DPU Offloading Programming with the OpenMP API

MUHAMMAD USMAN, SERGIO ISERTE, ROGER FERRER, and ANTONIO J. PEÑA, Barcelona Supercomputing Center (BSC), Spain

Data processing units (DPUs) as network co-processors are an emerging trend in our community, with plenty of opportunities yet to be explored. These have been generally used as domain-specific accelerators transparent to application developers; In the HPC field, DPUs have been used as MPI accelerators, but also to offload some tasks from the general-purpose processor. However, the latter required application developers to deploy MPI ranks in the DPUs, as if they were remote (weak) compute nodes, hence considerably hindering programmability. The wide adoption of OpenMP as the threading model in the HPC arena, along with that of GPU accelerators, is making OpenMP offloading to GPUs a wide trend for HPC applications. In this paper we introduce, for the first time in the literature, OpenMP offloading support for network co-processor DPUs. We present our design in LLVM to support OpenMP standard offloading semantics and discuss the programming productivity advantages with respect to the existing MPI-based programming model. We also provide the corresponding performance analysis demonstrating competitive results in comparison with the MPI baseline.

CCS Concepts: • **Computing methodologies** → *Distributed computing methodologies*.

Additional Key Words and Phrases: BlueField, DPUs, ODOS, OpenMP Offloading

ACM Reference Format:
Muhammad Usman, Sergio Iserte, Roger Ferrer, and Antonio J. Peña. 2023. DPU Offloading Programming with the OpenMP API. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3624062.3624165

1 INTRODUCTION

Data Processing Units (DPUs) are a new class of programmable processor system on a chip (SoC) that combine software-programmable processing elements (CPUs), a high-performance network interface, and other acceleration engines. The main manufacturer of DPUs in production is NVIDIA, which provides them with a specific software development kit (SDK) for their BlueField DPU devices, DOCA.

DPU devices were initially designed to accelerate common datacenter operations next to the network, such as I/O or encryption. However, the presence of general-purpose programmable CPUs opens the door to further opportunities in the HPC field. So far, these have been explored to offload MPI operations [9], but also to execute general-purpose computing [5]. However, the current programming opportunities involve (1) using the low-level DOCA APIs, or (2) leveraging the fact that these DPUs can run as self-hosted devices and be exposed as regular compute nodes in the network. While the former is too low-level for domain developers (with all the implications that brings), the latter is, as well, far from ideal from the point of view of programming productivity and portability: application developers have to deal with MPI ranks executing in two widely different processor architectures yielding considerably different performance, which is unnatural for the MPI programming model.

One of the most widely used programming models in the HPC arena is OpenMP, which is used to leverage data parallelism in multi-core processor architectures. Starting with version 4.0[1], the OpenMP Standard incorporated capabilities to offload parallel computation to devices. This support has been materialized mainly for GPU computing, and is being widely adopted. In this work, we have implemented support for OpenMP offloading to NVIDIA BlueField DPUs in LLVM. Our proposal, which we name ODOS (OpenMP DOCA Offloading Support), supports the standard OpenMP offloading syntax and semantics. The only difference with respect to traditional OpenMP GPU offloading support lies on the fact that, since we target multi-core processors instead of a massively-parallel architecture, users are expected to leverage traditional parallelization semantics (e.g., `parallel for`) instead of those specific for embarrassingly parallel devices, such as GPUs (e.g., `teams distribute`).

In this paper, after describing the ODOS approach, we discuss the programmability of targeting DPU devices by the existing MPI approach versus OpenMP offloading as enabled by ODOS, concluding that OpenMP offloading is a considerably more natural and easier approach for application developers. Our solution is evaluated with a series of benchmarks that prove the concept of DPU offloading with OpenMP via the native BlueField SDK (DOCA), yielding competitive performance with the baseline based on MPI.

The recent emergence of the brand-new BlueField 3 devices[2], much more powerful than their predecessor, supporting PCIe 5.0 with up to 400 Gb/s bandwidth capacity, 16 ARM A78 processors, and 32GB DDR5 RAM, opens the door to benefit a much wider range of applications than those already reported for BlueField 2 devices. Furthermore, the fourth generation is already announced and it is expected to feature further bandwidth and computational power. We believe that this paper, along with the ODOS software, added to the new capabilities of BlueField 3 devices, have the potential to spark much broader research on the use of DPU co-processors in the field of HPC: The use of OpenMP shall lower considerably the efforts to program these devices. We believe that all this together has the potential to spark a new trend in HPC computing.

The rest of the paper is structured as follows. Section 2 briefly introduces the hardware devices and the related technology. Section 3 summarizes the current state–of–the–art on using DPUs in HPC environments. Section 4 thoroughly describes the implementation of the backend and the necessary plugins for DPU offloading with OpenMP in LLVM. Section 5 evaluates the presented tool and measures performance on different benchmarks. Section 6 concludes this work and overviews future research plans.

2 BACKGROUND

In this section, we first introduce the hardware we target in our work. Next, native general-purpose programming possibilities as provided by the vendor are described.

2.1 BlueField DPUs

BlueField DPU is a product line of data processing units developed by NVIDIA [2]. It combines the capabilities of data processing with integrated networking functionality. These integrate ARM CPU cores, networking interfaces, and domain-specific accelerators onto a single chip. This integration allows for a highly efficient and streamlined approach to data processing in data center environments. By combining networking and processing capabilities, BlueField DPUs can offload tasks traditionally performed by separate components, potentially resulting in improved performance, reduced latency, and increased overall system efficiency.

[1]https://www.openmp.org/uncategorized/openmp-40/
[2]https://resources.nvidia.com/en-us-accelerated-networking-resource-library/datasheet-nvidia-bluefield

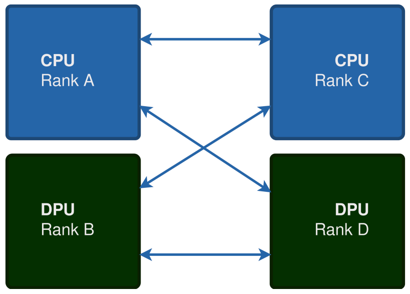

Fig. 1. Simple diagram illustrating the deployment of different MPI ranks in CPU and DPU processors.

The networking capabilities of BlueField DPUs include high-speed InfiniBand interfaces and support for various networking protocols. This enables efficient packet processing, network virtualization, and enhanced security features. The integrated ARM CPU cores provide general-purpose processing power that can run an operating system and applications. The relatively widely-available BlueField 2 DPUs feature PCIe 4, 8 ARM A72 cores, and 32GB of on-board DDR4 DRAM, while the more recent—and still scarce—BlueField 3 version support PCIe 5 and equip 16 ARM A78 processors and 64GB DDR5 DRAM.

2.2 Programming BlueField DPUs

2.2.1 Datacenter on a Chip Architecture (DOCA). DOCA[3] allows developers to leverage the capabilities of NVIDIA BlueField DPUs and provides industry-standard open application programming interfaces (APIs) and frameworks for networking, security, storage, HPC, or artificial intelligence (AI) applications. The frameworks simplify application offload with integrated NVIDIA acceleration packages.

DOCA includes a rich set of APIs and abstractions that simplify the development of networking and security applications. With DOCA, developers can benefit from the programmability and acceleration capabilities of BlueField DPUs to implement advanced networking features such as virtual switching, virtual routing, load balancing, and firewalling. It provides a high-performance data path and a flexible programmable pipeline that allows for customized packet processing.

2.2.2 MPI. Since DPU devices may run an operating system and be exposed to the network as separate compute nodes, deploying MPI processes in those is currently a possibility to execute general-purpose code on these devices. Hence, these feature separate rank numbers, as depicted in Figure 1. Applications may deploy an intercommunicator via the MPI connect/accept semantics, or execute an MPI MPMD (multiple-program multiple-data) session, usually via the ":" syntax of common MPI launchers. Anyhow, CPU and DPU execute separate MPI rank numbers, which poses a burden to application developers, which have to deal with different MPI ranks executing in different architectures yielding widely different performance.

3 RELATED WORK

This section briefly describes other works that are closely related to the research performed in this manuscript. For this purpose, we set the spotlight to HPC-related solutions. These are presented bottom-up: from compiler to runtime to

[3]https://developer.nvidia.com/networking/doca

user applications. Notice that all related works were published after 2020, reflecting that this is a young research field with an increasing interest among the scientific community.

A low-overhead implementation of Remote OpenMP Offloading is publicly available in the LLVM/OpenMP compiler infrastructure [6]. The work includes detailed studies on microbenchmarks, as well as scaling results for two HPC proxy applications. The basic idea behind the Remote OpenMP Offloading implementation is to provide a transparent communication channel between the target-independent library on the host and the target-dependent library on the remote system. Besides, the solution leverages the remote procedure calls (RPC) idiom. Regarding distributed heterogeneous systems, contrariwise to Message Passing Interface (MPI) solutions, the Remote OpenMP Offloading does not need modification of sources, compilers, or languages. However, the version presented in the work is only intended for remote CPUs and GPUs, although it may be extended for other device types.

The Three-Chains framework [1] leverages the idea of remote binary code injection. Three-Chains offloads computations along with communications. It can transfer code to any network device and run it on the receiving end. It uses the BitCODE representation of the LLVM compiler and completes the compilation on the executing end. Three-Chains provides function caching during compilation, which can be translated into faster executions on generic hardware. Similarly, the Floem project [7] provides language, compiler, and runtime for offloading, but specifically to SmartNICs.

BluesMPI [8] is a technology that guarantees the full overlap of communication and computation for MPI *alltoall* collective operations, while providing on-par pure communication latency to CPU-based on-loading designs. It is designed to achieve the best pure communication latency for offloading `MPI_Ialltoall` from the host CPU to a Smart NIC or DPU. BluesMPI is evaluated with OSU microbenchmarks, improving the execution time up to 44% for `MPI_Ialltoall` operations and up to 33% in the P3DFFT application. Seamlessly offloading MPI communication to BlueField cannot be expressed for complex patterns using existing standards; new MPI APIs were also proposed to express such patterns and offload them to DPU for expressing offloading communication [9].

Regarding applications, a version of the MiniMD molecular dynamics miniapplication capable of leveraging DPUs for offloading certain stages of the computation, was recently presented [5]. Particularly, its authors created a MiniMD variation where DPUs are used to force computations among particles during neighbor-build iterations, while the host rebuilds the neighbor list. This new MiniMD version attains speedups of up to 20% compared to the original code.

Multiple novel designs to efficiently offload phases of deep learning (DL) training to DPUs were also proposed and evaluated [3]. A DL training process may consist of multiple phases: data augmentation, training, or validation of the trained model. Traditionally, these phases are executed mostly sequentially on the CPUs or GPUs due to a lack of additional computing resources to offload independent phases of DL training. In that work the authors decouple the data augmentation and validation phases from the training, in order to offload them to DPUs, claiming up to 17.5% improvement in the overall training time.

Different from previous literature, our work is the first exposing DPU network coprocessors as local accelerators for general-purpose computing, in order to ease application developer efforts. In addition, no prior work leverages the original SDK provided along BlueField devices (while it is customary to see works based on the analogous CUDA for NVIDIA GPUs); Instead, previous works leverage alternatives which add overhead (e.g., networking) and/or ad-hoc code restructurings.

4 METHODOLOGY

This section thoroughly describes the integration of offloading features for BlueField DPU into the LLVM/OpenMP target offloading construct. Unlike other approaches, this solution is fully based on the native communication provided by the BlueField SDK, DOCA, avoiding unnecessary overheads provided by other communication layers above it.

This DOCA-based solution is composed of three main modules:

- Cross-compilation: LLVM feature to generate a binary that can run in the DPU.
- OpenMP BlueField plugin: Runtime for interacting with BlueField DPUs.
- OpenMP DOCA service: Runs on BlueField DPUs to accept and complete requests received from the host, through the OpenMP plugin.

4.1 Cross-Compilation

Cross-compilation is required when the target architecture is different from the host architecture. Particularly, BlueField devices are equipped with a 64-bit ARM processor which is not necessarily the same as the host processor. For instance, assuming the host to be an x86 Intel architecture, the offloaded code needs to be cross-compiled such that it can be properly executed on BlueField devices.

Cross-compilation is a built-in feature of LLVM. It supports multiple architectures, including x86-64, aarch64, and riscv64. The LLVM project follows the global assumption that there exists only one *sysroot* in a system, in other words, there is only a root directory to locate headers and libraries. The BlueField device runs its own operating system with its corresponding file system and thus has a separate *sysroot*. In this regard, we applied platform-specific changes to LLVM/Clang to support cross-compilation for BlueField DPUs. The compiler generates a “fat” host binary, which contains a cross-compiled target binary in itself. This mechanism is native in LLVM and it provides the necessary support for GPU offloading. Consequently, we get a binary image in plugin that is compiled for BlueField DPU and is conveyed to it as a different Linux host via DOCA. The cross-compiled target binary image is utilized by the OpenMP BlueField plugin to execute the binary image of the offloaded codes on target devices.

The source code is compiled twice to Intermediate Representation (IR) by Clang—once, the IR is generated for the host, and separately the IR is generated for the target. Then the *Clang Offload Packager* is used to combine these and generate the final IR, which is then compiled and linked. Instead of using the classical way of linking based on the host linker, when linking our fat binary we rely on *Clang Linker Wrapper*. *Clang Linker Wrapper* compiles for architectures with different `sysroots` (by default, `clang` assumes only one `sysroot`). With this new method, we generate the appropriate fat binary containing the cross-compiled target regions to be offloaded.

4.2 OpenMP BlueField Plugin

The OpenMP BlueField plugin requires a common API to be implemented for a specific target (device), which is then utilized by the OpenMP runtime to manage the device. As of LLVM version 14.06, the primary functionality of the plugin requires the implementation of the methods: *info*, *loading binary*, *data transfer*, and *running target blocks*.

Our implementations of the BlueField Plugin and the OpenMP DOCA service communicate through the DOCA Communication Channel (CC) module. The module is provided by the vendor to be used for the exchange of data between the host and DPU. It provides an abstraction that is secure and efficient and communicates over PCI-e. Moreover, the module can support multiple applications together by using semaphores and locking resources between processes.

Following, the required methods are described:

- Info: Basic information is collected from the DOCA Core module and some subsequent relevant modules. This information helps identify the device and associate it with the appropriate device number. Moreover, it provides further information about device capabilities such as interface name, IB device name, PCI-e address, among others, via the DOCA Core module.
- Load Binary: The OpenMP runtime loads binaries for all the target regions in a single shared object. The runtime provides names of all the required symbols and expects addresses of each in return. The image is sent to BlueField DPU over DOCA CC. The DOCA OpenMP Service loads the image using `dlopen` and returns addresses for each symbol requested through a `dlsym` call each.
- Data Management: Data transfers are transparently maintained over DOCA CC. Pointers are exchanged plainly and the OpenMP runtime utilizes them as needed.
- Run Target Region: The OpenMP runtime requests the entry address (provided against a symbol name) to run with the arguments (as BlueField pointers). The information is again transferred to the OpenMP DOCA Service using DOCA CC, which runs the code accordingly.

4.3 OpenMP DOCA Service

Correspondingly, the OpenMP DOCA service on the device side implements the server version of the implemented methods in the OpenMP BlueField plugin.

- Load Binary: The binary image is received by the device over DOCA CC. It is saved as a temporary file and opened as a dynamic library using the `dlopen` routine. The required symbols are checked for respective addresses using the `dlsym` routine. These addresses are sent back over DOCA CC.
- Data Management: Data management is transparently managed over DOCA CC. Memory is allocated and the same pointer returned by the `malloc` routine is sent back to the host. The host later uses it to submit and/or retrieve data. The same pointer is used to release the data allocation.
- Run Target Region: The same address provided by `dlsym` is conveyed to the OpenMP runtime through the BlueField plugin to specify the region to run. Arguments are transferred as well. A low-level foreign function interface library, named `libffi`, is used to run the target region. With the help of the methods `ffi_prep_cif` and `ffi_call`, regions can be executed at run time, since arguments are not known at compile-time for the DOCA OpenMP Service. `ffi_prep_cif` prepares the structure `ffi_cif` according to the data size, types and alignment. The structure `ffi_cif` is then used by `ffi_call` to run the provided function on the DPU.

5 EVALUATION

This section presents the results of testing and evaluating the DPU offloading OpenMP extension developed in this work. For this purpose, we leverage two BlueField-capable clusters from the HPC-AI Advisory Council Network of Expertise[4]. On the one hand, Thor, a 36-node cluster where each node is equipped with a Dual Socket Intel® Xeon® 16-core CPU E5-2697A V4 at 2.60 GHz, 256 GB DDR4, and an NVIDIA BlueField-3 NDR 200 Gb/s DPU (BF3). On the other hand, Jupiter, a 32-node cluster whose nodes are composed of a Dual Socket Intel® Xeon® 10-core CPUs E5-2680 V2 at 2.80 GHz, 64 GB DDR3, an NVIDIA K40 GPU, and a BlueField-2 HDR100 100 Gb/s DPU (BF2). While BF2 consists of 8 ARMv8 A72 cores and 16 GB DDR4 RAM, BF3 counts with 16 ARMv8.2+ A78 cores and 16 GB of DDR5 memory.

[4]https://www.hpcadvisorycouncil.com

Benchmark	Description	# of transfers	Transferred
postencil	Thermodynamics	3	294 MiB
pep	Embarrassingly parallel random number generation	3517	7.06 KiB
polbm	CFD Lattice Boltzmann method	83	16.17 GiB
pcsp	Scalar Penta-Diagonal Solver	89	656.76 MiB
pomirq	Medicine	11	61.34 MiB

Table 1. Description of the microbenchmarks.

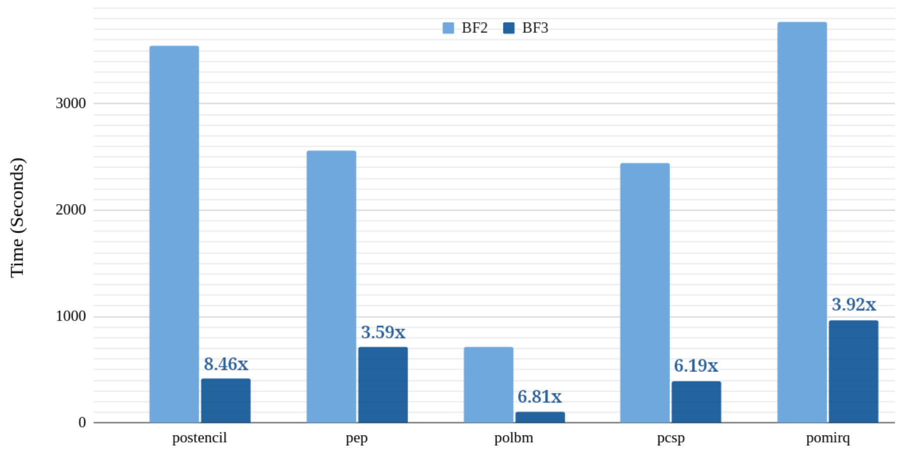

Fig. 2. Performance comparison of SPEC ACCEL benchmarks in different HPC clusters.

For the software stack, we have performed the experiments using DOCA 2.0.2, LLVM 14.0.6, and OpenMP 5.0. The executed experiments are based on the SPEC ACCEL suite [4]. SPEC ACCEL provides a set of tests to evaluate the performance of OpenMP Offloading APIs. Particularly, Table 1 summarizes the microbenchmarks used in the evaluation and provides information about their data transfers. We have only focused on the C-based codes, since the presented framework is developed for the Clang compiler. The Fortran support from the Flang compiler is expected to be released soon.

Notice that the benchmarks were initially targeted for GPU-based applications. Since GPUs are computationally efficient for massively parallel applications, these leverage the capability of `teams distribute` OpenMP constructs. DPUs, on the other hand, are composed of a few ARM cores. Therefore, removing these constructs helps the code compile to a more efficient binary for DPU. Moreover, since the ARM architecture supports SIMD instructions, the OpenMP `simd` construct is utilized to further improve the performance of DPUs. Constructs such as `parallel for` enable multicore capabilities. The use of these constructs determine processor exploitation. The benchmarks govern if the target code runs on a single core or multiple cores.

First, the DPU Offloading for OpenMP feature is evaluated over different BlueField devices, in particular, in the BF2 and BF3 machines. Figure 2 compares the execution times; It shows how BF3 outperforms BF2, as expected, in the range of 3.6x-8.5x, depending on the application.

Once it has been proved that our solution passed the tests in different DPU devices, next, it is time to evaluate how it behaves compared to a baseline. For this purpose, we have implemented an MPI version of several microbenchmarks in SPEC ACCEL, where the original offloaded kernels will be executed in the DPU within an MPI rank context. The rank in the DPU communicates with the rank in the host and executes the kernel as it is done with OpenMP offloading, following the programming model described in Section 2.2.2.

The code snippets presented in Listings 1 and 2 demonstrate the fundamental structure of the benchmarks using MPI and OpenMP, where data is sent to the device, computations are performed on the device, and the results are received on the host. A clear comparison between these approaches highlights the significantly simpler and more straightforward programming paradigm offered by OpenMP.

```
void main( int argc, char* argv[] ) {
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &dpu_rank );
    // initializations
    if ( !dpurank ) {       //dpu rank receives data
        MPI_Recv( &buffer_, BUFSZ_, MPI_UNSIGNED_LONG, 0, 0, MPI_COMM_WORLD, NULL );
    } else { // host rank sends data
        MPI_Send( &buffer_, BUFSZ_, MPI_UNSIGNED_LONG, 1, 0, MPI_COMM_WORLD );
    }
    for( int t = 1; t <= TIMESTEPS; t++ ) {
        if ( dpu_rank ) {
            // offloaded compute
        } else {
            // host compute
        }
    }
    if ( dpu_rank ) {  // dpu rank send data
        MPI_Send( buffer_, BUFSZ_, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD );
    } else {        // host rank receive data
        MPI_Recv( buffer_, BUFSZ_, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, NULL );
    }
    MPI_Finalize();
}
```

Listing 1. "MPI implementation"

```
void main( int argc, char* argv[] ) {
    // initializations
    #pragma omp target data map(tofrom:buffer_[0:BUFSZ_])
    {
        for( int t = 1; t <= TIMESTEPS; t++ ) {
            #pragma omp target
                // offloaded compute
            // host compute
            }
        }
    }
}
```

Listing 2. "OpenMP implementation"

Figure 3 compares the execution time of the MPI and OpenMP microbenchmark versions, where minor differences in performance can be appreciated. Since the underlying hardware is the same, a priori, there can be two explanations for these minuscule discrepancies: compiler and data transfer management.

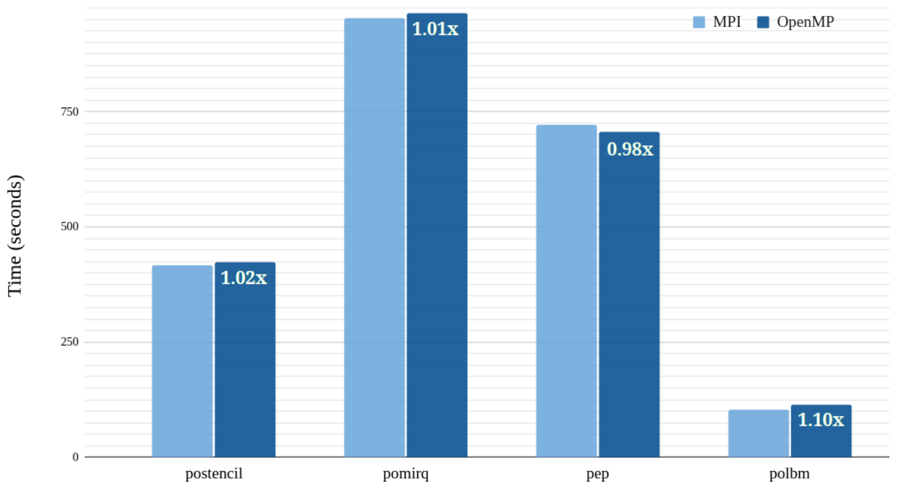

Fig. 3. Comparison of OpenMP and corresponding MPI ported benchmark performance.

Benchmark	Communication	Computation	Ratio
postencil	0.145 s	416.39 s	0.035%
pomirq	0.035 s	960.04 s	0.004%
pep	0.067 s	711.65 s	0.009%
polbm	7.307 s	102.43 s	7.134%

Table 2. Breakdown times in Thor Cluster.

On the one hand, the OpenMP compiler is able to pack into the binary the input parameters for configuring the execution. These parameters can be sent using only one transfer if required. Instead, these parameters need to be transferred separately to the ranks on the MPI version. For this reason, *pep* OpenMP performs slightly better.

On the other hand, the different communication mechanisms in MPI and OpenMP can cause different transfer behaviors. In this regard, Table 2 contains the communication/computation time and ratio for the microbenchmarks.

Apart from *pep*, the MPI implementations of the microbenchmarks provide slightly better times than the OpenMP counterparts. Although all applications are compute-bound, communications in *polbm* represents 7% of the execution time, which explains the largest time difference between versions. For this purpose, following, a thorough analysis of the host–device communication is presented, studying the bandwidth and latency in order to understand which role the networking management of the runtime plays.

The benchmarks for profiling bandwidth and latency are based on OSU Micro Benchmarks (OMB)[5]. OMB is a suite of benchmarks that evaluates MPI performance. We use point–to–point bandwidth and latency tests for measuring MPI performance. To provide a fair comparison, we have adapted OMB for CUDA and DOCA. Particularly, for CUDA, messages are sent with `cudaMemcpyAsync`, while for DOCA, we use `doca_comm_channel_ep_sendto` (and `doca_comm_channel_ep_recvfrom` on the other end). OMB considers large messages those greater than 8KiB, and embracing OMB philosophy, the benchmarks are structured as follows:

- Warming up:

[5]http://mvapich.cse.ohio-state.edu/benchmarks

  - Bandwidth: 10 iterations for small sizes and 2 for large sizes.
  - Latency: 1000 iterations for small sizes and 100 for large sizes.
- Evaluation:
  - Bandwidth: 90 iterations for small sizes and 8 for large sizes.
  - Latency: 9000 iterations for small sizes and 900 for large sizes.
- Synchronization:
  - A message is sent back in the opposite direction; and
  - A barrier is explicitly instantiated if possible `MPI_Barrier` for MPI, `taskwait` construct for OpenMP, `cudaStreamSynchronize` for CUDA, and event polling for DOCA (on file descriptors received from `doca_comm_channel_ep_get_event_channel`).

Notice that CUDA experiments have been performed in the CTE-Power cluster at BSC. This cluster provides IBM Power9 processors and NVIDIA V100 GPUs with CUDA version 10.1.

Figures 4 and 5 showcase bandwidth and latency, respectively, for different message sizes under the different execution environments. OpenMP DPU demonstrates superior performance compared to OpenMP GPU in terms of both bandwidth and latency up to a size of 64 KiB. However, once this threshold is reached, certain unavoidable synchronicity primitives are enabled in ODOS, causing OpenMP DPU to exhibit relatively lower performance compared to OpenMP GPU. Communications of OpenMP DPU are based on DOCA, while OpenMP GPU are on CUDA. In this regard, ideally, we would expect both OpenMP implementations to expose a similar behavior with respect to their underlying low-level programming models, and of course, never outperform them. Thus, for small sizes, OpenMP implementations clearly offer lower bandwidth than their low-level counterparts. In mid-size messages, there are a series of ups and downs, until convergence in larger sizes. We found that for large sizes, the reported bandwidth decreases because of the processing time needed for pinning memory in these sizes.

In DOCA, mid-size messages (2–64 KiB) slow down because the maximum message size than can be sent over a single DOCA CC call is 2 KiB. Thus, messages need to be fragmented by the processor for larger sizes.

The processing time of large messages appears in CUDA and OpenMP GPU runs as the bandwidth decreases after a payload of 4 MiB. For DOCA-based executions, a synchronization message is required in the opposite direction to keep synchronicity among sizes beyond 64 KiB. Therefore, the bandwidth begins to decrease after that point.

Regarding latency, the pairs OpenMP-SDK show a more stable behavior until their convergence. With this experiment, we aim to demonstrate that OpenMP DPU provides the expected performance behavior.

6 CONCLUSIONS AND FUTURE WORK

This paper presents the first OpenMP offloading solution for DPU in–network coprocessors in the literature, based on LLVM and the BlueField DOCA SDK. While the existing solutions to address DPU network coprocessors involved either (1) low-level device programming or (2) unnatural MPI usage deploying processes executing in widely different architectures, we provide to the community a much simpler and natural approach. The recent emergence of BlueField 3 devices, much more powerful than their predecessor, along with our programming proposal, has the potential to open the door to a much wider variety and quantity of research and production results.

HPC applications could benefit from DPU devices offloading part of their load to them. For example, when training deep neural networks, data augmentation or validation stages, could be offloaded to to less power accelerators such as DPUs [3]. In turn, in large distributed multiphysics simulations could offload the halo exchange operation making DPUs

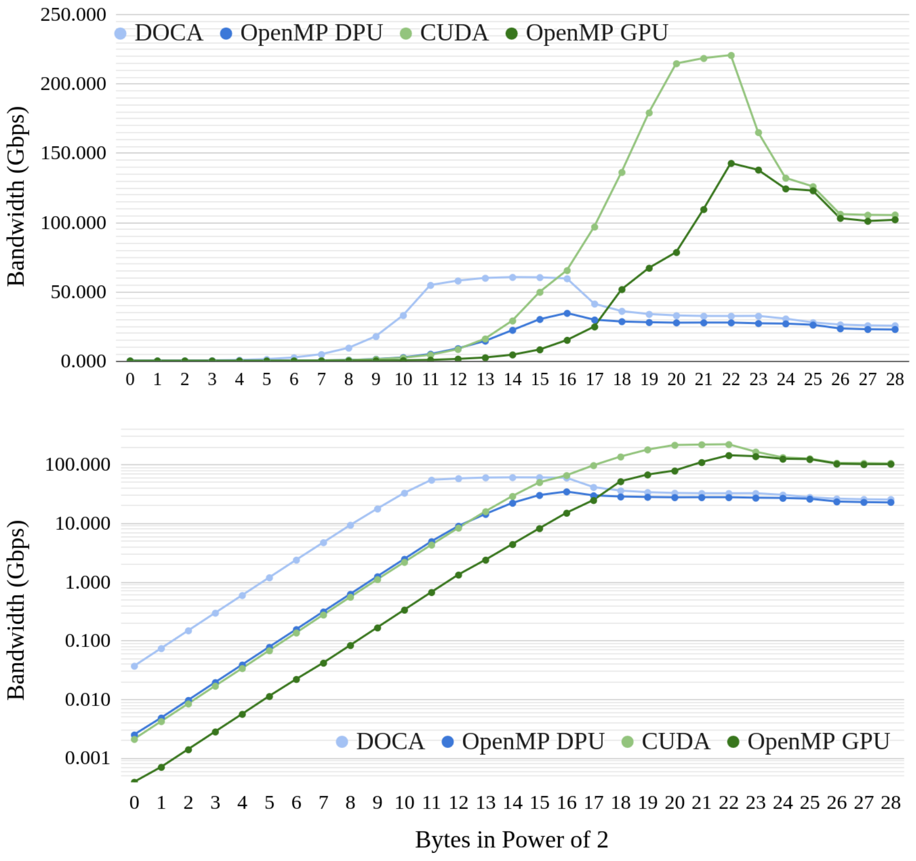

Fig. 4. Bandwidth comparison of transfers using DOCA, OpenMP DPU, CUDA, and OpenMP GPU implementations (bottom chart represents the results in logarithmic scale).

responsible for communicating and computing the halo among neighbors. For this purpose, we are already working on MPI integration, i.e., providing support for MPI calls within offloaded tasks.

ODOS software can be downloaded from: https://www.bsc.es/discover-bsc/organisation/scientific-structure/accelerators-and-communications-hpc/team-software

ACKNOWLEDGMENTS

The authors would like to express their sincere gratitude to Gilad Shainer, Richard Graham, Gil Bloch, and Yong Qin for their support, insightful comments, and suggestions. The research producing this paper received funding from NVIDIA and the Ministerio de Ciencia e Innovación—Agencia Estatal de Investigación (PCI2021-121958). The authors are grateful for the support from the Department of Research and Universities of the Government of Catalonia to the AccMem (Code: 2021 SGR 00807). Antonio J. Peña was partially supported by the Ramón y Cajal fellowship RYC2020-030054-I funded by MCIN/AEI/10.13039/501100011033 and, by “ESF Investing in your future”

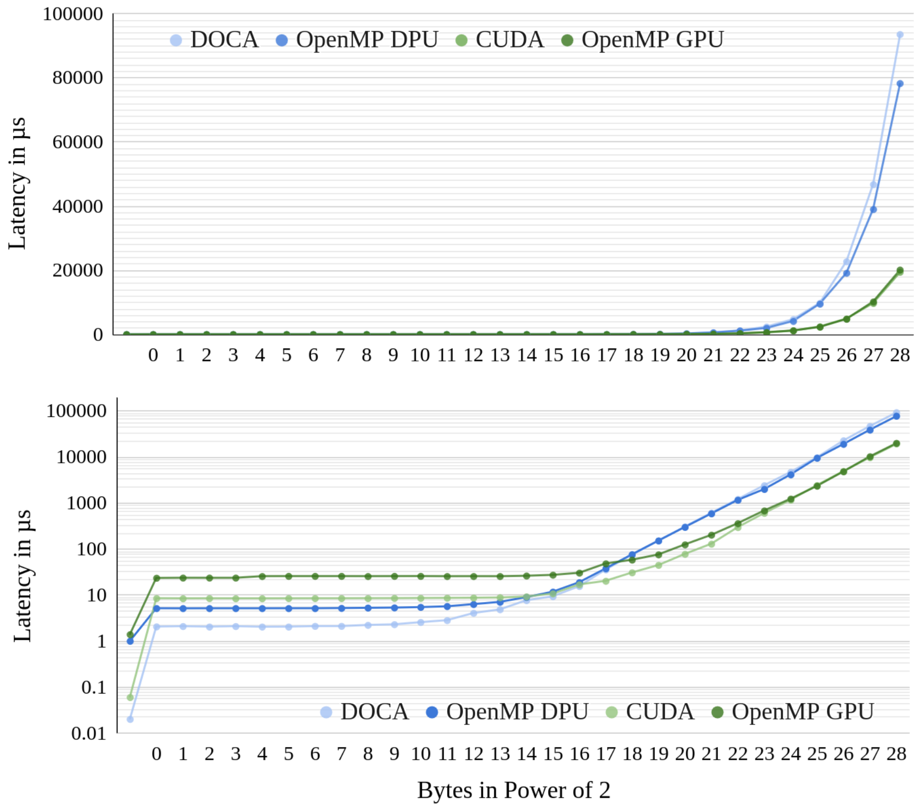

Fig. 5. Latency comparison of transfers using DOCA, OpenMP DPU, CUDA, and OpenMP GPU implementations (bottom chart represents the results in logarithmic scale). Notice that x-axis scale represents power of 2 bytes, so the first tick, left to zero and without a label, is 0 bytes, the second tick corresponds to 2^0 bytes, etc.

REFERENCES

[1] M. Bayatpour, N. Sarkauskas, H. Subramoni, J. M. Hashmi, and D. K. Panda. 2021. BluesMPI: Efficient MPI non-blocking alltoall offloading designs on modern BlueField smart NICs. In *ISC High Performance*. 18–37. https://doi.org/10.1007/978-3-030-78713-4_2

[2] Idan Burstein. 2021. NVIDIA data center processing unit (DPU) architecture. In *IEEE Hot Chips 33 Symposium (HCS)*.

[3] Arpan Jain, Nawras Alnaasan, Aamir Shafi, Hari Subramoni, and Dhabaleswar K. Panda. 2022. Optimizing distributed DNN training using CPUs and BlueField-2 DPUs. *IEEE Micro* 42 (2022), 53–60. Issue 2. https://doi.org/10.1109/MM.2021.3139027

[4] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, et al. 2014. SPEC ACCEL: A standard application suite for measuring hardware accelerator performance. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. 46–67.

[5] S. Karamati, C. Hughes, K. S. Hemmert, R. E. Grant, W. W. Schonbein, S. Levy, T. M. Conte, J. Young, and R. W. Vuduc. 2022. 'Smarter' NICs for faster molecular dynamics: A case study. *IEEE 36th International Parallel and Distributed Processing Symposium* (2022), 583–594.

[6] Atmn Patel and Johannes Doerfert. 2022. Remote OpenMP offloading. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 441–442. https://doi.org/10.1145/3503221.3508416

[7] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A programming System for NIC-Accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 663–679.

[8] N. Sarkauskas, M. Bayatpour, T. Tran, B. Ramesh, H. Subramoni, and D. K. Panda. 2021. Large-message nonblocking MPI_Iallgather and MPI_Ibcast offload via BlueField-2 DPU. In *IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 388–393.

[9] K. Suresh, B. Michalowicz, N. Ramesh, J. Contini, S. Yao, A. Xu, H. Shafi, Subramoni, and D. K. Panda. 2023. A novel framework for efficient offloading of communication operations to Bluefield SmartNICs. In *37th IEEE Int. Parallel and Distributed Processing Symposium* (St. Petersburg, Florida, USA).