Optimizations for Sparse Matrix-Vector Multiply on a Migratory Thread Architecture

Thomas Rolinger[1,2]; Christopher Krieger[2]
[1]University of Maryland College Park, [2]Laboratory for Physical Sciences
tbrolin@cs.umd.edu, krieger@lps.umd.edu

Abstract

Sparse linear algebra kernels are a crucial component in many large-scale data analytic applications, such as tensor decomposition and graph analytics. Many of these kernels are comprised of Sparse Matrix-Vector Multiplication (SpMV), which is one of the fundamental operations in linear algebra. Achieving high performance for SpMV on today's cache-memory based systems is challenging due to irregular access patterns and weak locality. To address these challenges, novel systems such as the Emu architecture have been proposed. The Emu design uses light-weight migratory threads, narrow memory, and near-memory processing capabilities to address weak locality and reduce the total load on the memory system.

In this work, we evaluate the impact of traditional optimizations for SpMV on the Emu migratory thread architecture. Our goal is to gain insight into the cost-benefit tradeoffs of standard sparse algorithm optimizations on Emu hardware.

Emu Architecture

The basic building block of an Emu system is a *nodelet* which consists of the following:

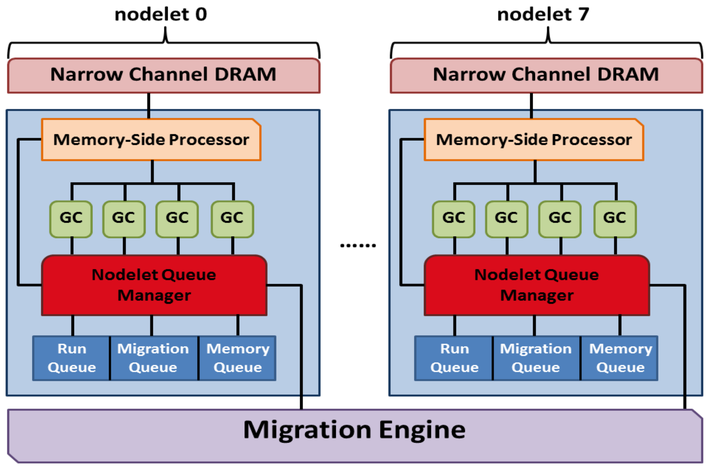
- **Gossamer Cores (GCs)***:* general purpose, cache-less processors that support up to 64 concurrent light-weight threads
- **Narrow Channel DRAM**: eight 8-bit channels rather than a single, wider 64-bit interface
- **Memory-side Processor**: performs atomic and remote operations

8 nodelets are combined together to make up a single node in the Emu architecture, as shown below.

When a thread on a GC makes a memory request to a remote address, a *migration* is generated. A migration involves:

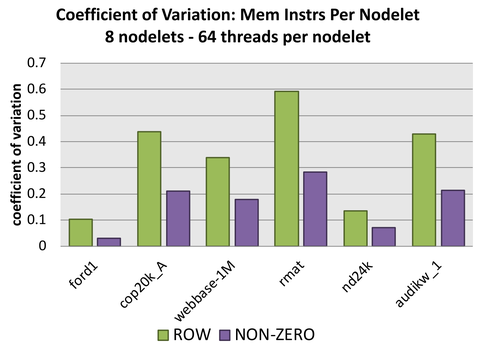
- A GC issuing a request to the **Nodelet Queue Manager** (NQM) to migrate the thread context to the nodelet where the desired data resides
- The thread contexts waits in the source nodelet's migration queue until it is accepted by the **Migration Engine** (ME), which is the communication fabric that connects multiple nodelets
- Once accepted, the thread context is sent over the ME and is processed by the destination nodelet's NQM.

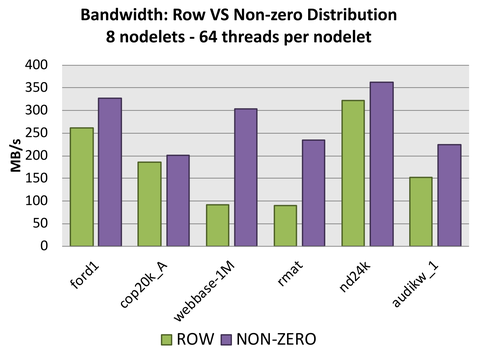
The size of a thread context is roughly 200 bytes.



Work Distribution Strategies

- **Row**: Evenly divide the rows amongst the nodelets
- **Non-zero**: Assign rows to such each nodelet receives roughly the same number of non-zeros.

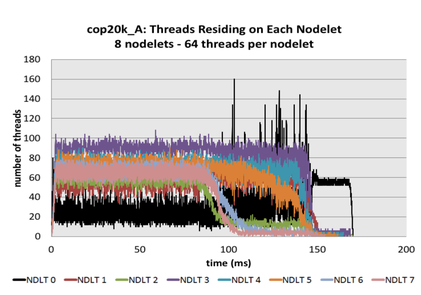


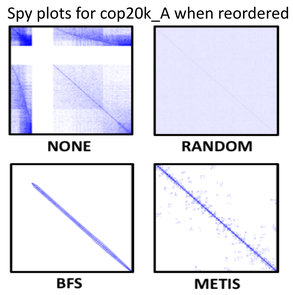


Results: Distributing by non-zero provides more uniform load balancing by enforcing each nodelet to issue a comparable amount of memory instructions. This leads to better overall performance for SpMV.

Load Balancing via Matrix Reordering

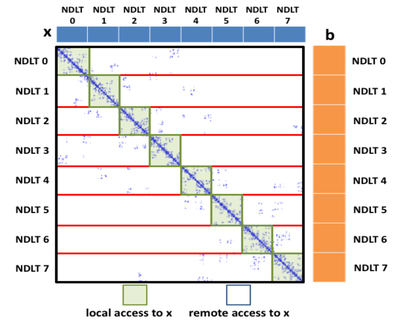
Despite efforts to lay out and distribute work evenly across the nodelets, all of the threads could migrate to a single nodelet and oversubscribe that nodelet's resources.

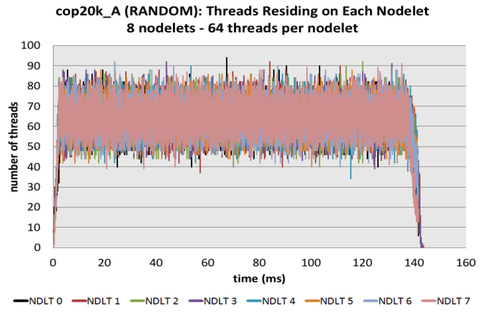


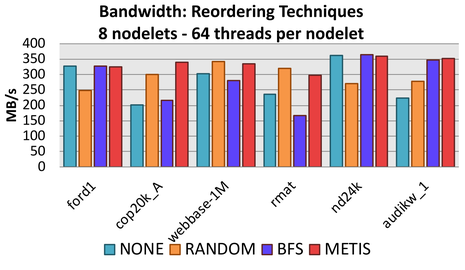


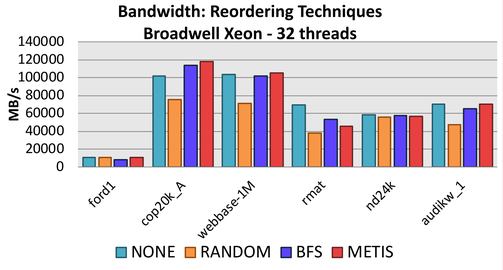
Known matrix reordering techniques can be used to encourage more consistent load balancing.

- **BFS and METIS:** cluster non-zeros on the main diagonal and produce balanced rows.
- **Random:** uniformly spreads out the non-zeros.





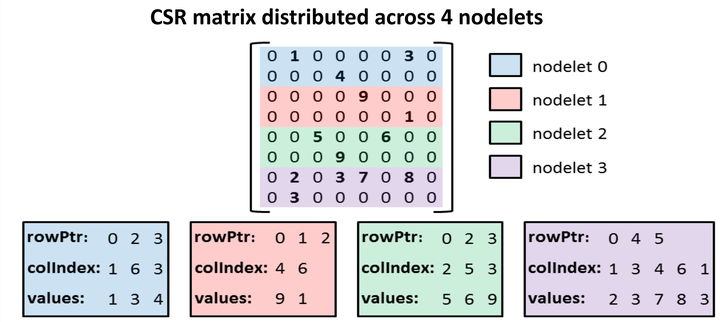




Results: Matrix reordering can improve SpMV performance on Emu by as much as 70% while a comparable SpMV implementation executed on a traditional architecture only receives at most a 16% improvement from matrix reordering.

Implementation

We leverage the **Compressed Sparse Row** (CSR) storage format to store sparse matrices, where blocks of rows are distributed to each nodelet. An example of this distribution is shown below.

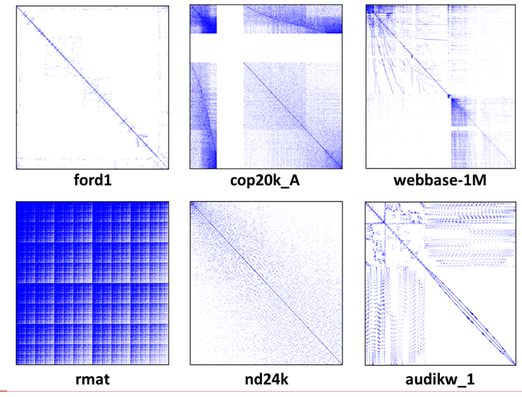


The optimizations we consider in this work are work distribution strategies and matrix reordering techniques.

Sparse Matrices Evaluated

Obtained from the University of Florida Sparse Matrix Collection. RMAT matrix generated with a = 0.45, b = 0.22 and c = 0.22 All matrices are square. "*" denotes non-symmetric matrices.

Name	Rows	Non-Zeros	Density
ford1	18K	100K	2.9×10^{-4}
cop20k_A	120K	2.6M	1.79×10^{-4}
webbase-1M*	1M	3.1M	3.11×10^{-6}
rmat*	445K	7.4M	3.74×10^{-5}
nd24k	72K	28.7M	5.54×10^{-3}
audikw_1	943K	77.6M	8.72×10^{-5}



Conclusions

- Work distribution and load balancing is of similar importance to reducing migrations in order to achieve high performance.
- Explicitly enforcing hardware load balancing for the Emu architecture is difficult due to thread migrations. Specifically, data placement and access patterns dictate the work performed by a given hardware resource and is irrespective of how much work is initially delegated to each processing element.
- Use of matrix reordering is more beneficial on the Emu architecture than a traditional cache-memory based system. We found that performance can be increased by as much as 70% on Emu while we observed a maximum gain of 16% on a traditional architecture. A random reordering can exhibit better performance on Emu than not reordering at all, which contradicts what we observe on a traditional system.