POSTER: SPiDRE: Accelerating Sparse Memory Access Patterns

Adrián Barredo
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
adrian.barredo@bsc.es

Jonathan C. Beard
Arm Research
jonathan.beard@arm.com

Miquel Moretó
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
miquel.moreto@bsc.es

***Abstract*—Development in process technology has led to an exponential increase in processor speed and memory capacity. However, memory latencies have not improved as dramatically and represent a well-known problem in computer architecture. Cache memories provide more bandwidth with lower latencies than main memories but they are capacity limited. Locality-friendly applications benefit from a large and deep cache hierarchy. Nevertheless, this is a limited solution for applications suffering from sparse and irregular memory access patterns, such as data analytics. In order to accelerate them, we should maximize usable bandwidth, reduce latency and maximize moved data reuse. In this work we explore the Sparse Data Rearrange Engine (SPiDRE), a novel hardware approach to accelerate these applications through near-memory data reorganization.**

I. Introduction

Processor speed and memory capacity have naturally exponentially evolved with advances in process technology. On the other hand, memory latencies have not seen the same evolution and they are a system performance limitation, a phenomenon known as the Memory Wall [1].

Locality-friendly applications can benefit from deep memory hierarchies. In this case, a combination of low-latency cache memories and prefetching hides main memory access latencies. On the other hand, applications with sparse and irregular memory access patterns do not see much improvement in large memory hierarchies. In many situations, they are counter-productive due to a low cache line utilization (i.e. cache pollution) and useless and difficult to predict prefetchings that represent an extra data movement. Making matters worse, vector instruction sets, which exploit data-level parallelism (DLP), have seen a comeback and their efficiency is similarly limited by memory bandwidth and latency.

In this work, we present the Sparse Data Rearrange Engine (SPiDRE), a novel hardware approach that performs data rearrangement near memory, transforming sparse data to dense data. SPiDRE improves performance first by decoupling (parallelizing) access and execute, the accesses performed in parallel with execution on the *host* core. SPiDRE also compacts data, making more efficient bandwidth usage and enabling more data to fit in the cache. Consequently, SPiDRE allows applications to take better advantage of the memory hierarchy (even prefetching) reducing memory latency experienced by the *host* core.

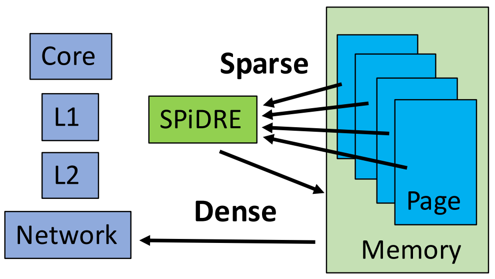

Fig. 1: SPiDRE device general picture. The device gathers data from sparse locations and creates a new and dense version.

II. SPiDRE Overview

SPiDRE is implemented for our evaluation as a near-memory device, connected to the main coherence bus with direct access to the memory controllers. SPiDRE is designed for applications with low cache block utilization [2], which access dispersed memory locations and cause high (but under-utilized) traffic on data transfer networks (e.g., coherence bus, interconnects, etc.) [3]. SPiDRE transforms a data structure into a new one using a rearrange function specified by the user. In the latter, elements are reorganized the way they are accessed by the *host*, with the goal of improving cache block and bandwidth utilization. The latency of this process can be overlapped with *host* computation. The reorganized data can be successfully prefetched using a simple next line prefetcher.

A. *SPiDRE Architecture*

SPiDRE is a programmable but relatively simple device. It can be implemented as a microcontroller placed next to the memory controller. It works as an accelerator on behalf of a requesting core process. The core sends commands to the device and suspends or computes, if there is any computation to be done in the meantime, until synchronization messages are received. Multiple devices may operate at the same time, applying the same rearrange function or different ones in parallel. Following lines explain the SPiDRE architecture.

1) Address Translation: In order to reduce area overhead, a direct virtual-to-physical address translation is performed in SPiDRE using simple base plus offset virtual memory calculations. It requires the data structures accessed by the devices to reside in contiguous physical pages. The *host* translates the base address and provides it to SPiDRE.

DOI 10.1109/PACT.2019.00056

2) Maintaining Memory Consistency: SPiDRE and the *host* core make use of a common coherence network. SPiDRE will often work on shared data with the host core. For this reason, if the data is contained in the caches or in the device scratchpads it must be flushed to main memory to make both units work with the most recent version of the data.

3) Micro-architectural Support: The rearranged data must not be accessed until it has been populated by SPiDRE. The *host* core and the prefetcher may issue memory requests to these memory locations while SPiDRE is operating, leading to undefined behavior. A new structure, the SPiDRE Control Table (*SCT*), keeps information about every rearrangement in flight from a *host* core. It is located in the core and in the prefetcher. If the data to be accessed is not ready, the memory request will be stored in a queue.

B. SPiDRE Phases

This section describes the phases involved in the rearrange process in sequential order.

1) Flushing of Sparse Data / Invalidation of Dense Data: Before the rearrangement, flush/invalidate operations are triggered on the SPiDRE device in order to maintain memory consistency,. Once flushes complete, data blocks will be available to the *host* in a shared state but read only and data to be rearranged will be in valid form for the SPiDRE units.

2) Allocation of SPiDRE Accelerators: Multiple cores may plan to use these devices at the same time. In the allocation, every core dynamically reserves SPiDRE devices depending on their availability and the minimum and a maximum number of accelerators they want to employ.

3) Offloading of Rearrange Functions: The *host* provides every associated allocated SPiDRE the rearrange function, the work boundaries (i.e. start and end indices in the final structure) and the translated physical base addresses for each structure to be accessed by the devices.

4) Rearrangement Trigger: The *host* core issues a command to notify the rearrangement starts. Our implementation assumes a write to a memory mapped register to initiate.

5) Execution of Rearrange Functions: In this phase, every SPiDRE device accesses the sparse data, performing the irregular memory accesses, and populating the dense data structures. Every accelerator has received its rearrange function, data pointers and work boundaries in the Offloading phase.

6) Synchronization between SPiDRE and Host Core: It is needed to ensure the *host* only accesses data when it is ready. Every time the devices rearrange a complete cache block, and it is flushed into main memory from their scratchpads, a signal is sent to the *host* so that it can consume it. The *SCT* controls that the *host* does not exceed the already rearranged limits.

7) Release of SPiDRE Devices: At the moment the last element assigned in the Offloading phase is rearranged, the SPiDRE device suspends and becomes available for future rearrangements.

III. Methodology

We employ *gem5* [4] to simulate an Arm full-system environment. We simulate a single out-of-order core processor extended with the micro-architectural support for SPiDRE. SPiDRE is modelled as an in-order core. The funcionalities and latencies from the previously described phases are modelled. The approach is evaluated using a set of representative benchmarks that show irregular memory access patterns.

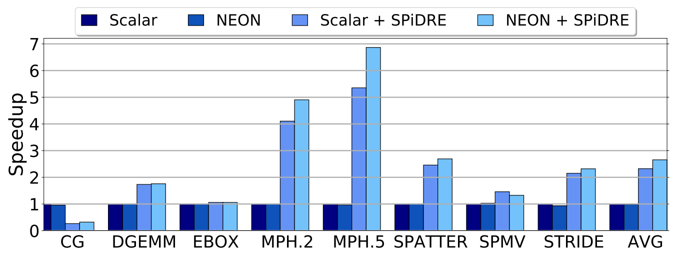

Fig. 2: Speedup using 8 SPiDRE devices per rearrangement. Numbers with scalar, NEON, scalar + SPiDRE and NEON + SPiDRE codes, normalized to the scalar scenario.

IV. Evaluation

Figure 2 shows the speedup *host* vs *host* + SPiDRE using eight SPiDRE accelerators. A SPiDRE + scalar scenario provices an average 2.3× speedup. The SPiDRE device creates a compacted version of the data and thus, new vectorization capabilities are exposed to the compiler. The original benchmarks cannot be efficiently vectorized due to the irregular memory accesses. Consequently, they do not obtain a significant benefit with NEON support. In contrast, a SPiDRE + NEON scenario achieves a 2.7× speedup.

Higher strides in the memory accesses imply a low cache line and bandwidth utilization.

In DGEMM, results depend on the matrix block sizes. In this case, SPiDRE transposes one of the input matrices. The bigger the blocks, the higher the distance between elements and the speedup. For instance, 200x200, 300x300 and 400x400 matrix blocks provide ≈1.3×, ≈2.3× and ≈3× speedups.

In SpMV, the performance depends on the non-zero element positions in the input matrix. They define the accesses to the vector. We selected matrix inputs from a wide variety of scientific domains. On average, a 1.62× speedup is obtained.

V. Acknowledgments

This work has been supported by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P, Ramon y Cajal fellowship number RYC-2016-21104 and FPI fellowship number BES-2017-080635), and by the Arm-BSC Centre of Excellence initiative.

References

[1] W. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” Charlottesville, VA, USA, Tech. Rep., 1994.

[2] J. C. Beard and J. Randall, “Eliminating dark bandwidth: a data-centric view of scalable, efficient performance, post-Moore,” in *HiPC*, 2017.

[3] J. Mellor-Crummey, D. Whalley, and K. Kennedy, “Improving memory hierarchy performance for irregular applications,” in *ICS*, 1999.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib Bin Altaf, N. Vaish, M. Hill, and D. Wood, “The gem5 simulator,” vol. 39, 2011.