Modelling Data Locality of Sparse Matrix-Vector Multiplication on the A64FX

Sergej Breiter*
sergej.breiter@nm.ifi.lmu.de
Ludwig-Maximilians-Universität München
Munich, Germany

James D. Trotter*
james@simula.no
Simula Research Laboratory
Oslo, Norway

Karl Fürlinger
karl.fuerlinger@nm.ifi.lmu.de
Ludwig-Maximilians-Universität München
Munich, Germany

ABSTRACT

One of the novel features of the Fujitsu A64FX CPU is the *sector cache*. This feature enables hardware-supported partitioning of the L1 and L2 caches and allows the programmer control of which partition is used to place data in. This paper performs an in-depth study of how to apply the sector cache to a frequently used sparse matrix-vector multiplication (SpMV) kernel. A performance model based on reuse analysis is used to better understand situations where the sector cache leads to improved reuse and to predict the cache behavior. The model correctly predicts the number of L2 cache misses within 2–3 % for sequential and parallel SpMV with 48 threads using a collection of 490 sparse matrices. Further experiments show the effect of various sector cache configurations on performance. A median speedup of about 1.05× is achieved, whereas the maximum speedup is about 1.6×.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; • **Computing methodologies** → **Shared memory algorithms**; • **Mathematics of computing** → **Mathematical software performance**; • **General and reference** → *Performance*; *Estimation*.

KEYWORDS

sparse matrix-vector multiplication, A64FX, cache partitioning, sector cache, performance model

ACM Reference Format:
Sergej Breiter, James D. Trotter, and Karl Fürlinger. 2023. Modelling Data Locality of Sparse Matrix-Vector Multiplication on the A64FX. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3624062.3624198

1 INTRODUCTION

Fujitsu's A64FX is an HPC-oriented processor developed for the Fugaku supercomputer [22]. Not only is it one of the few ARM-based CPUs designed for HPC workloads [16], but the A64FX emerged through a co-design process involving hardware, software and scientific applications, which resulted in unique features such as high-speed HBM2 memory and Scalable Vector Extensions (SVE) with 512-bit vectors [22]. Several studies have investigated the performance of the A64FX on workloads ranging from microbenchmarks and proxy applications to realistic scientific computing workloads [3, 7, 12, 20]. So far little attention has been paid to one of the A64FX's more unusual features, namely the *sector cache*, which provides finer control over data placement in the CPU's data caches than is usually allowed. The underlying idea is simple: by partitioning the cache into sectors and assigning different data to each sector, one can avoid cache pollution or conflicts that lead to unnecessarily or prematurely evicting data that might instead be reused.

For problems involving dense matrices or structured grids, effective use of the sector cache has been shown to reduce cache misses by 15–50 % [5]. Moreover, in the case of dense matrix-vector multiplication, performance improves by a factor of 2.8× for vector sizes in the range 2–6 MB [1]. The benefit of the sector cache in this case is well understood. However, in the case of sparse matrix-vector multiplication (SpMV), previous studies indicate that the sector cache provided little benefit when tested on a range of sparse matrices, yielding only a minor performance improvement of up to about 10–20 % in rare cases [1].

The aim of this paper is to study how to use the A64FX sector cache for sparse kernels, such as SpMV, that are faced with irregular and indirect memory access patterns. To do so, we present a performance model based on reuse analysis to describe the cache behavior of SpMV from the sparsity pattern and dimensions of the input matrix. The model is applied to a typical SpMV kernel for the Compressed Sparse Row (CSR) format and incorporates the effects of the sector cache. Finally, we present measurements of cache misses and performance under different sector cache configurations on the A64FX using a variety of input matrices, which show that our model accurately describes the last-level cache behavior and the effect of cache partitioning for large matrices. In summary, our paper makes the following contributions:

- A comprehensive performance study and in-depth analysis of CSR SpMV on the A64FX, including the effect of the sector cache feature
- A locality analysis based on reuse distance of sparse matrices using their sparsity pattern
- A cache miss model for parallel codes on multicore architectures with multiple shared caches including the effect of cache partitioning

The rest of the paper is organized as follows. First, we explain the cache partitioning mechanism of the A64FX and reuse distance in Section 2. Second, we analyze the influence of matrix dimensions

*Both authors contributed equally to this research.

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624198

together with the effect of cache partitioning on SpMV, and describe our approach to derive the cache miss model in Section 3. The description of our experiments and an analysis of the results follows in Section 4. Related work is discussed in Section 5 before we conclude in Section 6.

2 BACKGROUND

In this section, we first describe the cache partitioning mechanism of the A64FX and how it can be applied to a CSR SpMV kernel. Afterwards, we briefly explain reuse distance and how it can be used to assess cache behavior.

2.1 Cache partitioning on A64FX

Cache partitioning allows dividing a cache into multiple partitions. Fujitsu's A64FX processor is equipped with a way-based hardware cache partitioning mechanism, named *sector cache* [10]. It enables dividing the private L1D and shared, last-level L2 caches into at most four partitions, called *sectors*, and assigning a program's data objects (e.g., arrays) to the different partitions. The *partitioning policy* (i.e., partition sizes and data assignment) can be chosen dynamically at runtime and without flushing the cache. Partition sizes are set by allocating a number of cache ways to each sector. The assignment of data to a partition is specified by setting a sector ID number on each memory instruction (e.g., loads and stores), which is encoded in the otherwise unused top byte of the virtual address. Further details on the sector cache mechanism can be found in the A64FX micro-architectural manual [10] and the A64FX HPC extension specification [9].

The Fujitsu C/C++ Compiler (FCC) [11] provides compiler directives to specify the partitioning policy in application code for the A64FX processor. Although FCC's directives are limited to a maximum of two sectors, and the same partition sizes must be used on every core, this is nonetheless sufficient for our usage in this paper. Listing 1 shows an example of using the directives for an SpMV kernel with a matrix in CSR format. Line 1 specifies the number of cache ways allocated to sector 1. That is, `N2` ways are allocated for the L2 cache, and, optionally, `N1` ways for the L1 cache. The remaining cache ways are allocated to sector 0. Line 2 specifies the data objects (arrays or pointers) assigned to sector 1. In this case, the arrays `a` and `colidx` are assigned to sector 1, whereas other data is assigned to sector 0 by default.

The kernel in Listing 1 computes the product $y \leftarrow y + Ax$ for vectors x and y, and a matrix A, where the nonzero matrix values `a[i]`, are stored in row-major order and `colidx[i]` stores the column index of the `i`-th nonzero matrix value. The outer loop iterates over the matrix rows in parallel using the OpenMP worksharing-loop construct. The inner loop uses the row pointers (`rowptr`) to range over the nonzeros in each row. Nonzero matrix entries are multiplied by elements from the input vector `x` as determined by the column indices and accumulated into the output vector `y`. This approach to parallelising CSR SpMV is considered standard, although more sophisticated schemes, such as merge-based CSR SpMV [18], can in principle be used to mitigate workload imbalance for matrices where the number of nonzeros varies greatly between rows.

In the course of a single SpMV operation, only elements of the vectors and the row pointers can be reused. The matrix data, which consists of the nonzero matrix values and their column indices, are used only once. Even in the case of repeated SpMV operations, the working set is often too large to fit in cache anyway. Therefore, assigning the non-temporal matrix data (`a` and `colidx`) to a partition of minimal size increases the effective cache space of the reusable data in this code.

Listing 1: SpMV in CSR format using FCC's sector cache compiler directives.

```
1 #pragma procedure scache_isolate_way     L2=N2 [L1=N1]
2 #pragma procedure scache_isolate_assign  a colidx
3 #pragma omp for
4 for (int r = 0; r < num_rows; r++)
5     for (int64_t i = rowptr[r]; i < rowptr[r+1]; i++)
6         y[r] += a[i] * x[colidx[i]];
```

2.2 Reuse distance

Reuse distance [4, 15], or *stack distance*, is a hardware-independent metric for locality of reference of programs and has proven useful to analyze cache behavior. Once computed, it allows one to assess cache behavior for arbitrary cache sizes. This is an advantage over certain other techniques, such as cache simulation (see, e.g., [26]), which must be recomputed for each particular cache size.

Given a trace T in the form of a sequence of memory accesses $T = m_0, m_1, \ldots$, the reuse distance $\mathrm{RD}(r)$ of a reference m_r is the number of *unique* memory locations referenced between a pair of accesses to the same memory location, or ∞ if the location m_r has not been referenced before. In the following, we consider memory locations and cache size as given at the granularity of cache lines. For a fully associative Least Recently Used (LRU) cache with a capacity of n cache lines, a memory access results in a cache hit if its reuse distance does not exceed the cache size:

$$\mathrm{miss}(r, n) = \begin{cases} 1, & \mathrm{RD}(r) \geq n, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Furthermore, reuse distance with Eq. (1) remains a good approximation for set-associative caches with (pseudo)-LRU replacement, especially for caches with high associativity [4]. Although the cache replacement policies of the A64FX processor have not been fully disclosed, we assume that a pseudo-LRU policy is used.

Typically, reuse distances are obtained via profiling by instrumentation of memory instructions and processing the resulting memory trace. However, this involves a significant overhead, and, recently, more lightweight techniques [21] have been developed based on hardware event sampling and statistical methods.

In parallel programs, the cache behavior depends on the relative timing of threads. Their memory references can be interleaved to model the cache behavior of a shared cache with *concurrent reuse distance* [23], which is defined as the number of distinct references between a consecutive pair of references to the same memory location among all threads sharing the cache. Its value depends on the interleaving order and the number of threads sharing a cache.

3 METHODOLOGY

We propose a lightweight approach to estimate cache behavior of SpMV based on reuse distance without requiring instrumentation, while also including the effect of cache partitioning. Our method estimates reuse distance and cache misses based solely on the sparsity

pattern of the input matrix and its dimensions. We first present an analysis of situations where cache partitioning can be beneficial for SpMV, and thereafter explain our method for obtaining the reuse distance from the matrix sparsity pattern.

3.1 Cache partitioning for SpMV

To better understand how use of the sector cache impacts the CSR SpMV kernel in Listing 1, we consider a commonly encountered scenario where the SpMV operation $y \leftarrow y + Ax$ is performed repeatedly. The aim is to model the cache behavior after a warm-up iteration (i.e., no cold misses). The challenging part is estimating cache misses due to references to `x`, because their locality depends on the matrix sparsity pattern.

In the worst case, poor spatial and temporal locality may cause an entire cache line to be transferred for each access to the `x`-vector per nonzero. Because the cache line size of the A64FX is 256 bytes (instead of the typical 64 bytes), accesses to the `x`-vector may account for up to 95 % of the data traffic volume (i.e., 256 bytes for accessing the `x`-vector versus 12 bytes for accessing `a` and `colidx` per nonzero).

In any case, cold misses occur during the first SpMV iteration. The capacity misses in the next iterations are modeled based on the following classification of matrices according to their input dimensions:

(1) The matrix and vectors together fit into cache.
(2) The matrix and vectors together do not fit into cache, but `x`, `y` and `rowptr` together fit into a cache partition.
(3) `x`, `y` and `rowptr` together do not fit into a cache partition, while either
  (a) `x` completely fits into a cache partition, or
  (b) `x` does not fit into a cache partition.

Matrices in class (1) are expected to not benefit from cache partitioning, since there are no capacity misses in this case.

Matrices from class (2) are expected to benefit most from cache partitioning in iterative SpMV, because misses caused by accesses to `x`, `rowptr`, and `y` are avoided due to the partitioning. Capacity misses in this case are only due to `a` and `colidx`, which together yields $\lceil 8K/L \rceil + \lceil 4K/L \rceil$ misses for a cache with line size L, an M-by-N matrix with K nonzeros, and 8- and 4-byte values used for `a` and `colidx`, respectively. However, if cache partitioning is not used, then $\lceil 8(M+1)/L \rceil + \lceil 8M/L \rceil$ additional misses result from accesses to `rowptr` and `y` which both use 8-byte values. Furthermore, references to `x` with reuse distance larger than the cache size also cause additional capacity misses.

Finally, when the matrix dimensions are large enough such that `x`, `rowptr`, and `y` together do not fit into cache in case (3), only cache misses due to accesses to `x` can be avoided. Thus, even if cache partitioning is used, the same number of capacity misses due to `a`, `colidx`, `y` and `rowptr` are incurred as in the previous case, i.e., $\lceil 8K/L \rceil + \lceil 4K/L \rceil + \lceil 8(M+1)/L \rceil + \lceil 8M/L \rceil$. At this point, it may be better to additionally assign `rowptr` and `y` to the small partition, leaving more space for `x` in the other. If `x` fits completely into the other cache partition, then it incurs no capacity misses. Otherwise, isolating `x` lowers the reuse distance of references to `x`, potentially avoiding capacity misses.

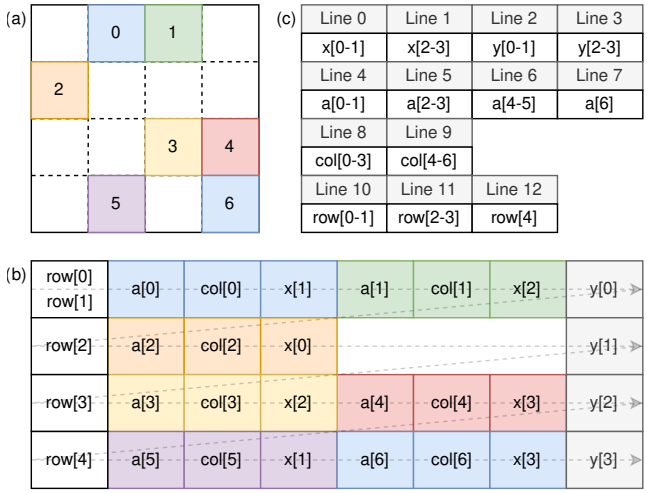

Figure 1: (a) Sparse matrix pattern with 7 nonzeros (b) Access pattern of CSR SpMV (c) Cache memory layout of involved data structures `x`, `y`, `a`, `colidx` and `rowptr` assuming a cache line size of 16 bytes and alignment to cache line boundaries.

We also note that in the case of a segmented shared cache, shared data may be replicated in the different segments. For example, the last-level cache of the A64FX consists of four 8 MiB segments, and the total cache space occupied by `x` may therefore vary between $8N$ and $4{\times}8N$, assuming an N-element array of 8-byte floating-point values.

3.2 Reuse distance analysis of SpMV with partitioned caches

In this section, we explain how to estimate cache misses occurring in SpMV from Listing 1, also in case of a partitioned cache. We also describe our method for computing reuse distances. Reuse distance allows us to model the total number of cache misses occurring in a program with a cache of capacity n by computing reuse distance for each memory reference r and using Eq. (1). This can be extended to a partitioned cache by treating the partitions as separate caches with capacities n_0 and n_1, such that $n_0 + n_1 = n$. The data is split between the two partitions, and reuse distances are computed with respect to the references assigned to each partition. In our considered case, the references to `a` and `colidx` are counted in partition 1, others are counted in partition 0:

$$\underbrace{\sum_{m_r \notin \{\texttt{a,colidx}\}} \mathrm{miss}(r, n_0)}_{\text{partition 0}} + \underbrace{\sum_{m_r \in \{\texttt{a,colidx}\}} \mathrm{miss}(r, n_1)}_{\text{partition 1}} \quad (2)$$

This equation corresponds to the partitioning policy from Listing 1. Disabling the cache partitioning is a special case of Eq. (2), where all references are counted in partition 0.

3.2.1 Computing reuse distances from the matrix sparsity pattern. In order to compute the reuse distances of memory references occurring in SpMV, we first obtain a memory trace. The trace is generated from the sparsity pattern of the matrix, which is used to

infer the memory access pattern that would be encountered during SpMV without running the SpMV kernel itself. In other words, cache line numbers are assigned to the elements in the involved data structures, and the matrix sparsity pattern is processed to deduce the memory locations that would be accessed when executing the SpMV kernel. This requires two passes for the case without and with partitioning: one pass where all references are accounted for in a single partition, and another pass where the references are divided into the partitions.

Fig. 1 shows an example of the approach described above. The access pattern of the SpMV with an example matrix (Fig. 1 (a)) is shown in Fig. 1 (b). The cache line numbers assigned to the elements of the data structures are shown in Fig. 1 (c). Each data structure is assumed to be aligned to a cache line boundary (i.e., 256 bytes for the A64FX). Finally, the reuse distance can be computed from the memory trace with a *stack processing algorithm*.

We use the stack processing algorithm by Kim et al. [13]. This algorithm has the property that, as opposed to other stack processing algorithms [2], the time complexity of computing reuse distance of each memory reference is independent of their locality within the memory trace. Note that, in principle, we could use another stack processing algorithm. We chose the algorithm by Kim et al. because of its constant time complexity per reference. We expect a more realistic interleaving of references in a trace when considering a shared cache in multi-threaded execution.

In the multi-threaded case, the memory trace is recorded in parallel using the same number of threads as would be used in the execution of the SpMV kernel. Each thread records the memory accesses of its assigned matrix rows. If a cache is shared by multiple threads, the memory accesses of these threads must be interleaved [23]. We achieve the interleaving using the queue-based MCS lock [17] to order and collate memory accesses submitted by different threads, because it provides starvation freedom and fairness (FIFO ordering).

3.2.2 Approximating reuse distance solely from column indices. The method described above needs two passes over the memory trace, one for the partitioned case and one for the case without cache partitioning. In this section, we describe an alternative method to approximate SpMV reuse distances for both cases from a single pass, only from the x-vector access pattern given by `colidx`. The influence of references to the other data structures is inferred from analytical considerations. Because the set of processed memory references is smaller, this method is much faster than processing the entire memory trace.

In case of a partitioned cache where only `x` is assigned to one partition, its reuse distances can be computed directly from the memory trace consisting only of references to `x`. However, we consider the case where `a` and `colidx` are assigned one partition and all the other data structures are assigned the second partition. Thus, `x` shares its cache partition with `rowptr` and `y`. This increases the reuse distances compared to assigning only `x` to a partition on average by a factor of $s_1 = (16 \times M/K + 8)/8$. In case of not using cache partitioning, this factor increases to $s_2 = (16 \times M/K + 20)/8$, because of the additional references to `a` and `colidx` (compare Fig. 1 (b)). These scaling factors are the ratio of the average number of bytes accessed per element of `x` and the data type size of `x`. Using these scaling factors, we can approximate the reuse distances of references to `x` for all three cases: (1) assigning `a` and `colidx` partition 0, (2) not using cache partitioning at all, and (3) assigning only `x` to partition 0 (not considered in this paper). Whether or not references to the other data structures hit in cache is determined based on the considerations from Section 3.1.

In the following, we refer to the approach discussed in Section 3.2.2 as *method (B)* and refer to the approach from Section 3.2.1 as *method (A)*. The advantage of using method (B) is that fewer memory references have to be processed and multiple cases are covered in a single stack processing pass. The disadvantage is a loss of accuracy, especially for matrices with a low average number and high coefficient of variation of nonzeros per row. We will discuss this in Section 4.5.

4 EVALUATION AND EXPERIMENTAL RESULTS

This section first describes our experimental setup and a performance comparison of our SpMV to prior work [1]. Afterwards, we discuss the impact of the sector cache on measured cache misses and performance in SpMV. Finally, we evaluate the our model by comparison to the cache miss measurements.

4.1 Experimental setup

The A64FX is a 48-core processor with private 64 KiB 4-way L1D caches grouped into four NUMA domains. Each NUMA domain has an 8 MiB 16-way last-level L2 cache, shared by 12 cores each, and is connected to an HBM2 module [10]. The theoretical peak memory bandwidth of the A64FX is 1024 GB/s, and a bandwidth utilization of over 800 GB/s can in practice be sustained [25].

The experiments in this study were carried out on a Fujitsu PrimeHPC FX1000 system on the Wisteria-BDEC01 cluster at The University of Tokyo. Our code was compiled using FCC 4.9.0 with the compiler options `-Kfast`, `-Kopenmp` and `-Kocl`.

Moreover, to pin each thread to its own core, we set the environment variables `OMP_PROC_BIND=close` and `OMP_PLACES=cores`. In addition, we perform NUMA-aware memory allocations by aligning allocations to a page and performing correct "first touch" initialisation of the data. Furthermore, huge pages are enabled by setting `XOS_MMM_L_HPAGE_TYPE=hugetlbfs` and a demand paging policy is enabled with `XOS_MMM_L_PAGING_POLICY=demand:demand:demand`.

To enable the sector cache features, we also set the environment variables `FLIB_HPCFUNC=TRUE`, `FLIB_SCCR_CNTL=TRUE` and `FLIB_L1_SCCR_CNTL=FALSE`.

Finally, a collection of 490 square, non-complex matrices from SuiteSparse [6] with more than 1 million and fewer than 1 billion nonzeros are used in the following experiments. The smallest matrix is about 11 MiB, which exceeds the size of a single 8 MiB segment of the L2 cache.

4.2 Performance of sparse matrix-vector multiplication on A64FX

First, we measure the performance of the SpMV kernel from Listing 1 on the A64FX processor using 48 threads without using the

Table 1: Performance (in Gflop/s) of CSR SpMV using 48 threads on A64FX.

Matrix	Rows	Non-zeros	Gflop/s Ours	Gflop/s [1]
pdb1HYS	0.036M	4.3M	82.9	40.2
Hamrle3	1.447M	5.5M	15.9	9.4
G3_circuit	1.585M	7.7M	10.8	11.2
shipsec1	0.141M	7.8M	94.0	16.7
pwtk	0.218M	11.5M	87.3	94.5
kkt_power	2.063M	14.6M	8.6	14.3
Si41Ge41H72	0.186M	15.0M	71.6	70.3
bundle_adj	0.513M	20.2M	7.6	66.6
msdoor	0.416M	20.2M	50.6	53.3
Fault_639	0.639M	28.6M	75.7	77.5
af_shell10	1.508M	52.7M	94.0	92.3
Serena	1.391M	64.5M	65.6	70.5
bone010	0.987M	71.7M	110.8	118.9
audikw_1	0.944M	77.7M	45.1	102.8
channel-500..	4.802M	85.4M	42.1	47.0
nlpkkt120	3.542M	96.8M	75.7	77.2
delaunay_n24	16.777M	100.6M	5.8	22.7
ML_Geer	1.504M	110.9M	117.8	120.5

sector cache. Table 1 shows the performance of matrices from SuiteSparse used by Alappat et al. [1] next to our results.

As expected, the performance depends highly on the sparsity pattern of the matrix, ranging from about 5 to 120 Gflop/s. These results agree with the performance reported by Alappat et al. [1], although the performance of a few matrices differs considerably and is therefore worth noting. In particular, for `kkt_power`, `bundle_adj`, `audikw_1` and `delaunay_n24`, we observe only 10–50 % of the performance reported by Alappat et al. However, in these cases Alappat et al. apply a combination of two performance optimizations not considered here, namely reordering using the Reverse Cuthill-McKee algorithm and a load balancing of the number of nonzeros per thread. On the other hand, we observe speedups of approximately 2 and 4 for the matrices `pdb1HYS` and `shipsec1`, respectively, which we found to be caused by using a demand paging policy for all three memory areas, whereas Alappat et al. instead used prepaging for the .bss area (i.e., `XOS_MMM_L_PAGING_POLICY` is set to `prepage:demand:demand`).

4.3 Impact of sector cache on memory traffic

Using the Performance Monitoring Units (PMUs) on the A64FX and the PAPI library [19], we measure the number of L1 and L2 cache misses incurred by the CSR SpMV kernel. The kernel is executed using 48 threads and PMU measurements are recorded per thread. The hardware event `L1D_CACHE_REFILL` is used to measure L1 cache misses, whereas L2 cache misses are measured by counting `L2D_CACHE_REFILL` and subtracting `L2D_SWAP_DM` and `L2D_CACHE_MIBMCH_PRF`, as described in [8, 10]. L2 demand misses, misses not caused by prefetching, are counted using the event `L2D_CACHE_REFILL_DM`. After measuring cache misses for the baseline without the sector cache, we thereafter enable the sector cache

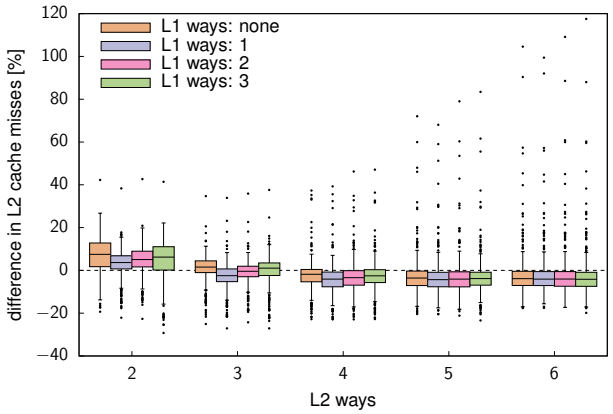

Figure 2: Distributions over 490 matrices of cache miss reduction or increase in SpMV for different sector cache configurations on Fujitsu A64FX. Lower and upper quartiles are indicated by the lower and upper ends of each box, whereas the median is shown as a horizontal line within a box. Whiskers indicate the interquartile range and outliers are plotted as individual points.

for the L2 cache and vary the number of L2 cache ways allocated to sector 1 (i.e., non-reusable data) from 2 to 6. This is combined with either disabling the L1 sector cache or allocating 1 to 3 L1 cache ways for non-reusable data. Fig. 2 shows the relative difference in L2 cache misses for different sector cache configurations compared to a baseline where the sector cache is disabled. Note that the impact of sector cache configurations may vary for different matrices. Therefore, a boxplot is presented for each cache configuration, thus showing the results as a distribution over all 490 matrices.

First, we note that configuration of the L2 sector cache is more important than the L1 sector cache. Second, we find that 4 or 5 L2 cache ways should in general be assigned to sector 1 (i.e., non-reusable data) to achieve a reduction in L2 cache misses for most matrices. The reduction in the total number of L2 cache misses is typically about 5 %.

Assigning more cache ways yields worse cache behavior for a small number of outliers. On the other hand, when using fewer cache ways, more cache misses are incurred for most matrices. This is surprising, since we expected using a minimum partition size (i.e., 2 L2 cache ways) for the non-temporal data would lead to best results. It turns out that a combination of a small sector and aggressive hardware prefetching of the non-reusable data (`a` and `colidx`) causes already prefetched data to be evicted prematurely. This effect was previously described by Alappat et al. [1]. We also confirmed the cause by reducing the prefetch distance of the A64FX hardware prefetcher, which can be adjusted in software on the A64FX using the *hardware prefetch assistance*, another part of the Fujitsu HPC extension [9]. After reducing the prefetch distance, we found that allocating only 2 L2 cache ways for sector 1 produced similar results to using 4 L2 cache ways.

We also found that the same effect occurs in the L1 cache using cache partitioning. We conclude that this is the reason why the L1 sector cache does not improve cache behavior. Nevertheless,

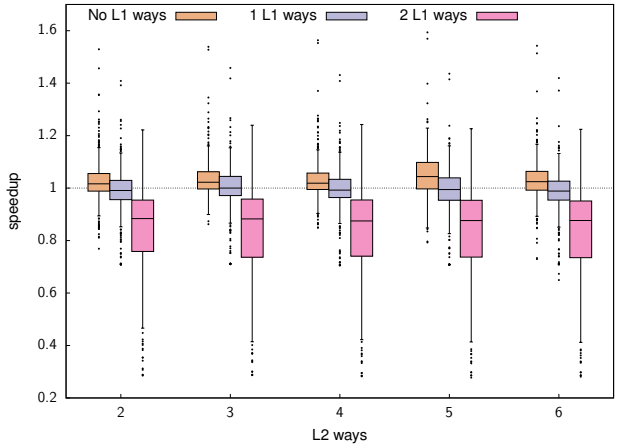

Figure 3: Speedup (or slowdown) of SpMV for different sector cache configurations on Fujitsu A64FX.

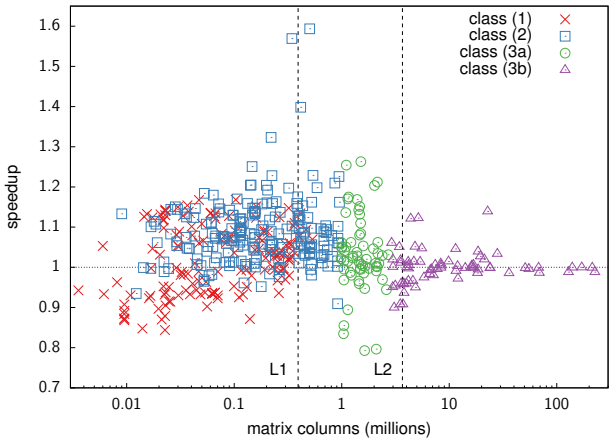

Figure 4: Speedup (or slowdown) versus vector size for SpMV for sector cache with 5 L2 ways. See Section 3.1 for a description of the different classes used to categorise matrices.

adjusting the prefetch distance of the L1 hardware prefetcher may alleviate this issue, enabling even further cache behavior improvements in SpMV using the L1 sector cache, but investigating this is left for future work.

4.4 Impact of sector cache on SpMV performance

Next, we assess the impact of various sector cache configurations on the performance of the SpMV kernel. We have already measured a baseline SpMV performance without sector cache. Thus, we now measure the performance again after enabling the sector cache for the L2 cache, while varying the number of L2 cache ways that are set aside for non-reusable data from 2 to 6. The L1 cache is either disabled or 1 to 2 L1 cache ways are allocated toward non-reusable data. These results are shown in Fig. 3 as distributions of speedups over the baseline (i.e., without sector cache) for all 490 matrices.

Generally speaking, enabling only the L2 sector cache and using 5 L2 cache ways for non-reusable data yields the best overall performance. In this situation, more than 75 % of matrices achieve performance equal to or better than the baseline, and 25 % of the matrices yield a performance improvement of about 10 % or more. The highest speedup attained is about 1.6×.

It was not beneficial to enable the L1 sector cache, and we found performance to degrade further with more L1 cache ways allocated to non-reusable data. This is already seen for 2 L1 cache ways, and we found that using 3 L1 cache ways for non-reusable data exacerbates the problem further resulting in slowdowns of 0.2× in the worst case. The explanation is the hardware prefetching and premature eviction of data as discussed in Section 4.3.

Fig. 4 shows the speedup versus the size of the x vector for the case of assigning 5 L2 ways to non-reusable data. Additionally, the figure distinguishes between the different matrix classes described in Section 3.1 (classes (1), (2), (3a) and (3b) are indicated by red crosses, blue squares, green circles and purple triangles, respectively). First, for matrices in class (1), matrix and vectors both fit in the L2 cache, and the performance is mostly within 5 % of the

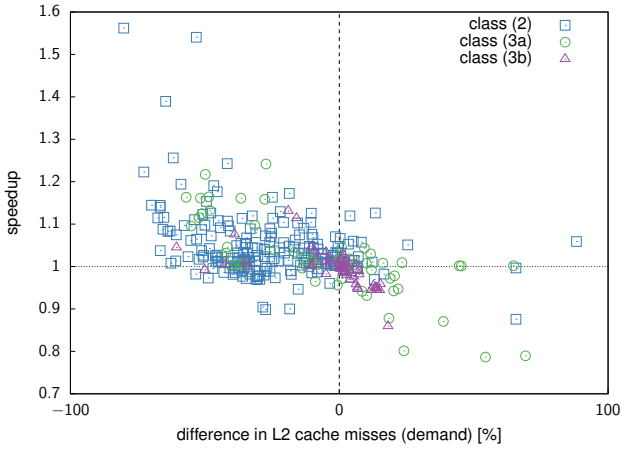

Figure 5: Speedup (or slowdown) versus difference in L2 demand misses of SpMV for sector cache with 5 L2 ways and working sets exceeding the L2 cache size. See Section 3.1 for a description of the different classes used to categorise matrices.

baseline. In some cases, however, the difference may be up to 20 %, possibly caused by the sector cache either increasing or reducing conflict misses. Second, when only the matrix no longer fits in L2 cache, i.e., class (2), then the sector cache almost always yields a performance improvement. The improvement is mostly up to 20 %, but this class also includes those cases where the speedup is highest, as expected. Finally, for matrices in class (3), the reusable data no longer fits in sector 0. The impact of the sector cache gradually lessens as the matrix dimensions increase and a smaller portion of the x vector fits in sector 0.

It is expected that the matrices experiencing higher speedup will suffer fewer demand cache misses after enabling the sector cache. These demand misses are associated with the irregular memory

accesses to the x-vector or conflict misses. Fig. 5 shows the relationship between SpMV speedup and the difference in L2 demand misses after enabling the sector cache with 5 L2 ways. In most cases, a speedup is accompanied by a reduction in L2 demand misses. A few cases experience a slowdown together with a reduction in L2 demand misses, in which case the performance may instead be limited by other factors, such as L1 cache traffic. Note that the matrices achieving the highest speedups of 1.2× or more also have a reduction of L2 demand misses of about 30–80%.

Our measurements show that only a few matrices approach the memory bandwidth limitation of about 800 GB/s on the A64FX, whereas the performance of many matrices is not directly limited by memory bandwidth. For example, the top 20 matrices in terms of memory bandwidth utilization range from 513 GB/s to 783 GB/s without the sector cache. However, none of the top 20 matrices in terms of speedup exceeds 400 GB/s bandwidth utilization. Their bandwidth utilization ranges from 74 GB/s to 376 GB/s without the sector cache, and even increases in most cases when the sector cache is enabled, even though the number of L2 cache misses is reduced. This indicates that other factors, such as the latency of handling demand misses, are limiting performance on the A64FX.

Memory bandwidth utilization can be measured with performance events on the A64FX using the following formula [8, 10]: Bandwidth [GB/s] = 10^{-9} × (L2D_CACHE_REFILL + L2D_CACHE_WB − L2D_SWAP_DM − L2D_CACHE_MIBMCH_PRF) × 256 / time

4.5 Accuracy of the cache miss model

To assess the accuracy of the cache miss model from Section 3.2, we use the Mean Absolute Percentage Error (MAPE),

$$\text{MAPE} = \frac{100}{N} \sum_{i=1}^{N} \left| \frac{x_i - \hat{x}_i}{x_i} \right|, \tag{3}$$

where x_i and $\hat{x}_i$ denote the measured and predicted number of L2 cache misses (see Eq. 2) for the i-th matrix, respectively.

Table 2 and Table 3 show the MAPE as well as the standard deviation of the absolute percentage error for the model's L2 cache miss prediction of method (A) and (B) in sequential and parallel iterative SpMV and varying cache partition sizes. We include only matrices above the L2 cache size (8 MiB sequential, 32 MiB parallel), since those are the cases where a high number of cache misses is expected. Otherwise, the MAPE is distorted by cases with few or no cache misses that are dominated by measurement noise, and thus becomes less meaningful.

4.5.1 Summary. In general, the accuracy of the cache miss predictions are mostly within about 5–10 %, though the accuracy depends on several factors and certain cases yield higher errors. Overall, the error using method (B) is just slightly worse compared to method (A) when the L2 sector cache is enabled. In this case, the error of method (A) and method (B) is about 2.5 % and 3 % in sequential SpMV and about 3 % and 4 % in the parallel case using a reasonable high number of cache ways for the non-temporal data, respectively. Method (B) has a high error predicting cache misses without cache partitioning compared to method (A).

The average overhead (t_A/t_B) of computing reuse distance with method (A) compared to method (B) is 4.21× and and 3.02× for 1 and 48 threads respectively. The average run-time of method (B) is 6.54s and 9.22s.

Table 2: Mean and standard deviation of the absolute percentage error for predicting L2 cache misses in sequential SpMV.

	method (A)		method (B)	
L2 Sector Cache	Mean	Std	Mean	Std
No Sector Cache	2.48 %	4.00 %	6.47 %	15.98 %
2 L2 ways	2.69 %	6.06 %	2.72 %	5.23 %
3 L2 ways	1.54 %	3.32 %	2.28 %	5.23 %
4 L2 ways	2.71 %	5.42 %	2.89 %	5.24 %
5 L2 ways	2.49 %	4.48 %	2.95 %	5.07 %
6 L2 ways	2.51 %	4.17 %	3.14 %	5.51 %
7 L2 ways	2.72 %	4.52 %	3.54 %	6.31 %

4.5.2 Accuracy for sequential SpMV. The model predicts the number of L2 cache misses in SpMV to within an accuracy of a few percent, both with and without the sector cache and for various partition sizes. Method (A) and method (B) perform similarly for predicting cache misses using cache partitioning. However, method (B) performs significantly worse without cache partitioning compared to method (A). This is caused by (1) the higher required scaling factor without cache partitioning and (2) due to a low average number (μ_K) and high coefficient of variation ($CV_K = \sigma_K/\mu_K$) of nonzeros per row. Note the relatively high standard deviation of the absolute percentage error, caused by few outliers with very high prediction error. Considering only the 254 matrices with $\mu_K \geq 8$ and $\sigma_K \leq 1$, the MAPE and standard deviation reduces to 3.25 % and 2.34 % for method (B) without partitioning.

4.5.3 Accuracy for parallel SpMV. For partition sizes of 4 L2 ways or more, the error using 48 threads is low and similar to the error in the sequential case. These results justify our approach for computing concurrent reuse distance as described in Section 3.2. The accuracy is worse for small partitions in the parallel case, because setting a too small partition can lead to the eviction of already prefetched cache lines before their first use as discussed in Section 4.3. In the sequential case, this issue does not occur using the L2 sector cache, because the cache partition is not shared by multiple threads. The more threads, the more space is occupied by non-reusable data that is prefetched before use.

Again, method (A) and method (B) perform similarly in the partitioned case, and method (B) performs significantly worse without partitioning compared to method (A). Considering only matrices with $\mu_K \geq 8$ and $\sigma_K \leq 1$, the MAPE and standard deviation is 7.02 % and 22.67 % respectively for method (B) without partitioning.

4.5.4 Accuracy of L1 cache miss prediction. The MAPE of method (A) and method (B) of L1 cache miss predictions is 8.40 % and 15.27 % in the sequential case without partitioning, and 8.91 % and 13.66 % in the parallel case, respectively. A high MAPE is expected on the A64FX, considering the low associativity of its L1 caches.

4.5.5 Discussion. Cache misses due to limited associativity and hardware prefetching are not taken into account with our cache miss model. Additionally, the model assumes a LRU policy, and an

Table 3: Mean and standard deviation of the absolute percentage error for predicting L2 cache misses in parallel SpMV using 48 threads (matrices > L2 cache).

	method (A)		method (B)	
L2 Sector Cache	Mean	Std	Mean	Std
No Sector Cache	3.47 %	4.34 %	10.80 %	29.04 %
2 L2 ways	15.11 %	6.46 %	15.79 %	9.85 %
3 L2 ways	8.69 %	4.70 %	9.68 %	8.98 %
4 L2 ways	4.79 %	3.61 %	5.60 %	7.39 %
5 L2 ways	3.14 %	3.65 %	4.03 %	6.71 %
6 L2 ways	2.56 %	3.35 %	3.70 %	6.74 %
7 L2 ways	2.63 %	3.96 %	4.07 %	7.75 %

interleaving of memory references that may not accurately represent actual parallel executions is performed. These factors generally limit the accuracy of modelling cache behavior with reuse distance.

For many matrices of class (2) and (3) in our data set, the majority of cache misses is caused by the easy-to-predict traffic due to streaming the matrix data. However, some matrices show poor spatial and temporal locality in the x-vector due to their irregularity, which means that the x-vector traffic constitutes a substantial part of the overall traffic. Thus, using the MAPE to aggregate results over the entire set of matrices may not accurately reflect the model's ability to predict the difficult cases.

However, for the case of sequential SpMV without the sector cache, the model using method (A) indicates that there are 42 out of 490 matrices where the x-vector causes 50 % or more of the overall traffic. Using only these hard-to-predict matrices, the MAPE for the L2 cache misses with and without using the sector cache is 8.14 % and 10.14 % respectively.

An alternative way to evaluate the model's accuracy would be to ignore regular data traffic from the matrix, and to only compare the model's prediction of cache misses due to x-vector accesses with appropriate performance event measurements for the related cache misses. Such performance event measurements require differentiating between cache misses caused by accesses to the x-vector and others, which may be feasible using event-based sampling.

5 RELATED WORK

Our paper is related to prior work on the following topics: performance analysis and optimization on the A64FX, SpMV, cache partitioning, cache miss models, and reuse distance analysis.

Earlier studies on the A64FX have focused on benchmarking and comparison to x86 [3] or other ARM CPUs [12], or in-depth comparison of the Fujitsu C compiler to LLVM and GNU compilers [7] for the A64FX.

Alappat et al. [1] apply the sector cache to dense matrix-vector multiplication on A64FX and observe a speedup of 2.8× for vector sizes in the range 2–6 MB. They also consider a CSR SpMV kernel, finding that the sector cache mostly provided no benefit at all, apart from only minor improvements of up to about 10 % in rare cases. Our study, on the other hand, finds that such an improvement of about 10 % is fairly common and can be expected for about half of the 490 matrices we selected from SuiteSparse. In contrast to the authors, we find that allotting 5 instead of 4 L2 cache ways to the matrix data improves performance most in general. Additionally, we do not find that using the L1 sector cache further improves cache behavior without considering hardware prefetch distances. Alappat et al. also consider other sparse storage formats, including SELL-C-σ [1] which achieved better performance on the A64FX than CSR. However, the authors did not investigate using the sector cache with this storage format.

One of the first works to model shared cache behavior of multithreaded SpMV was done by Song et al. [24]. The authors use a concept similar to reuse distance computed from per-thread memory traces, and statistically infer the number of cache misses of co-running threads. The average error for their considered two sparse matrices is 2.41 % compared to the simulation of a fully associative shared cache using two co-running threads. Lu et al. developed a reuse distance- and profiling-based framework [14] that performs automatic software cache partitioning using a technique called *page coloring* for serial programs. The authors improved SpMV performance in a conjugate gradient benchmark, reducing cache misses up to 35 % with cache partitioning. Breiter et al. [5], developed a profiling tool based on dynamic binary instrumentation, recognizing code regions where the sector cache should be applied. While cache miss reductions of 15–45 % were measured for benchmark problems involving dense matrices or structured grids, a sparse, conjugate gradient benchmark showed marginal reductions in L2 cache misses of about 1 %.

6 CONCLUSION

In this paper we have analyzed and measured cache behavior and performance of CSR SpMV on the A64FX processor using a variety of realistic sparse matrices. Additionally, we have studied how to use the sector cache feature of the A64FX to reduce cache pollution and improve performance for this important sparse kernel. We classified matrices based on their dimensions, which provides additional insight about when using the sector cache is beneficial. Detailed measurements of cache performance events show the high correlation between a reduction in demand cache misses and speedup in SpMV due to using the sector cache. The matrix showing most benefit has a speedup of about 1.6× when the number of demand misses was reduced by about 80 %.

To gain even further insight, we proposed a method for reuse distance analysis to provide accurate predictions for cache misses resulting from irregular memory traffic that also includes the effect of cache partitioning. The average prediction error of L2 cache misses is around 2.5 % and 3 % for sequential and parallel SpMV using 48 threads, respectively. Beyond explaining how to use the A64FX's sector cache for CSR SpMV, our proposed method for SpMV cache miss estimation can be extended to other kernels or other hardware architectures with different memory hierarchies. We therefore think our approach is useful, for example in a co-design process to determine optimized cache sizes, or to decide whether to integrate a cache partitioning mechanism such as the sector cache in future computing systems.

As future work, we plan to investigate the impact on demand misses and performance of tuning the hardware prefetch distance and using software prefetching in conjunction with the sector cache

for SpMV kernels on the A64FX. Additionally, it is worth investigating how the sector cache can be applied in the case of other sparse matrix storage formats or SpMV kernels, which can potentially offer better performance on A64FX.

ACKNOWLEDGMENTS

This work was supported by the EuroHPC Joint Undertaking grant agreement 956213 (SparCity), and the Federal Ministry of Education and Research of Germany (project number 16HPC045). The research presented in this paper has benefited from the Wisteria/BDEC-01 supercomputer at the University of Tokyo through JHPCN Joint Research Project jh230041. This research has also made use of the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053, and systems in the test environment BEAST (Bavarian Energy Architecture & Software Testbed)[1] at the Leibniz Supercomputing Centre. The authors would like to thank Christie Alappat for providing data related to the performance of SpMV with the CSR storage format.

REFERENCES

[1] Christie Alappat, Nils Meyer, Jan Laukemann, Thomas Gruber, Georg Hager, Gerhard Wellein, and Tilo Wettig. 2022. Execution-Cache-Memory modeling and performance tuning of sparse matrix-vector multiplication and Lattice quantum chromodynamics on A64FX. *Concurrency and Computation: Practice and Experience* 34, 20 (2022), e6512.

[2] George Almási, Călin Caşcaval, and David A Padua. 2002. Calculating stack distances efficiently. In *Proceedings of the 2002 workshop on Memory system performance*. 37–43.

[3] Fabio Banchelli, Kilian Peiro, Guillem Ramirez-Gargallo, Joan Vinyals, David Vicente, Marta Garcia-Gasulla, and Filippo Mantovani. 2021. Cluster of emerging technology: evaluation of a production HPC system based on A64FX. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, Los Alamitos, CA, USA, 741–750. https://doi.org/10.1109/Cluster48925.2021.00110

[4] Kristof Beyls and Erik D'Hollander. 2001. Reuse Distance as a Metric for Cache Behavior. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*. ISCA, Winona, MN, USA, 617–622.

[5] Sergej Breiter, Josef Weidendorfer, Minh Thanh Chung, and Karl Fürlinger. 2023. A Profiling-Based Approach to Cache Partitioning of Program Data. In *Parallel and Distributed Computing, Applications and Technologies*, Hiroyuki Takizawa, Hong Shen, Toshihiro Hanawa, Jong Hyuk Park, Hui Tian, and Ryusuke Egawa (Eds.). Springer Nature Switzerland, Cham, 453–463.

[6] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[7] J. Domke. 2021. A64FX – Your Compiler You Must Decide!. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, Los Alamitos, CA, USA, 736–740. https://doi.org/10.1109/Cluster48925.2021.00109

[8] Fujitsu Limited 2020. *A64FX PMU Events Errata* (version 1.0 ed.). Fujitsu Limited. https://github.com/fujitsu/A64FX/blob/master/doc/

[9] Fujitsu Limited 2020. *A64FX specification Fujitsu HPC extension* (version 1 ed.). Fujitsu Limited. https://github.com/fujitsu/A64FX/blob/master/doc/

[10] Fujitsu Limited 2022. *A64FX Microarchitecture Manual* (version 1.8.1 ed.). Fujitsu Limited. https://github.com/fujitsu/A64FX/blob/master/doc/

[11] Fujitsu Limited 2023. *FUJITSU Software Compiler Package C User's Guide* (version 1.0l21 ed.). Fujitsu Limited. https://software.fujitsu.com/jp/manual/manualindex/p22000026e.html

[12] A. Jackson, M. Weiland, N. Brown, A. Turner, and M. Parsons. 2020. Investigating Applications on the A64FX. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, Los Alamitos, CA, USA, 549–558. https://doi.org/10.1109/CLUSTER49012.2020.00078

[13] Yul H Kim et al. 1991. Implementing Stack Simulation for Highly-Associative Memories. *SIGMETRICS Perform. Eval. Rev.* 19, 1 (1991), 212–213. https://doi.org/10.1145/107972.107995

[14] Qingda Lu, Jiang Lin, et al. 2009. Soft-OLP: Improving Hardware Cache Performance through Software-Controlled Object-Level Partitioning. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 246–257. https://doi.org/10.1109/PACT.2009.35

[15] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems journal* 9, 2 (1970), 78–117.

[16] Simon McIntosh-Smith, James Price, Andrei Poenaru, and Tom Deakin. 2020. Benchmarking the first generation of production quality Arm-based supercomputers. *Concurrency and Computation: Practice and Experience* 32, 20 (2020), e5569.

[17] John M. Mellor-Crummey and Michael L. Scott. 1991. Synchronization without Contention. *SIGPLAN Not.* 26, 4 (1991), 269–278. https://doi.org/10.1145/106973.106999

[18] Duane Merrill and Michael Garland. 2016. Merge-Based Sparse Matrix-Vector Multiplication (SpMV) Using the CSR Storage Format. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, USA, Article 43, 2 pages. https://doi.org/10.1145/2851141.2851190

[19] Philip J Mucci, Shirley Browne, et al. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, Vol. 710.

[20] Tetsuya Odajima, Yuetsu Kodama, Miwako Tsuji, Motohiko Matsuda, Yutaka Maruyama, and Mitsuhisa Sato. 2020. Preliminary Performance Evaluation of the Fujitsu A64FX Using HPC Applications. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 523–530. https://doi.org/10.1109/CLUSTER49012.2020.00075

[21] Muhammad Aditya Sasongko, Milind Chabbi, et al. 2021. ReuseTracker: Fast Yet Accurate Multicore Reuse Distance Analyzer. *ACM Trans. Archit. Code Optim.* 19, 1 (2021), 1–25. https://doi.org/10.1145/3484199

[22] Mitsuhisa Sato, Yuetsu Kodama, Miwako Tsuji, and Tesuya Odajima. 2022. Co-Design and System for the Supercomputer "Fugaku". *IEEE Micro* 42, 2 (2022), 26–34. https://doi.org/10.1109/MM.2021.3136882

[23] Derek L. Schuff, Benjamin S. Parsons, and Vijay S. Pai. 2010. Multicore-aware reuse distance analysis. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–8. https://doi.org/10.1109/IPDPSW.2010.5470780

[24] Fengguang Song, Shirley Moore, and Jack Dongarra. 2007. L2 Cache Modeling for Scientific Applications on Chip Multi-Processors. In *2007 International Conference on Parallel Processing (ICPP 2007)*. IEEE Computer Society, Los Alamitos, CA, USA, 51–51. https://doi.org/10.1109/ICPP.2007.52

[25] Sarat Sreepathi and Mark Taylor. 2021. Early Evaluation of Fugaku A64FX Architecture Using Climate Workloads. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 719–727. https://doi.org/10.1109/Cluster48925.2021.00107

[26] James D Trotter, Johannes Langguth, and Xing Cai. 2020. Cache simulation for irregular memory traffic on multi-core CPUs: Case study on performance models for sparse matrix–vector multiplication. *J. Parallel and Distrib. Comput.* 144 (2020), 189–205.

[1] https://www.lrz.de/presse/ereignisse/2020-11-06_BEAST/