High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV

Yixiao Du[1], Yuwei Hu[1], Zhongchun Zhou[2]*, Zhiru Zhang[1]
[1]Cornell University [2]Tsinghua University
{yd383,zhiruz}@cornell.edu

ABSTRACT

Sparse linear algebra operators are memory bound due to low compute to memory access ratio and irregular data access patterns. The exceptional bandwidth improvement provided by the emerging high-bandwidth memory (HBM) technologies, coupled with the ability of FPGAs to customize the memory hierarchy and compute engines, brings the potential to significantly boost the performance of sparse linear algebra operators.

In this paper we identify four challenges when developing high-performance sparse linear algebra accelerators on HBM-equipped FPGAs — low HBM bandwidth utilization with conventional sparse storage, limited on-chip memory capacity being the bottleneck when scaling to multiple HBM channels, low compute occupancy due to bank conflicts and inter-iteration carried dependencies, and timing closure on multi-die heterogeneous fabrics. We conduct an in-depth case study on sparse matrix-vector multiplication (SpMV) to explore techniques that tackle the four challenges. These techniques include (1) a customized sparse matrix format tailored for HBMs, (2) a scalable on-chip buffer design that combines replication and banking, (3) best practices of using HLS to implement hardware modules that dynamically resolve bank conflicts and carried dependencies for achieving high compute occupancy, and (4) a split-kernel design methodology for frequency optimization. Using the techniques, we demonstrate HiSparse, a high-performance SpMV accelerator on a multi-die HBM-equipped FPGA device. We evaluated HiSparse on a variety of matrix datasets. The results show that HiSparse achieves a high frequency and delivers promising speedup with increased bandwidth efficiency when compared to prior arts on CPUs, GPUs, and FPGAs. HiSparse is available at https://github.com/cornell-zhang/HiSparse.

CCS CONCEPTS

• **Hardware** → **Hardware accelerators**; • **Computer systems organization** → *Data flow architectures.*

KEYWORDS

FPGA Acceleration; HBM; HLS; Sparse Matrix-Vector Multiplication

ACM Reference Format:
Yixiao Du, Yuwei Hu, Zhongchun Zhou, Zhiru Zhang . 2022. High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '22), February 27-March 1, 2022, Virtual Event, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3490422.3502368

1 INTRODUCTION

Sparse linear algebra operators, such as sparse matrix-vector multiplication (SpMV), sparse matrix-sparse vector multiplication (SpMSpV), and sparse matrix-matrix multiplication (SpMM), are key computational primitives used in a broad range of applications, such as linear system solvers [1], graph processing [2], and inference of compressed neural networks [3]. FPGAs are an appealing platform for accelerating these sparse linear algebra operators. Compared to CPUs and GPUs, FPGAs can better exploit the fine-grained parallelism in these operators by customizing the memory hierarchy and compute engines [4]. In addition, FPGAs typically consume less power than CPUs and GPUs. There have been continuous efforts on building FPGA-targeted sparse linear algebra accelerators [5–8], most of which are designed for DDR memory systems.

The emerging high-bandwidth memory (HBM) has the potential to significantly boost the performance of sparse workloads, which are memory bound due to low compute to memory access ratio and irregular data access patterns. HBM devices deliver a much higher bandwidth than DDR memories by providing multiple memory channels that can service memory requests concurrently. HBMs have been adopted into modern FPGAs, such as Intel Stratix 10 MX and Xilinx Alveo U280.

In this paper, we perform a case study on SpMV to explore techniques for building highly efficient sparse linear algebra accelerators on HBM-equipped FPGAs. We identify four major challenges to effective SpMV acceleration on FPGAs with HBM support, including (1) low HBM bandwidth utilization with conventional sparse storage, (2) limited on-chip memory capacity being the bottleneck when scaling to multiple HBM channels, (3) low compute occupancy due to bank conflicts and inter-iteration carried dependencies, and (4) timing closure on multi-die heterogeneous fabrics.

To address these challenges, we propose and develop HiSparse, a high-performance SpMV accelerator on multi-die HBM-equipped FPGAs. To fully utilize the available bandwidth of HBM for loading in the sparse matrix, HiSparse stores the sparse matrix in a customized format that allows vectorized, streaming accesses to each HBM channel and concurrent accesses to multiple HBM channels. To maximize the data reuse in accesses of the input vector, HiSparse introduces a scalable on-chip buffer design that combines

∗ This work was performed while Zhongchun was a (remote) research intern with Cornell University.

vector replication and banking to feed a large number of parallel processing engines (PEs). Furthermore, to support arbitrarily large matrices (within the capacity of HBM), we partition the matrices along both rows and columns according to the size of the on-chip buffers. One synchronization per partition is required since the buffers need to be cleared when switching partitions.

We implement HiSparse using high-level synthesis (HLS). While recent years have seen a rapidly increasing adoption of HLS for accelerator development, a majority of existing HLS designs target dense computations, such as dense matrix multiplication [9–11], image/video processing [12–14], and convolutional neural networks [15–17]. Developing high-performance sparse accelerators using HLS is more challenging because the irregular compute pattern of sparse workloads causes bank conflicts and carried dependencies. We manage to develop a pipelined, non-blocking arbiter and a pipelined PE with load-store forwarding to resolve bank conflicts and carried dependencies, respectively, using HLS through iteration-level modeling and a proper coding style.

We tackle the challenge of timing closure on multi-die HBM-equipped FPGAs by adopting a split-kernel design methodology. More concretely, we split the hardware modules of HiSparse (e.g., data loaders, PEs) into multiple groups, implement each group as one OpenCL kernel, and use pipelined interfaces for inter-kernel communication. We further apply two optimizations: (1) Confining each kernel to a specific die during floorplanning to eliminate die boundary crossings caused by centralized combinational control signals; (2) Adding registers and relay units between the HBM and data loaders that are placed at a die far from the HBM. The split-kernel design achieves a higher frequency than a monolithic counterpart — 237 MHz vs. 117 MHz. The main drawback of the split-kernel design is the increased programming effort.

We implement HiSparse on a Xilinx Alveo U280 FPGA platform, using 18 HBM channels delivering 258 GB/s bandwidth in total. Evaluation results on a variety of matrix datasets show that compared to MKL running on a 32-core Xeon CPU, HiSparse achieves 4.1× higher throughput and 4.6× higher bandwidth efficiency; compared to cuSPARSE running on a GTX 1080 Ti GPU, HiSparse achieves comparable throughput and 1.9× higher bandwidth efficiency. HiSparse is 37× and 3.7× more energy-efficient than MKL and cuSPARSE, respectively. We further compare HiSparse to Vitis Sparse Library (VSL), which is the only existing SpMV accelerator on HBM-equipped FPGAs to our knowledge. The current SpMV implementation in VSL cannot handle large matrices and is 30% slower than HiSparse on small matrices. We give a detailed comparison against VSL in the evaluation and related work sections.

The main contributions of this paper are as follows:

- We identify the opportunities of using HBM-equipped FPGAs for accelerating sparse linear algebra and discuss the challenges in four aspects — HBM bandwidth utilization, on-chip memory utilization, compute occupancy, and timing closure.
- We propose HiSparse, a high-performance SpMV accelerator on HBM-equipped FPGAs. Using HiSparse as a case study, we present techniques to tackle the aforementioned four challenges. We study HiSparse under both fixed-point and floating-point data types. Evaluation results show that HiSparse delivers promising speedup with increased bandwidth efficiency when compared to prior arts on CPUs, GPUs, and FPGAs.
- We illustrate best practices of using HLS to implement hardware modules that dynamically resolve bank conflicts and carried dependencies for achieving high compute occupancy. We also discuss potential enhancements in HLS tools to better support developing high-performance sparse accelerators. HiSparse is available at https://github.com/cornell-zhang/HiSparse.

The rest of this paper is organized as follows. Section 2 reviews the background on SpMV and modern multi-die HBM-equipped FPGAs and discusses the major challenges to effective SpMV acceleration on FPGAs with HBM support. Section 3 presents the sparse matrix format and accelerator architecture co-design of HiSparse. Section 4 describes frequency optimizations. Section 5 studies HiSparse under floating-point datatype. We evaluate HiSparse in Section 6 and discuss how to extend HiSparse to support sparse linear algebra operators beyond SpMV in Section 7. We talk about related work in Section 8 and summarize in Section 9.

2 BACKGROUND AND MOTIVATION

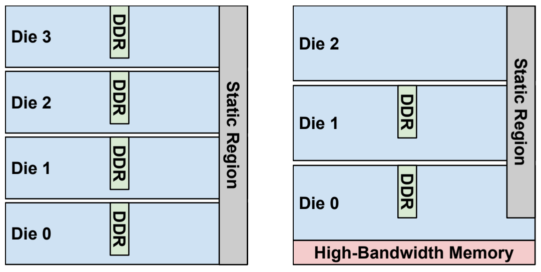
2.1 Sparse Matrix-Vector Multiplication (SpMV)

Representative applications of SpMV include graph analytics, such as PageRank [18], as well as inference of compressed neural networks, such as Transformers [19]. PageRank can be computed by iteratively applying SpMV, where the sparse matrix is the adjacency matrix of the graph and the dense vector is the ranks of the vertices. One layer of a compressed Transformer can be computed as an SpMV, where the sparse matrix is the compressed weight and the dense vector is the embedding. In PageRank, the sparse matrices are usually large (with millions of rows and columns) and highly sparse with a typical density below 0.1%. In contrast, the sparse matrices of compressed Transformers are smaller (with thousands of rows and columns) and less sparse with a typical density of 10-40%.

In this work, we focus on SpMV mainly for three reasons: (1) There are two major data access patterns in SpMV — streaming accesses of the sparse matrix exhibiting no data reuse and random accesses of the dense vector exhibiting data reuse. Both data access patterns are typical in sparse linear algebra, and each poses a unique challenge to the accelerator design. The first pattern demands a high off-chip memory bandwidth, while the second demands efficient use of on-chip buffers; (2) SpMV, similar to other sparse linear algebra operators, has irregular compute patterns, which pose challenges to achieving high occupancy of the parallel processing engines (PEs) in an accelerator; (3) In addition, it is possible to extend an SpMV accelerator to handle other sparse linear algebra operators. For example, we can compute SpMM as a batch of SpMV; we can support SpMSpV by considering the sparsity of the vector.

2.2 Multi-Die HBM-Equipped FPGAs

HBM[20] is a new memory technology that offers high bandwidth by vertically stacking multiple memory dies. Logically, an HBM device provides multiple memory channels that can be accessed concurrently. HBMs have been adopted into modern FPGAs such as Intel Stratix 10 MX and Xilinx Alveo U280. To fully utilize the high bandwidth of HBM devices, the hardware must perform parallel, vectorized, and streaming accesses at a high clock frequency.



(a) U250. **(b) U280.**

Figure 1: Comparison between Xilinx Alveo U250 and U280.

Modern FPGAs integrate multiple chip dies in a single package to increase the available on-chip resources [21], allowing for building large designs. However, the interconnections across dies have to travel through the silicon interposer and incur non-trivial delays. Moreover, when an HBM device is integrated into a multi-die FPGA, not all dies have direct connections to the HBM interface. Figure 1 shows the heterogeneous architecture of U280 where all the HBM channels connect to a single die, compared with the homogeneous architecture of U250 where four DDR channels each connects to one die. On U280, a hardware module's access to HBM would incur a long cross-die delay when the module is placed on a die far from the HBM interface, raising challenges to timing closure.

2.3 Challenges to SpMV Acceleration

There are several major challenges to unleashing the full potential of HBM-equipped FPGAs for accelerating SpMV:

- It is difficult to make full use of HBM bandwidth with the compressed sparse row (CSR) sparse matrix format. CSR stores all non-zeros and the corresponding column indices row-by-row in contiguous memory locations. It uses a separate row pointer array to denote the start location of each row. The row pointer array in CSR prevents fully streaming accesses to the non-zeroes — the accelerator has to first access the row pointer array before reading the non-zeros. Also, the continuous storage of non-zeros prevents cross-row vectorized accesses.
- The limited on-chip memory capacity becomes a bottleneck when the accelerator uses a large number of PEs to saturate the HBM bandwidth. Since every PE requires random accesses to the dense vector, simply allocating a vector buffer for every PE without resource sharing will run out of the on-chip memories.
- The irregularity in the compute pattern of SpMV causes bank conflicts and carried data dependencies, leading to low compute occupancy. Both bank conflicts and carried dependencies pose challenges to contemporary HLS compilers, which typically use static scheduling to generate a conservative pipeline with low throughput.
- The last challenge is to tackle the heterogeneous architecture of HBM-equipped FPGAs to achieve a high frequency, as described in Section 2.2.

3 HISPARSE DESIGN

In this section, we first present the customized sparse matrix format that is suitable for saturating the bandwidth of HBM (Sec 3.1) and give an overview of the accelerator architecture (Sec 3.2). We then describe the shared vector buffer with shuffle unit (Sec 3.3) and the PE with load-store forwarding (Sec 3.4).

3.1 Sparse Matrix Format

We customize the sparse matrix format to support vectorized, streaming accesses to each HBM channel and concurrent accesses to multiple channels. We construct a matrix in our format through partitioning, streamizing, and packing. Figure 2 illustrates our format on an example 24×24 matrix with a configuration of two HBM channels and pack size 2.

Partitioning is required to handle large matrices that exceed the buffer size. Since we buffer both the vector and the output, we need to partition the matrix along both the rows and columns. Figure 2a illustrates the partitioning scheme — assume both the vector buffer size and the output buffer size are 12, then we partition the 24×24 matrix into 4 (i.e., 2×2) partitions. We double buffer the vector buffer to hide the cost of vector loading when switching partitions.

The next step is streamizing, which cyclically assigns rows to PEs and concatenates the rows that are assigned to one PE into one stream. We insert a next-row marker at the end of a row to avoid indirect addressing as in CSR, thus enabling fully streaming accesses. For example, in Figure 2b, the next-row marker of value +2 in stream 0 indicates the end of row 0 and the start of row 8; row 4 is empty, hence skipped. Since partitioning may generate many empty rows on highly sparse matrices, skipping empty rows is crucial for compact storage.

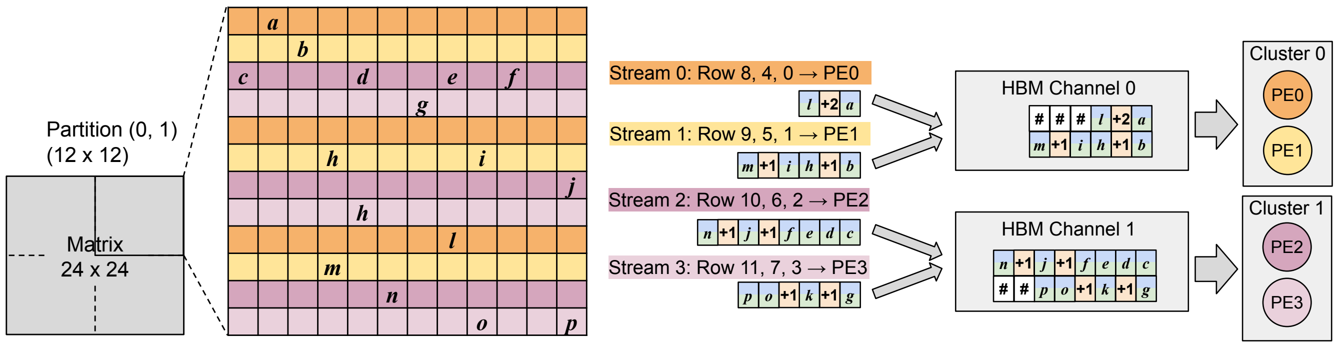
The final step is packing the streams of elements into streams of packets. Assume one element requires a 32-bit value and a 32-bit column index, and one HBM channel delivers 128 bits per access, then we pack two (i.e., 128 / (32 + 32)) elements into one packet. We store one stream of packets in one HBM channel, which is accessed by a cluster of two PEs, each processing one stream of elements.

3.2 Accelerator Architecture Overview

HiSparse makes efficient use of limited on-chip memory to exploit data reuse by sharing the vector and achieves high compute occupancy with hardware modules that dynamically resolve bank conflicts and carried data dependencies.

Figure 3 depicts the design of the SpMV accelerator. It is a dataflow architecture consisting of multiple clusters, a vector loader, and a result draining unit. Each cluster connects to one HBM channel. The vector loader reads in the dense vector from the off-chip memory and feeds each cluster a replica of the vector, which will be shared inside that cluster. The draining unit concatenates the outputs from the clusters to generate the final output and writes it back to the off-chip memory.

A cluster consists of a matrix loader, a set of vector buffer access units (VAUs), two shuffle units, an unpack unit, a pack unit, and a collection of processing engines (PEs). The matrix loader reads in a stream of packets, unpacks it into streams of elements, and decodes the next-row markers to restore the row indices. Each VAU handles requests to one vector buffer bank. It also manages double buffer control of the bank. The shuffle units route the requests/responses to/from the VAUs, with a non-blocking resolution of bank conflicts. Each PE multiplies the incoming non-zero value from the matrix with the corresponding vector value, and accumulates the product to the output buffer location indicated by the row index.


Partition (0, 1)
(12 x 12)
Matrix
24 x 24
Stream 0: Row 8, 4, 0 → PE0
Stream 1: Row 9, 5, 1 → PE1
Stream 2: Row 10, 6, 2 → PE2
Stream 3: Row 11, 7, 3 → PE3
HBM Channel 0
HBM Channel 1
Cluster 0
Cluster 1
PE0
PE1
PE2
PE3

(a) Partitioned matrix.

(b) Concatenation of rows to streams and packing of streams.

Figure 2: The customized sparse matrix format on an example 24 × 24 matrix — A "+" with a number represents a next-row marker. A "#" represents additional padding.

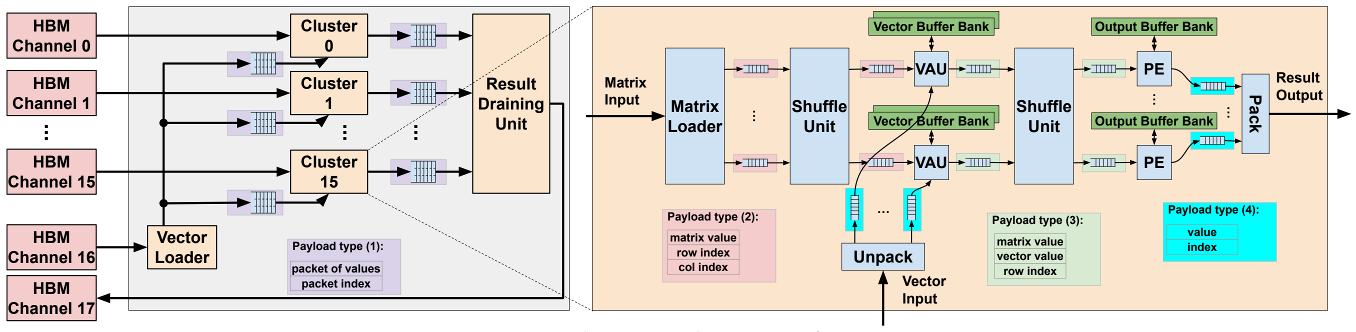


Figure 3: Accelerator architecture of HiSparse.

Our parallelization scheme ensures that no bank conflicts would occur on the output buffer. Since we pack the inputs to better utilize the memory bandwidth, we need an unpack unit to unwrap the incoming packets into individual elements and feed them into the VAUs. We also have a pack unit to collect outputs from the PEs, pack them, and stream the packets into the draining unit.

There are four types of payloads in the accelerator: (1) a value packet with a packet index, used for the inter-cluster packed vector payloads; (2) the matrix non-zero value with the corresponding row index and column index, used between the matrix loader and the VAUs; (3) the matrix non-zero value with the corresponding row index and vector value, used between the VAUs and PEs; (4) value-index pairs, used for both the input dense vector to VAUs and the result dense vector from PEs.

In payload types (1) and (4), the index information for a dense vector is redundant and can be safely removed. Nevertheless, we decide to keep the index information so that the architecture can be easily extended to support other sparse linear algebra operators that deal with a sparse vector, such as SpMSpV.

3.3 Shared Vector Buffer with Shuffle Unit

Logically, each PE requires random accesses to the same input dense vector. Replicating the input vector for every PE is not scalable to multiple HBM channels when using a large number of PEs. To tackle this problem, we share the dense vector buffer across all PEs within one cluster and bank the shared buffer to increase throughput. The input vector across different clusters remains replicated.

```
/* N is the number of input lanes */
while (!exit) {
#pragma HLS pipeline // II will be N
    for (int i = 0; i < N; i++) {
#pragma HLS unroll
        payload[i] = in_lane[i].read();
        out_lane[payload[i].target].write(payload[i]);
    }
}
```

Figure 4: Shuffle unit with implicit control logic.

The key hardware module that manages banking is the shuffle unit, which is challenging to implement using HLS. Figure 4 shows a sub-optimal coding style in which the control logic is implicit. In this case, HLS tools using static scheduling cannot generate proper arbitration logic since the compile-time analysis assumes the worst-case traffic pattern. Hence the resulting HLS schedule is to process input lanes sequentially, with an initiation interval (II) of N.

In HiSparse, we explicitly implement a pipelined arbiter and re-sending logic that dynamically resolves bank conflicts by reordering the payloads in an input lane. More concretely, We implement the re-sending logic in HLS by iteration-level modeling, which refers to defining which operations the datapath should perform at a given iteration; if the datapath requires information from previous iterations, special data structures are implemented to store and forward the information.

Figure 5 shows the architectural diagram and the HLS code of the shuffle unit. The crossbar at lines 31–35 provides all-to-all connectivity from the input lanes to the output lanes. The arbiter instantiated at line 29 detects conflicts on the output lanes and controls the crossbar and the re-sending logic. For input payloads with the same target output lane, the arbiter only grants access to one payload in a round-robin manner. The arbiter defers the denied payloads by re-sending them in a non-blocking manner to the arbiter input as listed in lines 20–26.

Figure 6 shows a pipeline diagram of our solution when the number of input lanes is 2 and the pipeline depth of the arbiter is 4. The datapath will require the arbitration results from 4 iterations away in the past to determine whether a read should be performed on the input, and it also needs the correct payloads to be re-sent into the arbiter. We make use of the pipeline registers inside the arbiter to store the on-the-fly payloads, and use the dependence pragmas at lines 15-18 in Figure 5b to direct the HLS tool to implement the feedback paths. The feedback distance is determined by the pipeline depth of the arbiter, which indicates that a combinational arbiter will require no feedback and simplifies the design. However, such an arbiter will cause timing problems. We verified that an 8-input combinational arbiter will limit the frequency under 100 MHz.

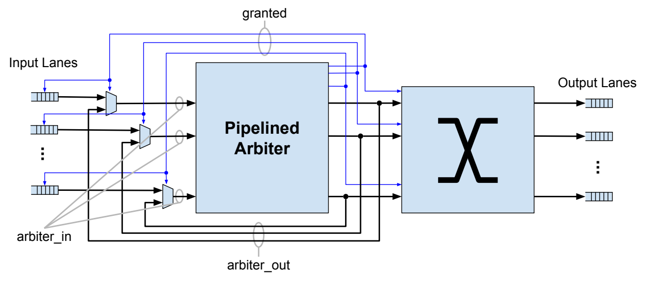
Reordering payloads raises challenges for synchronization. We need to finish reading every input lane and wait until all the on-the-fly payloads in the arbiter are processed. The total number of extra waiting cycles equals the number of input lanes times the depth of the arbiter pipeline, which is 8 in the example.

3.4 Pipelined PE with Load-Store Forwarding

In SpMV, accumulation on the output incurs read-after-write (RAW) dependencies. With a large output buffer, both read and write take multiple cycles, resulting in more RAW dependencies.

A straightforward approach is to store the results of accumulation in registers, and only performs read and write when the PE switches rows (i.e., writes the result of the current row from the register to the output buffer and reads the initial value of the next row from the output buffer to the register). Figure 8a shows a pipeline diagram of this approach. The downside is an extra iteration on every row switching and the associated pipeline bubbles. Moreover, reordering of payloads due to re-sending can cause additional row switches. This row switching overhead limits the throughput when processing extremely sparse matrices such as the adjacency matrix of graphs.

To fully pipeline the PE (i.e., achieve II = 1), we implement a load-store forwarding mechanism that dynamically resolves RAW dependencies, following the iteration-level modeling approach mentioned in Section 3.3. Figure 7 shows the PE architecture and corresponding HLS code. The PE derives the local bank address of the input payload by the `get_addr` unit and does multiplication and buffer read at the same time. Lines 10–21 implement the dependence resolution logic, which overwrites the buffer readout with the forwarded value if a RAW dependency is detected. In lines 22–33, the PE performs addition, writes the buffer, and updates the *in-flight write queue* (IFWQ), which is a shift register used for load-store forwarding. The IFWQ stores the address and value of writes initiated in the past iterations, with a valid bit to discriminate the nonexistent writes in the first few iterations. The depth of the IFWQ is equal to the sum of the read latency and the write latency of the output buffer. Figure 8b shows an example pipeline diagram with a depth-5 IFWQ, assuming the read latency is 3 and the write



(a) Architecture.

```
1  /* N is the number of input lanes
2     M is the number of output lanes
3     ARB_DEPTH it the depth of the arbiter pipeline */
4  // whether an input payload is granted access
5  bool granted[N];
6  // internal signals for arbiter input & output
7  payload_t arbiter_in[N], arbiter_out[N];
8  // sel: selection signal of the crossbar,
9  // generated by the arbiter according to
10 // the target field of input payloads
11 unsigned sel[M];
12
13 while (!exit) {
14 #pragma HLS pipeline
15 #pragma HLS dependence variable = granted \
16 inter RAW true distance = ARB_DEPTH
17 #pragma HLS dependence variable = payload_out \
18 inter RAW true distance = ARB_DEPTH
19     // read inputs or take re-send
20     for (int i = 0; i < N; i++) {
21 #pragma HLS unroll
22         if (granted[i])
23             arbiter_in[i] = in_lane[i].read();
24         else
25             arbiter_in[i] = arbiter_out[i];
26     }
27     // pipelined arbiter, depth = ARB_DEPTH
28     // arbiter_out is the copy of arbiter_in
29     arbiter(arbiter_in, sel, granted, arbiter_out);
30     // crossbar
31     for (int i = 0; i < M; i++) {
32 #pragma HLS unroll
33         if (granted[sel[i]])
34             out_lane[i].write(arbiter_out[sel[i]]);
35     }
36 }
```

(b) HLS code.

Figure 5: Shuffle unit with explicit control logic.

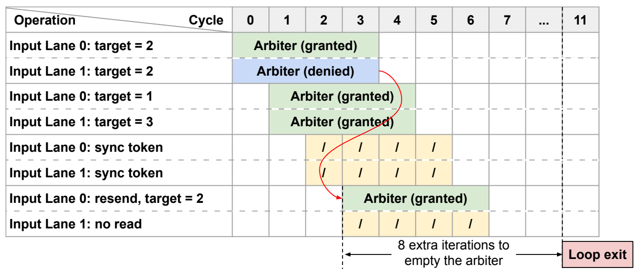
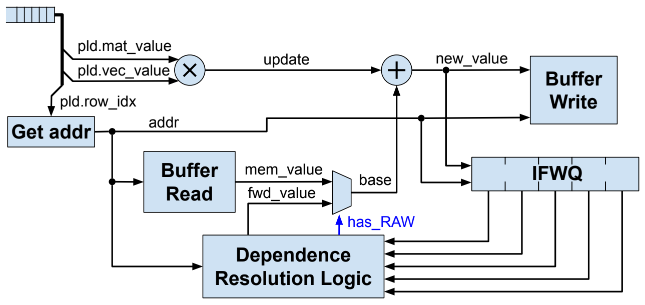


Figure 6: Pipeline diagram of the shuffle unit with a pipelined arbiter — The red arrow indicates re-sending of the denied payload.

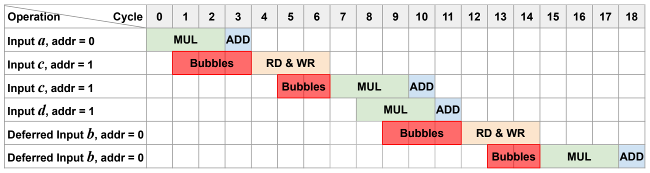


(a) Architecture.

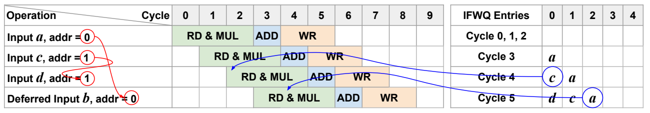
```
while (!exit) {
#pragma HLS pipeline
#pragma HLS dependence variable=out_buffer inter RAW false
    // fetch input and get bank address
    pld = in.read();
    addr = get_addr(pld.row_idx);
    // multiplication and read
    update = pld.mat_value * pld.vec_value;
    mem_value = out_buffer[addr];
    // dependence resolution logic
    fwd_value = 0;
    has_RAW = false;
    for (int i = 0; i < IFWQ_DEPTH; i++) {
#pragma HLS unroll
        if (addr == IFWQ[i].addr && IFWQ[i].valid) {
            has_RAW = true;
            fwd_value = IFWQ[i].data;
            break;
        }
    }
    base = has_RAW ? fwd_value : mem_value;
    // addition and write
    new_value = base + update;
    out_buffer[addr] = new_value;
    // update IFWQ
    // IFWQ[0] stores the latest in-flight write
    for (i = IFWQ_DEPTH - 1; i > 0; i--) {
#pragma HLS unroll
        IFWQ[i] = IFWQ[i - 1];
    }
    IFWQ[0].addr = addr;
    IFWQ[0].data = new_value;
    IFWQ[0].valid = true;
}
```

(b) HLS code.

Figure 7: PE with load-store forwarding.



(a) Using registers.



(b) Using load-store forwarding — Red arrows indicate the RAW dependencies. Blue arrows indicate the data forwarding to resolve dependencies.

Figure 8: Pipeline diagrams of resolving RAW dependencies.

latency is 2. Load-store forwarding fully pipelines the PE regardless of the order of the input payloads.

Using the IFWQ to resolve RAW dependencies only works when the addition stage is single-cycle, which is true for integer and fixed-point data types. For floating-point design, the addition can take multiple cycles and break the IFWQ-based data forwarding. We explore alternative methods to tackle this problem in Section 5.

4 TIMING CLOSURE ON MULTI-DIE FPGAS

In this section, we present our split-kernel design methodology to address the challenge to timing closure when implementing HiSparse on a multi-die FPGA.

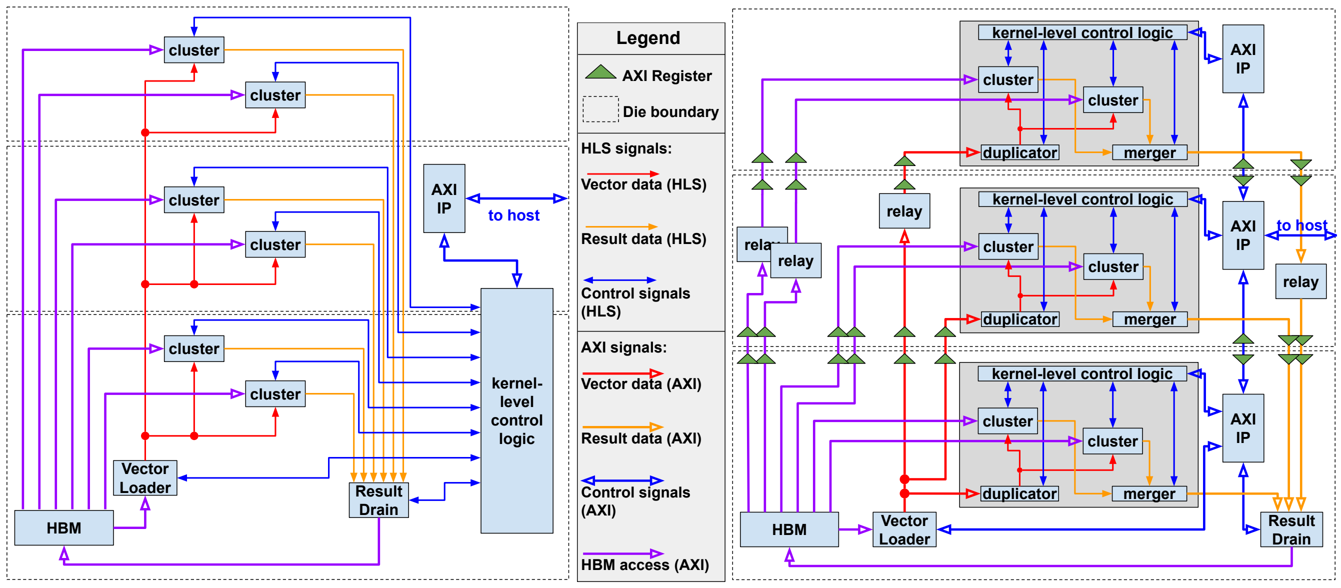
One straightforward implementation of the SpMV accelerator is writing a nested for-loop in one OpenCL kernel to iterate all matrix partitions, leading to a monolithic SpMV accelerator. Since the monolithic SpMV does not consider the physical layout of the device, it is difficult to find an optimal place-and-route solution. If no manual floorplanning is applied, the placement tool will try to squeeze all logic onto the same chip die to minimize the die boundary crossings. Such a highly condensed placement will increase the routing congestion level and degrade the timing of the design. Applying manual floorplanning to the monolithic SpMV to optimize the place-and-route results cannot solve the timing issue either. Figure 9a illustrates an example sub-optimal floorplanning solution. The HLS-generated signals, both for data accesses and control, contain combinational logic which cannot be pipelined to accommodate the die-boundary-crossing timing penalty. In short, the device limitation and HLS shortcomings result in poor timing closure of the monolithic SpMV.

We take the split-kernel approach to address this problem. Split-kernel means explicitly splitting the accelerator into multiple OpenCL kernels to minimize die-boundary crossings and pipelining necessary inter-die connections. We confine the logic to be placed on the same chip die into one kernel, and we connect different kernels with a protocol that can be pipelined to accommodate the latency of die-boundary-crossing. The kernel-level communication protocol we use is the AXI standard [22] as it is widely supported by FPGA design tools. Figure 9b shows the floorplanning of the split-kernel SpMV accelerator. The vector loader and result drain are also individual kernels placed close to the HBM interface. The split-kernel approach also well fits the heterogeneous architecture of the FPGA platform. We insert additional registers and relay units to paths that cross die boundaries to pipeline the remote connection. For high-fan-out and high-fan-in logic like the vector duplication and result concatenation, we implement them as multi-level structures to improve timing.

With the same number of clusters and the same buffer size, the split-kernel SpMV achieves 237 MHz, greatly outperforming the monolithic SpMV with 117 MHz.

5 FLOATING-POINT IMPLEMENTATION

Using fixed-point data types on FPGAs offers efficient hardware with accurate enough computation for a rich set of applications such as machine learning inference on compressed models [19], graph traversal algorithms like single-source shortest path (SSSP) [23], and graph analysis algorithms like PageRank [24]. However,



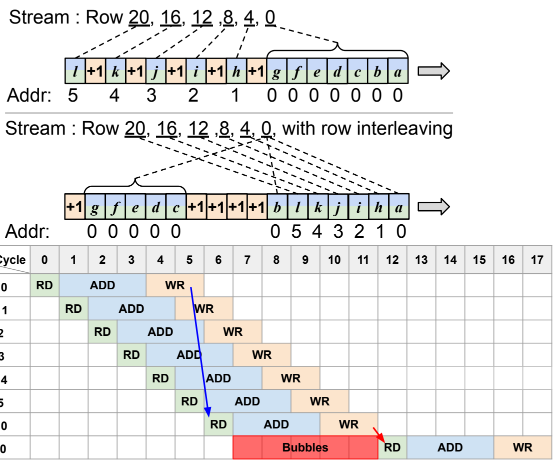
(a) **Monolithic SpMV** — inter-die signals degrade timing.

(b) **Split-kernel SpMV** — inter-die signals are pipelined.

Figure 9: Floorplanning of HiSparse.

there are cases where the large dynamic range of floating-point data type is preferred such as scientific computing. Therefore, we also study floating-point SpMV to evaluate the hardware cost.

On FPGA platforms with hardware floating-point units (e.g., Intel Stratix 10 NX and Agilex F-Series), the floating-point addition only takes a single cycle. Hence it will require minimal modifications to the PE pipeline of HiSparse to support floating-point SpMV. However, on FPGAs that use soft floating-point IPs, implementing single-cycle floating-point adders would not only consume more resources but also result in severe timing degradation. If we relax the latency of the floating-point addition, we must explore approaches to resolve the inter-iteration carried dependencies.


Stream : Row 20, 16, 12 ,8, 4, 0
l +1 k +1 j +1 i +1 h +1 g f e d c b a
Addr: 5 4 3 2 1 0 0 0 0 0 0 0
Stream : Row 20, 16, 12 ,8, 4, 0, with row interleaving
+1 g f e d c +1 +1 +1 +1 b l k j i h a
Addr: 0 0 0 0 0 0 5 4 3 2 1 0

Operation \ Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Input a, addr = 0	RD	ADD			WR													
Input h, addr = 1		RD	ADD			WR												
Input i, addr = 2			RD	ADD			WR											
Input j, addr = 3				RD	ADD			WR										
Input k, addr = 4					RD	ADD			WR									
Input l, addr = 5						RD	ADD			WR								
Input b, addr = 0							RD	ADD			WR							
Input c, addr = 0								Bubbles					RD	ADD			WR	

Figure 10: PE with row interleaving — Blue and red arrows indicate the dependencies resolved by row interleaving and stalling, respectively. "Addr" indicates the output buffer bank address.

One solution is to duplicate the output buffer into n partial buffers and rotate the PE between different partial buffers, where n is the latency of buffer access and floating-point add. Each partial buffer is updated every n cycles, which resolves the carried dependencies. We add up n partial buffers to get the final results. If the read latency is 1, the write latency is 2, and the floating-point add latency is 3, a total of 6 partial buffers are required. This approach can provide the same peak throughput as the fixed-point designs, but at a cost of excessive on-chip memory usage. A small output buffer will increase the number of complete vector loads, and a small vector buffer will increase the synchronization overhead and decrease the compute occupancy. Therefore, the partial buffer approach is only acceptable when the matrix is so small that the increased tiling and synchronization overhead is affordable.

Another solution is to add stall logic to resolve carried dependencies at run time and interleave different rows to avoid stalling. Figure 10 shows how row interleaving works on an example stream with 6 rows. With row interleaving, the PE first processes the first payload from row 0, then processes the first payload from rows 4, 8, etc., instead of processing all payloads in row 0 before moving to row 4. Since the number of payloads in each row is not the same, at the end of the stream, there are not enough rows to be interleaved. The payloads marked with $\boldsymbol{c}$ to $\boldsymbol{g}$ illustrate such situations. Therefore, the stall logic is still necessary. The key advantage of row interleaving is that it preserves the high on-chip memory utilization efficiency, and is thus more suitable than the partial buffer approach when processing large matrices. However, due to the inevitable stalls, the overall throughput would be lower than the fixed-point counterpart.

6 EVALUATION

6.1 Experiment Setup

HiSparse Configuration. We implement HiSparse on Xilinx Alveo u280 using 18 HBM channels (16 for the matrix, 2 for the input and result vectors), offering a memory bandwidth of 258 GB/s. Further scaling to more than 18 HBM channels causes routability issues. The input vector buffer size is 128 KB, and the output vector buffer size is 4 MB. We use UltraRAM (URAM), a high capacity on-chip memory, as the implementation for both buffers. The sizes of the buffers are determined by our exploration of the design space in Section 6.2. The pack size is 8 since one HBM channel delivers 512 bits per access and a value-index pair takes 64 bits. HiSparse runs at 237 MHz. Table 1 shows the resource utilization of HiSparse.

Table 1: Resource utilization of HiSparse.

LUT	REG	DSP	BRAM	URAM
544K (47.27%)	528K (22.7%)	688 (7.63%)	128 (7.22%)	512 (53.33%)

Configuration of Floating-Point Design. We also evaluate two floating-point variants of HiSparse (i.e., partial buffer and row interleaving) on the same platform. We use the IEEE-754 floating-point standard and the addition latency is 4. For the partial buffer design, the number of partial buffers is 8 so the logical output buffer size is only 0.5 MB. For the row interleaving design, the buffer sizes are the same as in the fixed-point design.

CPU and GPU Baselines. We compare HiSparse with vendor-provided sparse libraries, specifically MKL (2019.5) on the CPU and cuSPARSE (10.1) on the GPU. We conduct CPU experiments using 32 threads on a two-socket 32-core 2.8 GHz Intel Xeon Gold 6242 machine with 384 GB DDR4 memory providing 282 GB/s bandwidth. We conduct GPU experiments on a GTX 1080 Ti card with 3584 CUDA cores running at a peak frequency of 1582 MHz and 11 GB GDDR5X memory providing 484 GB/s bandwidth.

FPGA Baselines. We also compare HiSparse with existing FPGA sparse accelerators — ThunderGP [24] and Vitis Sparse Library (VSL) [25]. ThunderGP is the state-of-the-art FPGA accelerator on graph processing, although it is not targeting FPGAs with HBMs. We compare with ThunderGP on several common graph datasets. VSL is an optimized HLS library released as part of the Vitis 2020.2; it utilizes HBM for acceleration. Due to a known issue [26], VSL is incapable of running large matrices so we only compare with it on small datasets. The SpMV accelerator from ThunderGP utilizes the fixed-point data type, while VSL implements a floating-point SpMV using the partial buffer approach. The ThunderGP design is implemented on two FPGA platforms: Xilinx VCU1525 and Alveo U250. Both platforms are equipped with 4 × 16GB DDR4 memory delivering 77 GB/s bandwidth. We take the better result for comparison. We compile VSL on a Xilinx Alveo U280 platform with 16 HBM channels and 2 DDR channels providing 268 GB/s bandwidth in total, which is the only configuration supported by VSL. ThunderGP accelerator runs at 250 MHz, and VSL runs at 220 MHz.

Metrics. (1) Throughput, measured in Giga operations per second (GOPS). The multiplication and addition are counted as 2 separate operations. (2) Bandwidth efficiency, measured by throughput per unit bandwidth, in MOPS/GBPS. (3) Energy efficiency, measured by throughput per unit power, in GOPS/W.

Datasets. Table 2 lists the matrices we used for the evaluation. `googleplus`, `hollywood`, and `pokec` are social network graphs; they have been widely used in benchmarking graph processing systems. `mouse-gene` is a graph from computational biology. `ogbl-ppa` and `ogbn-products` are from OGB [27], a benchmark suite for the emerging graph neural networks. `transformer-x` is one layer from a compressed Transformer [19] model with a sparsity of x%.

Table 2: Matrix datasets.

Dataset	Size	Density	Dataset	Size	Density
`transformer-50`	$512 \times 33K$	50%	`mouse-gene`	$45K \times 45K$	1.42%
`transformer-60`	$512 \times 33K$	40%	`googleplus`	$108K \times 108K$	1.2×10^{-3}
`transformer-70`	$512 \times 33K$	30%	`ogbl-ppa`	$576K \times 576K$	127.9×10^{-6}
`transformer-80`	$512 \times 33K$	20%	`hollywood`	$1069K \times 1069K$	98.5×10^{-6}
`transformer-90`	$512 \times 33K$	10%	`pokec`	$1632K \times 1632K$	11.5×10^{-6}
`transformer-95`	$512 \times 33K$	5%	`ogbn-products`	$2449K \times 2449K$	20.6×10^{-6}

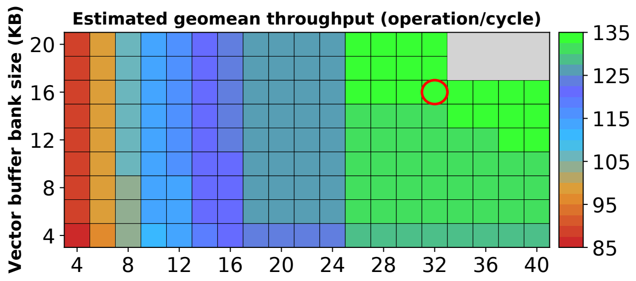


Figure 11: Design space exploration — We pick the design point highlighted by a red circle. The top right region represents design points that exceed the available URAM blocks.

6.2 Design Space Exploration

We explore the design space of HiSparse by varying the size of the vector buffer and the output buffer. To speed up the design space exploration, we build a performance model of HiSparse which provides an estimation of the execution time without the need of running hardware emulation. Our model calculates the cycle count based on the matrix size, matrix density, buffer size, pack size, the number of PEs, and the compute occupancy. All factors except the compute occupancy are known given an input matrix and a design configuration. The compute occupancy is determined by format overhead — the markers and padding are not involved in computation — and vector buffer bank conflicts. We analyze the formatted matrices to obtain format overhead and run software simulations to estimate the bank conflict ratio.

We exhaustively search the design space using the performance model, on all datasets used in the evaluation. Figure 11 shows the estimated geometric mean throughput on all datasets. When the output buffer bank size is larger than 25KB and the vector buffer bank size is larger than 16 KB, the performance plateaus at 130 to 135 operations per cycle.

We select the design point with 32-KB output buffer banks and 16-KB vector buffer banks, where the performance just reaches the plateau region. Because the size of one URAM block is 4096 words, we omit other design points with a smaller bank size to avoid URAM under-utilization. The logical size of the output buffer is 32 KB × 128 = 4 MB. The logical size of the vector buffer is 16 KB × 8 = 128 KB. This design point runs at 237 MHz and delivers 16.65 GOPS throughput.

6.3 SpMV Evaluation

Table 3 shows the comparison of HiSparse with CPU and GPU baselines. Compared to MKL, HiSparse is 4.1× higher in throughput, and 4.6× higher in bandwidth efficiency. Compared to cuSPARSE, HiSparse achieves the same throughput within the range of error, and is 1.9× higher in bandwidth efficiency. On graph datasets, HiSparse is 4.4× and 1.3× higher in throughput than the MKL and cuSPARSE,

Table 3: Throughput (GOPS) and bandwidth efficiency (MOPS/(GB/s)) compared to MKL and cuSPARSE.

Dataset	Throughput			Bandwidth efficiency		
	MKL	cuSPARSE	HiSparse	MKL	cuSPARSE	HiSparse
`transformer-50`	5.9	26.9	21.9	20.9	55.5	84.7
`transformer-60`	5.6	21.5	18.9	19.9	44.5	73.4
`transformer-70`	5.2	17.7	16.5	18.3	36.6	63.9
`transformer-80`	4.1	19.4	14.8	14.6	40.1	57.4
`transformer-90`	2.3	13.6	9.7	8.1	28.0	37.8
`transformer-95`	1.2	10.7	5.7	4.3	22.2	22.0
Geomean	3.5	17.5	13.3	12.5	36.2	51.7
`mouse-gene`	12.1	29.0	27.2	43.0	59.9	105.4
`googleplus`	5.1	27.2	21.2	18.0	56.4	82.2
`ogbl-ppa`	4.1	18.0	24.4	14.7	37.2	94.6
`hollywood`	4.4	22.6	24.9	15.6	46.6	96.7
`pokec`	3.0	10.5	11.2	10.7	21.8	43.6
`ogbn-products`	3.1	5.0	20.6	11.0	10.3	79.9
Geomean	4.7	16.0	20.8	16.6	33.1	80.7
Overall Geomean	4.1	16.8	16.7	14.0	33.2	64.5

respectively; also 4.9× and 2.4× higher in bandwidth efficiency. On the Transformer datasets, the numbers are 3.8× and 0.8× in throughput, 4.1× and 1.4× in bandwidth efficiency, respectively.

Table 4: Comparison with ThunderGP.

Dataset	Throughput		Bandwidth efficiency	
	ThunderGP	HiSparse	ThunderGP	HiSparse
`mouse-gene`	8.4	27.2	109.0	105.4
`hollywood`	9.7	24.9	126.0	96.7
`pokec`	8.7	11.2	113.7	43.6
Geometric mean	8.9	19.6	116.0	76.3

Table 5: Comparison with Vitis Sparse Library.

Dataset	Throughput		Bandwidth efficiency	
	VSL	HiSparse	VSL	HiSparse
`transformer-50`	17.5	20.6	65.2	79.8
`transformer-60`	14.6	17.8	54.4	69.0
`transformer-70`	13.0	15.3	48.7	59.5
`transformer-80`	10.5	13.4	39.1	51.9
`transformer-90`	5.8	10.6	21.8	41.0
`transformer-95`	3.3	5.1	12.4	19.7
Geometric mean	9.4	12.6	34.9	48.9

Table 4 shows the comparison with ThunderGP [24]. HiSparse delivers a 2.2× higher throughput than ThunderGP, but with a lower bandwidth efficiency (at 0.7×). The main reason is ThunderGP assigns more processing engines to one memory channel than HiSparse, compensating for the under-utilization of PEs due to bank conflicts and load imbalance. This approach is feasible in ThunderGP since the total number of memory channels is only 4. When scaled to 16 or more channels, the complexity of the shuffle unit can easily cause routability problems. Therefore, to further increase the bandwidth efficiency, a more lightweight shuffle unit is required to assign more PEs to one memory channel.

Table 5 shows the comparison with VSL. Since the SpMV in VSL is a floating-point design using the partial buffer approach, we also use the partial-buffer floating-point design for a fair comparison. HiSparse is 1.4× higher in both throughput and bandwidth efficiency. The main reason is that the VSL SpMV only assigns 4 PEs to one HBM channel. In addition, the vector buffer size and output buffer size are only 2 KB and 8 KB, respectively. Using the smaller buffers also increases the tiling and synchronization overhead.

Table 6: Power consumption and energy efficiency.

	MKL 32 threads	cuSPARSE	HiSparse
Power (W)	276	153	45
Energy efficiency (GOPS/W)	0.01	0.10	0.37

Table 6 shows the real-measured power consumption and energy efficiency of MKL, cuSPARSE, and HiSparse. HiSparse is 37× and 3.7× more energy-efficient than MKL and cuSPARSE.

6.4 Floating-Point vs. Fixed-Point

Table 7: Fixed-point (FX), partial buffer (PB) floating-point, and row interleaving (RI) floating-point designs.

Dataset	Size	Throughput		
		FX	PB	RI
`transformer-80`	512 × 33K	14.8	13.4	6.3
`mouse-gene`	45K × 45K	27.2	25.0	13.1
`pokec`	1632K × 1632K	11.2	3.4	9.1
`ogbn-products`	2449K × 2449K	20.6	6.7	16.3

Table 7 shows the comparison among the fixed-point design and two floating-point designs proposed in Section 5 with similar total on-chip buffer utilization. The operating frequency of the partial buffer (PB) design and the row interleaving (RI) design are 218 MHz and 206 MHz, respectively. On small matrices such as `mouse-gene` and `transformer-80`, the PB design is comparable to the fixed-point design, while the RI design only achieves less than 50% of the throughput of the fixed-point design. However, on large datasets, the increased tiling overhead of the PB design significantly degrades the performance. The RI design, on the other hand, achieved 80% throughput of the fixed-point counterpart. The results clearly show that adopting fixed-point delivers the best performance with high hardware efficiency. The partial buffer design is better at processing small matrices while the row interleaving approach is suitable to handle larger datasets.

6.5 Preprocessing Cost

Preprocessing refers to converting a CSR matrix into our custom format. Table 8 shows the preprocessing cost of HiSparse and VSL, both using only one thread. On 8 out of the 12 datasets, the preprocessing can finish within a second. Even on the largest dataset, `ogbn-products`, the preprocessing can finish within 11 seconds. In comparison, the preprocessing of VSL on `ogbn-products` takes 49 seconds. For graph analytics such as PageRank, the preprocessing cost is amortized over multiple iterations; for compressed machine learning inference like Transformers, the preprocessing cost can be ignored since a trained model runs inference for a long time, at least days, before being retained.

7 DISCUSSION

Extending HiSparse beyond SpMV. HiSparse can be easily extended to other sparse linear algebra operators such as SpMSpV and SpMM, by reusing the optimized hardware modules.

SpMSpV exploits the sparsity in the input vector and only loads the necessary columns of the sparse matrix. We can accelerate SpMSpV using similar architecture as HiSparse with small modifications.

Table 8: Preprocessing time with one thread.

Dataset	Time (s)		Dataset	Time (s)	
	HiSparse	VSL		HiSparse	VSL
transformer-50	0.24	7.72	mouse-gene	0.87	20.58
transformer-60	0.16	6.18	googleplus	0.39	8.09
transformer-70	0.11	4.38	ogbl-ppa	1.89	17.24
transformer-80	0.08	2.80	hollywood	4.68	59.40
transformer-90	0.04	1.66	pokec	3.43	13.00
transformer-95	0.02	0.74	ogbn-products	10.60	49.45

The vector loader assigns different vector values to different clusters instead of duplicating the same vector value. The result draining unit merges partial results from clusters by addition rather than concatenation. The clusters compute the scalar-vector product between one vector value and the corresponding column from the matrix. Since the accesses to the output buffer are random, we can reuse the shuffle unit to assign matrix non-zeros to PEs and resolve the bank conflicts on the output buffer. The compute pattern in the PEs is not changed, so the PEs can also be reused.

SpMM can be expressed in a batch of SpMV, therefore it has the same access pattern as SpMV. The rows of the sparse matrix are streamed in and duplicated to multiple SpMV instances. The vector loaders of different SpMV instances load different columns from the input dense matrix. Each SpMV instance generates one column of the result dense matrix.

Potential Enhancements in HLS Tools. Based on the lessons learned from developing HiSparse, we give the following suggestions on how HLS tools can be improved to better support developing high-performance sparse accelerators. First, sparse linear algebra operators all contain random access patterns with data reuse, which requires efficient on-chip buffering. We suggest hardware templates similar to the shuffle unit be included in the HLS libraries. Second, sparse linear algebra operators often imply accumulation. The load-store forwarding in HiSparse can serve as a general approach to resolving inter-iteration carried dependencies. The HLS tools can offer pragmas or options to implement the IFWQ and the dependence resolution logic at compile time. Finally, when any module consumes resources more than one die, the HLS tool can automatically switch to pipelined communication protocols and insert registers or at least warn the programmer of potential timing degradation.

8 RELATED WORK

Sparse Formats and Sparse Accelerators. There is an active body of research on accelerating sparse linear algebra operators [5, 28–34]. The cyclic channel interleaving scheme in our customized format is adopted from cyclic channel sparse rows (C^2SR), a format proposed for a sparse-sparse matrix multiplication (SpGEMM) accelerator [29]. One major difference between C^2SR and our format is that C^2SR performs vectorized memory accesses to every single row, while our format performs vectorized memory accesses to packed rows. The latter better exploits the parallelism in SpMV. Our format also draws inspiration from compressed interleaved sparse rows (CISR), a format proposed for an SpMV accelerator [5]. Our format borrows from CISR the general idea of explicitly encoding parallelism into the sparse matrix format, but avoids the centralized row encoding/decoding in CISR. Therefore we achieve higher throughput when scaling to multiple HBM channels and lower preprocessing cost.

Graph Accelerators on FPGAs. ThunderGP [24] implements SpMV using the programming model for graph algorithms. ThunderGP provides optimized kernels with high bandwidth efficiency and frequency, but the performance is limited by DDR memory bandwidth. GraphLily [34] formulates graph algorithms with SpMV in one unified FPGA bitstream. Although utilizing HBM, GraphLily only operates at 165 MHz, which limits its performance. We believe the techniques mentioned in this paper, especially kernel splitting, can be applied to GraphLily to further improve its performance.

Vitis Sparse Library. To the best of our knowledge, the Vitis Sparse Library (VSL) [25] is the only work prior to HiSparse that accelerates SpMV on a multi-die HBM-equipped FPGA. It adopts compressed sparse column format to exploit the parallelism across columns, with the floating-point data type. Our performance gain over VSL comes from the increased number of PEs and buffer size. VSL utilizes combinational arbiters to resolve bank conflicts so that the numbers of PEs and shared banks are limited to 4 for high frequency. The compute pattern of VSL requires the results from different PEs to be added instead of concatenated, and it follows the partial buffer approach to handle floating-point. These two factors significantly decrease the on-chip buffer utilization efficiency.

Timing Optimization of FPGA HLS. There are several prior attempts aiming at improving the frequency of the HLS designs by considering the physical information at an early stage. AutoBridge [21] proposes floorplan-guided pipelining for HLS dataflow designs to achieve substantial timing improvements on multi-die FPGAs. AutoBridge does not support pipelining the control signals and the remote accesses to HBM; hence currently it cannot be applied to our sparse accelerator design. Guo et al. [35] propose to pipeline the high-fanout signals generated by HLS, although this work does not target multi-die HBM-equipped FPGAs. Zheng et al. [36] propose an iterative HLS flow that incorporates place-and-route (PAR) to gradually optimize the critical paths. For large-scale designs that saturate the HBM bandwidth, the long running time of PAR would undermine the productivity benefits of HLS.

9 CONCLUSION

This paper proposes HiSparse, a high-performance SpMV accelerator on HBM-equipped FPGAs. We present techniques to tackle challenges in four aspects — HBM bandwidth utilization, on-chip memory utilization, compute occupancy, and timing closure on multi-die heterogeneous fabrics. Evaluation results verify the advantages of HiSparse over competitive CPU, GPU, and FPGA baselines in throughput and bandwidth efficiency. We further discuss how to extend HiSparse to support sparse linear algebra operators beyond SpMV, such as SpMSpV and SpMM. Our study also provides guidance on potential enhancements in HLS tools to better support developing high-performance sparse accelerators.

ACKNOWLEDGEMENTS

This research was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, NSF Awards #1453378, #1909661, and research gifts from Xilinx. We thank the anonymous reviewers for their feedback on earlier versions of this manuscript.

REFERENCES

[1] Paul Grigoras, Pavel Burovskiy, Eddie Hung, and Wayne Luk. Accelerating SpMV on FPGAs by compressing nonzero values. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2015.

[2] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical foundations of the GraphBLAS. *IEEE High Performance Extreme Computing Conf. (HPEC)*, 2016.

[3] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

[4] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. Programming and synthesis for software-defined FPGA acceleration: status and future prospects. *ACM Trans. on Reconfigurable Technology and Systems*, 2021.

[5] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S Chung, and Greg Stitt. A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2014.

[6] Srinidhi Kestur, John D Davis, and Eric S Chung. Towards a universal FPGA matrix-vector multiplication architecture. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2012.

[7] Ling Zhuo and Viktor K Prasanna. Sparse matrix-vector multiplication on FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2005.

[8] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. *Int'l Symp. on Microarchitecture (MICRO)*, 2020.

[9] Sam Skalicky, Christopher Wood, Marcin Łukowiak, and Matthew Ryan. High level synthesis: where are we? A case study on matrix multiplication. *Int'l Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, 2013.

[10] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, et al. Susy: A programming model for productive construction of high-performance systolic arrays on FPGAs. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2020.

[11] Jason Cong and Jie Wang. PolySA: Polyhedral-based systolic array auto-compilation. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018.

[12] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. Rosetta: A realistic high-Level synthesis benchmark suite for software-programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[13] Jiajie Li, Yuze Chi, and Jason Cong. HeteroHalide: From image processing DSL to efficient FPGA acceleration. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2020.

[14] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. SODA: stencil with optimized dataflow architecture. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[15] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[16] Yifan Yang, Qijing Huang, Bichen Wu, Tianjun Zhang, Liang Ma, Giulio Gambardella, Michaela Blott, Luciano Lavagno, Kees Vissers, John Wawrzynek, et al. Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2019.

[17] Yichi Zhang, Junhao Pan, Xinheng Liu, Hongzheng Chen, Deming Chen, and Zhiru Zhang. FracBNN: Accurate and FPGA-efficient binary neural networks with fractional activations. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[18] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Conf. on Neural Information Processing Systems (NeurIPS)*, 2017.

[20] HBM3: Cheaper, up to 64GB on-package, and terabytes-per-second bandwidth. https://arstechnica.com/gadgets/2016/08/hbm3-details-price-bandwidth/.

[21] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. AutoBridge: coupling coarse-grained floorplanning and pipelining for high-frequency HLS Design on multi-die FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[22] AMBA AXI4 interface protocol. https://www.xilinx.com/products/intellectual-property/axi.html#overview.

[23] Guoqing Lei, Yong Dou, Rongchun Li, and Fei Xia. An FPGA implementation for solving the large single-source-shortest-path problem. *IEEE Trans. on Circuits and Systems II: Express Briefs*, 2015.

[24] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. ThunderGP: HLS-based graph processing framework on FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[25] Xilinx. Vitis sparse library. https://xilinx.github.io/Vitis_Libraries/sparse/2021.1/overview.html.

[26] CSCMV test does not run for all matrices. https://github.com/Xilinx/Vitis_Libraries/issues/55.

[27] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[28] Dong-Hyeon Park, Subhankar Pal, Siying Feng, Paul Gao, Jielun Tan, Austin Rovinski, Shaolin Xie, Chun Zhao, Aporva Amarnath, Timothy Wesley, et al. A 7.3 M output non-zeros/J, 11.7 M output non-zeros/GB reconfigurable sparse matrix–matrix multiplication accelerator. *IEEE Journal of Solid-State Circuits (JSSC)*, 2020.

[29] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. *Int'l Symp. on Microarchitecture (MICRO)*, 2020.

[30] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C Hoe, Larry Pileggi, and Franz Franchetti. Efficient SPMV operation for large and highly sparse matrices using scalable multi-way merge parallelization. *Int'l Symp. on Microarchitecture (MICRO)*, 2019.

[31] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2018.

[32] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. Sparch: Efficient architecture for sparse matrix multiplication. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2020.

[33] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2020.

[34] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2021.

[35] Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. Analysis and optimization of the implicit broadcasts in FPGA HLS to improve maximum frequency. *Design Automation Conf. (DAC)*, 2020.

[36] Hongbin Zheng, Swathi T. Gurumani, Kyle Rupnow, and Deming Chen. Fast and effective placement and routing directed high-Level synthesis for FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2014.