Stop Crying Over Your Cache Miss Rate: Handling Efficiently Thousands of Outstanding Misses in FPGAs

Mikhail Asiatici and Paolo Ienne
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH–1015 Lausanne, Switzerland

ABSTRACT

FPGAs rely on massive datapath parallelism to accelerate applications even with a low clock frequency. However, applications such as sparse linear algebra and graph analytics have their throughput limited by irregular accesses to external memory for which typical caches provides little benefit because of very frequent misses. Non-blocking caches are widely used on CPUs to reduce the negative impact of misses and thus increase performance of applications with low cache hit rate; however, they rely on associative lookup for handling multiple outstanding misses, which limits their scalability, especially on FPGAs. This results in frequent stalls whenever the application has a very low hit rate. In this paper, we show that by handling thousands of outstanding misses without stalling we can achieve a massive increase of memory-level parallelism, which can significantly speed up irregular memory-bound latency-insensitive applications. By storing miss information in cuckoo hash tables in block RAM instead of associative memory, we show how a non-blocking cache can be modified to support up to three orders of magnitude more misses. The resulting miss-optimized architecture provides new Pareto-optimal and even Pareto-dominant design points in the area-delay space for twelve large sparse matrix-vector multiplication benchmarks, providing up to 25% speedup with 24× area reduction or to 2× speedup with similar area compared to traditional hit-optimized architectures.

ACM Reference Format:
Mikhail Asiatici and Paolo Ienne. 2019. Stop Crying Over Your Cache Miss Rate:, Handling Efficiently Thousands of Outstanding Misses in FPGAs. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), February 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3289602.3293901

1 INTRODUCTION

FPGAs can accelerate compute-intensive applications by implementing massively parallel datapaths. FPGAs also provide spatially distributed SRAM memories with low latency and large aggregate bandwidth, which can store small datasets or implement custom memory hierarchies backing the external DRAM. However, important application classes such as sparse linear algebra and graph analytics are embarrassingly parallel and yet their memory access pattern is such that, with caches that can be realistically implemented on FPGA, most of the accesses are cache misses. In this paper, we will radically rethink the balancing between cache and miss handling logic to optimize the throughput of *read* operations when a large fraction of cache misses is inevitable. We introduce a generic approach, orthogonal to application-specific optimizations, that can provide significant speedup with little design effort. Thanks to its generality, it might be particularly valuable for solutions generated by high-level synthesis tools.

FPGA '19, February 24–26, 2019, Seaside, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6137-8/19/02…$15.00
https://doi.org/10.1145/3289602.3293901

1.1 The Curse of Sparse Narrow Data

Custom memory hierarchy design and automatic generation usually rely on access patterns that have temporal and spatial locality (caches), are regular (scratchpads) or are at least known at compile-time (memory banking and address scrambling) [1, 4, 12, 30]. When access patterns have poor locality and are irregular and data-dependent, one can at best maximize memory-level parallelism (MLP) by emitting enough outstanding memory operations to fully exploit the DRAM latency; however, the throughput of the memory system is still limited to one operation per cycle per DRAM channel, at most. This imposes severe limitations on the amount of datapath parallelism that is worth implementing, limiting the advantage of using an FPGA.

The effective bandwidth gets even more limited if the operands are narrow, such as in sparse linear algebra and graph applications, which often involve irregular accesses to 32- or 64-bit scalars. This relates to the architecture of DRAM memory controllers on FPGA, which expose the memory through wide data interfaces in order to give access to the full DRAM bandwidth despite the slow FPGA clock. For example, exploiting the full 12.8 GB/s DDR3 bandwidth at 200 MHz requires transferring 512 bits per cycle. In this case, an accelerator that operates on 64-bit inputs could at best exploit one eighth of the peak DDR3 bandwidth, and most of the 512 bits returned by the DRAM controller would be discarded. Multiple accelerators whose requests are mutually uncorrelated can only be served by time-multiplexing the memory channel, canceling out any benefits due to parallelization. The only way to improve bandwidth utilization, and thus performance, would be to use larger portions of each block returned from memory.

1.2 Misses Are a Fact of Life

Figure 1 shows the histogram of the number of reuses of 512-bit blocks for an application with poor locality—accesses to the dense vector of 32-bit integers during sparse matrix-vector multiplication (SpMV) with the CSR-encoded pds-80 matrix from SuiteSparse [8]—and if the same number of read operations were performed

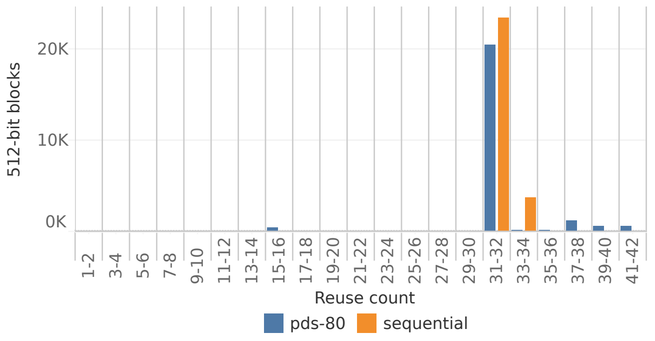

Figure 1: Spatial locality. The histogram shows the reuse count for each 512-bit block of data, for SpMV of pds-80 and for the same number of read operations scanning sequentially the same memory space. Despite showing very different cache hit rates, both memory traces have similar amounts of data reuse across the entire application execution.

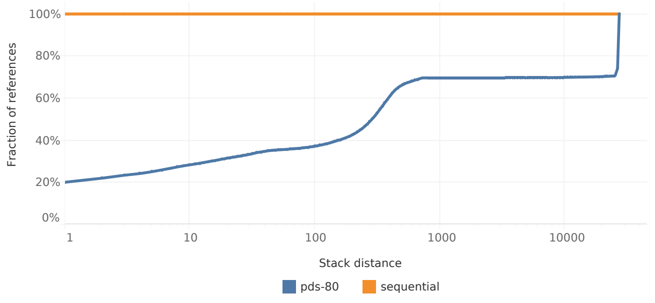

Figure 2: Temporal locality. The graph shows the fraction of 512-bit block references that have stack distance $\leq x$, for SpMV of pds-80 and a sequential memory trace. A large fraction of the reuses that occur in pds-80 are interleaved with references to many different blocks. Blocks can be stored in a cache hoping for future reuse; however, because large stack distances are common, cache lines are likely to be evicted before the next reuse, unless a large cache is used.

sequentially over the same address span. Both access patterns offer very similar opportunities for reuse; however, while the sequential access pattern achieves a $\frac{15}{16}$ = 94% hit rate on any cache with 512-bit cache lines, the hit rate of SpMV on a 128 kB direct-mapped blocking cache is only 57%. In fact, in a cache, eviction limits the time window where data reuse could occur. For the same two applications, Figure 2 shows the cumulative frequency of *stack distances*, i.e., the number of different 512-bit blocks that have been referenced between two consecutive references to the same blocks [6]. For example, for the memory trace: {746, 1947, 293, 5130, 293, 746}, the stack distances for the last access to blocks 293 and 746 are 1 and 3 respectively. While the stack distance of the sequential pattern is always zero, the SpMV cumulative histogram grows very slowly, meaning that a large fraction of reuses have large stack distance. With an ideal fully associative cache with N lines and LRU replacement, reaccessing a cache line with stack distances larger than N will always be a miss; on a realistic cache, even reuses with stack distance lower than N could be misses.

1.3 More MSHRs or More Cache?

The previous example showed that even applications with poor temporal locality may still have some spatial locality, which caches struggle to harness due to large stack distances between reuses. Even worst, a blocking cache actually hampers performance if the hit rate is too low to compensate for the stall cycles due to the misses. Non-blocking caches reduce stall penalties by handling one or more misses without stalling. On the first miss of a cache line (a *primary* miss), the address of the cache line is sent to memory and stored in a *miss status holding register* (MSHR); the offset of the requested word within the cache line, together with the request source/tag, is stored in a *subentry* for that MSHR. Subsequent misses to the same cache line (*secondary* misses), only require the allocation of a subentry on the same MSHR with no additional memory requests. When the missing cache line is received, it is both stored in the cache and used to serve all of its pending misses [10]. In practice, the time window where a cache line could be reused now includes the time between the first miss and the arrival of the data. For the purpose of widening the reuse window, adding an MSHR with its subentries is equivalent to adding one cache line to a fully associative cache; however, storing the miss metadata may require less bits than storing the entire cache line if the number of reuses is small. Moreover, each request to memory serves multiple requests from the accelerators, effectively increasing bandwidth utilization and pushing maximum MLP beyond the DRAM latency as long as there are available MSHRs and subentries [23]. Furthermore, unlike caches, MSHRs can serve primary misses without stalling the entire system, which is crucial for throughput maximization.

1.4 Exploring MSHR-Rich Caches

Non-blocking caches are extensively used in processors; however, to minimize latency, MSHRs are usually searched associatively, which limits their number to a few tens. In practice, on realistic CPUs, there is often little benefit in dramatically increasing the number of MSHRs beyond this limit [21, 25]. On FPGAs, associative searches are even less scalable than in ASICs; yet, massively parallel, high throughput FPGA accelerators that emit a vast number of outstanding reads to hide memory latency [7, 22] could potentially benefit from an MSHR-rich architecture even more than a general-purpose processor.

In this paper, we propose a novel miss handling architecture (MHA) optimized for bandwidth-bound FPGA accelerators that perform irregular accesses to external memory. By using the abundant on-chip RAM, we show how we can efficiently implement and access thousands of MSHRs and sub-entries on FPGA. Without loss of generality, we evaluate our MHA on a simple parallel SpMV accelerator operating on a set of SuiteSparse matrices [8], which we use as representative of latency-tolerant and bandwidth-bound applications with various degrees of locality. Our architecture extends the space of possible custom memory hierarchies on FPGA, providing additional Pareto-optimal and even Pareto-dominant points in the area-performance space compared to only using blocking caches or non-blocking caches with associative MSHR lookup. Furthermore,

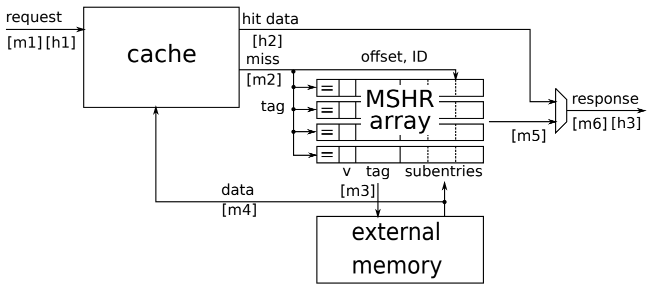

Figure 3: Structure of a non-blocking cache. On a hit (steps [h1]-[h3]), it behaves just like any cache. On a miss (steps [mi] to [m6]), the miss address and data is stored in an MSHR [m2]. If it is the first miss to a particular cache line, a memory request is sent [m3]. When the data returns from memory [m4], it is stored in the cache and used to respond to all the pending misses to this cache line [m5], [m6].

we will show that the benefit of repurposing some FPGA memory from cache to MSHRs increases as the memory access pattern gets irregular.

2 BACKGROUND

2.1 Non-Blocking Caches

Figure 3 shows the organization of a typical non-blocking cache. A non-blocking cache contains a miss handling architecture (MHA), based on an array of miss status holding registers (MSHRs), which keeps track of the in-flight misses. Each MSHR refers to one missing cache line and contains a valid bit, the tag of the cache line and one or more subentries to handle multiple misses to the same cache line.

2.2 FPGA On-Chip Memory

Modern FPGAs have at least three types of on-chip memory: flip flops, LUTRAM, and block RAM. Each bit of flip flop-based memory is exposed to the FPGA fabric, providing the highest flexibility in terms of number, type, and width of memory ports and the largest bandwidth. However, flip flop bits are the least abundant and some LUTs must be consumed to implement their access logic. LUTRAMs use LUTs to realize single-, dual-, or quad-port memories with medium depth (32-64 entries). However, they compete with combinational logic for LUTs. Block RAMs are dedicated memory resources implemented as hard logic. They provide the highest memory density and do not require any soft logic; however, they generally provide only two ports and are optimized for narrow and deep memory arrays (at least 512 entries). Therefore, the challenge is to use block RAM as much as possible, followed by LUTRAM and flip flops.

3 KEY IDEAS

3.1 Scalable MSHR Lookup and Storage

For each additional MSHR, the memory system can handle one more primary miss without stalling; similarly, each additional subentry allows servicing an extra secondary miss with no additional traffic to the external memory. Each MSHR has modest storage requirements: ~20-30 bits for the cache line tag and its valid bit, plus ~10-20 bits for offset and request ID for each of the ~4-8 subentries. This is significantly smaller than a 512-bit cache line with its tag. Therefore, within a given on-chip memory budget, bandwidth-bound applications with irregular memory access patterns could benefit more from an increase of the number of MSHRs or subentries, which increase MLP, rather than from an expansion of the cache. In practice, however, scaling up the fully associative MSHR array (Figure 4a) also requires additional comparators and a wider multiplexer, which increase area and hurt the critical path. Moreover, on FPGA, associative MSHRs can only be mapped to flip flops, whereas an n-way set associative cache can be implemented with n block RAM modules.

A set-associative MSHR memory (Figure 4b), indexed by the lowest significant bits of the tag, can be easily mapped to block RAM and, as long as there are no collisions, lookups, insertions, and deletions can be performed in a single step. Stalling is the simplest collision handling mechanism; however, we will show in Section 6.3 that this strongly limits the maximum load factor. Using linear probing would result in *expected* constant time lookup, insertion, and deletion and, whenever any operation cannot be completed in a single step, incoming misses must be stalled.

To overcome these limitations, we propose to store MSHRs using cuckoo hashing (Figure 4c). Cuckoo hashing uses d hash tables and d hash functions $h_0, \ldots, h_{d-1}$; each key x can be stored in any hash table in bucket $H_i[h_i(x)]$. Lookups and deletions require *worst case* constant time: both involve one lookup per hash table, plus one update for deletions. For insertions, key x can be inserted in any hash table whose bucket $h_i(x)$ is empty. If all possible locations $H_i[h_i(x)]$ are occupied, a *collision* occurs: the new key x displaces an existing entry to one of its alternative locations. If all possible buckets of the displaced entry are also occupied, the process is repeated recursively until an entry can be inserted into an empty bucket. This means that insertions can still require more than one operation, during which no other misses can be handled. Expected amortized insertion time is constant as long as the load factor is bounded; the bound is 50% for $d = 2$ and grows very quickly with d [11]. To de-amortize insertion, Kirsch et al. proposed to temporarily store displaced entries in a small content-searchable queue (*stash*) [18] (Figure 4d). As soon as the input interface is idle, the MHA tries to insert the oldest entry from the stash; if this results in a collision, another entry from a different hash table is moved to the stash. By doing so, entry reinsertion effectively happens in the background without slowing down incoming requests; incoming allocations are stalled only when the stash gets full.

3.2 Flexible Subentry Storage

For their explicitly addressed MSHR architecture, Farkas and Jouppi propose to use a fixed number of subentry slots per MSHR (Figure 5a) and to stall the MHA whenever all slots of an MSHR are used. However, waiting for the specific MSHR that is full to be deallocated may take a long time, during which the MHA may miss opportunities for merging requests to in-flight cache lines, which is particularly bad in out context. Increasing the number of slots per MSHR would reduce the probability of stall at the expense of

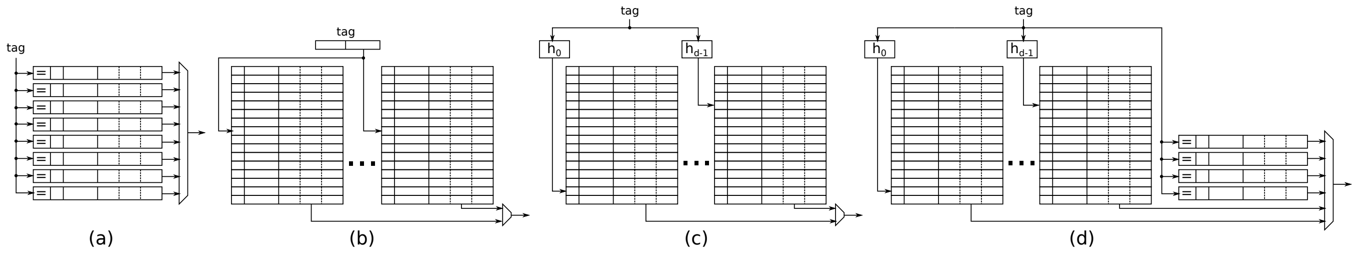

Figure 4: MSHR-rich architectures for FPGAs. Because of the associative lookup, a traditional architecture (a) does not scale beyond a few MSHRs and MSHRs can only be mapped to flip flops. Using a set-associative memory with a single hash function (b) allows MSHRs to be mapped to block RAM but stalling on every collision results in low load factors. Cuckoo hashing (c) reduces the probability of collision and a stash (d) allows collisions to be handled in the background when the unit is idle.

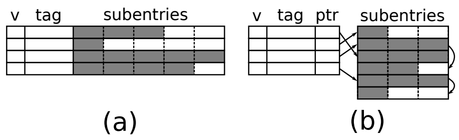

Figure 5: Subentry organization in memory. Allocating a fixed number of subentries to every MSHR (a) results in a difficult tradeoff between a low maximum load factor and a high probability of stall, especially if there is a large variation in the number of secondary misses per cache line. Using a separate buffer to store blocks of subentries organized as linked lists (b) provides greater flexibility at a modest cost.

an increase in area or, in other words, a decrease in load factor due to increased internal fragmentation. To mitigate these drawbacks, we propose a hybrid approach (Figure 5b): we store subentries in a separate buffer and we dynamically allocate blocks of subentries to each MSHR. Specifically, the subentry buffer, mapped to block RAM, contains N_R subentry rows, each comprising n_s slots. Each MSHR is initially assigned one subentry row; whenever a row gets full, an additional row is allocated for that MSHR. Subentry rows are logically organized as a linked list: the head pointer is stored in the MSHR buffer and each subentry row contains a field for the pointer to the next row. We will evaluate the benefits of the linked-list architecture in Section 6.4.

4 DETAILED ARCHITECTURE

Figure 6 shows the top-level view of our memory controller based on an MSHR-rich MHA. To simplify the design and to maximize the scope for memory access optimization, our controller can return responses out of order, which is not unusual among high performance memory systems [14, 15]. Therefore, requests must be tagged with an ID, which will be used to match it with the corresponding response. Requests received from each of the N_i input channels are redistributed across N_b banks by means of a crossbar. We use a multi-banked structure in order to handle multiple requests and responses per cycle. Requests pertaining to consecutive cache lines are served by different banks, an interleaved scheme commonly used in multi-banked caches. Each bank consists of a set-associative cache, an MSHR buffer, and a subentry buffer. Data for requests that hit in the cache are immediately returned to the crossbar, while

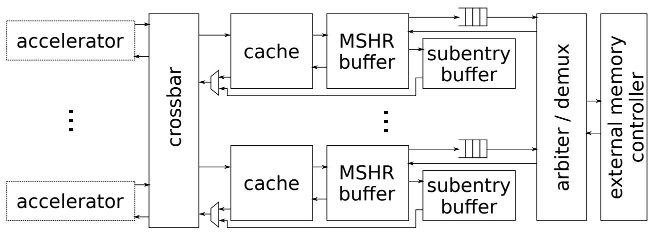

Figure 6: Top-level view of our memory controller. A crossbar steers memory requests from N_i accelerators to N_b banks according to their address. Each bank consists of a cache, an MSHR buffer and a subentry buffer. An arbiter time-multiplexes the input channel(s) of the external memory controller among banks.

misses are handled and stored by the MSHR and subentry buffer. On a primary miss, we also generate a memory request; requests from each bank are forwarded to a round-robin arbiter and then to the external memory controller. Cache lines received from the external memory controller are multicasted both to the cache and to the subentry buffer, which generates the responses to the cache line's pending misses.

4.1 MSHR Buffer

For the MSHR buffer, we use one block RAM per hash table, with the address of the cache line (tag) as key. We use universal hash functions in the form $h_a(x) = (ax \bmod 2^{w_t}) \operatorname{div} 2^{w_t - w_M}$ with w_t being the number of bits of the tag, $w_M = \log_2(M)$ where M is the number of buckets per hash table, and a is a random positive odd integer with $a < 2^w_t$ [29]. Each bucket contains a valid bit, the tag of the missing cache line, and the address of the first subentry row in the subentry buffer as described in Section 3.2. The stash is a content-associative memory made of flip-flops. To integrate the stash in the pipeline, we include the stash entries among the locations that are searched during lookups or that can be deallocated when a response is received.

4.2 Subentry Buffer

Figure 7 shows implementation and operation of the subentry buffer. A subentry consists of an (ID, offset) pair; a subentry row contains a) n_s subentry slots, b) the number of allocated subentries, and c)

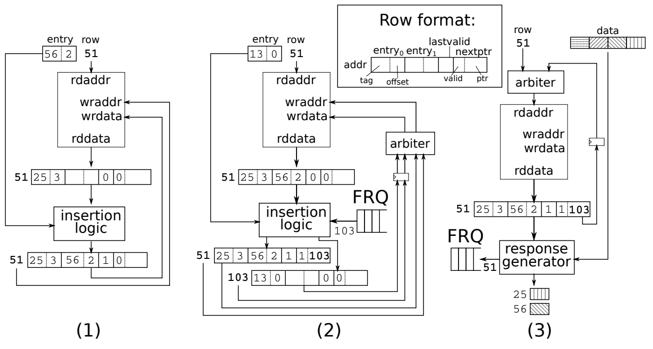
Figure 7: Block diagram and operation of the subentry buffer. For requests, the subentry buffer receives ID, offset, and the address of the first subentry row (head row) from the respective MSHR. The head row is firstly retrieved from the buffer. If it is not full (1), the row is updated with the new entry and written back to the buffer. If the row is full (2), the new entry is inserted in a new row, whose address is stored in the previous row. When a response is received (3), all subentries are retrieved by traversing the subentry row list. After all subentries have been forwarded to the response generator, the row is deallocated by pushing its address to the free row queue.

a pointer to the next subentry row with its valid bit. To allocate a subentry, the first subentry row is retrieved from the buffer. If the row is not full (1), the new entry is appended and the row is written back to the buffer. If the row is full (2), a new row must also be allocated. We use a FIFO (free row queue, FRQ) to store the addresses of the empty rows, and allocating a row simply means extracting the first element of the FRQ. The FRQ is also shared with the MSHR buffer to allow the allocation of the first subentry row for newly allocated MSHRs. When the FRQ gets empty, further allocations are stalled.

When a cache line is received, the corresponding MSHR is deallocated from the MSHR buffer and its subentry rows retrieved from the buffer. The response generator parses the subentry rows and emits one response per allocated subentry. The row is then recycled by inserting its address into the FRQ and the process is repeated for the entire linked list of rows.

4.3 Pipeline Efficiency and Throughput

As long as an MSHR has a single subentry row, the primary and all secondary misses can be handled without stalling the pipeline as they require no more than one read and one write per dual-ported block RAM: lookup in the MSHR buffer, allocation of the MSHR for primary misses, lookup in the subentry buffer for secondary misses, and row update in the subentry buffer. Each block RAM has a data forwarding circuit to ensure that we always read the most up-to-date data despite reads having two-cycle latency. MSHR collisions are handled transparently when the unit is idle, as long as there are free entries in the stash. Allocating an additional subentry row requires stalling the pipeline for one cycle to perform two writes: inserting the pointer of the newly allocated row into the tail of the

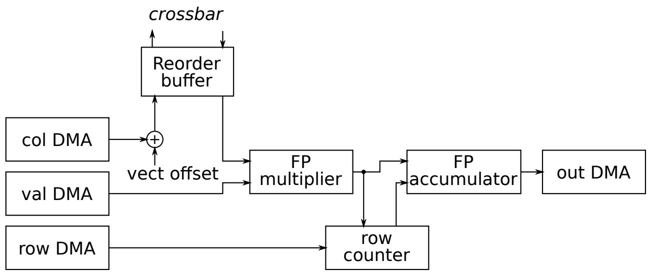

Figure 8: Structure of our benchmark sparse matrix-vector multiplication accelerator. Xilinx AXI DMAs are used to stream all CSR vectors accessed sequentially. The values of the `col` array are used to compute the addresses of the vector elements that are retrieved through our memory controller.

list, and writing the new subentry into the newly allocated row. Allocating a subentry on an MSHR that has more than one row requires traversing the linked list, which costs an extra read per additional row. The traversal cost can be significant for MSHRs with many subentries: to mitigate it, we use an 8-entry fully-associative cache indexed by the head pointer of the subentry list to jump directly to the tail whenever possible. In our subentry architecture, the tradeoff between internal fragmentation and stall cycles, which depend on the number of subentries per row, remains; however, the cost of a full subentry row is reduced from completely stalling the pipeline until the full MSHR is deallocated to a few bubbles in the pipeline. Responses whose MSHR has a single subentry row can also be handled without stalls; each additional subentry row costs one stall cycle.

Most of the operations are therefore fully pipelined, with the caveat that a single pipeline is shared between accelerator requests and memory responses. However, the more secondary misses we can merge to the same memory request, the fewer memory responses we will have to handle, reducing the cost of pipeline sharing. Ultimately, N_b fully-pipelined banks can supply up to $N_b - 1$ responses per cycle from a single-ported external memory.

5 EXPERIMENTAL SETUP

We wrote our memory controller in Chisel 3, compiled it with Vivado 2017.4 and tested it on a ZC706 board with an XC7Z045 SoC. The board has 1 GB of DDR3 on the processing system side, connected to the dual-core ARM's memory controller, and 1 GB of DDR3 on the programmable logic side which can be accessed directly from the FPGA. The FPGA has 437,200 flip flops, 218,600 LUTs—which could implement up to 4.3 Mib of LUTRAM, if no LUTs were used for logic—and 1090 18 kib block RAM modules (19.2 Mib of block RAM).

As a benchmark, we implemented a simple accelerator for sparse matrix-vector multiplication (SpMV), an important kernel in a broad range of scientific applications [3] and to which many sparse graphs algorithms can be mapped [17]. Moreover, SpMV can easily generate a wide range of access patterns depending on the matrix sparsity pattern. Our accelerator, shown in Figure 8, is an almost direct implementation of Algorithm 1 for SpMV of a CSR-encoded sparse matrix; we do not include any SpMV-specific optimizations as our

matrix	NZ	rows	vect size	st. dist. percentiles		
				75%	90%	95%
dblp-2010	1.62M	326k	1.24 MB	2	348	4.68k
pds-80	928k	129k	1.66 MB	26.3k	26.6k	26.6k
amazon-2008	5.16M	735k	2.81 MB	6	6.63k	19.3k
flickr	9.84M	821k	3.13 MB	3.29k	8.26k	14.5k
eu-2005	19.2M	863k	3.29 MB	5	26	69
webbase_1M	3.10M	1.00M	3.81 MB	2	19	323
rail4284	11.3M	4.28k	4.18 MB	0	13.3k	35.4k
youtube	5.97M	1.13M	4.33 MB	5.8k	20.6k	32.6k
in-2004	16.9M	1.38M	5.28 MB	0	4	11
ljournal	79.0M	5.36M	20.5 MB	19.3k	120k	184k
mawi1234	38M	18.6M	70.8 MB	20.9k	176k	609k
road_usa	57.7M	23.9M	91.4 MB	31	601	158k

Table 1: Properties of the benchmark matrices we used. The number of columns corresponds to the vector size divided by 4 bytes and, except for pds-80 and rail4284, it corresponds to the number of rows.

controller aims for a generic architectural solution for any applications with irregular memory access pattern. Indices are 32-bit unsigned integers while values are single-precision floating point values. All CSR vectors, accessed sequentially, are provided via AXI4-Stream through DMAs; the dense vector, accessed randomly, is read through an AXI4-MM port connected to our memory controller. The 8192-entry reorder buffer provides the vector values to the multiply-accumulation pipeline, which is based on floating-point Xilinx IPs. We use the index vector to clear the accumulator every time a new row begins, and the output vector is streamed to DDR through a DMA. Each accelerator can process one non-zero matrix element (NZ) per cycle; we parallelize the SpMV by interleaving rows across multiple accelerators.

All data structures are stored in DDR3 memory. Access to the programmable logic (PL) memory occurs through the soft MIG controller, which exposes a single AXI-MM port. The only way to possibly exploit the full DDR3 bandwidth (12.8 GB/s) is to use a 512-bit wide interface running at 200 MHz. Therefore, we set the clock frequency of all of our designs to 200 MHz. The processing system (PS) memory can be accessed from the FPGA through five 64 bit-wide ports on the processing system-programmable logic bridge logic. On non burst accesses, we noticed a higher performance degradation on the PS ports compared to the MIG; therefore, we decided to use the PS memory to store the vectors that are accessed sequentially and the PL memory for the vector that is accessed randomly through our controller. By isolating random and sequential accesses, we also prevent any possible influence of the DMA accesses on the performance of the random memory operations.

Algorithm 1 Sparse matrix-vector multiplication (SpMV)

```
1: for $r \leftarrow 0$ to $ROWS - 1$ do
2:     $out[r] \leftarrow 0$
3:     for $i \leftarrow idx[r]$ to $idx[r+1]$ do
4:         $out[r] \leftarrow out[r] + val[i] \times vect[col[i]]$
5:     end for
6: end for
```

	LUT	FF	BRAM	DSP
4 accelerators	**4640**	**6472**	**0**	**32**
16 DMAs	**12692**	**17192**	**144**	**0**
MIG	**13373**	**9380**	**2**	**0**
4x4 Crossbar	1644	3412	0	0
Bank arbiter/demux	334	1101	1	0
Cache with assoc. MSHR	9664	14000	0	0
Traditional MHA	**12301**	**19391**	*	**0**
Cache with cuckoo MSHR	1900	8396	*	*
Subentry buffer	7696	4228	*	0
Proposed MHA	**11186**	**18208**	*	*

Table 2: Resource utilization of the entire system. Rows not in bold represent MHA sub-modules. The asterisks denote values that depend on the controller's configuration. The proposed MHA has very similar LUT and FF utilization than the baseline 16 MHSR, 8 subentry associative MHA.

Each iteration of the inner loop of Algorithm 1 consumes three 32-bit words: two provided by the DMAs (`val[i]` and `col[i]`) and one from the vector. Given a measured bandwidth of 3.5 GB/s on all PS ports and 12.0 GB/s on the MIG, the bottleneck will be on the PS memory, which limits the system throughput to ~2.4 NZ/cycle (ignoring, for now, the minor bandwidth requirements for the `idx` and `out` vectors). However, without any memory system on the single-port MIG side, the theoretical throughput would be limited to 1 NZ/cycle instead, and in practice even up to 40% less due to DRAM row conflicts [2]; therefore, there is still at least a 2.4× scope for speedup that relies entirely on the optimization of the random memory accesses. To ensure that neither the accelerators, nor the controller's banks, nor the crossbar become the bottleneck, we instantiate four SpMV accelerators and four banks. We use one of the ARM cores to initialize the data in the DDR memories, to orchestrate the DMAs and the accelerators and to check the correctness of the output vectors.

Table 1 shows the properties of our benchmark matrices, which are essentially the largest benchmarks used in prior work on SpMV [3]. All benchmarks are available on SuiteSparse [8]. We use the stack distance, introduced in Section 1.2, to characterize the regularity of the access pattern to the dense vector. All benchmarks except for dblp-2010 and pds-80 operate on a vector that does not fit in the FPGA memory, which means that they are out of the scope of approaches based on transferring the entire vector to block RAM [9, 13].

6 EXPERIMENTAL RESULTS

6.1 Resource Utilization

Table 2 shows the resource utilization of the entire system, with proposed and baseline MHA. The traditional MHA contains the largest number of associative MSHRs that can be implemented under the 200 MHz clock constraint: 16 MSHRs with 8 subentries each. We do not consider the case of a blocking cache because it performs significantly worse than the non-blocking cache for modest area savings. The values for the proposed MSHR refer to three cuckoo hash tables and a 2-entry stash; four hash tables and a 4-entry

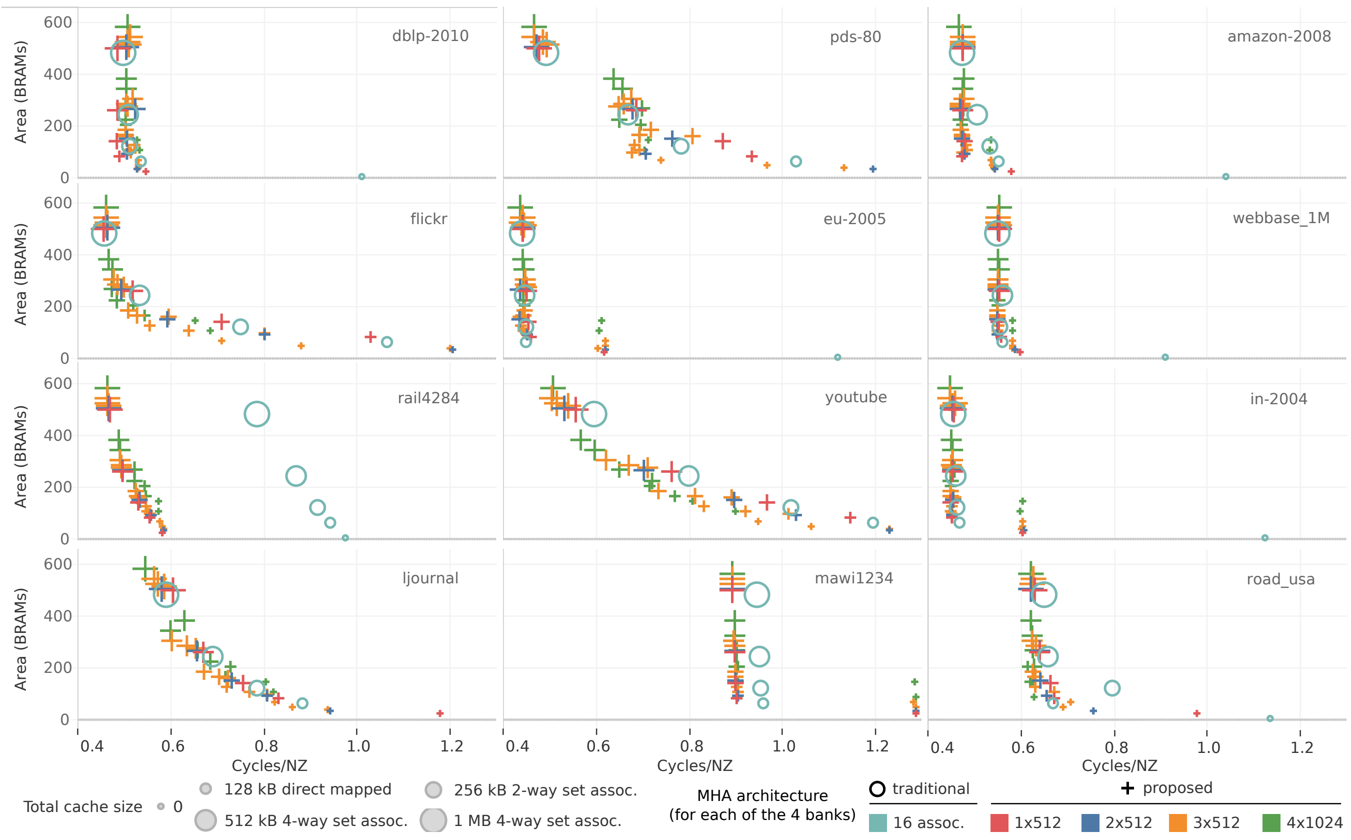

Figure 9: Area of the memory system and normalized execution time for all benchmarks and a broad range of MHA architectures. For the cuckoo architectures, we indicate the number and depth of cuckoo hash tables in each of the four banks, whereas the cache size refers to the entire multi-banked structure. Charts are sorted by increasing vector size and have been truncated at 1.3 cycles/NZ. On half of the benchmarks, all the Pareto-optimal designs are cuckoo MHAs, except for the smallest possible but low-performing design with no cache and associative MSHRs. For the other benchmarks, our MHA provides additional Pareto-optimal designs, especially on the low area side. Designs with no cache nor MSHRs, where the DRAM is time-multiplexed among the accelerators, are 1.9×-7.8× (4.1× geomean) slower than the best performing design. On the other hand, with a 1 MB blocking cache, the slowdown is 1.4×-14.6× (4.5× geomean).

stash costs about 300 LUTs and FFs more. The BRAM utilization of cache, MSHR and subentry buffers depend on their sizing, which will be explored in Section 6. Indicatively, the minimum cache that is worth implementing due to the minimum block RAM depth – a single 32 kB way (512 lines × 512 data bits) – has similar block RAM requirements as 3×512 MSHRs with 3×2048 subentries. In general, the cache requires 15 block RAMs per 32 kB per cache way, the MSHR buffer requires 1 block RAM per 512 MSHRs per cuckoo hash table for storage plus 1 block RAM per 512 MSHRs for the request queue to the external memory arbiter, and the subentry buffer requires 2 block RAMs per 512 subentry rows of up to 3 subentries each, plus 1 block RAM every 1024 subentry rows for the FRQ. Each cuckoo hash function also uses 1 DSP block.

6.2 Performance Evaluation

We ran our benchmarks on a set of different memory controllers, both with associative and cuckoo-based MHAs. We used 4-way set associative caches except in the smallest caches due to the limited minimum block RAM depth (see Section 6.1). For the associative MHAs, we only consider the best architecture that can run at 200 MHz, with 16 MSHRs with 8 subentries each. For the cuckoo MHAs, we fixed the number of subentries per row to three since, due to the finite choice of block RAM port widths, they occupy the same amount of block RAMs as two and provide a good compromise between utilization and stall cycles (see Section 6.4). We also fixed the stash size to two entries, which provides timing closure in all cases. We explored the number and depth of MSHR hash tables, as well as the depth of the subentry buffer.

Figure 9 summarizes the results. Our MSHR-rich MHAs provide the highest performance benefit to the benchmarks with the highest stack distance percentile (90% and 95%), i.e. the most challenging ones for caches. With rail4284, misses to multiple cache lines are so frequent that even the smallest MHA with no cache at all performs 25% better than the traditional MHA with the largest cache, which has a 24× larger area. On mawi1234, a small cache is enough to capture any existing temporal locality; after that, investing 2% of

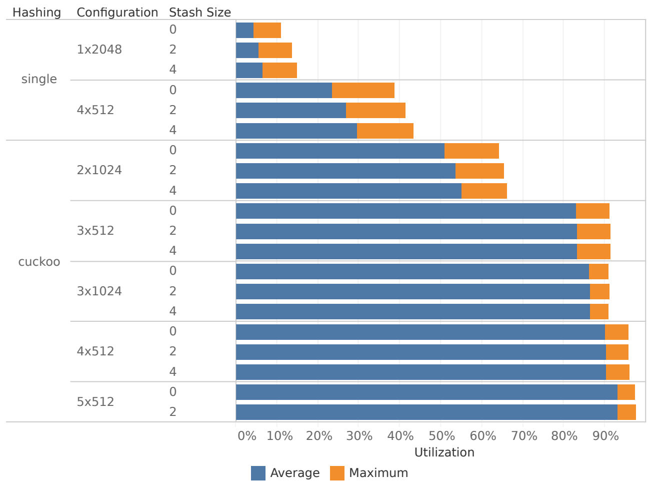

Figure 10: Achievable MSHR storage load factor for several MSHR architectures. The 5×512 system with a 4-entry stash did not meet timing constraints. Single-hash architectures cannot utilize more than 40% of the storage space. Cuckoo hashing can handle collisions more efficiently and three hash tables are enough to achieve more than 80% average and 90% peak load factors, even without stash.

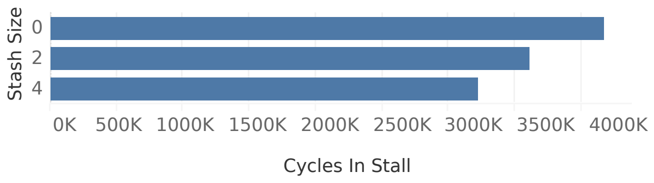

Figure 11: Number of cycles lost due to stalls for collision resolution during the execution of a uniformly distributed benchmark. A 4-entry stash, which occupies less than 0.1% of LUTs and FFs, reduces the number of stall cycles by 30%.

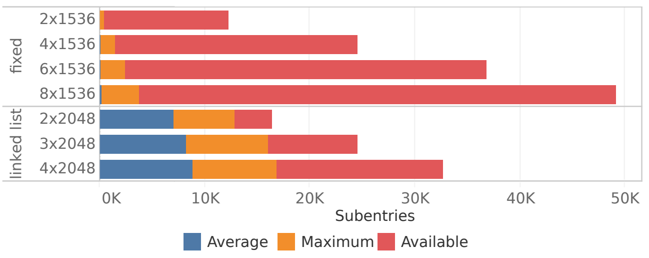

Figure 12: Average and maximum subentry utilization during the execution of ljournal with a 3x512 cuckoo MSHR. Allocating a fixed number of subentries per MSHR results in less than 1% average utilization and resource waste. Linked-list architectures provide a more efficient usage of the subentry memory.

block RAMs for the smallest proposed MHA provides higher returns than any further increase in cache size. Pds-80, flickr, youtube, and ljournal offer a more gradual area-delay tradeoff and can benefit from the largest MSHR solutions, which constitute most of the Pareto-dominant points. On these benchmarks, we achieve 10% to 25% throughput increase with the same area or 35% to 60% area reduction at constant throughput. Dblp-2010, eu-2005, in-2004, and webbase_1M have higher locality and thus benefit more than other benchmarks from larger caches; however, the simplest proposed MHA with no cache, which uses 3× fewer BRAMs than the smallest cache, is enough to saturate the PS DRAM bandwidth only by merging memory requests. On eu-2005 and in-2004, the performance gain provided by the cache-less MHAs is limited by handling the subentry linked lists. Applications with higher temporal locality may thus benefit from an increase of subentries per row. Benchmarks with few non-zero elements per row such as mawi1234 and road_usa have a lower maximum performance due to the higher bandwidth requirements for the sequential vectors; however, they are among the eight benchmarks that do not saturate the PS DRAM bandwidth without an MSHR-rich MHA.

6.3 Number of MSHR Hash Tables and Stash Size

Figure 10 analyzes the performance of the MSHR storage architectures described in Section 2.2. For each architecture, we measure average and peak utilization of the MSHR storage space. To make sure the benchmark always uses all of the available MSHRs, we use a synthetic 1M×1M matrix with 5M uniformly distributed non-zero elements generated with the Python function `scipy.sparse.random(1e6, 1e6, 5e-6)`, no cache, and each bank contains 4096 subentry rows with 3 subentries each. All architectures have 2048 MSHRs per bank or the closest possible value.

Because any collisions result in a stall that lasts until one of the colliding MSHRs is deallocated, all of the single-hash architectures achieve poor utilization: even by introducing a stash to tolerate up to four collisions, a 4-way set-associative architecture does not go beyond 30% average and 45% peak load factors. Even a simple 2-way cuckoo hash table achieves 50% average and 70% peak utilization, and three ways enough to reach more than 80% average utilization, which is consistent with prior findings on cuckoo hashing [11]. Interestingly, using a 3-way 512-entry architecture (1536 MHSRs) has higher absolute utilization than a 2-way, 1024-entry organization (2048 MSHRs). For three or more ways, adding a stash does not affect MSHR utilization but decreases the number of stall cycles by up to 30% with a 4-entry stash (Figure 11), which is the largest stash that we could implement within the 200 MHz constraint.

6.4 Subentry Organization

We performed a similar analysis for the memory organization of the subentries, as described in Section 3.2. We use the ljournal benchmark, which has a large number of secondary misses, and an MHA with no cache and a 3×512 cuckoo MSHR buffer per bank. As shown in Figure 12, with a fixed number of subentries per MSHR, stalls are so frequent (Figure 13) that they prevent misses from accumulating in the buffers, resulting in very low utilization but also fewer opportunities for request merging and thus a higher traffic to external memory (Figure 14). We believe this problem is more pronounced in an MSHR-rich architecture than in an associative

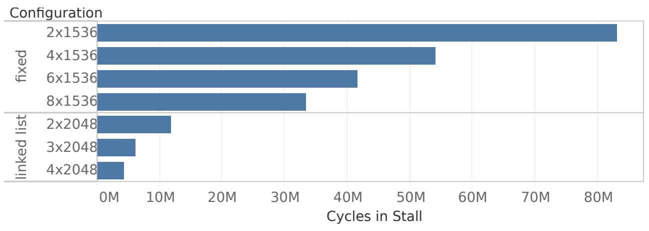

Figure 13: Number of cycles lost due to subentry-related stalls. Stalls occur when (a) filling all subentries of an MSHR for the fixed architectures or (b) handling the linked list or running out of subentry rows for the linked list architectures. The smallest linked list architecture has three times fewer stall cycles than the largest fixed architecture despite having three times fewer subentries.

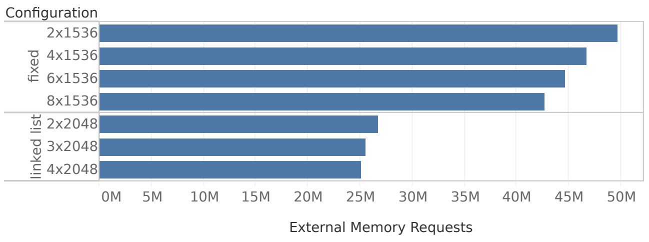

Figure 14: Number of external memory requests during the execution of ljournal with a 3x512 cuckoo MSHR and no cache. By increasing subentry utilization, linked list architectures increase the number of accelerator requests that can be served by the same external memory request, resulting in a 37% decrease of external memory traffic.

MHA because it is far more likely to encounter at least one MSHR that needs more than a given number of subentries when handling thousands of misses rather than a few tens of them. Our linked list-based architectures provide much higher average and maximum buffer utilization, fewer stall cycles and decrease the number of DDR memory requests by a factor 1.3× to 2×, with evident great energy impact.

7 RELATED WORK

7.1 Miss Handling Architectures

The first non-blocking cache was proposed by Kroft in 1981 [20]. Farkas and Jouppi [10] evaluate a number of alternative MHA organizations, including the explicitly-addressed MSHRs that inspired our MHA. They observed that non-blocking caches can reduce the miss stall cycles per instruction by a factor 4 to 10 compared to blocking caches, that the most aggressive architectures (such as explicitly-addressed) are beneficial even for large cache sizes, and that overlapping as many misses as possible allows processors to maximize the benefit provided by non-blocking caches.

Tuck et al. [25] introduced a novel MHA for single processor cores with very large instruction windows. They propose a hierarchical MHA, with a small explicitly-addressed MSHR file for each L1 cache bank and a larger shared MSHR file. MSHRs are explicitly-addressed and shared MSHRs have more subentries than the dedicated ones. On a number of SPEC2000 benchmarks running on a 512-entry instruction window superscalar single-core processor, dedicated files with 16 MSHRs and 8 subentries and a shared file with 30 MSHRs and 32 subentries achieve speedups that are close to those provided by an unlimited MHA. However, we believe that a set of parallel accelerators is fundamentally different from a single-core processor even with a large instruction window for two reasons: a) parallel accelerators with, for instance, decoupled access/execution architectures [7, 22] could generate even more requests per cycle with no fundamental limitations on the total number of in-flight operations, and b) requests to be merged can come from the same as well as a different accelerator, so it is important to have a shared MHA to maximize the merging opportunities. Our results indeed showed that, for parallel accelerators with massive MLP, small caches with thousands of MSHRs can achieve similar or even better performance of larger caches with few MSHRs.

7.2 Memory Systems for Irregular Memory Accesses

Several pieces of work aimed at improving the efficiency of traditional caches on non-contiguous memory accesses. Impulse [5] introduces an additional address translation stage to remap data that is sparse in the physical/virtual memory space into contiguous locations in a shadow address space. However, it is a processor-centric system which relies on the intervention from the OS to manage the shadow address space. Traversal caches [24] optimize repeated accesses to pointer-based data structures on FPGA. Such approach is however limited to pointer-based data structures that are repeatedly accessed and that can fit entirely in the FPGA block RAM.

Another line of work explored the automatic generation of application-specific memory systems. Bayliss et al. [4] proposed a methodology that automatically generates reuse buffers for affine loop nests, which reduce the amount of memory requests and of DRAM row conflicts. However, the approach is restricted to kernels consisting of an affine loop nest whose bounds are known at compile time. TraceBanking [30] does not rely on static compiler analysis and uses a memory trace to generate a banking scheme that is provably conflict-free. It also supports non-affine loop nests but requires the dataset to fit entirely in the block RAM. ConGen [16] focuses on optimizing DRAM accesses without relying on any local buffering on FPGA. It uses a memory trace to generate a mapping from the addresses generated by the application to DRAM addresses such that the number of row conflicts is minimized. All of these solutions rely on exact information about the application's memory access pattern at hardware compile time. Our approach is application-agnostic, fully dynamic and does not make any assumptions on the access pattern properties.

Coalescing aims at increasing bandwidth utilization between datapath and DRAM by merging multiple narrow memory accesses into fewer, wider ones. Modern GPUs dynamically coalesce accesses from the same instruction executed by different threads [26] in the same warp, and the load-store units instantiated by the Intel FPGA OpenCL compiler can perform both static and dynamic burst coalescing [28]. To increase the opportunities for coalescing and thus the utilization of the bandwidth to the GPU L1 cache, Kloosterman et al. propose an inter-warp coalescer [19]. Wang et al. [27] proposed a dynamic coalescing unit for HMC memories in a multi-core system, implemented on a small RISC-V core. Incoming requests are stored in a binary tree and forwarded to the HMC after a timeout or after receiving 128 bytes of requests. All of these approaches have a very short window where coalescing can occur, at most a few requests wide. We showed that explicitly-addressed MSHRs also perform coalescing, on wider request windows and over multiple bursts at the same time (one per MSHR).

8 CONCLUSION

Conventional wisdom has it that some form of local buffering such as caching is the best way to optimize the access to external memory, hence the vast effort in maximizing the hit rate under all possible scenarios. Non-blocking caches are one of the few architectures for *miss* optimization instead. In this paper, we took the key idea behind non-blocking caches to the extreme: we designed a scheme to handle three orders of magnitude more misses without stalls compared to classic fully-associative MHAs. We presented an efficient FPGA implementation of such MSHR-rich cache, where we map tens of thousands of MSHRs and subentries to the abundant FPGA block RAM and all stages of miss handling are pipelined with minimal stalls. On twelve sparse matrix-vector multiplication benchmarks, most of which cannot fit in the FPGA block RAM, we showed that, under a limited block RAM budget, repurposing some block RAMs from cache to MSHRs can provide higher performance gains when the access pattern is such that a relevant amount of misses cannot be avoided. This is especially true for the benchmarks with the lowest temporal locality, but even on more regular access patterns, MSHRs can complement caches by optimizing long-distance reuse, providing similar performance gains as a larger cache at lower area costs. Therefore, we believe MSHR-rich MHAs open up new opportunities to increase performance of bandwidth-bound, latency-insensitive applications with irregular memory access patterns. Our MHA, as well as the benchmark SpMV accelerators, can be downloaded as an open-source project from `https://github.com/m-asiatici/MSHR-rich`.

ACKNOWLEDGMENTS

The authors would like to thank Dick Sites, Stephen Neuendorffer, Kristof Denolf, and Kees Vissers for their valuable feedback and suggestions.

REFERENCES

[1] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. 2011. LEAP scratchpads: automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, Calif., 25–28.

[2] Adrian Cosoroaba. 2013. *White Paper 383 - Achieving High Performance DDR3 Data Rates*. Xilinx Inc.

[3] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. New Orleans, La., 781–792.

[4] Samuel Bayliss and George A. Constantinides. 2011. Application specific memory access, reuse and reordering for SDRAM. In *Proceedings of the 7th International Symposium on Applied Reconfigurable Computing*. Belfast, 41–52.

[5] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, and Terry Tateyama. 1999. Impulse: Building a smarter memory controller. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*. Orlando, Fla., 70–79.

[6] Calin Cascaval and David A. Padua. 2003. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th annual international conference on Supercomputing*. Phoenix, Az., 150–159.

[7] Tao Chen and G. Edward Suh. 2016. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. Taipei, Taiwan, 46.

[8] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.

[9] Richard Dorrance, Fengbo Ren, and Dejan Marković. 2014. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-BLAS on FPGAs. In *Proceedings of the 22nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, Calif., 161–170.

[10] K. I. Farkas and N. P. Jouppi. 1994. Complexity/Performance Tradeoffs with Non-blocking Loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*. Chicago, Ill., 211–222.

[11] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. 2005. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems* 38, 2 (2005), 229–248.

[12] Nithin George, Hyoukjoong Lee, David Novo, Tiark Rompf, Kevin Brown, Arvind Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. 2014. Hardware System Synthesis from Domain-Specific Languages. In *Proceedings of the 24th International Conference on Field-Programmable Logic and Applications*. Munich, 1–8.

[13] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43th Annual International Symposium on Computer Architecture*. Seoul, 243–254.

[14] Intel Inc. 2016. *Hybrid Memory Cube Controller IP Core User Guide*. Intel Inc.

[15] Intel Inc. 2018. *Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual*. Intel Inc.

[16] Matthias Jung, Deepak M. Mathew, Christian Weis, Norbert Wehn, Irene Heinrich, Marco V. Natale, and Sven O. Krumke. 2016. Congen: An application specific dram memory controller generator. In *Proceedings of the 2nd International Symposium on Memory Systems*. Alexandria, Va., 257–267.

[17] Jeremy Kepner and John Gilbert. 2011. *Graph algorithms in the language of linear algebra*. SIAM.

[18] Adam Kirsch and Michael Mitzenmacher. 2007. Using a queue to de-amortize cuckoo hashing in hardware. In *Proceedings of the 45th Annual Allerton Conference on Communication, Control, and Computing*, Vol. 75. Monticello, Ill., 751–758.

[19] John Kloosterman, Jonathan Beaumont, Mick Wollman, Ankit Sethia, Ron Dreslinski, Trevor Mudge, and Scott Mahlke. 2015. WarpPool: sharing requests with inter-warp coalescing for throughput processors. In *Proceedings of the 48th Annual International Symposium on Microarchitecture*. 433–444.

[20] David Kroft. 1981. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*. Minneapolis, Minn., 81–87.

[21] Sheng Li, Ke Chen, Jay B. Brockman, and Norman P. Jouppi. 2011. *Performance impacts of non-blocking caches in out-of-order processors*. HPL Tech Report.

[22] Feng Liu, Soumyadeep Ghosh, Nick P. Johnson, and David I. August. 2014. CGPA: Coarse-grained pipelined accelerators. In *Proceedings of the 51st Design Automation Conference*. San Francisco, Calif., 1–6.

[23] Mario D. Marino and Kuan-Ching Li. 2017. System implications of LLC MSHRs in scalable memory systems. *Microprocessors and Microsystems* 52 (2017), 355–364.

[24] Greg Stitt, Gaurav Chaudhari, and James Coole. 2008. Traversal caches: A first step towards FPGA acceleration of pointer-based data structures. In *Proceedings of the 6th International Conference on Hardware/Software Codesign and System Synthesis*. Atlanta, Ga., 61–66.

[25] James Tuck, Luis Ceze, and Josep Torrellas. 2006. Scalable cache miss handling for high memory-level parallelism. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*. Orlando, Fla., 409–422.

[26] Vasily Volkov. 2016. *Understanding latency hiding on GPUs*. Ph.D. Dissertation. UC Berkeley.

[27] Xi Wang, John D. Leidel, and Yong Chen. 2016. Concurrent dynamic memory coalescing on GoblinCore-64 architecture. In *Proceedings of the Second International Symposium on Memory Systems*. 177–187.

[28] Felix Winterstein and George Constantinides. 2017. Pass a pointer: Exploring shared virtual memory abstractions in OpenCL tools for FPGAs. In *Proceedings of the 2017 International Conference on Field Programmable Technology*. Melbourne, 104–111.

[29] Philipp Woelfel. 1999. Efficient strongly universal and optimally universal hashing. In *International Symposium on Mathematical Foundations of Computer Science*. Szklarska Poreba, 262–272.

[30] Yuan Zhou, Khalid Musa Al-Hawaj, and Zhiru Zhang. 2017. A New Approach to Automatic Memory Banking Using Trace-Based Address Mining. In *Proceedings of the 25th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, Calif., 179–188.