Using BlueField-3 SmartNICs to Offload Vector Operations in Krylov Subspace Methods

Kaushik Kandadi Suresh, Benjamin Michalowicz, Nick Contini,
Bharath Ramesh, Mustafa Abduljabbar, Aamir Shafi, Hari Subramoni, Dhabaleswar Panda
Department of Computer Science and Engineering
The Ohio State University Columbus, USA
{kandadisuresh.1, michalowicz.2, contini.26, ramesh.113, abduljabbar.1, shafi.16, subramoni.1, panda.2}@osu.edu

Abstract—**Modern SmartNICs are capable of performing both computation and communication operations. In this context, past works on accelerating HPC/DL applications have manually selected some computational phases for offloading them to the SmartNICs. In this work, we identify Vector Multiply-Adds (VMA), Distributed Dot Products (DDOT), and Sparse Matrix-Vector Multiplication (Matvec) as three fundamental operations that are widely used in Krylov Subspace methods. We propose a generic scheme to automatically offload a selected set of the above operations to NVIDIA's latest BlueField-3 SmartNICs. Our proposed method works for any variant of the PCG solver algorithms. We also propose an optimization to reduce data transfer cost for offloading Matvec operation. Our proposed schemes demonstrate up to 1) 24% improvement in PCG, and Pipelined PCG algorithms on 256 processes on Intel Broadwell CPUs and 11% improvement on a system with Intel Sapphire Rapids CPU with BF3 SmartNIC using the PETSc and HYPRE solver libraries. To the best of our knowledge, this is the first work to propose a framework to efficiently offload VMA, DDOT, and Matvec operations to the DPU and show improvements on a modern CPU-based system.**

Index Terms—**HPC, Infiniband, MPI, SmartNIC, DPU, Offload, Reduction**

I. Introduction

Modern CPU-based supercomputers are equipped with multi/many-core CPUs, accelerators, and high-performance RDMA-based Network Interface Cards (NICs) such as NVIDIA's ConnectX-7 [18]. SmartNICs (enhanced versions of NICs) provide certain specialized hardware units that perform network functions offloading, encryption, compression, etc. Apart from providing these specialized units, SmartNICs such as the BlueField Data Processing Units (DPU)[16] also provide general-purpose processors and DRAM units that are capable of issuing commands to the NICs. The release of NVIDIA's BlueField-3 (BF3) DPU also shows an increase in compute power (caches, clock speed, bandwidth, etc.) over its predecessor.

Past research on offloading communication has shown that SmartNICs can be leveraged to accelerate CPU-based HPC applications. This has led to the deployment of SmartNICs in CPU-based clusters [15]. Given the presence of SmartNICs in HPC clusters, people started exploring the usage of the general-purpose cores of SmartNICs to improve the performance of HPC/DL applications by offloading computation operations. Though GPUs are better candidates for this, the existing CPU-based clusters with SmartNICs can also benefit from the programmable cores which are otherwise unused for a compute-intensive application. Since a SmartNIC's performance is not superior to that of the CPU (in terms of cache, CPU core count, Memory Bandwidth, etc), it is not beneficial to offload the entire application or just any portion of the application. The challenge here is to identify independent tasks to offload to SmartNICs. Additionally, this introduces communication latency arising from host and DPU data transfer. Past works such as [8], [11] accelerate CPU-based HPC and DL workloads by offloading certain computational phases to the SmartNIC. They achieve this by manually identifying the suitable tasks that can be offloaded. In this work, we choose Krylov Solvers as a case study to select and offload fundamental operations to the SmartNIC. Each variant of a solver algorithm may have different data-flow dependencies as explained in Section III. As a result, the manual selection of operations to offload is not very efficient. Therefore, we propose a generalized scheme to select and offload tasks to the SmartNIC. Furthermore, we also propose a scheme to minimize the data transfer cost.

Specifically, we identify Distributed Dot-Products (DDOT), Vector-Multiply-Add (VMA), and Sparse Matrix-Vector operations as commonly used computation operations across different Krylov Solvers. Then, we provide a framework to efficiently offload these operations to the Bluefield SmartNIC. Our offload framework uses the DPU cores to perform the computational phase of DDOT, VMA, and Matrix-Vector operations. Furthermore, our framework uses the NIC of DPU to perform the communication phase (MPI_Allreduce) of the DDOT operation. Using this framework, we modify some of the commonly used Krylov solvers such as PCG, and PIPECG in numerical libraries such as PETSc [2], [1] and HYPRE[6]. These accelerated solvers in turn can be used to accelerate any application using PETSC/HYPRE.

To the best of our knowledge, this is the first work to propose 1) a generalized splitting scheme for efficiently offloading different variants of PCG algorithms, and 2) an Onloading Scheme to optimize the Matvec operations by

*This research is supported in part by NSF grants #1818253, #1854828, #2018627, #2311830, #2312927, and XRAC grant #NCR-130002.

reducing the data transfer cost.

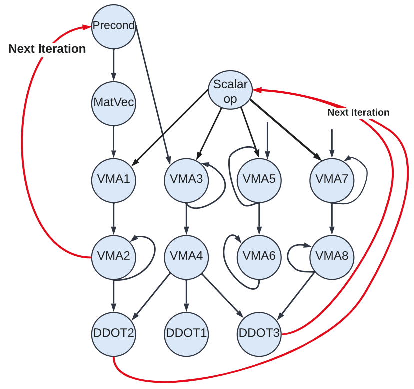


Fig. 1. Data Flow graph for Non-Blocking variants of the Pipelined PCG Algorithm. Not all operations can be offloaded to the DPU. Moreover, different variants have different sets of dependencies. This is to illustrate the challenge of finding the best set of operations that can be offloaded to the DPU to improve runtime performance.

II. Testbed Information

All experiments performed in the subsequent sections use one of the following testbeds:

Testbed 1: It has 32 nodes. Each of the 32 nodes has a Dual-socket Intel Xeon 16-core CPU (E5-2697A V4 @ 2.60 GHz), NVIDIA ConnectX-6 HDR100 100Gb/s InfiniBand Adapters, and 256GB DDR4 RDIMMs running at 2400 MHz. These nodes are also equipped with BF3 SmartNICs. The BF3 is connected via a single port of 200 Gb/s NDR InfiniBand.

Testbed 2: It is a single node testbed with a Dual-socket 48-core Intel(R) Xeon(R) Platinum 8468 (Sapphire Rapids) CPU, NVIDIA ConnectX-6 HDR100 100Gb/s InfiniBand Adapters, 256GB DDR5 RDIMMs and BF3 SmartNIC (with NDR).

All our experiments have 16 DPU processes per BF3 since BF3 DPU has 16 cores. The number of host processes per node (PPN) is 32 for Testbed 1 and 96 for Testbed 2 since they have 32 and 96 cores respectively unless stated otherwise.

III. Motivation

VMA, DDOT, Matvec, and Preconditioner are commonly used operations in PCG algorithms. A preconditioner, depending on the complexity, may also use VMA, DDOT, and Matvec operations. Past research on characterizing [14] BF3 DPUs has shown that BF3 gives up to 82% improvement in bandwidth on the STREAM benchmark over the BF-2 due to an increase in core frequency, cache size, and DRAM bandwidth. This motivates us to evaluate the potential of offloading computation operations used in PCG algorithms to the BF3 DPU.

Let $Spcg$ denote the set of operations in a given PCG algorithm. Let Sh and Sd denote two subsets of $Spcg$ such that $Sh \cup Sd = Spcg$ and $Sh \cap Sd = \emptyset$. To understand the maximum improvement one can obtain by offloading computation to BF3, consider the following equation: $T_{pcg} = T_h + T_d$, where T_{pcg}, T_h, and T_d are the time taken to execute the operations in $Spcg$, Sh, and Sd on the CPU respectively. If the operations in Sd are executed in the BF3 DPU while the operations Sh are executed on the host, then the total time for execution is $max(T_h, T_{dpu} + T_{comm})$, where T_{dpu} is the time to execute operations in Sd on the BF3 DPU and T_{comm} is the time to exchange data needed for Sd to be executed on the DPU. If T_{dpu} is $k * T_d$ where k is the slowdown in the performance of the operations in Sd on the DPU compared to the host, and $T_d = p * T_{pcg}$, where p is the time percentage of time spent in executing the operations in Sd compared to the total time, $T_{comm} = c * T_{pcg}$ where c is the percentage of time spent in moving data to the DPU from the Host, T_{off} is $max((1-p), k*p+c)*T_h$, then the %-improvement is given by the equation 1. In this equation, everything is represented as a percentage of T_pcg. The symbols are explained in Table I.

$$\%(improvement) = (1 - max((1-p), k*p+c)) * 100 \tag{1}$$

TABLE I
Symbol Table for Equation 1

Symbols	Description
T_{pcg}	CPU execution time for operations in $Spcg$
T_h	CPU execution time for operations in Sh
T_d	CPU execution time for operations in Sd
T_{dpu}	DPU execution time for operations in Sd
T_{comm}	CPU to DPU data transfer time
k	Slowdown when operations in Sd are executed in the DPU
p	percentage of time taken to execute operations in Sd
c	percentage of time taken to move data to the DPU
T_{off}	Total execution time

TABLE II
Estimating the maximum achievable % improvement by offloading some operations from PCG algorithms to BF3 DPU from Sapphire Rapids CPU on Testbed 2

Problem Sizes	32X32X32	64X32X32	64X64X32
Observed Slowdown	4.33X	3.28X	3.21X
Max %Improvement for 15% communication	16	19	20
Max %Improvement for 30% communication	13	16.3	17
Max %Improvement for 60% communication	7.5	9.3	9.5

Table II shows the relative performance of the AMG-PCG benchmark on Testbed 2 with 16 Host and DPU processes. Observed Slowdown in Table II shows the ratio of Solve time for the AMG-PCG code executed on Sapphire Rapids CPU and BF3 DPU for representative problem sizes. Plugging in these slowdown values for k in equation 1 gives the maximum % improvement that we can obtain if the percentage of communication time (which is c) ranges from 15 to 60. We observe it is possible to get up to 20% improvement depending on the time spent in communication. **This motivates us to use BF3 SmartNICs for optimizing the Krylov solvers.**

IV. Challenges

To improve the performance of PCG solvers by offloading certain operations to the DPU, we need to increase the improvement represented by the equation 1. The % improvement is inversely proportional to the slowdown factor k, % communication time (c). The % improvement is also directly proportional to p till a certain point after which it starts decreasing. In this work, given k, we 1) identify the operations that constitute Sd and 2) Propose solutions to reduce %communication time denoted by c.

Identifying operations to offload: Based on empirical evaluations we characterize the operations into two sets: dominant and non-dominant. Non-dominant operations such as VMA and DDOT are computationally less intensive compared to Matvec and Preconditioner. To efficiently offload these operations, we explore two methods to offload them to the DPU: i) Multi-Op, and ii) intra-Op offload methods. The Multi-Op offload method allows offloading a set of non-dominant operations to the DPU with the rationale of finding an effective set that minimizes the communication latency between the host and the DPU. On the other hand, the Intra-Op offload method enables a portion of a single dominant operation to be offloaded to the DPU. This is useful when the entire operation cannot be offloaded. Scalable variants of PCG [5], GMRES have been explored by researchers (see Section IX). Some of the examples are Non-blocking PCG, Pipelined PCG, and Single Allreduce PCG. Since there are many variants of PCG/GMRES [5], each of them may have a different dataflow pattern. For example, Figure 1 shows the data flow graph for the Pipelined PCG algorithm. To optimize this algorithm we cannot offload any/all operations to the DPU. Therefore, we need to find the right set of operations to offload which brings us to our next challenge: **Can we come up with a common grouping scheme to offload different operations to the DPU for all the variants of these solvers such that they minimize the data transfer latency and also exhibit good overlap with the host code?**

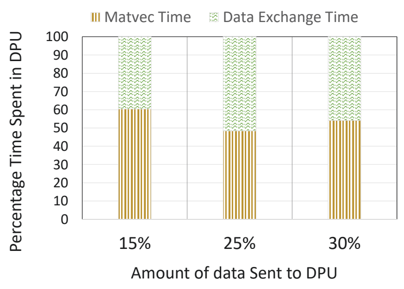


Fig. 2. Percentage of time spent in matrix-vector product computation and data transfer on the BF3 DPU for different amounts of data sent to the DPU from the host

Reduce the data transfer time: To offload these operations to SmartNICs, data has to be moved to them and then moved back to the host memory after the offload. We quantify the amount of time spent in moving data between SmartNIC and the host. Figure 2 shows the percentage of time spent in the DPU reading input data and performing MatVec operations on the data. The X-axis represents the data required to move for different percentages of the rows of a given matrix. We used HYPRE's AMG code to generate this matrix of size 32Kx32K elements for Krylov solvers. This is on 1 Node, 32 processes on the host moving data to 16 DPU processes on Testbed 1. We observe that the data transfer cost could range from 35% to 50% depending on the input size. Similar trends are observed for large matrixes. This becomes worse with the increase in process count. Data transfer required to perform DDOT and VMA operations is another bottleneck; current offload schemes purely for communication may not be optimal as shown by Figure 5. This brings us to the second challenge: **How can we efficiently offload VMA, DDOT, and Matvec operations by identifying the operations to offload and reducing** the communication latency?

V. Contributions

The above observations lead to the following broad challenge: **Can we design a framework to optimize Krylov solvers by efficiently offloading VMA, DDOT, and Matvec operations to the DPU?** In this paper, we take up this challenge and propose a framework to offload three commonly used operations such as VMA, DDOT, and MatVec to SmartNICs. Specifically, we propose a splitting scheme that gives us two sets of operations Sh and Sd for a given PCG algorithm. Sh and Sd are generated such that offloading Sd to the DPU ensures maximum overlap when Sh is executed on the host. We provide APIs to offload the operations in Sd to the DPU. Then, to reduce the data transfer cost between the host and the DPU, we propose the onloading scheme. In this scheme, a set of leader processes offloads a portion of their operation to the DPU, and the non-leader processes onload a portion of their operation to the leader processes. This way we reduce the number of processes doing the host to DPU data transfer. With our proposed optimization schemes we accelerate the solver algorithms such as PCG and Pipelined PCG. Our key contributions are as follows:

- Propose a generalized **splitting scheme** that can be applied to offload any PCG algorithm.
- Optimize Pipelined PCG with the proposed splitting scheme that gives up to 24% improvement compared to the host-based implementation
- Reduce the data transfer overhead by designing and implementing the proposed **onloading scheme**. We observe up to 21% improvement with the Matvec optimization.
- Our proposed schemes show up to 11% improvement on a system with modern Sapphire Rapids CPU and Bluefield SmartNIC.

VI. Background

A. Distributed matrix-vector multiplication

$$Y = A * X + B \tag{2}$$

$$Y = A_1 * X_{local} + A_2 * X_{remote} + B \tag{3}$$

$$Y = A_1 * X_{local} + B \tag{4}$$

$$Y = Y + A_2 * X_{remote} \tag{5}$$

Numerical solver libraries like PETSc and HYPRE provide a distributed matvec product method that performs the product described by Equation 2, where A is a matrix, and X, Y, and B are vectors. These are typically distributed such that each process owns a portion of the matrix and the vectors. The distributed Sparse-Matrix vector multiplication involves multiple phases: In the first phase, those elements of each row are multiplied by the vector for which the vector data resides in the local process memory. While this happens, additional vector data needed for the product is obtained from the peer processes using a communication library such as an MPI library. We call this compute phase 1. After the communication phase, all the rows are multiplied with the newly exchanged vector data obtained from the remote processes. Then, the resulting vector of the second compute phase is added to the vector obtained from the first compute phase to obtain the final result. This process is described by Equation 3, where X_{local} is the local vector and X_{remote} is the remote vector. Equation 4 describes computation at phase 1, and Equation 5 describes the final computation. We used the default parallel Compressed Sparse Row (CSR) format of the matrix in PETSc and HYPRE.

VII. Framework to perform Multi-Op and Intra-Op offload

In this section, we give an overview of our framework which consists of 1) the splitting scheme, and 2) Multi-Op and Intra-Op offload interface and implementation. Figure 3 shows the

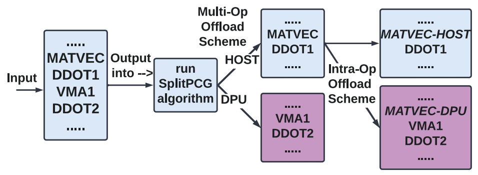


Fig. 3. Steps to our framework to offload VMA, DDOT, and Matvec operations. The splitting scheme outputs the set of non-dominant operations. The Multi-Op offload APIs are used to offload the same. Intra-Op offload API is used to offload the Matvec operation

steps in offloading VMA, DDOT, and Matvec operations in our proposed framework. First, the splitting scheme is executed for a given PCG algorithm. This gives the set of operations of non-dominant operations. Then, the PCG algorithm is modified to use the Multi-Op interface proposed in section VII-B, to offload the non-dominant operations to the DPU. Thirdly, the Intra-Op offload method and optimization, described in section VII-C, allows the users to efficiently offload the Matvec operation to the DPU. In our offload framework runtime, we launch a set of processes on the DPU called worker/proxy processes. Each host MPI rank is mapped to a proxy/DPU process. A worker DPU process can be mapped to multiple host processes.

A. Designing the Offloading Scheme

In this subsection, we outline our splitting scheme that outputs the non-dominant operations to be offloaded to the DPU. The scheme expects an input file in which each line has the following: 1) an operation type, 2) an input vector list, and 3) an output vector list. The splitting scheme outputs the dominant and non-dominant sets. Users can then use the Multi-Op APIs to select and offload the non-dominant set. This addresses the challenge of automatically identifying the set of operations to be offloaded to the DPU for any PCG algorithm.

1) Understanding the solver algorithm graphs: All variants of PCG algorithms have Matvec, Preconditioner (PC), and a set of DDOT and VMA operations in some order. Figure 4 shows a generalized graph for these solvers. Note that, in addition to the above operations, some solvers may have more operations that are not shown in the graph. To understand which operations are suitable for multi/intra-op offload methods, we measure the relative time spent in different operations for PCG. Our profiling showed that Matvec and PC operations are dominant among the three operations profiled for PCG. We also observe similar trends for other algorithms as well. Since SmartNIC cores are not as efficient as CPU cores, offloading domant operations to SmartNIC will degrade the performance because the T_{dpu} will exceed T_{host}. Conversely, it is beneficial to group a set of VMA and DDOT operations and overlap them with the dominant portion, which includes Matvec and Preconditioner operations. In some cases, it may be better not to offload anything at all. Consider Figure 1 to understand the reason for this. If all VMA and DDOT operations are offloaded, then the total time will become $T_{host} + T_{dpu} + T_{comm}$ because there is no operation to overlap with PC and Matvec operations on the host. This brings us to the following grouping scheme to selectively offload VMA and DDOT operations.

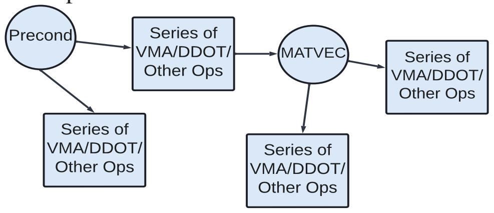

Precond
Series of VMA/DDOT/ Other Ops
MATVEC
Series of VMA/DDOT/ Other Ops
Series of VMA/DDOT/ Other Ops
Series of VMA/DDOT/ Other Ops

Fig. 4. General Code flow of PCG/GMRES algorithms. In these types of algorithms, the preconditioner and Matvec together could generally dominate the total runtime. Therefore, it is beneficial to offload those VMA/DDOT operations that are not part of the dominant loop as explained in Section VI

2) Proposed splitting scheme: Based on observations in the previous sections, we design a scheme to split the computational operations in any PCG algorithm into two sets: dominant and non-dominant sets. The dominant set consists of computationally intensive operations such as PC, and Matrix-Vector multiplication (Matvec). All operations in the dominant set form a cycle because the output of the last operation is used as the input to the first operation in the next iteration. The non-dominant set consists of all other operations that are not a part of the dominant set. The dominant and the non-dominant sets can be executed in parallel by different compute engines. We next describe the splitting scheme.

Algorithm 1 is used for splitting a set of input operations into dominant and non-dominant sets. This algorithm takes in a list of operations from any PCG algorithm. Each operation is defined by the type (PCG, Matvec, VMA, DDOT, SCALAR), input, and output vectors/scalars. Given this list of operations, first, we construct a directed graph in which each node is an operation. If the output vector of one node is used as the input of another node, there is a direct edge from the first node to the second node. After constructing the graph, we find PC and Matvec nodes from the graph in lines two and three. Then, we find simple paths starting from the PC node to the Matvec node and another set of paths starting from the Matvec node to the PC node. This will give us all the paths containing PC, and Matvec nodes. If there are multiple paths from PC to Matvec, then we select a computationally dominant path. A similar filtering is done to choose one path from the Matvec node to the PC node. The nodes from the selected PC-to-Matvec and Matvec-to-PC paths are added to the dominant set. All other nodes from the graph are added to the non-dominant set.

We use the following logic to find the dominant path: first, we assign work units to DDOT and VMA operations to compare them. Specifically, we assign one work unit to a VMA operation and K work units to a DDOT operation, where K is the ratio of the arithmetic intensity of DDOT and VMA operations. Then, for each path, we sum up the work unit for each node to determine the total path sum and choose the path with the maximum sum. For a more accurate estimate, we can also determine K empirically by finding the ratio of execution times of DDOT and VMA operations for different sizes.

After constructing dominant and non-dominant sets, we print the sequence of operations when each set is respectively executed in the host and DPU. Based on this sequence, our Multi-Op interface can be used to offload the non-dominant set to the DPU. First, we perform a topological sort of the origin $opGraph$. Then, a new host and DPU list of nodes are generated based on the sorted order as shown in line 8 of Algorithm 1. This gives an ordering of operations to be executed on the host and DPU to ensure correctness. The 'next-iteration' edges are not considered for the sorting process since they form a cycle. After this, we add 'send-toDPU(vector)', and 'waitfor(vector)' operations to the host and DPU list to perform data transfer and wait for the inputs. This is needed when a DPU operation requires a vector from the Host. Our splitting scheme ensures that there are only host-to-DPU transfers of vectors so that the host process does not wait on any vector from the DPU, but only scalars.

The above scheme applies to the main loop of the iterative PCG algorithms. The initialization phase before the start of the iteration remains the same.

B. Multi-Op offload

1) Interface: In this section, we expose our offloading schemes as a library containing methods to offload VMA, DDOT, and Matvec operations. In our framework/library, we launch a set of processes on the DPU called worker/proxy processes. Each host MPI rank is mapped to a proxy process by our library. Our library uses the MPI library for performing communication operations.

Algorithm 1: Algorithm to split PCG algorithms to host and DPU components

```
1 Function splitPCGAlgo(opList):
2     opGraph ← buildOpGraph(opList)
3     pcOp ← findPCop(opList)
4     mvOp ← findMatvecop(opList)
5     pcmvPaths ← findAllSimplePaths(pcOp, mvOp)
6     hostSplit.add(findDominantPaths(pcmvPaths))
7     dpuSplit.add(findOtherNodes(opGraph, hostSplit))
8     (sHostSplt, sDpuSplit) ← topoSort(opGraph,
        hostSplit, dpuSplit)
9     (sHostSplt, sDpuSplit) ←
        addDataExchOps(opGraph, sHostSplit, sDpuSplit)
```

```
VMA_Offload(void *VX, double alpha, void *VA, void *
    VB, size_t size, Request *req);
DDOT_Offload(void *VX, void *VY, void *VR, size_t
    size, MPI_Comm comm, Request *req);
send_to_DPU(void *Vec, size_t size, Request *req);
wait_for(void *Vec, size_t size, Request *req);
Multi_Offload_begin(Request *req);
Multi_Offload_end(Request *req);
Multi_Offload_call(Request *req);
Multi_Offload_wait(Request *req);
```

Listing 1. Offload API

Listing 1 describes the API of the proposed multi-op offloading framework. To offload a set of VMA/DDOT operations, one must first call `Multi_offload_begin` methods which mark the beginning of an epoch to record the set of subsequent offload calls. After calling `Multi_offload_begin`, any number of `VMA_Offload`, `DDOT_Offload` methods can be called with the request object provided by the `Multi_offload_begin` method. Then, to end the epoch `Multi_offload_end` is called. Then `Multi_Offload_call`, `Multi_Offload_wait` are used to initiate and complete the offload of the set of VMA/DDOT operations recorded by the request object. This way one can offload any sequence of VMA/DDOT operations.

2) Offloading PIPECG Using the splitting scheme: In this section, we show the working of the splitting scheme by using the PIPECG algorithm (see Figure 1) as an example. The path from PC to Matvec does not have a node. The dominant path from Matvec to PC contains VMA1 and VMA2 operations. These nodes form the dominant set. Other nodes form the non-dominant set. The order of operations listed by the splitting scheme is shown in the listing 2. Based on the output in listing 2, we can modify the PCG algorithm using the APIs proposed in section VII-B, to execute them in the host and DPU respectively. Note that the listing shows how to construct a single iteration of the PIPECG algorithm. The X vector, which contains the final solution is sent to the Host after the last iteration. For space reasons, we have shown the application of our splitting scheme only for Pipelined variant of PCG. This scheme can be applied to split any other variants of the PCG algorithm such as non-blocking PCG [4], etc.

```

HOST: PC send(m) Matvec waitfor(scalar) VMA1 VMA2
    send(w);
```

```
DPU: SCALAR VMA7 VMA8 waitfor(m) VMA3 VMA4 DDOT3
    VMA5 VMA6 waitfor(w) DDOT2;
```

Listing 2. Output of Splitting Scheme for PIPECG

C. *Intra-Op offload*

In this section, we explore the Intra-Op offloading strategy. In the previous section of Multi-Op offload, the entire operation was offloaded to the DPU. Such a scheme would be useful when there is enough independent computation in the algorithm to be offloaded. In the intra-Op offload scheme, a part of the operation is offloaded to the DPU. Therefore, the offloaded part is overlapped with the non-offloaded part of the operation.

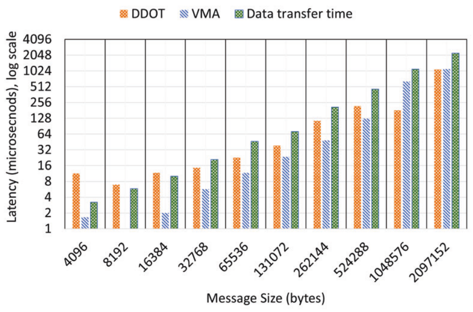


Fig. 5. Comparison of data transfer time against DDOT and VMA executions in a pure-host setting. Even without moving operations to the DPU, data transfer becomes the dominant factor.

1) Can we offload VMA and DDOT?: First, we explore the possibility of offloading DDOT and VMA operations. To understand the potential performance of offloading, we look at the transfer time and average performance time for DDOT and VMA operations. We ran the PCG problem from LLNL's AMG [26] benchmark with a problem size of 32x32x32 on 2 Nodes with 32 PPN on Testbed 1. We measured the average time to perform DDOT and VMA operations for different vector sizes. To understand the data transfer latency we ran OMB's multi-latency benchmark [19] with 16 processes on Host and 16 processes on the DPU for the same set of vector sizes. We clearly observe that the data transfer time is at least twice as high as the time to perform the operations as shown in Figure 5. The data transfer time shown here is a conservative estimate since in the actual setting there would be 32 processes on the host sending data to 16 DPU processes for our cluster setup. The data should also be brought back to the host which would double the latency. For instance, if we were to chunk the problem size into two and perform DDOT operations such that T_{ddot1} and T_{exch1} are the times to perform DDOT and data transfers for the chunk, then we need T_{ddot2} to be greater than $T_{ddot1} + T_{exch1}$ to achieve overlap and get improvements. However, based on the transfer latencies, we find that only if we chunk the message to $1/16th$ the original size, we may achieve a good overlap. This means that intra-Op offloading for DDOT and VMAs for the above sizes may not give significant performance gains. Therefore, we look at the Matvec operation which has more arithmetic intensity.

```
Matvec_Offload_call(void *Ai, size_t aiSize, void *
    Aj, size_t ajSize, void *aData, size_t aDataSize
    , void *X, size_t xSize, void *B, void bSize,
    Request *req);
Matvec_Offload_wait(Request *req);
```

Listing 3. Offload API

Listing 3 describes the API of the proposed intra-op offloading framework. This `Matvec_Offload_call` method takes in the details of matrix A, vectors X, and B as per the equation 2. This method initiates a matrix-vector multiplication operation. `Matvec_Offload_wait` completes the operation initiated for a given request `req`.

Next, we present our scheme to decompose the data and offload them to the DPU. Our goal is to offload the operation described in Equation 2.

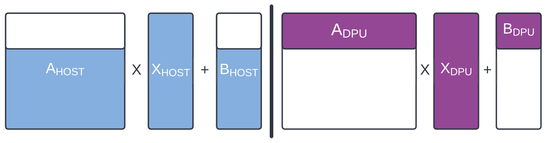


Fig. 6. Basic scheme to offload a portion of Matvec to the DPU

3) Basic Scheme: In this scheme, we split the matrix into two matrices of the same number of columns but different numbers of rows as shown in Figure 6. Let N be the number of rows and columns of the A matrix, and k be the fraction of data transferred to the DPU. In this scheme, we send kN elements of the B vector and receive the same-sized output vector from the DPU. The size of the X vector sent to the DPU is N.

4) Onloading Scheme to Reduce the Data Transfer Cost: This section describes the 'Onloading' scheme to address the second challenge of reducing the data transfer cost from host processes to the DPU processes. Though we describe this scheme in the context of Intra-Op Matvec offload, this can also be extended to the Multi-Op offload scheme.

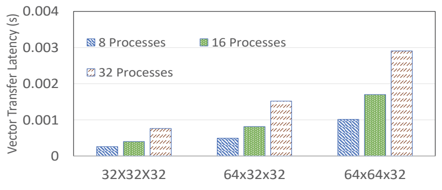


Fig. 7. Impact of increasing the number of processes in the vector transfer latency.

Before explaining the scheme, we perform a motivating experiment on Testbed 1 in which different sets of host processes transfer data to DPU in such a way that the total data transferred for each set remains the same. Figure 7 shows the time to transfer the vectors required for AMG-PCG matrix multiplication code from the host processes to the

DPU processes for problem grid sizes 32x32x32, 64x32x32, and 64x64x32. For each of these problems, we transfer 60% of rows for 8 processes, 30% of rows for 16 processes, and 15% of rows for 32 processes on a single node. The reason for doing this is to keep the size of the total data transferred the same for all the cases. We observe that the latency of the transfer increases with the number of host processes despite the total data size remaining constant. From this, we infer that it is beneficial to tune the number of processes participating in the data transfer to minimize contention. This motivates our onloading scheme.

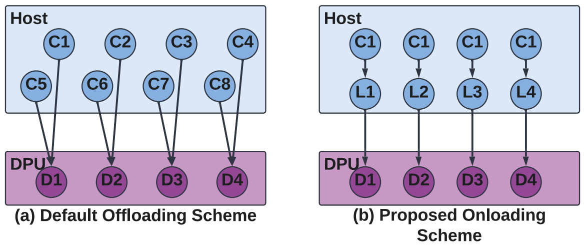


Fig. 8. Comparison of the "Default" Offloading and Proposed Onloading Schemes. In the former, each host "client" (Cx) offloads an equal amount to a given proxy process on the DPU (Dx). In the latter, selected leaders (Lx) are given work downloaded by the clients, who then offload said work alongside their own.

As shown in Figure 8(a), in the default offload scheme all the host processes offload a certain portion of their data to the DPU. The amount of data offloaded is tuned for different architectures to give the best performance. Figure 8(b) shows the onloading scheme. In this scheme, only a fixed number of candidate processes (called leaders) will offload a portion of their workload to the DPU. The other non-leader processes (called clients) will offload their work to the leader processes. To balance the workload, the amount of work offloaded by the leader processes is more than the amount of work offloaded by the clients. This way, we can bring down the communication time for a given amount of data to be offloaded to the DPU. Note that both in the default offloading and the onloading scheme, the total amount of data transferred is the same. However, in the onloading scheme data transferred per leader is increased.

Algorithm 2 describes the steps involved in the onloading scheme. `Matvec_Onload_call` method is invoked by all host processes to either offload the workload to the DPU or onload the workload to a leader host process. $onloadPercent$ describes the amount of data to be onloaded by every process. In the context of the Matvec operation, it determines the number of rows/columns to be onloaded. There is a shared queue among the host processes within a node to exchange the onload workload information. The client process enqueues the workload information to the shared queue after scaling the input data based on the $onloadPercent$. This is described by the `Onload` method. The leader processes scales the input and offloads the workload to the DPU process. Since the leader process will perform work for $clientsPerLeader$ clients, the leader process offloads $onloadPercentX(clientsPerLeader + 1)$ percentage of the input data. For example, if there are 32 processes per node, 8 leaders per node, for a $onloadPercent$ of 15%, each leader will offload $4 \times 15 \rightarrow 60\%$ of the input to the DPU process. After initiating offload to the DPU, leader processes will poll for clients' work request information in the shared queue, perform the operation, and update a status buffer to indicate the completion of the client's request. This is done by the `schedClientOnloadReq` method. In this method, for each client's work request, the leader process maps the client's input data buffers to its local address space through XPMEM[9]. This way the leader process can directly perform the computation on the mapped buffers and the data transfer is implicit. Note that all methods take onloadPercent as input, which the user provides. offloadPercent is calculated based on onloadPercent. The number of leaders can be tuned to obtain the best overall time.

The `OffloadDPU` method first scales the workload inputs according to the calculated offload percentage. Then, it sends matrix A, vectors X and B to the DPU and initiates the Matvec offload by calling `InitiateDPUMatvec`. Transferring matrix A to the DPU is expensive, therefore it is done so only when A is updated by the solver algorithm. Typically, PCG algorithms do not update A during every iteration. Therefore, in the common case, A is only transferred in the first iteration.

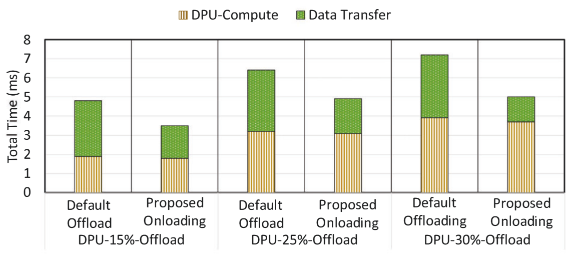


Fig. 9. Comparison Profiling of the Compute and Data Transfer when offloaded to the DPU for our Default Offloading and Proposed Onloading Schemes

To understand the impact of the onloading scheme, in Figure 9 we show the amount of time spent in each DPU process waiting for the data and performing Matvec computation for both schemes. We used AMG-PCG benchmark with problem size 32x32x32 for this experiment. This is a single node experiment on Testbed 1 with 32 host and 16 DPU processes. For the onloading scheme, we have manually tuned the number of leaders. We can see that the time spent waiting for the data is reduced for the optimized scheme, and the best possible split for the optimized scheme is 30%.

D. Extending the On-loading Scheme for the Multi-op Scheme

In the multi-op offloading scheme, we offload a set of non-dominant operations as per the splitting scheme. To understand the performance of our multi-op scheme, we offloaded the PIPECG algorithm of PETSc to the SmartNIC. We compared the performance of the offloaded scheme with the pure host runs of the PIPECG algorithm on Testbed 2 (from section VIII-A) with 96 processes on the host (one node) and 16

Algorithm 2: Onloading Scheme to reduce data transfer cost

```
1  Function Matvec_Onload_call(workInfo, onloadPercent):
2      $dop \leftarrow$
3      getOffloadPercent($onloadPercent$)
4      if $isLeader$ then
5          OffloadDPU($workInfo, dop$)
6          schedClientOnloadReq($sharedQueue$)
7      else
8          OnloadLeader($workInfo, onloadPercent$)
9  Function OnloadLeader(workInfo, onloadPercent):
10     $scWinfo \leftarrow$
       scaleWorkload($workInfo, onloadPercent$)
11     workInfoEnque($scWinfo$, $sharedQueue$, $myLeader$)
12 Function getOffloadPercent($onloadPercent$):
13     return $onloadPercentX(clientsPerLeader + 1)$
14 Function OffloadDPU($workInfo, dop$):
15     $scWinfo \leftarrow$ scaleWorkload($workInfo, dop$)
16     if $matinfoCache.isMatAUpdated$ then
17         SendtoDPU($scWinfo.A$)
18     SendtoDPU($scWinfo.X$, $scWinfo.B$)
       InitiateDPUMatvec($scWinfo$)
```

processes on the SmartNIC. We used the KSPbench benchmark as described in the section VIII. We observed that the offloaded scheme performed up to 30% worse than the pure host-based scheme. Further profiling showed that this was due to 1) high data transfer time, and 2) an imbalance in the amount of computation offloaded to the DPU. To improve the performance, we extended the onloading scheme proposed for Matvec offload to the multi-op offload. In this scheme, only a selected set of leader processes offload all the non-dominant operations to the DPU processes. The non-leader processes split their non-dominant operations with their leader process such that all the processes get an equal amount of computational workload. We divide the vector sizes among the leader and non-leader processes equally to balance the computation. For instance, assume that there are N leader processes and M non-leader processes. If K is the size of the vectors used in each non-dominant operation, then each non-leader process performs non-dominant operations on $M/(M+N) \times K$ elements and leader processes perform the non-dominant operation on $N/(M+N) \times K$ elements.

Figure 10 shows the performance gain of the onloading scheme compared to the default offloading scheme for the PIPECG algorithm on problem sizes ranging from 192×192×192 to 288×288×288. We observe up to 38% improvements with our proposed onloading scheme with 48 leaders compared to the default offloading scheme.

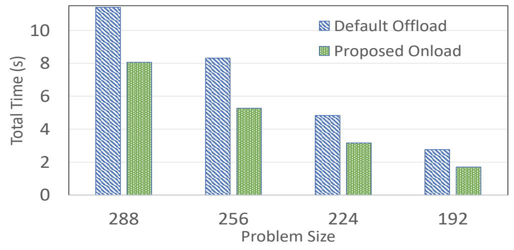


Fig. 10. Benefits of the onloading scheme for multi-op offload with PETSc PIPECG algorithm with 48 leaders

E. Applicability of the proposed schemes to other PCG algorithms

Non-Blocking PCG, Pipelined PCG, and 2-Iteration Pipelined PCG are some examples of scalable variants of the PCG algorithm [5]. These variants are aimed at overlapping the MPI_Allreduce operation with Matvec and Preconditioner operation at the cost of additional VMA operations [5]. BiCGStab is another Krylov Solver similar to PCG but for non-symmetric linear matrixes. All these solvers have a computational graph similar to the generic graph shown in figure 4. The main differences between these algorithms are 1) the number of VMA, DDOT, and Scalar operations and 2) the data dependencies of these computation operations. Therefore, we can apply the Algorithm 1 to get the host and DPU operations. For example, when we apply the Algorithm 1 to the Non-Blocking PCG algorithm [5], the host split will include the Matvec, Preconditioner, VMA operations $z \leftarrow z - alpha * S$, $s \leftarrow Z + beta * s$. We have used the same notations as [5]. This way we can use Algorithm 1 to determine the host and DPU split for any new PCG algorithm. The Multi-Op offload API shown in the listing 1 can be used to offload the DPU operations. The Intra-Op offload API shown in 3 can be used to offload the Matvec operation.

VIII. Experimental Evaluation

A. Experimental Setup

For our Matvec schemes, we modified HYPRE's Matvec offload method. Since HYPRE has a powerful multigrid preconditioner we modified the LLNL AMG benchmark's HYPRE implementation. Since HYPRE does not have native support for the pipelined CG algorithm, we modified PETSc's PIPECG solver to offload the pipelined CG algorithm. The DPU part of the proposed scheme is implemented in our custom runtime that is capable of launching DPU processes.

We use PETSC's ksp_bench to evaluate the performance of our PIPECG optimization. This benchmark solves 3D Laplacian with a 27-point finite difference stencil.

We observed that the final norm of the offloaded version of our workloads exactly matched the pure-host version for our evaluations. To ensure this, we implemented the DDOT, VMA, and Matvec in the same way the host version of the PETSc/HYPRE implemented these operations. To ensure identical network performance, we force the same BF3 as the HCA for pure-host and offloaded runs. For all our evaluations, we report an average of 5 runs.

Since **Testbed 2** has one node, we show single node results on **Testbed 2** in Figure 14 and scaling results on **Testbed 1** in Figures 11, 12, and 13.

B. Multi-Op offload results with pipelined PCG algorithm

In this section, we evaluate the performance of modified PETSc's PIPECG algorithm by offloading the VMA/DDOT operations as described in Section VII-B. Figure 11, shows the strong scaling results of offloading the PIPECG algorithm on **Testbed 1** for a fixed overall problem size of 256 which gets divided evenly across 64, 128, and 256 processes. The node counts were 2, 4, and 8 respectively with 32 processes per node. We observe improvements up to 24% with the proposed offload method. Figure 14(a) shows the one node results on **Testbed 2**, with 96 host processes where we observe up to 10.4% improvement compared to the pure host baseline. The Multi-Op offload scheme used DPU cores for performing the VMA and DOT product operations. DPU NIC was used to transfer data from the host to the DPU and to perform MPI_-Allreduce for DDOT operations.

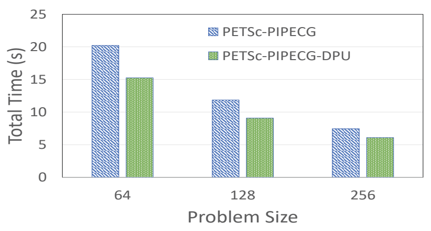


Fig. 11. Strong scaling results showing the benefits of offloading VMA/DDOT in PIPECG for a problem size of 256

C. Matvec offload benefits

In this section, we compare the performance of our Intra-Op offload Matvec schemes with that of pure host Matvec. For our evaluations, we timed the performance of the Matvec operation used in AMG's PCG benchmark. For these runs, we have compared the max latency (across 256 processes) of our proposed schemes with that of the pure host performance. In Figure 12(a), we ran the PCG benchmark with a problem size of $64\times32\times32$ which resulted in a matrix of 64K rows and columns. Here, we observe up to 21% improvement in scheme 2, whereas with scheme 1 we see improvement only up to 8%. For both the schemes the best improvement occurs when 25% of the entries are offloaded to the DPU. In Figure 12(b), the matrix size is doubled to have 128K rows and columns. In this case, we observe that the proposed scheme gives up to 19% improvement compared to the host, and the default scheme only degrades compared to the host. Finally, in Figure 12(c), the rows and columns are doubled to 256K entries. In this case, we observe that the proposed scheme gives up to 20% improvement and the default scheme degrades even further compared to a matrix of size 128K. Similarly, we show single node results for Matvec offload on Testbed 2 in Figure 14(b). For space reasons, we only show the best onloading numbers for each problem size on Testbed 2. We observe around 19% improvement for the problem of 32X32X32 and 13% for the largest problem size of 64X64X64. On Testbed 2, the best performance occurred when 12% of the entries were offloaded using the onload scheme. The Matvec offload scheme used BF3 cores for performing the Matvec computation. BF3 NIC was used to transfer data from the host to the BF3.

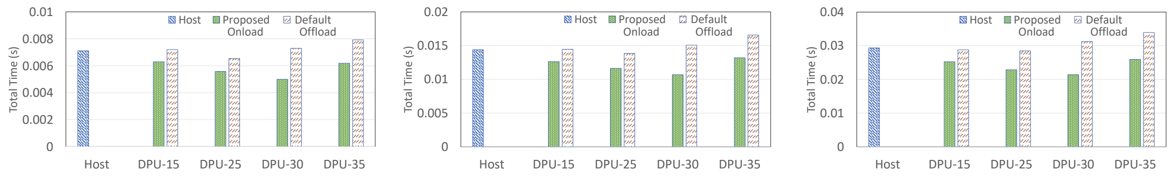
D. AMG HYPRE results

In this section, we use the LLNL AMG benchmark to study the benefits of our proposed schemes. We show results for the PCG problem. The preconditioner used is the BoomerAMG preconditioner from HYPRE. We report weak scaling numbers by keeping the per-process problem the same for different numbers of processes.

Figure 13(a) shows that for a problem size of $32\times32\times32$, the proposed DPU-offload scheme gives up to 24% improvements. In, Figure 13(b), we observe that the offloaded PCG shows up to 22% for a problem size of $64\times32\times32$ compared to the pure host runs. For a problem size of $64\times64\times32$ as shown in Figure 13(c), the DPU offload scheme does 21% better than the host-only scheme. For $64\times64\times64$, we get up to 22% improvement. To understand the reason for improvements we profiled and found that the preconditioner takes about 75%, Matvec takes about 15% of the time, and the rest is consumed by VMAs and DDOTs. Therefore, the majority of the benefits arise from preconditioner and Matvec offloading. We further found that even in the preconditioner, Matvec takes the maximum amount of time, therefore we see improvement in the preconditioner's execution time which in turn improves the PCG's execution time. Similarly, on Testbed 2 we observe benefits from 9.5% to 11.5% as shown in Figure 14(c). As explained in Sections VIII-B, VIII-C, we use the DPU's NIC and DPU cores according to perform data movement and computation phase of VMA, DDOT and Matvec operations.

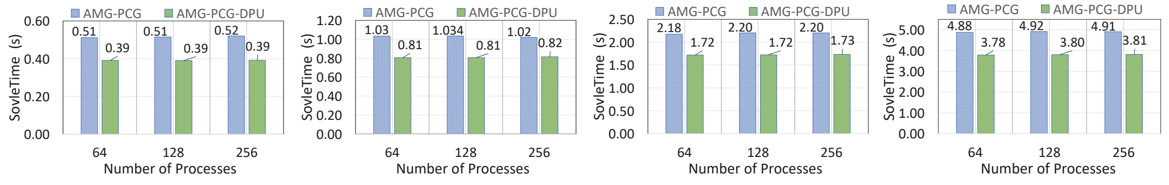
IX. Related Work

Multiple works have been published over the past three years to utilize SmartNICs for offloading communication and computation. Karamati et al. [11] modified the MiniMD application to decrease dependencies to increase task-level parallelism, making it possible to offload lighter computation to the BlueField-2 DPU (BF2). Williams et al. [24] investigated the use of the BF2 SmartNICs in accelerating scientific workloads like the PENNANT proxy application. They also investigated/proposed an independent API, OpenSNAPI, as a possible use case and tool to develop middleware to fully utilize SmartNICs [23]. Jain et al. [8] investigated the use of the SmartNICs for accelerating Distributed Deep Learning by offloading either the data augmentation, training, or a mix of both to the BF2s. A common theme in the above work is the manual selection of phases to offload to the SmartNIC. Unlike these works, our work provides a general scheme to automatically select operations to offload any variant of the PCG algorithm. Furthermore, we also provide an efficient scheme to reduce data transfer costs. On the communication



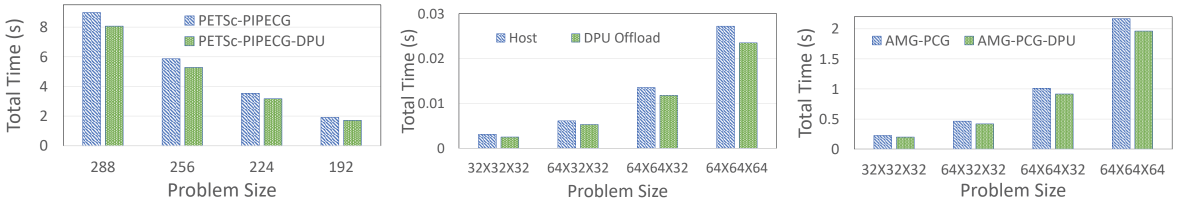
(a) Problem: 64×32×32 (b) Problem: 64×64×32 (c) Problem: 64×64×64

Fig. 12. Matvec 8N 32 PPN numbers showing the performance 1) Host, 2) DPU offload Scheme-1, 3) DPU offload Scheme-2



(a) Problem: 32×32×32 (b) Problem: 64×32×32 (c) Problem: 64×64×32 (d) Problem: 64×64×64

Fig. 13. Weak Scaling results with fixed problem size per process showing overall Solve Time for AMG PCG benchmark on 2, 4, and 8 nodes with 32 PPN.



(a) PETSc PIPECG (b) Matvec (c) AMG-PCG

Fig. 14. Single Node results for PIPECG, Matvec, and AMG-PCG algorithms on **Testbed 2** with 96 host processes and 16 proxy processes

side, the authors of [3], [21], and [22] have proposed increasingly advanced techniques for improving nonblocking alltoall and broadcast communication by offloading these primitives to the DPU; their approaches were less restrictive than offloading computation, as every design was placed within the context of an MPI library. Moreover, the DDOT operation offloaded by our work also contains MPI_Iallreduce communication primitive.

Scalable variants of PCG [5], GMRES [25], [7] have nonblocking MPI_Iallreduce that can be overlapped with an independent computation at the cost of more VMA, and DDOT operations compared to the base version of the algorithm. They provide more opportunities for communication libraries such as MPI, to optimize the algorithms. Kandalla et al. [10], have optimized a similar variant of PCG by efficiently offloading MPI_Iallreduce operation to the Host Channel Adapter (HCA) using the CoreDirect feature. The shortcomings of this work are 1) they only optimize one variant of PCG and 2) on modern HCAs MPI_Iallreduce are not a bottleneck for PCG solvers. Our work offloads computational operations including MPI_Iallreduce that can work on any variant of the PCG algorithm.

The following works attempt to create frameworks that aid in the development of applications that can be offloaded to the SmartNICs by reducing the programming overhead. Some examples are Floem [20] and iPipe [13], [12]. Floem also includes a language, compiler, and runtime that allows developers to define "elements" of C code that are executable on SmartNICs. iPipe is a hybrid scheduler that aids in process sharing and measures execution costs at runtime. Other tools outside the realm of SmartNICs revolve around functionality like in-network computing; NVIDIA's SHARP [17] leverages their (NVIDIA) switches to execute in-network operations such as All/Reduce to offload compute from the host processor.

X. Conclusion and Future Work

Modern SmartNICs are capable of performing general-purpose computation and communication operations. Specifically, NVIDIA's BF3s have sufficiently robust programmable cores to run custom programs and initiate communication operations. This work identified three key computational operations to offload: DDOT, VMA, and Matrix-Vector multiplications in the context of Krylov Subspace methods. We provided a framework to efficiently offload some commonly used solver algorithms. In our framework, we designed multi-Op and intra-Op schemes to offload these operations to the DPU. For our multi-Op offloading strategy we proposed a generalized splitting scheme to segregate a set of VMA and DDOT operations for different variants of the baseline solver algorithms. We showed the applicability of this scheme to the Pipelined PCG algorithm. For intra-Op offloading, we provided an onloading scheme to reduce the data transfer latency for partially offloading Matrix-Vector operations to the DPU. Using our offloading schemes, we showed up to 21% improvement in Matvec operation, up to 24% improvement in PCG, Pipelined PCG algorithms compared to the CPU baseline on 256 processes. We also showed that our proposed schemes could provide up to 11% improvement on a system with Sapphire Rapids CPU and BF3. An 11% to 21% improvement can save 2.5 to 5 hours of execution time when the solvers are executed for a whole day. As a future work, we would like to extend our optimization to other algorithms such as FFTs, QR algorithms for finding Eigenvalues, etc., by identifying and offloading their fundamental Matrix and Vector-related operations.

References

[1] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, J. Faibussowitsch, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, and J. Zhang, "PETSc/TAO Users Manual," Argonne National Laboratory, Tech. Rep. ANL-21/39 - Revision 3.20, 2023.

[2] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[3] M. Bayatpour, N. Sarkauskas, H. Subramoni, J. Maqbool Hashmi, and D. K. Panda, " BluesMPI: Efficient MPI Non-blocking Alltoall Offloading Designs on Modern BlueField Smart NICs ," in *High Performance Computing*, B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, Eds. Cham: Springer International Publishing, 2021, pp. 18–37.

[4] G. M. D. Corso, O. Menchi, and F. Romani, "Krylov subspace methods for solving linear systems," 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:37427715

[5] P. R. Eller and W. Gropp, "Scalable Non-blocking Preconditioned Conjugate Gradient Methods," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 204–215.

[6] R. D. Falgout and U. M. Yang, "hypre: A Library of High Performance Preconditioners," in *Computational Science — ICCS 2002*, P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 632–641.

[7] P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose, "Hiding Global Communication Latency in the GMRES Algorithm on Massively Parallel Machines," *SIAM Journal on Scientific Computing*, vol. 35, no. 1, pp. C48–C71, 2013. [Online]. Available: https://doi.org/10.1137/12086563X

[8] A. Jain, N. Alnaasan, A. Shafi, H. Subramoni, and D. K. Panda, "Accelerating CPU-based Distributed DNN Training on Modern HPC Clusters using BlueField-2 DPUs," in *2021 IEEE Symposium on High-Performance Interconnects (HOTI)*, Aug 2021, pp. 17–24.

[9] K. Pedretti and B. Barrett, "XPMEM: Cross-Process Memory Mapping," https://gitlab.com/hjelmn/xpmem, [Online; accessed October 31, 2024].

[10] K. Kandalla, U. Yang, J. Keasler, T. Kolev, A. Moody, H. Subramoni, K. Tomko, J. Vienne, and D. K. Panda, "Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters: A Case Study with Conjugate Gradient Solvers," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2012.

[11] S. Karamati, C. Hughes, K. S. Hemmert, R. E. Grant, W. W. Schonbein, S. Levy, T. M. Conte, J. Young, and R. W. Vuduc, ""Smarter" NICs for faster molecular dynamics: a case study," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2022, pp. 583–594.

[12] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading Distributed Applications onto SmartNICs Using IPipe," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 318–333. [Online]. Available: https://doi.org/10.1145/3341302.3342079

[13] M. G. Liu, A. Krishnamurthy, S. Peter, and K. Gupta, "iPipe : A Framework for Building Datacenter Applications Using In-networking Processors," 2018.

[14] B. Michalowicz, K. K. Suresh, H. Subramoni, D. K. D. Panda, and S. Poole, "Battle of the BlueFields: An In-Depth Comparison of the BlueField-2 and BlueField-3 SmartNICs," in *2023 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2023, pp. 41–48.

[15] NVIDIA, "HPC Researchers Seed the Future of In-Network Computing With NVIDIA BlueField DPUs." [Online]. Available: https://blogs.nvidia.com/blog/bluefield-dpus-hpc-isc2022/

[16] NVIDIA, "NVIDIA BlueField-3 DPU Data Sheet," 2022. [Online]. Available: https://resources.nvidia.com/en-us-accelerated-networking-resource-library/datasheet-nvidia-bluefield?lx=LbHvpR&topic=networking-cloud

[17] ——, "NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP)," 2022. [Online]. Available: https://docs.nvidia.com/networking/display/sharpv300

[18] ——, ""nvidia connectx7 ndr400 infiniband adapter card"," 2023. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/networking/infiniband-adapters/infiniband-connectx7-data-sheet.pdf

[19] OSU Micro-benchmarks, http://mvapich.cse.ohio-state.edu/benchmarks/.

[20] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson, "Floem: A Programming System for NIC-Accelerated Network Applications," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 663–679.

[21] N. Sarkauskas, M. Bayatpour, T. Tran, B. Ramesh, H. Subramoni, and D. K. Panda, "Large-Message Nonblocking MPI_Iallgather and MPI Ibcast Offload via BlueField-2 DPU," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2021, pp. 388–393.

[22] K. K. Suresh, B. Michalowicz, B. Ramesh, N. Contini, J. Yao, S. Xu, A. Shafi, H. Subramoni, and D. Panda, " A Novel Framework for Efficient Offloading of Communication Operations to Bluefield SmartNICs ," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 123–133.

[23] B. K. Williams, S. W. Poole, and W. K. Poole, "Exploring OpenSNAPI Use Cases and Evolving Requirements [Slides]," 8 2021. [Online]. Available: https://www.osti.gov/biblio/1814738

[24] B. K. Williams, W. K. Poole, and S. W. Poole, "Investigating Scientific Workload Acceleration using BlueField SmartNICs [Slides]," 3 2020. [Online]. Available: https://www.osti.gov/biblio/1607904

[25] I. Yamazaki, M. Hoemmen, P. Luszczek, and J. Dongarra, "Improving Performance of GMRES by Reducing Communication and Pipelining Global Collectives," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 1118–1127.

[26] U. Yang, R. Falgout, and J. Park, "Algebraic Multigrid Benchmark, Version 00," 8 2017. [Online]. Available: https://www.osti.gov//servlets/purl/1389816