Evaluating Best and Worst Case Scenarios on Two-Level Memory Systems

Ryan J. Huber
University of Minnesota
Minneapolis, MN, USA
huber395@umn.edu

Edgar A. León
Lawrence Livermore National Laboratory
Livermore, CA, USA
leon@llnl.gov

I. Introduction

To achieve the capacity and bandwidth requirements of an exascale memory system, four TB/s per-node bandwidth and 128 PB total system capacity [1], vendors are resorting to multiple levels of memory: a small high-bandwidth memory close to the processor and a large but low-bandwidth memory, e.g., Intel Knights Landing (KNL) architecture [2]. The most significant challenge for an application is to use the memory system at the bandwidth of the small memory with the capacity of the large memory. One approach may involve application developers choosing data structures to place in fast memory. However, making these decisions manually requires detailed knowledge of an application and is impractical for large HPC production codes [3], [4].

Our long term goal is to study metrics for selecting good candidate objects to place in fast memory, relieving application developers from this task. In this poster, we start by quantifying the potential gains of leveraging the fast memory effectively and the potential negative effects of not doing so. This helps us understand the magnitude of the problem and provides a lower and upper bound on the performance of these emerging systems. We also evaluate a data placement policy guided by an application developer with detailed knowledge of his code. Furthermore, we are interested in evaluating the relative tradeoffs of a two-level memory (TLM) system using existing hardware with single-level memory. If the relative tradeoffs are similar to a real TLM system, application developers could use this infrastructure to understand the effect on their codes before porting them to these emerging systems.

II. Methodology

We used two systems for evaluation: an Intel KNL processor and a dual-socket Intel Sandy Bridge machine. The latter is used to emulate a TLM system and consists of 8 cores per socket, 2 hardware threads per core, and 32 GB of DRAM memory. We use the local memory of one socket as the fast memory and the memory on the opposite socket as the slow memory[1]. The KNL system has 64 cores, 4 hardware threads per core, 16 GB of on-chip high-bandwidth memory (MCDRAM), and 96 GB of off-chip DRAM. The cluster and memory modes used were *quadrant* and *flat*, respectively.

[1]https://github.com/memkind/memkind

To control allocation of memory, we used the Linux NUMA utilities and the Intel memkind[1] library. To quantify the memory bandwidth ratio between fast and slow memory we executed the STREAM benchmark [5] and summarize the results in Table I. There is a large memory bandwidth differential of $4.8X$ on the KNL and a modest $2.6X$ differential for the dual-socket system.

TABLE I
Memory Bandwidth Measured with STREAM.

	Memory	**Far socket**	**Local socket**
Dual socket	Copy (GB/s)	10.75	28.02
	Scale (GB/s)	10.74	28.13
	Add (GB/s)	12.21	31.61
	Triad (GB/s)	12.23	31.76
	Normalized over slow memory	$\mathbf{1}\times$	$\mathbf{2.60}\times$
	Memory	**Off chip**	**On chip**
Intel KNL	Copy (GB/s)	50.78	206.39
	Scale (GB/s)	50.75	256.63
	Add (GB/s)	57.04	279.09
	Triad (GB/s)	57.09	295.17
	Normalized over slow memory	$\mathbf{1}\times$	$\mathbf{4.81}\times$

For our experiments we used LULESH, an MPI+OpenMP hydrodynamics proxy-application [6]. LULESH provides a simplified source code that contains the data access patterns and computational characteristics of larger hydrodynamics codes. It uses an unstructured hexahedral mesh with two centerings and solves the Sedov problem. We ran LULESH with one MPI task and as many OpenMP threads as CPU cores were present on each system (8 and 64).

Execution time was measured under three data placement policies. Two of these are *best-case* and *worst-case* scenarios, where all data is allocated to fast memory and all data is allocated to slow memory, respectively. These policies represent upper and lower bounds on the application's runtime for a given architecture. For our final policy, we used the recommendations of a developer with extensive knowledge of LULESH to selectively allocate data objects into the fast memory and everything else into slow memory [6]. Note that the entire data footprint for a proxy like LULESH fits into the fast memory, but this may not be the case for larger or multi-physics applications, which motivates selectively allocating data into fast memory.

III. Results

Figure 1 shows LULESH's overall execution time. The best-case policy achieves a 32% speedup on the dual-socket configuration and a 43% speedup on the KNL relative to the worst case. The different speedups between the two architectures is similar when taking into consideration the fast-slow memory bandwidth ratios shown in Table I ($2.6X$ and $4.8X$). Leveraging the high-bandwidth memory effectively is imperative to achieve good performance on these systems. We also included the performance of the KNL configured in the *cache* memory mode (fast memory as a hardware cache of the slow memory). In this mode performance approaches that of the best case. This is encouraging since this mode does not require changes to the application. However, this positive result is due to the regular patterns of LULESH and its small working set size. For a full-fledged application, we expect more contention and evictions in cache mode.

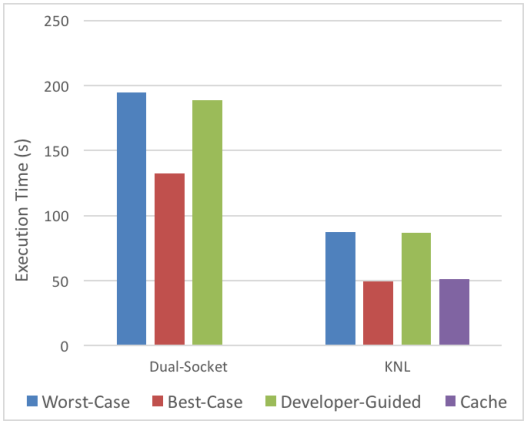

Fig. 1. Overall LULESH execution time for different allocation policies.

Surprisingly, the developer-guided policy only improves performance by 3% on the dual-socket and less than 1% on the KNL. To better understand this behavior, we measured the performance of LULESH broken down into five code regions that account for over 90% of its execution time [7]. Figure 2 illustrates the speedup of these regions relative to the worst-case. For Region 3, the developer-guided policy performs well, which may be explained by its high memory bandwidth requirements. However, on the KNL Regions 1 and 2 show a significant speedup when placing all the data on the high-bandwidth memory but the developer-guided policy performs poorly. We are now correlating these results with performance counters to explain this phenomenon. These results though show one data point where detailed application knowledge may not be enough to effectively leverage the fast memory. We expect that other applications may provide more insight to understand the effectiveness of developer-guided policies.

IV. Conclusions

This work indicates that leveraging the fast memory effectively is key to achieve reasonable application performance on TLM systems, particularly for HPC codes that impose

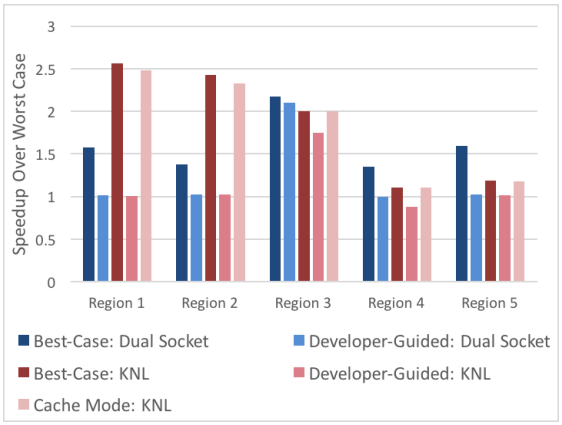

Fig. 2. Average speedup by code region, normalized to the worst-case policy.

significant pressure on memory bandwidth and whose data set will not fit in fast memory. Our best-case and worst-case scenarios establish an empirical maximum performance gain for the KNL and emulation system. We observed that emulation with a dual-socket architecture can provide useful insight to understand performance tradeoffs on a TLM system but it is important to account for the fast to slow memory bandwidth ratio of each system.

Finally, this work raises the question whether selecting data structures based on developer knowledge may or may not be enough to utilize the fast memory effectively. This motivates future work in semi-automatic metric-based allocation of data objects into fast memory and evaluation of several applications to better understand the effectiveness of developer-guided policies. We also plan to evaluate CPU+GPU based systems.

Acknowledgments

We would like to thank Alfredo Gimenez, David Poliakoff, Ian Karlin, and Matthew Legendre from LLNL for their assistance with this work. Prepared by LLNL under Contract DE-AC52-07NA27344. LLNL-ABS-698633.

References

[1] "Exascale research and development: Request for information," U.S. Department of Energy, Tech. Rep. RFI 1-KD73-I-31583-00, Jul. 2011.

[2] A. Sodani, "Knights Landing (KNL): 2nd generation Intel Xeon Phi processor," in *Hot Chips: A Symposium on High Performance Chips*. Cupertino, CA: IEEE, Aug. 2015.

[3] M. R. Meswani, G. H. Loh, S. Blagodurov, D. Roberts, J. Slice, and M. Ignatowski, "Toward efficient programmer-managed two-level memory hierarchies in exascale computers," in *Hardware-Software Co-Design for High Performance Computing*, ser. Co-HPC'14, Nov 2014.

[4] C. Su, E. A. León, G. Loh, D. Roberts, K. W. Cameron, D. S. Nikolopoulos, and B. R. de Supinski, "HpMC: An energy-aware management system of multi-level memory architectures," in *International Symposium on Memory Systems*, ser. MEMSYS'15, Oct. 2015.

[5] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[6] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[7] E. A. León, I. Karlin, and R. E. Grant, "Optimizing explicit hydrodynamics for power, energy, and performance," in *International Conference on Cluster Computing*, ser. Cluster'15. Chicago, IL: IEEE, Sep. 2015.