2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

Heterogeneous Active Messages for Offloading on the NEC SX-Aurora TSUBASA

Matthias Noack*, Erich Focht†, and Thomas Steinke*
*Zuse Institute Berlin, Germany
{noack, steinke}@zib.de
†NEC Deutschland GmbH
erich.focht@emea.nec.com

***Abstract*—The NEC SX-Aurora TSUBASA is a new generation of vector processing architectures that combines a standard Intel Xeon host with the newly developed NEC Vector Engine coprocessor cards. One way to use these coprocessors is offloading suitable parts of the program from the host to the Vector Engines. Currently, the only vendor-provided offloading solutions are the low-level Vector Engine Offloading (VEO) library, and a builtin reverse-offloading mechanism named VHcall.**

In this work, we extend the portable Heterogeneous Active Messages (HAM) based HAM-Offload framework with support for the NEC SX-Aurora TSUBASA. Therefore, we design, implement, and evaluate two messaging protocols aimed at minimising offloading cost. This sheds some light on how to achieve fast communication between host CPU and the Vector Engines of the NEC SX-Aurora TSUBASA. Compared with VEO, the DMA-based protocol reduces offloading overhead by a factor of 13×. The resulting framework enables users to write portable offload applications with low overhead, that do neither require a language extension like OpenMP, nor a special language like OpenCL. Existing HAM-Offload applications are now ready to run on the NEC SX-Aurora TSUBASA.

I. Introduction

The NEC SX-Aurora TSUBASA adds a new platform design to the diverse landscape of heterogeneous HPC architectures. Unlike it's predecessors from the SX product line, the vector processors now integrate 48 GiB of fast HBM2 memory, and reside on accelerator cards inside a standard Intel Xeon host system. Offloading parts of an application to these accelerators is one natural way of programming such systems.

Differerent accelerator architectures provide different levels of programming and execution models. While all of them provide some kind of offloading model, some architectures also allow native program execution with little or no involvement of the host CPU. GPUs reside on one end of this spectrum. They only provide an offload model like CUDA [1], OpenCL [2], or recently OpenMP [3], that typically comes as a proprietary implementation. The Intel Xeon Phi (Knights Corner) accelerator is an example of the opposite end of this spectrum. It basically is a complete compute node running its own Linux operating system, and being able to execute applications natively, i.e. independent of the host system. There are open APIs available that allow implementers to develop own middleware solutions and backends, such as HAM-Offload [4].

The NEC SX-Aurora TSUBASA lies somewhere in between. It can run an application natively but does not run its own operating system. Instead, every application requires an additional host process for executing system calls. The recommend usage model for the NEC SX-Aurora TSUBASA is native execution on the Vector Engines. This avoids the complexity and overhead of memory transfers between host and accelerator, but also requires the non-data-parallel code to run in a rather slow scalar execution mode. Whether native execution or offloading is the right match in practise depends on the application at hand, specifically the amount of scalar code, I/O, and the existing structure of the code. Many HPC applications can already use GPUs and are tuned for an offloading model, while tuning the scalar parts for the NEC SX-Aurora TSUBASA is an open task. With VEO (Vector Engine Offloading), NEC provides an open-source, low-level offloading API. However, none of the vendor-independent offloading models, like OpenCL [2], OpenACC [5], or OpenMP's target construct [3] are currently supported.

The main contributions of our work are:

- We extend the HAM-Offload [4] framework with support for the NEC SX-Aurora TSUBASA to provide a portable high-level offloading solution for that platform. HAM-Offload was originally developed as a flexible, low-overhead offloading framework that does not require additional compilers or language extensions. It uses Heterogeneous Active Messages (HAM) to delegate code execution between processes and comes as a pure C++ library.
- We use the NEC VEO API as a low-level layer to implement a HAM-Offload communication backend for the NEC Vector Engine.
- We present and evaluate two messaging protocols and investigate means to achieve low latency and high bandwidth communication through DMA.

A. HAM and HAM-Offload

HAM-Offload is an offloading framework built on top of HAM, which is short for Heterogeneous Active Messages [4]. Figure 1 illustrates the layered architecture with the HAM-Offload API on top. It combines the active message infrastructure of HAM with an abstract communication backend to transfer them between host and target processes.

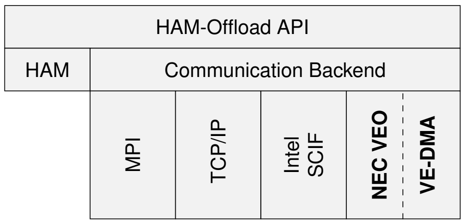

Fig. 1. The layer architecture of the HAM-Offload framework. The NEC communication backend component is highlighted.

In general, an active message refers to a concept where messages contain actions rather than just data. A trivial active message implementation on a homogeneous system could contain a function address that is executed by the receiver. HAM provides the basic means to create active messages that can be safely transferred and executed between heterogeneous binaries of the same program. It uses the C++ type system and template meta-programming to automatically generate handler functions for every message, as well as data structures for function address translation between the different binaries. This translation allows the runtime to use globally valid handler keys when communicating, while providing a fast translation in O(1) from and to locally valid function addresses that can be called.

Each offloading implementation needs to solve two problems: a) deploying the code on the offload target, and b) executing it. In HAM-Offload, the deployment is solved by simply building the exact same application for the host and the target. This reliefs the programmer of annotating, manually dividing, or pre-processing the code. The code execution is solved by using HAM and its efficient address translation of message handler functions to execute active messages on a (remote) target process. Function arguments and return values are transported inside the active message. A special type wrapper provides hooks to transparently do serialisation and de-serialisation of (complex) data types if necessary. Larger data buffers are managed and transferred explicitly, similar to frameworks like OpenCL. Table II provides an overview of the main API elements of HAM-Offload. Figure 2 exemplifies its use to offload the computation of an inner product.

The abstract communication backend is implemented for different networking interfaces. For most HPC systems, the MPI-backend is the adequate choice as MPI is widely available and enables remote offloading between different compute nodes and their accelerators in a supercomputer or cluster. The most generic backend uses TCP/IP, which focuses on interoperability rather than performance. It enables experiments like offloading over the internet, or between host and target combinations where MPI is not possible, e.g. from an x86 to an ARM system [6]. For the Intel Xeon Phi accelerator, a SCIF-backend is available. SCIF (Symmetric Communication Interface) is a low-level communication library from Intel, which provides means for one-sided communication over PCIe DMA transfers. It is limited to local offloads, but allows for low

```
/* HAM-Offload Example */
// inner product of vector a and b
double inner_prod(buffer_ptr<double> a,
                  buffer_ptr<double> b, size_t n) {
  double r = 0.0;
  for (size_t i = 0; i < n; ++i)
    r += a[i] * b[i];
  return r;
}

int main() {
  // host memory
  constexpr size_t n = 1024;
  std::array<double, n> a, b;

  // initialise host memory
  // ...

  // target memory
  node_t target = 1;
  auto a_target = offload::allocate<double>(target, n);
  auto b_target = offload::allocate<double>(target, n);

  // transfer memory
  offload::put(a.data(), a_target, n);
  offload::put(b.data(), b_target, n);

  // async offload, returns a future<double>
  auto result = offload::async(target,
    // function and arguments
    f2f(&inner_product, a_target, b_target, n)
  );

  // do something in parallel on the host
  // ...

  // sync on result future
  double c = result.get();

  return 0;
}
```

Fig. 2. Example program using the HAM-Offload API (highlighted) to compute the inner product of two vectors.

offloading overhead close to the latency of the PCIe bus. For the NEC SX-Aurora TSUBASA, an additional communication backend is required, as heterogeneous MPI jobs, spanning host and Vector Engine processes, are not yet available.

B. The NEC SX-Aurora TSUBASA

The NEC SX-Aurora TSUBASA is NEC's latest implementation of a vector processor architecture. Unlike the mainframe-sized computers of the previous NEC SX generations which were running the proprietary SuperUX variant of UNIX, the SX-Aurora is implemented as a PCIe Gen3 x16 card, plugged into Linux servers. This way, an Aurora node combines vector processing capabilities on one or multiple *Vector Engines (VE)* with scalar processing, I/O and communication capabilities on a default Intel Xeon or AMD based *Vector Host (VH)* which may be equipped with InfiniBand HCA cards.

The NEC SX-Aurora 1.0 Vector Engine is a vector processor with 8 cores, shared vector cache and 1.22 TB/s memory bandwidth to its 48 GiB high bandwidth memory built from six HBM2 3D memory stacks.

Its vector ISA supports long vectors of 256 64-bit words with vector length explicitly controlled by a register. Each

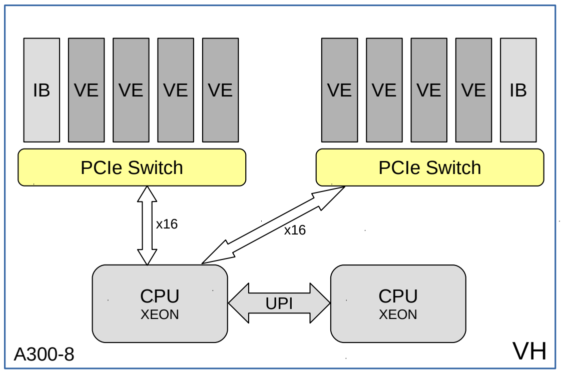

Fig. 3. Block diagram of the NEC SX-Aurora TSUBASA A300-8 system. The Vector Host (VH) is a two socket Intel Skylake system with eight Vector Engines (VE) connected through two PCIe switches. The InfiniBand (IB) cards are optional.

TABLE I
SPECIFICATIONS OF A SINGLE VECTOR HOST CPU AND VECTOR ENGINE (VE) OF THE USED NEC SX-AURORA TSUBASA A300-8 SYSTEM. THE WHOLE SYSTEM CONSISTS OF 2 CPUS AND 8 VES. SEE FIGURE 3 FOR A BLOCK DIAGRAM. THE UNIT GIB REPRESENTS 2^{30} BYTE, WHILE GB DENOTES 10^9 BYTE.

	Intel CPU Xeon Gold 6126	NEC VE Type 10B
Cores	12	8
Threads	24 (= 12×2)	8
Vector Width (double)	8	256
Clock Frequency	2.6 GHz	1.4 GHz
Peak Performance	998.4 GFLOPS	2150.4 GFLOPS
Max. Memory	384 GiB (DDR4)	48 GiB (HBM2)
Memory Bandwidth	128 GB/s	1228.8 GB/s
L3/LLC	19.25 MiB	16 MiB
TDP	125 W	300 W

core has 64 vector registers, three FMA (Fused Multiply-Add) vector units and two fixed point arithmetic and logical vector units. The processing of full length vector registers is done in a 32-fold SIMD and 8 steps deep pipelined fashion, which, when used properly, can run with outstanding power efficiency.

The key properties of the NEC SX-Aurora TSUBASA A300-8 used in this paper are summarised in Table I. An in-depth introduction and evaluation of the NEC SX-Aurora TSUBASA architecture is given by Komatsu et al. [7].

The VEs run no operating system and no kernel. The OS functionality of VEOS (Vector Engine Operating System) is offloaded to the host system and consists of three components:

- The *veos* daemon running in user space with root privileges takes care of memory and process management, scheduling, and DMA. Each VE has its own instance of VEOS.
- The Linux kernel modules *ve_drv* and *vp*.
- A user process corresponding to each VE process that handles the loading of the VE program into the engine and monitors and handles interruptions and exceptions from the VE program. This process is executing the VE syscalls in the user's context and under Linux.

The offloading of VEOS to the Linux host allows the VE programs to use almost any Linux system call, thus bringing a Linux look-and-feel and compatibility to the VE.

While OS offloading provides Linux compatibility for native VE programs built for example with the auto-vectorising C++, C or Fortran compilers, the NEC SX-Aurora TSUBASA platform also offers the opportunity for hybrid programming, using either the VE for well vectorised code or the VH CPUs for scalar code. The mechanisms provided by NEC are:

- VHcall: VE code can call code on the VH in a synchronous fashion, with syscall semantics. Intended for native VE programs.
- VEO (Vector Engine Offloading): VH code can call VE functions asynchronously, like usual accelerator codes. VEO semantics are close to OpenCL and aimed at native VH code that executes VE kernels.
- Hybrid MPI: MPI processes spread over both VEs and VHs and are able to communicate with each other, even over VH node boundaries. (Not yet available.)

VEO and VHcall have explicit memory transfer mechanisms between VH and VE. They are initiated from the VH side and executed in the context of VEOS. They use the system or privileged DMA engine which is shared by all cores of one VE and controlled in the DMA manager component of VEOS. Transfers using the privileged DMA need to translate the addresses into absolute or physical memory addresses. This is done on the fly in the DMA manager inside VEOS.

In addition, each VE core has a user DMA engine that can be used for data transfers initiated by VE code. Since the VE has no IOMMU (Input/Output Memory Management Unit), memory buffers from the VH or from other VEs need to be registered in the DMAATB (DMA Address Translation Buffers) after which they are accessible byte and word-wise through the *LHM* and *SHM* (Load/Store Host Memory) instructions or by explicit DMA requests.

The implementation of HAM-Offload discussed in this paper uses VEO as well as privileged DMA and user DMA of the VEs to implement the two communication protocols presented.

II. Related Work

Offloading is a common programming model to effectively utilise host-attached accelerators. In the following, we first discuss work related to offloading on the NEC SX-Aurora TSUBASA platform or utilising HAM-Offload.

Cramer et al. [8], [9] implemented a source-to-source transformation tool to support the OpenMP Offload programming model on the NEC SX-Aurora TSUBASA. The *clang*/LLVM-based tool uses the *libomptarget* library for the OpenMP outlining process. Similar to one of our approaches, VEO is used for the low-level communication between VH and VE.

TABLE II
OVERVIEW OF THE MAIN COMPONENTS OF THE HAM-OFFLOAD API. FOR MOST FUNCTIONS, SYNCHRONOUS AND ASYNCHRONOUS VERSIONS ARE AVAILABLE. EACH HOST OR TARGET RUNS A PROCESS PERFORMING EITHER THE HOST OR OFFLOAD TARGET ROLE.

`node_t`	– Address type of a process, i.e. an offload host or target.
`node_descriptor`	– Contains information on a node (e.g. name or device-type).
`buffer_ptr<T>`	– Pointer to a target memory address of type `T`. The node address is included.
`future<T>`	– Lazy synchronization to an asynchronous offload operation with result type `T`. Provides non-blocking `test()` and blocking `get()` accessors.
`Functor f2f(Function, Arguments...)`	– Function to functor conversion: binds arguments to a function and returns an offloadable functor object.
`Functor::result_type sync(node_t n, Functor f)`	– Performs a synchronous offload of `f` to node `n`.
`future<Functor::result_type> async(node_t n, Functor f)`	– Performs an asynchronous offload of `f` to node `n`.
`buffer_ptr<T> allocate<T>(node_t n, size_t s)`	– Allocates memory on an offload target `n`.
`void free(buffer_ptr<T> p)`	– Frees memory at `p`.
`future<void> put(buffer_ptr<T> src, T* dst, size_t s)`	– Writes data from host memory at `src` into target memory at `dst`.
`future<void> get(buffer_ptr<T> src, T* dst, size_t s)`	– Reads data from target memory at `src` into host memory at `dst`.
`future<void> copy(buffer_ptr<T> src, buffer_ptr<T> dst, size_t s)`	– Performs a direct copy between memory on two offload targets. The operation is orchestrated by the host.
`size_t num_nodes()`	– Returns the number of processes of the running application.
`node_t this_node()`	– Returns the address of the current process.
`node_descriptor get_node_descriptor(node_t n)`	– Returns the descriptor of node `n`.

As the work is still in progress, no performance numbers have been presented yet.

In [10], Malý et al. use HAM-Offload for the Intel Xeon Phi coprocessors implementation of a domain decomposition solver based on local Schur complements with respect to the skeleton of the computational domain. In the FETI step, a large number of dense matrices kernel operations are applied which can be offloaded. With HAM-Offload, the authors are able to implement a simple load-balancing strategy to efficiently utilise both the host CPU and the available coprocessors during the individual iterations.

In [11], HAM-Offload is compared with other offloading models for the Intel Xeon Phi coprocessor, namely OpenMP, Intel LEO [12], and hStreams [13]. The authors find HAM-Offload to be the best-performing solution. Chen et al. [14] review and relate different programming methods for the Intel MIC coprocessor platform, including HAM-Offload, with one another. The authors classified the level of abstraction of the methods. Besides HAM-Offload, other methods providing a high-level of abstraction are hStreams, LEO/OpenMP 4, MYO, and an extension of LEO by Ravi et al. [15].

HAM-Offload has been identified as a potential model for enabling offload over fabric for the PyMIC [16] Python Module.

The following, selected works cover interesting research results on generic high-level offload methods. Hahnfeld et al. [17] implemented a CG solver on multi-device accelerator nodes with GPUs and Xeon Phi coprocessors using the programming models CUDA, OpenCL, OpenACC, and OpenMP, respectively. In particular, the performance of directive based models with a higher abstraction level are compared with lower level paradigms.

Capotondi et al. [18] developed a runtime system for a cluster-based manycore accelerator, optimised for the concurrent execution of offloaded computation kernels from different programming models on high-end embedded systems low-power microservers. The runtime supports spatial partitioning and deploys generic offload requests into the target programming model semantics.

Grasso et al. [19] presented *libWater*, a library for simplifying the programming of heterogeneous distributed systems by providing a high-level interface to OpenCL. The library includes device query and management of a large number of OpenCL devices in a distributed environment.

III. USING VEO FOR HAM-OFFLOAD

A. Existing HAM-Offload Backends

Currently, none of the existing HAM-Offload communication backends matches with the available APIs on the NEC SX-Aurora TSUBASA. While heterogeneous MPI is not yet available, the TCP/IP backend could be used in principle, but there is no operating system running natively on the Vector Engine. That means, both VH and VE would use the same TCP/IP stack of the host kernel, and every communication operation on the VE would reverse offload a system call to the host. Also TCP/IP in general is more designed for reliability than performance, i.e. comes with avoidable protocol overhead on top of the reverse offload. Hence, a new backend is needed to support the NEC SX-Aurora TSUBASA.

B. HAM-Offload Requirements

In order to implement a suitable HAM-Offload communication backend for the NEC SX-Aurora TSUBASA, and to run HAM-Offload applications, certain requirements must be met:

- Compilation of the whole application into two binaries, one for the VH and one for the VE.
- Running those binaries on VH and VE.
- Reliable communication between the VH and VE process.

From the available APIs, the VEO library is the best match for HAM-Offload. It provides means for data exchange, and to run code on the VE. Together with the NEC compiler for creating the VE binary, all the requirements above are met.

C. Code Deployment and Execution

Implementing a communication backend using VEO means mapping our requirements onto the VEO API and its code deployment and runtime mechanisms. Figure 4 illustrates the result.

In order to offload a kernel, its code must be available on the target. Most offload frameworks require separating host and target code, either manually by putting it into separate files, or by annotating it such that a tool can separate the code during the compilation process. In HAM-Offload, we simply build the whole application for both, the host and the offload target architecture. This relieves the application programmer from caring about separation aspects, while it also makes sure, that the code generated by the HAM template library matches on both sides.

For every function that is offloaded inside code running on the host, the `f2f()` construct (see Figure 2) causes a chain of C++ template instantiations, that generate a corresponding active message type and handler, as depicted in Figure 6. By implicitly including the host source into the compilation process for the target binary, we guarantee that this code generation needed for offloading happens in the same way for both binaries. The host side can be compiled with any standard C++ compiler e.g. GCC, Clang/LLVM, or the Intel C++ compiler. For the target, the NEC NCC is currently the only compiler capable of generating code for the VE. An LLVM backend is in development [20].

All prior targets supported by HAM-Offload require building two executable binaries which are then started on the host and the target processor to compose a HAM-Offload application. In a VEO application, things are different, as illustrated in Figure 4. The VEO API provides all the primitives needed to start a VE process, to load a pre-compiled library into that process, to call C-functions with basic argument and return types from that library, and to allocate and transfer memory between VH and VE process. Unlike with the MPI backend, where target and host executable are used to start heterogeneous processes inside a single MPI job, we have to use VEO to create the target process on the VE from inside the host process.

In a typical VEO program, we load a shared library with the offloaded code, and can then query the functions by their symbol name and execute them as needed inside the VE process. To bring this model together with HAM-Offload's approach of having the whole application on both sides, we compile the application into a library instead of an executable for the VE, and transparently rename the `main()` function in the process to avoid naming conflicts. At runtime, we then load this application library via VEO, and perform a VEO offload that asynchronously starts the `ham_main()` function on the VE side. Inside that `ham_main()` function,

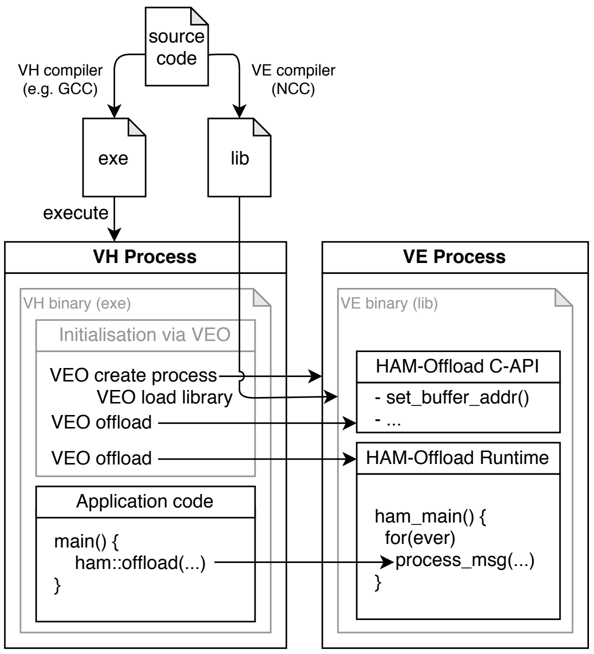

Fig. 4. The compile-time and runtime-setup of a HAM-Offload application using VEO on the NEC SX-Aurora TSUBASA. Two binaries, an executable for the VH (Vector Host), and a library for the VE (Vector Engine) are created from the same source. The VH executable sets up the VE process and initialises the HAM-Offload runtime before handing control over to the program's `main()` function.

the HAM-Offload Runtime takes over and starts processing active messages. This is the same result as if two executables were started on host and target.

D. A VEO-based Communication Protocol

After the initialisation phase, HAM-Offload only requires the data-transfer primitives of VEO to implement its message passing. VEO's read and write operations are sufficient to create a conceptually one-sided communication protocol, where the VH manages and accesses the communication data structures on the VE. Figure 5 shows the resulting protocol for offloading a function call via HAM-Offload through VEO. The VE memory holds two sets of communication buffers, each consisting of a set of message buffers and corresponding notification flags. The VH writes messages into free receive buffers on the VE. It then signals completion by setting the corresponding flags from an invalid value to an index. Every time the HAM-Offload runtime on the VE runs idle from executing an active messages, it polls the notification flag of the next receive buffer to check whether the host has written a new message into its memory. If the flag is set, the corresponding message is executed through HAM by looking up and calling the right message handler.

To communicate back the return value of the offload, the VE side generates a result message, that is written into one of the send buffers of the VE, followed by setting the corresponding flag. The synchronisation on that value happens through a

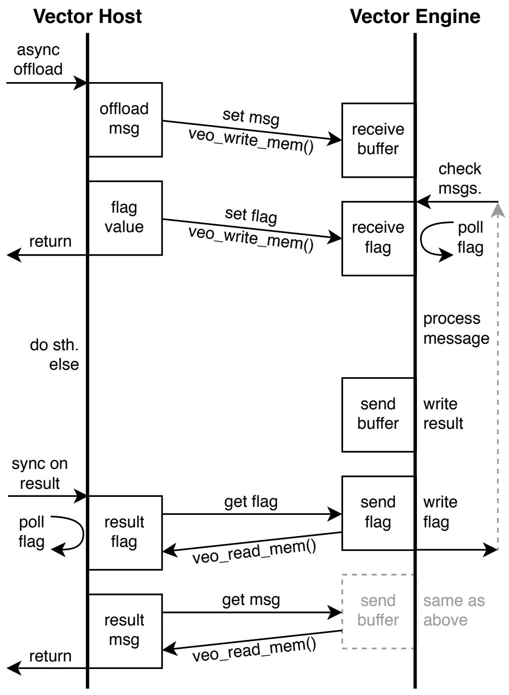

Fig. 5. The message transfer protocol implemented by the VEO communication backend. Message buffers for receiving offload messages and sending result messages reside on the VE and are accessed through VEO calls. Notification is implemented through flags, set after the corresponding message buffers have been written.

future object on the VH, which will check the flag of the send buffer on the VE and either return the result, or wait until it is available. All buffers are managed by the host and the information which buffer to receive from next, and where to send the result is piggybacked through the flags and offload messages. This one-sided communication approach makes sure that the VH can write messages via PCIe into the VE memory while the VE is executing a previously received active messages in parallel—thus enabling overlap of communication and computation.

To set up this communication protocol, the VH uses VEO's offload and memory allocation features. Prior to executing the HAM-Offload runtime on the VE, by calling the `main()` through VEO, the VH sets up all communication buffers and flags through VEO. The VH does the book keeping and management of these buffers. The VE-side of the communication backend exposes a set of C-function (HAM-Offload C-API in Figure 4) that are compatible with VEO's offload mechanism. They are called by the VH through VEO to communicate the memory addresses of the communication data structures from the host to the VE, and to transfer some additional HAM-specific parameters to the VE.

Since all the data structures are created and managed by the host, no synchronisation is needed during initialisation. The host program might already start offloading, by writing into the VE memory, before the communication backend on

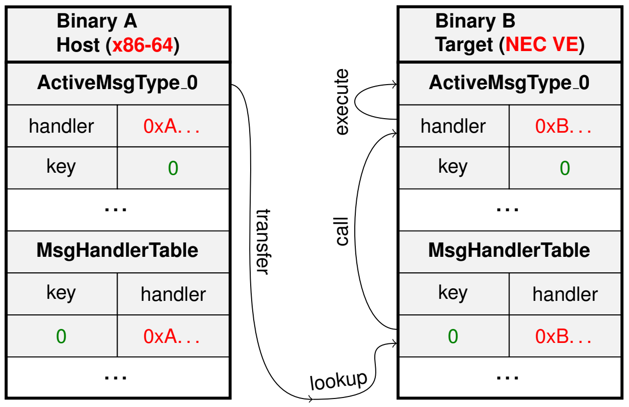

Fig. 6. This schematic shows HAM's active messaging data structures and the address translation process between the heterogeneous binaries on the host and target architecture. The addresses (red) are only valid within a single binary, while the handler keys (green) are valid across binaries. The transfer step is performed by the communication backend.

the target is even initialised. As soon as the HAM-Offload runtime on the VE begins its message processing loop, it will start polling on the first flag and may or may not find it's first message to be already there.

E. *Offloading Code via HAM*

Figure 6 illustrated the HAM perspective of an offload, where the address translation and message execution between the heterogeneous binaries on the VH and VE happen. The transfer step is performed by the communication backend, while HAM makes sure that active messages can be executed regardless of differing code addresses in the involved binaries. Therefore, each active message type is associated with the globally valid handler key, which is transferred alongside the message. On the receiving target, a generic handler is used to look up the address in the message handler table to call the local handler code for the specific message type received. The message handler performs a typecast back into the actual data type of the active message, which can then be executed as a functor. This process paves the way for the typeless bytes of the receive buffer back into the typesafe world of C++.

Setting up the translation tables relies on the C++ *typeid* feature. All processes collect the type names and local addresses of their message handlers in a table during program initialisation. While the type names are typically the same on both sides, the addresses are not. Sorting the table in a lexicographic order of the type names results in the same order on both sides without further communication. The index inside the sorted table can then be used as a globally valid reference that can be translated into the local handler address in O(1). Using *typeid* requires the used C++ compilers to have a compatible ABI. Compatible means that the type names, which are typically the wrangled symbol names, are either the same on both sides, or at least guaranteed to have the same lexicographical order. Most C++ compilers (GCC, Clang/LLVM, Intel, ...) reference the Intel Itanium ABI [21] for the name-wrangling convention [22]. The NEC ABI [23] remains unspecific in its current version, but aims for GCC compatibility which practically results in the same name-wrangling scheme as employed by the used host compiler.

The communication protocol presented in this section enables HAM-Offload applications to run on the NEC SX-Aurora TSUBASA. However, the performance of the VEO's read/write operations for small data transfer is sub-optimal, as their current implementation bears a lot of overhead.

As mentioned in Section I-B the VH-VE memory transfer operations in VEO use the privileged DMA descriptors which are controlled by the VEOS. These descriptors require absolute (physical) addresses and the virtual to physical translation is done on the fly inside the DMA manager component of the VEOS. The latency of these transfers is furthermore increased because setting up the DMA involves three components which have to communicate with each other: the pseudo process of the VE-process running at user privileges, VEOS running as root and the kernel modules *ve_drv* and *vp*. For larger buffers of a few MiB and more, the bandwidth achieved by using this mechanism reaches and exceeds 11 GB/s with the improved DMA manager from VEOS 1.3.2-4dma [24] when huge pages are employed on the VH side. The improved DMA manager uses bulk virtual to physical translations overlapping descriptor generation and DMA transfers. It was accepted for inclusion into the official VEOS.

In the next section, we look at means to further reduce the cost of transferring the small active messages used for offloading.

IV. Reducing Offload Cost through DMA

A. VE-Initiated Communication

In addition to VEO, which is used in the VH program, the VE-side provides some means of communication which are closer to the metal: The user DMA engine that can be programmed via an API, as well as special LHM and SHM instructions (Lost/Store Host Memory) from the VE ISA that can be used via inline assembly code to access single words of the host memory. Both require the accessed memory to be registered at the DMA Address Translation Buffer (DMAATB) and mapped into the VE process address space as VE Host Virtual Address (VEHVA). Unlike the privileged DMA used in VEO, the access to VEHVA with the above mentioned mechanisms requires no virtual to physical translation and no interaction with other OS components.

In order to use those features, a rather complex setup process is required. The resulting memory layout of both processes is illustrated in Figure 7. The VH needs to set up a SystemV Shared Memory Segment, that is mapped into the virtual address space of the VH process. A key to that memory segment can then be used on the VE-side to get a handle of the segment and to register it in the DMAATB. For the VE-side of the DMA, a corresponding segment of the local HBM2 memory needs to be registered as well. This segment is mapped to the virtual address space of the VE process. Both VH and VE process can access their local memory as usual. In addition, the VE can issue DMAs and load/store instructions between the shared memory segment of the host and the registered local memory.

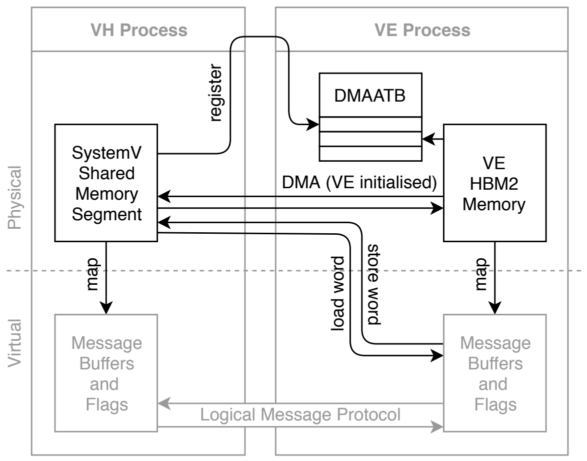

Fig. 7. The memory setup between VH and VE processes to enable fast data transfers through DMA and direct load/store (through LHM and SHM instructions). The VH memory is shared with the VE process using the SystemV shared memory interface. Both memory regions need to be registered for DMA in the DMAATB on the VE side. All communication operations are initiated by the VE.

B. DMA-based Communication Protocol

The protocol described in the prior section used one-sided communication coordinated by the VH. In order to use the aforementioned features, the communication model needs to be adapted to one-sided communication issued from the VE. Figure 8 shows the resulting protocol. All the communication memory does now reside on VH, thus rendering all the operations on the host side local memory accesses. The VE-side of the communication backend uses the SHM and LHM instruction (load/store word) to access the flags and transfers the messages via DMA. The VE now needs to actively fetch its messages, after the flag was set by the VH, which costs some time before the message can be executed. For the result message on the other hand, the VH is now the passive receiver who finds its message already in its local memory as soon as the flag is set by the VE. Starting the application, initialisation and data exchange are still performed through the VEO API.

Conceptually, both protocols are one-sided with the active side being either the VH or the VE. However, as discussed at the end of the prior section, the VEO data transfers come with much larger latency due to the need for virtual to physical translation and the involvement of several software components on the VH, communicating with each other. Using user DMA and special instructions on the VE explicitly makes sure the hardware capabilities are fully exploited. The next section evaluates the measured performance for both approaches.

V. Evaluation

In order to evaluate the performance of the HAM-Offload framework, we use microbenchmarks to measure the cost

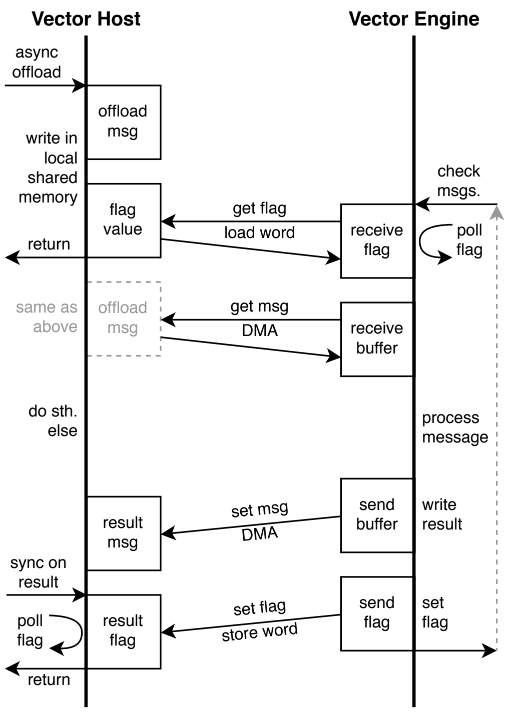

Fig. 8. The one-sided DMA-enabled protocol initiates all data transfers from the VE. There currently is no API for initiating DMA from the VH.

TABLE III
CONFIGURATION OF THE BENCHMARK SYSTEM.

System	NEC SX-Aurora TSUBASA A300-8
VH CPUs	2× Intel Xeon Gold 6126
VH Memory	192 GiB DDR4
VE Cards	8× NEC VE Type 10B, 48 GiB HBM2
PCIe Config.	see Fig. 3
VH OS	CentOS Linux release 7.6.1810, kernel 3.10.0-693
VH compiler	GCC 4.8.5
VEOS	1.3.2-4dma
VEO	1.3.2a
VE compiler	NEC NCC 1.6.0

for offloading a kernel, and to compare the bandwidth for copying data between VH and VE using the different methods described. We also benchmarked the VEO API, and use the results as a reference. The theoretical peak bandwidth without any overhead of a PCIe Gen3 x16 card is 14.7 GiB/s. Given the maximum payload size of 256 byte for the NEC Vector Engine and the PCIe protocol overhead [25], at most 91 %, i.e. 13.4 GiB/s, are achievable. Table III describes the system and software versions used for the benchmarks, processor specifications are summarised in Table I. The VH process runs on the first CPU socket, and offloads to the first VE (see Figure 3).

Measurements for offloading kernels were repeated 10^6 times, data transfers 10^3 times for every data size. Timings were preceded by 10 warm-up iterations to avoid distortion from effects like cold caches. The host CPU's frequency was set to its maximum frequency by selecting the *performance* scaling governor to eliminate noise from dynamic frequency scaling. All shown numbers are averages over all runs.

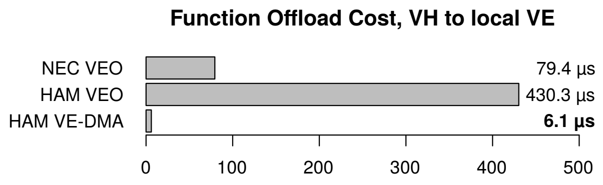

Fig. 9. Comparison of the offloading overhead as the time for offloading an empty kernel, i.e. the minimal cost that occurs with every offload.

A. Offloading Cost

Figure 9 compares the cost for offloading an empty function call. We use this value as a measure for the offload overhead that applies to every offloaded function. The amount of data transferred here typically is in the order of tens or hundreds of bytes, depending on the number of function arguments. Thus, the offload cost is dominated by communication latency and the framework overhead, i.e. the facilitating code executed prior and posterior to the offloaded function.

HAM-Offload with the VEO backend is $5.4\times$ slower than using VEO directly. The reason for this is, that HAM-Offload uses VEO's means of data transfers to send offload message containing a functor object from the VH to the VE, and then a result message back from the VE to the VH. The VEO benchmark on the other hand uses the native low-level function offload of VEO that can be used to retrieve a function by its symbol name in the VE binary. The latter is not suitable for HAM-Offload, as it is limited to a few basic types for arguments and return types. The message-based approach of HAM-Offload has much richer semantics, as it allows arbitrary types, with the possibility to specify serialisation and deserialisation code on a per-type basis.

With the DMA-enabled protocol, the overhead of HAM-Offload is drastically reduced and the offload is $13.1\times$ faster than a native VEO offload, and $70.8\times$ faster than the HAM-Offload backend using VEO data transfers for message passing. The measured 6.1 µs adds only 5 µs of framework overhead to the 1.2 µs of PCIe round-trip time measured in [4]. Performing the offload from the second CPU, which has to communicate with the VE through its UPI connection with the first CPU socket (Figure 3), adds up to 1 µs to the DMA measurement. How much these numbers affect application runtimes depends on the frequency and granularity of offloading, i.e. the runtime of the offloaded code. In a similar study with the Intel Xeon Phi accelerator [4], a reduction in offloading cost of $13.7\times$ on values of the same order of magnitude translated into speed-up of up to $2.6\times$ for a real world application.

B. Data Transfer Bandwidth

Figure 10 visualises the measured bandwidth for the three available means of communication between VH and VE. Table IV shows the maximum bandwidths observed. To achieve

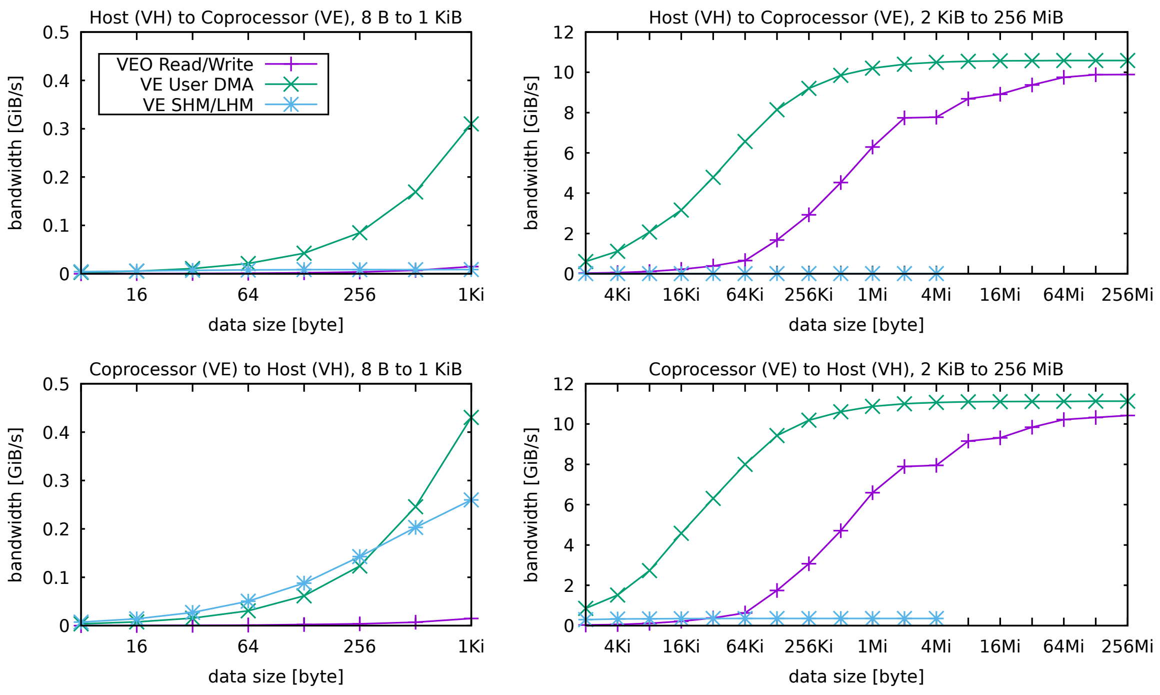

Fig. 10. Bandwidth comparison for copying different amounts of data between VH and VE. The left column shows small data sizes up to 1 KiB, the right one up to 256 MiB. The upper plots show the VH to VE transfers, the lower ones the VE to VH direction. The first data series (VEO Read/Write) represents the VH-initiated VEO-based communication used in Section III-D, while the other two series show the user DMA and special instructions available on the VE side (Section IV-B). The data series for the SHM/LHM instructions was only measured up to 4 MiB due to prohibitive runtimes.

TABLE IV
MAX. PCIE BANDWIDTHS BETWEEN VECTOR HOST (VH) AND VECTOR ENGINE (VE) USING DIFFERENT TRANSFER METHODS (SEE FIGURE 10).

Transfer Method	VH $\Rightarrow$ VE	VE $\Rightarrow$ VH
VEO Read/Write	9.9 GiB/s	10.4 GiB/s
VE User DMA	10.6 GiB/s	11.1 GiB/s
VE SHM/LHM	0.01 GiB/s	0.06 GiB/s

these numbers, it is important to use huge pages of at least 2 MiB.

VE user DMA is always faster than VEO's read and write operation. It achieves close to peak bandwidths already for smaller messages down to 1 MiB, as compared with 64 MiB for VEO. For the VH to VE direction, the bandwidth difference ranges from 24 times for small messages to at least 7 % for large ones. When transferring data from the VE to the VH, VE user DMA is up to 35 times faster for small messages, and again 7 % for the large transfers. When comparing both directions for a given transfer method, transferring data from the VE to the VH is in general faster. The peak bandwidths between the directions differ by up to 5 % for VEO and VE User DMA transfers.

The SHM/LHM instructions of the VE show an interesting behaviour. The load instruction (LHM), used to transfer data from the VH to the VE memory, is only faster than the VE user DMA for writing one or two 64 bit words into the host memory. However, the store instruction (SHM) outperforms VE user DMA for payloads of up to 256 byte, starting with 89 % faster transfer times for a single word down to 16 % for 256 Byte. This could be exploited for small messages, sent from the VE to the VH. Compared with VEO's host initiated read, the VE-issued store instruction is faster for messages up to 32 KiB.

Offloading only pays off as reduced time to solution, if the gain by either faster program execution on the offload target, or by using host and target in parallel, exceeds the offload overhead. The overhead is the cost for offloading the kernel, as well as for transferring data between host and target memory. Our results show that both costs can be significantly reduced by fully utilising the architecture of the NEC SX-Aurora TSUBASA. Lower overhead means that more code of an application becomes a feasible target for offloading, and offloads can become more fine-grained as well. This increases the set of applications that can benefit from such heterogeneous architectures.

In order to verify the portability and usability of HAM-Offload on the NEC SX-Aurora TSUBASA, we built and ran several tests and example applications using HAM-Offload. We could verify that they worked as expected without changing the application code. Of course, most applications will require adaptations inside the offloaded code section to make good use of the NEC Vector Engine, which is outside the scope of this paper.

VI. Conclusion and Outlook

We adapted HAM-Offload, an efficient high-level C++-only offloading framework, for the NEC SX-Aurora TSUBASA. We designed and implemented two communication backends for HAM-Offload, one using VEO from the Vector Host (VH), the other one using user DMA and load/store instruction from the Vector Engine (VE) to transfer the heterogeneous active messages. The cost for offloading a function could be reduced by a factor of over 13× with the VE DMA approach as compared to using VEO's native offload functionality. We also compared different means for data transfers between VH and VE, and achieved up to 11.1 GiB/s transfer rate between VE and VH memory. The findings of this work will be incorporated into future versions of VEO.

Our solution provides a portable, high-level offloading API for the NEC SX-Aurora TSUBASA that is simple to use, and offers the benefits of modern C++. It allows to run existing HAM-Offload applications without changes and has a slim API that makes it easy to port other applications to the model. As soon as NEC's MPI will support heterogeneous jobs, that are combining processes running on the host and on the Vector Engines, HAM-Offload applications will also benefit from remote offloading capabilities, again without changes in the application code. HAM-Offload is open-source and available at GitHub [26].

Acknowledgment

We thank Alexander Reinefeld for helpful discussions. The authors acknowledge the North-German Supercomputing Alliance (HLRN) for providing access to the NEC SX-Aurora TSUBASA A300-8 system operated at ZIB.

References

[1] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[2] Khronos OpenCL Working Group, *The OpenCL Specification, Version 2.2*, October 2018. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf

[3] *OpenMP Application Program Interface, Version 5.0*, OpenMP Architecture Review Board, November 2018. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[4] M. Noack, F. Wende, T. Steinke, and F. Cordes, "A Unified Programming Model for Intra- and Inter-node Offloading on Xeon Phi Clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 203–214. [Online]. Available: https://doi.org/10.1109/SC.2014.22

[5] *The OpenACC Application Programming Interface, Version 2.7*, OpenACC-Standard.org, November 2018. [Online]. Available: https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf

[6] D. Deppisch, "Advancing the heterogeneous active messages approach," Master's thesis, Humboldt-Universität zu Berlin, Faculty of Mathematics and Natural Siences, Department of Computer Science, November 2018.

[7] K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi, "Performance Evaluation of a Vector Supercomputer SX-aurora TSUBASA," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 54:1–54:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291656.3291728

[8] T. Cramer, "NEC and RWTH Aachen University Collaboration: OpenMP Offload Programming Model for SX-Aurora TSUBASA," 2018. [Online]. Available: https://www.hpc.nec/events/isc-18/bof/presentations/05_BOF_OpenMP_VEO_TimCramer.pdf

[9] T. Cramer and E. Focht, "OpenMP Offload Programming Model for NEC SX Aurora TSUBASA," 2018. [Online]. Available: https://www.jara.org/files/jara/bilder/JARA-HPC/Poster%20und%20Screenposter/ISC%202018/12_OpenMP1.pdf

[10] Malý, Lukáš and Zapletal, Jan and Merta, Michal and Čermák, Martin, "Xeon Phi acceleration of domain decomposition iterations via heterogeneous active messages," *AIP Conference Proceedings*, vol. 1978, no. 1, p. 360004, 2018. [Online]. Available: https://aip.scitation.org/doi/abs/10.1063/1.5043963

[11] L. Maly, J. Zapletal, M. Merta, L. Riha, and V. Vondrak, "Evaluation of the Intel Xeon Phi offload runtimes for domain decomposition solvers," *Advances in Engineering Software*, vol. 125, pp. 146 – 154, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0965997817309341

[12] C. Newburn, R. Deodhar, S. Dmitriev, R. Murty, R. Narayanaswamy, J. Wiegert, F. Chinchilla, and R. McGuire, "Offload Compiler Runtime for the Intel Xeon Phi Coprocessor," in *Supercomputing*. Springer Berlin Heidelberg, 2013, pp. 239–254.

[13] C. J. Newburn, G. Bansal, M. Wood, L. Crivelli, J. Planas, A. Duran, P. Souza, L. Borges, P. Luszczek, S. Tomov, J. Dongarra, H. Anzt, M. Gates, A. Haidar, Y. Jia, K. Kabir, I. Yamazaki, and J. Labarta, "Heterogeneous streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 611–620.

[14] C. Chen, F. Yang, F. Wang, L. Deng, and D. Zhao, "Review of Programming and Performance Optimization on CPU-MIC Heterogeneous System," in *2018 IEEE 3rd International Conference on Image, Vision and Computing (ICIVC)*, June 2018, pp. 894–900.

[15] N. Ravi, Y. Yang, T. Bao, and S. Chakradhar, "Semi-automatic Restructuring of Offloadable Tasks for Many-core Accelerators," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 12:1–12:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503285

[16] M. Klemm and J. Enkovaara, "pyMIC: A Python Offload Module for the Intel Xeon Phi Coprocessor," in *4th Workshop on Python for High Performance and Scientific Computing, New Orleans, LA*, 11 2014.

[17] J. Hahnfeld, C. Terboven, J. Price, H. J. Pflug, and M. S. Müller, "Evaluation of Asynchronous Offloading Capabilities of Accelerator Programming Models for Multiple Devices," in *Accelerator Programming Using Directives*, S. Chandrasekaran and G. Juckeland, Eds. Cham: Springer International Publishing, 2018, pp. 160–182.

[18] A. Capotondi, A. Marongiu, and L. Benini, "Runtime Support for Multiple Offload-Based Programming Models on Clustered Manycore Accelerators," *IEEE Transactions on Emerging Topics in Computing*, vol. 6, no. 3, pp. 330–342, July 2018.

[19] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer, "A uniform approach for programming distributed heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3228 – 3239, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001440

[20] "LLVM for SX-Aurora TSUBASA Vector Engine," 2018. [Online]. Available: https://github.com/SXAuroraTSUBASAResearch/llvm

[21] *Itanium C++ ABI, v1.86*, Intel Corporation.

[22] M. Matz, J. Hubika, A. Jaeger, and M. Mitchell, *System V Application Binary Interface, AMD64 Architecture Processor Supplement, Draft v0.99.6*.

[23] *System V Application Binary Interface VE Architecture Processor Supplement, v1.1*, NEC Corporation. [Online]. Available: https://www.nec.com/en/global/prod/hpc/aurora/document/VE-ABI_v1.1.pdf

[24] E. Focht, "Release v1.3.2-4dma: Experimental DMA acceleration and external profiling," 2018. [Online]. Available: https://github.com/efocht/build-veos/releases/tag/v1.3.2-4dma

[25] H. Nakamura, H. Takayama, Y. Yamaguchi, and T. Boku, "Thorough analysis of PCIe Gen3 communication," in *2017 Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2017, pp. 1–6.

[26] M. Noack, "HAM-Offload GitHub Repository," 2019. [Online]. Available: https://github.com/noma/ham