Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

MATTHIAS NOACK

HAM - Heterogenous Active Messages for Efficient Offloading on the Intel Xeon Phi

Preprint

ZIB-Report 14-23 (June 2014)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7
D-14195 Berlin-Dahlem

Telefon: 030-84185-0
Telefax: 030-84185-125

e-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

HAM - Heterogenous Active Messages for Efficient Offloading on the Intel Xeon Phi

Matthias Noack

Abstract

The applicability of accelerators is limited by the attainable speed-up for the offloaded computations and by the offloading overheads. While GPU programming models like CUDA and OpenCL only allow to optimise the application code and its speed-up, the available low-level APIs for the Intel Xeon Phi provide opportunity to address the overheads, too. This work presents an Heterogeneous Active Message (HAM) layer that minimises software overheads for offloading on Intel's Xeon Phi. It provides the basis for an offload API with similar semantics as the Intel Language Extensions for Offload (LEO). In contrast to LEO, HAM works within the C++ language and needs no additional compiler support. We evaluated HAM on top of SCIF and MPI as communication backends. While the SCIF backend offers the best performance, the MPI backend allows for inter-node offloads which are not possible with other offload solutions. Benchmark results show that the cost for offloading a function call can be decreased by a factor up to 18 compared with LEO.

1 Introduction

Offloading is the standard programming model for accelerators such as GPGPUs and Intel's Xeon Phi. It is implemented in frameworks like CUDA, OpenCL, OpenACC, and also in the recently released OpenMP 4.0. Their common abstraction is a kernel, i.e. a region of code, whose execution is offloaded to the accelerator. Whether or not it is worthwhile to offload a computation is determined by the attainable speed-up and by the overhead costs for offloading. Assuming a computation runs faster on an accelerator, then there is a minimal amount of work that needs to be offloaded for the overheads to amortise. This minimal amount of work imposes a lower bound on the granularity of offloaded computations and thus limits the range of applications that can benefit from accelerators.

The offload frameworks mentioned above only allow to optimise the application code, i.e. the speed-up. The overheads are out of reach, hidden in the runtime environment. The Intel Xeon Phi, which is an x86 many-core coprocessor, has changed this situation. While GPGPUs are only accessible through their offload frameworks, the Xeon Phi runs a Linux operating system and offers low-level APIs for communication between host and coprocessor. This new level of flexibility opens new opportunities for research and allows to develop novel mechanisms for heterogeneous programming.

Preliminary benchmarks of Intel's offload implementation show a gap between the achieved performance and its theoretical optimum which is bounded by the latency and bandwidth of the communication channel between host and coprocessor. Especially latency-critical offloads, that do not involve large data transfers, could benefit from a reduced overhead.

That is the starting point of this work, which tries to reduce software overheads that currently limit the use of accelerators. We developed an Heterogeneous Active Message (HAM) layer that introduces minimal overheads while still offering a high-level of abstraction. This layer is the base for an alternative offload API, to which we refer to as HAM-Offload. It offers the same functionality as existing solutions and thus allows for a direct comparison. We evaluate HAM-Offload with MPI and Intel's Symmetric Communications InterFace (SCIF) as communication backends. The results show: overheads for an empty function call can be reduced by a factor of up to 18 compared with Intel's Language Extensions for Offload [1].

Besides the reduced overhead, HAM-Offload offers some additional advantages. For instance, the implementation uses standard C++ without the need of compilers-specific language extensions. With MPI as backend, remote offloading between different nodes is possible. Reverse offloading [2], where the main programme runs on the coprocessor and offloads tasks that cannot be accelerated to the host, is another option. In fact, the main programme can run on any host or coprocessor and offload to any host or coprocessor within the same node (SCIF) or across nodes (MPI).

The rest of this paper is structured as follows. Section 2 gives background information on the Xeon Phi, offloading overheads and Intel's offload solution. In Section 3, we present HAM, the offload API on top of it, and the communication backends. Benchmark results are presented and discussed in Section 4. Related work is listed in Section 5. The work closes with conclusions and future work in Section 6.

2 Background

2.1 Intel Xeon Phi

The Intel Xeon Phi is an x86-compatible many-core coprocessor that is based on the Intel Many Integrated Cores (MIC) Architecture. It offers up to 61 in-order cores with 512 bit wide SIMD units. Each core exposes 4 hardware threads, of which at least 2 are generally needed to fully utilise the device. The Intel Xeon Phi 7120 series provides 1.2 TFLOPS double precision performance and a memory bandwidth of 352 GB/sec at 300 W. Host and Xeon Phi are connected via PCIe 2.0. The coprocessor runs a Linux operating system that is fully accessible by the user. The Intel Manycore Platform Software Stack (MPSS) contains the software to work with the Xeon Phi. There are multiple high-level programming models available, e.g. MPI, Intel Language Extensions for Offload (LEO), OpenMP, and the Intel Math Kernel Library (MKL). Applications can either run on the host and *offload* computations to the coprocessor, run in a *symmetric* mode on host and coprocessor (using MPI), or run *natively* on the coprocessor only. Libraries like the MKL also offer automatic offloading, such that the coprocessor becomes transparent to the application programmer.

An extensive guide to programming the Xeon Phi can be found in [1]. The low-level communication API, which is used directly or indirectly by most other components of the software stack, is the Symmetric Communication InterFace (SCIF). An InfiniBand Verbs implementation on top of SCIF (IB-SCIF) provides compatibility with existing code.

2.2 Offloading Overheads

Offloading overheads comprise everything that has to be done in order to perform a calculation on an accelerator that would not have been necessary, if the computation would be carried out on the host instead. This includes data transfers for operands, results, and possibly programme code, as well as coordination and synchronisation between host and accelerator. These overheads are inevitable, but they can be minimised up to some lower bound that is determined by hardware characteristics like latencies and bandwidths of buses and network connections. A more realistic definition for this bound also includes the software overheads of the vendor software stack for the communication devices.

The offload overhead can be quantified by comparing the offload implementations performance with the corresponding values for the raw communication API. The round-trip time corresponds to an empty offload call which takes at least one round-trip plus any further overhead. The time for transferring an amount of data via the communication API can be directly compared with the time for transferring the same amount of data via the offload API.

2.3 Intel Language Extensions for Offload

This section provides some insights into the mechanisms behind Intel's Language Extensions for Offload (LEO). A more detailed description of the offload compiler runtime used by LEO can be found in [3]. Another source for details are the Intel Coprocessor Offload Infrastructure (COI) sources which come with documentation and examples as part the MPSS package. LEO is part of the Intel compiler and is not open source.

LEO provides `pragma` directives for offloading code regions to Xeon Phi coprocessors and to transfer data between host and coprocessors. The compiler translates those directives into calls to the COI library. Offloaded code regions are thereby expanded into uniquely named functions that contain additional helper code (e.g. for unpacking input data). The unique names of the offload functions, together with their addresses, are collected in a look-up table. The executable binary contains the host code and the Xeon Phi code for offloaded code regions.

At runtime, the initialisation of COI creates a process on the coprocessor from a generic start-up image that performs further initialisation at the coprocessor side. The embedded coprocessor code from the host binary is transferred to the device and then loaded like a shared library by the coprocessor process. COI also resolves dependencies with additional shared libraries by copying their coprocessor version to the Xeon Phi.

The invocation of offloaded code is basically performed by passing the unique name of the corresponding offload function, along with input data, to a thunk object via COI's invocation mechanism. The communication between host

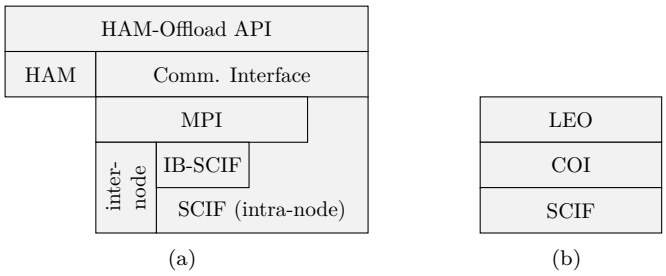

Figure 1: The layer architectures of a) HAM-Offload, and b) Intel LEO.

and coprocessor works via a FIFO command pipeline. On the coprocessor, the invocation is then carried out by the thunk object that uses the compiler-generated look-up table to translate the unique function name into an address which is called subsequently. This invocation mechanism contains avoidable overheads, like transferring function names as strings and using them as keys for table look-ups.

Memory allocation, deallocation and transfer is also performed via a set of COI routines. Internally, COI uses RMA via SCIF (see Section 3.3) for communication. For the best performance it is necessary to align source and target buffer at cache-line boundaries (64 byte). Figure 1(b) depicts LEO and its underlying layers. Intel's OpenMP 4.0 implementation uses the same mechanisms, but lacks some of LEO's flexibility, like keeping allocated memory across offloaded code regions. Hence, the results regarding LEO also apply to OpenMP.

3 Heterogeneous Active Messages for Offloading

This section describes the architecture of HAM, the HAM-Offload mechanism, and the communication backends. Figure 1(a) depicts the layer architecture of the whole system.

3.1 HAM

HAM is a C++ template library that provides the means to create type-safe heterogeneous active messages that can be transferred via any reliable communication channel. Although it is motivated by efficient offloading for the Xeon Phi, HAM is designed as an independent software layer and can be used in any context where such functionality is needed.

In conventional message passing systems, messages are passive pieces of data. Active messages instead, are units of execution. That is, active messages process themselves. A common and efficient way to implement this concept is to include the code address of a handler function into the message. When a message is received, the code at the embedded handler address is executed. The address of the message's payload is passed as an argument to its handler. In object-oriented active message systems, where messages are callable objects, the handler's job is

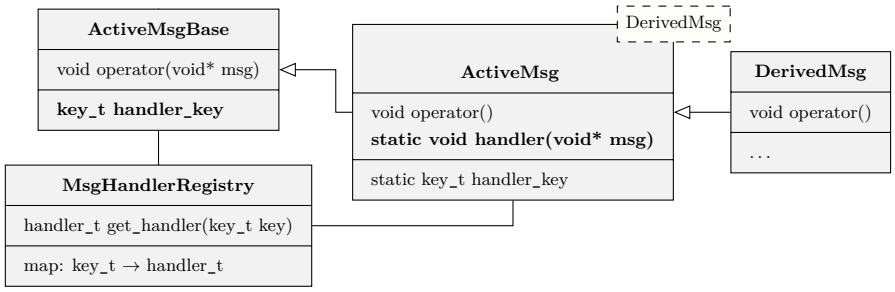

Figure 2: This simplified class diagram of the Heterogeneous Active Message implementation depicts the basic active message type structure.

to perform a type conversion from the typeless bytes of a network buffer back to an object of the actual message type, which can then be called like a function.

The problem here is to know the remote handler address when sending a message. The TACO framework [4] for example, solves this by using the exact same binaries for all processes. This way, the handler addresses are the same on every node and can be safely transferred and executed without any further effort. With the Xeon Phi in the picture, this approach is no longer viable because coprocessor and host use different binaries with differing addresses. The question now is, how to translate between the handler addresses of those heterogeneous binaries – with minimal cost. This problem is solved by HAM. The basic idea is to add a level of indirection by sending some reference to the handler inside each message that can be converted to the respective address by the receiver. This obviously requires some kind of look-up table or map. The technical novelty is the way this mechanism is efficiently implemented in pure C++ without the need for a language extension.

Figure 2 provides a simplified image of HAM's architecture. The basic components are a class template that all active messages have to inherit from, a registry for message handlers, and execution policies (not in the figure). The latter contain the actual handler function. The default policy simply executes the message. A framework on top of HAM can add new policies as needed, for instance to execute each message in a new thread, to interact with a runtime environment, or to handle some kinds of messages differently than others.

The message handler registry encapsulates the translation process and acts like a map from a key to a handler address. At programme initialisation, the registry must be filled with entries on all processes. Instead of transferring an actual handler address, each active message contains the key for its handler. The handler key must be set at the sending side and must be translated back into a handler address at the receiving side. The difficult part here is to do both steps with a time complexity of $O(1)$. Each instantiation of the active message template registers its handler via static member initialisation. This happens in an undefined order before the programme's `main()`. The handler address is registered together with the handler's C++ typeid-name. The typeid-name is not defined by the C++ language standard, but is usually defined within the ABI. Both, host [5] and coprocessor [6] ABI refer to the Intel IA-64 ABI [7] for C++. It defines the typeid-name as the mangled type name whose format is also defined by this ABI. This guarantees identical handler names on host and coprocessor.

At the beginning of each processes `main()` function, an initialisation call to HAM performs the second part of the initialisation. The handler addresses are now written into an array in lexicographical order of their typeid-name. At this point, each process has an ordered array of its handler addresses where the handler order is the same for all processes – without any communication. The handler key is an index to this array and is defined by the handler address position in the array. This way, the translation from handler key to address on the receiving side is a simple array element access, which is $O(1)$. Still during initialisation, a connection between handler key and active message type is established by writing the key into a static member variable of its message type. Now each time an active message is constructed on the sending side, the handler key is simply copied into a non-static member – in $O(1)$.

3.2 HAM-Offload API

The HAM-Offload API is a thin layer on top of HAM. It offers the same basic functionality as LEO's offload `pragma` directives. This allows a direct comparison of their offload efficiency. Listings 1 and 2 illustrate the most important offload operations in LEO and HAM-Offload.

The different methods of the HAM-Offload API use a set of predefined active message templates, for offloading function calls, memory allocation/deallocation, and for initiating data transfers. The `f2f()` functor generator (f2f is short for function-to-functor) provides a convenient way to create functor types for offloading function calls. It works similar to `std::bind()` and takes the address of a function, and a list of arguments. The difference is, that the generated functor object provides the means to be bit-wise transferred. Therefore, the contained arguments are wrapped by a `Migratable` template which can be constructed from and converted to the wrapped object's type. This offers the necessary hooks for serialisation and deserialisation. A specialised Migratable instance can be provided for any type (e.g. strings, or containers) and also be used to flag types as not offloadable. This allows to safely handle arbitrary types which is not possible with LEO or OpenMP. Internally, the functor type is used to instantiate an active offload message that, when executed, calls the functor and handles the transfer of its return value. The address of the function inside an `f2f()`-generated functor is part of the functor type not its value, so it is translated implicitly in the process of message execution. Once the type of the received message is restored by its handler, the local function address is automatically used. Figure 3 visualises the offload mechanism.

Listing 1: Intel LEO Offload examples

```
// tell the compiler to generate coprocessor code for fun_mul()
__attribute__((target(mic)))
float fun_mul(float a, float b) { ... }

float result = ..., a = ..., b = ...;
// offload a call to fun_mul() to device 0
#pragma offload target(mic:0) in(a, b) out(result)
{ result = fun_mul(a, b); }

// allocate cache-line aligned memory (64 byte) on host
size_t n = ...; float* buffer = ...;
// allocate memory on coprocessor
#pragma offload target(mic:0) nocopy(buffer:length(n) alloc_if(1)
    free_if(0))
// free memory on coprocessor
#pragma offload target(mic:0) nocopy(buffer:length(n) alloc_if(0)
    free_if(1))
// copy data from host to coprocessor into a pre-allocated buffer
#pragma offload target(mic:0) in(buffer:length(n) alloc_if(0)
    free_if(0))
// copy data from coprocessor to host from a coprocessor buffer
#pragma offload target(mic:0) out(buffer:length(n) alloc_if(0)
    free_if(0))
```

Listing 2: HAM-Offload examples

```
// create a functor that binds fun_mul() with arguments
auto functor = f2f(&fun_mul, a, b);
// perform the offload to device 1
float result = Offload::sync(1, functor);

// allocate cache-line aligned memory (64 byte) on host
size_t n = ...; float* buffer_host = ...;
// allocate memory on coprocessor
auto buffer_mic = Offload::allocate<float>(1, n);
// free memory on coprocessor
Offload::free(1, buffer);
// copy data from host to coprocessor into a pre-allocated buffer
Offload::write(1, buffer_mic, buffer_host, n);
// copy data from coprocessor to host from a coprocessor buffer
Offload::read(1, buffer_mic, buffer_host, n);
```

It would be easily possible to enhance the offload API to support more object-oriented constructs which are not part of LEO. For instance, object construction could be offloaded to directly allocate objects on the coprocessor and to subsequently offload method calls on those objects. This basically leads to a global object space across host and coprocessors.

In contrast to LEO, all processes run a complete binary of the programmes source code and are created during programme start-up. The programme's `main()` only runs on the first process, all others receive and execute offload messages. Like LEO, HAM-Offload does not offer the means for parallelism and relies on the use of other frameworks inside the offloaded code. Of course, it is possible to create multiple processes on the coprocessors. In fact, offload processes can be created on the host too. Also roles can be switched for reverse offloading, such that a coprocessor runs the `main()` and offloads are processed by the host. This might be useful when the host is merely needed for I/O tasks.

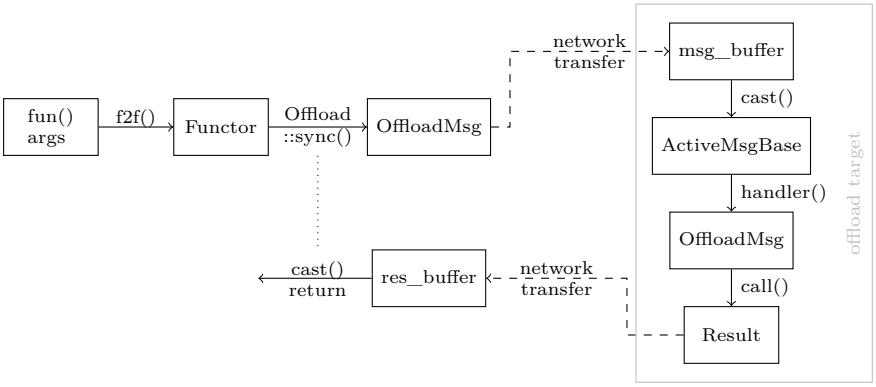

Figure 3: This figure illustrates how offloading a function call works in HAM-Offload.

3.3 Communication Backends

We have implemented two communication backends, one using MPI, the other one using SCIF. Each communication backend presents the same interface to the HAM-Offload layer which includes the means to address other processes, to transfer active messages, and to transfer data. The interface also offers the means to allocate memory which is already registered for RMA. Data transfers are initiated and synchronised via active control messages.

MPI

The MPI backend implementation is very straight-forward, since MPI already offers everything that is needed. Transferring active messages as well as transferring data is performed via point-to-point operations. Internally, any MPI implementation uses SCIF either directly or indirectly via the IB-SCIF InfiniBand Verbs interface. In contrast to SCIF, MPI also offers inter-node communication which allows for offloading to remote Xeon Phis.

SCIF

SCIF is the fastest, but also the most low-level API for intra-node communication. It offers a sockets-like API for establishing connections and simple send/receive operations. Additionally, there is a set of RMA operations which allow to register memory for DMA, to perform read and writes, and to map remote memory into the virtual address space of a process.

For performance reasons, the implemented SCIF backend mainly uses RMA. The send and receive operations are only used during initialisation to exchange the offsets of registered RMA buffers. Since the active message size typically ranges from less than a hundred byte up to a few KiB (if multiple arguments of complex types are involved), latency is the critical factor. The best latency over SCIF can be achieved by writing into mapped remote receive buffers.

Figure 4 shows the protocol our SCIF backend uses for active message transfer. The receiver polls on a flag, which is written by the sender after the receive buffer. Accesses to mapped remote memory must be ordered by memory fences which also ensure the visibility of changes by flushing the write-combining buffer

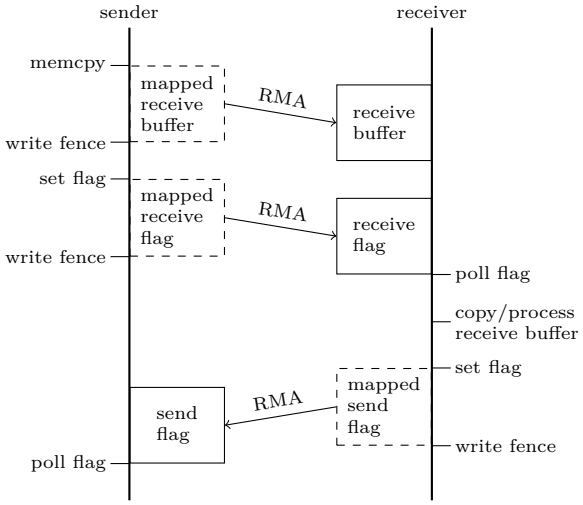

Figure 4: The RMA-based protocol that is used for message transfers over SCIF in HAM-Offload. Polling the send flag makes sure that the corresponding receive buffer can be safely re-used. This can also be deferred up until the next send to the same buffer.

of the host. Once the receive buffer can be safely re-used by the sender, the receiver sets the corresponding flag on the sending side.

4 Evaluation

All measurements were taken on a 2-socket system with Intel Xeon E5-2670 CPUs and 64 GiB RAM that was equipped with two Intel Xeon Phi 7120P coprocessors (16 GiB, 1.238 GHz, 61 cores). The test system was running CentOS 6.3 with the Intel C++ compiler in version 14.0.1. Intel MPSS was installed in version 2.1.6720-19. The MPI benchmarks were performed with Intel MPI in version 4.1.1. For all measurements, threads were pinned on both sides. The function call benchmarks were performed 100 000 times, the bandwidth benchmarks for at least 1 000 times depending on the message size.

In order to quantify the software overhead for offloading code regions, we measured the time for offloading an empty function call and the round-trip times for SCIF and MPI between the host and a local Xeon Phi. An offload requires at least one round-trip from host to coprocessor and back, so everything above the round-trip time is overhead introduced by the offload framework. Figure 5 shows the measured times. Table 1 contains the framework overhead as percentage of the overall offload cost, as percentage of the round-trip time, and the speed-up of HAM compared to LEO.

The framework overhead measured for LEO is 95 %. HAM is able to reduce this to less than 13 %. With respect to the overall offload cost, a single offload with LEO causes as much overhead as roughly 18 offloads with HAM. The absolute overhead for HAM with SCIF is around 300 ns. Even with MPI, that has a 10 times higher round-trip time than SCIF, the overall offload overhead nearly halves compared to LEO. One reason for this difference is the efficient

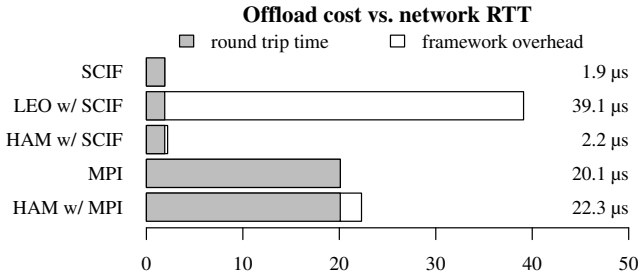

Figure 5: The measured times for offloading an empty function call compared to the round-trip time of the communication channel (SCIF and MPI). The difference is the overhead introduced by the offload framework.

	framework overhead relative to offload cost	**RTT**	**speed-up vs. LEO**
LEO w/ SCIF	95.1 %	1922.0 %	1.0×
HAM w/ SCIF	12.9 %	14.8 %	17.6×
HAM w/ MPI	9.8 %	10.9 %	1.8×

Table 1: The first tow columns shows the software overhead relative to the overall offload cost (see Figure 5) and relative to the round-trip time (RTT). The third column contains the speed-up of an empty offload relative to LEO.

active message mechanism of HAM that optimises the code invocation on the coprocessor (see Section 3). Also, the HAM-Offload runtime is extremely thin compared to COI. The latter might change when we investigate real-world applications and extend HAM-Offload, e.g. by introducing additional threads for message receiving and processing.

While latency is the critical factor for offloading code execution, bandwidth is important for transferring input and output data before and after executing a kernel. We measured the bandwidth for data transfers with LEO, HAM (with SCIF and MPI), and pure MPI using the Intel MPI benchmark. For SCIF, there is no standard benchmark. The results for data transfers between host and coprocessor and vice versa are plotted in Figure 6.

LEO shows the best performance for small buffers up until 1 MiB at which point the performance drops, probably due to switching protocols for larger buffers. Our SCIF backend does not show this behaviour and hence is faster for a buffer sizes from 2 to 16 MiB. At that point, LEO becomes faster again while our SCIF backend stagnates around 5 GiB/s without saturating the hardware. A look into the source code of the COI layer between LEO and SCIF revealed no obvious difference, that would explain the measured bandwidth differences. Both COI and HAM's SCIF backend use the same RMA operations provided by SCIF. However, a detailed analysis of the protocols employed by COI should provide an answer. It also might be possible, that LEO defers parts of the actual operation by buffering data on the host. Improving the achievable bandwidth of the SCIF backend is an important issue with respect to applications. Otherwise, the reduced kernel invocation overhead might be (over-)compensated by the

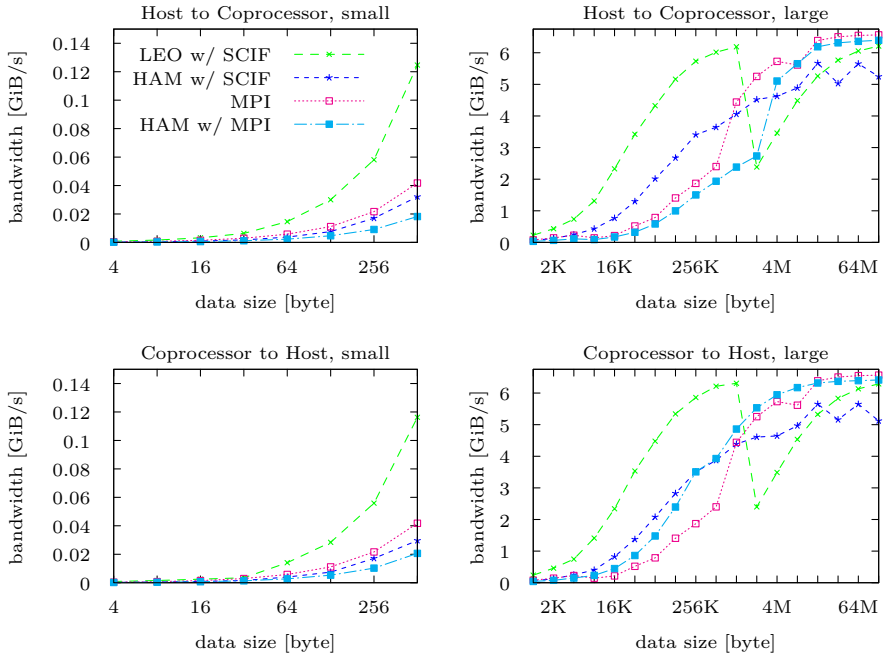

Figure 6: Bandwidths for data transfers from and to the coprocessor via offload and direct use of the communication channel.

reduced bandwidth – depending on the buffer size.

A major difference between host-to-coprocessor and coprocessor-to-host bandwidth only emerged with the MPI backend. In this transfer direction, it even outperforms the SCIF backend for buffer sizes larger than 256 KiB for coprocessor-to-host transfers. The faster bandwidth of the MPI backend compared with the Intel MPI benchmark (IMB) can be explained as follows. The IMB PingPong benchmark always measures symmetrically by sending the same amount of data in both directions. Hence, the IMB results have to be interpreted as the average between both transfer directions.

5 Related Work

To the best of our knowledge, this work is the first that directly attends to offload overhead minimisation for the Xeon Phi. The functor-based active message concept of HAM is inspired by the TACO framework [4] in whose continuous development the author is involved. TACO implements a Global Object Space (GOS), but does not support heterogeneous systems. With some modifications, HAM could be used to substitute TACO's own active message layer. Another active message based framework that offers the abstraction of an Active Global Object Space (AGAS, which is synonymous with GOS) is HPX [8]. It supports the Xeon Phi since version 0.9.6 and it seems possible to implement functor based offloading similar to HAM-Offload with HPX. In [9], HPX is used as backend

for LibGeoComp (a library for geometric decomposition) and also evaluated on the Xeon Phi. However, the results focus on the parallel efficiency and hence cannot be compared with our results.

Results on leveraging SCIF for efficient intra-node communication for the MVAPICH MPI implementation can be found in [10]. For improved inter-node communication, a proxy-based framework for MVAPICH is presented in [11]. The MPI communication between two Xeon Phi coprocessors is addressed in [12]. All three works contain improvements over Intel MPI which we used. So HAM-Offload over MPI can be expected to benefit from advances in MPI implementations for local and remote offloads between any combination of hosts and coprocessors. The Intel Offload Compiler Runtime which, is used by LEO and the offload primitives in Intel's OpenMP 4.0 implementation, is described in [3] and in Section 2.3.

6 Conclusion

We have introduced Heterogeneous Active Messages (HAM) as means for efficient offloading to Intel's Xeon Phi. Our approach effectively reduces offload overheads while it offers a high-level C++ API that, unlike existing frameworks, is not a language extension. We evaluated offloading via HAM with MPI and SCIF as a communication backend and compared the results with Intel's Language Extensions for Offloading (LEO). The overall overhead per offload is reduced by a factor of up to 18 compared with LEO, i.e. the minimal amount of work that benefits from being offloaded is also reduced by a factor of up to 18. This is a big step towards fine-grained offloading which makes heterogeneous computing accessible to new applications. The achieved bandwidth for data transfers between host and coprocessor is not yet satisfying. While there are buffer sizes for which our backends outperforms LEO, the latter is significantly faster for small buffers. Since Intel's COI is open source, an analysis of the employed protocols should reveal how to use SCIF in order to reproduce the same performance results as LEO. There is further potential in the exploration of efficient hybrid MPI implementations to narrow the gap between MPI and SCIF for intra-node communication [10, 11].

As a next step, we plan to investigate HAM with real-world applications and to further develop the HAM-Offload API in the process. We plan to add runtime components to address parallelism on the coprocessor as well. Using the MPI-backend, we will evaluate remote offloading to multiple Xeon Phis, as well as less common offload patterns like reverse offloading, or host-to-host and Phi-to-Phi offloading. The offload layer is just one possible abstraction for programming the Xeon Phi via Heterogeneous Active Messages. Lifting the distinction between host and offload-target would maximise flexibility and allow for arbitrary application patterns build from processes that exchange and process active messages.

Acknowledgements

The Intel Xeon Phi nodes and Intel TrueScale Fabric are kindly donated by Intel for the "Research Center for Many-core High-Performance Computing" at ZIB. This work was partly supported by the North German Supercomputer Alliance

HLRN. We would also like to thank Thorsten Schütt for fruitful discussions and valuable suggestions for improving this document.

References

[1] Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High Performance Programming. Elsevier Science (2013)

[2] Pakin, S., Lang, M., Kerbyson, D.: The reverse-acceleration model for programming petascale hybrid systems. IBM Journal of Research and Development **53**(5)

[3] Newburn, C., Deodhar, R., Dmitriev, S., Murty, R., Narayanaswamy, R., Wiegert, J., Chinchilla, F., McGuire, R.: Offload compiler runtime for the intel xeon phi coprocessor. In: Supercomputing. Springer Berlin Heidelberg (2013)

[4] Nolte, J., Ishikawa, Y., Sato, M.: TACO: prototyping high-level object-oriented programming constructs by means of template based programming techniques. SIGPLAN Not. **36** (December 2001) 35–49

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

[6] Lu, H., Girkar, M., Matz, M., Hubika, J., Jaeger, A., Mitchell, M.: System V Application Binary Interface, K1OM Architecture Processor Supplement, v1.0

[7] Intel et al.: Itanium C++ ABI, v1.86

[8] Kaiser, H., Brodowicz, M., Sterling, T.: Parallex an advanced parallel execution model for scaling-impaired applications. In: Parallel Processing Workshops, 2009. ICPPW '09. International Conference on. (Sept 2009) 394–401

[9] Heller, T., Kaiser, H., Schäfer, A., Fey, D.: Using hpx and libgeodecomp for scaling hpc applications on heterogeneous supercomputers. ScalA '13, New York, NY, USA, ACM (2013) 1:1–1:8

[10] Potluri, S., Venkatesh, A., Bureddy, D., Kandalla, K., Panda, D.: Efficient intra-node communication on intel-mic clusters. In: CCGrid 2013 13th IEEE/ACM International Symposium on. (2013) 128–135

[11] Potluri, S., Bureddy, D., Hamidouche, K., Venkatesh, A., Kandalla, K., Subramoni, H., Panda, D.K.D.: Mvapich-prism: A proxy-based communication framework using infiniband and scif for intel mic clusters. In: Proceedings of SC13. SC '13, New York, NY, USA, ACM (2013) 54:1–54:11

[12] Si, M., Ishikawa, Y., Tatagi, M.: Direct mpi library for intel xeon phi co-processors. In: IPDPSW, 2013 IEEE 27th International. (May 2013) 816–824