The Name of the Title Is Hope

BEN TROVATO* and G.K.M. TOBIN*, Institute for Clarity in Documentation, USA
LARS THØRVÄLD, The Thørväld Group, Iceland
VALERIE BÉRANGER, Inria Paris-Rocquencourt, France

A clear and well-documented LaTeX document is presented as an article formatted for publication by ACM in a conference proceedings or journal publication. Based on the "acmart" document class, this article presents and explains many of the common variations, as well as many of the formatting elements an author may use in the preparation of the documentation of their work.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Additional Key Words and Phrases: Do, Not, Us, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:
Ben Trovato, G.K.M. Tobin, Lars Thørväld, and Valerie Béranger. 2018. The Name of the Title Is Hope. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 18 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

2 Motivation

Several existing code matching and replacement tools rely on code structure to detect linear algebra patterns [4, 6]. While such approaches fit well-structured naive implementations, they are prone to fail for more complex implementations. It is worth noting that that the goal of the matcher is to perform variable mapping as well as function detection. Performing such a task using a set of code patterns and constraints is extremely challenging for non-naive implementations, such as codes using AVX2 intrinsics as shown in Fig. **??**. Other approaches such as the ATC compiler [12] rely on neural program classification [3] to get a set of candidate functions before examining their behavior using input/output analysis. The need for training makes extending the matcher to more function classes expensive, and makes the matching process dependent on the quality of the training data set.

Mapping the software to a hardware accelerator additionally defines some requirements for the accelerator design. First, we cannot predict the size of the input beforehand. Second, we cannot know whether the input is sparse or dense. Therefore, it is crucial that the hardware supports varying sizes, and is additionally optimized for handling dense and sparse inputs.

Several recent works have explored SpMV acceleration on FPGA-HBM platforms [5, 10, 11, 14]. However, all those works are optimized for handling sparse matrices using well-known or customized sparse formats. While a dense matrix can be converted to a sparse format, such as CSR or CSC, such handling of a dense matrix would involve much unneeded overhead. First, the data required to be transferred to the accelerator is up to 3X that needed for a dense matrix. As an example, the CSR format defines three arrays: values, row pointers, and column pointers. For a sparse matrix, such a format makes sense due to the small number of non-zeros relative to the overall matrix size. For a dense matrix consisting solely of non-zeros, the three arrays invoke a large data transfer overhead. Additionally, several SpMV accelerators involve crossbars or NoCs to route matrix elements to their suitable vector banks. Such implementations cause performance overheads for dense matrices.

*Both authors contributed equally to this research.

Authors' Contact Information: Ben Trovato, trovato@corporation.com; G.K.M. Tobin, webmaster@marysville-ohio.com, Institute for Clarity in Documentation, Dublin, Ohio, USA; Lars Thørväld, The Thørväld Group, Hekla, Iceland, larst@affiliation.org; Valerie Béranger, Inria Paris-Rocquencourt, Rocquencourt, France.

Manuscript submitted to ACM

(a) Naive Matrix-Vector Multiplication

```
for (int i = 0; i < N); ++i) {
 y [i] = 0;
 for (int j = 0; j < N; ++j) {
   y [i] +=  A [i *  N  + j] *  x [j];
 }
}
```

(b) Unrolled Matrix-Vector Multiplication

```
for (int i = 0; i < N; ++i) {
 y [i] = 0;
 for (int j = 0; j <  N ; j += 4) {
   y  [i] +=  A [i *  N  + j] *  x [j];
  y [i] += A [i * N + j + 1] * x [j + 1];
  y [i] += A [i * N + j + 2] * x [j + 2];
  y [i] += A [i * N + j + 3] * x [j + 3];
 }
}
```

(c) AVX Intrinsics Matrix-Vector Multiplication

```
#include <immintrin.h>


for (int i = 0; i < N; ++i) {
 __m256 vec_y = _mm256_setzero_ps();
 for (int j = 0; j <  N ; j += 8) {
  __m256 vec_a = _mm256_loadu_ps(& A [i *  lda  + j]);
  __m256 vec_x = _mm256_loadu_ps(& x [j]);
  vec_y = _mm256_fmadd_ps(vec_a, vec_x, vec_y);
 }
  y [i] =  beta  * y [i];
 float temp[8];
 _mm256_storeu_ps(temp, vec_y);
 for (int k = 0; k < 8; ++k) {
  y [i] +=  alpha  * temp[k];
 }
}
```

(d) FPGA Accelerator Pass

```
void fpga_gemv(float * A , float * x , float * y
      , float  alpha , float  beta , int  lda , int  N
      ) {
 // Transfer data to FPGA
 transfer_to_fpga(A, x, y, alpha, beta, lda, N);

 // FPGA computation
 start_fpga_computation();

 // Retrieve results from FPGA
 retrieve_from_fpga(y);
}
```

Fig. 1. Matrix-Vector Multiplication Implementations and FPGA Accelerator Pass

2.1 Our Approach

2.1.1 Compiler. We design a compiler tool which relies solely on analysing a given code behaviour rather than its structure. Input/output analysis is used to determine whether the code matches a linear algebra function (GEMM or GEMV). The tool does not require training or profiling. The compiler performs variable mapping to determine variables corresponding to array sizes and constants. Next, the compiler uses the knowledge about function name and the mappings of its arguments to determine array sizes.

2.1.2 Profitability. This step is crucial for determining whether offloading the function call to hardware is profitable or not. Profitability analysis is performed using a heuristic tailored for FPGA-HBM platforms, which are the target hardware. The heuristic can easily be extended to support different hardware platforms. Once the heuristic determines that execution on the FPGA-HBM is profitable, the compiler replaces the user code with a call to the accelerator.

2.1.3 Legality. Behavioural equivalence methods, such as input/output testing, does not guarantee that the code is a linear algebra function. The same applies to pattern matching methods such as Kernelfarer [4], where there is no formal proof that the code is a specific operation. However, previous work as well as our experiments [12] have shown that, in practice, there are no false positives generated by the input/output testing method. For a more solid guarantee, a prompt can be issued to the user for confirming the detection output.

2.1.4 Hardware. We design our hardware focusing on matrix-vector multiplication operations. Our accelerator supports flexible input sizes since it leverages a streaming dataflow architecture. Preprocessing is performed such that the input presented to the accelerator is optimized for both sparse and dense input matrices. Additionally, the accelerator's design ensures high performance for both input types.

3 Related Work

4 Linear Algebra Code Matcher

4.1 Function Filtering

We begin by loading the LLVM Intermediate Representation (IR) module of the given source file. This is achieved using llvm::LLVMContext and llvm::Module objects, which handle and store the module, respectively. The functions within the module are iterated, and metadata is employed to filter out non-user-defined functions. This ensures that our analysis focuses solely on functions defined within the provided source file, excluding those from external libraries. A heuristic is applied to identify potential GEMM and GEMV functions based on their argument types. The heuristic examines each function's arguments to count pointers to integer, float, and double types. Functions with multiple pointer arguments and a void or pointer return are flagged as potential candidates for GEMM and GEMV.

4.2 Input/Output Detection

Accurate detection of input and output variables is crucial for the subsequent input/output testing step. Input-output detection involves identifying which function arguments serve as inputs (read-only) and which ones act as outputs (write-only or read-write). In other words, which arguments are livein and/or liveout variables. Liveness analysis can be performed using static analysis methods, which suits well-structured programs. However, static analysis is prone to fail for more diverse real-world codes, which may use assembly code or intrinsic functions.

Our approach uses dynamic analysis to detect which arguments are inputs and which are outputs. The function under test is executed using LLVM's Just-In-Time (JIT) compilation, with controlled input values to determine the memory behavior of each argument. In C, variables are passed by value so non-pointers variables are always livein. Our focus here is on pointer arguments (or arrays). The function is executed twice with different sets of inputs. Pointer arguments which change their value in memory are considered liveout, while the rest are livein. The second execution, with altered inputs, helps identify which variables are both livein and liveout. This dual-execution approach is critical for distinguishing between input and output arguments.

By comparing memory states before and after execution, we classify arguments as inputs, outputs, or both. Arguments with unchanged memory regions are classified as inputs, while those with changed regions are categorized as outputs. This comparison is performed by hashing the memory contents and comparing the hashes before and after function execution.

4.3 Variable Mapping

In the variable mapping phase, we aim to recognize which arguments of the function correspond to matrices dimensions, and which correspond to constants associated with the operation. More specifically, the GEMM and GEMV operations are represented by equations (1) and (2) respectively. The operations variables and constants are defined table 1. Our goal is to find which, if any, of the function arguments correspond to the operation variables and constants, such that the operation matches a GEMM or a GEMV.

The General Matrix-Matrix Multiplication (GEMM) equation is given by:

$$C := \alpha \cdot A \cdot B + \beta \cdot C \tag{1}$$

The General Matrix-Vector Multiplication (GEMV) equation is given by:

$$y := \alpha \cdot A \cdot x + \beta \cdot y \tag{2}$$

The mapping algorithm we utilize is designed to dynamically verify if a given function performs matrix-matrix multiplication correctly. It leverages LLVM's JIT capabilities to execute the target function with predefined inputs and compare the outputs against expected results. The algorithm supports various data types, including float, double, and int.

The first step is defining matrix dimensions and initializing three matrices: A, B, and C. The matrices are populated with test data to serve as inputs for the target function. The mapper then analyzes the arguments of the target function to identify which arguments correspond to the matrices and their dimensions. This identification process classifies arguments into pointers (for matrices) and integers (for dimensions and constants). The mapper iterates through possible permutations of the argument indices to determine the correct mapping of function arguments to matrices and dimensions. This exhaustive search ensures that all possible configurations are tested.

Variable	GEMM	GEMV
α	Scalar multiplier for $A \cdot B$	Scalar multiplier for $A \cdot x$
β	Scalar multiplier for C	Scalar multiplier for y
A	$m \times k$ matrix	$m \times n$ matrix
B	$k \times n$ matrix	$n \times 1$ vector
C	$m \times n$ matrix	$m \times 1$ vector
lda	Leading dimension of A	Leading dimension of A
ldb	Leading dimension of B	Not applicable
ldc	Leading dimension of C	Not applicable

Table 1. Comparison of variables in GEMM and GEMV

Due to the need for calling functions whose signatures are not know apriori, we utilize the foreign function interface library (libffi) to call JIT for the various permutations we test. Using the libffi library, the mapper sets up the necessary calling interface to execute the target function. The argument types and values are prepared for the JIT-compiled function call. The target function is executed with the prepared arguments. The resulting output matrix is compared against the expected result, which is computed using a reference matrix-matrix multiplication implementation. The comparison allows for a tolerance to account for floating-point inaccuracies.

4.3.1 Classifying operations between square matrices. As an additional verification step, we develop a recognizer model which exploits the properties of square matrices to verify the multiplication operation. Given the square matrices A, B, and C with order n. There are no constraints on usual matrices multiplication regarding their orders. The usual matrices multiplication AB is given by Eq. (3). We assume that the function under test performs a linear algebra operation between the matrices (multiplication, dot product, addition). Figure 2 shows the algorithm diagram. The multiplication operation is defined as:

$$AB = \sum_{k=1}^{n} a_{ik} b_{kj}, \quad i, j = 1, \ldots, n. \tag{3}$$

where a_{ik} and b_{kj} are the matrix entries of A and B respectively, and n is the order of the square matrices. The multiplication property is defined as:

$$(cA)B = c(AB) \tag{4}$$

For the square matrix A, the trace $\mathrm{tr}(A)$ is defined as shown in Eq. (5).

$$\mathrm{tr}(A) = \sum_{i=1}^{n} a_{ii} \tag{5}$$

The basic properties of the trace are as follows:

(1) $\mathrm{tr}(A + B) = \mathrm{tr}(A) + \mathrm{tr}(B)$
(2) $\mathrm{tr}(\lambda A) = \lambda \mathrm{tr}(A)$

where λ is a scalar number. Based on the multiplication and trace properties shown above, we get:

$$\mathrm{tr}(\lambda AB) = \lambda \mathrm{tr}(AB) \tag{6}$$

Considering that $\mathrm{tr}(\lambda A + B) = \lambda \mathrm{tr}(A) + \mathrm{tr}(B)$, we can see some differences between matrix multiplication and. for example, matrix addition with respect to the trace properties. We can use this difference to classify these matrix operations when the operators and results are known. For instance, considering the matrices A, B, and C such that:

$$A?B = C \tag{7}$$

(1) The square matrices A, B, and C must have the same order.
(2) If the operation is multiplication or dot product, the equation $\mathrm{tr}(\lambda AB) - \lambda \mathrm{tr}(AB) = 0$ is true (see Eq. (6)). Otherwise, the operation is addition.

(3) If C is a scalar, the operation is a dot product; otherwise, it is multiplication.

Deriving from these properties, we first pass two random-input square matrices and check the trace property. Next, we construct the first input matrix as an upper triangular matrix. The second input matrix is filled as the identity matrix. The input data to the model consists of the following features for each test sample:

(1) Element $[1, n]$ of the result matrix C (last element of the first row of matrix C), denoted as C_{1n}.
(2) Trace of matrix A, denoted as $\text{tr}(A)$.
(3) Element $[1, 1]$ of matrix A (first element of the first row of matrix A), denoted as A_{11}.
(4) Element $[1, n]$ of matrix B (last element of the first row of matrix B), denoted as B_{1n}.
(5) Trace of matrix B, denoted as $\text{tr}(B)$.
(6) Trace of the result matrix C, denoted as $\text{tr}(C)$.

The model uses these features to classify the operation performed on the matrices as follows:

- **Trace Property for Addition**: The model checks if the sum of the traces of matrices A and B equals the trace of the resulting matrix C:
$$\text{tr}(A) + \text{tr}(B) = \text{tr}(C)$$
If this condition is met (within a numerical tolerance ϵ), the operation is classified as matrix addition.
- **Upper Triangular and Identity Matrix Property for Multiplication**: The model checks if the element C_{1n} equals the sum of the elements A_{11} and A_{1n}:
$$C_{1n} = A_{11} + A_{1n}$$
This condition is based on the assumption that A is an upper triangular matrix and B is an identity matrix with $B_{1n} = 1$. If this condition is met (within the numerical tolerance ϵ), the operation is classified as matrix multiplication.
- **Other Operations**: If neither of the above conditions is met, the operation is classified as another type.

The model evaluates its performance by calculating the accuracy of its predictions compared to the true labels.

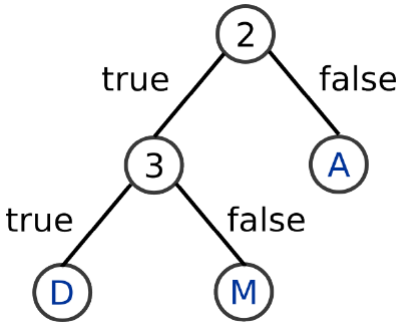


Fig. 2. Example SVG inclusion

4.4 Size Detection

Input size detection is an important step in our compiler toolflow, to ensure that we are offloading to the hardware only kernels which are worth offloading. In other words, the input size is required to be large enough to utilize the HBM. There are various methods in literature for input size detection. Static analysis [13] is suitable for simple programs, but can fail for complex cases. Dynamic testing [12], by executing the function with variable sized inputs and recording points at which out-of-bound errors are reported, works well for complex codes. However, it is time consuming.

We use the knowledge about the function name and variable mappings from the previous step to locate the function call. Then, we perform a function replacement pass to replace the function with a simple function which prints the matrices dimensions and their values in the call into a csv file. Since there is no decision yet about whether it is profitable to offload the function or not, we make a copy of the original code file and perform the replacement there. After offloading profitability is analyzed, a replacement pass is performed on the original code file.

4.5 Code Replacement

This step is performed in the case that the offloading profitability heuristic, discussed in Section 5, indicates that the function should be offloaded. Since we focus in our hardware on matrix-vector multiplication, we focus here on those functions rather than the GEMM functions. The toolflow can be easily extended to support hardware accelerating GEMM. Using LLVM's powerful code transformation capabilities, we implement a function replacement pass. This pass scans the intermediate representation (IR) of the user's code, identifies calls to the chosen function, and replaces them with a custom function.

The custom function, which replaces the original matrix-vector multiplication function, first converts the dense matrix to CSR format. This conversion involves extracting non-zero elements and their indices, which are then stored in a space-efficient manner. Additionally, it performs a sparsity test as this is a required input for our accelerator. The CSR matrix is saved in an NPZ file using Python's Numpy library, which is invoked from within the C++ code via the Python C API. This NPZ file serves as input to the FPGA accelerator.

Next, we consider how to call the accelerator. Our accelerator code follows the Vitis flow. The HLS code for the accelerator is compiled as an xclbin. A C++ host program initializes the FPGA, loads the appropriate xclbin bitstream, and configures the kernels required for hardware execution. It also handles the data transfer between the host and the FPGA, executes the computation on the FPGA, and retrieves the results. The host program uses Xilinx's OpenCL runtime to interface with the FPGA. To improve modularity and ease of use, the host code is compiled as a shared library. This encapsulates the FPGA setup and execution logic, making it easier to invoke from other programs or frameworks. The library that can be dynamically linked to other applications. By compiling the host code as a library, we achieve better separation of concerns and allow for more straightforward integration with various software workflows. This library can then be linked against by the LLVM-based replacement function or other programs. The replacement function dynamically loads the shared library and calls the necessary functions to perform the matrix-vector operation on the FPGA.

5 Offloading Profitability Prediction

We utilize a heuristic method we derive from our previous work, Auto-DOK **??**, to predict the profitability of offloading a matrix-vector multiplication function to an FPGA-HBM platform. The first metric we utilize from Auto-DOK is input size. Auto-DOK identifies a threshold (th_{in}) at which it determines that a given input size offers good HBM PC utilization. More specifically, if the input is too small to be distributed among the HBM PCs, the bandwidth utilization degrades drastically. The access latency of HBM is not lower than that of DDR4 [**? ?**]. The benefit of HBM comes from its high bandwidth. Fig. 3.a. records the achieved bandwidth for a microbenchmark which performs sequential memory reads. Additionally, Fig. 3.b. records the bandwidth for a matrix-vector multiplication example for different input sizes. For both benchmarks, the bandwidth is 80% of HBM theoretical bandwidth (460 GB/s) at 64MB per PC.

The second metric is Arithmetic intensity. The Alveo U280 does not provide the FPGA with access to the host memory. Therefore, any data we need to stream must first be copied from the host to the FPGA. This data movement is expensive. We observe that for an FPGA-HBM accelerator to have a speedup over a host CPU, the kernel must exhibit enough arithmetic computations to balance the data movement cost. To quantify this, we calculate the ratio between the number of arithmetic and relational operations executed by the kernel to the size of data moved. As this definition is close to that of Arithmetic Intensity, we call this measure $AI_{cpu \rightarrow FPGA}$.

The thresholds, th_{in} and th_{AI}, are tuned based on design of experiment (DoE). To evaluate the accuracy of the thresholds, we show the effect of varying every metric while assuming all other metrics are over their thresholds. We show results for such tests for 2 pairs of benchmarks, where each pair has similar functionality but different HLS implementations. The first pair is sparse and dense matrix-vector multiplications(MV, SpMV). The second pair are vector operation (VO), and pseudorandom access (PS).

Fig. 4.a. shows FPGA-HBM speedup over CPU when $AI_{cpu \rightarrow FPGA}$ is varied. Input per PC is set to 64MB so that it is above th_{in}, and the 4 benchmarks pass the regularity check. $AI_{cpu \rightarrow FPGA}$ is varied by incorporating a *Repeats* number as explained in III.D. Similarly, Fig. 4.b. shows the speedup when data size per PC is varied while keeping $AI_{cpu \rightarrow FPGA}$ above th_{AI}. The graphs show that at small input sizes and low arithmetic intensities, the kernels have very low (or no) speedups. Our thresholds define the metrics at which the kernels are close to achieving their maximum speedup.

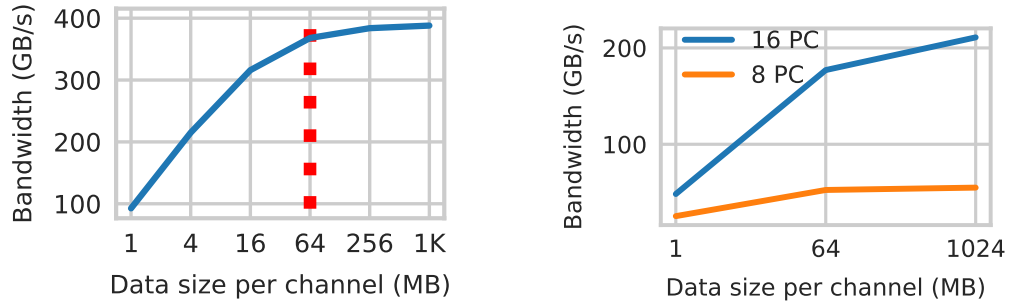


Fig. 3. Bandwidth for varying input sizes. (a) Sequential memory access. (b) MV.

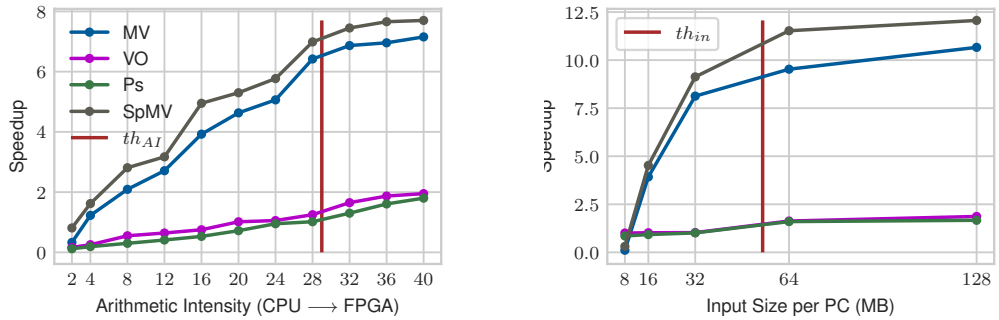


Fig. 4. Effect of varying (a) $AI_{CPU \rightarrow FPGA}$. (b) $Size_{in}$ on FPGA-HBM speedup over CPU.

6 Matrix-Vector Multiplication Acceleration on FPGA-HBM

6.1 Custom Matrix Format

SpMV operation has the potential for extensive parallelism across the rows of a sparse matrix, with each row conducting a dot product with the dense vector. However, the commonly used compressed sparse row (CSR) format is not ideal for leveraging this parallelism. CSR distributes the necessary data for different processing elements (PEs) across distant memory locations, hindering both intra-channel vectorized and inter-channel concurrent memory accesses. The coordinate list format (COO) groups the element value, row, and column together, enabling random order storage of the matrix. Nonetheless, sparse matrix storage formats are not well-suited for storing dense matrices.

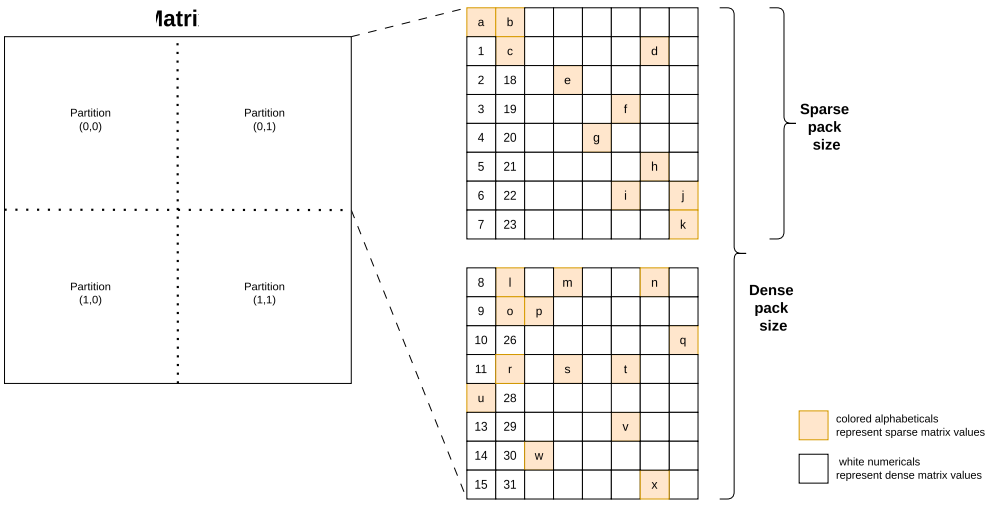
In this work, we propose a novel algorithm, Adaptive Cyclic Packed Streamed Rows (ACPSR), to handle sparse and dense input matrices to read the input matrix in a streamized manner without transferring unneeded information. The algorithm first partitions the matrix, then writes the matrix elements to packets whose structure varies according to the matrix type. Fig. **??** illustrates the preprocessing steps for an architecture with 2 HBM channels.

The first step in ACPSR is partitioning the matrix along the rows and columns. Since both the vector and output are buffered on-chip, partitioning is essential to handle large matrices whose size exceeds buffer sizes. Therefore, the row partition size depends on the output buffer size, and similarly the column partition size depends on the vector buffer size.

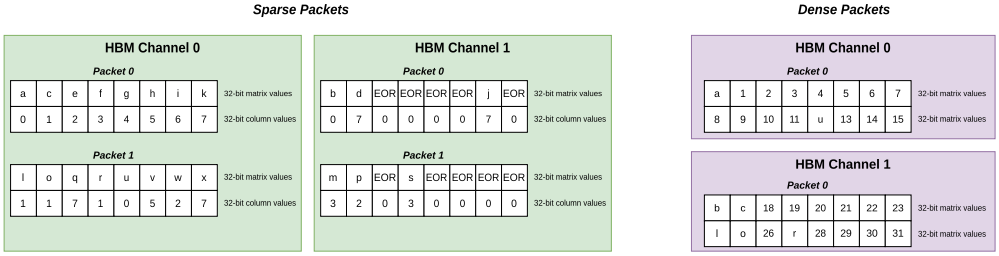
The next step is streamizing, which cyclically assigns rows to PEs and concatenates the rows that are assigned to one PE into one HBM channel. ACPSR marks the end of a row by inserting a special symbol (EOR) to the arrays of columns IDs and values. This approach avoids a third array for storing the row length information and simplifies the hardware design. To pack the streams assigned to a given HBM channel, ACPSR traverses the streams in a column-major order. That is, it places the first element of each stream at contiguous locations, then places the next element of of each stream at contiguous locations; this process continues until

all streams run out of elements. For alignment, dummy elements are padded to the end of the shorter stream. This data placement enables all PEs to run in parallel, each processing one stream of data.

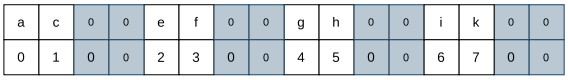
The pack size (how many streams to pack) is adaptive depending on whether the matrix is dense or sparse. Since we can read 512 bits at a time from one HBM channel, this translates to 16 32-bit values per packet. For a sparse matrix, we need to associate column information with each element value, thereby defining the sparse pack size as 8. For a dense matrix, we already know that the column-majr traversal implies that all elements within a packet have the same column index. Therefore, we drop the column information for dense packets, leading to a dense pack size of 16.



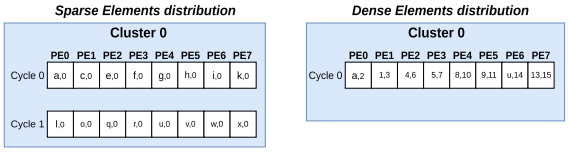
(a) Figure A



(b) Figure B



(c) Figure C



(d) Figure D

Fig. 5. Main caption for all subfigures

6.2 Accelerator Architecture

Fig. 6 shows an overview of the accelerator architecture. It consists of several clusters of matrix-vector computation, a vector loader, and a result draining unit. The computation clusters are memory-centric, enabling efficient streaming of both sparse and dense matrices. Additionally, the memory-centric design enables rapid scalability. Users can easily customize the hardware according to their need on various memory channels and bandwidths. All modules are connected through streaming FIFOs in a dataflow architecture.

The vector loader unit reads the dense vector values from HBM, and sends each cluster a copy of the data. Each computation cluster is connected to one HBM channel, where the matrix loader module reads streams of input matrix values, unpacks them to restore row indices, and distributed them to the vector read/write units. Each cluster includes four vector read/write units, which are responsible for double buffering of the incoming vector values, as well as reading out vector values according to the column index of the received matrix element.

Eight processing elements are implemented per cluster. They perform the multiplication of the non-zero matrix element with the matching vector element, and write the result to the PE's local output buffer. The ACPSR format ensures that the row indices per packet are unique, and the hardware architecture ensures that each PE is responsible for a set of row indices in order to avoid conflicts and dependencies. After PE execution, a result packer collects all results per cluster, and sends them to the global result draining unit. This, in turn, collects cluster results and writes them back to the HBM.

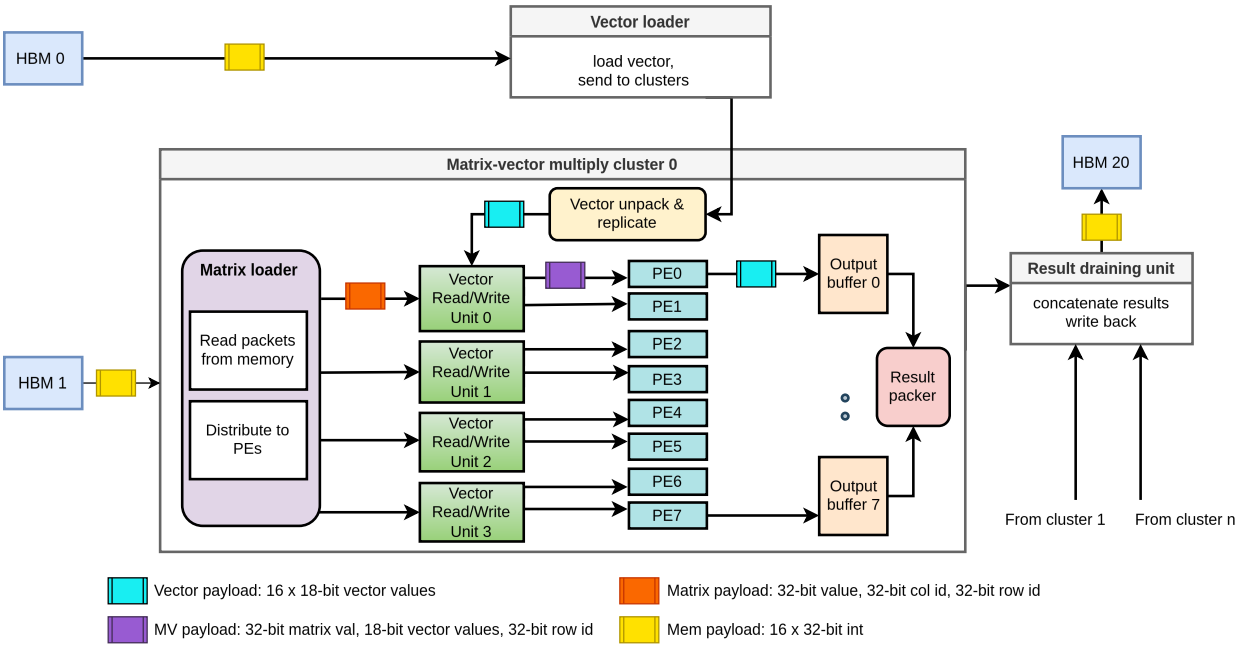


Fig. 6. Example SVG inclusion

6.2.1 Matrix Loader. The matrix loader unit is responsible for unpacking the matrix after loading from HBM by restoring its row indices, and then passing the data to the vector read/write units. Given that we use the same dataflow and the same modules for both dense and sparse input matrices, the matrix loader unit has to adapt the size of the sparse input to match that of the dense. More specifically, the ACPSR format enables the loading 16 dense matrix elements since the column information is discarded. Therefore, the matrix loader performs zero padding after every 2 sparse elements, such that it produces 16 elements to be passed to the vector units, similar to the dense elements. Fig. **??** illustrates the padding operation.

6.2.2 Vector Read/Write Units. Each PE requires random accesses to the same input dense vector. Several solutions have been proposed in recent litarature, such as crossbars and shuffle units [5, 11], in order to grant every matrix element access to the vector bank it requires. However, such solution are not a good fit for GEMV, since they introduce an extra overhead. For instance, the shuffle unit in HiSparse [5] implements resending logic and a crossbar to manage vector bank accesses. For input payloads with the same target output lane, the unit only grants access to one payload in a round-robin manner. Since a dense matrix packet has elements with the same column index, this causes a performance degradation, in addition to the complexity of the design which affects clock frequency.

Replicating the input vector for every PE in a separate memory is not scalable to multiple HBM channels when using a large number of PEs. To tackle this problem, we utilize index coalescing to improve URAM utilization. More concretely, we use 4 URAMs per vector read/write unit. We refer to each URAM as a bank. The minimum bit width of a URAM configuration is 72. It is a waste

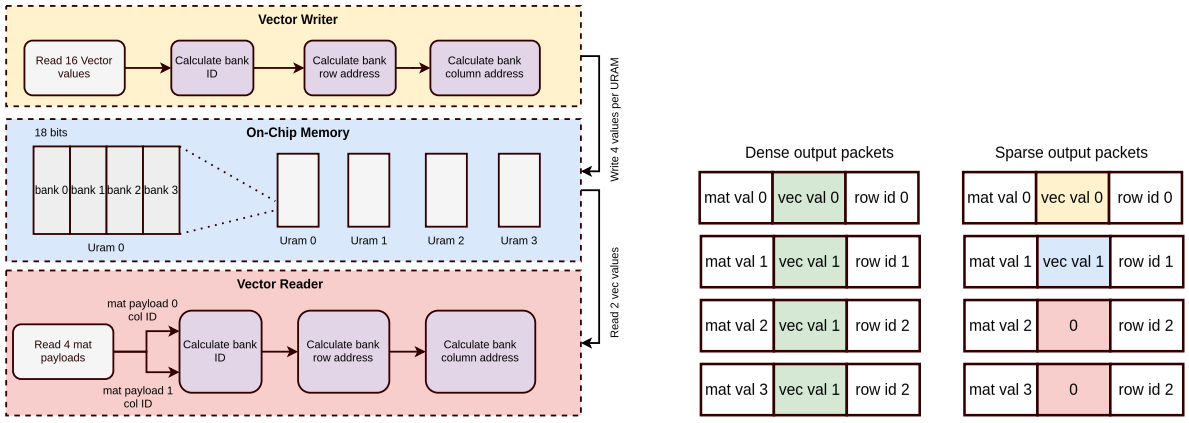
to store just one vector value to one URAM address (entry). Thus, we first approximate the vector values to 18 bits, and we coalesce 4 vector values per URAM address, which we refer to as bank row. The address of a given element within a bank row is the bank column address. Thus, the vector writer and reader operate according to the following formulas:

$$\text{bank ID} = \left(\frac{\text{col\_idx}}{4}\right)\%4 \tag{8}$$

$$\text{bank row address} = \left(\frac{\text{col\_idx}}{16}\right)\%\text{bank\_size} \tag{9}$$

$$\text{bank column address} = \text{col\_idx}\%4 \tag{10}$$

Each cluster packs 4 vector read/write units; hence each vector unit receives 4 matrix input payloads. The URAM is configured as a two-port memory, therefore two memory reads can be performed simultaneously without violating the pipeline. In the case of sparse input, the vector unit receives two input matrix payloads and two zero paddings. Therefore, two memory accesses are performed for the first two inputs. The two resulting vector elements are written to the first two output streams. Zeros are written in the third and fourth output streams. For dense input, all input payloads have the same column index, therefore it is also sufficient to perform two memory accesses for the first 2 inputs such that the logic works for both input types. The vector values read from the two accesses are replicated for the third and fourth output streams. The vector units operations are illustrated in Fig. 7.



(a) Figure A (b) Figure B

Fig. 7. Main caption for all subfigures

6.2.3 Processing Elements with History Keeping. The accumulation operation performed in matrix-vector multiplication over the output buffer causes read-after-write (RAW dependencies). Especially on large output buffers, such as the ones implemented in our design, the issue is demanding since both read and write consume multiple cycles, leading to more dependencies. Hence, a dynamic RAW resolution method is needed in order to fully pipeline the PE and avoid erroneous results.

More specifically, we introduce a data structure, called the Most Recent Queue (MRQ), which acts as a shift register and keeps track of the most recent results calculated by the PE. Such a data structure enables the PE to perform load-store forwarding [5, 10] regardless of the input payload order. The MRQ stores the bank address and value per entry, and a valid bit to tackle the non-existent writes in the first cycles. Since the addition is single-stage for fixed point values, the read latency is 3, and the write latency is 2, the depth of the MRQ is chosen as 5.

Fig. 9 illustrates the overall operation of the PE. First, the PE calculates the bank address of the output buffer where the result should be written. Simultaneously, the PE performs the multiplication operation of the matrix and vector values. Next, an accumulation needs to be performed with the old (existing) value of the row. The PE checks whether the calculated bank address is found in the MRQ. If it is found, this indicates that a RAW dependency is detected. Hence, the old value is read from the MRQ. Else, it is read from the URAM. Afterwards, the MRQ is updated and the buffer write is performed.

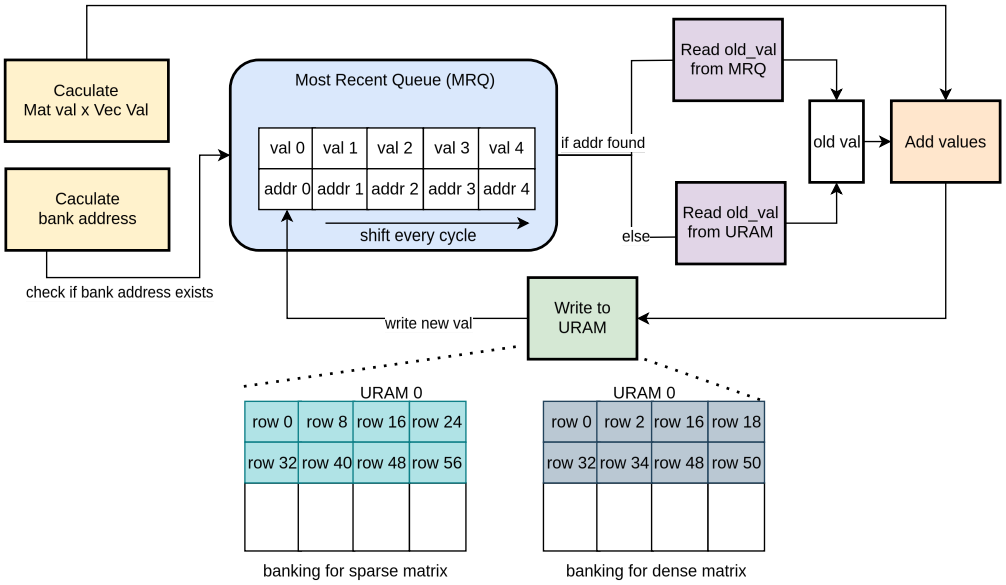


Fig. 8. Example SVG inclusion

Similar to the vector buffer, index coalescing is used to fully utilize the URAM used for the output buffer. Each PE has its own URAM buffer, and each URAM entry consists of 4 row bank addresses. Each PE reads 2 input payloads from the vector unit. For both sparse and dense inputs, 1 read operation is required to get the old value from the output buffer. The bank address calculation is managed such that for sparse and dense inputs such that we get a fully pipelined implementation requiring only 1 write operation. The input indexes are arranged such that each PE gets 2 non-consecutive payloads. PE 0 receives payloads 0 and 2. PE 1 receives payloads 1 and 3, and so on. For the sparse input, the PE reads 1 sparse payload and a zero payload. Thus, the PE needs to perform 1 write operation for the non-zero value. For the dense input, 1 write operation is needed since we read 1 URAM word at a time, and the addresses have a difference of 2, therefore they are always within the same URAM entry. Fig. **??** illustrates the complete dataflow inside a cluster.

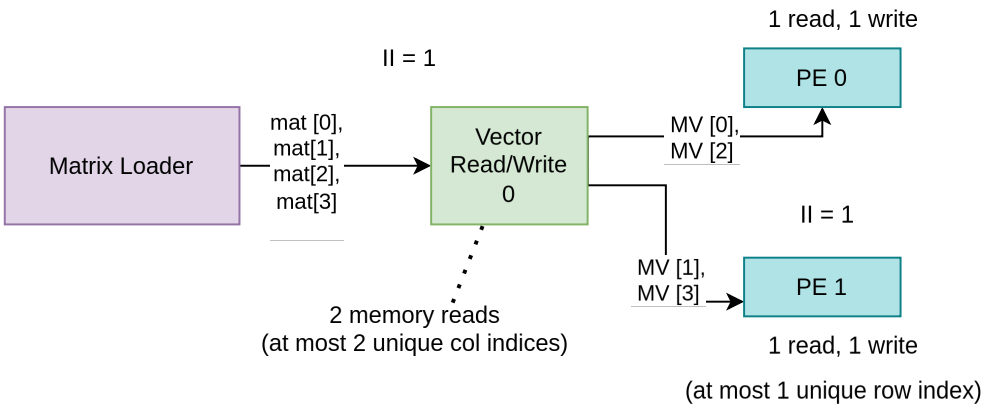


Fig. 9. Example SVG inclusion

6.3 Implementation Flow Enhancements

6.3.1 Split-Kernel Design. Modern FPGA-HBM platforms incorporate several chip dies within a single package to increase on-chip resources [8]. However, the interconnections across dies must traverse the silicon interposer, thus incurring non-trivial delays. Moreover, in an Alveo U280 for instance, not all dies have direct connections to the HBM interface. All the HBM channels connect

to a single die, Super Logic Region 0 (SLR0). A hardware module's access to HBM would incur a long cross-die delay when the module is placed on another SLR, posing challenges to achieving timing closure.

Designing the accelerator as one monolithic kernel would pose challenges for the physical layout we described, since the floorplanner would likely be prone to crowding all the logic on the die nearer to the HBM due to the memory-centric computation clusters. This, in turn, leading to congestion and timing closure issues. Therefore, we split our design into multiple kernels, where each kernel packs several computation clusters. We set the placement of each kernel in the Vitis connectivity file, such that a balanced floorplanning is achieved among the FPGA dies. The vector loader and result draining unit are also individual kernels placed on SLR 0 to be close to the HBM.

Additionally, to broadcast vector data to all dies in the vector loader kernel, we insert extra registers and relay units to pipeline the connection, thereby avoiding costly inter-die crossings. For logic with high fan-out and high fan-in, such as vector duplication and result concatenation, we implement these as multi-level structures to enhance timing performance.

6.3.2 CAD Directives. AMD/Xilinx Vivado tool provides various directives to customize the FPGA implementation flow. Each directive yields different implementation outcomes. Identifying the optimal combination of directives for a specific design is challenging due to their complex behaviors and interactions. In our work, we run the tools in 'ALL' implement strategies mode, which simultaneously initiates multiple combinations of directives. We select the directive combination based on the resulting clock frequency. For Physical optimization phase, "AggressiveExplore" is chosen. For Routing, "NoTimingRelaxation" directive is selected, resulting in an clock frequency of 260MHz. All results are obtained using this combination of directives.

7 Compiler Evaluation

7.1 Experimental Setup

The compiler tool and the native GEMM and GEMV codes are run on an AMD 1950X 2.19 GHz 16-Core machine. LLVM-14 and clang-14 were used for the compiler tool.

7.1.1 User Code. For ease of comparison with state-of-the-art code detection methods, we use the same GEMM benchmarks used for evaluating the ATC compiler [12]. These benchmarks are 50 C and C++ GEMM implementations which were collected from Github. They are categorized by the ATC compiler authors as follows:

- Naive: naive implementations with the known 3 nested loops.
- Naive Parallel: naive but with simple outer loop parallelization.
- Unrolled: naive with unrolled loops.
- Kernel Calls: implementations that divide the loops into different function calls.
- Blocked: tiled implementations.
- Goto: implementations of the Goto algorithm [7]
- Strassen: implementations of the Strassen algorithm [15]
- Intrinsics: implementations using Intel intrinsics

In addition, we select 50 non-GEMM codes from the OJClone benchmarks and non-GEMM linear algebra code to check whether any of the approaches gave false positives. We follow the same procedure for GEMV benchmarks, where we collect user codes from Github, and select 20 programs after eliminating duplicates and programs which had compilation or implementation errors. Table 3 shows the characteristics of the test codes and their categorization.

7.1.2 Baselines. We evaluate our approach against 5 state-of-the-art matchers:

- IDL: Idioms are described using an idiom description language [6], which is translated into a set of constraints over LLVM IR.
- KernelFaRer: Uses different pattern matching to detect specific code constructs, matching specific matrix-multiplication structures [4].
- FACC*: FACC uses neural embeddings and behavioral syn- thesis to detect candidates for acceleration [17]. FACC* improves over it by supporting multi-dimensional arrays.

- ATC: the ATC compiler uses a neural program classifier to detect regions of code that are likely to be linear algebraic operations. Afterwards, it uses input/output analysis to match user program variables with the particular API formal parameters. We share with ATC the input/output detection step which relies on monitoring argument values before and after execution, as well as the concept of using behavioral equivalence for code detection.

7.2 Detection Accuracy

Fig. 10 shows the percentage of GEMM programs matched by each technique across each of the 8 categories of code. IDL was able to match only 6 cases out of 50. The reason for this is that the constraint-based matching method it uses only detects traditional implementations. The detected codes we largely in the naive category, with only 2 cases for unrolled implementations detected. Matching more complex code requires manually writing new constraints to describe each code category. KernelFaRer's patterm matching approach was more successful than IDL. It was able to detect 11 GEMMs codes. However, all the 11 detected cases were naive implementations. All other variations were undetected.

FACC*, ATC, and our approach share the broad concept of input/output testing. FACC* performed poorly on naive implementations, but better on others. the reason for this is that FACC* limits the size of the mapping search space, with a timeout of less than 10 minutes. Within this time limit, FACC* was able to was able to detect 10 cases. ATC is significantly more robust than the rest of the baselines across all categories, matching 42 out of 50 cases. Our approach achieves even higher accuracy than ATC, matching 45 out of the 50 cases within much less execution time. Similarly for GEMV detection, our approach achieves 90% detection accuracy. Table 3 shows the matching result for each code.

Both our approach and ATC are able to detect all naive implementations and the majority within each other category, and are the only methods which detect GEMMs and GEMVs in codes containing kernel calls and intrinsic instructions. None of the methods classified any of the 50 non-GEMMs as a GEMM. Across all methods, there were no false positives.

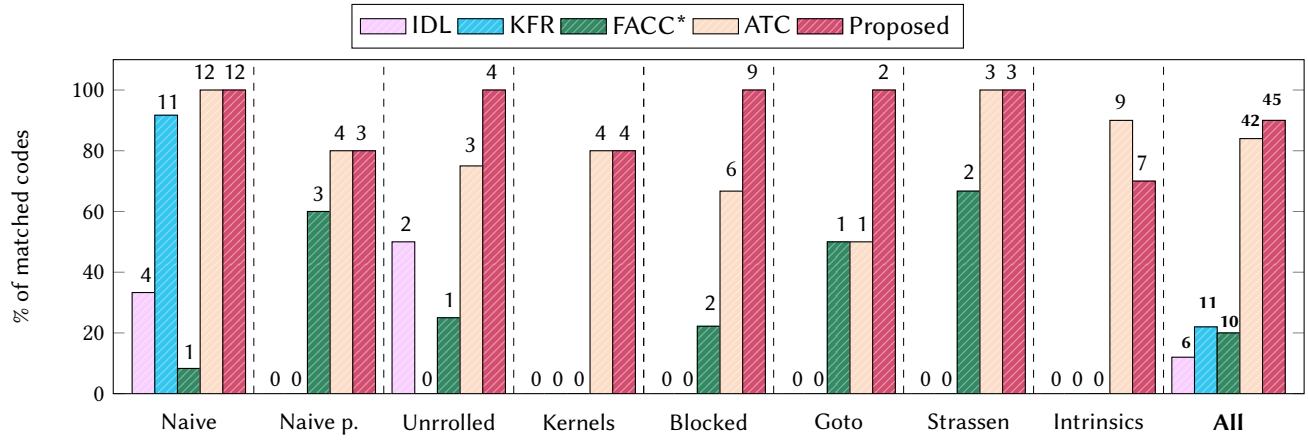


Fig. 10. Bar plot of compiler detection data.

7.3 Performance

Fig. 11 shows the total time needed to execute the tool over the 50 GEMM benchmarks. The y-axis is plotted in log scale. It can be observed from the figure that our compiler is significantly faster than ATC (45x) and FACC* (244x). There are two main reasons for being much faster than ATC. First, ATC performs input size detection by passing inputs of random sizes to the function under test, and executes it with JIT. ATC records when it receives an out-of-bounds error, and experiments until it detects the correct size. On the other hand, we perform size detection using a single replacement pass on a copy of the code, which records the input sizes in a file. The second reason is that ATC generates random inputs to test the function and records the output. This input randomness leads to the necessity of performing many experiments. For our approach , however, we rely on input and output pairs which utilize the mathematical properties of matrices. Therefore, the number experiments we require is greatly reduced.

Table 2. GEMV Test Cases

Algorithm	Code	LoC	Optimizations	Matched?
Naive	1	15	None	yes
	2	26	None	yes
	3	14	None	yes
	4	30	None	yes
	5	19	None	yes
	6	22	None	yes
Parallel	7	29	OpenMP	yes
	8	48	OpenMP	yes
	9	42	OpenMP	yes
	10	44	OpenMP	yes
	11	26	C++ threads	no
Unrolled	12	36	None	yes
	13	34	None	yes
	14	20	None	yes
	15	76	None	yes
Vectorized	16	84	Intrinsics (AVX2)	yes
	17	156	Intrinsics (AVX2)	no
	18	47	Intrinsics (AVX2)	yes
	19	35	Intrinsics (SSE)	yes
	20	188	Intrinsics (SSE)	yes

Table 3. gemv codes

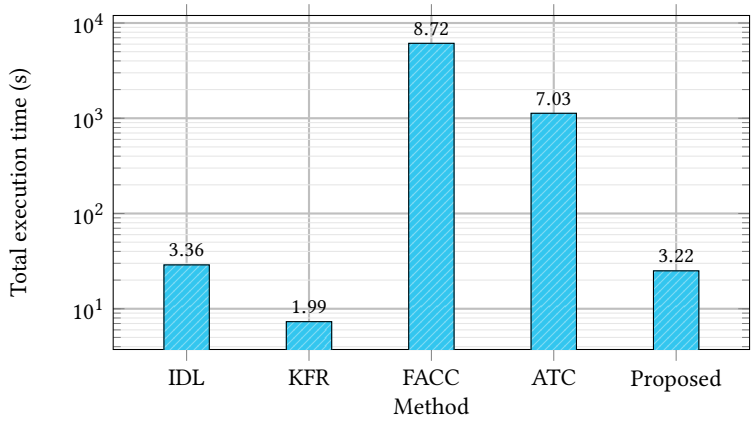


Fig. 11. Bar plot of execution time (log scale).

8 Hardware Evaluation and Benchmarking

8.1 Experimental Setup

We implement our accelerator on AMD/Xilinx Alveo u280 using 21 HBM channels (19 for the matrix, 2 for the input and result vectors), offering a memory bandwidth of 272 GB/s. Further scaling to more than 21 HBM channels is possible, at the expense of degrading the design frequency. The accelerator is implemented in HLS using Xilinx Vitis [**?**] 2022.2. The input vector buffer and the output vector buffer sizes are both 16 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 design space exploration. The pack size is 8 for sparse input and 16 for dense input, since one HBM channel delivers 512 bits per access. Our accelerator runs at runs at a frequency of 260 MHz.

8.1.1 CPU and GPU Baselines. For sparse operations, we compare our accelerator 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. GPU results are reported from

[5], where the experiments were run 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.

For dense operations, we compare against vendor-provided hipBLAS GPU library for dense linear algebra on GPUs. The results are reported from [1], where the experiments were conducted on an AMD MI50 GPU with 16GB HBM2 memory, 60 compute units, and 3840 stream processors with peak frequency of 1 GHz.

8.1.2 FPGA Baselines. We are not aware of any recent work which performs matrix-vector multiplication with an optimization for both sparse and dense inputs. Therefore, we compare our accelerator with state-of-the-art FPGA sparse accelerators — HiSparse [5], Serpens [14], FCCM23 [11], and Vitis Sparse Library (VSL) [18]. All baselines target FPGAs with HBM. Similar to our design, HiSparse uses fixed-point fixed-point data types. Serpens and FCCM23 use FP32 arithmetic. VSL is an optimized HLS library released as part of the Vitis 2020.2. All baselines are open source, except for FCCm23. Additionally, the open source version of Serpens requires building with tapa [2]. In our experiments, we encountered some version mismatch issues for building it with new releases of tapa. The released bitstream requires older versions of the Alveo platforms. Therefore, we use the reported results in the paper.

In our experiments, we were unable to run VSL with large matrices so we only compare with it on small datasets. We compile VSL on a Xilinx Alveo U280 platform with 16 HBM channels and 2 DDR channels providing 268 GB/s bandwidth in total. For Serpens, the authors implement a Vitis-based version and another version which uses tapa [2] to increase frequency. We use the metrics for the Vitis-based version for fairness of comparison with the other works.

8.1.3 Metrics. (1) Throughput, measured in Giga operations per second (GOPS). (2) Bandwidth efficiency, measured by throughput per unit bandwidth, in MOPS/GBPS. (3) Energy efficiency, measured by throughput per unit power, in GOPS/W.

8.1.4 Datasets. Table 4 lists the matrix benchmarks used for the evaluation. googleplus, hollywood, and pokec are social network graphs, widely used in benchmarking graph processing systems. mouse-gene is a graph from computational biology. ogbl-ppa and ogbn-products are from OGB [9], a benchmark suite for emerging graph neural networks. transformer-x is one layer from a compressed Transformer [16] model with a sparsity of x%.

Dataset	Size	Density	Dataset	Size	Density
transformer-50	512 × 33K	50%	transformer-70	512 × 33K	30%
transformer-60	512 × 33K	40%	transformer-80	512 × 33K	20%
transformer-90	512 × 33K	10%	transformer-95	512 × 33K	5%
tsopf-rs-b2383	38K × 38K	1.113×10^{-3}	mouse-gene	45K × 45K	1.42%
crankseg-2	63K × 63K	1.75×10^{-3}	googleplus	108K × 108K	1.2×10^{-3}
ogbl-ppa	576K × 576K	127.9×10^{-6}	hollywood	1069K × 1069K	98.5×10^{-6}
pokec	1632K × 1632K	11.5×10^{-6}	ogbn-products	2449K × 2449K	20.6×10^{-6}

Table 4. Benchmark datasets with sizes and densities.

8.2 Accelerator Evaluation

Table 5 shows a comparison of our accelerator with CPU and GPU baselines for sparse matrix benchmarks. Compared to MKL, our implementation has 6x higher throughput and 8.6x higher bandwidth efficiency. Compared to cuSPARSE, we achieve 1.5x higher throughput and 3.6x higher bandwidth efficiency.

We now compare our accelerator to the accelerators dedicated for SpMV. We show the results for every baseline in a separate table to allow for a fair comparison, since not all benchmarks results are reported for all baselines. Table 6 shows a comparison with VSL. Our implementation is 2.2x faster than VSL, with 3.33x higher bandwidth efficiency. Regarding HiSparse, Table 7 shows that we outperform it by 1.5x in terms of throughput, and 1.9x in terms of bandwidth efficiency. Tables 8 and 9 illustrate how our architecture compares to Serpens and FCCM23 respectively. We achieve roughly the same performance in both cases. The FCCM23 architecture was not evaluated for larger benchmarks, therefore it is not clear how to would perform for matrix sizes larger than 63K x 63K. In summary, we conclude that for sparse benchmarks, our architecture achieves the same performance as that of dedicated sparse accelerators in worst cases, and up to 2.2x higher throughput in best cases.

Dataset	Throughput (GOPS)			Bandwidth Efficiency (MOPS/GBPS)		
	MKL (CPU)	cuSparse (GPU)	Ours	MKL (CPU)	cuSparse (GPU)	Ours
Transformer-50	5.9	26.9	34.31	20.9	55.5	133.87
Transformer-60	5.6	21.5	31	19.9	44.5	129.96
Transformer-70	5.2	17.7	26.48	18.3	36.6	131.25
Transformer-80	4.1	19.4	24.54	14.6	40.1	118.13
Transformer-90	2.3	13.6	15.87	8.1	28	118.13
Transformer-95	1.2	10.7	7.05	4.3	22.2	78.75
mouse-gene	12.1	29	42.51	43	60	132.05
google-plus	5.1	27.2	32.5	18	56.4	132.51
ogbl-ppa	4.1	18	29.65	14.7	37.2	132.75
hollywood	4.4	22.6	36.18	15.6	46.6	134.08
pokec	3	10.5	15.24	10.7	21.8	133.68
ogbn-products	3.1	5	26.8	11	10.3	133.98
Geometric mean	4.06	16.75	**24.5**	14.42	34.62	**124.6**

Table 5. Performance comparison between CPU (MKL), GPU (cuSparse), and our implementation, including throughput and bandwidth efficiency.

Dataset	Throughput (GOPS)		Bandwidth Efficiency (MOPS/GBPS)	
	Vitis	Ours	Vitis	Ours
Transformer-50	17.5	34.31	65.2	133.87
Transformer-60	14.6	31	54.4	129.96
Transformer-70	13.0	26.48	48.7	131.25
Transformer-80	10.5	24.54	39.1	118.13
Transformer-90	5.8	15.89	21.8	118.13
Transformer-95	3.3	7.05	12.4	78.75
Geometric mean	9.34	**20.64**	34.96	**116.57**

Table 6. Performance comparison between Vitis Sparse Library and our implementation, including throughput and bandwidth efficiency.

Dataset	Throughput (GOPS)		Bandwidth Efficiency (MOPS/GBPS)	
	HiSparse	Ours	HiSparse	Ours
Transformer-50	21.9	34.31	84.7	133.87
Transformer-60	18.9	31	73.4	129.96
Transformer-70	16.5	26.48	63.9	131.25
Transformer-80	14.8	24.54	57.4	118.13
Transformer-90	9.7	15.89	37.8	118.13
Transformer-95	5.7	7.05	22.0	78.75
mouse-gene	27.2	42.52	105.4	132.05
google-plus	21.2	32.5	82.2	132.51
ogbl-ppa	24.4	29.65	94.6	132.75
hollywood	24.9	36.18	96.7	134.08
pokec	11.2	15.24	43.6	133.68
ogbn-products	20.6	26.8	79.9	133.98
Geometric mean	16.64	**24.5**	64.55	**124.6**

Table 7. Performance comparison between HiSparse and our implementation, including throughput and bandwidth efficiency.

We now focus on dense benchmarks results. Matrix sizes ranging between 1024 x 1024 to 16K x 16K are tested. We attempted to obtain the performance of HiSparse with such benchmarks. However, we could only successfully run the 1024 x 1024 benchmark. Table 10 shows the throughput comparison of our architecture, HiSparse, and the hipBLAS GPU library. Our accelerator achieves 2.6x higher throughput than HiSparse. In comparison to hipBLAS, we achieve 1.26x higher throughput. The table additionally report bandwidth efficiency for our implementation, where we achieve a geometric mean of 131.17. Although we were unable to measure the dense benchmarks results for the rest of the FPGA baselines, it is very much expected that they, similar to HiSparse, achieve lower throughput than ours. The reason for this is that their dataflow is optimized for sparse benchmarks. All of the baselines can receive at most 8 matrix elements per HBM channel at a time. In contrast, we can process 16 dense elements or 8

Dataset	Throughput (GOPS)	
	Serpens	Ours
tsopf-rs-b2383	44.39	47.2
mouse-gene	42.26	42.52
google-plus	14.71	32.5
ogbl-ppa	42.26	29.65
hollywood	36.2	36.18
pokec	14.29	15.24
ogbn-products	39.9	26.8
Geometric mean	**30.41**	**31.17**

Table 8. Performance comparison between Serpens and our implementation.

Dataset	Throughput (GOPS)	
	FCCM'23	Ours
mouse-gene	38	42.52
tsopf-rs-b2383	47	47.2
crankseg-2	50.6	43.6491
Geometric mean	**44.88**	**44.41**

Table 9. Performance comparison between FCCM23 and our implementation.

sparse elements. Additionally, many of the baselines depend on shuffling and routing the sparse elements to their corresponding vector banks. Such optimizations fit sparse benchmarks, but degrade the performance for dense benchmarks.

Dataset	Throughput (GOPS)			Bandwidth Efficiency (MOPS/GBPS)
	HiSparse	hipBLAS	Ours	Ours
1024 x 1024	6.99301	15	18.47	256
2048 x 2048	-	30	47.82	256
4096 x 4096	-	60	86.61	264
8192 x 8192	-	93	104.49	268
16k x 16k	-	105	107.74	268
Geometric mean	-	48.33	**61.24**	131.17

Table 10. Performance comparison for dense matrix-vector multiplication between HiSparse, hipBLAS, and our implementation, including throughput and bandwidth efficiency.

In summary, we conclude that in terms of throughput and bandwidth efficiency, our architecture outperforms dedicated sparse accelerators, as well as several CPU and GPU libraries. In terms of FPGA resource utilization, Table 11 shows that the consumed resources by our architecture are comparable to the FPGA baselines, although our architecture supports optimization of both sparse and dense inputs. Finally, we examine the power and energy efficiency of the architectures. Table 12 shows the measured power consumption and energy efficiency of MKL, cuSPARSE, HiSparse, Serpens, and our implementation. Our architecture is 48x and 4.8x more energy-efficient than MKL and cuSPARSE. Additionally, our accelerator is 1.3x more energy-efficienct that HiSparse. Compared to Serpens, we achieve 76% of its energy efficiency.

Implementation	LUT %	FF %	DSP %	BRAM %	URAM %	Frequency (MHz)
HiSparse	47.27	22.7	7.63	7.22	53.33	237
Serpens (Vitis version)	15	14	8	36	40	223
FCCM23	38	45	7	23	40	310
Ours	45	25.05	7.44	20.11	47.5	260

Table 11. Resource utilization comparison among different implementations.

	HiSparse	Serpens	MKL 32 threads	cuSPARSE	Ours
Power (W)	45	48	276	153	52
Energy efficiency (GOPS/W)	0.37	0.63	0.01	0.1	0.48

Table 12. Comparison of power and energy efficiency among different implementations.

References

[1] Cade Brown, Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. 2020. Design, Optimization, and Benchmarking of Dense Linear Algebra Algorithms on AMD GPUs. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC43674.2020.9286214

[2] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. 2021. Extending High-Level Synthesis for Task-Parallel Programs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 204–213. https://doi.org/10.1109/FCCM51124.2021.00032

[3] Chris Cummins, Zacharias Fisches, Tal Ben-Nun, Torsten Hoefler, Michael O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Thirty-eighth International Conference on Machine Learning (ICML)*.

[4] João P. L. De Carvalho, Braedy Kuzma, Ivan Korostelev, José Nelson Amaral, Christopher Barton, José Moreira, and Guido Araujo. 2021. KernelFaRer: Replacing Native-Code Idioms with High-Performance Library Calls. *ACM Trans. Archit. Code Optim.* 18, 3, Article 38 (jun 2021), 22 pages. https://doi.org/10.1145/3459010

[5] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. 2022. High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV. 54–64. https://doi.org/10.1145/3490422.3502368

[6] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. 2018. Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 139–153. https://doi.org/10.1145/3173162.3173182

[7] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (may 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[8] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. *FPGA. ACM International Symposium on Field-Programmable Gate Arrays* 2021, 81–92. https://doi.org/10.1145/3431920.3439289

[9] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: datasets for machine learning on graphs. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1855, 16 pages.

[10] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9. https://doi.org/10.1109/ICCAD51958.2021.9643582

[11] Abhishek Kumar Jain, Chirag Ravishankar, Hossein Omidian, Sharan Kumar, Maithilee Kulkarni, Aashish Tripathi, and Dinesh Gaitonde. 2023. Modular and Lean Architecture with Elasticity for Sparse Matrix Vector Multiplication on FPGAs. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 133–143. https://doi.org/10.1109/FCCM57271.2023.00023

[12] Pablo Antonio Martínez, Jackson Woodruff, Jordi Armengol-Estapé, Gregorio Bernabé, José Manuel García, and Michael F. P. O'Boyle. 2023. Matching Linear Algebra and Tensor Code to Specialized Hardware Accelerators. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction* (Montréal, QC, Canada) *(CC 2023)*. Association for Computing Machinery, New York, NY, USA, 85–97. https://doi.org/10.1145/3578360.3580262

[13] Tristan Ravitch, Steve Jackson, Eric Aderhold, and Ben Liblit. 2009. Automatic generation of library bindings using static analysis. *SIGPLAN Not.* 44, 6 (jun 2009), 352–362. https://doi.org/10.1145/1543135.1542516

[14] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. 2022. Serpens: a high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 211–216. https://doi.org/10.1145/3489517.3530420

[15] V. STRASSEN. 1969. Gaussian Elimination is not Optimal. *Numer. Math.* 13 (1969), 354–356. http://eudml.org/doc/131927

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[17] Jackson Woodruff, Jordi Armengol-Estapé, Sam Ainsworth, and Michael F. P. O'Boyle. 2022. Bind the gap: compiling real software to hardware FFT accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 687–702. https://doi.org/10.1145/3519939.3523439

[18] Xilinx. 2021. Vitis Sparse Library. https://xilinx.github.io/Vitis_Libraries/sparse/2021.1/overview.html.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009