Random Access Schemes for Efficient FPGA SpMV Acceleration

Yaman Umuroglu*, Magnus Jahre

Department of Computer and Information Science
Norwegian University of Science and Technology
Trondheim, Norway

Abstract

Utilizing hardware resources efficiently is vital to building the future generation of high-performance computing systems. The sparse matrix – dense vector multiplication (SpMV) kernel, which is notorious for its poor efficiency on conventional processors, is a key component in many scientific computing applications and increasing SpMV efficiency can contribute significantly to improving overall system efficiency. The major challenge in implementing SpMV efficiently is handling the input-dependent memory access patterns, and reconfigurable logic is a strong candidate for tackling this problem via memory system customization. In this work, we consider three schemes (all off-chip, all on-chip, caching) for servicing the irregular-access component of SpMV and investigate their effects on accelerator efficiency. To combine the strengths of on-chip and off-chip random accesses, we propose a hardware-software caching scheme named *NCVCS* that combines software preprocessing with a nonblocking cache to enable highly efficient SpMV accelerators with modest on-chip memory requirements. Our results from the comparison of the three schemes implemented as part of an FPGA SpMV accelerator show that our scheme effectively combines the high efficiency from on-chip accesses with the capability of working with large matrices from off-chip accesses.

Keywords: sparse matrix–vector multiplication, FPGA, memory system, cache, efficiency

1. Introduction

High energy efficiency is a key challenge for building the next generation of computing systems where exascale performance levels are desired [1]. One strategy for achieving this goal is to build highly efficient primitives (accelerators) to implement computational kernels commonly encountered across applications. Sparse Matrix – Vector Multiplication (SpMV) is one such computational kernel, which is widely encountered in the scientific computation domain and frequently constitutes a bottleneck for such applications [2]. Besides the classical use in numerics, SpMV can be used to implement a variety of graph algorithms by customizing the add and multiply operators [3, 4, 5].

A defining characteristic of the SpMV kernel is the *irregular memory access pattern* caused by the sparse storage formats. A critical part of the kernel depends on accesses to memory addresses that correspond to non-zero element locations of the matrix, which are only known at runtime. These accesses constitute a large problem especially when SpMV is used for analysis of social graphs [5], since the resulting matrices are unstructured and are growing in size as part of the Big Data trend. The kernel is otherwise characterized by little data reuse and large per-iteration data requirements [2], which makes the performance memory bandwidth-bound. Storing the kernel inputs and outputs in high-capacity high-bandwidth DRAM is considered a cost-effective solution [6]; however, the burst-optimized architecture of DRAM contrasts with the fine-grained SpMV random accesses. Engineers and computer architects thus face an ever-growing "irregularity wall" in the quest for enabling efficient SpMV implementations.

Recently, there has been increased interest in FPGA-based acceleration of computational kernels. The primary benefit from FPGA accelerators is the ability to create customized memory systems and datapaths that align well with the requirements of each kernel, enabling stall-free and highly efficient execution. From the perspective of the SpMV kernel, the ability to deliver high external memory bandwidth, embedded SRAM blocks (which we refer to as *on-chip memory (OCM)*) to provide high-bandwidth random accesses and run-time specialization via dynamic reconfiguration are attractive properties.

Several FPGA implementations for the SpMV kernel have been proposed, either directly for SpMV or as part of larger algorithms like iterative solvers [7, 8, 9], some of which present order-of-magnitude better energy efficiency and comparable performance to CPU and GPGPU solutions. These accelerators tackle the irregular access problem by buffering the entire random-access data in OCM. Unfortunately, this strategy is limited to operat-

*Corresponding author
Email address: yamanu@idi.ntnu.no (Yaman Umuroglu)

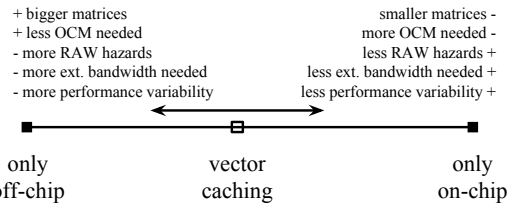


Figure 1: Tradeoffs in SpMV result vector random accesses.

ing on smaller matrices where the random-access data can fit in OCM. An alternative solution is performing random accesses to DRAM and mitigating the high latencies with many in-flight requests and memory-level parallelism. This approach is not limited by FPGA OCM size, but exhibits variable efficiency depending on the matrix structure since DRAM is not optimized for fine-grained random accesses. Finally, our previous work in [11] proposed a specialized vector caching scheme to take advantage of SpMV-specific reuse and access patterns, which showed promising results despite the limited evaluation in simulation and low efficiency for caches under a certain size due to lack of memory-level parallelism.

In essence, these three approaches line up along a tradeoff axis as illustrated in Figure 1. In this paper, we adopt a processor-like view of the SpMV reducer hardware for each random-access approach to better understand the tradeoffs and facilitate comparison. To combine the strengths of the on-chip-only and off-chip-only accesses, we extend upon our previous work in [11] to build the *Nonblocking Cooperative Vector Caching Scheme* (*NCVCS*). Afterwards, we compare these approaches by implementing them as part of a fully-functional FPGA SpMV accelerator. Our results indicate that *NCVCS* effectively balances the use of off- and on-chip memory bandwidth, achieving 73% average computational efficiency across a set of large sparse matrices while occupying half of the OCM on a Zynq Z7020. This work makes the following new contributions:

- A processor-like view of SpMV reducer hardware to analyze and compare vector random access schemes.
- A simplified preprocessing algorithm for providing cold miss skip capabilities with low overhead.
- A pipelined, nonblocking vector cache architecture for *NCVCS* that supports cold miss skip as well as exploiting memory-level parallelism.
- Three open-source SpMV accelerator implementations that make use of different random access methods, and their evaluation with respect to performance, efficiency and resource usage on the ZedBoard.

2. Background

2.1. The SpMV Kernel and Sparse Matrix Storage

The SpMV kernel $\vec{y} = \mathbf{A} \cdot \vec{x}$ consists of multiplying an $m \times n$ sparse matrix $\mathbf{A}$ with NZ nonzero elements by

$$\begin{bmatrix} 1.1 & 0 & 0 \\ 0 & 2.2 & 3.3 \\ 4.4 & 0 & 5.5 \end{bmatrix}$$

```
colptr={0 2 3 5}
values={1.1 4.4 2.2 3.3 5.5}
rowind={0 2 1 1 2}
```

```
for(j=0 to n-1)
  for(i=colptr[j] to colptr[j+1])
    y[rowind[i]] += values[j] * x[j]
```

Figure 2: A matrix, its CSC representation and SpMV pseudocode. The clause causing random-access to `y` is highlighted.

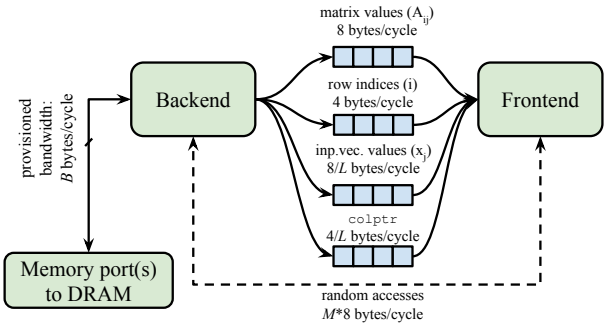


Figure 3: Decoupled SpMV accelerator architecture.

a dense vector $\vec{x}$ of size n to obtain a result vector $\vec{y}$ of size m. The sparse matrix is commonly stored in a format which allows storing only the nonzero elements of the matrix. Many storage formats for sparse matrices have been proposed, some of which specialize on particular sparsity patterns, and others suitable for generic sparse matrices. Among the most popular storage formats for generic sparse matrices are Compressed Sparse Row (CSR) and its column-major counterpart Compressed Sparse Column (CSC). Figure 2 illustrates a sparse matrix, its representation in the CSC format, and the pseudocode for performing column-major SpMV. We use the `variable` notation to refer to CSC SpMV data in memory (e.g. `values`, `rowind`, `colptr`, `x`, `y`). As highlighted in the figure, the result vector `y` is accessed depending on the `rowind` values, causing the random access patterns that are central to this work.

In this paper, we will focus on column-major sparse matrix traversal (in line with [9, 6, 12]) and the CSC storage format with 4-byte indices and double-precision floating point values. Column-major traversal enables simpler SpMV datapaths by interleaving different rows the adder pipeline (see Section 3) and permits maximum temporal reuse of the input vector and `colptr` values; each of these values gets reused $L = \frac{NZ}{n}$ times.

2.2. Bandwidth-Bound Performance and Efficiency

SpMV has a low computation-to-memory ratio and typically requires double-precision floating point arithmetic when used as part of scientific computation. As such, the inputs can be very large and are stored in external DRAM, which makes the kernel performance bounded by DRAM bandwidth. Our previous work in [12] proposed the decou-

pled architecture illustrated in Figure 3 to match the computational capability with memory bandwidth and identify sources of inefficiency. Here, the *backend* is provisioned with B bytes per cycle of external memory bandwidth, and is responsible for fetching matrix and input vector data from DRAM. The backends feeds a *frontend* with data, which performs the actual computation. In order to keep a frontend supplied with enough data to perform one multiply and one add per cycle, the accelerator should be provisioned with $B \geq 12 + \frac{12}{L}$ bytes per cycle. In this work, we consider an extended SpMV frontend where some or all result vector accesses may be serviced from DRAM to allow working with bigger matrices, with random accesses indicated with the dashed line in Figure 3. If M is the ratio of random vector accesses that go to DRAM, achieving full throughput with $M > 0$ requires more bandwidth, as described in Equation 1.

$$B \geq 8 \cdot M + 12 + \frac{12}{L} \tag{1}$$

However, DRAM is not optimized for fine-grained random accesses and may not deliver the desired bandwidth, which causes the frontend to stall and decreases efficiency. The goal of this work is to devise a random access scheme that minimizes these stalls by minimizing M and exploiting memory-level parallelism, thus maximizing efficiency and performance.

2.3. Sparse Matrix Preprocessing

Since the random-access behavior of the SpMV kernel is dependent on the particular sparse matrix used, a preprocessing step is often introduced to optimize performance for a particular matrix. Many different forms of preprocessing for SpMV have been proposed to date. A common form of preprocessing is using the Cuthill–McKee (CMK) algorithm [13] or its reverse (RCMK) to reorder the matrix for smaller bandwidth. Oliker et al. [14] compares the effects of different forms of matrix partitioning and reordering on parallel supercomputers. Kourtis et al. [15] and Wilcock and Lumsdaine [16] propose using preprocessing to compress the matrix and reduce the required memory bandwidth. Pichel et al. [17] investigate the performance effects of reordering techniques on GPUs, and report speedups of up to 2.6x.

Preprocessing has a certain time cost, as the original matrix must be first read into memory, then the preprocessing performed and the results written back. Fortunately, algorithms that make heavy use of SpMV tend to do so iteratively, i.e. they multiply the same sparse matrix with many different vectors, which enables ameliorating the cost of preprocessing across speed-ups in each SpMV iteration. Toledo et al. [18] reports the cost of preprocessing to be 1–3 times the SpMV cost for simpler reordering techniques (CMK, RCMK), 4–15 times for blocking, and 20–200 times the SpMV cost for a nested dissection-type reordering. They estimate that the speedup from blocking pays its preprocessing cost in approximately 75 iterations. We also adopt a preprocessing step in our scheme to enable optimizing for a given sparse matrix, but unlike previous work, our preprocessing stage produces extra information to enable specialized cache operation instead of changing the matrix structure.

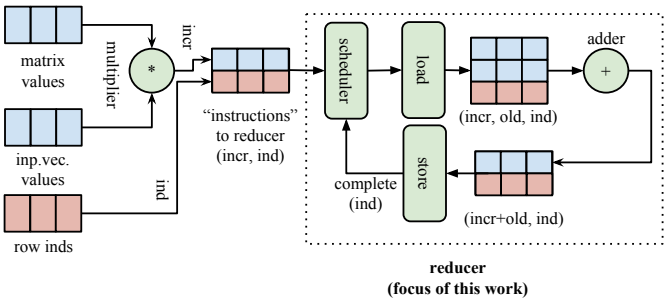


Figure 4: Overview of a column-major SpMV accelerator frontend.

3. Frontend Architectures and Random Access

The frontend of a column-major SpMV accelerator is essentially a multiply-accumulator with feedback from a random-access memory, as illustrated in Figure 4. The first part of the frontend is the multiplier, which produces the value we refer to as `incr` by multiplying each matrix nonzero with its corresponding input vector element. This does not pose a significant design problem as the multiplier only needs to support backpressure and can have an arbitrary number of pipeline stages. The pair `(incr, ind)` is passed to the *reducer* to perform `y[ind] += incr`. If we think of the reducer as a very simple processor, `(incr, ind)` can be thought of as an "accumulate" instruction that increments the register `ind` by `incr`. The reducer loads the current value of `old=y[ind]`, compute `incr+old`, and write `y[ind]=incr+old` to complete each instruction. Note that instructions with the same `ind` target the same register, which constitutes a read-after-write (RAW) hazard that prevents these instructions from running in parallel. A *scheduler* prevents this by controlling instruction dispatch, monitoring which `ind` values are in-flight (i.e. not yet completed) and stalling[1] incoming instructions with hazards. The scheduler *issue window* determines the maximum number of in-flight values.

In the following subsections, we will use this processor-like view of the reducer to view the three random-access policies proposed in prior work on FPGA SpMV acceleration in a common frame. Our intent here is not to exhaustively categorize how accesses could be handled, but rather illustrate how response latency and its variability affect the frontend architecture. For instance, random accesses could also be serviced from a large, external SRAM

[1]Dynamically re-ordering the dispatches can give fewer stalls, though we do not investigate their benefit in this work.

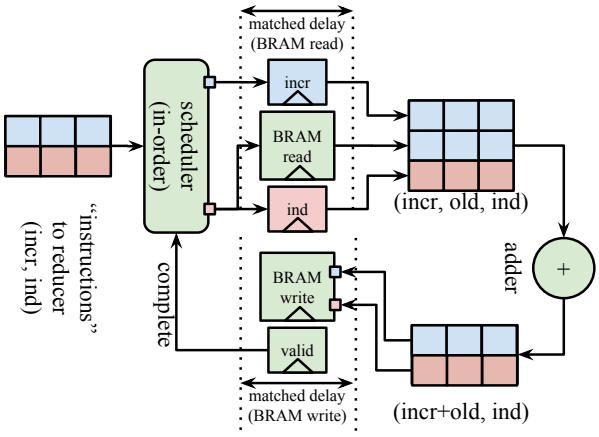


Figure 5: Architecture of a *BufferAll* reducer.

chip, which would have a deterministic random-access latency that is larger compared to OCM but less compared to DRAM.

3.1. All on-chip: BufferAll

Figure 5 illustrates a *BufferAll* reducer, which uses FPGA OCM to store and access the entire result vector during SpMV. This offers deterministic and low latency for random reads and writes, which results in highly efficient accelerator implementations. Each instruction is completed after a fixed number of cycles, i.e read/write complete signals can be generated by matching the BRAM read/write delays with a shift register chain and all responses return in-order, which makes the scheduler simple. An issue window of size equal to adder stages plus read and write latency is enough to achieve full throughput, which results in fewer RAW hazard stalls. Keeping all `y` accesses on-chip (i.e. $M = 0$ in Equation 1) frees up external memory bandwidth for reading more matrix data and enables higher performance. Unfortunately, this is only possible if the entire result vector `y` can fit within FPGA OCM, which limits the maximum number of rows in the matrix that the accelerator can process.

Much of the prior work on FPGA SpMV acceleration [8, 9, 19, 20, 7] uses OCM for buffering the entire random accessed component and the maximum matrix dimension (rows in column-major, columns in row-major) becomes limited by OCM capacity. The largest sparse matrix dimension in the evaluation is 8127 in the work by Jain-Mendon and Sass [20], 16K in the work by Fowers et al. [7], 17281 in the work by Zhang et al. [19] and 63838 in the work by Chow et al. [8], whereas we consider sparse matrices of up to a million rows in our evaluation. The work by Dorrance et al. [9] illustrates both the benefits and drawbacks of this strategy. While they report a high average E of 92%, the matrix height is limited by OCM (reported maximum 218K rows) and they resort to reduced precision data types to allow larger sparse matrices.

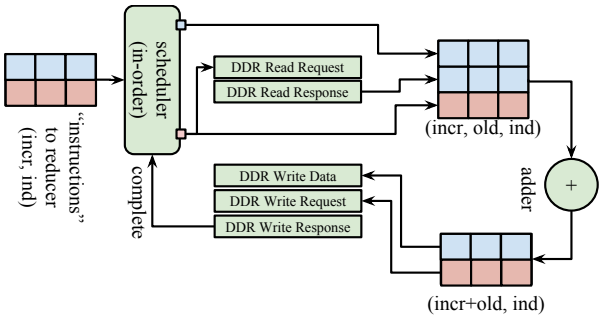


Figure 6: Architecture of a *BufferNone* reducer.

3.2. All off-chip: BufferNone

A *BufferNone* reducer, which services the random accesses directly from DRAM, is depicted in Figure 6. Although DRAM latency is much higher compared to OCM, DRAM-based memory systems permit multiple outstanding requests to mitigate latency. By using a large issue window, the accelerator can take advantage of memory-level parallelism and achieve high throughput, but a large issue window may result in more RAW hazards. Due to the variability of latencies, read/write-completion must be signaled directly by the memory system instead of delay-matched registers. Responses may also be returned out-of-order to maximize DRAM efficiency, though some systems can handle this internally and return in-order to allow for simpler user logic, which we assume in this work. Overall, the *BufferNone* approach does not require large amounts of OCM and the matrix size is only limited by the DRAM capacity, which is a natural limitation for high-performance systems. The primary disadvantage is the additional consumption of valuable external memory bandwidth ($M = 1$) and performance variability due to DRAM bank conflicts and increased RAW hazards. It should also be noted that DRAM is optimized for large bursts; fine-grained random accesses may not achieve peak bandwidth.

To our knowledge, this strategy has been only used for row-major and mixed-traversal SpMV accelerators. These may handle RAW hazards differently but still use memory-level parallelism to mitigate latency. The work by Halstead and Najjar [10] uses many in-flight requests for high-throughput DRAM random accesses to the input vector. They report efficiency ranging from 10% to 72% depending on the particular sparse matrix used, with an average of 48%. Townsend and Zambreno [21] also use multiple memory requests to access the input vector in DRAM, although they split the matrix into 16-row chunks to mix column-major and row-major traversal and use data compression to enhance performance. They report performance ranging from 2.1 to 13.6 GFLOPS (average 7.5 GFLOPS), which corresponds to an average efficiency of 40% for the peak performance of 19 GFLOPS.

3.3. Balanced: BufferCache

A *BufferCache* reducer uses a chunk of OCM to buffer a portion of the result vector according to some caching

policy, and uses DRAM to service the elements that are not in the cache. It has the potential to combine the strengths of *BufferAll* and *BufferNone*, though a "one size fits all" caching policy is difficult to construct due to matrix-dependent memory accesses.

Gregg et al. [6] use a simple, blocking direct-mapped cache to supply `y` values, backed by DRAM to enable scaling to large matrices. To avoid the major efficiency penalties due to cache misses blocking the entire accelerator, they proposed to split the matrix into cache-sized chunks. They report efficiency ranging from 14% to 72% and mention that splitting can introduce overheads due to increased matrix storage costs and more RAW hazard stalls. Nagar and Bakos [22] include a 8192-element direct mapped cache in their accelerator, with performance of 1.17 – 3.95 GFLOPS (corresponding to 12%–41% of the 9.6 GLOPS peak), but they do not discuss how cache misses influence performance. Finally, our previous work in [11] examined how sparse matrix structure relates to cache misses and proposed a vector caching scheme to combine software preprocessing with special hardware to avoid cold misses. We present an improved version of this vector caching scheme in Section 4.

4. The Nonblocking Cooperative Vector Caching Scheme

It is desirable to balance and combine the strengths of the *BufferAll* and *BufferNone* approaches into a *BufferCache*-type scheme and enable high efficiency on large sparse matrices without being constrained by FPGA OCM. To achieve this, we propose[2] the Nonblocking Cooperative Vector Caching Scheme (*NCVCS*), which uses a combination of software preprocessing and specialized hardware to achieve a high cache hit rate with an OCM footprint smaller than *BufferAll*. Specifically, the *preprocessing step* analyzes the matrix structure to provide a minimal estimate of the required cache capacity and mark the locations where cold cache misses will occur. The capacity estimation can be used to resize (via runtime reconfiguration) the *vector cache hardware*, which can make use of the cold miss markings and memory-level parallelism to offer high random-access performance.

In the following sections, we first describe how matrix structure relates to data reuse and cache misses, then formulate a preprocessing algorithm and describe a nonblocking cache architecture suitable for FPGAs for realizing *NCVCS* in hardware.

4.1. Row Lifetime Analysis

To relate the vector cache usage to the matrix structure, we start by defining a number of structural properties for sparse matrices. First, we note that each row has a strong correspondence to a single result vector element, i.e `y[i]` contains the dot product of row `i` with `x`. The period in which `y[i]` is used is solely determined by the period in which row `i` accesses it. This is the key observation that we use to specialize *NCVCS* for a given sparse matrix.

4.1.1. Calculating `maxAlive` and `maxColSpan`

For a matrix with column-major traversal, we define the *aliveness interval* of a row as the column range between (and including) the columns of its first and last nonzero elements, and will refer to the interval length as the *span*. Figure 7a illustrates the aliveness intervals as red lines extending between the first and last non-zeroes of each row. For a given column `j`, we define a set of rows to be *simultaneously alive* in this column if all of their aliveness intervals contain `j`. The number of alive rows for a given column is the maximum size of such a set. Visually, this can be thought of as the number of aliveness interval lines that intersect the vertical line of a column. For instance, the dotted line corresponding to column 5 in Figure 7a intersects 8 intervals, and there are 8 rows alive in column 5. Finally, we define the *maximum simultaneously alive rows* of a sparse matrix, further referred to as `maxAlive`, as the largest number of rows simultaneously alive in any column of the matrix. Incidentally, `maxAlive` is equal to 8 for the matrix given in Figure 7a – though the alive rows themselves may be different, no column has more than 8 alive rows in this example.

Calculating `maxAlive` requires preprocessing the matrix. If the accelerator design is not under very tight OCM constraints, it may be desirable to estimate `maxAlive` instead of computing the exact value in order to reduce the preprocessing time. If we define aliveness interval and span for columns as was done for rows, the largest column span of the matrix `maxColSpan` provides an upper bound on `maxAlive`. The column 3 in Figure 7a has a span of 14, which is `maxColSpan` for this matrix.

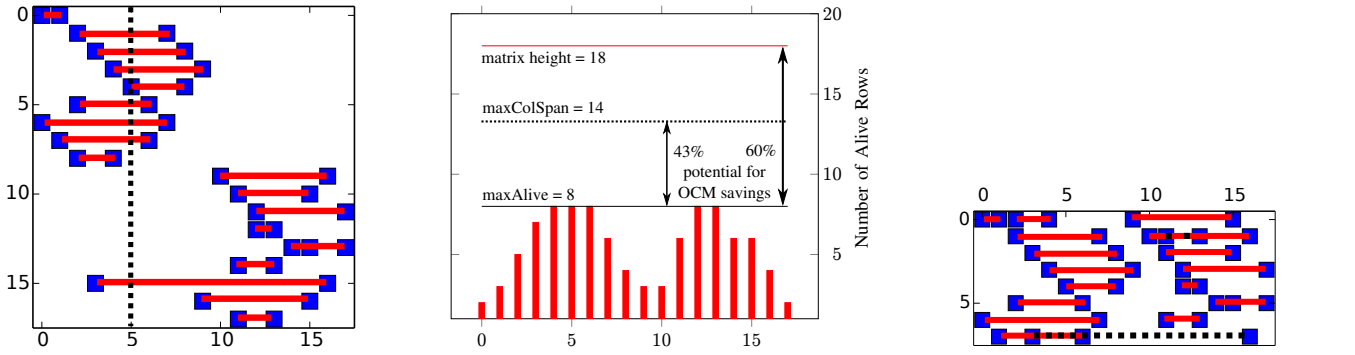
4.2. Avoiding Vector Cache Misses

We now use the canonical cold/capacity/conflict classification to break down cache misses into three categories and explain how accesses to the result vector relate to each category. For each category, we will describe how misses can be related to the matrix structure and avoided where possible.

4.2.1. Cold Misses

Cold (compulsory) misses occur when a vector element is referenced for the first time, at the start of the aliveness interval of each row. For matrices with very few elements per row, cold misses can contribute significantly to the total cache misses. Although this type of cache miss is considered unavoidable in general-purpose caching, a special case exists for SpMV. Consider the column-major SpMV operation $y = Ax$ where the y vector is random-accessed using the vector cache. The initial value of each y element is zero, and is updated by adding partial sums for

[2] *NCVCS* is an extension of our previous work in [11], and avoids the shortcomings of high LUTRAM consumption and transpose operations in preprocessing, as well as improving performance for small cache sizes by exploiting memory-level parallelism.



(a) Matrix and row lifetimes. (b) Alive rows in each column. (c) Conflicts for an 8-line direct-mapped cache.

Figure 7: Matrix Pajek/GD01_b (from [23]) and its vector caching analysis.

each nonzero in the corresponding matrix row. If we can distinguish cold misses from the other miss types at runtime, we can avoid them completely: a cold miss to a y element will return the initial value, which is zero[3]. Recognizing misses as cold misses is critical for this technique to work. We propose to accomplish this by introducing a *start-of-row bit* marked during preprocessing, as described in Section 4.3.

4.2.2. Capacity Misses

Capacity misses occur due to the cache capacity being insufficient to hold the SpMV result vector working set. Therefore, the only way of avoiding capacity misses is ensuring that the vector cache is large enough to hold the working set. Caching the entire vector (the *BufferAll* strategy) is straightforward, but is not an accurate working set size estimation due to the sparsity of the matrix. While methods exist to attempt to reduce the working set of the SpMV operation by permuting the matrix rows and columns, they are outside the scope of this paper. Instead, we will concentrate on how the working set size can be estimated. This estimation can be used to reconfigure the FPGA SpMV accelerator to use less OCM, which can be reallocated for other components. In this work, we make the assumption that a memory location is in the working set if it will be reused at least once to reap all the caching benefits. Thus, the cache must have a capacity of at least `maxAlive` to avoid all capacity misses. This requires the computation of `maxAlive` during the preprocessing phase. If OCM constraints are more relaxed, the `maxColSpan` estimation described in Section 4.1 can be used instead. Figure 7b shows the row lifetime analysis for the matrix in Figure 7a and how different estimations of the required capacity yield different OCM savings compared to *BufferAll*.

4.2.3. Conflict Misses

Conflict misses arise when two simultaneously alive vector elements map to the same cache line. This is determined by the nonzero pattern, number of cachelines and the chosen hash function. Assuming that the vector cache has enough capacity to hold the working set, avoiding conflict misses is an associativity problem. Since content-associative memories are expensive in FPGAs, direct-mapped caches are often preferred. The conflicts arising from this type of mapping can be visualized by "folding" the matrix height to be equal to the cachelines, as shown in Figure 7c. Here, the conflicts from the example matrix on a 8-line direct-mapped cache are visible as dotted lines on cachelines 1 and 7. Our results in Section 6.4.1 suggest that conflicts are few for most matrices even with a direct-mapped cache, as long as the cache capacity is sufficient. Jouppi [24] describes how a small fully-associative memory called a *victim cache* can significantly reduce conflict misses in a direct-mapped cache, which could be implemented for cases with many conflict misses.

[3]The more general SpMV form $y = Ax + b$ can be easily implemented by adding the dense vector b after $y = Ax$ is computed.

Algorithm 1 Marking row starts or ends.

```
function MarkRowStarts(CSCMatrix A, bool reverse)
    seen[0..n − 1] ← 0
    isRowStart[0..NZ − 1] ← 0
    for e ← 0..NZ − 1 do
        nzind ← (reverse ? NZ − 1 − e : e)
        rowind ← A.rowind[nzind]
        if seen[rowind] == 0 then
            seen[rowind] ← 1
            isRowStart[nzind] ← 1
        end if
    end for
    return isRowStart
end function
```

Algorithm 2 Finding `maxAlive`.

```
function GetMaxAlive(CSCMatrix A)
    isRowStart[0..NZ − 1] ← MarkRowStarts(A, false)
    isRowEnd[0..NZ − 1] ← MarkRowStarts(A, true)
    maxAlive ← 0, currentAlive ← 0
    for e ← 0..NZ − 1 do
        currentAlive ← currentAlive + isRowStart[e] − isRowEnd[e]
        maxAlive ← Max(maxAlive, currentAlive)
    end for
    return maxAlive
end function
```

4.3. Preprocessing

Having established how the matrix structure relates to vector cache misses, we will now formulate the preprocess-

Algorithm 3 Finding `maxColSpan`.

```
function GetMaxColSpan(CSCMatrix A)
    maxColSpan ← 0, currentColSpan ← 0
    for j ← 0..n − 1 do
        firstRowInCol ← A.rowinds[A.colptr[j]]
        lastRowInCol ← A.rowinds[A.colptr[j + 1] − 1]
        currentColSpan ← lastRowInCol − firstRowInCol
        maxColSpan ← Max(currentColSpan, maxColSpan)
    end for
    return maxAlive
end function
```

ing step. We assume that the preprocessing step will be carried out by the general-purpose core prior to copying the SpMV data into the accelerator's memory space. Note that the original preprocessing algorithm in [11] was formulated on the row-major matrix representation (which requires a transpose operation) and required a priority queue. The algorithms we present here operate directly on the column-major (CSC) and with simple operations, which allow for low-overhead preprocessing.

One task that the preprocessing needs to fulfill is marking the start of each row to avoid cold misses. This is accomplished by Algorithm 1, which generates a boolean array with one element per matrix nonzero, indicating whether that nonzero is the first element of a row. We reserve the highest bit of each `rowind` value as a flag to indicate the start of a row. Although this decreases the maximum possible matrix that can be represented, it avoids introducing even more data into the already memory-intensive kernel, and can still represent matrices with over 2 billion rows for a 32-bit `rowind`. At the time of writing, this is 16x larger than the largest matrix in the University of Florida sparse matrix collection [23].

To minimize OCM usage of the cache by the methods described in Section 4.1, either `maxAlive` or `maxColSpan` needs to be computed. Algorithm 2 can be used to compute `maxAlive` by first determining matrix elements where rows start and end, then iterating over these arrays to find the maximum number of rows alive. Alternatively, Algorithm 3 can be used to compute `maxColSpan` by iterating over each column of the sparse matrix and finding the one with the greatest span, which has less preprocessing cost but tends to overestimate the needed capacity.

4.4. NCVCS *Hardware*

We now describe a nonblocking cache architecture that forms the hardware component of *NCVCS*, with several improvements over the implementation of our previous work in [11]. For the simple, blocking cache architecture presented in [11], a cache miss will cause the reducer to stall until data is fetched from DRAM. In case the cache capacity is insufficient to contain the working set size, there will be many such stalls, which decreases efficiency significantly. In this work, we address this problem with a *nonblocking* cache that allows other requests to be processed while a cache miss is handled. Additionally, cache tags were kept in combinational-read LUTRAM in [11], which made scaling to larger cache sizes difficult due to

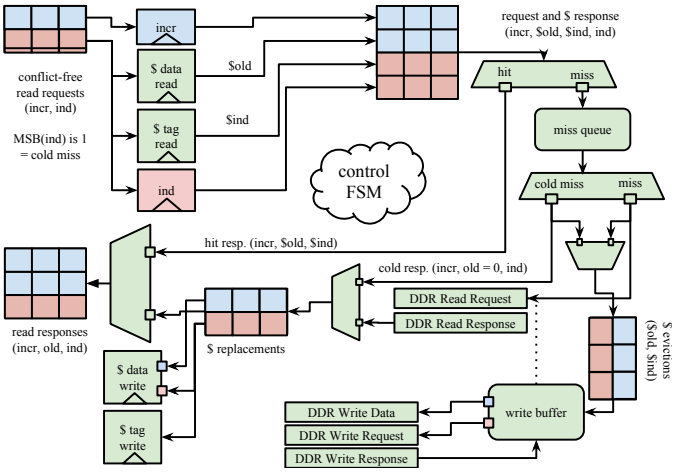


Figure 8: Nonblocking cache read path.

the resource bottleneck and frequency penalties. In this design, both result vector data and tag/valid information is kept in synchronous-read OCM and support pipelining, which frees up valuable LUTRAM and exhibits better scalability. Finally, we include a small write buffer to avoid having to wait for completion of all pending DRAM writes, which further improves upon [11]. Otherwise, we keep the design choices from [11] for associativity (direct-mapped) and write policy (write-back), since these allow for an FPGA-suitable implementation that conserves DRAM write bandwidth.

4.4.1. Cache Reads

Figure 8 illustrates the read path for the nonblocking cache. Most of the design is built to operate in a continuous data-flow manner, although a state machine is still used to direct the flow of data in the read path when necessary. Each read request to the cache includes the row index (with the most significant bit used as the start-of-row flag) and an operand to the adder. The request-to-response read path is structured as follows: first, tag and data are read from OCM, and stored in the request–response queue together with the original request. Each item in the request–response queue is sent along one of the three response paths, which are the following in order of decreasing priority:

1. **Cache hit:** If the cached tag is equal to the request tag, the data response and request index are directly sent to the read response.
2. **Cold miss:** If the start-of-row bit is set, this is a cold miss and a zero is returned as read data. The cache data and tag are updated, which may cause an eviction to DRAM.
3. **Regular miss:** Otherwise, this is a regular miss and a DRAM read request is issued. Upon read response from DRAM, the cache read response is emitted, and the cache data and tag are updated, which may cause an eviction to DRAM.

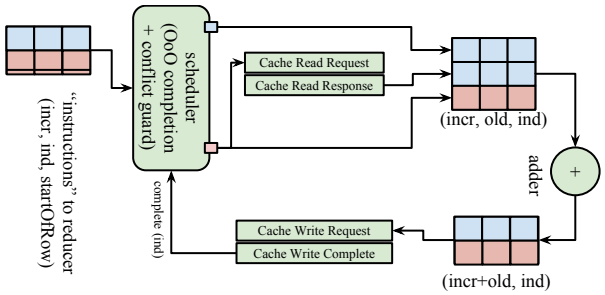


Figure 9: Architecture of the *NCVCS*-reducer.

For cache misses, all information regarding the miss (from the request–response queue) is put onto the *miss queue*, which prevents misses from blocking the request–response queue. This is the critical component that enables nonblocking cache operation, and the size of the miss queue determines how many outstanding misses the cache can have before blocking. Permitting a larger number of outstanding misses lets the cache exploit more MLP, and can improve performance when cache misses are frequent. When the read request causing the miss is ready to be served (i.e. immediately for a cold miss, or when DRAM read data returns for a regular miss) it is removed from the miss queue and its read response emitted. Due to the out-of-order operation, the original operand and row index are always emitted as part of the read response along with the returned read data.

4.4.2. Evictions and the Write Buffer

Since the cache uses a write-back policy to conserve DRAM bandwidth, there is a risk for RAW hazards on the DRAM level. Specifically, if a DRAM read request is made to a location with a pending DRAM write, an outdated (incorrect) read response will be returned. In [11] this was avoided by waiting for all pending writes to complete before issuing a DRAM read. This is detrimental for performance, especially for a nonblocking cache. Our design addresses this problem by including a *write buffer*, which is a small (8-entry) associative memory that keeps track of pending DRAM writes. The write buffer contents are checked prior to issuing DRAM reads to prevent RAW hazards. Note that the current implementation does not store the write data itself. Buffering the write data as well would act as a victim cache, which was originally proposed by Jouppi [24] to decrease conflict miss penalties in direct-mapped caches.

4.4.3. The NCVCS*-reducer and Cache Writes*

Figure 9 presents a *NCVCS*-reducer that uses the nonblocking vector cache for handling random accesses. As the nonblocking property causes out-of-order cache responses, we use a scheduler with in-order dispatch and out-of-order retirement. Additionally, we use the scheduler as a *cache conflict guard* to simplify the cache design. Instead of comparing the entire row index of the head instruction, the scheduler compares only the part that corresponds to cache index bits. This prevents instructions with the same cache index (i.e. conflict misses) from entering the reducer; all in-flight instructions map to a different cache index. Since the cache tag is allocated during read response that precedes the write, all incoming cache writes are guaranteed to hit in the cache. This makes the write path of the cache trivial, as data can be written directly to OCM without checking tags. Out-of-order completion requires the row index of the completed instruction to be signaled to the scheduler, which is generated by the cache upon a write complete.

Table 1: Characteristics of the ZedBoard.

System-on-a-Chip	Zynq Z7020
CPU core	Dual ARM Cortex-A9, 666 MHz
CPU cache	32 KB L1D+L1I, 512 KB L2
DRAM and bandwidth	512 MB DDR3, 3.2 GB/s
FPGA logic resources	53200 slice LUTs, 220 DSP slices
FPGA OCM	560 KB (140 BRAMs)

5. Experimental Setup

To evaluate and compare how *BufferAll*, *BufferNone* and *NCVCS* perform as part of a real FPGA SpMV accelerator, we implemented an accelerator system[4] that follows the architectural template in Figure 5. To characterize performance on different sparse matrices, we use the matrix suite from Williams et al. [2] which contains a variety of large sparse matrices from different domains. The properties of each matrix is listed in Table 2.

The accelerator system is deployed on the ZedBoard, whose characteristics are shown in Table 1. The Zynq chip offers dual ARM Cortex-A9 cores and FPGA fabric, both of which can access the on-board DRAM. A single ARM core running at 666 MHz was used with bare-metal software for loading matrices from the SD card, executing the preprocessing algorithm and controlling accelerator execution. Most of the SpMV accelerator hardware was built in the hardware description language Chisel [25], except a few components (large BRAMs, large FIFOs and the double-precision floating point operators) which were generated with Xilinx Core Generator and imported as Verilog blackboxes. Vivado 2014.4 was used for synthesis, place and route. The FPGA OCM (BRAM) is used for the FIFOs between the backend and frontend, for storing `y` in *BufferAll* and for storing cache data and tags in *BufferCache*. The double-precision floating point operators are pipelined with 8 stages for the multiplier and 4 stages for the adder. All accelerator variants are set to operate at 100 MHz, which corresponds to a peak performance of 200 MFLOPS (one add and one multiply per clock cycle). We provision the accelerator with a bandwidth of $B = 16$ bytes per cycle (1.6 GB/s) from two AXI high-performance

[4]The source code is available from http://git.io/vsMNJ

Table 3: Resources and frequency for best-case configurations.

Resource	*BufferAll*	*NCVCS*	*BufferNone*
Frontend LUTs	2346 (4%)	2978 (6%)	2411 (5%)
Total BRAMs	117 (84%)	77 (55%)	3 (2%)
Total LUTs	6470 (12%)	7443 (14%)	6665 (13%)
F_{max}	133 MHz	131 MHz	145 MHz

(HP) ports on the Zynq, which is rate-matched with the peak performance for the matrix suite if M is close to zero (see Section 2.2 and Equation 1).

We consider only a single SpMV processing element in our performance evaluation. The reason for this is the intricate link between parallel partitioning of sparse matrices and random accesses. Partitioning essentially creates smaller sparse matrices with new access patterns, whose performance can differ significantly. As there are many possible partitionings for each matrix, the experimental space becomes very large and makes it hard to concentrate on the differences between random vector access schemes themselves. Our results indicate that the matrices in the suite contain sufficient irregularity to study different SpMV memory behavior and interactions with the vector access schemes, even in unpartitioned form.

6. Results

We now present and discuss our experimental results, which are organized into four parts. We start with FPGA synthesis results and a best-case performance comparison of the accelerator variants, then provide a more detailed analysis on *BufferNone* and *NCVCS* performance in the last two subsections.

6.1. FPGA Resources and Frequency

Table 3 compares the LUT and BRAM resource utilization for the best-performing configurations (Section 6.2). All frontend variants include the floating-point adder (1110 LUTs) and multiplier (359 LUTs, 10 DSP slices). The logic for the scheduler and random vector access handling uses ~1000 LUTs for *BufferAll* and *BufferNone* and ~1500 LUTs for *NCVCS*, accounting for ~15% of the entire accelerator. The most marked difference for the three variants is in BRAM utilization. Note that the BRAM utilization limits the largest power-of-two sized *BufferAll* `y` to 64K[5] elements and *NCVCS* to 32K elements (smaller than *BufferAll* due to cache tag overhead).

The maximum clock frequency for each configuration is also reported in Table 3. The slightly higher F_{max} for *BufferNone* is likely due to lower BRAM usage compared to *BufferAll* and *NCVCS*. As the goal of our study is to maximize efficiency for a given amount of external memory bandwidth, we did not perform detailed timing analysis or frequency optimizations in this work. If this is desired, the latency-insensitive construction of our accelerator system can accommodate deeper pipelining and retiming to increase the clock frequency.

[5]We use NK to refer to $N \cdot 1024$ `y` elements.

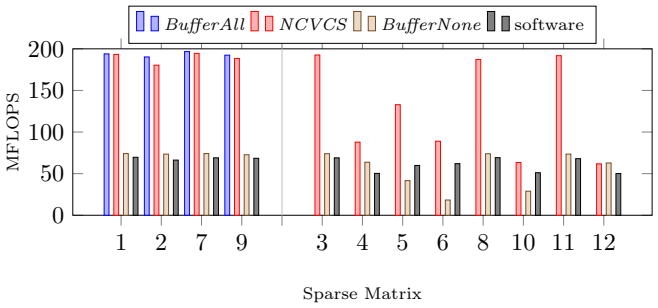


Figure 10: Comparison of best-case SpMV performance.

6.2. Best-Case Performance

We now present a comparison of maximum SpMV accelerator performance with different random access methods. For each access method, we empirically determined the following parameters to maximize performance:

- *NCVCS*: maximum sized cache (32K elements), cold miss skip enabled, nonblocking with 16-element issue window and miss queue.
- *BufferNone*: 32-element issue window.
- *BufferAll*: maximum sized OCM (64K elements), 8-element issue window.
- *software*: software SpMV on the CPU, `-O2` flag.

Figure 10 summarizes the SpMV performance from these best-case configurations. Firstly, we note that *BufferAll* cannot be used for 8 of the 12 matrices in the test suite as there is insufficient OCM on the FPGA for these matrices. For the remaining 4 matrices that do fit into OCM, *BufferAll* achieves near-peak (96%) computational efficiency, averaging at 193 MFLOPS. *BufferNone* exhibits performance similar to software SpMV, with an average performance of 61 MFLOPS that corresponds to 30% computational efficiency. Significant deviations from *BufferNone* average performance are visible for matrices 5, 6 and 10. Finally, *NCVCS* outperforms or closely matches the performance of the other methods, with 147 MFLOPS and 73% computational efficiency on average. Sections 6.3 and 6.4 provide a deeper analysis of performance and parameters for *BufferNone* and *NCVCS*.

6.3. Analysis of BufferNone

As we note in Section 3.2, *BufferNone* needs a large issue window to tolerate DRAM's high access latency and enable high performance, but at the risk of increasing RAW hazard stalls. Figure 11 plots the performance of *BufferNone* with increasing issue window size. We can classify the matrices into three groups according to their

Table 2: Suite with `maxColSpan` and `maxAlive` values for each sparse matrix.

#	Name	Rows (Cols)	Nonzeroes	NZ/col (L)	Problem Type	`maxColSpan`	`maxAlive`
1	cant	62451	4007383	64.17	FEM cantilever	549	549
2	conf5_4-8x8-05	49152	1916928	39	quantum chromodynamics	48392	13836
3	consph	83334	6010480	72.13	FEM concentric spheres	46481	9074
4	cop20k_A	121192	2624331	21.65	accelerator cavity design	121052	99843
5	mac_econ_fwd500	206500	1273389	6.17	macroeconomic model	2481	431
6	mc2depi	525825	2100225	3.99	model of epidemic	770	770
7	pdb1HYS	36417	4344765	119.31	protein database	34475	8499
8	pwtk	217918	11634424	53.39	wind tunnel stiffness matrix	189337	16070
9	rma10	46835	2374001	50.69	3D CFD model	25380	19269
10	scircuit	170998	958936	5.61	circuit simulation	170975	80408
11	shipsec1	140874	7813404	55.46	ship section detail	10145	9797
12	webbase-1M	1000005	3105536	3.10	web connectivity matrix	997552	283024

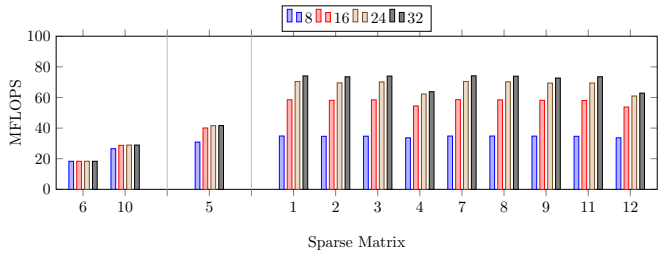


Figure 11: *BufferNone* performance with issue window size.

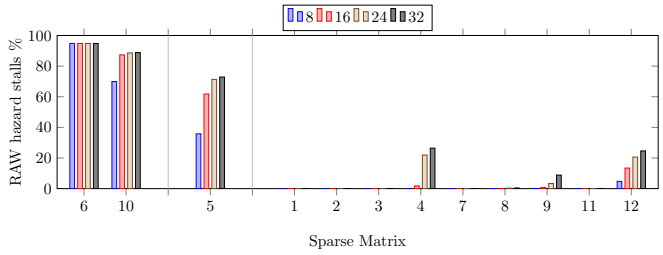


Figure 12: Percentage of RAW hazard stalls with issue window size.

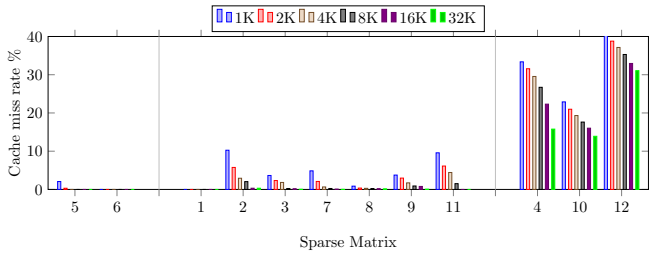


Figure 13: *NCVCS* cache miss rate with cache sizes.

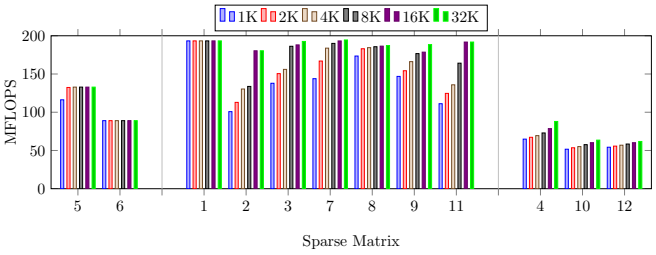


Figure 14: SpMV performance with different cache sizes.

behavior. The first group includes matrices 6 and 10, which get no performance benefit from more memory requests. The second group includes matrix 5, with limited performance growth going from issue window 8 to 16, and leveling off afterwards. The third and final group includes all other matrices, which experience significant performance growth with increasing window size.

To understand these different behaviours, we plot the percentage of RAW hazard stalls within all frontend stalls in Figure 12. A close correspondence between hazard stall growth and lack of performance increase from larger issue window can be observed. Namely, the first group has a high percentage (above 70%) of RAW stalls for even the smallest issue window, while the second group starts out with a moderate percentage of hazard stalls that rapidly increases with a larger issue window. The third group does not experience significant hazard stalls even with large issue window sizes. They experience performance growth up to a window size of 32, after which the memory port becomes saturated and does not deliver more bandwidth for result vector accesses. The expected bandwidth-bound peak performance for *BufferNone* is between 140 to 160 MFLOPS for the matrix suite, whereas the observed RAW hazard-free peak performance is around 70 MFLOPS. This suggests that the DRAM ports are unable to deliver full bandwidth. Permitting out-of-order DRAM operation and prioritizing random vector accesses in shared memory system resources can help increase efficiency, which is left for future work.

6.4. *Analysis of* NCVCS

We now present results and discussion on different aspects of *NCVCS*, including the performance impact of cache size and miss queue size. We also evaluate the cost of preprocessing and show how it compares with the obtained performance and OCM savings. We use a miss queue size of 16 and cold miss skip enabled as baseline parameters for these experiments.

6.4.1. *Impact of Cache Size*

A key indicator of cache performance is cache miss rate (i.e. the ratio of accesses that do not hit in the cache),

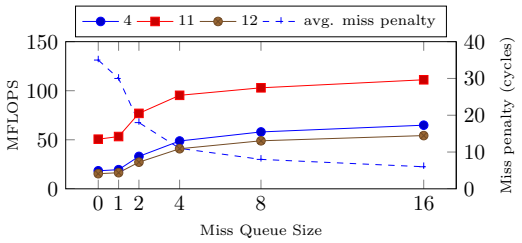


Figure 15: Miss queue size against performance and miss penalty.

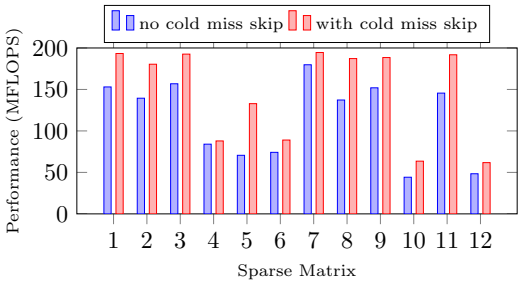


Figure 16: Performance impact of cold miss skip with a 32K-cache.

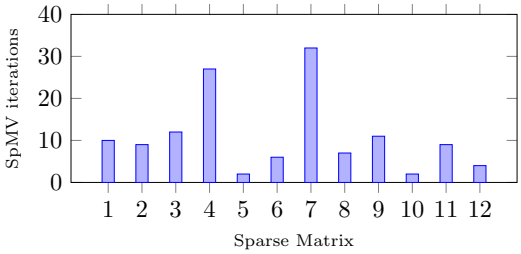


Figure 17: Iterations to break-even for cold miss skip preprocessing.

which is plotted in Figure 13. 9 of the 12 matrices in the suite have `maxAlive` values smaller than the largest cache (32K elements) we can deploy on this FPGA, and together with cold miss skip achieve zero or near-zero cache miss rate. This also implies that conflict misses are not a problem for direct-mapped caches with at least `maxAlive` capacity, confirming the results from [11]. The remaining matrices 4, 10 and 12 require cache capacity greater than what we can deploy on this FPGA, and exhibit higher miss rates.

Although cache hit rate is critical to overall accelerator performance, it is not the sole determinant. Figure 14 illustrates the SpMV accelerator performance across the matrix suite with different cache sizes, which reveals three groups of behavior. Perhaps most striking are the results for matrices 5 and 6, which have no cache misses but achieve only 50% to 70% of peak performance. This is due to a large number of RAW hazard stalls, which limits the rate at which instructions can be sent to the reducer and decreases efficiency. Statically or dynamically reordering the matrix can help increase performance in these cases, although we do not investigate their benefit in this work. The three matrices with high cache miss rates (4, 10 and 12) display lower performance averaging around 70 MFLOPS. Although matrices 4 and 10 have similar `maxAlive` values and cache miss rates, matrix 10 benefits less from larger caches due to RAW hazards. Finally, the remaining seven matrices have close to zero in both miss rates and RAW hazard penalties, and achieve on average 95% computational efficiency with a `maxAlive`-element cache or larger.

6.4.2. Impact of Nonblocking Cache

To illustrate how nonblocking helps improve performance with small caches and high miss rates, we plot the performance of matrices 4, 11 and 12 with a 1K-sized vector cache with varying miss queue sizes in Figure 15. All of these matrices have `maxAlive` larger than 1K and high cache miss rates. The performance with a miss queue size of 0 (which is a blocking cache) is 18, 51 and 15 MFLOPS for these three matrices, respectively. With a miss queue size of 16, the performance levels rise to 65, 11 and 54 MFLOPS, which correspond to over tripled performance for the larger matrices 4 and 12, and over doubled performance for matrix 11. The increase in performance is directly linked to how the miss penalty decreases with larger miss queue size, which can be examined by dividing the number of frontend stalls (excluding RAW hazards) by the number of cache misses. The average number of stall cycles, which is plotted as a dashed line in Figure 15, decreases from 35 for a blocking cache to 6 for a miss queue size of 16.

6.4.3. Impact and Cost of Cold Miss Skip

Figure 16 compares the performance of a 32K-element nonblocking cache with and without the cold miss skip optimization across the matrix suite. On average, cold miss skip improves performance by 30%, although the exact benefit varies between 5% to 88%. Due to interactions[6] between matrix structure and how cold miss skip influences latency, the performance benefits cannot be trivially linked to matrix dimensions and sparsity in a nonblocking cache. For instance, matrices 5 and 6 have both zero non-cold cache misses, few elements per row and experience performance degradation due to RAW hazards. However, the performance benefit from cold miss skip is quite different: while matrix 6 performance improves by 20%, matrix 5 almost doubles its performance with 88% improvement.

It is important to remember that the cold miss skip requires marking the start of each row via preprocessing, which has a certain cost. As discussed in Section 2.3, the cost of preprocessing can be outweighed by the resulting speedups across multiple SpMV iterations. To characterize the cost of preprocessing for cold miss skip, we use the break-even iterations as an indicator. We define S as the

[6]For instance, in a cache without cold miss skip, a nonblocking cold miss followed by hits can mimic the benefits of cold miss skip.

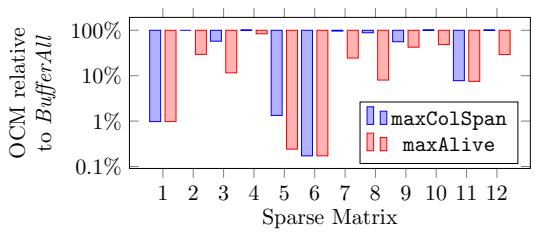


Figure 18: OCM requirements compared to *BufferAll*.

SpMV iteration count when cold miss skip peformance improvement matches or exceeds its preprocessing cost, i.e.:

$$S = \left\lceil \frac{T_{CPU-preproc}}{T_{FPGA-spmv-no-cms} - T_{FPGA-spmv-cms}} \right\rceil$$

Figure 17 depicts the number of break-even iterations for a cache with 32K elements, which ranges between 2 to 32 iterations (average 11) for the matrix suite. This compares favorably to the 75 break-even iterations estimated by Toledo [18] for matrix reordering, and can be improved further by using a more powerful CPU or hardware for the preprocessing.

6.4.4. Row Lifetime Analysis and OCM Savings

In Section 4.2.2 we described how the minimum cache size to avoid all capacity misses could be calculated for a given sparse matrix, either using `maxColSpan` or `maxAlive`. The rightmost columns of Table 2 list these values for each matrix. However, a vector cache also requires tag and valid bit storage in addition to the cache data storage, which must be taken into consideration. To calculate the OCM savings with vector caching, we use the *BufferAll* OCM requirements as a baseline, which is $64 \cdot m$ bits (one double-precision floating point value per `y` element). For a matrix with m rows, the minimum storage requirement in terms of number of bits for a vector cache of W elements, including tag and valid bits, is given by the following equation:

$$(64 + \lceil log_2 m \rceil - \lceil log_2 W \rceil + 1) \cdot W$$

In Figure 18 we plot the required OCM of `maxColSpan`- and `maxAlive`-sized caches relative to *BufferAll*. On average, a vector cache offers OCM savings of 76% with `maxAlive` elements and 41% with `maxColSpan` elements. This shows that substantial OCM savings can be achieved by dimensioning the cache based on row lifetime analysis. Matrices 1, 5, 6 and 11, which have most of their elements clustered around the diagonal, already gain significant storage savings with `maxColSpan`. The other matrices require `maxAlive` to shrink the required OCM by at least half, with the exception of matrix 4. For this matrix, even `maxAlive` is only 17.6% smaller than the number of rows, and the OCM benefit from vector caching is limited.

Since preprocessing is required to dimension the vector cache, it is useful to characterize the cost of this preprocessing. If we denote the time cost of a single software SpMV iteration cost as T, our results indicate that the preprocessing costs are on average $0.13T$ for `maxColSpan` and $2.6T$ for `maxAlive`. In comparison, Toledo [18] estimates the cost of Cuthill-McKee matrix reordering to be $1T$ to $3T$ and nested dissection reordering to be $20T$ to $200T$. Furthermore, the preprocessing times will improve by using a better CPU with a more powerful memory system or parallel preprocessing, making it worthwhile to perform row lifetime analysis if a smaller OCM footprint is desired.

7. Conclusion and Future Work

In this paper, we considered the problem of handling random accesses to large volumes of data to enable efficient SpMV implementations. We have proposed a processor-like view of the SpMV reducer, and framed three approaches with the random accesses handled strictly on-chip, strictly off-chip and balanced across off- and on-chip to understand how each approach can achieve high efficiency. To combine the strengths off- and on-chip random accesses, we have proposed *NCVCS*, which combines software preprocessing with a customized nonblocking cache for servicing random accesses.

To compare the *BufferAll*, *NCVCS* and *BufferNone* approaches, we deployed them as part of a FPGA SpMV accelerator, and evaluate their performance and efficiency with a suite of large sparse matrices. Our results indicate that the three methods have similar FPGA logic resource utilization, but significant differences in the required FPGA OCM (BRAMs) and performance. We confirm that *BufferAll* enables highly efficient (96% of peak) accelerators, but the size of the largest sparse matrix it can handle is limited by FPGA OCM. *BufferNone*, which services all random accesses from DRAM, performed the worst among the three schemes with 30% average efficiency and significant RAW hazard penalties. Finally, *NCVCS* either outperforms or performs almost as well as the other schemes for all matrices, with average 73% efficiency. A cache miss rate close to 0% is achieved on matrices with at least `maxAlive`-sized caches. This indicates that `maxAlive` is a good indicator of minimum required capacity and can provide significant OCM savings. Our results also show that the preprocessing necessary for cold miss skip and row lifetime analysis has reasonably low cost, and is well worth the benefits.

As future work, we plan to investigate how well these random access schemes work with parallelization. Our results indicate that the remaining inefficiencies in SpMV acceleration are mostly concentrated around RAW hazard stalls and low bandwidth for DRAM random accesses, which could also be investigated to further improve efficiency.

References

[1] D. A. Reed, J. Dongarra, Exascale Computing and Big Data, Communications of the ACM 58 (7).

[2] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of sparse matrix–vector multiplication on emerging multicore platforms, Parallel Computing 35 (3).

[3] J. Kepner, J. Gilbert, Graph algorithms in the language of linear algebra, Vol. 22, SIAM, 2011.

[4] Y. Umuroglu, D. Morrison, M. Jahre, Hybrid Breadth-First Search on a Single-Chip FPGA-CPU Heterogeneous Platform, in: Field Programmable Logic and Applications (FPL), 2015 25th International Conference on, 2015.

[5] D. Buono, J. A. Gunnels, X. Que, F. Checconi, F. Petrini, T.-C. Tuan, C. Long, Optimizing Sparse Linear Algebra for Large-Scale Graph Analytics, Computer 48 (8).

[6] D. Gregg, C. Mc Sweeney, C. McElroy, F. Connor, S. McGettrick, D. Moloney, D. Geraghty, FPGA based sparse matrix vector multiplication using commodity DRAM memory, in: Field Prog. Logic and Applications, Int. Conf. on, 2007.

[7] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, G. Stitt, A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication, in: Field-Programmable Custom Computing Machines, IEEE Int. Symp. on, 2014.

[8] C. Gary C.T., P. Grigoras, P. Burovskiy, W. Luk, An efficient sparse conjugate gradient solver using a benes permutation network, in: Field Prog. Logic and Applications, Int. Conf. on, 2014.

[9] R. Dorrance, F. Ren, D. Marković, A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-BLAS on FPGAs, in: Proc. of the ACM/SIGDA Int. Symp. on FPGAs, 2014.

[10] R. J. Halstead, W. Najjar, Compiled Multithreaded Data Paths on FPGAs for Dynamic Workloads, in: Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, 2013.
URL `http://dl.acm.org/citation.cfm?id=2555729.2555732`

[11] Y. Umuroglu, M. Jahre, A Vector Caching Scheme for Streaming FPGA SpMV Accelerators, in: Applied Reconfigurable Computing, Vol. 9040, 2015.

[12] Y. Umuroglu, M. Jahre, An Energy Efficient Column-Major Backend for FPGA SpMV Accelerators, in: Computer Design, IEEE Int. Conf. on, 2014.

[13] E. Cuthill, Several strategies for reducing the bandwidth of matrices, in: Sparse Matrices and Their Applications, Springer, 1972, pp. 157–166.

[14] L. Oliker, X. Li, P. Husbands, R. Biswas, Effects of ordering strategies and programming paradigms on sparse matrix computations, Siam Review 44 (3) (2002) 373–393.

[15] K. Kourtis, V. Karakasis, G. Goumas, N. Koziris, CSX: an extended compression format for SpMV on shared memory systems, in: ACM SIGPLAN Notices, 2011.

[16] J. Willcock, A. Lumsdaine, Accelerating sparse matrix computations via data compression, in: Proceedings of the 20th annual international conference on Supercomputing, ACM, 2006, pp. 307–316.

[17] J. C. Pichel, F. F. Rivera, M. Fernández, A. Rodríguez, Optimization of sparse matrix–vector multiplication using reordering techniques on GPUs, Microprocessors and Microsystems 36 (2) (2012) 65–77.

[18] S. Toledo, Improving the memory-system performance of sparse-matrix vector multiplication, IBM Journal of Res. and Dev. 41 (6).

[19] Y. Zhang, Y. Shalabi, R. Jain, K. Nagar, J. Bakos, FPGA vs. GPU for sparse matrix vector multiply, in: Field-Programmable Technology, International Conference on, 2009.

[20] S. Jain-Mendon, R. Sass, A hardware–software co-design approach for implementing sparse matrix vector multiplication on fpgas, Microprocessors and Microsystems 38 (8) (2014) 873–888.

[21] K. Townsend, J. Zambreno, Reduce, reuse, recycle (r 3): A design methodology for sparse matrix vector multiplication on reconfigurable platforms, in: Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on, IEEE, 2013, pp. 185–191.

[22] K. K. Nagar, J. D. Bakos, A Sparse Matrix Personality for the Convey HC-1, in: Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on, IEEE, 2011, pp. 1–8.

[23] T. A. Davis, Y. Hu, The University of Florida Sparse Matrix Collection, ACM Trans. Math. Softw. 38 (1). doi:10.1145/2049662.2049663.
URL `http://doi.acm.org/10.1145/2049662.2049663`

[24] N. P. Jouppi, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, in: Computer Architecture, Proc. of the Int. Symp. on, 1990.

[25] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, K. Asanović, Chisel: constructing hardware in a Scala embedded language, in: Proc. of the Design Automation Conference, 2012.