FP-AMG: FPGA-Based Acceleration Framework for Algebraic Multigrid Solvers

Pouya Haghi*, Tong Geng*, Anqi Guo*, Tianqi Wang†, Martin Herbordt*
*Department of Electrical and Computer Engineering, Boston University, Boston, MA, USA
†Department of Physics, University of Science and Technology of China, Hefei, China
Email: *{haghi, tgeng, anqiguo}@bu.edu, †tqwang@mail.ustc.edu.cn, *herbordt@bu.edu

***Abstract*—Partial Differential Equations (PDEs) are fundamental to many real-world scientific computing applications and so their optimization has undergone decades of study. Algebraic multigrid (AMG) is one of the most well-known solvers, being widely adopted in High Performance Computing (HPC) due to its good scalability. Acceleration of AMG is known to be very challenging, due to the following reasons: (1) irregular computation patterns, (2) random memory access, and (3) a large number of kernels with various computation types. To the best of our knowledge, there is no prior work on FPGA-based acceleration of AMG.**

To tackle these challenges, we propose an efficient FPGA-based reconfigurable framework, called FP-AMG, for high-performance AMG calculation. In order to obtain full pipeline utilization, we propose a novel and scalable architecture that can be reused for all kernels in AMG. Given that AMG is strictly memory-bound, we propose algorithmic and architectural optimizations to ensure nearly ideal use of memory bandwidth. The efficiency of FP-AMG is evaluated with six well-known benchmarks on two FPGA devices: one with and one without high bandwidth memory (HBM). The experimental results are compared with a highly optimized Intel Xeon E5-2680-V4 implementation of the state-of-the-art *HYPRE* library. Our experiments show that FP-AMG can achieve average speedups of 2.5× and 6.6×, for FPGAs without and with HBM, respectively.

I. Introduction

Computer simulation is the core of HPC and is playing an increasingly important role in scientific research and industrial development. A large fraction, if not the majority, of computer simulations require solving PDEs; in the most widespread applications, these involve solving large systems of sparse linear equations. The demand for shorter run time while achieving a high accuracy is paramount in solving these large linear systems. Although traditional (geometric) multigrid solvers provide linear (*O(N)*) computational complexity, solving highly unstructured problems (*e.g.* computational fluid dynamics) with geometric multigrid is very complex or even impossible [1]. The advent of Algebraic Multigrid (AMG) [2]–[4] has made solving systems of equations with unstructured grids possible, though still challenging, while maintaining the same asymptotic complexity as geometric multigrid.

Researchers have spent much effort on parallelizing and optimizing AMG for CPU clusters [5]–[7] and accelerating it with GPUs [8]. In both cases, however, it is difficult to achieve desired speedups because of architectural limitations. For CPUs a single node does not provide sufficient parallelism; therefore large-scale CPU clusters are generally used. But while clusters supply sufficient FLOPs, communication becomes the bottleneck and severely limits scalability [9], [10]. For GPUs, the irregular data accesses and complex data dependencies of AMG algorithms lead to poor cache locality, and therefore low utilization, again limiting performance with even a single GPU [11]. In contrast, FPGAs' attributes of (1) customizable datapaths and, (2) flexible memory and computing subsystems, have enabled efficient acceleration of various applications with characteristics similar to AMG [12]–[15] as well as their use in scalable HPC clusters [16]–[19].

Still, acceleration of AMG on FPGAs is quite challenging and complicated. In fact, AMG invokes a variety of kernels on a large but unpredictable volume of data, making high hardware utilization difficult to achieve. Furthermore, not only does AMG demand irregular memory access, but also the size of required memory is not known in advance [7]; together these attributes of AMG require a well-considered design of the memory subsystem.

In this paper, we propose a novel reconfigurable framework, FP-AMG, to accelerate AMG on FPGAs. To address the challenges mentioned above, FP-AMG is designed with (1) a smart memory subsystem with finely-tuned architecture design and (2) novel reconfigurable computation engines that support and can be efficiently used by all kernels in AMG. All functions–including Sparse Matrix Multiplication (SpGEMM), sparse matrix vector multiplication (SpMV), interpolation construction, and parallel large-scale maximum search–are fully supported by the proposed engine. Although we do not pursue this idea further here, this design can potentially be generalized to (the many) other applications that rely on scalable SpMV or SpGEMM. We summarize the contributions of this work:

- The first FPGA-based AMG accelerator, FP-AMG.
- A novel reconfigurable and highly scalable architecture that can be successively used by various kernels in AMG and so ensure continuously high device resource utilization.
- A smart memory subsystem architecture design to address the irregular memory access issue of AMG and also a number of optimizations to reduce the memory demand.
- A novel methodology for design parameter tuning, which embraces the dynamic nature of AMG to efficiently map FP-AMG to different FPGAs.
- Experimental results that show FP-AMG provides 6.6× and 2.5× speedup for FPGAs with and without HBM

support, respectively, compared with an optimized implementation running on a server-class Intel Xeon CPU.

The organization of this paper is as follows. Section II introduces AMG. Section III describes in detail the FP-AMG framework including architecture, design, memory partitioning, data flow, optimizations, and mapping methodology. Section IV presents and analyzes experimental results. In Section V, related work is discussed. Finally, Section VI provides a summary and future directions.

II. Background

Suppose that the linear systems of equations to be solved are given as follows:

$$Ax = b \tag{1}$$

where *A* is a sparse matrix with elements a_{ij} ($A \epsilon R^{n \times n}$), *x* is an unknown vector to be calculated ($x \epsilon R^n$), and *b* is the right hand side (RHS) of the equation ($b \epsilon R^n$). Let us define the grid as the set of grid points in which each grid point *i* corresponds to x_i and the grid is denoted by $\omega = \{1, 2, ..., n\}$. The systems of equations, which may be derived from PDE discretization of a given problem, govern the relationships between grid points.

Multigrid solvers start with an initial guess of the solution. Then smoothing is iteratively applied to reduce the high frequency portion of the error *e*. The low frequency portion of the error is eliminated by solving the error on a coarser (smaller) grid and then interpolating it back onto a finer (larger) grid. Hence, multigrid solvers in general have *L* levels; there are two *grid transfer operators* to transfer from one level to another: restriction *R*, which converts a fine grid to a coarse grid, and interpolation *P*, which does the opposite. The size of the restriction (interpolation) matrix is $n \times \hat{n}$ ($\hat{n} \times n$), where $\hat{n}$ is much less than *n*.

In multigrid methods, interpolation, restriction, and thus obtaining the next grid, is trivial and predefined; in contrast AMG makes use of the information of matrix *A* (not the grid) which is not known in advance. Therefore, in AMG there is a *setup phase* in which coarsening and interpolation are constructed in terms of matrices. Then in the *solve phase* the problem is solved. Note that FP-AMG supports both setup and solve phases.

A. Setup Phase

Assume that ω^l, A^l, P^l, and R^l are the grid, grid operator, interpolation, and restriction in level *l*, respectively. Algorithm 1 [20] shows the AMG setup phase. First, in **Calc_StrengthMat** function, strength matrix (S) is calculated (line 3). The strength matrix (of size $n \times n$) is defined as

$$S_{ij} = \begin{cases} 1 \text{ if } i \neq j \text{ and } -a_{ij} \geq \theta \times \max_{k \neq i}(-a_{ik}) \\ 0 \text{ otherwise} \end{cases} \tag{2}$$

where θ is a positive constant number < 1. We say that grid point *i strongly depends* on point *j* if S_{ij} is equal to one.

Next, an unidirected graph $G = (V, E)$ is defined in **Build_Graph** function (line 4) where $E = \{\{i, j\} \epsilon \{V \times V\} | S_{ij} = 1 \; or \; S_{ji} = 1\}$. Subsequently, in each level the grid is partitioned into coarse grid points (C-points) and fine grid points (F-points) (line 5) which is done by the **coarsening** algorithm (Algorithm 2 [20]). C-points are those that will be chosen in the next coarse level.

Algorithm 1 AMG Setup Phase

Input: A^0
Output: lastLevel, minGridSize, A^i, P^{i-1}, R^{i-1} $\forall$ i $\in$ {1, …, lastLevel-1}
1: $k \leftarrow 0$
2: **while** $\omega^k \geq minGridSize$ **do**
3: $S \leftarrow$ **Calc_StrengthMat**(A^k)
4: $G \leftarrow$ **Build_Graph**(S)
5: $\{C, F\} \leftarrow$ **coarsening**(G)
6: $\omega^{k+1} \leftarrow C$
7: $P^k \leftarrow$ **interpolation**(A^k, S, C, F)
8: $R^k \leftarrow$ **Interpolation**(A^k, S, C, F) ▷ often $R^k = (P^k)^T$
9: $A^{k+1} \leftarrow R^k A^k P^k$ ▷ **triple matrix multiplication**
10: $k \leftarrow k + 1$
11: **end while**
12: $lastLevel \leftarrow k$

Algorithm 2 PMIS Coarsening Algorithm

Input: G = (V, E) with weights w(i) = S_i^{count} + Rand ([0, 1]) $\forall$ i $\in$ V
Output: F (set of F-points), C (set of C-points)
1: $F \leftarrow \{i \in V S_i^{count} = 0\}$
2: $C \leftarrow \emptyset$
3: $V' \leftarrow V \backslash F$ ▷ Take the F-points out of the remaining vertex set.
4: $G' \leftarrow (V', S)$ ▷ The subgraph induced by the remaining vertex set.
5: **while** $V' \neq \emptyset$ **do**
6: i $\in$ I $\iff$ w(i) > w(j) $\forall$ j with {i, j} $\in$ E' ▷ Choose an independent set I of V' in G'.
7: $C \leftarrow C \cup I$ ▷ Make all elements of I, C-points.
8: $F \leftarrow F \cup F_{new}$, with $F_{new} = \{j \in V' \backslash I | \exists i \in I : i \in S_j\}$ ▷ Make all elements of V' $\backslash$ I that are strongly influenced by a new C-point, F-points.
9: $V' \leftarrow V' \backslash \{I \cup F_{new}\}$ ▷ Remove all new C- and F-points.
10: $G' \leftarrow (V', S)$
11: **end while**

Then the **interpolation** function creates the *R* and *P* matrix (line 7-8). It specifies how F-points can be expressed in terms of C-points. For each F-point *i*, the *neighborhood* of *i* is defined as the set of all points $j \neq i$ such that $a_{ij} \neq 0$. These points can then be categorized:

- strongly depends on *i*, is a C-point, and denoted as C_i
- strongly depends on *i*, is an F-point, and denoted as D_i^s
- does not strongly depend on *i* and is denoted as D_i^w

The interpolation matrix can then be constructed by the following formula [4]:

$$P_{ij} = -\frac{a_{ij} + \sum_{m \epsilon D_i^s} \left(\frac{a_{im} a_{mj}}{\sum_{k \epsilon C_i} a_{mk}} \right)}{a_{ii} + \sum_{n \epsilon D_i^w} a_{in}} \tag{3}$$

Next, grid operators are created by a triple matrix multiplication (two SpGEMMs) (line 9). Finally, the algorithm terminates whenever the number of C-points is less than a threshold. We divide the setup phase into two parts: **restriction-interpolation construction** (line 3-8) and **triple matrix multiplication** (line 9). It should be noted that there are a large number of algorithms for coarsening and interpolation. In this work, we focus on classical interpolation and, due to its high parallelism, the PMIS coarsening algorithm [20] as it is used in [7].

B. Solve Phase

Algorithm 3 describes a popular V(μ_1, μ_2)-cycle [20]. The entire V-cycle is repeated for a number of iterations until the error is below a threshold tolerance. Three important kernels used in this phase are **smoothing**, **residual computation**, and **restriction/interpolation**. The last two ones rely on pure SpMV kernel. The formula for residual computation is as follows:

$$r = b - Ax \tag{4}$$

For smoothing, we focus on hybrid Gauss-Seidel in this work due to its satisfactory convergence which also used in [7]. We call it a *pseudo* SpMV kernel due to its similar computation pattern to that of SpMV. The so-called *pseudo* SpMV contains a parallel and a sequential part. Given *P* parallel processors, the following formulas illustrate *pseudo SpMV* for $\frac{n}{P}$ iterations:

$$x_k^{new} = \frac{b_k - A_k x^{in}(k,r) + a_{kk} x_k^{old}}{a_{kk}} \tag{5}$$

$$x_j^{in}(k,r) = \begin{cases} x_j^{new} \text{ if } k > j \text{ and } \frac{n}{P} \times r \leq j < \frac{n}{P} \times (r+1) \\ x_j^{old} \text{ otherwise} \end{cases} \tag{6}$$

where $k = \frac{n}{P} \times r + i \quad \forall\ r \epsilon \{0, ..., P-1\}$ and x_k, b_k, and a_k denote the k^{th} element of x and b vector and k^{th} row of matrix *A*, respectively, and *i* represents iteration number. x^{new} is the updated x vector after smoothing while x^{old} is x vector before smoothing.

Algorithm 3 AMG Solve Phase

Input: u^0 (initial guess), f^0, $(A^i \forall i \in \{0, \ldots, lastLevel-1\})$, $(P^i, R^i \forall i \in \{0, \ldots, lastLevel-2\})$
Output: u^0 (solution)

V-Cycle $(A^k, R^k, P^k, u^k, f^k)$
if k == lastLevel-1 **then**
  Solve $A^k u^k = f^k$ with a direct solver.
else
  Apply smoother μ_1 times to $A^k u^k = f^k$. ▷ **Smoothing**
  $r^k \leftarrow f^k - A^k u^k$ ▷ **residual computation**
  $r^{k+1} \leftarrow R^k r^k$ ▷ **restriction**
  V-Cycle $(A^{k+1}, R^{k+1}, P^{k+1}, e^{k+1}, r^{k+1})$
  $e^k \leftarrow P^k e^{k+1}$ ▷ **interpolation**
  $u^k \leftarrow u^k + e^k$ ▷ **correct solution**
  Apply smoother μ_2 times to $A^k u^k = f^k$. ▷ **Smoothing**
end if

III. FP-AMG Framework

In this section, we discuss the design details of FP-AMG framework. First, the proposed architecture including the efficient memory subsystem and reconfigurable computation engine, is introduced. Second, we elaborate the proposed memory partitioning method. Third, the data flow of FP-AMG at each phase is presented. Fourth, three extra optimizations on memory storage requirement, memory bandwidth, and data reuse are introduced. Finally, the methodology of how FP-AMG is mapped onto FPGAs as well as the parameter tuning method of that mapping is presented.

A. Architecture

As stated in Section I, different kernels must be invoked in AMG. One solution is to design a specialized hardware for each kernel and connect the intermediate buffers in a pipeline manner. This approach, however, is not possible in AMG since each kernel must wait until previous kernel has finished. In this work, we take advantage of the similarities of computation patterns among various kernels to design a flexible architecture which can be reused recursively by all AMG kernels.

Various kernels in AMG require random access to on-chip memory. As the number of processing units increases (for more parallelism) the number of simultaneous memory requests also grows. This increases both the memory access time and the complexity of the memory interface. To solve this problem, we propose a scalable island-based architecture with bank resolvers to efficiently process the irregular data access requests.

The overall architecture is depicted in Fig. 1. It consists of a number of islands where each island has a 2-D PE array, input buffers, output buffers, banked vectors, and banked matrices. Banked vector (matrix) buffers, as the name implies, are divided into banks to efficiently supply multiple memory requests from PEs. Banked vectors are replicated for each island to reduce the complexity and memory access time of a large number of requests.

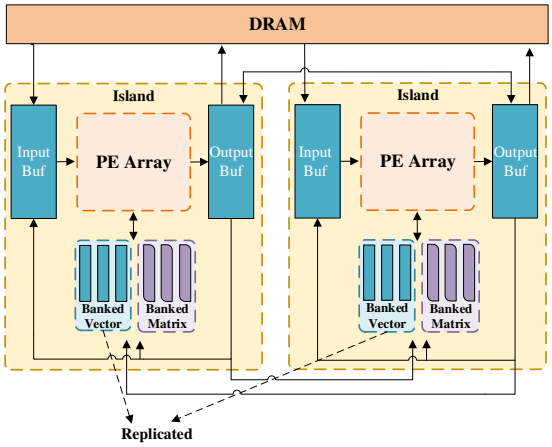

Fig. 1. FP-AMG Overall Architecture

Fig. 2 shows the detailed architecture of an island. It is formed by a variety of components including PEs, end of row processing unit (EoR-PU), bank resolver, and dynamic buffers. In the following subsections, we elaborate on each of these components.

1) PE: Each PE in FP-AMG can perform multiplication, accumulation, comparison, or their combinations as required by AMG kernels. The datapath is highly configurable to enable or disable modules during each AMG phase. Moreover, forward data flow and backward propagation of data between PEs is supported. This versatility allows FP-AMG to support different kernels.

2) EoR-PU: For each row of the array there is an EoR-PU which performs the post-processing and writes to the output

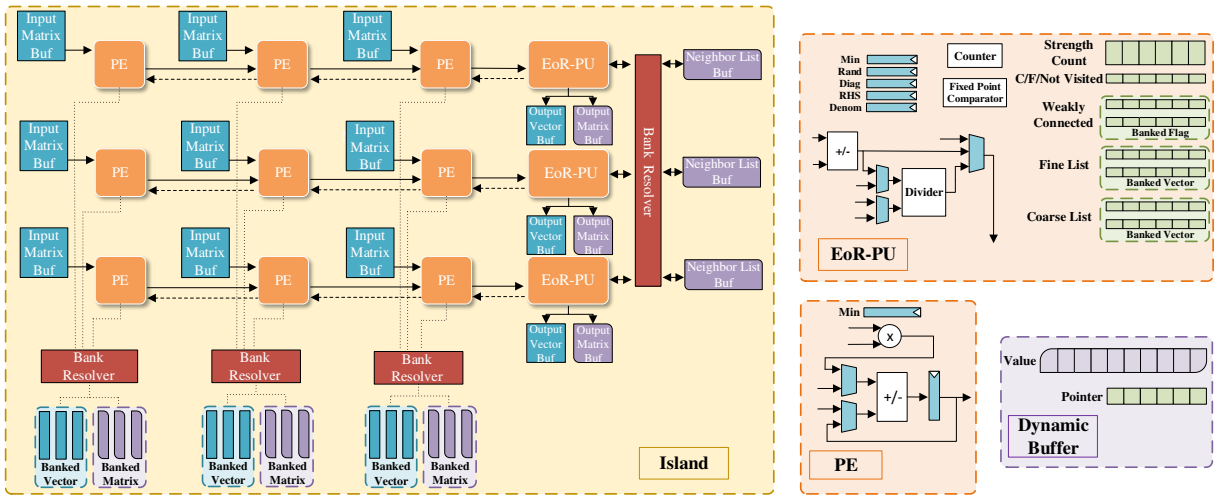

Fig. 2. FP-AMG Detailed Architecture

buffers. There are also some buffers which store intermediate results during the coarsening/interpolation construction phase.

3) Buffers: Banked vector (banked matrix) is employed in the SPMV and *pseudo SpMV* (SpGEMM) kernels and holds the input vector (matrix). Banked vector contains two distinct *new* and *old* buffer required by *pseudo* SpMV kernel. *Neighbor list*, *strength count*, and *weakly connected* buffers store the indices of *strong* neighbors, number of strongly connected points (S_i^{count}), and a flag that determines the weak connection for the current grid point, respectively. C/F/not-visited is a 2-bit buffer which indicates whether the grid points are C-point, F-point, or in not-visited state. Fine list (coarse list) is a small register file (RF) which holds a number of F-points (C-points). It will be discussed in the next subsections.

4) Bank Resolver: This component manages irregular memory access to either banked vector or banked matrix (depending on the type of kernel) and directs the requests to the correct bank. For the banked vector in *pseudo* SpMV kernel, it selects either the *new* or *old* buffer according to *ROW* and the requested address. In case of bank conflicts, the bank resolver prioritizes requests and queues the IDs together with the addresses. Also, it can broadcast data if all of the requests are for the same address within a bank.

5) Dynamic Buffer: This type of memory is itself consisted of two buffers: *value* and *pointer* (points to *value* buffer). As a result, the *value* buffer could be partitioned into segments with desired sizes for different iterations. The proposed dynamic buffer design is adopted in the output matrix buffers for the SpGEMM kernel and the *neighbor list* buffer during the restriction/interpolation construction phase.

B. Memory Partitioning

Memory partitioning can have a substantial effect on the overall performance by simplifying data access control and so eliminating access conflicts and enhancing memory concurrency. In this work, the memory is partitioned as shown in Fig. 3. We use a 3×2 PE array (six colors) as an example to illustrate the proposed memory partitioning. Zero values are represented by rectangles with dotted lines. The input matrix is statically partitioned in rows into chunks and columns in an interleaved manner. Banked vector together with banked matrix are divided by the number of COLs (of the PE array), and the output vector and output matrix are partitioned in rows. The elements of the input matrix and banked vector/matrix are accessed and consumed by PEs with the same color. As is evident in Fig. 3, each PE is assigned with 6 elements from 2 rows of the input matrix and 3 elements from the banked vector. Each element of the banked vector is shared and can be accessed by PEs at the same COL (which is why each element block has 3 colors). In this work, we use Compressed Sparse Row (CSR) format to store matrices.

Although there are other partitioning methods, such as column-wise partitioning and interleaved partitioning of matrix A, the following reasons make them unusable. First, it is possible to partition the first input matrix into either rows or columns. However, the sequential nature of hybrid Gauss-Seidel smoothing mandates the use of row partitioning in *pseudo* SpMV. Second, during the restriction/interpolation construction phase, matrix A should be partitioned in rows when evaluating the maximum in Eq. 2.

C. Data Flow

Before describing data flow, assume that *ROW* (*COL*) denotes the row (column) of the PE array to distinguish it from those of the input matrix. There is global synchronization between the PEs, both within islands and between islands, across each phase and step described below.

1) Restriction/Interpolation Construction: This phase is accomplished in three steps as elaborated below.

First, in **Calc_StrengthMat** step, the maximum negative element (minimum) of each row in matrix A is determined (Eq. 2). Elements of matrix A are fetched from the input matrix buffer to the PEs. The smaller element of each *COL* and previous *COL*, which are compared in PEs, moves forward

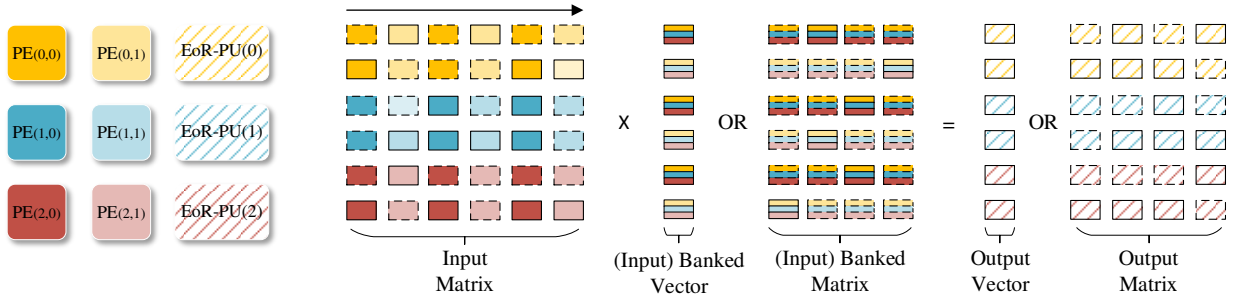

Fig. 3. Memory Partitioning in FP-AMG

horizontally until it reaches the EoR-PU, where the minimum value will be held. After visiting all of elements in the row, the minimum value in EoR-PU moves backward to all PEs in the same *ROW*. Before proceeding to the next row of the matrix, each ROW builds a *neighbor list*, *strength count*, and *weakly connected* buffers. This process is repeated for the entire chunk of data. Also, there is no need to implement **Build_Graph** function as it is done by storing the number of neighbors in *strength count* buffer while creating *neighbor list* buffer.

Second in **coarsening** step, each EoR-PU with the aid of *neighbor list* and *strength count* determines whether the corresponding grid point is a C-point or F-point. This can be accomplished by comparing *strength count* buffer and updating the *C/F/not-visited* buffer. At first, all of the entries of the *C/F/not-visited* buffer are filled with *not-visited* and the process of coarsening algorithm finishes whenever none of the entries is in *not-visited* state.

Finally, in **interpolation** step, PEs in all *COL*s accumulate matrix *A* elements if they are weakly connected points for the current grid point. In case they are not being weakly connected, the column index and value is stored (only for the current grid point) in *fine list* and *coarse list* if it is a F-point and C-point, respectively. Then the sum of the result of accumulation in *COL*s is computed while it is moved forward horizontally and the denominator in Eq. 3 is stored in the *denom* register in EoR-PU. Again, before proceeding to the next row of matrix *A*, for all of the valid elements of *coarse list*, the numerator of Eq. 3 is computed and the result of matrix *P* element is written to the output matrix buffer.

2) Triple Matrix Multiplication: In this phase (also called *RAP*), first $Z = A \times P$ and then $R \times Z$ is computed. In either case, the elements in input matrix buffers are stationary until all the corresponding banked matrix elements are fetched. For each pair of elements the product is computed in the PE and move to the next *COL* where the column indexes are compared. If they are equal the partial sum is added to the product of two input matrix elements in the new PE. The partial sum is moved forward horizontally until the result is written to the output matrix buffer by EoR-PU.

3) Smoothing: As shown in Eq. 5, most of the computation in smoothing relies on the SpMV kernel. In SpMV, matrix *A* elements are fetched one by one from the input matrix buffer. The corresponding banked vector, according to the column index of matrix *A* element, is fetched, multiplied with the value of *A* elements, and accumulated. Finally, after computing all partial sums, they are added together as they move forward until they reach EoR-PU where final addition to *RHS* and the *diag* register (a_{ii} in Eq. 3) and division takes place.

4) Residual Computation: This procedure looks like the smoothing except the post-processing step (Eq. 4).

5) Restriction/Interpolation: This procedure is a pure SpMV kernel, which is discussed in previous subsections.

D. Optimizations

1) Optimization 1: In AMG, it often happens that the size of a buffer is not known in advance [7]. More importantly, the storage demand varies significantly in different rows of the matrix. Assuming a fixed size for each row may lead to wasted resources. Instead, we reduce the needed memory by using a dynamic buffer.

2) Optimization 2: Instead of storing a large matrix *S* of size $n \times n$ we store three matrices of much smaller size: a *neighbor list* buffer, a *strength count* buffer, and a *weakly connected* flag. *Strength count* (*neighbor list*) can be reused multiple times by storing entirely (a memory tile) in on-chip memory. Next subsection provides a definition for memory tile.

3) Optimization 3: During restriction/interpolation construction, the denominator of Eq. 3 is computed once and will be reused multiple times when constructing the *P* matrix. Moreover, the value and column index of C- and F-points for the current grid point are stored into *coarse list* and *fine list* when computing the denominator and will be read later when computing the numerator of Eq. 3.

E. FPGA Mapping Methodology

Having an array-like architecture in FP-AMG greatly simplifies the complicated task of mapping to an FPGA with constrained resources. There are, however, several design parameters that still need to be determined.

In real applications, since matrices (*i.e. A, R*, and *P*) can be quite large, they are stored in off-chip DRAM. But because these sizes vary significantly between levels, there is also a possibility of storing *entire* matrices at higher levels (having smaller grids). We refer to the *threshold level* as the first

level where matrices are stored on-chip. This on-chip storage (denoted as a *memory tile*) also applies to the *neighbor list* and *weakly connected* buffers due to their large memory requirements.

Other design parameters are the size of the PE array and the number of islands. Let *#ROW, #COL* denote the number of rows and columns within each island and *n* denote the number of islands. In this work, we assume *#ROW* is 32. *#COL* should be small enough so that PEs do not become idle due to the small average number of non-zeros per matrix row. It should be noted that, as *n* increases, more on-chip memory is needed due to replication. Although having more islands does not improve the performance if we are in the memory bound phase (the first few levels of the solve phase), it does boost the performance otherwise (rest of the solve phases and the setup phase).

There are three constraints within this framework:

- **Constraint1**: Bounded by the amount of parallelism ($n \times \#ROW$) that the smoothing step offers.
- **Constraint 2**: Bounded by the on-chip memory as a result of a high threshold level or a large *n*.
- **Constraint 3**: Bounded by LUTs, FFs, or DSP blocks as a result of a there being a large number of processing units (PE or EoR-PU).

In memory-bound applications it is important to measure how deep this bound is. We define the memory bound ratio:

$$MBR = \frac{\#ROW \times \#COL \times n \times numBytes \times f}{bandwidth} \quad (7)$$

where *numBytes* is the number of bytes that each PE requires from off-chip memory and f is the operating frequency of FPGA device. If *MBR* is large enough, it is better to invest more on-chip memory to store matrices in levels with larger grid (lower levels) rather than increasing *n*.

Fig. 4 shows the design parameter tuning used in this framework to find *n* and threshold level. We first start with setting *n* and threshold level to one and the last level, respectively. Then, *MBR* determines the order of tuning *n* and threshold level. In this work, *k* is set to 2.

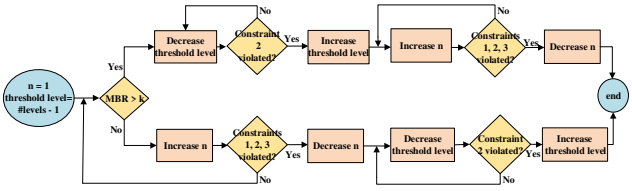

Fig. 4. Design Parameter Tuning

IV. Experimental Evaluation

A. Experimental Setup

Since AMG is mostly memory bound we employ two kinds of FPGA boards, one without HBM (Xilinx VCU118) and another with HBM (Xilinx Alveo U280). The maximum memory bandwidth of the VCU118 is 4×19.2 GB/s made up of four memory channels each connected to a DRAM DDR4-2400. The Alveo U280 is equipped with HBM2 with maximum total bandwidth of 460 GB/s. The design is coded in *Verilog HDL* and the utilization results are reported by *Vivado Design Suite 2018.3*. The operating frequency of the current implementations is 250 MHz.

To evaluate FP-AMG, we compare it with a well-known CPU-based solver, BoomerAMG, which is based on the *HYPRE* library. Evaluation is based on the open-source code [21] by modifying it to run AMG standalone (rather than as a preconditioner). We run the solver on an *Intel Xeon E5-2680-V4* (14 cores) for six sparse matrix benchmarks (with level 2 optimization). Table I summarizes the specifications of the benchmarks; *lap3d* is from [21] while the others are from the SuiteSparse Matrix Collection [22]. To further show the efficacy of FP-AMG, we compare its performance with AmgX running on an Nvidia Tesla P100 (with CUDA 10.1) based on the open-source code [23]. The numerical precision for CPU, GPU, and FP-AMG implementations is based on double-precision and single-precision floating point, and mixed single-precision floating point with customized-precision integer arithmetic, respectively.

TABLE I
Benchmark Specifications

benchmark	#rows	#non-zeros/row
apache2	715,176	7
atmosmodd	1,270,432	7
ecology2	999,999	5
G3_circuit	1,585,478	5
lap3d	1,000,000	27
parabolic_fem	525,825	7

B. Resource Utilization

The resource requirements for a single processing unit (PE or EoR-PU) in FP-AMG (except BRAM) is given in Table II. The results are after implementation on a VCU118 under timing constraint of 4 *ns*. While EoR-PU requires more LUTs and FFs, the resources can be amortized over a row since there is only one EoR-PU per ROW.

After obtaining the resource requirements for PE and EoR-PU, we apply the proposed design parameter tuning to find *n* and threshold level. Table III shows these parameters together with #COL in the considered benchmarks for both FPGAs. According to Table I, since the number of non-zeros per row of matrix *A* in *lap3d* is much higher than that of other benchmarks #COL is set to 8 while it is considered as 4 for other benchmarks. It is evident from Table III that HBM-enabled FPGA (Alveo U280) typically has higher *n* and threshold level as a result of higher MBR provided by increased memory bandwidth.

TABLE II
Resource Requirement for PE and EoR-PU

Component	LUT	FF	DSP
PE	385	638	4
EoR-PU	1080	1776	2

TABLE III
DESIGN PARAMETER TUNING RESULTS IN SIX BENCHMARKS FOR VCU118 AND ALVEO U280

Benchmark	HBM?	# levels	threshold level	n	# COLs
apache2	—	8	2	1	4
	HBM		3	4	
atmosmodd	—	8	3	2	4
	HBM		5	3	
ecology2	—	8	2	2	4
	HBM		3	3	
G3_circuit	—	9	3	1	4
	HBM		4	2	
lap3d	—	6	1	3	8
	HBM		1	3	
parabolic_fem	—	8	1	4	4
	HBM		1	4	

Fig. 5 gives the breakdown utilization of different components in FP-AMG for both FPGAs under six benchmarks after design parameter tuning. In this figure, utilities denote C/F/Not-visited buffer, memory controller, PE array controller, and other miscellaneous components. In most of the benchmarks, PE contributes LUT and FF utilization more than other components (EoR-PU, bank resolver, and utilities). Matrices *A, R, P*, and *Z* as well as vector *x* are stored in UltraRAM (URAM) due to their relatively large size. Thus, URAM utilization is comprised of input matrix, banked matrix, and banked vector buffers. In contrast, *neighbor list*, *weakly connected*, *strength count*, *C/F/Not-Visited* buffers are stored in block RAM (BRAM).

The left chart in Fig. 5 (Xilinx VCU118) shows that utilization is limited by URAM since AMG is mostly memory-bound. LUT, FF, and DSP utilization depend on *n* and $\#COL$. For instance, *lap3d* has the highest DSP utilization since it has the highest product of *n* and $\#COL$. However, the size of required memory for different levels as well threshold level determines the BRAM and URAM utilization. Note that the utilization breakdown for *lap3d* is different from the other benchmarks. This is because the *strength count* buffer is stored in all levels while the *neighbor list* buffer is stored only from the threshold level. Since the size of matrix *A* varies drastically between the first level and threshold level in *lap3d*, *strength count* utilization is higher than that of *neighbor list*.

As it is depicted in the right chart of Fig. 5 (Xilinx Alveo U280), the overall utilization is higher due to larger *n* (as a result of higher bandwidth). Also, URAM utilization is mostly dominated by the banked vector since the threshold level in this FPGA is larger than that of VCU118 (non-HBM). A higher threshold level requires less memory storage for matrices and other buffers because the size of the problem drops off drastically as the level increases. But, vector *x* must still be stored in banked vector independent of the value of threshold level. Also, *atmosmodd* benchmark has the smallest BRAM utilization with respect to other benchmarks since the threshold level is larger than the others.

C. Performance

Fig. 6 shows the execution time of running AMG on the baseline CPU, a GPU, an FPGA without HBM (VCU118), and an FPGA with HBM (Alveo U280). In this work, three modes are considered for the CPU baseline: CPU-1 (1 thread), CPU-14 (14 threads), and CPU-28 (28 threads), all of them are utilizing one node. Execution time for the CPU-1 mode is normalized to 100 to facilitate comparison. In most of the cases (except *G3_circuit*) CPU-14 mode shows slightly better performance than CPU-28 mode. The underlying reason is that hyperthreading has no benefit here due to resource constraints and also because the number of iterations it takes for AMG to converge increases, which slows down the solve phase. Although a large fraction of time is devoted to the solve phase (green, light, and dark blue colors), it is also known that the setup time is one of bottlenecks in multi-node implementations and contributes a larger fraction of time [7].

Overall the performance of FP-AMG is on average 6.6× (2.2×) and 2.5× (0.8×) compared to that of the best CPU (GPU) implementation for FPGAs with and without HBM support, respectively. Generally the benchmarks that exhibit better performance on the FPGAs have 1) small threshold level, 2) small number of non-zero elements per matrix row (spending less time in memory-bound region), and 3) large *n*. Hence, G3_circuit, due to its small amount of parallelism (*n*), and lap3d, due to a large number of non-zero elements per row, have low performance in both FPGA configurations.

V. RELATED WORK

There has been a substantial effort on creating high performance implementations of AMG. One of the earliest [5] presents *BoomerAMG*, an AMG solver for distributed-memory architectures. This is extended in [6] to support OpenMP for multi-core architectures. J. Park, *et al*. [7] develop and analyze a set of optimizations for AMG targeting modern x86 multi-core processors. In [8] a library called AmgX is proposed for AMG acceleration in multi-GPUs. Although there is a work on FPGA-based multigrid acceleration [24], [25] for molecular dynamic simulations [26], [27], there is no existing work on accelerating AMG on FPGA as far as we are aware.

In FP-AMG, a number of kernels rely on the computation of SpMV and SpGEMM. Thus, it is essential to review some of that prior work. In [28] an FPGA optimized SpMV kernel is proposed which utilizes a banked vector together with a customized encoding. By using a banked vector replication of buffers is eliminated. However, a *without replication* scheme limits their scalability to only 32. Currently, the increase in on-chip memory and the availability of HBM allows more processing elements to work in parallel. The SpMV kernel in our work is based on [28] but generalized to offer more scalability for this current generation of FPGAs. The authors in [29] propose an SpMV architecture relying on the CSC format. Although they could eliminate the need to store the input vector in on-chip memory, the output vector must be stored in on-chip memory instead. In [30] a design space exploration for SpGEMM on FPGAs is proposed to study performance, energy-delay, and power-delay product. Their data partitioning differs from ours as they partition the second input matrix into columns, while all PEs can have access to the entire of the first matrix; this is not appropriate for FP-AMG.

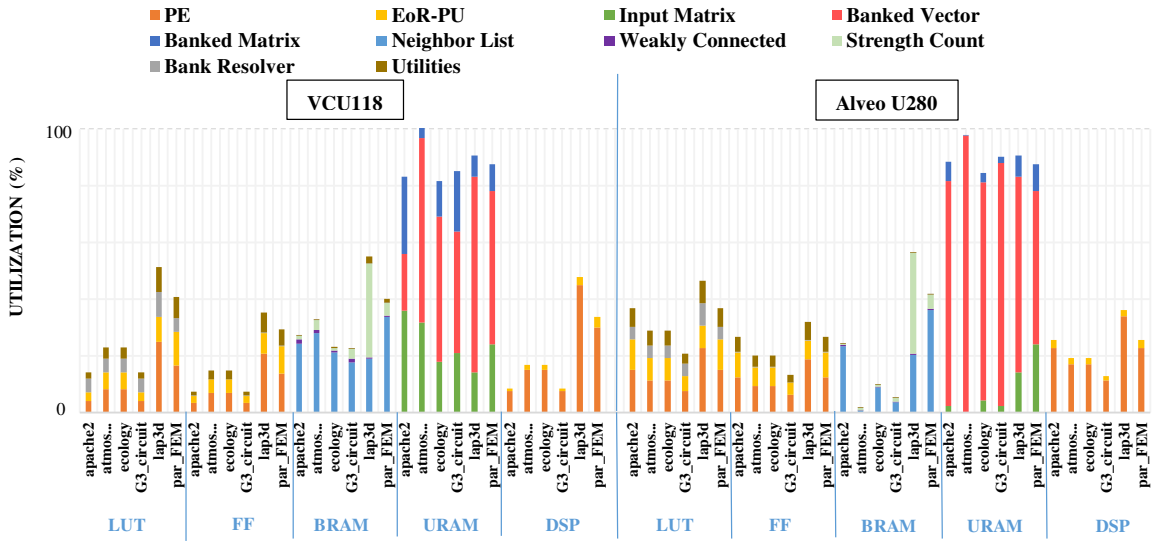

Fig. 5. Resource Utilization under Six Benchmarks for Xilinx VCU118 and Xilinx Alveo U280

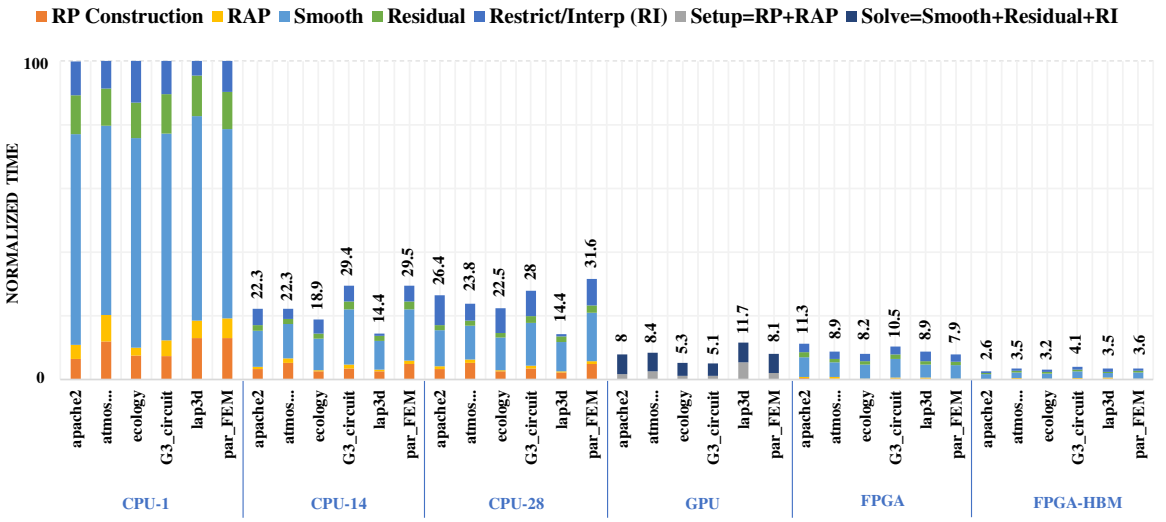

Fig. 6. Performance Comparison of *HYPRE* CPU Implementation, AmgX GPU Implementation, and FP-AMG for Six Benchmarks

VI. Conclusion

In this work, we present a reconfigurable framework to accelerate AMG solvers. In this framework, we propose a scalable architecture that can be used successively by various AMG kernels. To address irregular memory access and reduce memory storage, a smart memory subsystem together with a number of optimizations are proposed. Given the dynamic memory demand in AMG, a methodology for design parameter tuning is developed to map FP-AMG to different FPGAs efficiently. We evaluate the efficiency of FP-AMG by running six sparse matrix benchmarks on two FPGAs: one with and one without HBM. The experimental results reveal that FP-AMG can provide on average 2.5× and 6.6× speedup compared to a server-class Intel Xeon CPU for an FPGA without and with HBM, respectively. Also, the HBM-enabled FPGA could outperform the GPU implementation by a factor of 2.2×.

FP-AMG demonstrates promising performance improvement on a single FPGA board. However, the emergence of AMG in extreme scale systems leads us to consider acceleration on a cluster of FPGAs. CPU clusters can have comparatively more difficulty with strong scaling, especially in 3D problems. Large communication overhead in levels with coarse (small) grids can drastically limit AMG performance. We would like to address this bottleneck in a cluster-based FPGA environment that benefits from direct FPGA to FPGA communication. Also, we would like to expand our framework to embrace a wider range of solvers and smoother.

Acknowledgments

This work was supported, in part, by the NSF through awards CCF-1618303 and CCF-1919130; the NIH through awards 1R41GM128533 and R44GM128533; and by a grant from Red Hat.

REFERENCES

[1] K. Stüben, "A review of algebraic multigrid," *Journal of Computational and Applied Mathematics*, vol. 128, no. 1, pp. 281 – 309, 2001, numerical Analysis 2000. Vol. VII: Partial Differential Equations.

[2] R. D. Falgout, "An introduction to algebraic multigrid," *Computing in Science Engineering*, vol. 8, no. 6, pp. 24–33, Nov 2006.

[3] J. W. Ruge and K. Stüben, *Algebraic Multigrid*, pp. 73–130. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611971057.ch4

[4] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial, Second Edition*, 2nd ed. Society for Industrial and Applied Mathematics, 2000. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898719505

[5] V. E. Henson and U. M. Yang, "BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner," *Appl. Numer. Math.*, vol. 41, no. 1, pp. 155–177, Apr. 2002.

[6] A. H. Baker, T. Gamblin, M. Schulz, and U. M. Yang, "Challenges of scaling algebraic multigrid across modern multicore architectures," in *2011 IEEE International Parallel Distributed Processing Symposium*, May 2011, pp. 275–286.

[7] J. Park, M. Smelyanskiy, U. M. Yang, D. Mudigere, and P. Dubey, "High-performance algebraic multigrid solver optimized for multi-core based distributed parallel systems," *International Conference for High Performance Computing, Networking, Storage and Analysis*, vol. 15-20-Nove, pp. 1–12, 2015.

[8] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka, "AMGX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods," *SIAM Journal on Scientific Computing*, vol. 37, no. 5, pp. S602–S626, 2015.

[9] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp, "Modeling the performance of an algebraic multigrid cycle on hpc platforms," in *Proceedings of the International Conference on Supercomputing*, 2011, pp. 172–181.

[10] W. Mitchell, R. Strzodka, R. D. Falgout, and S. F. McCormick, "Parallel performance of algebraic multigrid domain decomposition (AMG-DD)," *CoRR*, vol. abs/1906.10575, 2019.

[11] S. Liu, C. Eisenbeis, and J. Gaudiot, "Speculative execution on gpu: An exploratory study," in *International Conference on Parallel Processing*, 2010, pp. 453–461.

[12] Q. Xiong, A. Skjellum, and M. Herbordt, "Accelerating MPI Message Matching Through FPGA Offload," in *Proc. IEEE Conf. on Field Programmable Logic and Applications*, 2018.

[13] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Xuy, R. Patel, and M. Herbordt, "FPDeep: Acceleration and Load Balancing of CNN Training on FPGA Clusters," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2018.

[14] T. Wang, T. Geng, X. Jin, and M. Herbordt, "FP-AMR: A Reconfigurable Fabric Framework for Block-Structured Adaptive Mesh Refinement Applications," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2019.

[15] ——, "Accelerating AP3M-Based Computational Astrophysics Simulations with Reconfigurable Clusters," in *Proc. Int. Conf. on Application Specific Systems, Architectures, and Processors*, 2019.

[16] A. Putnam, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," in *Proc. Int. Symp. on Computer Architecture*, 2014, pp. 13–24.

[17] J. Sheng, C. Yang, and M. Herbordt, "Towards Low-Latency Communication on FPGA Clusters with 3D FFT Case Study," in *Proc. International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, 2015.

[18] A. George, M. Herbordt, H. Lam, A. Lawande, J. Sheng, and C. Yang, "Novo-G#: A Community Resource for Exploring Large-Scale Reconfigurable Computing Through Direct and Programmable Interconnects," in *IEEE High Perf. Extreme Computing Conf.*, 2016.

[19] J. Sheng, C. Yang, A. Caulfield, M. Papamichael, and M. Herbordt, "HPC on FPGA Clouds: 3D FFTs and Implications for Molecular Dynamics," in *Proc. IEEE Conf. on Field Programmable Logic and Applications*, 2017.

[20] H. De Sterck, U. M. Yang, and J. J. Heys, "Reducing complexity in parallel algebraic multigrid preconditioners," *SIAM Journal on Matrix Analysis and Applications*, vol. 27, no. 4, pp. 1019–1039, 2006. [Online]. Available: https://doi.org/10.1137/040615729

[21] "Algebraic multigrid benchmark," https://github.com/LLNL/AMG.

[22] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: https://sparse.tamu.edu/

[23] "Distributed multigrid linear solver library on gpu," https://github.com/NVIDIA/AMGX.

[24] Y. Gu and M. Herbordt, "FPGA-based multigrid computations for molecular dynamics simulations," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2007, pp. 117–126.

[25] T. VanCourt and M. Herbordt, "Application-dependent memory interleaving enables high performance in FPGA-based grid computations," in *Proc. IEEE Conf. on Field Programmable Logic and Applications*, 2006, pp. 395–401.

[26] C. Yang, T. Geng, T. Wang, J. Sheng, C. Lin, V. Sachdeva, W. Sherman, and M. Herbordt, "Molecular Dynamics Range-Limited Force Evaluation Optimized for FPGA," in *Proc. Int. Conf. on Application Specific Systems, Architectures, and Processors*, 2019.

[27] C. Yang, T. Geng, T. Wang, R. Patel, Q. Xiong, A. Sanaullah, C. Lin, V. Sachdeva, W. Sherman, and M. Herbordt, "Fully Integrated FPGA Molecular Dynamics Simulations," in *Proc. ACM/IEEE Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, 2019.

[28] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication, year=2014," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, May, pp. 36–43.

[29] R. Dorrance, F. Ren, and D. Markoviundefined, "A Scalable Sparse Matrix-Vector Multiplication Kernel for Energy-Efficient Sparse-Blas on FPGAs," in *Proceedings International Symposium on Field-Programmable Gate Arrays*, 2014, p. 161–170.

[30] C. Y. Lin, Z. Zhang, N. Wong, and H. K. So, "Design space exploration for sparse matrix-matrix multiplication on FPGAs," in *2010 International Conference on Field-Programmable Technology*, Dec 2010, pp. 369–372.