haHyperEF: hardware accelerated HyperEF

Author

Gaurav Dubey

CPE800 Master's Project Research Course

Stevens Institute of Technology, Department of Electrical and Computer Engineering

May 8, 2023

I pledge to adhere to the Stevens Graduate Student Code of Academic Integrity.

[illegible]

(Gaurav Dubey)

haHyperEF: hardware accelerated HyperEF

Gaurav Dubey
Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ, USA
gdubey@stevens.edu

Zhuo Feng
Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ, USA
zfeng12@stevens.edu

***Abstract*—Graph coarsening techniques are tools that break down complex hypergraphs into simpler ones. Such techniques are employed because, unlike processing a simple hypergraph, processing complex hypergraphs engages computing resources for a longer time. HyperEF [1] reduces the size and complexity of a hypergraph while retaining most of the original hypergraph information and provides improved partitioning of hypergraph while achieving 70x speed up [1]. However, HyperEF employs memory-bound operators such as sparse matrix-vector multiplication (SpMV), and the processing time of memory-bound operations on general-purpose machines is lengthy [2] [3] [4]. In this work, we identify Filter as the primary computationally intensive kernel of HyperEF and propose haHyperEF, a hardware accelerator intending to boost the execution of Filter on FPGA equipped with high-bandwidth memory (HBM). We present the architecture of haHyperEF and its implementation with results that show a 2x speed up of Filter for hypergraphs with hyperedge not exceeding 60,000.**

***Index Terms*—FPGA Acceleration, hardware acceleration, hypergraph coarsening, HLS, Sparse Matrix Algebra**

I. Introduction

Graphs are ubiquitous and widely used to represent information and relationships between entities of a network structure. Deep learning models relying on graphs effectively process and predict information in various areas, assisting humanity in tackling challenging problems [1]. As more and more solutions to real-world issues rely on machine learning and artificial intelligence, graphs and graph learning are becoming more relevant.

Data scientists now use Hypergraphs to model complex problems comprising many connected nodes, as it is a more generalized graph. More affordable memory allows the storage of large data sets, helping data scientists to represent problems using hypergraphs. Learning such a hypergraph is becoming more demanding as network structure can be too big for current available compute resources. Gigantic hypergraphs inevitably force the use of graph techniques that could reduce the size of these networks while maintaining most of the information detailed in the original hypergraphs [1].

HyperEF is a graph framework technique that can decompose a large hypergraph into a hypergraph with fewer inter-cluster hyperedges. HyperEF significantly reduces the computation load on the graph learning processing unit, offering more time for reactive actions. HyperEF technique requires multiple sparse matrix linear algebra operators; these operators demand random accesses to memory, which results in a large number of cache misses which increases compute time [3] [4]. Although HyperEF can significantly reduce graph learning computations, running this algorithm on a generic computer will add latency and power consumption.

General-purpose computers may have 16-24 compute units but conventionally offer limited memory bandwidth depending on the number of memory controllers available to the processor. Traditional computers typically have access to two to four memory controllers, and the latest high-performance graphics processing units are equipped with up to 7 dedicated memory controllers [5] [6]. Performance boost due to the parallelism on common compute platforms of algorithms like HyperEF is limited and depends on the memory bandwidth as it requires high memory bandwidth and random memory accesses.

HyperEF is an emerging graph algorithm that enables the creation of a simple, smaller hypergraph without losing the key spectral features of the original hypergraph. Hardware that accelerates this graph technique would only promote its usage, increasing user-group interested in this technique, as an accelerator is expected to make this highly-efficient technique faster. Exploring this technique on FPGA could pave the way for increasing understanding of graph learning algorithms and efficient graph computation on FPGAs.

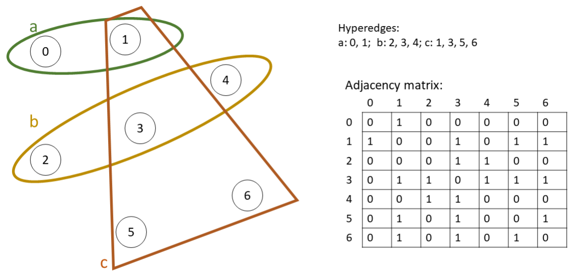
Accelerating HyperEF on an FPGA, on the other hand, is a challenging task. First, sparse matrix formats are commonly used to represent hypergraphs, and SpMV operation requiring access to non-contiguous memory locations makes saturating the bandwidth of DDR memories difficult [3]. Second, large hypergraphs can be dense, making multiple matrices challenging to fit in system memory; some applications require multiple matrices to be stored so that processing can be seamless without continuous write operations to update inputs. Finally, while writing hardware behavior with HLS tools is less complicated than VHDL/Verilog, it is still tricky because one may spend significant time debugging unexpected low-level implementation problems, such as hardware-software behavior mismatch, which slows accelerator development [7].

We suggest haHyperEF, a hardware accelerator specifically designed to accelerate HyperEF on FPGA equipped with high-bandwidth memory (HBM) [8]. haHyperEF is the first attempt to accelerate HyperEF on an FPGA. haHyperEF accelerated kernels can be called in an OpenCL host application, giving users the flexibility to tailor the application to their needs. haHyperEF incorporates a proven memory-efficient spare matrix-vector multiplication accelerator to saturate memory bandwidth. haHyperEF is designed to handle large hypergraphs with provisions to accept hypergraphs represented in popular sparse matrix formats. Results of haHyperEF are precise but may slightly differ from the original implementation of HyperEF in Julia [9].

The rest of the report is organized as follows. Section I-A1 provides a background of popular sparse matrix representations, and Section I-A2 provides an overview of SpMV operation. Section II defines the research problem this work is trying to address. Section III present our analysis of the problem, and hardware architecture of haHyperEF. Section IV presents analysis, describes experimental setup, the test plan, results, and limitations of our work. Section V summaries our work.

A. Background

1) Sparse Matrix Representations: Hypergraphs are commonly used to represent networks, system-component interactions, and relationships. Learning hypergraph plays an essential role in making predictions for solving practical problems using machine learning algorithms. Machine learning algorithms generally require hypergraphs to be represented as adjacency matrices.

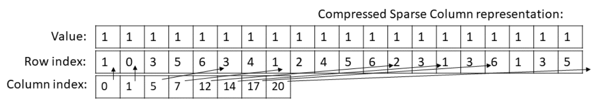


	0	1	2	3	4	5	6
0	0	1	0	0	0	0	0
1	1	0	0	1	0	1	1
2	0	0	0	1	1	0	0
3	0	1	1	0	1	1	1
4	0	0	1	1	0	0	0
5	0	1	0	1	0	0	1
6	0	1	0	1	0	1	0

Fig. 1. Example of a hypergraph representation as an adjacency and sparse matrix.

Several practical hypergraphs feature adjacency matrices that are primarily composed of zero values. And commonly, hypergraphs are represented using common sparse matrix formats. There are several popular sparse matrix representations, but we will focus on Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats for our work. In CSR format, the sparse matrix is stored in three 1D arrays. Two 1D arrays are of length nnz, where nnz is a number of non-zero elements; these two arrays contain non-zero values and column index. Values and column index are stored by reading value first by row. The third array is of length m+1, where m is the matrix's number of rows, representing each row's extent. CSC format is similar to CSR except that column indexes are compressed instead of rows (the length of this compressed array is n+1, where n is the number of columns of the matrix), and values are read by row. haHyperEF can input hypergraphs represented in other formats by converting representation to CSR or CSC.

Figure 1 depicts a hypergraph on the left and its representation as an adjacency matrix (7x7 elements) on the right. The adjacency matrix of Fig. 1 can be represented as a Compressed Sparse Column (CSC) format as depicted in Fig. 2. Observe that CSC representation requires less memory (only 20 non-zero elements) to represent the same 49 elements of the original adjacency matrix, and the length of the compressed column array is 8 (7+1).



Value:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Row index:	1	0	3	5	6	3	4	1	2	4	5	6	2	3	1	3	6	1	3	5
Column index:	0	1	5	7	12	14	17	20												

Fig. 2. Hypergraph representation as a sparse matrix.

2) Sparse Matrix-Vector Multiplication (SpMV): SpMV is one the most important operators of the sparse matrix linear algebra. This operator takes a sparse matrix and multiplies it with an input vector, algorithm 1 [4] presents the pseudo-code of SpMV operation for a CSR matrix. In the pseudo-code, nnz is the number of non-zero values, m is the number of rows, and n is the number of matrix columns. 1D arrays val, col, and idptr contain values, column indexes, and row extent of the sparse matrix, and V is the input vector. Values of the sparse matrix are read by rows, top to bottom, left to right. Variables col_start and col_end are used to establish the extent of the row while traversing all non-zero values in a row. Variable idx points to the vector element of V to be used for multiplication in an iteration.

Algorithm 1 SpMV()

```
1:  procedure SpMV($val[nnz], col[nnz], idptr[m+1], V$)
2:      $Vn[1:n] \leftarrow 0$
3:      for $i$ = 1 to $m$ do
4:          $col\_start \leftarrow idptr[i]$
5:          $col\_end \leftarrow idptr[i+1]$
6:          $sum \leftarrow 0$
7:          for $j$ = $col\_start$ to $col\_end$ do
8:              $idx \leftarrow col[j]$
9:              $sum \leftarrow sum + A[j] * V[idx]$
10:         end for
11:         $Vn[i] \leftarrow sum$
12:     end for
13:     return $Vn$
14: end procedure
```

II. Problem Statement

HyperEF requires multiple mathematical operations, and the execution time of such processes may significantly increase power consumption and latency. These operations also increase the overall computing time of graph learning running on a general-purpose computer. Dedicated application-specific hardware like FPGA is generally used to accelerate algorithms to shorten execution time and reduce latency and power utilization so that compute units can take meaningful actions in real time, consuming less power. For example, the Tensor processor was developed by Google to accelerate convolution and activation functions heavily used for machine learning applications. HyperEF is one cog in a wheel of graph learning framework, and accelerating this algorithm using FPGA is highly desirable.

This work intends to develop a computer architecture that exploits concurrency in the HyperEF. The HyperEF algorithm computations will have to be divided into a part tasked to run on the host processor sequentially and a concurrent part, which executes in parallel to offer more excellent performance in terms of power, area, and execution time compared to the algorithm running on a general-purpose computer. HyperEF requires multiple matrix-vector multiplication operations per iteration, making it memory bound. Hence, unconventional hardware solutions must be considered to saturate memory bandwidth to better the performance of HyperEF on FPGA.

III. haHyperEF

The first step to accelerating any algorithm on hardware is to partition the algorithm into two sections, i.e., software and hardware parts. The parallelizable part of the algorithm can be accelerated on hardware, as FPGAs offers parallelism. However, FPGA accelerated kernels operate at a lower clock frequency than a general-purpose computer, most of the FPGA kernels can only run at 200MHz (Modern computers operate at 3-4 GHz). It is best to run the non-parallelizable part on software. The non-parallelizable part can be set to run on the host machine and mostly performs less compute-intensive tasks like configuration, pre-processing data, or post-processing of data. We aimed to accelerate routines that cannot be accelerated on a traditional machine by code modifications or software platform changes. FPGA can only outperform generic computers if FPGA kernel implementation has a higher degree of parallelism than equivalent implementation on a traditional computer.

Algorithm 2

Filter() [1] [2]

```
procedure FILTER($D, AD, k, ...$)
    $sm \leftarrow FilterInit\_rv()$                ▷ O($N$)
    $sm\_vec \leftarrow null$                       ▷ Empty set. O(1)
    for $i$ = 1 to $k$ do
        $sm \leftarrow D * sm$         ▷ Matrix-vector product O($N^2$)
        $sm \leftarrow AD * sm$        ▷ Matrix-vector product O($N^2$)
        $sm \leftarrow D * sm$         ▷ Matrix-vector product O($N^2$)
        $sm\_vec \leftarrow FilterCp(sm)$            ▷ O($N$)
    end for
    $V \leftarrow FilterWr(sm\_vec)$                 ▷ O($N$)
    return $V$                  ▷ a set of smoothed vectors O(1)
end procedure
```

After analyzing the run-time complexity of the HyperEF, we identified Filter() with the worst-case complexity among all the other kernels of HyperEF. The pseudo-code of the Filter() kernel is presented in Algorithm 2 [1]). As shown in Algorithm 2 [1]), Filter() has a worst-case complexity of O(N^3) as the Matrix-vector multiplication operator is invoked for k times in a loop. Filter() requires sparse linear algebra operators like matrix-vector multiplication, and the worst-case complexity of these operations is O(N^2) (refer Algorithm 1 [1]). Filter() was found to be the perfect candidate for hardware acceleration with the worst-case complexity of O(N^3).

TABLE I
COMPARISON OF HYPEREF AND ITS PRIMARY KERNEL (FILTER) EXECUTION TIME.[a]

dataset	Computation time Filter (ms)	Computation time HyperEF	Percentage of compute time taken by Filter (%)
ibm01	190.99	220.94	86.44
ibm02	288.80	314.98	91.68
ibm03	357.85	388.67	92.07
ibm04	422.10	444.67	94.92
ibm05	595.05	643.41	92.48

[a]Tests were run on a computer for software development only.

Further, to confirm that Filter() is the primary kernel that takes most of the compute time, we tested to measure the compute times of HyperEF routines. Using standard benchmark tools, we observed that operations of the Filter() operated for about 90% of the total run time of HyperEF. Different hMetis [10] datasets were considered for benchmarking, and the observed execution time of HyperEF and Filter() are tabulated in Table I. Filter() accounts for at least 86% (see Fig. 3) of the total execution time. We also compared the execution time of various subroutines of Filter() and concluded that matrix-vector product is indeed the most time-consuming routine, which requires acceleration in order to boost the performance of HyperEF.

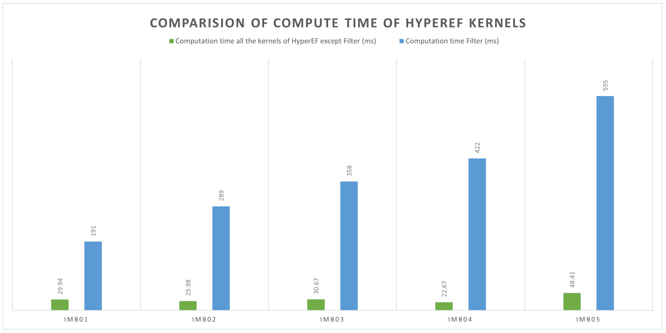


Fig. 3. Comparison of Execution time.[b]

[b]Tests were run on a computer for only software development.

[1]Pseudo code is for worst-case analysis illustration only.
[2]The actual implementation may vary.

A straightforward approach to accelerate matrix-vector product operation would be to use multi-threading on general-purpose computers or to have a parallel processing unit framework in FPGA to accelerate this operation. In fig.4, a parallel processing unit framework for FPGA is shown with memory read schedules. A multi-threaded application on a generic computer will also have a similar memory read schedule. Fig. 4 shows four processing unit's read request schedules for three back-to-back calculations, and the schedules in red text indicate read requests to uncommon memory locations. The memory controller interfaced with the processing units is required to fetch data from five different locations for a single operation of the four threads. A single operation is the product of an element of a matrix with an element of a vector. Common computing platforms are only equipped with a single memory unit, i.e., only one memory controller for all the processing units. Caches are used to tackle the memory bandwidth requirement, but given the nature of the schedule shown in Fig. 4, we will see cache misses. These cache misses will increase read latency and degrade the algorithm's performance.

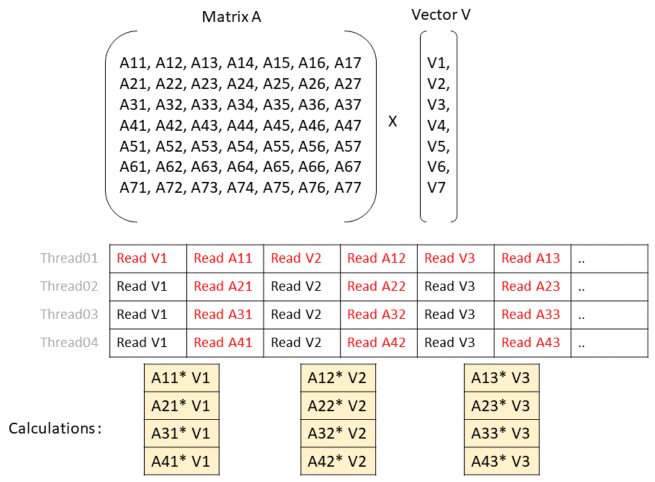

Matrix A
Vector V
A11, A12, A13, A14, A15, A16, A17
A21, A22, A23, A24, A25, A26, A27
A31, A32, A33, A34, A35, A36, A37
A41, A42, A43, A44, A45, A46, A47
A51, A52, A53, A54, A55, A56, A57
A61, A62, A63, A64, A65, A66, A67
A71, A72, A73, A74, A75, A76, A77
x
V1, V2, V3, V4, V5, V6, V7
Thread01 Read V1 Read A11 Read V2 Read A12 Read V3 Read A13 ..
Thread02 Read V1 Read A21 Read V2 Read A22 Read V3 Read A23 ..
Thread03 Read V1 Read A31 Read V2 Read A32 Read V3 Read A33 ..
Thread04 Read V1 Read A41 Read V2 Read A42 Read V3 Read A43 ..
Calculations:
A11* V1 A21* V1 A31* V1 A41* V1
A12* V2 A22* V2 A32* V2 A42* V2
A13* V3 A23* V3 A33* V3 A43* V3

Fig. 4. Memory read request schedule of a multi-thread application for a matrix-vector product operation.

Operations like matrix-vector products cannot be accelerated by simply increasing parallelism. Its performance also depends on the memory bandwidth. Such computational problems are also known as memory-bound functions. To boost the performance of such a computational problem, we also need a very fast memory with a small read and write latency. One way to accelerate memory-bound problems would be to have dedicated memory units per processing unit. With such an arrangement, we have to also ensure that all the information needed for a thread is available in the memory unit interfaced with it. HyperEF is memory-bound due to heavy use of SpMV operations, limited by 77 GB/s bandwidth offered by convectional DDR memories. We opted for the Alveo U280 data center card as a hardware platform to accelerate HyperEF, as it is equipped with HBM (High Bandwidth Memory), which can provide up to 460 GB/s memory bandwidth. This Alveo card offers 8 GB of memory with 32 HBM pseudo channels (each 2 Gb). Each channel can provide 14.375 GB/s of memory bandwidth and an overall bandwidth of 460 GB/s (32 x 14.725 GB/s). So, using this hardware, we can segment the matrix and vector data such that each processing unit has access to a 2 GB memory unit capable of providing 14 GB/s of data throughput.

Filter() requires three back-to-back matrix-vector multiplication operations per iteration. Two SpMV operations required by Filter() depend on the result of the last SpMV operation. Hence, developing three SpMV accelerators to increase HyperEF performance is futile as two SpMV accelerator operations must be stalled until the result of the first SpMV operation is not ready. Once the result of the first SpMV is ready, only the second SpMV accelerator can proceed, while the third should wait for the result of the second SpMV operation to prevent data hazards. In our quest to accelerate HyperEF, we evaluated existing works like ThunderGP [2], Graphlily [3], and HiSparse [4]. In our evaluation we found HiSparse to accelerate SpMV more effectively compared to other works. We repurposed the existing FPGA HLS accelerator HiSparse [4] to shorten the development work.

TABLE II
COMPARISON OF EXECUTION TIME OF FILTER ACCELERATED BY HISPARSE V/S SOFTWARE HYPEREF IMPLEMENTATIONS

dataset	Julia implementation of Filter() [a]	C++ implementation of Filter() [b]	HLS implementation of Filter() [c d]
ibm01	3.912 s	0.808 s	1.691 s
ibm02	3.982 s	1.200 s	2.073 s
ibm03	4.090 s	1.517 s	1.949 s
ibm04	4.212 s	1.732 s	2.357 s
ibm05	4.088 s	1.793 s	2.657 s

[abc]Tests were run on a computer used for only hardware development.
[d]The FPGA accelerator for Filter() operates at clock rate of 200MHz.

HiSparse is specifically designed to accelerate sparse matrix-vector multiplication (SpMV) that can aptly cater requirements of HyperEF. HiSparse can efficiently accelerate SpMV by saturating HBM bandwidth using 20 HBM channels. HiSparse is a newer work based on GraphLily [3] and operates at a frequency of 200 MHz outperforming ThunderGP and Graphlily. In fig. 5 shows the architecture of HiSparse. HiSparse consists of a total of 16 processing engines (PE) that run in parallel to compute matrix-vector products (SpMV). HBM channels 0 to 15 act as dedicated memory modules for each of these 16 processing engines, one per PE. HBM channel 20 is employed to stream vector data for each PE. The result of the product is written to HBM channel 21. Note that any other HBM channel cannot assess the contents of a particular HBM channel. Each HBM channel is a standalone memory module, although Xilinx FPGA provides ways in which a PE can access multiple HBMs by enabling an AXI crossbar switch. However, HiSparse architecture does not have this AXI crossbar enabled.

As discussed earlier, that Filter requires three sequential SpMV operations and SpMV operations can only be started once the last operation is complete. Also, the result of the last SpMV operation is the input vector for the upcoming SpMV operation. While evaluating HiSparse to accelerate the Filter kernel of HyperEF, we had to use host application software to perform device-host-device transfers. Such device-host-device transfers are required to move data from HBM channel 21 to HBM channel 20 for the next SpMV operation once the last SpMV operation is complete. Such transfers incurred significant overhead and required architectural change for a performance boost. Table II shows how HiSparse fared in our evaluation experiments. We created a host application that could run Filter using HiSparse bitstream. This application delivered speedup of Filter, but for larger datasets, the performance deteriorated. As discussed, performance could be further boosted if HBM AXI crossbar could be enabled in HiSparse.

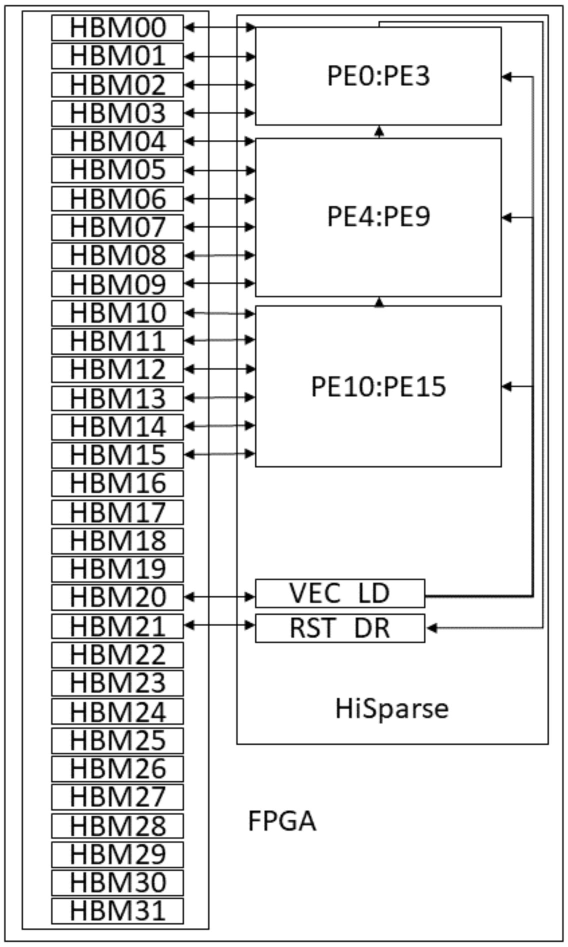


Fig. 5. Architecture of HiSparse.

In fig. 6, we present the hardware architecture of haHyperEF to accelerate Filter. The architecture of haHyperEF is similar to that of HiSparse, except we have added an AXI crossbar, as shown in Fig. 6. We repurposed HiSparse and modified it to add seamless switching between HBM channels. This switching enabled haHyperEF to use the result of the last SpMV operation as an input for the next SpMV operation.

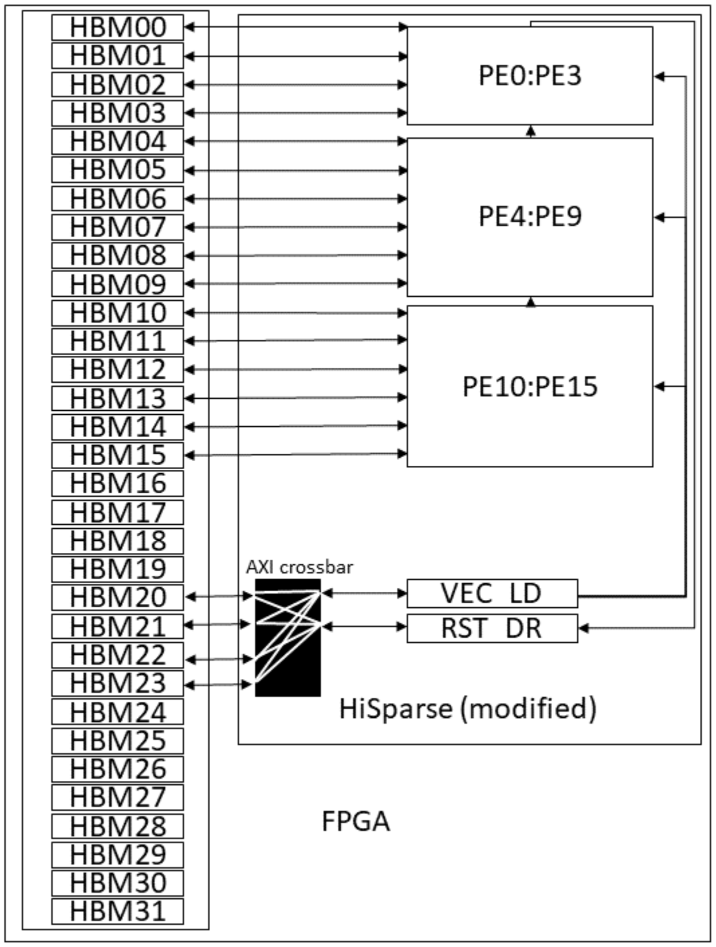


Fig. 6. Architecture of haHyperEF.

We identified other potential subroutines for speedup, like FilterCp, which first demeans a vector and later normalizes it. Also, two SpMV operations involve a diagonal matrix, and a diagonal matrix can be compactly represented as a vector to replace the SpMV operation with the Hadamard product of two vectors. SpMV involving a diagonal matrix is illustrated in Fig. 7. Adding acceleration for Hadamard product and FilterCp()

$$\begin{array}{|c|c|c|c|c|c|}\hline A & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & B & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & C & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & D & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & E & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & F \\ \hline \end{array} \quad X \quad \begin{array}{|c|}\hline a \\ \hline b \\ \hline c \\ \hline d \\ \hline e \\ \hline f \\ \hline \end{array} \quad = \quad \begin{array}{|c|}\hline Axa \\ \hline Bxb \\ \hline Cxc \\ \hline Dxd \\ \hline Exe \\ \hline Fxf \\ \hline \end{array}$$

Fig. 7. SpMV involving a diagonal matrix.

worsened the routing, inhibiting the HLS tool from translating the HLS to HDL. We also evaluated HetroCL [7] to generate

HLS, but that, too, resulted in routing congestion. We may create a separate bitstream which could speedup FilterCp and other post-processing task but further acceleration is beyond the scope of this work.

IV. Numerical Results and Analysis

We now present the evaluation of haHyperEF and compare it's performance against Filter's C++ and Julia implementation in this section.

A. Baseline

The original POC (proof of concept) of HyperEF is implemented in Julia, utilizing in-built Julia libraries that can perform SpMV and other sparse matrix algebra. But, the Xilinx HLS tool only supports C/C++ as high-level languages, which necessitated the implementation of a C++ data structure to translate the SpMV operator and helper functions that store data in HBM channels of the FPGA. We used HiSparse code for SpMV implementation and added other helper functions to haHyperEF that converts hypergraph stored in hMetis file to a CSC sparse matrix, segments data of sparse matrix represented in CSC or CSR representation over HBM channels for processing, and performs post-processing of SpMV data.

The process of translating the HLS design from C++ to bitstream is known to be time-consuming [4], taking approximately 20 hours to build the haHyperEF bitstream. Furthermore, the process of building the bitstream and testing the design's basic functionality on hardware can be slow and protracted, taking more than one day. In order to expedite the logic verification process, we chose to utilize an HLS testbench that was capable of testing minimal functionality of the HDL in just a few hours. Notably, the HLS tool necessitates a C++ testbench to verify the generated Verilog/VHDL code, enabling users to examine the inner workings of the generated HDL design and verify the HLS. As we decided to use HiSparse to accelerate SpMV, we had to implement sparse matrix algebra capable of receiving HiSparse HLS data structures. Additionally, the HLS testbench had to be a single-threaded C++ implementation.

We considered performance of the single-threaded C++ implementation and POC Julia implementation, which we customized to run only the Filter routine as the baseline for our experiments to evaluate haHyperEF.

B. Experimental Setup

In order to conduct functionality and performance tests on haHyperEF, we utilized a machine equipped with an FPGA U280 board capable of running programs written in both C++ and Julia. In order to facilitate communication between the FPGA board and the host machine, we leveraged Xilinx Run Time (XRT), a communication layer that includes APIs and drivers. However, it should be noted that XRT is exclusively supported on Linux operating systems. In order to monitor CPU utilization and execution time, we employed Linux tools.

TABLE III
Datasets used for experiments are listed with the number of hyperedges

Dataset	Hyperedges
ibm01	14111
ibm02	19584
ibm03	27401
ibm04	31970
ibm05	28446
ibm06	34826
ibm07	48117
ibm08	50513
ibm09	60902
ibm10	75196
ibm11	81454
ibm12	77240
ibm13	99666
ibm14	152772
ibm15	186608
ibm16	190048
ibm17	189581
ibm18	201920

C. Test plan for functional verification and performance

The proof of concept (POC) Julia implementation performs all floating-point calculations with double precision, which is a representation of real numbers as 64-bit floating-point numbers. In contrast, HiSparse only supports single-precision floating-point calculations, where real numbers are represented as 32-bit floating-point numbers. Therefore, Filter's Julia implementation will perform mathematical operations with greater accuracy than haHyperEF, and their results will vary with a small error. When comparing the results of the two implementations, there will be a slight difference or mismatch, but the difference will be minimal.

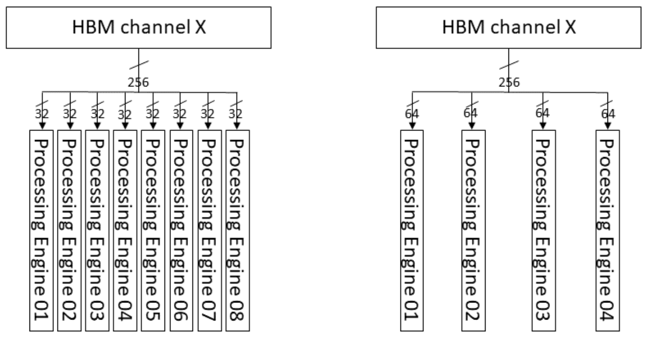


Fig. 8. On the left, we have eight processing engines(PEs) as numbers are represented using a 32-bit floating-point, while on the right, we have 4 PEs since numbers are represented using a 64-bit floating-point.

Although upgrading the floating-point operator in HiSparse could enhance the precision of calculations, we decided not to pursue this option due to memory and performance limitations. Two 32-bit floating-point numbers can be stored in a memory element which stores a 64-bit floating-point number. Also, we can only access 256-bit data per cycle through a U280 FPGA HBM (High Bandwidth Memory) channel, and either four double-precision calculations or eight single-precision calculations per cycle (see fig. 8) can be performed. Fig. 8 shows how a 256-bit data from a HBM channel is distributed based on floating-point precision. Thus, upgrading to double-precision calculations would significantly reduce the number of calculations that can be performed per cycle, slowing down the overall performance of haHyperEF by a factor of two.

haHyperEF uses the Xilinx logiCORE IP Floating-Point operator [11] for single-precision floating-point computations, which does not support denormalized numbers. Therefore, it is expected to observe a small difference between values generated by haHyperEF and Filter's C++ implementation running on the host computer, even though both values are generated with single-precision floating-point computation. Expecting such differences in results, it necessitates to have a functional tests that ignores tiny differences in results. Test checker while ignoring small errors must keep a count of mismatches as small differences could be due to the nature of inputs of SpMV. To do so we added checker to sum all the error and flag whenever the sum is greater than a particular threshold. We added test (Test 01, table IV) to verify equivalence of C++ implementation and HLS. And other functional test (Test 02, table IV) that we use to establish the equivalence of C++ implementation and Filter's Julia implementation. This way if C++ implementation is equivalent to Julia implementation and HLS is equivalent to C++ we can conclude that HLS is equivalent to the original POC implementation by using transitive property of logical equivalence.

XRT requires software application that runs Filter on FPGA U280 to program FPGA board with the bit-stream of the accelerated application. This bit-steam programming is an overhead and to quantize the cost of programming we added performance test (Test 03, table IV) that program FPGA while we run Filter only once and compare the computation time to performance tests that program FPGA once but run Filter a given number of time to ascertain the bit-stream programming overhead.

Lastly, the number of computations per SpMV operation largely depends on the size of the input matrix; the bigger the matrix larger the number of computations. So we added test case (Test 04, table IV) to compare the execution time of Filter Kernel on FPGA, Filter's C++ implementation, and Filter's Julia implementation with different standard hMetis datasets. All the datasets considered for experiments are listed in table III with number of hyperedges.

D. Results

After running Test 01 listed in table IV, we found haHyperEF results to be equivalent to Filter's C++ implementation and Filter's C++ implementation equivalent to Filter's Julia implementation. All differences between the results of haHyperEF and Filter's C++ implementation were found to be less than 1e-6, and the sum of all the absolute errors did not exceed 1e-4. Minute differences in results are expected if the Filter's C++ implementation and haHyperEF are equivalent as both calculate floating-point calculations with single precision.

TABLE IV
FUNCTIONAL VERIFICATION AND PERFORMANCE TESTS

Test type	Test	Description
Functional test	01	Code equivalence check: haHyperEF v/s Filter's C++ implementation
Functional test	02	Code equivalence check: Filter's C++ implementation v/s Filter's Julia implementation
Performance test	03	haHyperEF bitstream programming time
Performance test	04	To quantify execution time of haHyperEF, Filter's C++ implementation, and Filter's Julia implementation

However, on observing data of Test 02(table IV) the difference in the results of Filter's C++ implementation and Filter's Julia implementation was significant as the C++ implementation uses single precision while the latter implementation uses double precision. Differences were significant enough to make Test 02 results inconclusive. So to establish logic equivalence, we changed the precision of the C++ implementation to double. We found that all differences in results were under 1e-6, and the sum of all the absolute errors did not exceed 5e-2. Tiny deviations in results indicate that C++ implementation and Julia implementation are functionally equivalent.

TABLE V
COMPARISON OF EXECUTION TIMES OF FILTER KERNEL ON FPGA WITH VARIOUS DATASETS AND REPETITIONS(TEST 03)

Dataset	Repetition	Execution time (s)
ibm12	1	2.66
ibm12	16	28.78
ibm15	1	5.48
ibm15	16	72.75

As discussed earlier in section IV-C, programming an FPGA board before running the Filter kernel on FPGA takes considerable time. We ran a series of tests (Test 03, see table IV) to quantify FPGA programming overhead which is dependent on both the FPGA board and the host machine's performance. First, we ran a test to run Filter on FPGA only once and measured the total execution time. Later we ran Filter a different number of times to approximately calculate the FPGA programming time. The processing times of haHyperEF for two datasets is presented in the table V. We observed that the host machine takes around $900ms$ to $1s$ to program the FPGA and only then proceed to run the desired application.

Table VI lists experiment data of Test 04, comparing the performance of haHyperEF against Filter's C++ and Julia implementation. In fig. 9, execution time is plotted for the different datasets with increasing numbers of hyperedges. The curve in green represents the execution time of haHyperEF. The orange curve represents the Filter's Julia implementation, while the yellow curve represents the Filter's C++ implementation. Suppose we ignore the programming time of the FPGA. In that case, we see that haHyperEF outperforms all other Filter's software implementations—the trend in Fig. 9 suggests that haHyperEF outperforms significantly for all the datasets not exceeding 100,000 hyperedges, i.e., ibm01-ibm13.

TABLE VI
COMPARISON OF EXECUTION TIMES OF FILTER KERNEL ON FPGA WITH DIFFERENT DATASET (FPGA PROGRAMMING TIME IS INCLUDED)

dataset	Julia (s)	C++ (s)	haHyperEF (s)
ibm01	3.1	0.69	1.4
ibm02	3.22	1.02	1.6
ibm03	3.27	1.29	1.7
ibm04	3.26	1.52	1.8
ibm05	3.28	1.54	1.89
ibm06	3.35	1.75	2.03
ibm07	3.58	2.51	2.43
ibm08	3.62	2.78	2.63
ibm09	3.72	3.09	2.76
ibm10	3.96	4.07	3.24
ibm11	4.05	4.11	3.31
ibm12	3.98	4.29	3.35
ibm13	4.28	5.1	3.92
ibm14	5.24	8.46	4.86
ibm15	5.53	10.21	5.56
ibm16	5.97	11.01	6.04
ibm17	6.07	11.56	6.13
ibm18	6.27	12.07	6.69

haHyperEF's software application can be modified only to exit when prompted by the user, which makes haHyperEF ready to process the next hypergraph, avoiding the re-programming of FPGA.

TABLE VII
SPEED UP ON FPGA COMPARED TO FILTER'S JULIA AND C++ IMPLEMENTATIONS (FPGA PROGRAMMING TIME IS IGNORED FOR THESE CALCULATIONS.)

dataset	Speed up compared to Julia	Speed up compared to C++
ibm01	6.2	1.4
ibm02	4.6	1.5
ibm03	4.1	1.6
ibm04	3.6	1.7
ibm05	3.3	1.6
ibm06	3.0	1.5
ibm07	2.3	1.6
ibm08	2.1	1.6
ibm09	2.0	1.7
ibm10	1.7	1.7
ibm11	1.7	1.7
ibm12	1.6	1.8
ibm13	1.4	1.7
ibm14	1.3	2.1
ibm15	1.2	2.2
ibm16	1.2	2.1
ibm17	1.2	2.2
ibm18	1.1	2.1

Table VII shows the speedup of Filter by haHyperEF relative to Julia and C++ Filter implementations. haHyperEF accelerates Filter by a factor of 2 for smaller hypergraphs with hyperedges not exceeding 60,000 when compared to POC Julia implementation. In our experiments, haHyperEF does not outperform Julia's implementation for the dataset with at least 200,000 hyperedges, and the speedup is not significant. In order to saturate the bandwidth of HBM, matrix data is segmented into multiple chunks. HaHyperEF needs the host processor to communicate information like, data chunk addresses and data sizes for every subtask of an SpMV operation.

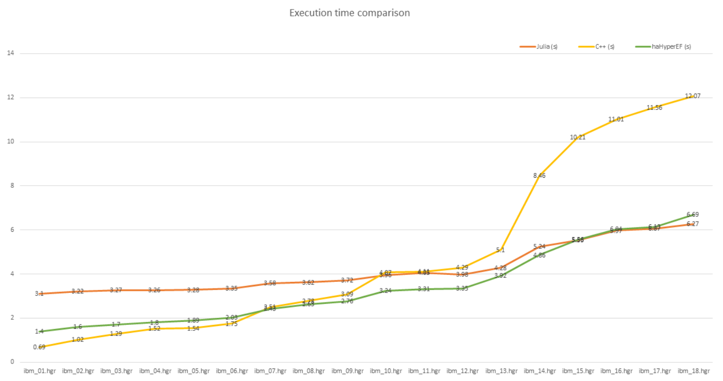


Fig. 9. Comparison of execution times of Filter kernel on FPGA with different dataset (FPGA programming time is included)

This communication overhead is significant for datasets with hyperedges exceeding 200,000; the larger the dataset, the more segments. Moreover, this communication overhead degrades the performance of haHyperEF for large datasets. However, the performance of haHyperEF always stays ahead of the Filter's C++ implementation.

V. CONCLUSION

This work presents the potential of FPGA to accelerate a slow algorithm and discusses the limitations of such hardware platforms in boosting performance. An HBM channel to FPGA fabric interfacing requires significant logic utilization, which causes routing congestion as we use multiple HBM channels. In our design, we could only use up to 20 HBM out of the available 32 HBM channels, as further utilization causes routing issues, limiting hardware acceleration. Also, host-to-FPGA device communication can be a significant overhead, degrading performance boost for large hypergraphs. In addition, while writing HLS for Filter(), we discovered that algorithm specifications are more straight-forward to write in high-level languages like Julia and Python than in C/C++. Writing algorithm specifications that the HLS tool can translate to RTL complicates matters further. Recent work like HetroCL [7] decouples algorithm specification from hardware customization, and such tools make the translation of algorithms from high-level language to HDL easier. However, we did not find these tools to solve the HBM interfacing routing congestion issue. In our experiments, haHyperEF outperforms single-threaded C++ implementation for all datasets and accelerates Filter 2x compared to Julia implementation for datasets not exceeding 100,000 hyperedges.

REFERENCES

[1] A. Aghdaei, Z. Feng, "HyperEF: Spectral Hypergraph Coarsening by Effective-Resistance Clustering", ICCAD, Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design, article. 14, pp. 1–9, December 2022.
[2] X. Chen, H. Tan, Y. Chen, B He, W.-F. Wong, D. Chen, "ThunderGP: HLS-based Graph Processing Framework on FPGAs", FPGA, Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp 69–80, February 2021.

[3] Y. Hu, Y. Du, . Ustun and Z. Zhang, "GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs", 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pp. 1–9, November 2021.

[4] Y. Du, Y. Hu, Z. Zhou, Z. Zhang, "High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV", FPGA, Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 54–64, February 2022.

[5] 12th Generation Intel® Core™ Processors Datasheet https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/002/memory-controller-mc/

[6] NVIDIA Multi-Instance GPU (NVIDIA H100, A100, and A30 Tensor Core GPUs)

[7] Y. H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, Z. Zhang, "HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing", FPGA, Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 2019.

[8] AMD Xilinx Alveo U280 Product Brief https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/alveo-u280-product-brief.pdf https://www.nvidia.com/en-us/technologies/multi-instance-gpu/

[9] HyperEF source code https://github.com/feng-research/hyperef

[10] hMetis http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview

[11] Xilinx Floating-Point Operator https://www.xilinx.com/products/intellectual-property/floating_pt.html