[1] Hybrid Coarrays: a PGAS Feature for Many-Core Architectures

Valeria CARDELLINI [a] and Alessandro FANFARILLO [a] and Salvatore FILIPPONE [a] and Damian ROUSON [b]

[a] *Dipartimento di Ingegneria Civile e Ingegneria Informatica*
Università di Roma "Tor Vergata", Roma, Italy
[b] *Sourcery, Inc.*
Berkeley, California, USA

Abstract. Accelerators such as NVIDIA GPUs and Intel MICs are currently provided as co-processor devices, usable only through a CPU host. For Intel MICs it is planned that this constraint will be lifted in the near future: CPU and accelerator(s) will then form a single, many-core, processor capable of peak performance of several Teraflops with high energy efficiency. In order to exploit the available computational power, the user will be compelled to write a code more "hardware-aware", in contrast to the common philosophy of hiding hardware details as much as possible. The simple two-sided communication approach often used in message-passing applications introduces synchronization costs that may limit the performance on the next generation machines. PGAS languages, like coarray Fortran and UPC, propose a one-sided approach where a process accesses directly the remote memory of another process without interrupting its execution. In this paper, we propose a CUDA-aware coarray implementation, capable of merging the expressive syntax of coarrays with the computational power of GPUs. We propose a new keyword for the Fortran language, which allows the user to map with a high-level syntax some hardware features of the many-core machines. Our hybrid coarray implementation is based on OpenCoarrays, the coarray transport layer currently adopted by the GNU Fortran compiler.

Keywords. PGAS, Coarrays, CUDA, Fortran, Accelerators

Introduction

In order to reach challenging performance goals, computer architecture will change significantly in the near future. A large amount of research about exascale challenges has been published and the main limitations to performance growth have been identified as: 1) energy consumption; 2) degree of parallelism; 3) fault resilience; 4) memory size and speed [1].

The HPC community (software/hardware vendors, academia, and government agencies) is designing the architecture for the next generation machines. In [2] the authors use an abstract machine model in order to represent a possible exascale node architecture (Fig. 1, reprinted with permission from [2])

[1] Accepted for publication in *Proceedings of International Conference on Parallel Computing - ParCo2015*, Edinburgh, UK, September 2015.

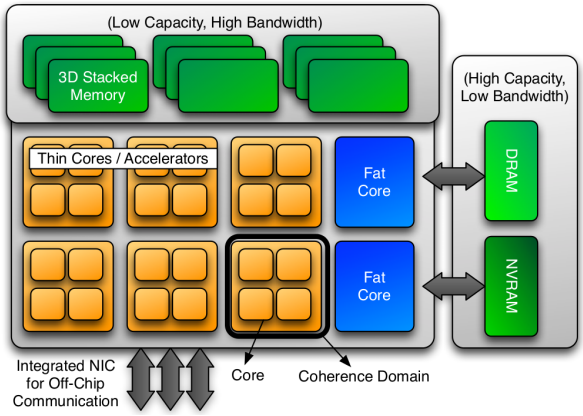

Figure 1. Model of exascale node architecture

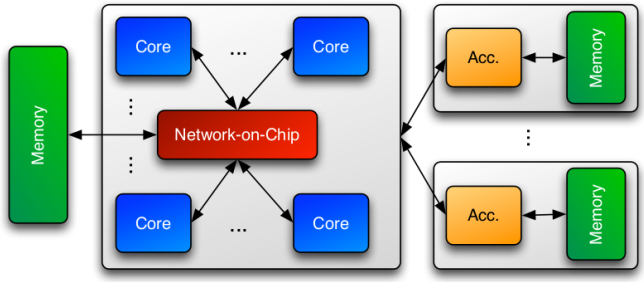

Figure 2. Current heterogeneous node with discrete accelerators

This representation depicts a heterogeneous node composed by two types of computational units: 1) *fat cores* (traditional CPU cores) suitable for algorithms with low level of parallelism and 2) *thin cores*, providing high aggregate performance and high energy efficiency. Another difference from today's architectures is the presence of a low capacity on-chip memory with high bandwidth. This new memory will likely be a multi channel DRAM (MCDRAM), a variant of the Hybrid Memory Cube (HMC) technology [3,4]. The idea is to create a DRAM stack and couple it with a logic process for buffering and routing tasks. This buffer layer makes the problem of data routing easier to solve.

The heterogeneous nature of the compute nodes (with fat and thin cores involved in the computation), the frequency scaling performed for energy reasons, and the possible fault of a compute node tend to break the assumption of homogeneous hardware considered by the bulk-synchronous model (BSP). In this model, each thread/process goes through a computational phase, and then waits until all reach a full barrier; this is a very regular pattern. The Partitioned Global Address Space (PGAS) parallel programming model, implemented for example by coarray Fortran and Unified Parallel C (UPC), is better suited for irregular and dynamic communication patterns and, in our opinion, is a valid alternative to the common MPI two-sided approach.

In this work, we combine the ease of programming provided by coarray Fortran with the power of NVIDIA GPUs; as far as we know, this is the first attempt to combine these two technologies. The most common heterogeneous architecture currently in use

is represented in Fig. 2, where a multi-core CPU is connected with discrete accelerator devices through a PCIe bus. In this paper we explore the use of hybrid coarrays on this reference architecture to demonstrate the suitability of the coarray programming model for the next generation of HPC platforms.

To account for the various memory layers present in the accelerators (in this case the GPU), we propose a new variable attribute called "accelerated". This attribute suggests the compiler to store/treat the variable in a "fast" way on the heterogeneous node, thereby delegating the memory management to the runtime environment. Work on heterogeneous (MCDRAM and external DRAM) memory management systems has already appeared in literature [5]; our proposal of an explicit keyword makes it possible to use such systems in a very convenient way.

1. Background on Accelerators

In the HPC world, many-core co-processors are referred to as "accelerators". Currently, the most common accelerator cards available on the market are GPUs (mainly provided by NVIDIA) and Intel Xeon Phi coprocessors, based on Intel Many Integrated Cores (MIC) architecture. These devices are provided as a separate card to be plugged on the PCI Express channel. Any such accelerator needs a CPU (called *host*), which actively interacts with the accelerator in order to send/receive data and/or invoke computational kernels. Accelerators are throughput-oriented, energy-efficient devices; since one of the main challenges for exascale computing is power consumption, accelerators currently represent the best option for breaking the "power wall".

In 2014, the National Energy Research Scientific Computing Center (NERSC) announced that its next supercomputer, named "Cori", will be a Cray system based on a next-generation Intel MIC architecture; this machine will be a self-hosted architecture, neither a co-processor nor an accelerator. In other words, the concept of accelerator as a separate co-processor will disappear in the foreseeable future; such a deep change in the processor architecture will require writing much more hardware-aware code in order to exploit all the available computational power.

1.1. Architectural Changes - Intel Xeon Phi

Table 1 provides a comparison between an Intel Ivy-Bridge processor and an Intel Xeon Phi Knights-Landing (KNL), thus illustrating the change between the current and future generations of HPC platforms[2].

There are two major changes when moving from a "classic" CPU to a KNL:

- More cores per node with longer vector registers;
- Two levels of memory with a small amount of very fast memory.

1.2. Architectural Changes - NVIDIA GPUs

The architectural changes proposed by NVIDIA for its next generation GPU (named Pascal) are similar to those proposed by the Intel Xeon Phi:

[2]Edison and Cori are the names of the systems installed or planned at NERSC.

Features	Edison (Ivy-Bridge)	Cori (Knights-Landing)
Num. physical cores	12 cores per CPU	60+ physical cores per CPU
Num. virtual cores	24 virtual cores per CPU	240+ virtual cores per CPU
Processor frequency	2.4-3.2 GHz	Much slower than 1 GHz
Num. OPs per cycle	4 double precision	8 double precision
Memory per core	2.5 GB	Less than 0.3 GB of fast memory per core and less than 2 GB of slow memory per core
Memory bandwidth	$\approx$ 100 GB/s	Fast memory has $\approx 5\times$ DDR4

Table 1. Architectural changes (source: NERSC)

- More (and slower) cores;
- Fast 3D (stacked) memory for high bandwidth and energy efficiency;
- A new high-speed CPU-GPU interconnect called NVLink from 5 to 12 times faster than the current PCIe.

With CUDA 6.0, NVIDIA accomplished one of the most important goals outlined in CUDA 2.0: to provide a Unified Memory, that is shared between the CPU and GPU, bridging the CPU-GPU divide.

Before CUDA 2.0, the only way to use memory on CUDA was to explicitly allocate a segment and manually copy the data from the CPU to the GPU using the `cudaMemcpy()` function. CUDA 2.0 introduced the zero-copy memory (also known as *mapped memory*); this feature allows to declare a portion of memory on the CPU to be directly accessible by the GPU. With this scheme the data movement is not directly coordinated by the user. CUDA 4.0 witnessed the introduction of the Unified Virtual Address (UVA) space: the CUDA runtime can identify where the data is stored based on the pointer value. UVA support makes it possible to directly access a portion of memory owned by a GPU from another GPU installed on the same node. In CUDA 6.0, NVIDIA introduced the concept of *managed memory*: data can be stored and migrated in a user-transparent way and the resulting code is thus much cleaner and less error-prone. At a first glance, zero-copy memory and managed memory look the same: both relieve the user from explicitly making copies from/to the GPU memory. The difference between the two is in *when* the memory access is done: for zero-copy, the transfer is started when the memory is accessed, whereas for managed memory the transfer is initiated immediately before the launch and after the kernel termination.

1.3. Clusters of GPUs

On hybrid clusters equipped with CPUs and GPUs, the most common way to exploit parallelism is through MPI for the inter-node communication, and CUDA for the GPU computation. This approach requires explicit data movement from/to GPU/CPU in order to send and receive data. In the latest evolution of both hardware and runtime libraries, this task has been either included in the MPI GPU-aware implementations [6,7] or performed with proprietary technologies, like GPUDirect. In [8] we compared the performance of various manual data exchange strategies with a CUDA-aware MPI implementation using PSBLAS [9]; we concluded that the MPI CUDA-aware implementation is largely sensitive to data imbalance.

2. Parallel Programming Models for Next Generation Architectures

The next computer architectures will expose hundreds of cores per single compute node; this will require adaptations of the commonly used programming models. Most importantly, to feed the cores with enough data, the memory hierarchy will have to expand, introducing additional layers between the cores and the main memory.

2.1. Hybrid MPI/OpenMP Approach

A common strategy to exploit the computational power provided by the many-core devices is to use a hybrid approach, combining MPI and OpenMP (or a similar directive-based language) for inter- and intra-node communication, respectively. This approach is common in GPU clusters, where the inter-node communication is performed with MPI and the actual computation is performed with CUDA or OpenCL on the local GPU(s) [8].

2.2. The PGAS Approach

An alternative to the MPI/OpenMP hybrid approach is to use a Partitioned Global Address Space (PGAS) model, as implemented for example by coarray Fortran (CAF) [10, 11] and Unified Parallel C (UPC). The PGAS parallel programming model attempts to combine the Single Program Multiple Data (SPMD) model commonly used in distributed memory systems with the semantics of shared memory systems. In the PGAS model, every process has its own memory address space but can expose a portion of its memory to other processes. At this time there are already publications [12,13] on the usage of PGAS languages on Intel Xeon Phi (KNC architecture); even though the evidence is not conclusive, it is our feeling that PGAS languages will play an important role for the next generation of architectures. This is especially because, as already mentioned, on an exascale machine equipped with billions of computing elements, the bulk-synchronous execution model adopted in many current scientific codes will be inadequate.

3. Introduction to Coarrays

Coarray Fortran (also known as CAF) began as a syntactic extension of Fortran 95 proposed in the early 1990s by Numrich and Reid [10]; it is now part of the Fortran 2008 standard (ISO/IEC 1539-1:2010) [11]. The main goal of coarrays is to allow language users to create parallel programs without the burden of explicitly invoking communication functions or directives such as with MPI and OpenMP.

A program that uses coarrays is treated as if it were replicated at the start of execution; each replication is called an *image*. Images execute asynchronously and explicit synchronization statements are used to maintain program correctness; a typical synchronization statement is `sync all`, acting as a barrier for all images. Each image has an integer image index varying between one and the number of images (inclusive); the run time environment provides the `this_image()` and `num_images()` functions to identify the executing image and the total number of them.

Variables can be declared as *coarrays*: they can be scalars or arrays, static or dynamic, and of intrinsic or derived type. All images can reference coarray variables located on other images, thereby providing data communications; the Fortran standard further provides other facilities such as locks, critical sections and atomic intrinsics.

3.1. GNU Fortran and LIBCAF

GNU Fortran (GFortran) is a free, efficient and widely used compiler; in 2012, GFortran started supporting the coarray syntax but only provided single-image execution, i.e., no actual communication. The main design point was to delegate the communication effort to an external library (LIBCAF) so that the compiler remains agnostic about the actual transport layer employed in the communication. Therefore, GFortran translates coarray operations into calls to an external library: OpenCoarrays. In [14] we presented two OpenCoarrays LIBCAF implementations, one based on MPI and the other based on GASNet [15]. Here we focus on LIBCAF_MPI, which assumes an underlying MPI implementation compliant with version 3.0. The following example shows how GFortran uses LIBCAF_MPI for a coarray allocation.

Coarray Fortran declaration of an array coarray with a dimension of 100 and an unspecified co dimension.

```
program alloc
implicit none
integer,dimension(100) :: x[*]
! More code here
```

Actual GNU Fortran call to OpenCoarrays function (C code)

```
x = (integer(kind=4)[100] * restrict) _gfortran_caf_register
(400, 0, (void * *) &caf_token.0, 0B, 0B, 0);
```

Actual memory and window allocation inside LIBCAF_MPI

```
MPI_Win_allocate(actual_size, 1, mpi_info_same_size,
                 CAF_COMM_WORLD, &mem, *token);
```

In the example, the total amount of memory requested is 400 bytes (100 elements of 4 bytes each). The local memory will be returned by the function and stored inside the *x* variable, whereas the variable used for remote memory access will be stored in the *caf_token.0* variable. In the case of LIBCAF_MPI, such token represents the MPI_Window used by the one-sided functions.

4. Hybrid Coarray Fortran

In this paper, we propose to merge the expressivity of coarray Fortran with the computational power of accelerators. As far as we know, this is the very first attempt to use coarray Fortran with accelerators. The idea is to exploit the Unified Memory provided by CUDA 6.0 to make a coarray variable accessible from either the CPU or the GPU in a completely transparent way. The only changes required in OpenCoarrays are: (1) to separate the MPI window allocation and creation, and (2) to synchronize the CUDA device before using the memory. In MPI-2, the only way to create a window is to locally allocate the memory (via `malloc` or `MPI_Alloc_mem`) and then use the `MPI_Win_create` for the actual window creation, whereas with MPI-3 there is the option of a single call to `MPI_Win_allocate`. Our approach is to allocate the local memory using the

`cudaMallocManaged` function in order to make that portion of memory "CUDA manageable", then call the `cudaSyncDevice` function, and finally create the window with `MPI_Win_create`. This approach is easy and general to implement, although it is not necessarily guaranteed to be the most efficient. A reasonable alternative would be either to delegate all communications to a CUDA-aware MPI implementation or to use a mapped memory approach, at the price of introducing a strong dependency on the quality of the MPI implementation. However, in our preliminary tests we found that the managed memory (Unified Memory) provided by CUDA 6.5 does not work too well with RDMA protocols (provided for example on Cray machines); we are fairly confident that such issue will be addressed in future CUDA implementations.

4.1. "ACCELERATED" Fortran Variables

With the approach introduced in the previous section, each coarray declared in the program requires interfacing with CUDA. What we suggest is a new variable attribute we call "accelerated". The meaning of this keyword is to mark a Fortran variable as "special", with faster access than a regular variable and suitable for accelerated computations.

In our current implementation an "accelerated" variable is CUDA-accessible; note that it is not necessarily also a coarray variable. The keyword is not meant to replace openACC statements for CUDA allocations, it just suggests the compiler to treat the variable differently than usual variables. We believe that such a keyword can play a significant role in the next generation architectures, where each processor will be an accelerator itself. As explained in Sec. 1.1, the Intel Knights-Landing will expose two types of memory: the first small and fast, the latter big and slow. Declaring a variable as "accelerated" would suggest the compiler that it could reside in the faster memory; in this case, the "accelerated" keyword assumes almost the same meaning as the "shared" keyword on CUDA. To test these ideas, we modified GFortran by adding this new keyword as an extension, currently affecting only allocatable variables. For coarray variables, we modify the `_gfortran_caf_register` by adding one more argument representing the accelerated attribute. For non-coarray variables, we force the allocation through `cudaMallocManaged` using a new function called `_gfortran_caf_register_nc` implemented in LIBCAF_MPI.

5. Experimental Results

To show the benefits of hybrid coarrays, we analyze in this section the performance of a matrix-matrix multiplication kernel based on the SUMMA algorithm [16]. We run the tests on Eurora, a heterogeneous cluster provided by CINECA, equipped with Tesla K20 and Intel Xeon Eight-Core E5-2658. We used the pre-release GCC-6.0, with OpenCoarrays 0.9.0 and IntelMPI-5. This unusual combination is because IntelMPI is the best MPI implementation provided on Eurora; however, OpenCoarrays can be linked with any MPI-3 compliant implementation.

5.1. MPI/CUDA vs. Hybrid CAF

On a cluster of GPUs, the most commonly used approach consists of employing MPI for the communication among GPUs, assuming that each process uses only one GPU, and

then calling the CUDA kernel on each process. This simple approach allows to use several GPUs on the cluster but it may suffer from the synchronization imposed by the two-sided functions (MPI_Send, MPI_Recv) provided by MPI. In order to invoke the CUDA kernels from Fortran using GNU Fortran, we make extensive use of the C-interoperability capabilities introduced in Fortran 2003. A typical example of C interoperability for the dot product $a \cdot b$ performed with CUDA is the following:

```
interface
 subroutine memory_mapping(a,b,a_d,b_d,n,img) &
                  &bind(C, name="memory_mapping")
   use iso_c_binding
   real(c_float) :: a(*),b(*)
   type(c_ptr) :: a_d, b_d
   integer(c_int),value :: n
   integer(c_int),value :: img
 end subroutine memory_mapping
 subroutine manual_mapped_cudaDot(a,b,partial_dot,n) &
                  & bind(C, name="manual_mapped_cudaDot")
   use iso_c_binding, only : c_float,c_int,c_ptr
   type(c_ptr),value :: a,b
   real(c_float) :: partial_dot
   integer(c_int),value :: n
 end subroutine
end interface
```

The two subroutines are interfaces for the C functions called `memory_mapping` and `manual_mapped_cudadot`. The first is used to map the memory previously allocated on the CPU for a and b onto the GPU; the function returns two C pointers called `a_d` and `b_d` which represent pointers usable on the GPU. The latter is the wrapper for the actual computational kernel. It takes as input arguments the GPU pointers returned by the `memory_mapping` function. NVIDIA claims that Unified Memory, besides reducing code complexity, could also improve the performance by transferring data on demand between CPU and GPU. There are already some studies [17] on Unified Memory performance, that shows the advantages to be strongly problem-dependent.

5.2. SUMMA Algorithm

SUMMA stands for Scalable Universal Matrix Multiplication Algorithm and is currently used in ScaLAPACK. The SUMMA algorithm is particularly suitable for PGAS languages because of the one-sided nature of the involved transfers.

Listing 1: Usual matrix product

```
do i=1,n1
  do j=1,n2
    do k=1,n3
      C(i,j) = C(i,j) &
               + A(i,k)*B(k,j)
    end do
  end do
end do
```

Listing 2: SUMMA approach

```
do k=1,n3
  do i=1,n1
    do j=1,n2
      C(i,j) = C(i,j) &
               + A(i,k)*B(k,j)
    end do
  end do
end do
```

Listings 1 and 2 allow to compare the pseudo-code for the usual matrix product to that of the SUMMA algorithm when we wan to multiply matrices A and B, resulting in matrix C. SUMMA performs n partial outer products (column vector by row vector). This formulation allows to parallelize the two innermost loops on i and j. Using MPI two-sided, each process has to post a send/receive in order to exchange the data needed for the computation; with coarrays, because of the one-sided semantics, each image can take the data without interfering with the remote image flow.

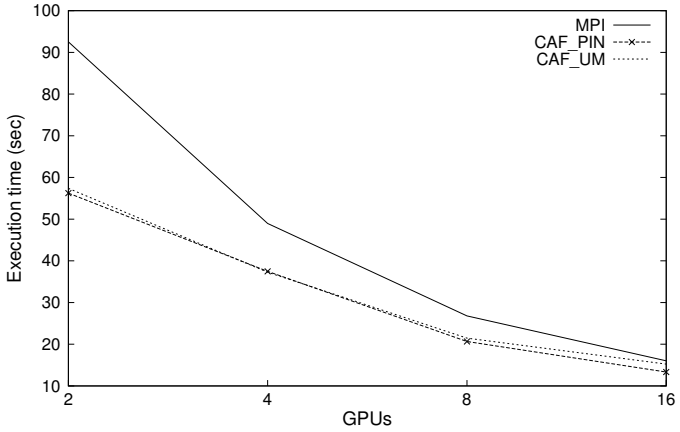

Figure 3. Performance of SUMMA: MPI-based vs. coarray Fortran implementations on Eurora cluster

Figure 3 compares the performance achieved by the coarray Fortran and MPI based implementations of the SUMMA algorithm. The chart shows the mean execution time on 10 runs using a matrix of size 4096x4096. We also report the performance of LIBCAF_MPI with the CUDA support based on CUDA mapped memory as well as on Unified Memory, labeled with CAF_PIN and CAF_UM respectively. We observe that the performance achieved with Unified Memory is equal or worse than that achieved with the usual pinned memory, as already noted in [17].

6. Conclusions

In this paper, we show how PGAS languages, and in particular coarray Fortran, can provide significant speedup in a hybrid CPU+Accelerator context. We show that using coarray Fortran, besides simplifying the code, improves the performance because of the one-sided semantic which characterizes PGAS languages. We also propose a new variable attribute called "accelerated" for the Fortran language. Such attribute instructs the compiler to treat the variable as suitable for acceleration. Based on what we currently know about future architectures, we think that such keyword can play a significant role in the post-petascale era, where heterogeneous code will be a must for exploiting all the computational power provided by complex and energy efficient architectures.

Acknowledgments

We gratefully acknowledge the support received from: CINECA for the access on Galileo/Eurora for the project OpenCAF under the ISCRA grant program for 2015; Na-

tional Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, for the access on Hopper/Edison under the OpenCoarrays grant.

References

[1] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *Proc. of 9th Int'l Conf. on High Performance Computing for Computational Science*, VECPAR '10, pages 1–25. Springer-Verlag, 2011.

[2] J. A. Ang et al. Abstract machine models and proxy architectures for exascale computing. In *Proc. of 1st Int'l Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '14, pages 25–32. IEEE, 2014.

[3] J. Jeddeloh and B. Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *Proc. of 2012 Symp. on VLSI Technology*, VLSIT, pages 87–88, June 2012.

[4] HMC Consortium. Hybrid memory cube, 2015. `http://www.hybridmemorycube.org/`.

[5] L.-N. Tran, F.J. Kurdahi, A.M. Eltawil, and H. Homayoun. Heterogeneous memory management for 3D-DRAM and external DRAM with QoS. In *Proc. of 18th Asia and South Pacific Design Automation Conf.*, ASP-DAC '13, pages 663–668, January 2013.

[6] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D.K. Panda. GPU-aware MPI on RDMA-enabled clusters: Design, implementation and evaluation. *IEEE Trans. Parallel Distrib. Syst.*, 25(10), 2014.

[7] A.M. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-C. Feng, K.R. Bisset, and R. Thakur. MPI-ACC: An integrated and extensible approach to data movement in accelerator-based systems. In *Proc. of IEEE 14th Int'l Conf. on High Performance Computing and Communication*, HPCC '12, 2012.

[8] V. Cardellini, A. Fanfarillo, and S. Filippone. Sparse matrix computations on clusters with GPGPUs. In *Proc. of 2014 Int'l Conf. on High Performance Computing Simulation*, HPCS '14, pages 23–30, 2014.

[9] S. Filippone and Buttari A. Object-oriented techniques for sparse matrix computations in Fortran 2003. *ACM Trans. Math. Softw.*, 38(4), 2012.

[10] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[11] R. W. Numrich and J. Reid. Co-arrays in the next Fortran standard. *SIGPLAN Fortran Forum*, 24(2):4–17, August 2005.

[12] M. Luo, M. Li, M. Venkatesh, X. Lu, and Panda D. UPC on MIC: Early experiences with native and symmetric modes. In *Proc. of Int'l Conf. on Partitioned Global Address Space Programming Models*, PGAS '13, October 2013.

[13] N. Namashivayam, S. Ghosh, D. Khaldi, D. Eachempati, and B. Chapman. Native mode-based optimizations of remote memory accesses in OpenSHMEM for Intel Xeon Phi. In *Proc. of 8th Int'l Conf. on Partitioned Global Address Space Programming Models*, PGAS '14, pages 12:1–12:11. ACM, 2014.

[14] A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle, and D. Rouson. OpenCoarrays: Open-source transport layers supporting coarray fortran compilers. In *Proc. of 8th Int'l Conf. on Partitioned Global Address Space Programming Models*, PGAS '14, pages 4:1–4:11. ACM, 2014.

[15] D. Bonachea. GASNet Specification, v1.1. Technical Report UCB/CSD-02-1207, Univ. of California Berkeley, 2002.

[16] R. A. van de Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurr. Comput.: Pract. Exper.*, 9:255–274, 1997.

[17] R. Landaverde, Z. Tiansheng, A.K. Coskun, and M. Herbordt. An investigation of unified memory access performance in CUDA. In *Proc. of IEEE High Performance Extreme Computing Conf.*, HPEC '14, pages 1–6, September 2014.