Accelerating SpMV on FPGAs through block-row compress: a task-based approach

José Oliver*†, Carlos Álvarez*†, Teresa Cervero*, Xavier Martorell*†, John D. Davis*, Eduard Ayguadé*†
*Barcelona Supercomputing Center, Barcelona, Spain
†Universitat Politècnica de Catalunya, Barcelona, Spain
E-mail: {jose.oliver, carlos.alvarez, teresa.cervero, xavier.martorell, john.davis, eduard.ayguade}@bsc.es

***Abstract*—Sparse Matrix-Vector multiplication (SpMV), computing $y = \alpha \cdot A \times x + \beta \cdot y$ where y, x are dense vectors, α, β two scalar constants, and A is a sparse matrix, is a key kernel in many HPC applications. It exhibits a kind of memory access that is extremely hard to perform efficiently, due to its random access. In this paper, we present a new approach to accelerate SpMV on FPGAs. As FPGAs lack a default memory hierarchy, they can adapt to specific applications better. Also, an increasing number of FPGAs include High Bandwidth Memory (HBM), making the SpMV problem especially appealing to tackle on these kind of devices. We define a new sparse matrix encoding format (*b8c*) and its corresponding SpMV implementation using OmpSs@FPGA and HLS. This format allows us to leverage many of the FPGA strengths for intensive data processing, such as data streaming, customizable datapaths widths, parallel memory access for off-chip memory in the case of multiple memory channels (like in HBM), parallel memory access for on-chip memory and pipelining. We tested our proposal for both DDR and HBM memories to show the adaptability and scalability of our design. The presented *b8c* SpMV implementation is able to achieve higher performance than the state-of-the-art FPGA implementation of SpMV over all the matrices in the data set, achieving 3.52x performance on average with a minimum of 1.82x and a maximum of 6.28x even when running at 75% the frequency.**

***Index Terms*—FPGA, SpMV, HBM, HLS, FP64, double-precision, High-performance computing, Sparse matrix representation.**

I. Introduction

In recent years, we have witnessed how Moore's Law is getting harder and harder to achieve. Clock frequencies and integration in microprocessor manufacturing are reaching a physical limit that prevents performance improvements as the ones we have observed during the previous decades. This scenario has led to the emergence of what we know as "accelerators": compute devices that are tailored for specific kinds of workloads, to which a microprocessor can offload parts of the execution. Field Programmable Gate Arrays (FPGAs) can be seen as blank canvases of unconfigured hardware, memories and I/O resources that can be connected in order to implement logic functions in hardware and reconfigured as many times as needed. The need for specialized accelerators combined with the flexibility of FPGAs has made them an interesting technology for developing such accelerators.

Sparse Matrix-Vector multiplication (SpMV) is a key kernel in a wide range of applications: from scientific computing (linear algebra solvers) [1] to artificial intelligence (sparse neural networks) [2] or graph processing [3], and is of such importance that it has been included among the seven dwarfs of parallel computing research (Sparse Linear Algebra) and is a core kernel in the HPCG benchmark [4]. SpMV exhibits a kind of memory access that is extremely hard to fulfill efficiently, due to its random access. In the case of FPGAs, which lack a "default" memory hierarchy including caches to hide the latency of main memory, obtaining good performance when running SpMV is especially challenging. However, for large matrices, caching can be arguably ineffective without further cache optimization. In this sense, FPGA's ability to customize the memory hierarchy is an interesting feature compared to conventional architectures.

In this paper, we present a new approach to accelerate the computation of the SpMV operation on FPGAs, specially, but not exclusively, using HBM [5], [6]. We define a new, FPGA-friendly, sparse matrix encoding format (*b8c* - block-8-compress) and its corresponding SpMV implementation using OmpSs@FPGA [7], a directive-based programming model that resembles OpenMP tasking [8], [9] originally based on OmpSs [10]. Our implementation targets the general case of SpMV ($y = \alpha \cdot A \times x + \beta \cdot y$) and does not restrict the size of the source or destination vectors and makes no assumption about the sparsity pattern of the matrix. Specifically, it assumes a worst-case scenario where neither the matrix data nor the source/destination vectors fit into on-chip memory (SRAM, BRAM/URAM) and, thus, the only limitation on the data size is the off-chip memory (DRAM) capacity. *b8c* SpMV implementation allows us to leverage many of the FPGA strengths for intensive data processing, such as stream processing, customizable datapaths widths, parallel memory access for off-chip memory in the case of multiple memory channels (specially in the case of HBM), parallel memory access for on-chip memory (via BRAM/URAM partitioning) and pipelining.

We have focused our implementation on the FP64 data type (double-precision IEEE), but both the *b8c* algorithm and the matrix representation could be easily adapted to support narrower data types. This approach has been implemented entirely in C++ on top of OmpSs@FPGA and HLS and tested on a Xilinx Alveo U280 FPGA making use of its 32 HBM memory channels and, without modifications, on a Xilinx Alveo U200 FPGA using its 4 DDR memory chips, which shows the adaptability and scalability of our design. In

 DOI: 10.1109/FPL60245.2023.00029

order to evaluate the performance of our approach, we have compared it with Xilinx's own SpMV implementation for the Alveo U280, which is included in the vendor's BLAS Vitis Accelerated Libraries [11].

II. Background and motivation

A. Double-precision SpMV on FPGAs with HBM

Vitis Sparse Library's double-precision SpMV (VSpMV) [12] is, to the best of our knowledge, the only performance-oriented 64-bit floating point implementation of SpMV on FPGAs equipped with HBM. However, unlike our design, even VSpMV architecture, as is, is not able to perform the generic form of SpMV ($y = \alpha \cdot A \times x + \beta \cdot y$) and does not fully exploit available HBM bandwidth. Other works have experimented with double-precision SpMV implementations on FPGAs with HBM [13], [14]. In both cases, however, their main goal is to evaluate different aspects of memory bandwidth management, not to improve SpMV arithmetic performance.

Zhuo et al. [15] designed a double-precision SpMV accelerator but it presented some limitations: they were not targeting HBM, the proposed architecture did not implement the generic form of SpMV and it was limited by the size of the y vector, which was computed and stored entirely on on-chip memory. Fowers et al. [16] proposed an SpMV accelerator on FPGA, but it was targeting DDR and single-precision arithmetic. Besides, the design was limited to problem sizes where x could fit into the aggregate on-chip vector buffer. Grigoras et al. [17] proposed an optimized SpMV. However, it is domain-specific, targeting dense block diagonal sparse matrices. Jain et al's SpMV design [18] is also demonstrated only in single-precision arithmetic and DDR memory, and only with problem sizes that fit into the on-chip buffers. Kestur et al. [19] proposed a generic architecture for SpMV that was able to perform on-the-fly conversion from different matrix input formats. This design, however, relied on DMA and a run-time coalescing and cache mechanism for x memory accesses, which results in sub-optimal memory performance compared to burst memory access, while consuming more hardware resources.

Serpens [20] and HiSparse [21] are two state-of-the-art SpMV accelerators on FPGAs and both target HBM. In both cases, however, the data type of their architecture is either fixed point or single-precision floating point. None of them has been demonstrated in dual-precision floating point, a change which is not trivial to perform, since it implies deep modifications on the overall architecture that can result in substantial changes in performance [22].

GraphLily [23] and ThunderGP [24] are FPGA accelerators for graph processing that can implement, but are not specialized in, SpMV. In both cases, however, they utilize DDR memory and single-precision or integer arithmetic. Sadi et al. [25] proposed a 2-way SpMV algorithm. It is also very graph-oriented and their main target is ASIC; their FPGA implementation limits maximum problem sizes and uses simulated HBM characteristics.

B. Sparse matrix representation formats

Sparse matrix representation formats on microprocessors (either standard, SIMD or Vector), such as CSR [26], ELL(PACK), CSR2 [27] or SELL-C-σ [28] don't work well for FPGA implementations, neither do GPU formats [29]: all of them assume an efficient cache hierarchy plus a mechanism for a relatively high number of outstanding, non-blocking, memory requests, which are hard to implement in high-level synthesis (HLS).

There have been different proposals for FPGA-specific matrix encodings and algorithms [16], [20], [21], [23], [25]. Some of these even allow on-the-fly transformation from multiple formats inside the FPGA [19]. Most of these FPGA-specific format proposals, however, target low-level kernel implementations using transfer-level languages (RTL) that are hard to customize and include in high-level synthesis by non-expert users. In many cases, they also require complex architectures (e.g. cache hierarchies, banked buffer with arbitration) that consume a high number of resources and introduce stalling mechanisms, making them difficult to scale in number and limiting performance.

C. Motivation

The lack of double-precision, HBM-focused, SpMV implementations for FPGAs and the limits of the existing implementations (such as not implementing the generic SpMV form or the underuse of HBM bandwidth) motivates us to develop *b8c* and address all those limitations.

III. b8c Matrix Encoding

A. b8c (Co)design constraints

Many good practices need to be followed in order to fully exploit FPGAs capabilities [30]. The most relevant ones that we consider for our co-design process are: a) fully pipelined design, ideally with an initiation interval (II) of 1, b) maximize the use of on-chip memory, c) maximize datapath width to off-chip memory, d) reduce the number of independent off-chip memory requests and use burst accesses as much as possible in order to reduce latency penalties, e) maximize parallelism at the low level (processing element): process as many elements per cycle as possible, f) easily parallelizable at the high (matrix) level in order to fully exploit multi-channel memory systems and/or be easily partitioned into tasks among different OmpSs@FPGA accelerators instances.

Besides these architectural considerations, additional constraints apply, specific to the target workloads. The size of the problems involved in HPC in the last years (and, thus, the size of the sparse matrices within them) has been growing [19]. Considering this fact, and our will to develop an SpMV algorithm not limited to a specific matrix size or pattern, additional constraints can be added to our co-design process: g) the size of the data structures involved in the SpMV process will be limited only by off-chip DRAM, h) assume that M does not fit into on-chip SRAM, i) assume that neither x nor y vectors fully fit into on-chip SRAM, j) assume x vector indexes in the original (CSR) representation to be 64 bits

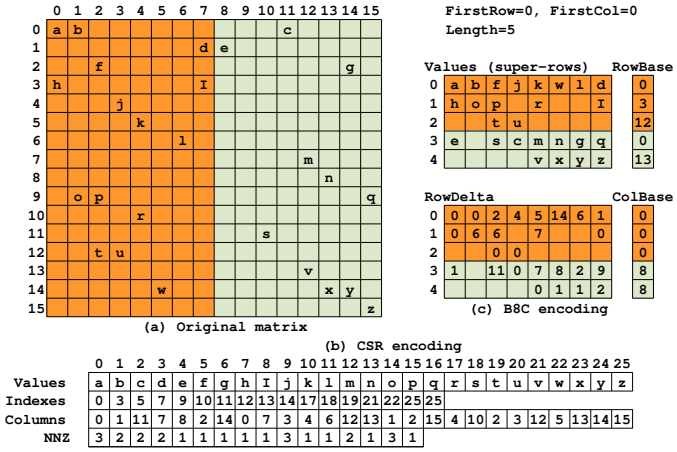


Fig. 1: CSR and b8c encodings for a sparse matrix

integers, k) assume values in the sparse matrix and dense vectors to be double-precision floating point (64 bits IEEE). The four last points are also enforced by the context in which we are developing this implementation: the porting of the HPCG benchmark into HBM FPGAs. The last two, however, as we will see later, could be parametrized in order to adapt our implementation to narrower data types.

B. The b8c matrix format representation

Figure 1.a includes an example of a matrix sub-block of size 16×16 that we will use to illustrate the steps in the *b8c* transformation. Figure 1.b shows the CSR encoding that would correspond to that subblock.

1) Blocking on x and y: In the first place, assuming neither x nor y will fit into on-chip SRAM, we decided that a blocking process of the matrix would be a good approach. This enables prefetching to load x values needed to process a sub-block of the matrix. Blocking on y allows also us to use the same technique for that vector.

2) M rows compression and the idea of super-rows*:* For each matrix sub-block, we divide it into vertical stripes of contiguous columns of size C. In the case of *b8c*, $C = DW/Be = 8$, where DW is the maximum data path width from off-chip memory (512 bits in the case of the Alveo boards) and Be is the size of each element of the matrix (64 bits). For each stripe we apply an iterative process: for each row of C elements in that stripe, we try to merge it with any of the previous rows. The conditions for two of these rows to be merged are that they are, at most, at a (parametrizable) row-distance l and that they are *disjoint*: they do not share any non-zero element in the same position (column) and $r_i \ (mod \ R) \not\equiv r_{j_k} \ (mod \ R)$, where r_i is the row index corresponding to the row being merged and r_{j_k} are the row indexes corresponding to each row k already merged into row j (the only exception is when $r_i = r_{j_k}$ for any k). As x and y block sizes, and l distance, R is an input parameter to *b8c* that defines the maximum number of matrix rows that can be modified in parallel in each accelerator at each cycle. Once a stripe has been processed, the transformation algorithm continues with the next one. Empty sub-blocks or stripes are, of course, not considered.

Each row resulting from this *compression* process is what we name a *super-row*, and they constitute the foundations of the *b8c* encoding: data structures able to enclose matrix values coming from different, conflict-free, rows and columns in the original matrix. Figure 1.c includes a set of *super-rows* grouped in the ”Values” data structure.

3) b8c *metadata:* Each super-row has a corresponding set of metadata associated to it: row/column information for each element *relative* to the first row/column of its block. As the current implementation forces elements in a super-row to have consecutive column indexes, only the column information for the first element is kept. Finally, for each sub-block, we need pointers to the starting indexes in y and x and the number of *super-rows* resulting from that sub-block. Figure 1 shows an original sparse matrix (a) represented in both CSR (b) and *b8c* (c), with (c) including the matrix values in the ”Values” structure and a set of metadata associated to each *super-row* (”RowBase”, ”RowDelta”,” ColBase”) and to the whole sub-block (”FirstRow”, ”FirstCol”, ”Length”). New metadata indexes are now relative to their subblock, allowing us to reduce the storage needed for them. For a 8192 block size, only 13 bits would be needed. Being SpMV a well-known memory-bound problem, this fact helps alleviate memory pressure, since it reduces the bytes/flop ratio. Our current implementation, however, does not fully exploit this possibility, and fixes 16 bits as the size for them, allowing theoretical block sizes up to 64K elements. With that setup, the theoretical metadata/data ratio that can be achieved is 3/16 (3 rows of 64 bytes of metadata for every 16 rows of 64 bytes of values), which translates to ∼84% of the data transferred for the matrix being useful data and a peak limit of 4.75 bytes/flop (versus 8 bytes/flop for a naive CSR implementation). Our current implementation, again, does not fully exploit this possibility and uses a 5/16 ratio, which corresponds to ∼76% of useful data transfers for M in *b8c* encoding, resulting in a peak limit of 5.25 bytes/flop.

4) b8c *data layout in off-chip memory:* In order to be able to process all matrix data in *streaming* mode (since it is single-use data), *b8c super-rows* are stored contiguously in memory. Since we also need the corresponding metadata to process each *super-row*, our implementation keeps that metadata combined with its corresponding data. Having this data layout allows us to read *b8c* data and metadata using burst mode, reducing the number of read requests and the associated latency. The algorithm implementation (and the corresponding architecture) takes care of decoupling data from metadata and feeding each one to the appropriate component of the hardware architecture. We chose to combine data and metadata in a single stream instead of having two dedicated memory ports to separated data that could be read in parallel not only because it is easier to place&route, but also because using two AXI channels for two combined-data accelerators offers a peak performance 52% higher than a single accelerator with parallel AXI channels for data and metadata

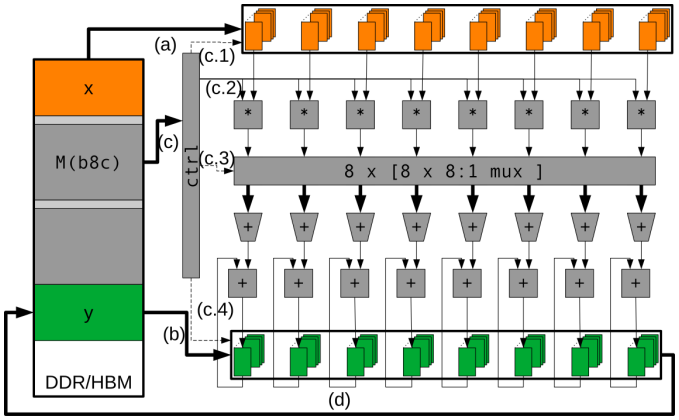


Fig. 2: b8c SpMV hardware diagram

$(speedup = \frac{2 \times T_{combined}}{T_{parallel}} = \frac{2 \times T_{parallel} \times 0.76}{T_{parallel}} = 1.52)$.

C. Task partitioning in OmpSs@FPGA using b8c

Our implementation leverages multiple HBM (or DDR) channels by placing, ideally, as many accelerators in the FPGA as channels are available, and mapping them 1:1. Each subblock of the matrix (in its *b8c super-rows* representation) is considered a task in OmpSs@FPGA, so it will be executed by an accelerator that will take care of all the needed processes. Moreover, in our case, in order to benefit from locality on y, we instruct OmpSs@FPGA (using the **onto** clause) to schedule all tasks updating the same segment of y on the same accelerator.

For assigning rows to accelerators, instead of computing the number of rows per accelerator, we compute the number of *nnz* (non-zero elements) per accelerator, as the number of total non-zero elements in the matrix divided by the number of accelerators we want to use to partition the work [31], [32]. This policy, called "adaptative nnz" in *b8c*, results in better balanced workloads in cases where the structure of the matrix is not naturally balanced.

IV. B8C ACCELERATOR ARCHITECTURE AND OPERATION

Figure 2 depicts the (simplified) hardware block diagram for the *b8c* SpMV algorithm implementation. It is important to remember that this hardware is inferred by the synthesis tool, after OmpSs@FPGA process and transformations, from ∼375 lines of C++ code, OmpSs@FPGA and HLS pragmas. We will use the simplified representation of a transformed matrix, corresponding to a single *b8c* block, depicted on the left hand side of the same figure, in order to illustrate how data/metadata usage occurs.

A. x and y prefetching

For each task, the corresponding segments of y and x are loaded into on-chip memory (steps a and b) before processing the elements of the sub-matrix. This process also takes care of skipping the data load if it is already present in the local memory of the accelerator due to previous tasks executions. Since x and y are the only reused data when processing a matrix sub-block, prefetching them into on-chip memory reduces the number of memory requests to off-chip memory. We also benefit from the fact that values for each x and y segments will be accessed with lower latency due to on-chip SRAM access when processing each matrix block (ideally, at 1-cycle latency). Leveraging FPGAs' ability to define partitioned memory structures, we instruct the HLS toolchain to use a *cyclic* partition of factor F_x for the on-chip memory for x ($F_x = \frac{C}{2}$) and also for y ($F_y = R$), assuming dual-port on-chip memory. Thanks to memory partitioning, the off-chip to on-chip load process can be performed using the maximum bandwidth available: C elements per cycle after an initial latency due to off-chip memory access.

B. computation phase

Computation is implemented using *dataflow*. *b8c* values and metadata are read using burst reads from a single, combined, data stream in memory (in Figure 2, light grey corresponds to metadata rows while dark grey corresponds to matrix values rows). At each computation cycle, due to the pipelined design of the accelerator (II=1), the following operations take place:

1) $A \times x$ partial products computation: Metadata of each super-row is used in (c.1) to select the x values (matrix columns) that must be multiplied by the corresponding elements of the matrix which the accelerator is streaming from memory (c.2). Due to the partitioned memory and conflict-free design of *b8c*, C elements of x can be read in parallel to compute C partial products per cycle.

2) Partial results grouping and addition: Partial products are routed into an interconnect that allows the accelerator, using again the meta-data information (c.3), to group values from partial results that belong to the same original row in order to add them using a set of R adder trees. In the example, eight accumulator trees are used, since we assume the *super-rows* in this example can hold values belonging to up to eight different rows in the original matrix, so any sub-product computed up to that point can be routed to any of the eight accumulator trees.

It is worth mentioning that advanced transformations like *b8c* naturally change the order in which accumulation over y is performed. This fact, combined with the adder tree reduction and small differences in each vendor's implementation of floating-point operations, may lead to slightly different results due to rounding errors. This needs to be considered when checking results (comparing against the *golden* reference, for example by using relative errors instead of absolute errors). This fact may result also, in the case of iterative solvers, in a different number of iterations to converge.

3) y update: Once the adder tree phase is finished, the results are used to update the corresponding positions of y, using the row indexes in the meta-data (c.4). This is a pipelined floating point operation which takes more than one cycle to execute (actually, it can take L, depending on the frequency, since it is implemented using a soft IP that is being pipelined), and that can generate carried dependences. During the *b8c* transformation phase, our algorithm checks, when merging a

new row into a *super-row* that no other *super-row* at a distance L updates the same position of y. If the algorithm detects that situation, it adds zero padding rows to the transformed data. Again, *b8c* design and implementation allow reading/updating R elements of y in parallel at each cycle.

C. y write-back

Once the whole sub-matrix has been processed, if no further processing needs to be done in the same segment of y, it gets written back to memory. As in the case of x and y preload, the partitioned memory plus the segmented design allows the accelerator to store C elements per cycle.

V. Experimental Results

In this section, we present an analyze the results achieved by *b8c*, comparing them to those obtained by the vendor's SpMV reference implementation using HBM [11].

A. Experimental setup

Our implementation has been tested on an Alveo U280 from Xilinx equipped with 8GB of HBM2 memory accessible through 32 parallel channels. The FPGA is installed in an AMD server running Linux (Ubuntu 20.04, kernel version 5.8.0-38-generic). The server's CPU is an AMD Ryzen 7 3800X 3.9GHz AM4 (8 cores/16 threads), and it is equipped with 4×32GiB DDR4 3200 MHz 64 bits (Kingston HyperX Fury). For *b8c*, the host binary is compiled with OmpSs@FPGA 3.3.0 and g++ 9.4.0, while the bitstream is generated using Vivado 2020.1. VSpMV host binary and bitstream are generated using Vitis 2021.1. The theoretical maximum throughput of HBM memory on the Alveo U280 is 460.8 GiB/s. The U280 is connected to the AMD host via PCIe. Since we are focusing on large iterative problems that will be completely off-loaded into FPGAs, we have decided to ignore memory transfers and measure performance exclusively of in-FPGA computations. This also allows us to make a fair comparison with VSpMV, since in their benchmarks, PCIe transfer time is also ignored, and so is matrix transformation time, which is assumed to be amortized in large iterative problems [4], [33].

B. AXI bonding for b8c in the Alveo U280

We designed *b8c* to be able to process 512 bits per cycle. The HBM in the Alveo U280 FPGA offers 32 channels with a native port width of 256 bits that can yield 512 bits per cycle when clocked at twice the frequency of the user logic. In order to reduce pressure on the place&route algorithm and also foreseeing a possible scalability issue due to the small size of some matrices in the VSpMV test data set, we opted to implement AXI bonding. This solution bonds together two 256 bits channels of HBM into a single 512 bits channel. This new "virtual" U280, now, offers us 16 AXI channels of 512 bits running at the same frequency as the user logic, which is an exact match with our *b8c* design. Using AXI bonding, we can reduce from 32 to 16 the maximum number of HBM channels while keeping the original bandwidth available. We will come back to this in the results evaluation section.

Board	Design	LUT	FF	DSP	BRAM	URAM
U280	VSpMV	28.1%	22.7%	10.0%	20.6%	13.3%
	b8c-8acc	30.6%	18.8%	15.0%	16.6%	7.4%
	b8c-10acc	37.0%	22.7%	18.7%	18.9%	9.1%
	b8c-16acc	55.1%	34.7%	29.9%	25.9%	14.1%
U200	b8c-4acc	26.7%	16.7%	10.2%	18.3%	4.1%
U280e	b8c-4acc	24.2%	15.5%	7.7%	19.6%	4.1%

TABLE I: Resource utilization. VSpMV and b8c-4acc @333MHz, b8c-{8,10,16} @250MHz. U280e represents extrapolated U280 resource usage: $\frac{resources\_used\_U200}{resources\_available\_U280}$.

C. VSpMV / b8c architecture comparison

VSpMV design is composed of 24 Compute Units (CUs) and uses 20 256-bit HBM channels. 16 CUs are devoted to computing 4 double-precision $M \times x$ products in parallel per cycle, each one. 3 CUs are in charge of accumulating row values and assembling/storing y using one HBM channel as output. 1 CU is in charge of loading and distributing row metadata using one HBM channel, and 3 CUs are in charge of loading x data and parameters using two HBM channels and distributing them to the 16 x multipliers. VSpMV pre-process matrices using python, dividing the input matrix into blocks of $N \times M$ (rows, columns), further subdividing each block into *HBM-channels* and stores the partitioned data on disk. We have been able to compile and generate the corresponding bitstream using Vivado, and we have used it to generate the performance numbers, which match those posted by Xilinx in the case of the original dataset. Without modifying the downloaded repository, VSpMV design is synthesized for a 333MHz clock frequency.

Our U280 design is composed of 16 accelerators (equivalent to Xilinx's CUs) devoted to *b8c* SpMV block computation (each one is connected to an AXI-bonded HBM channel), and one additional small accelerator in charge of creating tasks for each one of the *b8c* blocks that need to be processed. We placed AXI SmartConnects in front of HBM channels being shared by more than one CU/IP (QDMA IP and Scheduler CU plus 1 SpMV block CU in each case). The maximum block size for x and y in our case has been set to 8K elements. Using Xilinx's Vivado in order to synthesize our design, we have been able to obtain bitstreams running at 250MHz clock frequency. Resource usage for each design is depicted in Table I. 8 and 10 instance cases of *b8c* are included for completeness: all performance metrics for the U280 in this section were obtained using the 16 instances bitstream, since we can selectively use from 1 to 16 accelerators of it. We also show resource usage for a 4-instances, 8K blocksize, *b8c* implementation for an Alveo U200, clocked at 333MHz.

D. Dataset

We have taken the set of 22 matrices used by Xilinx in its benchmark, coming from the SuiteSparse Matrix Collection [34], and we have augmented it with 12 matrices generated using HPCG's matrix pattern. For the added matrices, we use different sizes and non-MPI/MPI structures, in order to be able to evaluate both size and pattern impact on performance. The

ID	Name	nrows	ncols	nnz	ID	Name	nrows	ncols	nnz	ID	Name	nrows	ncols	nnz
M1	bcsstk15	3948	3948	117816	M13	nasa2910	2910	2910	174296	M25	hpcg_24-1	13824	13824	343000
M2	bcsstk24	3562	3562	159910	M14	nasasrb	54870	54870	2677324	M26	hpcg_24-27	13824	17576	373248
M3	bcsstk28	4410	4410	219024	M15	nd3k	9000	9000	3279690	M27	hpcg_32-1	32768	32768	830584
M4	bcsstk36	23052	23052	1143140	M16	nd6k	18000	18000	6897316	M28	hpcg_32-27	32768	39304	884736
M5	bodyy4	17546	17546	121550	M17	olafu	16146	16146	1015156	M29	hpcg_48-1	110592	110592	2863288
M6	bodyy6	19366	19366	134208	M18	raefsky4	19779	19779	1316789	M30	hpcg_48-27	110592	125000	2985984
M7	cbuckle	13681	13681	676515	M19	s2rmq4m1	5489	5489	263351	M31	hpcg_64-1	262144	262144	6859000
M8	ct20stif	52329	52329	2600295	M20	s3rmt3m3	5357	5357	207123	M32	hpcg_64-27	262144	287496	7077888
M9	ex9	3363	3363	99471	M21	ted_B	10605	10605	144579	M33	hpcg_80-1	512000	512000	13481272
M10	gyro_k	17361	17361	1021159	M22	ted_B_unscaled	10605	10605	144579	M34	hpcg_80-27	512000	551368	13824000
M11	msc10848	10848	10848	1229776	M23	hpcg_16-1	4096	4096	97336					
M12	msc23052	23052	23052	1142686	M24	hpcg_16-27	4096	5832	110592					

TABLE II: Matrices dataset properties.

	b8c	Avg.	Max.	Min.
GFLOP/s	U200-4acc	1.39x/1.18x	2.82x2.51x	0.57x0.57x
	U280-8acc	2.16x/1.79x	3.66x/3.38x	0.98x/0.98x
	U280-10acc	2.56x/2.10x	4.07x/3.82x	1.22x/1.22x
	U280-16acc	3.52x/2.75x	6.29x/4.52x	1.82x/1.82x
GFLOP/Watt	U280-16acc	2.43x/1.86x	4.11x/3.22x	1.15x/1.15x

TABLE III: Improvement of *b8c* over VSpMV on the augmented dataset/original dataset. *b8c* U280 designs @250MHz, U200 designs @333MHz, VSpMV design @333MHz.

properties of the dataset are shown in Table II. We will show results in both, the augmented and the original dataset (M1-M22), in order to avoid the illusion of bias.

E. Performance results and analysis

In order to separate the contributions provided by OmpSs@FPGA versus *b8c*, we implemented a vanilla CSR SpMV on the same FPGA. Performance obtained by *b8c* version was $\sim$3 orders of magnitude better than the naive CSR implementation, thus we can state that performance of *b8c* shown in this section comes entirely from itself. Figure 3 shows the performance comparison between VSpMV and *b8c* for the U280 board in three specific cases: 8, 10, and 16 accelerators in the case of *b8c*. These three cases correspond, respectively, to the case with the same number of per-cycle $M \times x$ operations, the case with the same number of HBM channels used and the case with the maximum scalability of each implementation. As can be observed in Figure 3 and Table III, *b8c* improves VSpMV results in all the cases but one, including the 16 VS 8 scenario (in which the worst result is 0.98x, which can be considered on-par), even though our design runs at 75% the frequency of VSpMV. We also include the results obtained using our same implementation on an Alveo U200 using 4 DDR modules and consequently only 4 accelerators, and 4K blocksize. Even with those limitations, it achieves better performance than VSpMV for half the cases. Scalability problems observed in Figure 3 in the case of small matrices when a large number of accelerators is used, support our decision to use AXI bonding to use fewer, but wider, accelerators, while keeping the peak bandwidth.

Many differences make our design perform better than VSpMV. Useful data ratio in the transformed matrix in the *b8c* matrix encoding is higher than VSpMV's in many cases, meaning that data streamed from memory contains less padding and, thus, can be processed faster. *b8c* format, as explained, is designed to expose row and column parallelism inside each *super-row* without the need for arbitration and stalling, which VSpMV may require, since their partition scheme does not guarantee conflict-free x index access. The only alternative to be able to perform parallel access to x without that guarantee is to replicate buffers to store multiple copies of x [35], which would probably translate into smaller buffers in order to keep resource usage on limits and alleviate pressure on place&route. Actually, block size and block scheduling are also relevant in the comparison. The default implementation of VSpMV uses 4K elements as block size but (probably to maximize x reutilization), each block is further divided into 16 HBM-channels or subblocks (the number of CUs in charge of $M \times x$ multiplications). That translates into an effective block size per CU of 256×4096, increasing blocking/scheduling overhead.

F. Bandwidth usage efficiency

Raw arithmetic performance is an important metric but, given the memory-bound behavior of SpMV, the amount of memory bandwidth used is also relevant. Figure 4a shows the absolute memory bandwidth used by *b8c* in two platforms (Alveo U200 and Alveo U280) for the largest and fastest implementations (4 accelerators at 333MHz and 16 accelerators at 250MHz, respectively). The peak achievable bandwidth for each design is also plotted (85.248 GiB/s for the Alveo U200, 256 GiB/s for the U280). Due to the combination of the number of instances, frequency and bus width (4 instances $\times$ 333 MHz $\times$ 64 bytes/cycle $=$ 85.248 GiB/s), the U200's design peak bandwidth is higher than the platform's (77 GiB/s) and, thus, unachievable.

We can observe how, in the U200 case, *b8c* achieves very good bandwidth usage compared to the design's peak, averaging a 71% bandwidth usage with a maximum of 78% and a minimum of 62%. These numbers improve by comparing with the platform's peak bandwidth (average 79%, maximum 86%, minimum 69%), something to be expected, since we used a design that can use more bandwidth than the one provided by the board.

In the U280 case, using 16 accelerators, average memory bandwidth usage compared to the design's peak is 67%, with a maximum of 90% and a minimum of 38%. Compared with the platform's peak bandwidth, the average, maximum and minimum achieved are, respectively, 37%, 50% and 21%. This

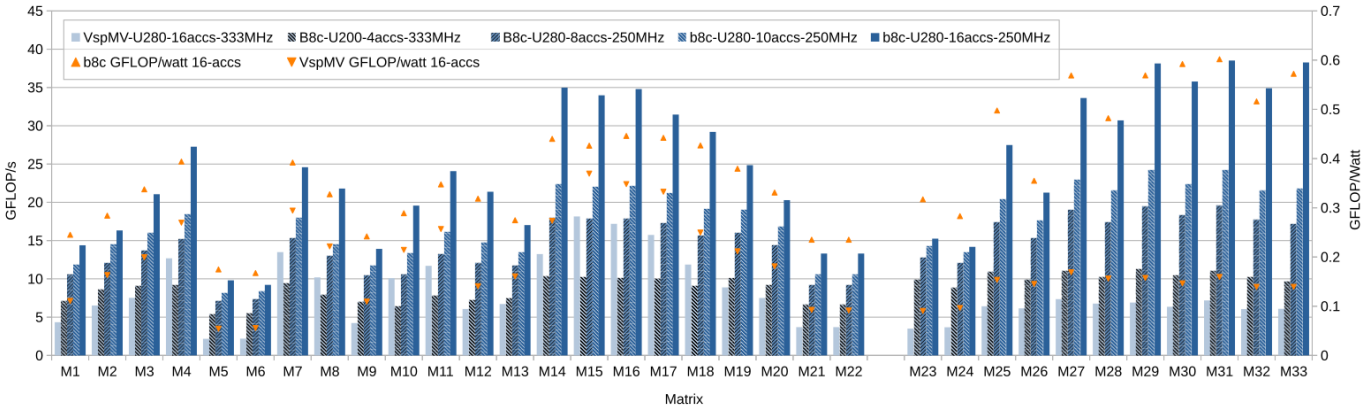
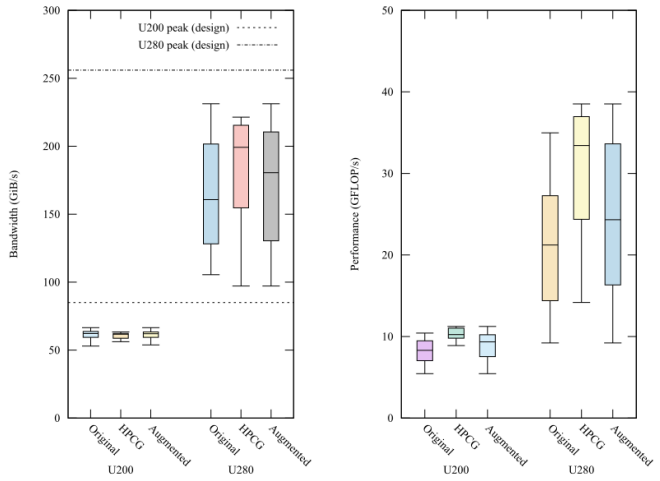


Fig. 3: b8c VS VSpMV performance comparison. 8K blocksize, adaptative-nnz policy for *b8c*.



(a) Bandwidth usage (b) Performance

Fig. 4: *b8c* bandwidth usage and performance obtained per dataset for the maximum accelerators/frequency scenarios (4 accelerators running at 333MHz in the case of the U200, 16 accelerators running at 250MHz in the case of the U280). Data is shown for the original dataset, the HPCG dataset and the augmented (including both) dataset.

is also an expected result, given that, in this case, the design's peak bandwidth is very low compared with the platform's peak bandwidth (256 GiB/s VS 460.8 GiB/s, ∼55%). In fact, unlike in the case of the U200, saturating the U280 HBM memory would require a design running at 450MHz.

Figure 4b shows the performance obtained for each matrix subset using the bandwidth shown in Figure 4a. As it can be observed, the HPCG subset is able to better exploit the bandwidth to obtain performance due to the suitability of the *b8c* encoding to this particular problem.

G. Energy efficiency

Energy consumption for the 16 CUs/accelerators cases was obtained using `xbutil` for VSpMV and OmpSs@FPGA's power monitor for *b8c*. As can be noted in Figure 3 and Table III, *b8c* yields also better performance per Watt ratio across all matrices, improving VSpMV results in every case.

VI. Conclusions and Future Work

In this work, we have introduced *b8c*, a sparse matrix representation and its corresponding SpMV FPGA co-designed implementation using OmpSs@FPGA and HLS. Our implementation can leverage most of the unique features of FPGAs: data-streaming and processing, 1-cycle memory accesses to on-chip SRAM and simultaneous processing of $memory\_width/element\_size$ elements per cycle per instance in order to outperform previous FP64 FPGA versions of SpMV. *b8c* simplifies hardware implementation by generating a conflict-free matrix representation that removes the need for caches or arbitration on the x and y accesses, while enabling prefetching. The results obtained, compared with the vendor's reference implementation, show that our approach achieves significantly higher performance either using HBM or DDR. We have also demonstrated how by using AXI bonding, the reconfigurable size/number of memory channels may benefit specific workloads that naturally suffer when trying to scale to a large number of accelerators. We plan to continue working on this line in order to further optimize the matrix representation and the SpMV algorithm using it and to test them against a wider data set.

VII. Acknowledgement

This work has received funding from the MEEP project (European High-Performance Computing Joint Undertaking (JU) under grant agreement No 946002), from "Lenovo-BSC Contract-Framework (2022)", from the Spanish Government (PID2019-107255GB-C21/AEI /10.13039/501100011033, CEX2021-001148-S funded by MCIN/AEI / 10.13039/501100011033), and from Generalitat de Catalunya (contract 2021-SGR-01007).

The authors would like to thank the rest of the members of the OmpSs@FPGA team (Antonio Filgueras, Juan Miguel de Haro, Miquel Vidal) and from the MEEP team (Elias Perdomo, Daniel Jiménez Mazure) at BSC for their helpful support and collaboration and the fruitful discussions.

REFERENCES

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*, ser. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 2003.

[2] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, Dec. 2015, pp. 1135–1143.

[3] J. Kepner, P. Aaltonen, D. A. Bader, A. Buluç, F. Franchetti, J. R. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. G. Mattson, and J. E. Moreira, "Mathematical foundations of the GraphBLAS," in *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*. IEEE, 2016, pp. 1–9.

[4] J. Dongarra, M. A. Heroux, and P. Luszczek, "A new metric for ranking high-performance computing systems," *National Science Review*, vol. 3, no. 1, pp. 30–35, 2016.

[5] "HIGH BANDWIDTH MEMORY (HBM) DRAM — JEDEC," https://www.jedec.org/standards-documents/docs/jesd235a.

[6] B. Black, "Die Stacking is Happening in Mainstream Computin," *Additional Conferences (Device Packaging, HiTEC, HiTEN, and CICMT)*, vol. 2014, no. DPC, pp. 001 183–001 206, Jan. 2014.

[7] J. M. D. H. Ruiz, J. Bosch, A. Filgueras, M. Vidal, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguadé, and J. Labarta, "OmpSs@FPGA framework for high performance FPGA computing," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2029–2042, 2021.

[8] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, Jan. 1998.

[9] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The Design of OpenMP Tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, Mar. 2009.

[10] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.

[11] V. S. L. Xilinx, "Double Precision SpMV Overview," https://xilinx.github.io/Vitis_Libraries/sparse/2021.1/user_guide/L2_spmv_double_intro.html.

[12] "Xilinx. Vitis Sparse Library," https://xilinx.github.io/Vitis_Libraries/sparse/2021.1/index.html.

[13] P. Malakonakis, G. Isotton, P. Miliadis, C. Alverti, D. Theodoropoulos, D. Pnevmatikatos, A. Ioannou, K. Harteros, K. Georgopoulos, I. Papaefstathiou, and I. Mavroidis, "Preconditioned Conjugate Gradient Acceleration on FPGA-Based Platforms," *Electronics*, vol. 11, no. 19, p. 3039, Jan. 2022.

[14] S. K. Prakash, H. Patel, and N. Kapre, "Managing HBM Bandwidth on Multi-Die FPGAs with FPGA Overlay NoCs," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2022, pp. 1–9.

[15] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA*. Association for Computing Machinery (ACM), 2005, pp. 63–74.

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

[17] P. Grigoras, P. Burovskiy, W. Luk, and S. Sherwin, "Optimising Sparse Matrix Vector multiplication for large scale FEM problems on FPGA," *FPL 2016 - 26th International Conference on Field-Programmable Logic and Applications*, 2016.

[18] A. K. Jain, H. Omidian, H. Fraisse, M. Benipal, L. Liu, and D. Gaitonde, "A Domain-Specific Architecture for Accelerating Sparse Matrix Vector Multiplication on FPGAs," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, Aug. 2020, pp. 127–132.

[19] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a universal FPGA matrix-vector multiplication architecture," *Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, FCCM 2012*, pp. 9–16, 2012.

[20] L. Song, Y. Chi, L. Guo, and J. Cong, "Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 211–216.

[21] Y. Du, Y. Hu, Z. Zhou, and Z. Zhang, "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*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, Feb. 2022, pp. 54–64.

[22] J. de Fine Licht, G. Kwasniewski, and T. Hoefler, "Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, Feb. 2020, pp. 244–254.

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

[24] X. Chen, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "ThunderGP: HLS-based Graph Processing Framework on FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, Feb. 2021, pp. 69–80.

[25] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient SpMV operation for large and highly sparse matrices using scalable multi-way merge parallelization," *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 347–358, 2019.

[26] J. Dongarra, "Sparse matrix storage formats," *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide. SIAM*, vol. 11, Jan. 2000.

[27] H. Bian, J. Huang, R. Dong, L. Liu, and X. Wang, "CSR2: A New Format for SIMD-accelerated SpMV," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, May 2020, pp. 350–359.

[28] C. Gómez, F. Mantovani, E. Focht, and M. Casas, "Efficiently running SpMV on long vector architectures," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21. New York, NY, USA: Association for Computing Machinery, Feb. 2021, pp. 292–303.

[29] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse Matrix-Vector Multiplication on GPGPUs," *ACM Transactions on Mathematical Software*, vol. 43, no. 4, pp. 30:1–30:49, Jan. 2017.

[30] "Vivado HLS Optimization Methodology Guide," https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2018_1/ug1270-vivado-hls-opt-methodology-guide.pdf, 2018.

[31] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors," in *Proceedings of the International Conference on Supercomputing*, Jun. 2013, pp. 273–282.

[32] C. Giannoula, I. Fernandez, J. G. Luna, N. Koziris, G. Goumas, and O. Mutlu, "SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 1, pp. 21:1–21:49, Feb. 2022.

[33] F. Sadi, J. Sweeney, S. McMillan, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "PageRank Acceleration for Large Graphs with Scalable Hardware and Two-Step SpMV," in *2018 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–7.

[34] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[35] B. Liu and D. Liu, "Towards High-Bandwidth-Utilization SpMV on FPGAs via Partial Vector Duplication," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '23. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 33–38.