A Vector Caching Scheme for Streaming FPGA SpMV Accelerators

Yaman Umuroglu(✉) and Magnus Jahre

Department of Computer and Information Science,
Norwegian University of Science and Technology, Trondheim, Norway
{yamanu,jahre}@idi.ntnu.no

Abstract. The sparse matrix – vector multiplication (SpMV) kernel is important for many scientific computing applications. Implementing SpMV in a way that best utilizes hardware resources is challenging due to input-dependent memory access patterns. FPGA-based accelerators that buffer the entire irregular-access part in on-chip memory enable highly efficient SpMV implementations, but are limited to smaller matrices due to on-chip memory limits. Conversely, conventional caches can work with large matrices, but cache misses can cause many stalls that decrease efficiency. In this paper, we explore the intersection between these approaches and attempt to combine the strengths of each. We propose a hardware-software caching scheme that exploits preprocessing to enable performant and area-effective SpMV acceleration. Our experiments with a set of large sparse matrices indicate that our scheme can achieve nearly stall-free execution with average 1.1 % stall time, with 70 % less on-chip memory compared to buffering the entire vector. The preprocessing step enables our scheme to offer up to 40 % higher performance compared to a conventional cache of same size by eliminating cold miss penalties.

1 Introduction

Increased energy efficiency is a key goal for building next-generation computing systems that can scale the "utilization wall" of dark silicon [1]. A strategy for achieving this is accelerating commonly encountered kernels in applications. Sparse Matrix – Vector Multiplication (SpMV) is a computational kernel widely encountered in the scientific computation domain and frequently constitutes a bottleneck for such applications [2]. Analysis of web connectivity graphs [3] can require adjacency matrices that are very large and sparse, with a tendency to grow even bigger due to the important role they play in the Big Data trend.

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 memory reads to addresses that correspond to non-zero element locations of the matrix, which are only known at runtime. The kernel is otherwise characterized by little data reuse and large per-iteration data requirements [2], which makes the performance memory-bound. Storing the kernel inputs and outputs in

© Springer International Publishing Switzerland 2015
K. Sano et al. (Eds.): ARC 2015, LNCS 9040, pp. 15–26, 2015.
DOI: 10.1007/978-3-319-16214-0_2

high-capacity high-bandwidth DRAM is considered a cost-effective solution [4]; however, the burst-optimized architecture of DRAM constitutes 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 execution (termed *streaming acceleration* in this paper). From the perspective of the SpMV kernel, the ability to deliver high external memory bandwidth owing to high pin count and dynamic (run-time) specialization via partial 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 [5,6], some of which present order-of-magnitude better energy efficiency and comparable performance to CPU and GPGPU solutions thanks to streaming acceleration. These accelerators tackle the irregular access problem by buffering the entire random-access data in *on-chip memory (OCM)*. Unfortunately, this *buffer-all strategy* is limited to SpMV operations where the random-access data can fit in OCM, and therefore not suitable for very large sparse matrices.

To address this problem, we propose a specialized vector caching scheme for area-efficient SpMV accelerators that can target large matrices while still preserving the streaming acceleration property. Using the canonical cold-capacity-conflict cache miss classification, we examine how the structure of a sparse matrix relates to each category and how misses can be avoided. By exploiting preprocessing (which is quite common in GPGPU and CPU SpMV optimizations) to specialize for the sparsity pattern of the matrix we show that streaming acceleration can be achieved with significantly smaller area for a set of test matrices. Our experiments with a set of large sparse matrices indicate that our scheme achieves the best of both worlds by increasing performance by 40% compared to a conventional cache while at the same time using 70% less OCM than the buffer-all strategy. The contributions of this work are four-fold. First, we describe how the structure of a sparse matrix relates to *cold*, *capacity* and *conflict* misses in a hardware cache. We show how cold misses to the result vector can be avoided by marking row start elements in column-major traversal. We propose two methods of differing accuracy and overhead for estimating the required cache depth to avoid all capacity misses. Finally, we present an enhanced cache with cold miss skip capability, and demonstrate that it can outperform a traditional cache in performance and a buffer-all strategy in area.

2 Background and Related Work

2.1 The SpMV Kernel and Sparse Matrix Storage

The SpMV kernel $\boldsymbol{y} = \mathbf{A} \cdot \boldsymbol{x}$ consists of multiplying an $m \times n$ sparse matrix $\mathbf{A}$ with NZ nonzero elements by a dense vector $\boldsymbol{x}$ of size n to obtain a result vector $\boldsymbol{y}$ 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

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

```
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]
```

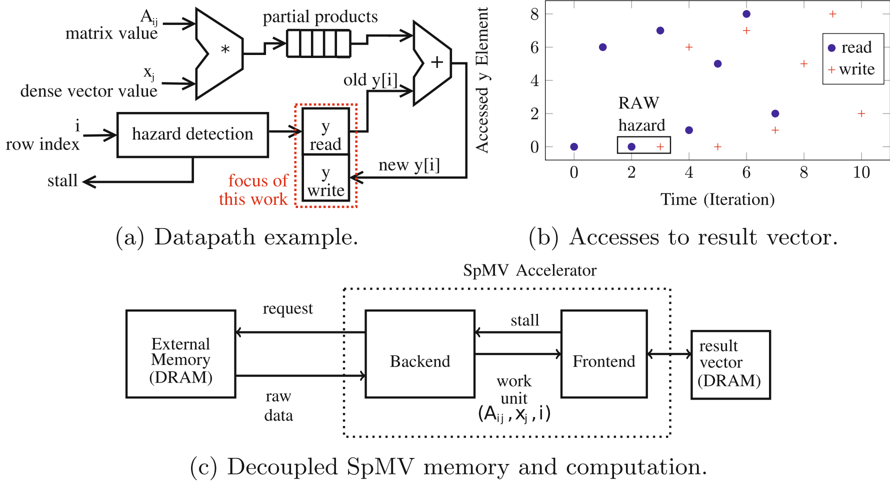
Fig. 1. A sparse matrix, its CSC representation and SpMV pseudocode. The random-access clause to `y` is highlighted.

sparse matrices have been proposed, some of which specialize on particular sparsity patterns, and others suitable for generic sparse matrices. In this paper, we will assume an FPGA SpMV accelerator that uses column-major sparse matrix traversal (in line with [4,6,7]) and an appropriate storage format such as Compressed Sparse Column (CSC). Column-major is preferred over row-major due to the advantages of maximum temporal locality on the dense vector access and the natural C-slow-like interleaving of rows in floating point multiplier pipelines, enabling simpler datapaths [6]. Additionally, as we will show in Section 3.2 it allows bypassing cold misses, which can contribute significantly to performance. Figure 1 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 such as `values` and `colptr`. 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.

2.2 FPGA SpMV Accelerators and Result Vector Access

The datapath of a column-major SpMV accelerator is a multiply-accumulator with feedback from a random-access memory, as illustrated in Figure 2a. New partial products are summed into the corresponding element of the result vector, which can give rise to read-after-write (RAW) hazards due to the latency of the adder, as shown in Figure 2b. Addressing this requires a read operation to `y[i]` to be delayed until the writes to `y[i]` are completed, which is typically avoided by stalling the pipeline or reordering the elements.

With growing sparse matrix sizes and typically double-precision floating point arithmetic, the inputs of the SpMV kernel can be very large. Combined with the memory-bound nature of the kernel, this requires high-capacity high-bandwidth external memory to enable competitive SpMV implementations. Existing FPGA SpMV accelerators [4–6] used DRAM as a cost-effective option for the storing the SpMV inputs and outputs, which is also our approach in this work. These designs typically address the random access problem by buffering the entire random-access vector in OCM [5,6]. Random accesses to the vector are thus guaranteed to be serviced with a small, constant latency. Unfortunately, this limits the maximum sparse matrix size that can be processed with the accelerator. To deal with `y` vectors larger than the OCM size while avoiding DRAM random access latencies, Gregg et al. [4] proposed to store the result vector in high-capacity DRAM and used a small direct-mapped cache. They also observed that cache misses present a significant penalty, and proposed reordering the matrix and processing in cache-sized chunks to reduce miss rate. However, this



(a) Datapath example.

(b) Accesses to result vector.

(c) Decoupled SpMV memory and computation.

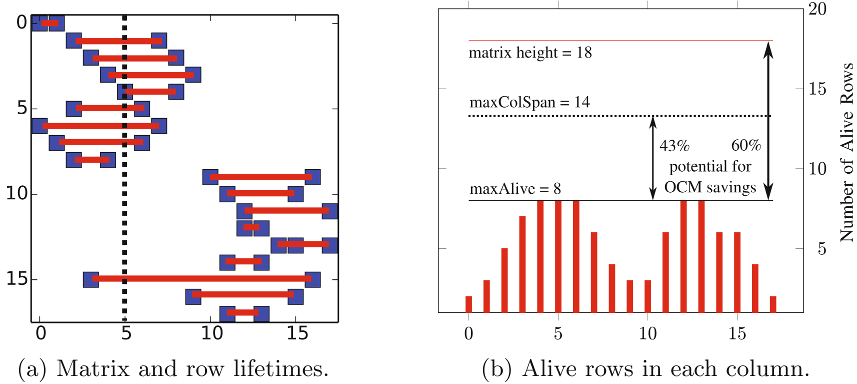
Fig. 2. A column-major FPGA SpMV accelerator design

imposes significant overheads for large matrices. In contrast, our approach does not modify the matrix structure; rather, it extracts information from the sparse matrix to reduce cache misses, which can be combined with reordering for greater effect. Prior work such as [8] analyzed SpMV cache behavior on microprocessors, but includes non-reusable data such as matrix values and requires probabilistic models. FPGA accelerators can exhibit deterministic access patterns for each sparse matrix, which our scheme exploits for analysis and preprocessing.

To concentrate on the random access problem, we base our work on a decoupled SpMV accelerator architecture [7], which defines a *backend* interfacing the main memory and pushing *work units* to the *frontend*, which handles the computation. Our focus will be on the random-access part of the frontend. Since we would like the accelerator to support larger result vectors that do not fit in OCM, we add DRAM for storing the result vector, as illustrated in Figure 2c.

2.3 Sparse Matrix Preprocessing

The memory behavior and performance of the SpMV kernel is dependent on the particular sparse matrix used, necessitating a preprocessing step at runtime for optimization. Fortunately, algorithms that make heavy use of SpMV tend to multiply the same sparse matrix with many different vectors, which enables ameliorating the cost of preprocesing across speed-ups in each SpMV iteration. This preprocessing can take many forms [9], including permuting rows/columns to create dense structure, decomposing into predetermined patterns, mapping to parallel processing elements to minimize communication and so on. 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 information to enable specialized cache operation instead of changing the matrix structure.



(a) Matrix and row lifetimes.

(b) Alive rows in each column.

Fig. 3. Example matrix Pajek/GD01_b and row lifetime analysis

3 Vector Caching Scheme

To tackle the memory latency problem while accessing the result vector from DRAM, we buffer a portion of the result vector in OCM and use a hardware-software cooperative *vector caching scheme* that enables per-matrix specialization. This scheme will consist of a runtime *preprocessing step*, which will extract the necessary information from the sparse matrix for efficient caching including the required cache size, and *vector cache* hardware which will use this information. Our goal is to shrink the OCM requirements for the vector cache while avoiding stalls for servicing requests from main memory.

3.1 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 our vector caching scheme for a given sparse matrix.

Calculating `maxAlive`: 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 3a 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 3a 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 3a – though the alive rows themselves may be different, no column has more than 8 alive rows in this example.

Calculating `maxColSpan`: 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 3a has a span of 14, which is `maxColSpan` for this matrix.

3.2 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.

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 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[1]. 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 3.3.

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 buffer-all 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

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

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 3.1 can be used instead. Figure 3b shows the row lifetime analysis for the matrix in Figure 3a and how different estimations of the required capacity yield different OCM savings compared to the buffer-all strategy.

Conflict Misses: For the case of an SpMV vector cache, 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. As described in Section 4.2, our experiments indicate that conflicts are few for most matrices even with a direct-mapped cache, as long as the cache capacity is sufficient. Techniques such as victim caching [10] can be utilized to decrease conflict misses in direct-mapped caches, though we do not investigate their benefit in this work.

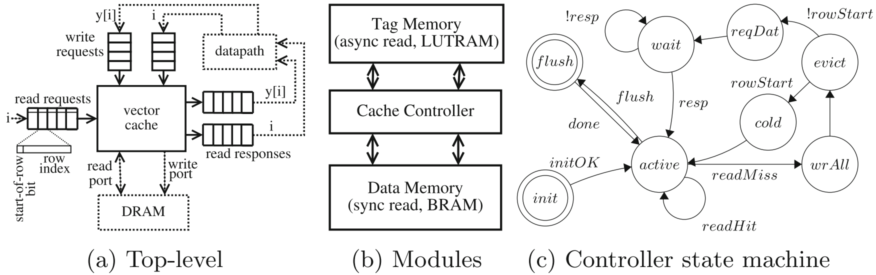
3.3 Preprocessing

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

Algorithm 1 Finding `maxAlive` and marking row starts.

```
function PreprocessMaxAlive(CSCMatrix A)
    Q ← priorityQueue(), currentAlive ← 0, maxAlive ← 0
    R ← toCSR(A)
    for i ← 0..m − 1 do
        start ← R.colind[R.rowptr[i]]; end ← R.colind[R.rowptr[i + 1] − 1]
        markRowStart(A, start, i)
        Q.insert(prio = start, elem = +1); Q.insert(prio = end, elem = −1)
    end for
    while !Q.empty() do
        currentAlive ← currentAlive + Q.pop(); maxAlive ← max(currentAlive, maxAlive)
    end while
    return maxAlive
end function
```

One task that the preprocessing needs to fulfill is to establish the required cache capacity for the sparse matrix via the methods described in Section 3.1. Another important function of the preprocessing is marking the start of each row to avoid cold misses. In this paper, we reserve the highest bit of the `rowind` field in the CSC representation to mark a nonzero element as 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,



(a) Top-level (b) Modules (c) Controller state machine

Fig. 4. Design of the vector cache

and can still represent matrices with over 2 billion rows for a 32-bit `rowind`. At the time of writing, this is 18x larger than the largest matrix in the University of Florida collection [3].

For the case of computing `maxAlive`, we can formulate the problem as constructing an interval tree and finding the largest number of overlapping intervals. Algorithm 1The values inserted are +1 and -1, respectively for row starts and ends. `maxAlive` is obtained by finding the maximum sum the sorted values during the iteration. We do not present the algorithm for finding `maxColSpan`, as it is simply iterating over each column of the sparse matrix and finding the one with the greatest span.

3.4 Vector Cache Design

The final component of our vector caching scheme is the vector cache hardware itself. Our design is a simple increment over a traditional direct-mapped hardware cache to allow utilizing the start-of-row bits to avoid cold misses. A top-level overview of the vector cache and how it connects to the rest of the system is provided in Figure 4a. All interfaces use ready/valid handshaking and connect to the rest of the system via FIFOs, which simplifies placing the cache into a separate clock domain if desired. Row indices with marked start-of-row bits are pushed into the cache as 32-bit-wide read requests. The cache returns the 64-bit read data, as well as the requested index itself, through the read response FIFOs. The datapath drains the read response FIFOs, sums the `y[i]` value with the latest partial product, and writes the updated `y[i]` value into the write request FIFOs of the cache.

Internally, the cache is composed of data/tag memories and a controller, depicted in Figure 4b. Direct-mapped associativity is chosen for a more suitable FPGA implementation as it avoids content-associative memories required for multi-way caches. To increase performance and minimize the RAW hazard window, the design offers single-cycle read/write hit latency, but read misses are blocking to respect the FIFO ordering of requests. To make efficient use of the synchronous on-chip SRAM resources in the FPGA while still allowing single-cycle hits, we chose to implement the data memory in BRAM while the tag

Table 1. Suite with `maxColSpan` and `maxAlive` values for each sparse matrix

#	Name	Dimension	Nonzeroes	NZ/col	Problem Type	`maxColSpan`	`maxAlive`
1	webbase-1M	1000005	3105536	3.10	web connectivity matrix	997552	283024
2	mc2depi	525825	2100225	3.99	model of epidemic	770	770
3	scircuit	170998	958936	5.61	circuit simulation	170975	80408
4	mac_econ_fwd500	206500	1273389	6.17	macroeconomic model	2481	431
5	cop20k_A	121192	2624331	21.65	accelerator cavity design	121052	99843
6	shipsec1	140874	3568176	25.33	ship section detail	10145	9797
7	pwtk	217918	11524432	52.88	wind tunnel stiffness matrix	189337	16070
8	consph	83334	6010480	72.13	FEM concentric spheres	46481	9074

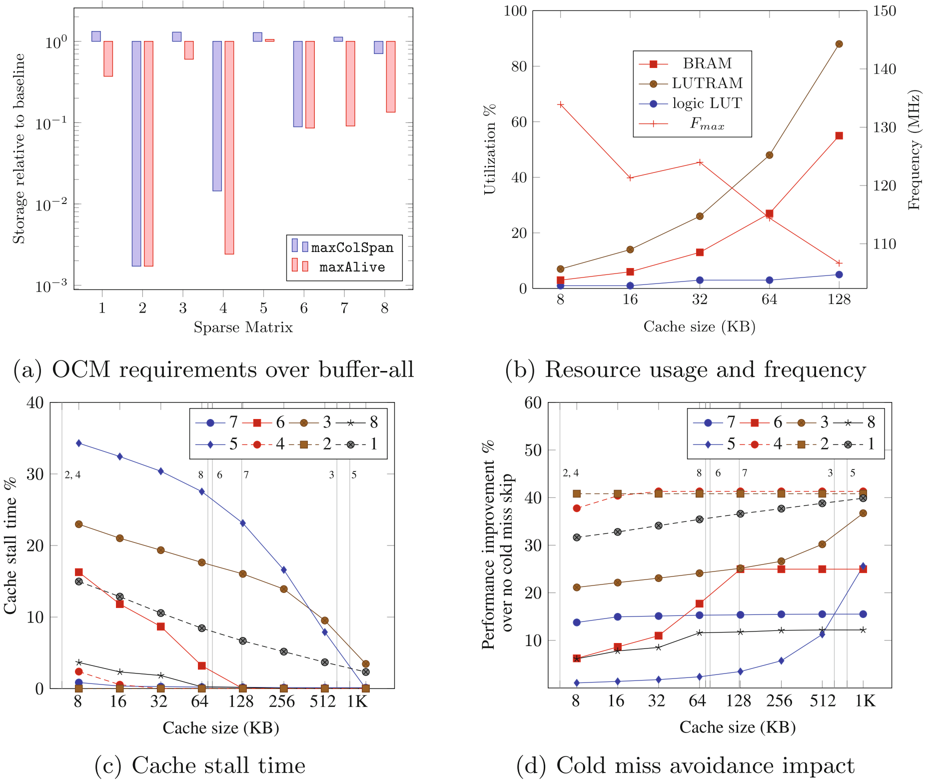
memory is implemented as look-up tables. The controller finite state machine is illustrated in Figure 4c. Write misses are directly transferred to the DRAM to keep the cache controller simple. Prior to servicing a read miss, the controller waits until there are no more writes from the datapath to guarantee memory consistency. Regular read misses cause the cache to issue a DRAM read request, which prevents the missing read request from proceeding until a response is received. Avoiding cold misses is achieved by issuing a zero response on a read miss with the start-of-row bit set, without issuing any DRAM read requests.

4 Experimental Evaluation

We present a two-part evaluation of our scheme: an analysis of OCM savings using the minimum required capacity estimation techniques, followed by performance and FPGA synthesis results of our our vector caching scheme. For both parts of the evaluation we use a subset of the sparse matrix suite initially used by Williams et al. [2], excluding the smaller matrices amenable to the buffer-all strategy. The properties of each matrix is listed in Table 1.

4.1 OCM Savings Analysis

In Section 3.2 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 1 list these values for each matrix. However, a vector cache also requires tag and valid bit storage in addition to the cache daha storage, which decreases the net OCM savings from our method. We compare the total OCM requirements of `maxColSpan`- and `maxAlive`-sized vector caches against the buffer-all strategy. The baseline is calculated as $64 \cdot m$ bits (one double-precision floating point value per `y` element), whereas the vector cache storage requires $(64 + \lceil log_2(W) \rceil + 1) \cdot \mathtt{W}$ bits to also account for the tag/valid bits storage overhead, where W is the cache size. Figure 5a quantifies the amount of on-chip memory required for the two methods, compared to the baseline. For seven of the eight tested matrices, significant storage savings can be achieved by using our scheme. A vector cache of size `maxAlive` requires 0.3x of the baseline storage on average, whereas sizing according to `maxColSpan` averaged at 0.7x of the baseline. It should be noted that matrices 2, 4 and 6, which have a more



(a) OCM requirements over buffer-all

(b) Resource usage and frequency

(c) Cache stall time

(d) Cold miss avoidance impact

Fig. 5. Results from vector caching scheme evaluation

regular structure with elements clustered around the diagonal, already gain significant storage benefits from the low-overhead `maxColSpan` estimation. On the other hand, the irregular matrices 1 and 3 have no storage reduction benefits by using a `maxColSpan`-sized cache, so `maxAlive` must be used. For matrix #5, even `maxAlive` is only 17.6% smaller than the entire `y`, and therefore the savings from vector caching is not large enough to offset the tag overhead.

4.2 Vector Cache Evaluation

We use Chisel [11] to create a parametrizable hardware description for the vector cache, which is converted to Verilog via the Verilog backend. The generated Verilog code is fed into XST for FPGA synthesis in order to obtain frequency and area results for the cache, and passed through the Verilator tool to generate a cycle-accurate SystemC model. The model is used in a in-house SpMV frontend simulator, which is stimulated with inputs corresponding to the chosen sparse matrix and models the behavior of the accumulator datapath and DRAM for performance assessment. We assume a 100 MHz clock for the frontend, with delays of 7 cycles for the accumulator datapath and 10 cycles for DRAM reads.

Area and Frequency: We report area and frequency results from synthesis for a Xilinx Spartan-6 LX45 FPGA with -2 speed grade, chosen to demonstrate the potential of the technique with mediocre OCM. Our results indicate that the cold skip enhancement is with very little extra hardware cost (less than 1% in logic LUTs for the largest tested design), hence we do not report separate results for a baseline cache without this enhancement. Figure 5b shows the percent utilization of BRAM, LUTRAM and logic resources for a range of vector cache sizes, and the maximum frequency F_{max} reported by the synthesis tool. A vector cache of 128 KB can fit on this relatively small FPGA, which is large enough to accommodate the `maxAlive`-sized working set of 5 of the 8 tested matrices. As can be expected, the utilization of BRAM and LUTRAM increases linearly with cache size, and the LUTRAM used for cache tags ultimately limits scaling to larger caches. Due to the simple design, the LUT utilization for implementing logic is rather small and occupies about 5% of the available resources for the largest design, which leaves plenty of room for implementing the more logic-intensive parts of the accelerator. The maximum attainable frequency is between 106 – 133 MHz for the tested designs, which is similar to the operating frequencies of previous SpMV accelerator designs. Further F_{max} improvements can be achieved by using a more powerful FPGA or design optimizations.

Cache Stall Time: As our goal is to enable a stall-free cache, we evaluate the impact of cache stalls with our scheme. Figure 5c depicts the percentage of total execution time the accelerator with up to 1 MB of cache is stalled due to cache misses. The `maxAlive` of each matrix is indicated with numbered lines in the background. For 6 of the 8 matrices, allocating at least a `maxAlive`-sized cache with cold miss avoidance capability is enough to remove almost all cache misses, also indicating there are very few conflict misses. #3 is an exception, which suffers from conflict misses even with a large cache due to its nonzero pattern. The web connectivity matrix 1 has a working set larger than the maximum tested cache size, although its miss rate is already quite low. Low miss rates with cache sizes smaller than `maxAlive` is also observed for matrices 8 and 7, indicating that more relaxed working set definitions could be used for further reduction in required storage. Overall, by allocating at least `maxAlive`-sized caches, the cache stall time for our scheme is only 1.1% averaged across the test suite.

Cold Miss Avoidance: To show the gains from the cold miss avoidance technique, we plot the overall performance improvement due to removal of cold miss stalls in Figure 5d. The baseline for each data point is a vector cache of equal size without cold miss avoidance capabilities. The average performance improvement for at least `maxAlive`-sized caches is 28.6%. As the cache grows larger, fewer capacity misses are encountered and cold misses make up a larger percentage of the total. This increases the benefit from cold miss avoidance, until there are no cache misses left and the benefit levels off. Since larger sparse matrices exhibit more cold misses due to large `y` size, the greatest benefit is observed for the large matrices 1, 2 and 4, with up to 40% improvement. For matrix 5, the number of capacity misses with small caches is very large and very little benefit is observed until a cache size of 16K elements.

5 Conclusion and Future Work

We have studied how matrix structure relates to cache misses, and proposed a scheme that uses preprocessing to enhance the operation of a traditional hardware cache for FPGA SpMV accelerators. Specifically, we have proposed two methods to estimate required cache depth to avoid all capacity misses, and a way of enhancing the matrix representation to avoid all cold misses. Our experiments with a suite of large sparse matrices indicate that the scheme can service random accesses to the result vector with no or few stalls, while avoiding cold miss penalties that hamper traditional hardware caches. Future work will include evaluating the vector caching scheme in a complete FPGA SpMV accelerator context.

References

1. Taylor, M.B.: Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse. In: Proc. of the Design Automation Conference (2012)
2. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. Parallel Computing **35**(3) (2009)
3. Davis, T.A., Hu, Y.: The University of Florida Sparse Matrix Collection. ACM Trans. Math. Softw. **38**(1) (2011)
4. Gregg, D., Mc Sweeney, C., McElroy, C., Connor, F., McGettrick, S., Moloney, D., Geraghty, D.: FPGA based sparse matrix vector multiplication using commodity DRAM memory. In: Int. Conf. on Field Prog. Logic and Applications (2007)
5. Fowers, J., Ovtcharov, K., Strauss, K., Chung, E.S., Stitt, G.: A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In: IEEE Int. Symp. on Field-Programmable Custom Computing Machines (2014)
6. Dorrance, R., Ren, F., Marković, D.: 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)
7. Umuroglu, Y., Jahre, M.: An energy efficient column-major backend for FPGA SpMV accelerators. In: IEEE Int. Conf. on Computer Design (2014)
8. Temam, O., Jalby, W.: Characterizing the behavior of sparse algorithms on caches. In: Proc. of the ACM/IEEE Conf. on Supercomputing (1992)
9. Toledo, S.: Improving the memory-system performance of sparse-matrix vector multiplication. IBM Journal of Res. and Dev. **41**(6) (1997)
10. Jouppi, N.P.: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In: Proc. of the Int. Symp. on Computer Architecture (1990)
11. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., Asanović, K.: Chisel: constructing hardware in a scala embedded language. In: Proc. of the Design Automation Conference (2012)