Planar: A Programmable Accelerator for Near-Memory Data Rearrangement

Adrián Barredo
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain
adrian.barredo@bsc.es

Adrià Armejach
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain
adria.armejach@bsc.es

Jonathan C. Beard
Arm Research
Austin, Texas, USA
jonathan.beard@arm.com

Miquel Moretó
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain
miquel.moreto@bsc.es

ABSTRACT

Many applications employ irregular and sparse memory accesses that cannot take advantage of existing cache hierarchies in high performance processors. To solve this problem, Data Layout Transformation (DLT) techniques rearrange sparse data into a dense representation, improving locality and cache utilization. However, prior proposals in this space fail to provide a design that (i) scales with multi-core systems, (ii) hides rearrangement latency, and (iii) provides the necessary interfaces to ease programmability.

In this work we present Planar, a programmable near-memory accelerator that rearranges sparse data into dense. By placing Planar devices at the memory controller level we enable a design that scales well with multi-core systems, hides operation latency by performing non-blocking fine-grain data rearrangements, and eases programmability by supporting virtual memory and conventional memory allocation mechanisms. Our evaluation shows that Planar leads to significant reductions in data movement and dynamic energy, providing an average 4.58× speedup.

CCS CONCEPTS

• **Hardware → Memory and dense storage**; • **Computer systems organization → Multicore architectures**; • **General and reference → Performance**; • **Computing methodologies → Vector / streaming algorithms**.

KEYWORDS

Data layout transformation, Sparse data, Near-memory accelerator

The final publication is available at ACM via
http://dx.doi.org/10.1145/3447818.3460368

Table 1: Comparison with state-of-the-art DLT proposals.

Features	Impulse [12]	DLT Acc. [27]	SPiDRE [7]	DRE [39]	Planar
Full design	✓	✓	✗	✓	✓
Scalable design	✗	✓	✓	✗	✓
Non-blocking DLT	✓	✗	✓	✗	✓
Fine-grain sync.	✓	✗	✗	✗	✓
VM support	✓	✓	✓	✗	✓
Normal allocator	✗	✓	✓	✗	✓

ACM Reference Format:
Adrián Barredo, Adrià Armejach, Jonathan C. Beard, and Miquel Moretó. 2021. Planar: A Programmable Accelerator for Near-Memory Data Rearrangement. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3447818.3460368

1 INTRODUCTION

Memory latencies have not experienced the near-exponential improvements seen in processing speed and memory capacity [20, 21]. As a result, data access times increasingly limit system performance, a phenomenon known as the Memory Wall [60]. Deep cache hierarchies are the natural solution to this trend, providing low-latency data access to high-performance out-of-order processing units. Applications that have locality of reference benefit from cache hierarchies [51, 54], while prefetchers act in the background to hide memory access latency [44].

In the presence of sparsity and irregular reuse distances, studies show that data prefetching is not effective [66], utilization of transmitted bandwidth can be as low as 20% [7], and that most blocks in the last level cache are not reused before eviction [8, 52]. In addition, for applications with dependent or indirect access loads, every cache level increases the overall round-trip access latency [18]. Finally, irregular and sparse patterns preclude harnessing data-level parallelism via vector instructions that operate on multiple data values (SIMD) [40, 53, 55]. Data movement not only affects performance: approximately two-thirds of the energy required to compute is consumed by data movement, specifically by the memory and interconnect [11].

Data Layout Transformation (DLT) mechanisms have been proposed to tackle these problems. DLT aims to rearrange sparse data into a dense representation to improve locality and make better use of the memory hierarchy. Table 1 qualitatively compares multiple state-of-the-art proposals. A balanced design should fulfill three principles. First, a comprehensive design that scales well with multi-core systems by carefully choosing where rearrangements occur. Second, maximize system performance by providing non-blocking fine-grain rearrangements to hide DLT latency. Third, ease programmability for the DLT engine and target applications by providing virtual memory (VM) support and conventional memory allocation mechanisms. Previous proposals make compromises on these design principles hindering their adoption.

In this paper we present a *ProgrammabLe Accelerator for Near-memory datA Rearrangement (PLANAR)*. PLANAR is located within the system-on-chip at the same level as the memory controllers, avoiding custom off-chip memory modifications that are difficult to adopt. Our design is non-blocking as it decouples access and execute, allowing overlap of data rearrangements and *host* core computation. In addition, we provide mechanisms for fine-grain synchronization between PLANAR and *host* cores to allow dense data to be consumed as it is rearranged, hiding rearrangement latency. PLANAR is programmable via simple library calls that can be inserted by a programmer or by a compiler pass. This simple programming interface is possible due to the fact that PLANAR has virtual memory support and employs well-known existing memory management mechanisms for the new dense data structures.

Moreover, PLANAR enables applications to take better advantage of the memory hierarchy by exploiting locality of dense data, and unlocks additional performance due to better prefetching and vectorization. On the latter, PLANAR allows compilers to optimize instruction emission for contiguous memory [57], which is critical to vector performance [45].
This paper makes the following contributions:

- We introduce minimal functional changes to incorporate PLANAR into a system-on-chip with out-of-order cores. By locating PLANAR devices at the memory controller level we enable the design to (i) scale with multi-core systems, (ii) perform fine-grain non-blocking data rearrangements, (iii) operate with virtual memory support, and (iv) be off-chip memory technology agnostic. No solution in the state-of-the-art provides all such properties.
- A detailed evaluation using a full-system cycle-accurate simulator shows that a multi-core system with PLANAR achieves an average speedup of 4.58× across a wide range of applications featuring sparse and irregular access patterns. This performance improvement is due to PLANAR reducing L1-D cache misses by an average of 89% and L1-D cache miss latency by an average of 53%. Overall, dynamic energy consumption is reduced by more than 40%. PLANAR also enables additional vectorization of rearranged codes, increasing the average speedup to 5.71×.
- We show that PLANAR outperforms software DLT techniques in Section 2 and two state-of-the-art hardware proposals, Impulse [12] and a DLT accelerator [27], in Section 5. Our comparison shows that, on average, PLANAR outperforms Impulse by 2.12× and the DLT accelerator by 2.23×. Thanks to non-blocking fine-grain rearrangements, PLANAR can hide DLT latency, allowing the *host* to consume dense data as it is rearranged.

```
1  void stride_kernel(double *x, int *idx, ...){
2      ....
3      for (len = 0; i < len; len++) {
4          v1s1m3(); v1s2m3(); v1s3m3(); v2s2m3(); v2s2m4();
5          v1s1i3(x, idx);
6      }
7  }
8  void v1s1i3(double *x, int *idx, ... ){
9      ....
10     for( j = 0; j < irep; j++ ) {
11         t1 = 1.0/(double)(j+1);
12         for( i = 0; i < n; i++ )
13             y[...] += t1*x[idx[i]]; //irregular accesses
14     }
15 }
```

Figure 1: Original STRIDE code.

```
1  void stride_kernel(double *x, int *idx, ...){
2      ....
3      for (len = 0; i < len; len++) {
4          v1s1m3(); v1s2m3(); v1s3m3(); v2s2m3(); v2s2m4();
5          v1s1i3_sw_rearr(x, idx);
6      }
7  }
8  void v1s1i3_sw_rearr(double *x, int *idx, ... ){
9      ....
10     x_rearr = malloc(size);
11     for( i = 0; i < n; i++ )
12         x_rearr[i] = x[idx[i]]; //software rearrangement
13
14     for( j = 0; j < irep; j++ ) {
15         t1 = 1.0/(double)(j+1);
16         for( i = 0; i < n; i++ )
17             y[...] += t1*x_rearr[i]; //regular accesses
18     }
19     free(x_rearr);
20 }
```

Figure 2: Software-rearranged STRIDE code.

2 MOTIVATION

To explain the limitations of DLT techniques in software, and the advantages of performing DLT with PLANAR, we have chosen a representative case study based on the STRIDE benchmark [1]. STRIDE is a memory intensive benchmark that consists of a loop where every iteration executes six different kernels. In the original code, *v1s1i3* is the kernel with sparse memory accesses (see lines 8-15 in Figure 1). The memory access pattern is governed by the *idx* array which is populated with a configurable input stride[2].

The programmer could decide to replace the indirect memory accesses from *x* with sequential ones in an *x_rearr* array using a software DLT solution, as shown in Figure 2. This extra code should be placed just before the original loop in *v1s1i3* (see lines 10-12 in Figure 2). This software rearrangement is beneficial as *x* is accessed *irep* times in the baseline with strided accesses, and only once in this new version. As a result, execution time improves 22.1% on average for different stride values in the indirection vector *idx*.

In this paper we present PLANAR, a hardware solution that performs near-memory data layout transformations. Figure 3 shows the pseudo-code of STRIDE compatible with PLANAR. The rearrange function (*offload* function in lines 1-7) performs the data layout transformation using the PLANAR devices. Several PLANAR devices can be allocated to do this transformation in parallel (line 12) and execute the rearrange function (line 13), extracting higher memory-level parallelism (MLP) than in the software rearrangement version. Finally, the PLANAR devices are released (line 6).

[1]Section 4 describes the benchmark in detail.
[2]Section 4 describes the strides employed in the evaluation.

```
1  void offload(double *x, int *idx, double *x_rearr, ...){
2      // Rearrange function executed on PLANAR
3      for( i = start_idx; i < end_idx; i++ )
4          x_rearr[i] = x[idx[i]];
5      // Release device if last element
6      planar_release();
7  }
8  void stride_kernel(double *x, int *idx, ...){
9      ....
10     for (len = 0; i < len; len++) {
11         x_rearr = malloc(size);
12         n_dev = planar_alloc(min, max);
13         offload«n_dev»(x, idx, x_rearr, size, ...);
14         v1s1m3(); v1s2m3(); v1s3m3(); v2s2m3(); v2s2m4();
15         v1s1i3_hw_rearr(x_rearr);
16         free(x_rearr);
17     }
18 }
19 void v1s1i3_hw_rearr(double *x_rearr, ... ){
20     ....
21     for( j = 0; j < irep; j++ ) {
22         t1 = 1.0/(double)(j+1);
23         for( i = 0; i < n; i++ )
24             y[...] += t1*x_rearr[i]; //regular accesses
25     }
26 }
```

Figure 3: Planar-rearranged STRIDE code.

This rearrangement can be done *ahead of time* while the *host* is operating on the first five kernels, thereby overlapping data rearrangements and *host* execution (see lines 14-15 in Figure 3). As a result, Planar effectively hides rearrangement latency Executing STRIDE with eight Planar devices provides average performance speedups of 2.77× and 3.39× over software-rearranged and the original versions, respectively.

Planar provides the required hardware support to enable fast data rearrangement near memory, converting sparse data to dense, resulting in a more efficient usage of the available bandwidth. This transformation is done while the *host* core performs useful computation, effectively decoupling access to memory and execution.

3 PLANAR DESIGN

Planar targets applications with irregular memory access patterns. In such applications, the memory subsystem is poorly utilized, leading to latency and bandwidth bottlenecks because of low cache block utilization [8] caused by disperse memory accesses that lead to high (but underutilized) traffic on data transfer networks (e.g., coherence bus, interconnects) [43].

Figure 4 shows a high level system overview with two Planar devices. Planar is implemented as a near-memory programmable accelerator connected to the main coherence bus with direct access to the memory controllers. Despite being programmable, it is a simple device that can be implemented as a microcontroller. It is comparable to an Arm Cortex *M*0+, with the addition of a 64-bit ALU and minimal support for data caching and address translation.

The design enables accesses from the cores to bypass the Planar units in normal operation, while allowing the Planar units to use the same memory controllers when commanded by the *host* core. In the figure, every core is augmented with a small *Rearrangement Control Table (RCT)* to monitor the status of ongoing transformations. The *RCT* has one entry per rearrangement in flight, within each *RCT* entry there is a slot for each Planar device and per-device sub-entries containing rearrangement progress information (three 64-bit entries to track virtual address range and status).

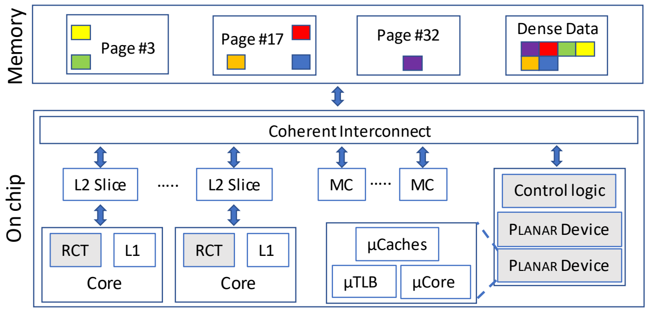

Figure 4: System overview with two Planar devices. Cores are augmented with a Rearrangement Control Table (*RCT*) to monitor ongoing rearrangements.

Planar creates a new structure whose elements are sorted the way they are to be accessed by the *host* core. This way, cache block and bandwidth utilization improves. The operation latency can be hidden if the rearrangement can start before the data is needed by the *host* core, and via fine-grain synchronization of rearranged data, overlapping rearrangement with computation. Multiple devices can apply the same rearrange function, or multiple rearrange functions can be done in parallel by different devices.

Figure 4 shows an example with two Planar devices. A *host* core has requested them to perform the near-memory data restructuring of the *sparse* elements in color from data pages 3, 17 and 32. The result is a dense version of the data placed into another data page. The *host* core may access this new dense structure via contiguous accesses instead of the original sparse accesses, reducing data movement and hiding latency. As an example, if the core only uses one 8B value from each of the cache blocks accessed (64B) the total payload needed would be 48B (six elements). In the original case, the core would have to access six different cache blocks (i.e., 384B). However, with Planar the reduced payload would be a single cache block (assuming they are aligned), i.e., 64B with 48B of the transfer actually utilized. This represents an 83% reduction in data movement for this simple case example. Moreover, dense data presents additional opportunities to improve performance: (i) simple next-block prefetching schemes are efficient, and (ii) data-level parallelism via vectorization can be exploited.

The following sections provide the operational details of Planar, including the required modifications at application level, the different phases involved in a rearrangement, and finally, a comprehensive step-by-step example of operation.

3.1 Modifications to Application Code

In our implementation, the programmer is responsible for providing a rearrange function to map the irregular data access to a dense data access. Most actions taken to offload to Planar units are handled either by the hardware or via library calls as shown in Figure 3.

- `planar_alloc()` takes a minimum and maximum number of devices to be allocated and returns the number of allocated devices. Planar devices are simple and can have just a few in-flight memory requests, so it is difficult for a single device to saturate memory bandwidth. Having several allocated devices leads to higher memory-level parallelism and to better utilization of the memory bandwidth.

- `offloadFunc<<<N>>>()` contains the code that is executed in Planar, including the rearrange function. The *N* parameter between triple chevrons determines the number of devices to offload to, and it is used to calculate the start and end bounds of the rearrange loop for each device.
- `planar_release()` signals the device to finish.

The information to produce a rearrange function is known at compile time although data is often dynamic (e.g., loop bounds), so a compiler supporting Planar may enable transparent rearrangements as suggested by prior works [30, 46].

3.2 Allocation of Memory and PLANAR

A memory region is allocated for the Planar devices (line 11 in Figure 3). It prevents the *host* core from accessing an outdated dense structure (i.e., from a previous rearrange task).

To allocate the devices, the `planar_alloc` function is used, as there can be multiple *host* cores planning to use these devices at the same time. In our proposal, the cores can access a list of the available Planar devices from a firmware table and dynamically choose a minimum and a maximum number of devices they want to use. Each device is accessed via a memory mapped work queue, which could be virtualized by the operating system using many existing mechanisms [56].

If there are not enough available Planar accelerators, the *host* core suspends until later notification is received when the minimum number of accelerators is available.

3.3 Offloading of Rearrange Function

When `offloadFunc<<<N>>>` is called, a command data packet is created for each of the *N* Planar devices. The packet consists of pointers to the sparse and final dense data, and the start address (virtual program counter) of the rearrange function. The boundaries of the dense data structure are used to split the workload among all the Planar devices (*N*) in charge of rearranging the same data structure. This data packet is the equivalent of two cache blocks of data (including a header of setup information for the *host* core). Approximately five cycles are needed to save this setup information. Subsequent transport of this setup information to the Planar devices is dependent on the topology of the interconnect and latency of cache write-back between the *host* and the Planar units. Further details about our configuration are given in Table 2. Once the command packet is sent, the *host* core updates the *RCT* entries of the allocated Planar devices.

Planar is by definition an accelerator. As such it must communicate results with the *host* core. To do so, it makes use of a common coherent interconnect. Planar will often work on shared data with the *host* core, meaning that modified data could exist within the *host* caches. In order to maintain memory consistency between the *host* core and Planar, flush operations are triggered from Planar before the rearrangement starts. To achieve this, after receiving the command packet, Planar issues cache maintenance commands [38] to flush the sparse data address range from caches. They are issued from a state machine co-located with the Planar device. Once it finishes, the data rearrangement can start.

3.4 Execution of Rearrange Functions

Every device has received its rearrange function, data pointers and work boundaries in the offloading phase. Therefore, in this phase every Planar device accesses the sparse data, performing the irregular memory accesses, and populating the dense data structure. It is worth noting that Planar is designed to have virtual memory support. This can be accomplished by connecting the Planar devices to an input-output memory management unit (IOMMU), which provides virtual-to-physical address translation for the direct memory accesses (DMA) that Planar performs. The operating system also keeps track of the pages being accessed by the Planar devices. Whenever a rearrangement is happening, the involved data blocks are available to the *host* core in shared state but read only. This way, memory consistency is ensured.

3.5 Fine-Grain Synchronization Between PLANAR and Host

Once a Planar device completely populates a set of cache blocks (number explored in Section 5.1) belonging to a dense structure, a cache maintenance operation is issued to flush the blocks from the cache, forcing a writeback to main memory.

A synchronization mechanism between Planar and the *host* core is needed to ensure the *host* only accesses data when it is ready. After the flush to a set of dense cache blocks is issued, a synchronization packet is created, containing the index of the last element in the last cache block. This packet is sent to the *host* core in order to update the corresponding *RCT* entry and to wake up the *host* in case it is suspended. The *RCT* keeps information for every rearrangement in flight from a *host* core, including the boundaries (virtual addresses) for every Planar device as well as the last rearranged element.

After the *RCT* is updated, a cache maintenance packet is sent to the core's cache to invalidate the set of cache blocks in case they are present in the caches. This is necessary as some hardware units, such as the prefetchers, may issue memory requests to these memory locations while Planar is operating, caching data that has not yet been rearranged. As explained in Section 3.4, data being rearranged is in read only state and it cannot be accessed by the *host* due to the *RCT*. For this reason, writebacks of these invalidated cache blocks are not needed. After invalidation, the *host* core or other hardware units can access a valid version of the dense data, located in main memory, as mediated by the *RCT*.

Whenever a load address is calculated in the execute stage of the *host* core, if the *RCT* contains valid entries it will be accessed and every virtual data range compared with the load address. If a match is found and the load address is part of the already rearranged region, the memory request can proceed. Otherwise, the load instruction is moved to a FIFO queue. The queue size is limited by the instruction window of the *host* core, since the *host* core will stall or suspend as it will not be able to proceed with the execution. Later Planar synchronization messages will notify the *host* core, which will check the FIFO queue, moving the instructions to the load-store queue as data becomes ready.

3.6 Release of PLANAR Devices and Memory

Planar devices are released via the `planar_release` call (Figure 3, line 6). The Planar device sends a packet to the *host* and suspends, becoming available for future operations. Once the *host* core receives the packet, the related *RCT* entry is cleared. Finally, after the dense data is consumed by the *host*, it is freed (Figure 3, line 16).

3.7 PLANAR Execution Example

Figure 5 shows a detailed example of operation with Planar. It considers a single rearrangement performed by one Planar device. It shows four main hardware components (top), the *host* core, the coherent interconnect, the Planar accelerator and a representation of several main memory pages. From the core, we show the view of the *RCT*, the data cache (D$), and the FIFO queue used to hold instructions that try to access data still not rearranged. From Planar, we show the logic that, amongst other things, is in charge of processing command packets from/to all the devices, and from the device itself, the core (μcore) and data cache ($\mu$$).

In *Phase #1* (Section 3.2), the dense structure is allocated ❶ via a *malloc* call to store the dense data. The `planar_alloc` function triggers the allocation of the devices. In the example, all the devices are free and one device is requested, therefore device *ID0* is reserved for the current process (*PID 33*), allocating entry 0 in the *RCT* ❷.

In *Phase #2* (Section 3.3), offloading begins by sending a command data packet ❶ from the core to Planar. This packet contains the information to program Planar, including the rearrange function and the loop bounds, which depend on the number of devices involved. After that, the core updates the corresponding *RCT* entry with the dense virtual address range and offset of rearranged elements ❷. When the control logic receives a command packet it uses a state machine to issue cache maintenance flush requests of the sparse data ❸. This ensures Planar devices will access the latest version from main memory. Finally, the control logic programs the Planar device to start the rearrangement ❹.

In *Phase #3* (Section 3.4), Planar starts executing the rearrangement, accessing the sparse data (@*A*) and writing to the dense data structure (@*A′*) ❶. Subsequent accesses will fill a cache block with dense data ❷ ❸ ❹. The device continues executing the rearrange function, filling dense cache blocks until the operation completes.

In *Phase #4* (Section 3.5), the synchronization phase ensures that the *host* core obtains the results produced by the Planar device in a timely manner, while preventing the core from accessing data that has not yet been rearranged. Note that actions in this phase can happen in parallel with *Phase #3* actions. Therefore, to achieve this fine-grain synchronization, in this example, for each dense cache block that is completely populated, a cache maintenance operation is issued to flush the block to the memory controller (MC)❶. This data is eventually written to main memory in the dense data page ❷. Additionally, a synchronization packet is sent to the core to notify a new dense block is available, updating the corresponding *RCT* entry offset fi eld❸. To prevent the core from accessing stale data, an invalidation is sent to the cache hierarchy ❹, ensuring the dense data will be fetched from main memory. Finally, the FIFO queue is checked for stalled instructions to the now available rearranged cache block ❺, which would be able to proceed. The other mechanism present in this phase is triggered when the *host* core issues a load and there are in-flight rearrangements ❻. The *RCT* table is checked to see if the virtual target address conflicts with an in-flight range for the executing process. If that is the case the current offset determines if the rearranged data is ready to be consumed. If that is not the case, the load instruction is stalled and placed into the FIFO queue ❽. Eventually, the target address of the load instruction will be rearranged and the FIFO checked, allowing it to execute.

In *Phase #5* (Section 3.6), when the dense structure has been fully populated ❶, Planar is released. A packet is sent to the *host* to indicate the operation has completed ❷, which clears the pertinent *RCT* entry ❸. In addition, the accelerator control logic is notified ❹ and the device suspends. Once dense data is consumed, it is freed.

Table 2: *gem5* simulation parameters.

Cores	8 single-threaded out-of-order cores, 2GHz
Core details	
Fetch, decode, rename width	4 insts/cycle
Dispatch, issue, commit width	8 insts/cycle
Branch target buffer	1 way, 2048 entries
Branch predictor	Bimode, 8K+8K entries, RAS 16 entries
Load and store queues	48 entries, 48 entries
Physical registers	256 integer + 256 floating point
Issue queue, re-order buffer	92 entries, 192 entries
Functional units	3 Int ALU + 2 FP/SIMD ALU
Instruction latencies (int)	add (1c.), mul (3c.), div (12c.)
Instruction latencies (FP)	add (5c.), mul (4c.), div (9c.)
L1 instruction cache	48KB, 3-way, 64B/block, 1 cycle access lat.
L1 data cache	32KB, 2-way, 64B/block, 2 cycle access lat.
L2 banked unified cache	2MB, 16-way, 64B/block, 12 cycle access lat.
Prefetcher	Stride prefetcher
Memory details	
Type	DDR4 2400
Channel	2 channels, 16GB/s per channel
Planar details	
Number of devices	8
μCore	in-order core, single-threaded, 2GHz
Functional units	1 Int ALU
Instruction latencies (Int)	add (3c.), mul (3c.), div (9c.)
L1 instruction μcache	1KB, 2-way, 64B/block, 1 cycle access lat.
L1 data μcache	1KB, 2-way, 64B/block, 2 cycle access lat.
Translation lookaside buffer (μTLB)	8 entries

4 METHODOLOGY

4.1 Full-System Simulation Infrastructure

We employ *gem5* [9] to simulate an Arm 64-bit architecture in detail with a full-system environment. The simulated system runs Ubuntu 18.10 with Linux kernel v4.15. We simulate an eight out-of-order core processor using the detailed CPU and memory models of *gem5*, extended with the micro-architectural support for Planar. Planar is modelled as a simple in-order core connected to the memory bus. Section 5 presents a design space exploration to size the Planar hardware. Table 2 shows the architectural parameters used.

As explained in Section 3, the offloading and synchronization phases require flush and invalidation mechanism to maintain cache consistency. In addition, command and synchronization packets to communicate with Planar and update the RCT in the *host* core are needed. These functionalities have been added to gem5 by means of ISA extensions and all operation latencies are modelled in detail.

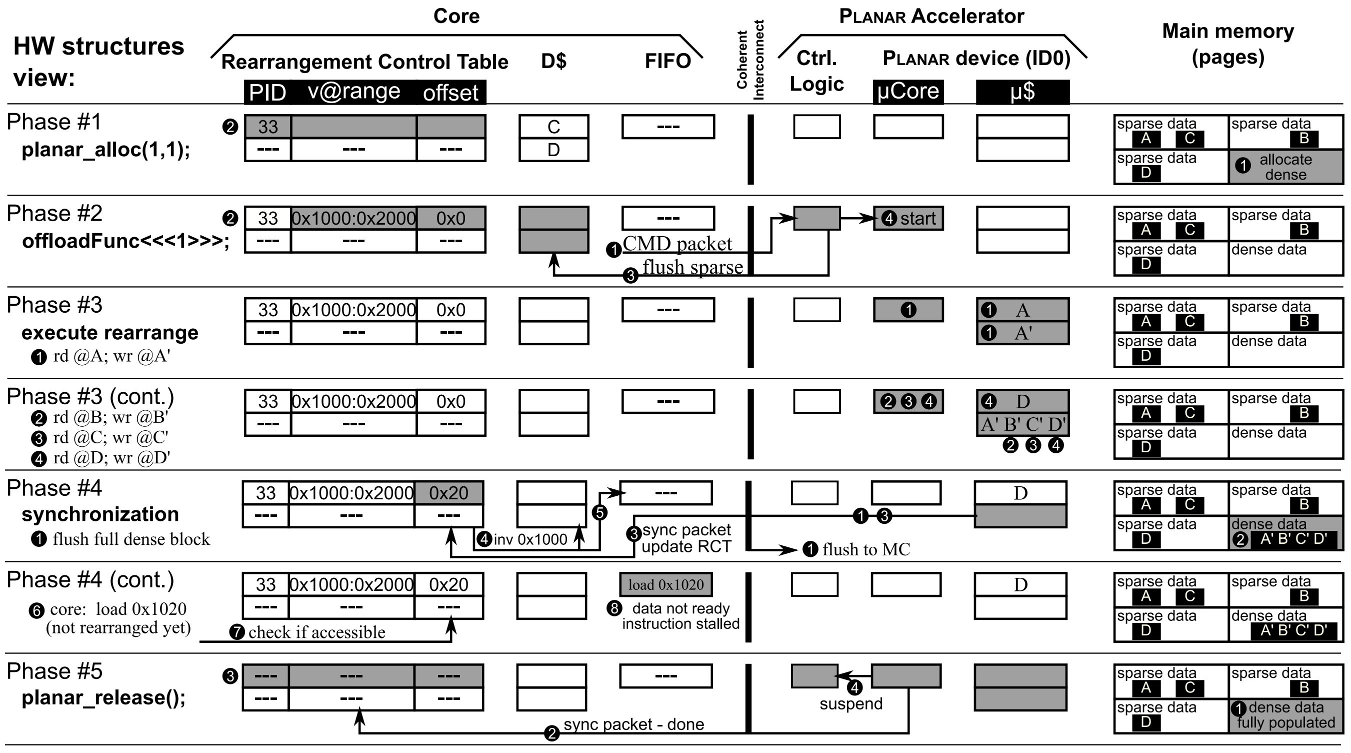

Figure 5: Execution example for a rearrangement that employs one Planar device.

4.2 Benchmarks

The evaluation considers a set of representative applications. Table 3 summarizes the employed benchmarks with their parameters and inputs. It contains the number of different rearrangements, split among the available devices. For example, if 8 Planar devices are available, every rearrangement will be performed on 4 devices in SymGS. The evaluated benchmarks contain strided and irregular memory accesses.

CompMG is the multigrid sparse solver present in HPCG [48]. *EBOX* [22] is an extended box filtering approximation of a Gaussian convolution for an image. *MatMul* [2] is an optimized matrix-matrix multiplication with blocking support. *Meabo* [3] is a multi-phased multi-purpose micro-benchmark, frequently used for energy efficiency studies. *Spatter* [35] is used for timing scatter/gather kernels on CPUs and GPUs. *SpMV* [18] is the sparse matrix-vector multiplication. The sparse matrix is represented in the CSR format [10] and the vector is dense. *STRIDE* [49] is a memory stress benchmark commonly used to characterize the memory system of HPC systems. Finally, *SymGS* is a Symmetric Gauss-Seidel smoother from HPCG [48], and performs a forward and backward triangular solve.

We modified these applications to use software rearrangement techniques similar to the one shown in Figure 2. All modified versions perform worse except *STRIDE*. This is due to the overhead to perform the data rearrangement on the core, which precludes computation overlap with the rearrangements, and due to involving the entire memory hierarchy, which adds additional latency and can lead to cache thrashing. For this reason, our evaluation in Section 5 uses the original unmodified application codes as the baseline.

Table 3: List of evaluated benchmarks.

Benchmark	Option	Input	# rearr. regions
CompMG		bcspwr10 (A), bcsstk15 (B), blckhole (C), circuit_1 (D), ex12 (E), lns_3937 (H), G30 (F), jan99jac100sc (G)	2
EBOX	Stride: 8, 16 and 32	400,000 double-type elems	4
MatMul		1x1 block of 400x400 elems, 2x2 blocks of 200x200 elems, 4x4 blocks of 100x100 elems, 2x2 blocks of 300x300 elems, 3x3 blocks of 200x200 elems, 6x6 blocks of 200x200 elems	1
Meabo	Phase2	300,000 double-type elems	1
Spatter	Distance: 1, 2, 4, 8, 16, 32, 64, 128, 256, random	300,000 double-type elems	1
SpMV		A, B, C, D, E, F, G, H	1
STRIDE	Distance: 1, 2, 4, 8, 16, 32, 64, 128	320,000 double-type elems	1
SymGS		A, B, C, D, E, F, G, H	2

The selected benchmarks have been modified to work with Planar. This process involves: (i) to define the rearrange function; (ii) to replace the original irregular accesses to the original data structures with the dense ones; and (iii) to add the Planar allocation, offload, and release calls. In most of the mentioned benchmarks, very few modifications are required to the original code: ≈20 lines of code for the rearrange function, the three Planar library calls, allocation via regular malloc/free of the dense data structure, and the code modifications to access the new dense data structure.

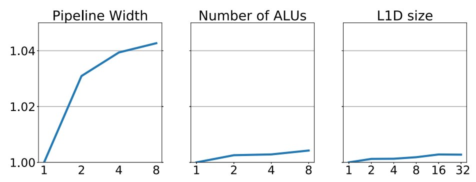

Figure 6: Normalized Planar design impact to performance in Spatter. Pipeline width (left), number of functional units (center) and L1-D cache size (right). In the x-axis the pipeline widths, number of functional units and cache size in *KB*.

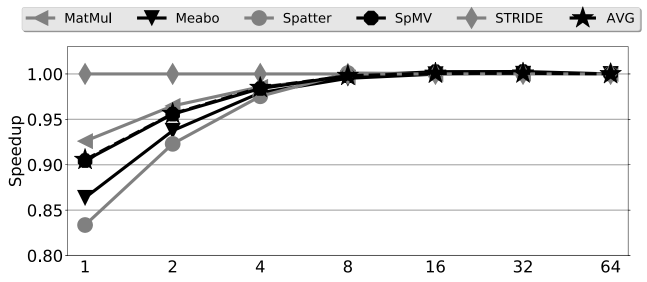

Figure 7: Performance relative to the number of Planar devices (x-axis), normalized to 64.

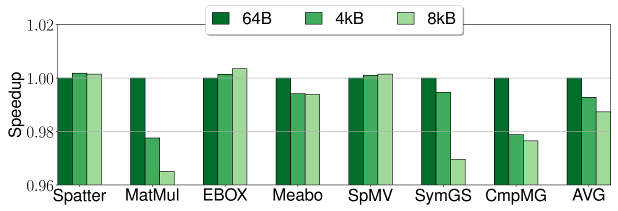

Figure 8: Speedup with eight Planar devices using different synchronization granularities. Results normalized to 64B.

5 RESULTS AND DISCUSSION

5.1 Design Space Exploration

In this section we size Planar hardware with a design space exploration study. Planar is envisioned as a simple microcontroller with in-order execution. For this reason, we explore the pipeline width, the data cache size, the synchronization granularity, and the number of functional units and Planar devices.

Pipeline width: Figure 6 (left) depicts the performance impact when changing the Planar pipeline width. Results are obtained using the average of all the inputs of the Spatter benchmark normalized to the single-issue scenario. When changing the input distance from 1 to 256 (see Table 3), Spatter provides a wide coverage of different irregular memory access patterns. Increasing the pipeline width from one to two provides between 2.0% and 4.0% improvements for the different inputs (3.1% on average). Further increasing the pipeline width provides diminishing improvements (3.9% and 4.3% on average for 4- and 8-wide pipelines, respectively). For small input distances, Spatter shows more cache locality. Thus, having a wider pipeline width in Planar provides higher performance benefits. As input distance increases, the latency of the memory requests hides the reduced performance of a narrow pipeline width.

Number of functional units: Figure 6 (center) shows the performance impact with respect to the number of functional units in Planar. Results are obtained using Spatter, normalized to one functional unit scenario. Increasing the number of functional units provides a marginal performance benefit, reaching an average 0.42% improvement with 8 units.

Cache size: Figure 6 (right) depicts the performance impact with respect to L1-D cache size. Spatter results are normalized to the 1KB scenario. In this case, increasing the data cache size provides negligible performance benefits (0.28% on average with 32KB). This is expected as the rearrange function has a streaming memory access pattern with nearly no temporal locality. Only for small input distances, Spatter shows some cache locality, providing reduced benefits. As distance increases, the cache size does not provide any performance benefit. Regarding the L1-I cache, the rearrange function requires less than 100 instructions in the evaluated benchmarks. Thus, it does not exceed the 1KB capacity.

Number of Planar devices: Figure 7 shows the impact of the number of Planar devices to performance. Due to hardware constraints, it is difficult for a single device to saturate the memory bandwidth, as not many outstanding memory requests are allowed per device. Results are obtained by performing the average across all inputs for the selected applications. We limit this study to the benchmarks that contain only a single rearrangement. This way we keep the number of devices per rearrangement constant. Results are normalized to the 64-Planar device scenario, which represents a *close-to-ideal* case in our simulation infrastructure. All benchmarks except STRIDE are sensitive to the number of Planar devices. With a single Planar device, performance degrades 9.5% on average with respect to 64 devices. Increasing the number of devices from 1 to 2 provides a 5.1% performance improvement, while moving from 2 to 4 provides an additional 3% improvement. With 8 devices all benchmarks are already within 1.0% the performance of 64.

Synchronization granularity: During the execution example of the design we assumed the RCT table is updated after every populated dense cache block (Section 3.7). However, synchronization between Planar and *host* can be done at a coarser granularity. We analyze scenarios where the *host* is notified it can consume rearranged data after 64B (cache block size), 4KB (page size), and 8KB. Fine-grain synchronization lets the *host* consume dense data as Planar rearranges it, but increases synchronization traffic. As Figure 8 shows, coarser grain granularities of 4KB cause less than 1% performance slowdown on average. This is because after the first dense chunk finishes, Planar and *host* can overlap subsequent chunk rearrangements with compute over already rearranged chunks. Using a synchronization granularity of 4KB reduces the total message count by 64 with respect to 64B.

Selected configuration: At the device level there is no significant improvement as the hardware complexity increases. For this reason, we choose a simple, low-power, dual issue in-order Planar accelerator, with a single integer functional unit, and a 1KB L1-D cache. The selected ratio of Planar devices with respect to off-chip bandwidth is one for every 4GB/s. Therefore, eight devices in our simulated system, as further increasing the number of devices provides negligible benefits. Finally, we chose a synchronization granularity of 4KB between Planar and *host* cores.

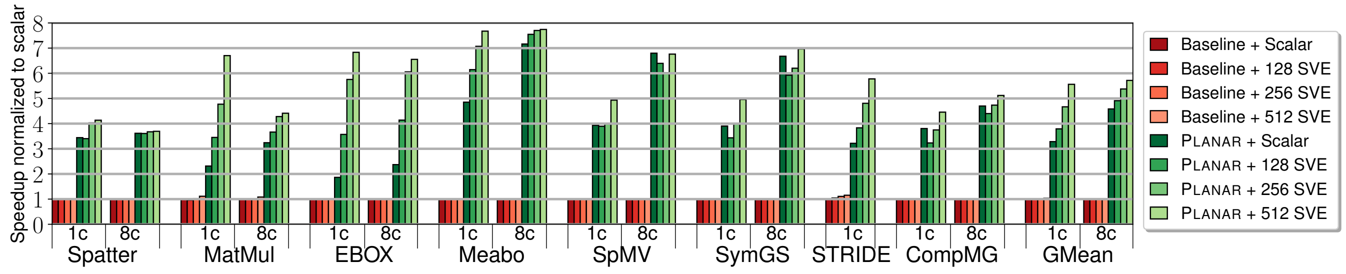

Figure 9: Speedups with eight Planar devices for one and eight *host* core runs. Both normalized to *Baseline + Scalar*.

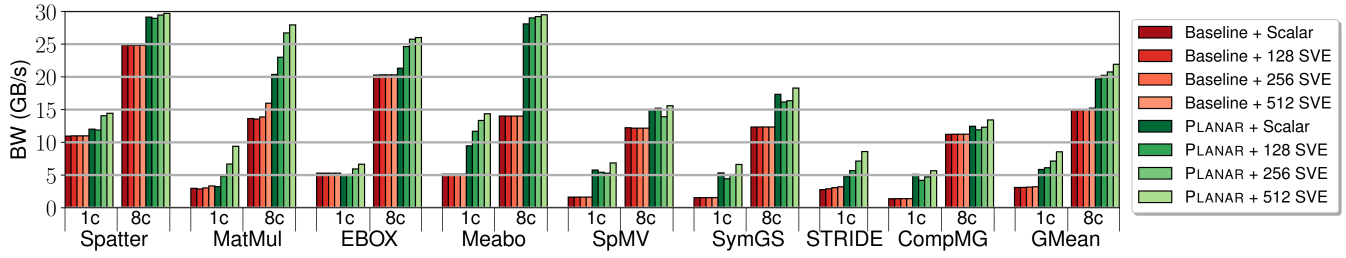

Figure 10: Average memory bandwidth usage with eight Planar devices for one and eight *host* cores.

5.2 Performance Evaluation

In this section, we analyze the performance impact of Planar. We use the applications described in Table 3. For each application we evaluate all the listed inputs and plot the average. We run simulations with one and eight threads to see their behavior. We also evaluate the impact of compiler auto-vectorization using the recently proposed Scalable Vector Extension (SVE) ISA [53]. SVE is vector length agnostic, meaning that a single binary can run on any target vector length [4]. Therefore, we evaluate a scalar binary and an SVE-enabled binary with vector lengths of 128, 256, and 512 bits. Figure 9 shows speedup for the evaluated benchmarks normalized to the scalar baseline system without Planar devices (*Baseline + Scalar*) for each core count. Figure 10 shows the average memory bandwidth usage.

In Spatter, a 3.44× speedup is achieved when using Planar and a single thread without vectorization. However, benefits are input-dependent. Low distances lead to lower sparseness and more cache locality. Higher distances affect execution time as there is a lower cache block utilization. Results also demonstrate that the original code is not auto-vectorized due to the irregular memory access pattern. However, Planar versions allow efficient auto-vectorization as memory accesses are now contiguous. Therefore, Planar unlocks further performance improvements through data-level parallelism, achieving 3.4×, 4.02× and 4.13× speedup for 128, 256, and 512-bit SVE, respectively. Memory bandwidth is better utilized with Planar as cores now bring useful dense data into their caches, while sparse accesses are done near-memory. With eight threads, the speedups remain significant at 3.61× for scalar, with similar results for the vectorized versions. In this case vectorization is not improving performance significantly because with Planar we are able to saturate memory bandwidth, driving 29GB/s out of the 32GB/s peak.

In MatMul, sparse memory accesses appear when accessing the second matrix. In this case, Planar dynamically transposes one of the input matrices to create a contiguous memory access pattern from the *host* core standpoint. The bigger the blocks, the higher the distance between elements. Using multiple matrix block sizes, an average 2.31× speedup is obtained on a single thread. In the baseline, vectorization provides a small performance benefit of 11%, as some phases of the application can be vectorized. Using Planar, SVE improves execution time as memory bandwidth is not a constraint. For instance, 512-bit SVE can drive an additional 6.05GB/s of memory bandwidth as the same baseline configuration, translating into a 6.70× speedup. With eight threads, Planar speedups are 3.24× for scalar. However, vectorization for wide vectors offers low returns as memory bandwidth saturates.

EBOX performs a Gaussian convolution by means of a filtering approximation. Filters take samples of the inputs to process the data. Consequently, Planar can be a good method to reorganize the input data and improve performance. In particular, EBOX extracts particular positions of an input and operates on them in two pairs (i.e., A[i] = p1*(B[-]-C[-]) + p2*(D[-]-E[-])). We have used Planar to create four dense structures, one for each element in the two pairs (i.e., B, C, D, E). As a result, we obtain a speedup of 1.86× for scalar and up to 6.83× for 512-bit SVE with good vector performance scaling. In the multi-threaded scenarios the performance behavior is similar. Note that in eight thread runs Planar again provides better normalized speedup compared to single-thread (i.e., 2.37× compared to 1.86×). This means that the overall design is well balanced in terms of compute, memory bandwidth and acceleration.

In Meabo, memory is accessed using a random indirection vector, which leads to non-existent locality and low cache block utilization. Single-thread runs with Planar obtain 4.85×, 6.14×, 7.07×, and 7.67× speedup for scalar, 128, 256, 512-bit SVE. Using a dense

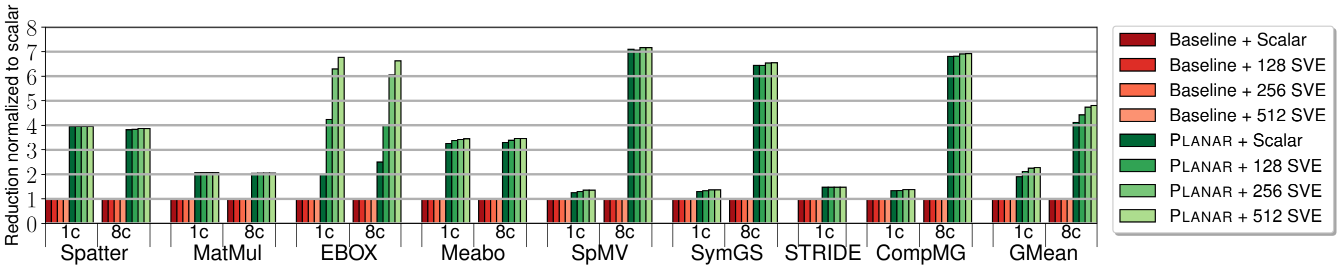

Figure 11: L1D miss reduction with 8 Planar devices for one and eight cores, both normalized to baseline scalar.

structure makes a large difference in this benchmark as memory bandwidth is poorly utilized in the baseline: due to (i) the low amount of reuse, and (ii) the small amount of memory level parallelism the cores are able to extract, as stalls are common due to long latency misses and contention. With Planar the memory bandwidth utilization almost doubles both for single and multi-threaded scenarios, saturating it in the latter.

In SpMV and SymGS, the matrix is compressed in CSR format and the vector is accessed sparsely, jumping from one element to the other. This vector is rearranged by Planar. Performance is dependent on the vector access pattern. For this reason, the selected input matrices that define the vector access pattern are obtained from a wide variety of scientific domains. In SpMV the matrix is traversed forward, while in SymGS it is done forward and backwards, requiring two rearrange tasks. On average, a 3.9× speedup is obtained for the scalar code on both applications. SVE 512-bit vectorization yields a 4.93× speedup, while the baseline cannot be efficiently vectorized by the compiler. The performance gap is larger on eight thread runs with a 6.7× speedup.

STRIDE is a memory-intensive application where the use of longer distances implies requesting memory more often, since fewer elements per cache block are accessed. In this particular benchmark, the *host* core and Planar can operate at the same time, competing for memory bandwidth resources. We evaluate multiple inputs to study this phenomenon and obtain an average speedup of 3.21× in the scalar version. Even though some phases in this benchmark are auto-vectorized in the baseline code, the phase with sparse memory accesses is again not vectorized. For this reason, baseline reaches an improvement of 1.15× using 512-bit SVE, while the Planar version obtains 5.77× for the same configuration.

Lastly, CompMG performs recursive calls that contain several calls to SpMV and SymGS. For every CompMG call only two different rearrange tasks are required, as the rearrange task in SpMV is the same as the first rearrange in SymGS (i.e., the forward matrix traversal). Planar speedups are 3.8×, 3.32×, 3.74× and 4.45× for scalar, 128, 256, and 512-bit SVE. Using eight threads we observe a better speedup than in the single thread case, such as a 5.19× in Planar with SVE 512-bit.

5.3 Impact to the Memory Hierarchy

Contiguous accesses to a dense data structure offer significant benefits compared to the original sparse access pattern, such as high cache block utilization and efficient data prefetching. Next lines demonstrate the impact Planar has on the memory hierarchy.

Figure 11 shows L1D miss reduction on *host* cores. The L1D is critical for the core's performance, and Planar rearrangements enable an average L1D miss reduction of 1.89× for one core. In Planar, all the elements contained in a rearranged cache block are referenced by the *host* core, whereas in the baseline, only one element is accessed in the worst case. The dense structure also causes a reduction of 53% in L1D miss latency. This is due to: (i) less touched cache blocks due to locality within a cache block, and (ii) memory access latencies being hidden due to better prefetching. Prefetching is less effective with irregular accesses, where data locality is difficult to exploit.

Figure 12 shows the data movement reduction in the L1D-L2 bus. The dense structure enables efficient cache block utilization and reduces cache pollution. On average, there is a 1.65× data movement reduction for one core.

In terms of DRAM accesses, one of the advantages of performing DLT is that subsequent accesses to the dense structure do not require accessing the intermediate data structure of the indirect memory access. In the baseline, both the intermediate and sparse structures are accessed. The latter may even have cache blocks accessed more than once, due to the significant cache pollution. Figure 13 depicts the normalized number of DRAM accesses. When using Planar, 1.6% of the total DRAM accesses are generated by Planar devices on average (up to 9.09% in CompMG). Overall, there is an average 41.89% DRAM access reduction. This reduction is primarily due to increased reuse, which can be observed indirectly through the increased L1D hit rate (see Figure 11) and L1D to L2 bandwidth reduction (see Figure 12). Despite writing the dense data structure back to memory, the actual accesses to DRAM are reduced because of better data cache utilization and far higher reuse of cached data. Writing data back to main memory reduces the repeated rearrangement calls needed by alternatives such as Impulse, and enables the dense data structure to be reused many times before being freed.

5.4 Area and Power Overhead

Planar devices can be compared to the Arm Cortex $M0+$ microprocessor, which is augmented with a 64-bit ALU (as discussed in Section 3). Using public data for an equivalent $M0+$ at 40LP [15], the dynamic power is given as $5.3\mu W/MHz$ and the floor planned area as $.008mm^2$. To estimate the area of a single microcontroller, we scale these numbers, considering fin pitch, gate pitch, and interconnect pitch, using data from [5, 14, 16, 58, 59, 62] to arrive at a 12× area reduction when moving from 40nm to 7nm and an estimated

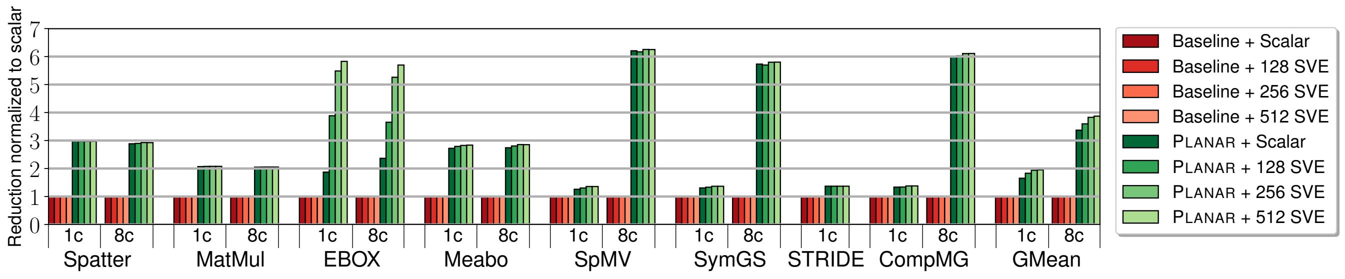

Figure 12: Byte reduction in the L1D-L2 bus with 8 Planar for one and eight cores, normalized to baseline scalar.

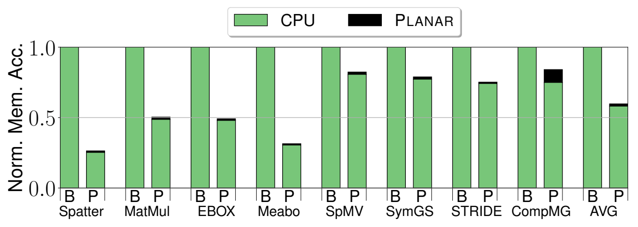

Figure 13: Normalized DRAM accesses for baseline (B) and Planar (P) on 8 cores with scalar codes.

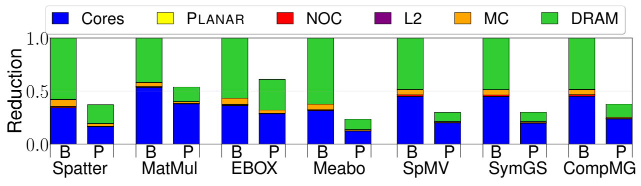

Figure 14: Dynamic energy reduction baseline (B) vs Planar (P) with scalar application binaries on eight *host* cores.

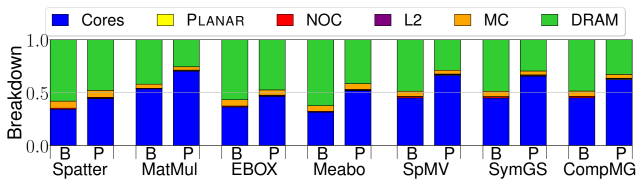

Figure 15: Dynamic energy breakdown baseline (B) vs Planar (P) with scalar application binaries on eight *host* cores.

reduction in power of 10×. Therefore, a system-on-chip could place 8 Planar devices, with their caches, using $< .25mm^2$. Equivalently, energy for this configuration would be $< .015W/GHz$.

We also estimate the area of the out-of-order core described in Table 2. We start from a similar production core at 20nm [41] and scale it to 7nm, arriving at a $\approx 4mm^2$ in area. The approximate area ratio of a Planar device to an out-of-order core is 1 : 16, i.e., 8 Planar devices are equivalent to 50% of a single out-of-order core. The ratio for dynamic power is 1 : 260 assuming both run at 2GHz.

To estimate the dynamic energy and power consumption for our Planar proposal we used McPAT 1.3 [37] with the enhancements proposed by Xi *et al.* [61]. We performed this estimation, using a process technology node of 22nm, a supply voltage of 0.8V, and the default clock gating scheme. Figure 14 depicts the dynamic energy reduction for the applications with eight *host* cores. Overall, dynamic energy is reduced by more than 40% and up to 70% in Meabo. Energy savings are mainly due to reduced data movement across the memory hierarchy and reduced execution time (speedup) as shown in Figure 9.

Figure 15 depicts the dynamic energy breakdown for the same applications. Compared to the baseline, Planar spends less DRAM energy. As explained in Section 5.3, Planar creates a dense structure and makes the applications more compute intensive form the host standpoint, as average data access latencies are lower.

The total dynamic power is higher in Planar. On average, the dynamic power increases with Planar by 2.41× in the cores, by 1.41× in the memory controller, and by 1.14× in DRAM. This is due to an increase in terms of activity per unit of time. However, taking into account the performance speedups of Planar, the overall energy spent is significantly reduced. As previously discussed, static power is merely affected when adding Planar.

5.5 Comparison to Other Proposals

In this section, we quantitatively compare Planar to Impulse [12] and to a recent DLT accelerator proposal [27]. Impulse is a hardware approach that creates a dense structure out of a sparse one. It performs data reordering in the memory controller as the *host* core accesses memory belonging to a *shadow address space*, which must be contiguous in physical memory. Thus, Impulse rearranges data *just in time*, not like Planar which is capable of rearranging data *before-hand* as the *host* is executing other code regions. Therefore, Impulse cannot hide the rearrangement latency. Moreover, in case the dense data is evicted and requested by the *host* again, Impulse will perform a new rearrangement, as it cannot assume that the original and new rearrage functions are the same. Finally, in a design with multiple memory controllers, the rearrange functions of different Impulse instances are not synchronized, limiting scalability.

Figure 16 depicts the performance comparison between Planar and Impulse for SpMV. Planar obtains an average 2.12× speedup compared to Impulse. This is due to: (i) higher MLP of data rearrangements in Planar; (ii) more data reuse; and (iii) less data movement in the cache hierarchy as dense data is created just once. For instance, snoop traffic from the core to the L2 cache is 21× higher in Impulse.

In contrast, the DLT accelerator is tightly coupled to the *host* core, whereas PLANAR is connected to the memory controller. It can bypass the cache hierarchy and directly access main memory, as PLANAR does, but requires an additional data bus. The accelerator does not allow the *host* to consume dense data as the device rearranges it, blocking memory accesses on the *host* during DLT operation to maintain data consistency. Moreover, it supports a maximum of four parallel operations, contrary to PLANAR, where more devices can be added to the system, enabling additional parallelism.

Figure 17 depicts the performance comparison between PLANAR and the DLT accelerator for several benchmarks. We employ up to 8 PLANAR devices and also allow up to 8 simultaneous operations on the DLT accelerator. On average, PLANAR obtains a 2.23× speedup compared to the DLT accelerator, with a maximum of 5.82× in SymGS. This is due to two main reasons: in PLANAR (i) the *host* is not blocked while devices are operating, as PLANAR effectively decouples rearrange and compute, and (ii) the *host* can consume dense data as it is rearranged, which hides rearrangement latencies.

6 RELATED WORK

Data layout transformation (DLT) is a solution to increase the efficiency of memory accesses [17, 36]. Nevertheless, it is not always obvious which layout will better serve a particular application and hardware combination. *DLT* in software can be driven by run-ahead or decoupled threads [47], or by customized data structures designed to reduce data movement [63]. Although run-ahead threads may run speculatively to hide memory access latency, they are limited by the instruction window they can execute ahead of the main thread. This situation significantly limits the overlap of data rearrangements and computation.

Others works have also focused on *DLT*, including: the Impulse memory controller [12, 65], the tightly-coupled DLT accelerator by Hoang *et al.* [27], DReAM [23], the data rearrangement engine (DRE) [39], and the *SPiDRE* system architecture [7]. We have compared PLANAR qualitatively to these proposals in Section 1, and quantitatively to the first two in Section 5.5. In this paper, we demonstrate PLANAR within a general-purpose system architecture, featuring a scalable design that is able to hide rearrangement latency while providing programming interfaces that can be easily incorporated into any application. As shown in Table 1, prior proposals do not provide some of these features, limiting their adoption. Moreover, prior proposals are largely demonstrated on a specific application domain (e.g., pointer chasing), or with high-level system architecture descriptions [7] that prevent quantitative comparisons.

DLT can be considered a form of Decoupled Access Execute (*DAE*). In this context we may include the works of Smith [50], Koukos *et al.* [32, 33], Jimborean *et al.* [29] and Charitopoulos *et al.* [13]. A key difference between standard *DAE* and PLANAR is that traditional *DAE* focuses primarily on maximizing memory-level parallelism and reducing memory latency experienced by the *host* core. However, PLANAR accomplishes both tasks while also reducing data movement (see Figures 12 and 13).

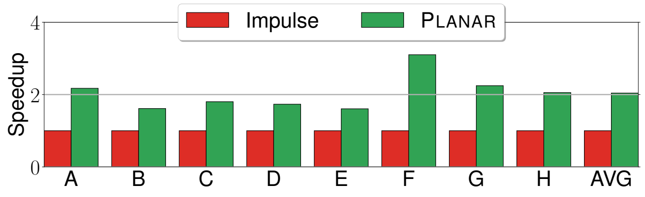

Figure 16: Performance of Impulse vs PLANAR in SpMV. In the x-axis, the matrix inputs from Table 3.

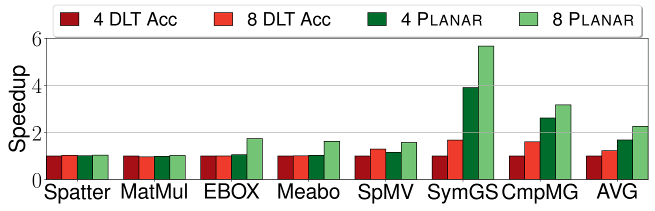

Figure 17: Performance of the DLT accelerator vs PLANAR.

Other works that belong to the Processing In/Near Memory (PINM) field, include: Computational Ram [19], Terasys [24], and DIVA [25]. More recent PINM efforts are well summarized by Zhang *et al.*. [64] and Balasubramonian *et al.* [6]. While they focus on the more general problem of PINM, we focus on offload with the purpose of *DLT* for traditional cores.

PLANAR also shares some points with the concept of data prefetching. Data prefetching, when data retrieved is fully utilized, can reduce latency while having a negligible impact on overall data movement. However, in many workloads data is not fully utilized leading to much wasted bandwidth and energy consumption [8]. PLANAR moves the data access outside of the primary core, just as standard prefetching does. In contrast, it is driven by access code derived from the application and it also reduces data motion whereas prefetchers typically increase it [28]. Prefetchers for workloads most similar to those evaluated here include the works of Ainsworth *et al.* [1], Kocberber *et al.* [31] and Kumar *et al.* [34] Unlike them, PLANAR is a general purpose approach suitable for multi-core scenarios. Moreover, PLANAR is not limited by the shape the data structure has, operates with virtual memory, reduces data motion, and exposes vectorization.

While it is often assumed that applications can be authored to minimize data movement for even the most irregular and sparse applications, many real applications cannot be. Several recent works have examined this phenomena for dynamic graphs [26, 42]. In summary, they found that static graphs could make use of specialized data structures. However, in many cases the processing time for setting up specialized structures for dynamic graphs far outweighed the cost to process a less optimal structure which is more amenable to dynamic updates. Instead of focusing on pre-processing for optimal locality, technologies such as PLANAR enable application developers to have the impact associated with specialized data structures (i.e., maximizing cache utilization) without spending time developing them.

7 CONCLUSIONS

Irregular memory accesses represent a challenge for current and future architectures. In this work, we present PLANAR, a programmable near-memory accelerator that rearranges sparse data into a dense representation. Contrary to prior proposals, the design of PLANAR scales with multi-core systems, hides operation latency by performing non-blocking fine-grain data rearrangements, and eases programmability by supporting virtual memory and conventional memory allocation mechanisms. Moreover, accesses to the dense structure expose locality, favouring prefetching and enabling efficient vectorization in applications with irregular memory accesses. No prior solution provides all such properties at once.

Our evaluation shows that PLANAR improves cache block utilization and reduces on-chip data movement. As a result, PLANAR improves performance for single-threaded runs by 3.28× and 5.56×, and for multi-threaded executions by 4.58× and 5.71×, for scalar and compiler-vectorized codes. Finally, we compare PLANAR to two state-of-the-art proposals, achieving 2.12× and 2.23× average performance improvements.

8 ACKNOWLEDGMENTS

A. Barredo has been partially supported by the Spanish Government under Formación del Personal Investigador fellowship number BES-2017-080635. A. Armejach has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Juan de la Cierva postdoctoral fellowship number IJCI-2017-33945. M. Moretó has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship number RYC-2016-21104. This work has been partially supported by the European Union's Horizon 2020 research and innovation program under the Mont-Blanc 2020 project (grant agreement 779877), by the Spanish Ministry of Science and Innovation (PID2019-107255GB-C21/AEI/10.13039/501100011033), by the Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328), and by the Arm-BSC Centre of Excellence.

REFERENCES

[1] Sam Ainsworth and Timothy M Jones. 2016. Graph prefetching using data structure knowledge. In *Proceedings of the 2016 International Conference on Supercomputing*.

[2] Sumaia Al-Ghuribi. 2012. Matrix Multiplication Algorithms. *International Journal of Computer Science and Network Security* (2012).

[3] Arm Limited. 2018. Meabo. Available at https://github.com/ARM-software/meabo.

[4] Adrià Armejach, Helena Caminal, Juan M. Cebrian, Rekai González-Alberquilla, Chris Adeniyi-Jones, Mateo Valero, Marc Casas, and Miquel Moretó. 2018. Stencil codes on a vector length agnostic architecture. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT 2018*. 13:1–13:12.

[5] F Arnaud, A Thean, M Eller, M Lipinski, YW Teh, M Ostermayr, K Kang, NS Kim, K Ohuchi, JP Han, et al. 2009. Competitive and cost effective high-k based 28nm CMOS technology for low power applications. In *IEEE International Electron Devices Meeting (IEDM)*.

[6] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-data processing: Insights from a MICRO-46 Workshop. *Micro* 34 (2014).

[7] Jonathan C Beard. 2017. The Sparse Data Reduction Engine (SPiDRE): Chopping Sparse Data One Byte at a Time. In *Proceedings of the Second International Symposium on Memory Systems*.

[8] Jonathan C Beard and Joshua Randall. 2017. Eliminating Dark Bandwidth: a data-centric view of scalable, efficient performance, post-Moore. In *International Conference on High Performance Computing*.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark Hill, and David Wood. 2011. The gem5 simulator. (2011).

[10] Susan Blackford. 2000. *Compressed row storage*. http://www.netlib.org/utk/people/JackDongarra/etemplates/node373.html Accessed December 2019.

[11] Shekhar Borkar. 2013. Role of interconnects in the future of computing. *Journal of Lightwave Technology* (2013).

[12] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, et al. 1999. Impulse: Building a smarter memory controller. In *High-Performance Computer Architecture, Proceedings. Fifth International Symposium On.*

[13] George Charitopoulos, Charalampos Vatsolakis, Grigorios Chrysos, and Dionisios N Pnevmatikatos. 2018. A decoupled access-execute architecture for reconfigurable accelerators. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*.

[14] Kuan-Lun Cheng, CC Wu, YP Wang, Da-Wen Lin, CM Chu, YY Tarng, SY Lu, SJ Yang, MH Hsieh, CM Liu, et al. 2007. A highly scaled, high performance 45 nm bulk logic CMOS technology with 0.242 μm2 SRAM cell. In *2007 IEEE International Electron Devices Meeting*.

[15] CortexM0 [n.d.]. *Arm Cortex-M0*. Accessed April 2019.

[16] Denis C Daly, Laura C Fujino, and Kenneth C Smith. 2018. Through the Looking Glass-The 2018 Edition: Trends in Solid-State Circuits from the 65th ISSCC. *IEEE Solid-State Circuits Magazine* (2018).

[17] Inderjit S. Dhillon and Dharmendra S. Modha. 2001. Concept Decompositions for Large Sparse Text Data Using Clustering. *Machine Learning* (2001). https://doi.org/10.1023/A:1007612920971

[18] Athena Elafrou, Georgios I. Goumas, and Nectarios Koziris. 2017. Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors. *CoRR* (2017). http://arxiv.org/abs/1711.05487

[19] Duncan G Elliott, W Martin Snelgrove, and Michael Stumm. 1992. Computational RAM: A memory-SIMD hybrid and its application to DSP. In *Custom Integrated Circuits Conference*.

[20] Amin Farmahini-Farahani, Sudhanva Gurumurthi, Gabriel Loh, and Michael Ignatowski. 2018. Challenges of High-Capacity DRAM Stacks and Potential Directions. In *Proceedings of the Workshop on Memory Centric High Performance Computing*.

[21] Gene Fuller. 2017. Future lithography technology. In *Single Frequency Semiconductor Lasers*.

[22] Pascal Getreuer. 2013. A Survey of Gaussian Convolution Algorithms. *Image Processing On Line* (2013).

[23] Mohsen Ghasempour, Aamer Jaleel, Jim D. Garside, and Mikel Luján. 2016. DReAM: Dynamic Re-arrangement of Address Mapping to Improve the Performance of DRAMs. In *Proceedings of the Second International Symposium on Memory Systems (MEMSYS '16)*.

[24] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in memory: The Terasys massively parallel PIM array. *Computer* (1995).

[25] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman, Apoorv Srivastava, et al. 1999. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*.

[26] Eric Hein and Tom Conte. 2016. DynoGraph: Benchmarking Dynamic Graph Analytics. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis. http://sc16. supercomputing. org/sc-archive/tech_poster/tech_poster_pages/post214. html Poster.*

[27] Tung Hoang, Amirali Shambayati, and Andrew Chien. 2016. A Data Layout Transformation (DLT) Accelerator: Architectural Support for Data Movement Optimization in Accelerated-centric Heterogeneous Systems.

[28] Mahzabeen Islam, Soumik Banerjee, Mitesh Meswani, and Krishna Kavi. 2016. Prefetching As a Potentially Effective Technique for Hybrid Memory Optimization. In *Proceedings of the Second International Symposium on Memory Systems (MEMSYS '16)*.

[29] Alexandra Jimborean, Konstantinos Koukos, Vasileios Spiliopoulos, David Black-Schaffer, and Stefanos Kaxiras. 2014. Fix the Code. Don'T Tweak the Hardware: A New Compiler Approach to Voltage-Frequency Scaling. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*.

[30] Wenting Jin. 2017. Feedback Compilation for Decoupled Access-Execute Techniques.

[31] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*.

[32] Konstantinos Koukos, David Black-Schaffer, Vasileios Spiliopoulos, and Stefanos Kaxiras. 2013. Towards more efficient execution: A decoupled access-execute approach. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*.

[33] Konstantinos Koukos, Per Ekemark, Georgios Zacharopoulos, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. 2016. Multiversioned Decoupled Access-execute: The Key to Energy-efficient Compilation of General-purpose Programs. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*.

[34] Snehasish Kumar, Arrvindh Shriraman, Vijayalakshmi Srinivasan, Dan Lin, and Jordon Phillips. 2014. SQRL: hardware accelerator for collecting software data structures. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*.

[35] Patrick Lavin, E. Jason Riedy, Rich Vuduc, and Jeffrey Young. 2018. Spatter: A Benchmark Suite for Evaluating Sparse Access Patterns. *CoRR* (2018). arXiv:1811.03743 http://arxiv.org/abs/1811.03743

[36] Ping Li, Trevor J. Hastie, and Kenneth W. Church. 2006. Very Sparse Random Projections. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[37] Sheng Li, Jung Ho Ahn, Richard Strong, Jay Brockman, Dean Tullsen, and Norman Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 469–480.

[38] Arm Limited. 2015. ARM Cortex-A Series. Programmer's Guide for ARMv8-A.

[39] Scott Lloyd and Maya Gokhale. 2015. In-Memory Data Rearrangement for Irregular, Data-Intensive Computing. *Computer* (2015).

[40] Chris Lomont. 2011. Introduction to Intel Advanced Vector Extensions. *Intel White Paper* (2011).

[41] Hugh T Mair, Gordon Gammie, Alice Wang, Rolf Lagerquist, CJ Chung, Sumanth Gururajarao, Ping Kao, Anand Rajagopalan, Anirban Saha, Amit Jain, et al. 2016. 4.3 A 20nm 2.5 GHz ultra-low-power tri-cluster CPU subsystem with adaptive power allocation for optimal mobile SoC performance. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*.

[42] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. 2017. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference USENIX ATC 17*.

[43] John Mellor-Crummey, David Whalley, and Ken Kennedy. 1999. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 13th international conference on Supercomputing*.

[44] Sparsh Mittal. 2016. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)* (2016).

[45] Leonid Oliker, Andrew Canning, Jonathan Carter, John Shalf, David Skinner, Ethier Ethier, Rupak Biswas, Jahed Djomehri, and Rob Van der Wijngaart. 2003. Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations. In *SC'03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*.

[46] Georgios Petrousis. 2017. *An Evaluation of Decoupled Access Execute on ARMv8*. Master's thesis. Uppsala University.

[47] T. Ramírez, A. Pajuelo, O. J. Santana, O. Mutlu, and M. Valero. 2010. Efficient Runahead Threads. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[48] Sandia Report, Jack Dongarra, and Michael A. Heroux. 2013. Toward a New Metric for Ranking High Performance Computing Systems.

[49] Oak Ridge, Argonne, and Livermore. 2018. The Coral-2 Benchmark Suite. Available at https://asc.llnl.gov/coral-2-benchmarks/.

[50] James E Smith. 1982. Decoupled access/execute computer architectures. In *ACM SIGARCH Computer Architecture News*.

[51] Jeffrey R Spirn and Peter J Denning. 1972. Experiments with program locality. In *Proc. of ACM Fall Joint Computer Conference, Part I*.

[52] James R Srinivasan. 2011. *Improving cache utilisation*. Technical Report. University of Cambridge, Computer Laboratory.

[53] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. 2017. The Arm scalable vector extension. *IEEE Micro* (2017).

[54] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L Chamberlain, Romain Cledat, H Carter Edwards, Hal Finkel, et al. 2017. Trends in data locality abstractions for HPC systems. *IEEE Transactions on Parallel and Distributed Systems* (2017).

[55] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. 2011. The RISC-v instruction set manual, volume I: Base user-level ISA. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* 116 (2011).

[56] John Wiegert, Greg Regnier, and Jeff Jackson. 2007. Challenges for scalable networking in a virtualized server. In *2007 16th International Conference on Computer Communications and Networks*. IEEE, 179–184.

[57] Michael Joseph Wolfe. 1996. *High performance compilers for parallel computing*. Addison-Wesley.

[58] Shien-Yang Wu, CY Lin, MC Chiang, JJ Liaw, JY Cheng, SH Yang, M Liang, T Miyashita, CH Tsai, BC Hsu, et al. 2013. A 16nm FinFET CMOS technology for mobile SoC and computing applications. In *2013 IEEE International Electron Devices Meeting*.

[59] Shien-Yang Wu, CY Lin, MC Chiang, JJ Liaw, JY Cheng, SH Yang, CH Tsai, PN Chen, T Miyashita, CH Chang, et al. 2016. A 7nm CMOS platform technology featuring 4th generation FinFET transistors with a 0.027 μm2 high density 6-T SRAM cell for mobile SoC applications. In *2016 IEEE International Electron Devices Meeting (IEDM)*.

[60] Wm Wulf and Sally A. McKee. 1994. *Hitting the Memory Wall: Implications of the Obvious*. Technical Report. Charlottesville, VA, USA.

[61] S. L. Xi, H. Jacobson, P. Bose, G. Wei, and D. Brooks. 2015. Quantifying sources of error in McPAT and potential impacts on architectural studies. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 577–589.

[62] R Xie, P Montanini, K Akarvardar, N Tripathi, B Haran, S Johnson, T Hook, B Hamieh, D Corliss, J Wang, et al. 2016. A 7nm FinFET technology featuring EUV patterning and dual strained high mobility channels. In *2016 IEEE International Electron Devices Meeting (IEDM)*.

[63] T. Yamada, S. Hirasawa, H. Takizawa, and H. Kobayashi. 2015. A Case Study of User-Defined Code Transformations for Data Layout Optimizations. In *2015 Third International Symposium on Computing and Networking (CANDAR)*.

[64] Dong Ping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph Greathouse, Mitesh Meswani, Mark Nutter, and Mike Ignatowski. 2013. A new perspective on processing-in-memory architecture design. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*.

[65] Lixin Zhang, Zhen Fang, Mide Parker, Binu K. Mathew, Lambert Schaelicke, John B. Carter, Wilson C. Hsieh, and Sally A. McKee. 2001. The Impulse Memory Controller. *IEEE Trans. Comput.* (2001).

[66] Xiaotong Zhuang and H-HS Lee. 2003. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *2003 International Conference on Parallel Processing, 2003. Proceedings*.