# Emulating In-Memory Data Rearrangement for HPC Applications

Chris Hajas
University of Florida, Gainesville, FL
hajas@hcs.ufl.edu

Scott Lloyd
Lawrence Livermore National Laboratory, Livermore, CA
lloyd23@llnl.gov

Maya Gokhale
Lawrence Livermore National Laboratory, Livermore, CA
gokhale2@llnl.gov

## ABSTRACT

As bandwidth requirements for scientific applications continue to increase, new and novel memory architectures to support these applications are required. The Hybrid Memory Cube is a high-bandwidth memory architecture containing a logic layer with stacked DRAM. The logic layer aids the memory transactions; however, additional custom logic functions to perform near-memory computation are the subject of various research endeavors.

We propose a Data Rearrangement Engine in the logic layer to accelerate data-intensive, cache unfriendly applications containing irregular memory accesses by minimizing DRAM latency through the coalescing of disjoint memory accesses. Using a custom FPGA emulation framework, we found 1.4x speedup on a Sparse-Matrix, Dense-Vector application (SpMV). We investigated the multi-dimensional parameter space to achieve maximum speedup and determine optimal cache invalidation and memory access coalescing schemes on various sizes/densities of matrices.

## 1. INTRODUCTION

Memory bandwidth is of increasing concern as we design next-generation architectures to meet the needs of high performance data-intensive applications. With the advent of 3D high bandwidth memory architectures containing a custom logic layer beneath stacked DRAM such as Micron's Hybrid Memory Cube[2] (HMC), there is significant promise in delivering the bandwidth necessary for such applications. However, these architectures will not significantly improve applications bound by memory latency.

To accelerate such applications, we propose adding a Data Rearrangement Engine (DRE) to the logic. The DRE uses DMA and gather/scatter hardware to present a coalesced view of the data to the application based on an access pattern passed to the logic. The new view contains a contiguous stream of data in the order it will be accessed, allowing the application to take advantage of spatial locality in cache and SIMD units in the CPU. We modify SpMV to achieve over 1.4x speedup on large matrices.

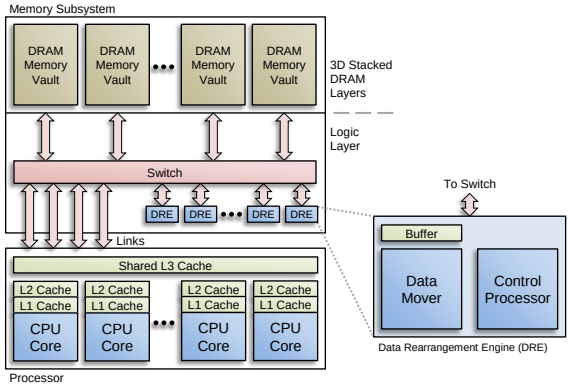


**Figure 1: DRE Emulation Framework**

## 2. EMULATION FRAMEWORK

An application can use the DRE without significant refactoring in many cases. However, because the CPU is not cache coherent with the DRE, the developer must mindfully use high-overhead cache invalidation subroutines to maximize performance.

An application uses the DRE by passing in a source array and index array containing the memory access pattern. The DRE uses gather/scatter hardware to read DRAM and update the SRAM scratchpad buffer that holds the rearranged data with the specified view. The resulting structure from the DRE is a rearranged array that can be accessed sequentially with high spatial locality, thereby minimizing cache misses and permitting additional performance benefits from SIMD coprocessors. There are three functions available to the developer for use in their application:

The **setup** subroutine performs basic DRE initialization and parameter loading. The host passes the pointer to the source array and the stride pattern defined by the index array to the DRE

The **fill** subroutine gathers data into a contiguous view. The host passes the address of the output view block, the size of the entire view being coalesced, and the offset of the view (for use when the block size is less than the view size). The view size should be maximized to make full use of the block

and minimize calls to the fill subroutine. The DRE then gathers the view from DRAM to SRAM according to the access pattern specified in the setup subroutine.

The **drain** subroutine is similar to fill except it scatters data into a contiguous view on the DRAM from the SRAM according to the specified access pattern.

Our framework is set up on a Xilinx Zynq 7000 FPGA + ARM A9 SoC dev board. The application runs on the ARM host processor and the DRE is in the programmable logic.

# 3. ACCELERATING SPMV

The BeBOP [1] SpMV benchmark multiplies a sparse matrix in CSR format by a dense vector. The sparse matrix is already in optimal arrangement so rearranging this data structure using the DRE would not result in performance gain. The dense vector is accessed irregularly NNZ (number of non- zeros) times per row. For very large vectors this will not fit in cache and the application's performance will be memory latency bound–an ideal candidate for the DRE.

The major modification in SpMV comes from setting up the data to efficiently use the view buffer. A naive method is to rearrange each row of data separately. However, for sparse matrices where many rows contain only dozens (or fewer) of non-zero elements, the overhead associated with filling the view buffer is much greater than the memory latency overhead. To achieve performance gain, the rows must be batched together to fill the view buffer. The resulting view containing the coalesced values can then be read sequentially with high cache performance and low latency.

## 3.1 Overall Performance

Using the DRE with scatter/gather hardware, we see 1.4x speedup over the stock ARM A9 CPU on large matrices. Due to the limited 1GB DDR3 in the Zynq dev board, the density and size of the matrices studied was constrained. Unless otherwise specified, the following experiments were done on banded matrices of size 2M x 2M, a range of the cache is invalidated, and view buffer block size of 32KB.

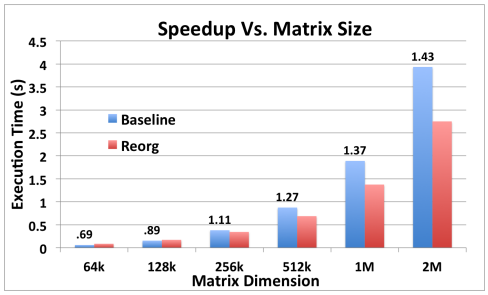


**Figure 2: Speedup as matrix size varied**

## 3.2 Sparse Matrix Density

Due to the overhead associated with the DRE and setting up the view, very sparse matrices performed slightly worse than higher density matrices.

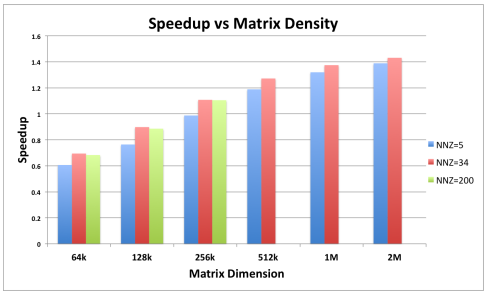


**Figure 3: Speedup as matrix density varied**

## 3.3 Invalidation Schemes

We studied two cache invalidation methods: Invalidating a range of addresses in the L1 cache and invaliding the entire L1 cache. When the view buffer block size exceeds 64 KB, invalidating the entire L1 cache was advantageous. For general cases, a range of addresses should be invalidated as the performance degradation is significant at smaller view buffer block sizes when invalidating the entire cache.

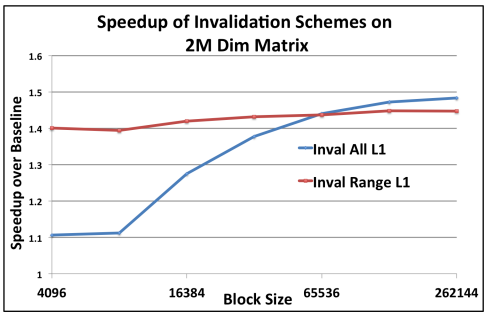


**Figure 4: Performance of cache invalidation schemes**

# 4. ACKNOWLEDGEMENTS

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

# 5. REFERENCES

[1] H. Gahvari, M. Hoemmen, J. Demmel, and K. Yelick. Benchmarking sparse matrix-vector multiply in five minutes. In *SPEC Benchmark Workshop (January 2007)*, 2007.

[2] Micron. Hybrid memory cube, 2015.