challenges apply to temporary non-DASH containers as well. Consider calculating an inner matrix-vector *product*. If data is globally distributed, one approach is to locally allocate temporary buffers and fetch remote data into it to accelerate local computation. Afterwards the copy is discarded. Proper reasoning about the life time and usage of temporal memory may result in a non-negligible impact on performance.

[1]https://github.com/dash-project/dash-apps

B. Measurement Methodology

All benchmarks have been executed in *flat mode* and *cache mode* which serves as the performance baseline. For *flat mode*, there exist two variants. In the first variant we allocate no data in MCDRAM, while the other relies on user-provided hints to schedule allocation either in DRAM or MCDRAM as specified in our memory space concept in Section III.

The codes were compiled using the provided Cray compiler wrapper and Intel ICC 18. The DASH runtime uses Cray MPI as the communication substrate. While we have 68 cores on a single KNL node we pin one unit[2] always on two cores, resulting in 34 units per node. We have found that scheduling more units on a single node causes high overhead and degrades performance in this setup. The Cowichan benchmarks have been run with different problem sizes differing in the number of elements per dimension. A problem size of $40k$ results in a $40k \times 40k$ matrix and a vector requires capacity for $40k$ elements. The problem sizes scale linearly starting from $20k$ to $80k$ while the number of nodes scale logarithmically from 1 up to 32 nodes. Due to lack of space, we report the median and standard error for one weak and one strong scaling study, each with 30 iterations. The memory requirements per node are plotted on the y-axis.

C. Discussion

Figures 3 and 4 show the results of our experiments. It is easy to see that *winnow* and *outer* dominate the execution time compared to the other kernels which is either reflected in the high memory requirements per node. If the memory requirements per node are relatively small and fits completely into the 16GB on-chip HBM our model of a flat memory space results in performance degradation of up to 7% compared to the *cache mode* version, even if we allocate all data on HBM. We attribute this to the additional library overhead since we have to track all memory allocations internally.

The results change as soon as we are close to the HBM capacity limit or slightly exceed it. The weak scaling study in Figure 3 shows that we can significantly outperform the *cache mode* if we strategically allocate data on both HBM and DRAM. In *winnow* we copy chunks of data back and forth between HBM and DRAM asynchronously to achieve good communication-computation overlap. Moreover, we explicitly allocate temporary STL containers to process local data either in DRAM or HBM. This is possible since our allocators satisfy the C++11 allocator concept. Read-only data is allocated in HBM while write-intensive data for sorting is allocated in DRAM. Both optimizations contribute to our speedup of 12%. Figure 3a confirms this strategy since the DRAM only variant is even faster than the cache mode variant.

Similar observations apply to the strong scaling study in Figure 4c. We start with a relatively high memory footprint per node and scale logarithmically the number of nodes. In the first experiment we can see the same effects as in the weak scaling case since the memory requirements exceed the available HBM capacity. Our hybrid HBM / DRAM variant achieves up to 16% compared to the cache mode variant. if we increase the number of nodes and reduce the memory footprint below the available HBM our performance advantage decreases, compared to cache mode.

[2]MPI process in this case

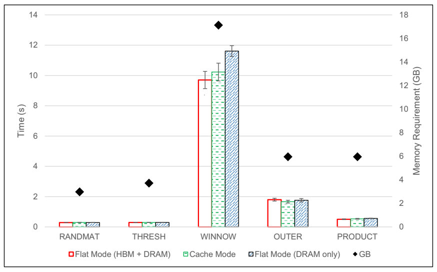
V. Related Work

Most scientific applications still use the traditional MPI programming model. This approach requires programmers to exploit the algorithmic locality and achieves high performance in many HPC scenarios. However, providing a more data-centric global view to work with hierarchically distributed data is challenging. Traditional PGAS approaches such as UPC [15] or OpenSHMEM [16] do not support the notion of hierarchical teams to reflect the locality of heterogeneous machines and provide only a 2-level (i.e., local or remote) distinction to reason about the access costs. More modern approaches such as Chapel [17] and X10 [18] provide the concept of *locales* or *places* to model heterogeneity and support memory spaces in recent releases. However, these approaches usually require to rewrite the whole application from scratch, which is difficult in most cases. Moreover, their integration with other numerical libraries is quite limited. Fortran and its included support for the Coarray data structure to support PGAS overcomes these limitations [19]. However, their model of heterogeneous memories is not as flexible as the presented work in this paper.

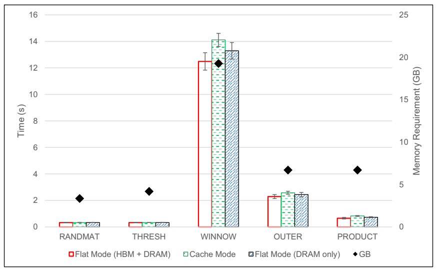
Kokkos [20] is conceptually close to this work. Like DASH, it is a C++ template library to provide n-dimensional arrays with a polymorphic compile-time data layout. However, it is limited to shared memory architectures and does not provide a global address space model. Phalanx [21] follows a similar approach and supports a distributed PGAS model. It does not expose an explicit memory space concept but provides built-in support for the OpenMP and CUDA backends. The work in this paper is more flexible in that we support any memory type, independent of the underlying platform.

VI. Conclusion and Future Work

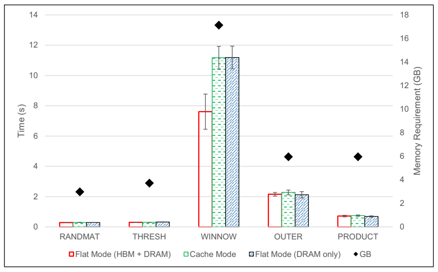
This paper presents a flexible design of heterogeneous memory spaces in a distributed PGAS model. The large number of related studies, addressing heterogeneous memory resources, confirms the significance of this topic. A prototypical implementation has been evaluated and demonstrates the feasibility of integrating heterogeneous memory spaces with current state of the art architectures. The proposed design enables to integrate emerging memory technologies without requiring any specific hardware support. The conducted case studies reveal that efficient utilization of heterogeneous memory resources requires application-specific knowledge of domain experts. In particular, identifying "hotspot" data structures with a high memory footprint and alternating memory access patterns are candidates for proper data placement strategies. In future work, we study memory allocation for temporary data structures in the DASH library to better utilize heterogeneous memory hierarchies. We further plan to abstract heterogeneous memory hierarchies based on a general capability model. The main



(a) $20k \times 20k$ matrix, 2 nodes

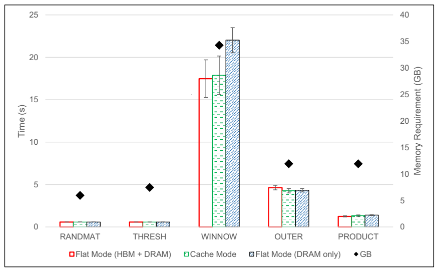


(b) $40k \times 40k$ matrix, 4 nodes

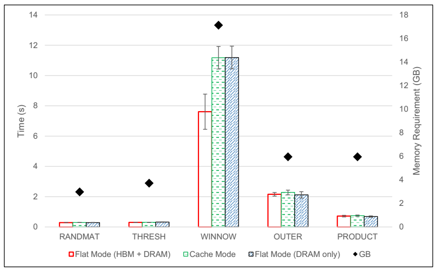


(c) $80k \times 80k$ matrix, 8 nodes

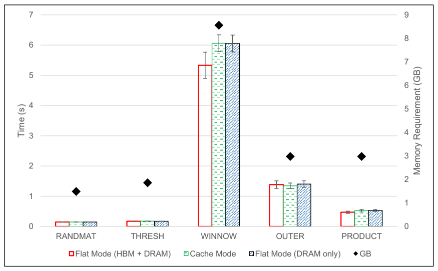
Fig. 3: Weak scaling, starting from 2 up to 8 nodes



(a) $80k \times 80k$ matrix, 4 nodes



(b) $80k \times 80k$ matrix, 8 nodes



(c) $80k \times 80k$ matrix, 16 nodes

Fig. 4: Strong scaling, starting from 4 up to 16 nodes

purpose is to provide a toolkit for user-specified hints which can be considered to deduce the concrete memory type.

References

[1] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance." IEEE, 2012.

[2] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Elsevier Science, 2016.

[3] Oak Ridge Leadership Computing Facility, "SUMMIT: Oak Ridge National Laboratory's next High Performance Supercomputer," `https://www.olcf.ornl.gov/summit` (Last Access: 09-2017).

[4] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories." IEEE, 2015, pp. 126–136.

[5] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," *ACM SIGARCH Computer Architecture News*, 2009.

[6] C. C. Chou, A. Jaleel, and M. K. Qureshi, "CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache." IEEE, 2014.

[7] C. Su, D. Roberts, E. A. León, K. W. Cameron, B. R. de Supinski, G. H. Loh, and D. S. Nikolopoulos, "HpMC: An Energy- Aware Management System for Multi-Level Memory Architectures," *Proceedings of the 2015 International Symposium on Memory Systems - MEMSYS '15*, 2015.

[8] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, and S. D. Hammond, "memkind: An Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies," Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2015.

[9] S. Nanz, S. West, K. S. Da Silveira, and B. Meyer, "Benchmarking usability and performance of multicore languages," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 2013, pp. 183–192.

[10] MPI Forum, "MPI: A Message-Passing Interface Standard. Version 3.1," June 4th 2015, available at: http://www.mpi-forum.org (Sep. 2017).

[11] K. Fuerlinger, T. Fuchs, and R. Kowalewski, "DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms." IEEE, 2016.

[12] T. Fuchs and K. Fürlinger, "Runtime Support for Distributed Dynamic Locality," in *Euro-Par 2017 International Workshops*. Springer, 2017, to appear.

[13] Intel Corporation, "Intel NVM Library," `http://pmem.io/nvml/` (Last Access: 09-2017).

[14] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, ser. FAST'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 5–5.

[15] UPC Consortium, "UPC language specifications, v1.2," Lawrence Berkeley National Lab, Tech Report LBNL-59208, 2005. [Online]. Available: http://www.gwu.edu/~upc/publications/LBNL-59208.pdf

[16] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010, p. 2.

[17] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," *International Journal of High Performance Computing Applications*, vol. 21, pp. 291–312, August 2007.

[18] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *ACM Sigplan Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[19] V. Cardellini, A. Fanfarillo, S. Filippone, and D. W. Rouson, "Hybrid coarrays: a pgas feature for many-core architectures." in *PARCO*, 2015, pp. 175–184.

[20] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[21] M. Garland, M. Kudlur, and Y. Zheng, "Designing a Unified Programming Model for Heterogeneous Machines," *High Performance Computing*, pp. 1–11, 2012.