Computing with Near Data

Xulong Tang
Pennsylvania State University
State College, PA, USA
xzt102@psu.edu

Mahmut Taylan Kandemir
Pennsylvania State University
State College, PA, USA
mtk2@psu.edu

Hui Zhao
University of North Texas
Denton, TX, USA
hui.zhao@unt.edu

Myoungsoo Jung
Yonsei University
Seoul, SOUTH KOREA
m.jung@yonsei.ac.kr

Mustafa Karakoy
TOBB University of Economics and
Technology
Ankara, TURKEY
m.karakoy@yahoo.co.uk

ABSTRACT

The cost of moving data between compute elements and storage elements plays a significant role in shaping the overall performance of applications. We present a compiler-driven approach to reducing data movement costs. Our approach, referred to as Computing with Near Data (CND), is built upon a concept called “recomputation”, in which a costly data access is replaced by a few less costly data accesses plus some extra computation, if the cumulative cost of the latter is less than that of the costly data access. Experimental result reveals that i) the average recomputability across our benchmarks is 51.1%, ii) our compiler-driven strategy is able to exploit 79.3% of the recomputation opportunities presented by our workloads, and iii) our enhancements increase the value of the recomputability metric significantly.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**;

KEYWORDS

Manycore systems; data movement; recomputation

ACM Reference Format:

Xulong Tang, Mahmut Taylan Kandemir, Hui Zhao, Myoungsoo Jung, and Mustafa Karakoy. 2019. Computing with Near Data. In *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '19 Abstracts)*, June 24–28, 2019, Phoenix, AZ, USA. ACM, New York, NY, USA, 2 pages. <https://doi.org/10.1145/3309697.3331487>

1 INTRODUCTION

The relative cost of a data access is continuously increasing compared to the relative cost of a computation. One can expect the situation to be even worse as big data becomes more prevalent. Motivated by this, more recently, a new computation paradigm called “near data computing” (NDC) has emerged [1, 4–6, 8]. The main idea

behind NDC is to perform computation in or around where data currently is, that is, moving computation to data, instead of moving data to computation (as in the traditional computing paradigm).

In this paper, we take a different look at the growing costs of data accesses, and propose “computing with near data”, CND, as another way of reducing data access and movement latencies. As opposed to NDC, our CND does *not* require any hardware modification, and instead employs a software-based concept called *recomputation*. In recomputation, a costly data access is replaced by a few less costly data accesses plus some extra computation, if the cumulative cost of the latter is less than that of the costly data access. An example would be recomputing the value of a data element that resides in off-chip memory using data elements that are available in on-chip caches, instead of directly performing the costly off-chip access. Thus, if implemented carefully, CND can successfully trade off data access with computation, and considering the continuously increasing latency gap between the two, doing so can significantly reduce the execution latencies of both sequential and parallel application programs.

Targeting emerging manycore systems and using a diverse set of single-threaded and multi-threaded workloads, this paper makes four main contributions:

- It defines a new metric called *recomputability* which captures the possibility of recomputing the value of a variable. It then presents a detailed analysis of various application workloads from a recomputability perspective. This experimental analysis also reveals the “limits” of recomputability if no extra effort is made to improve it. The collected experimental results indicate that the average recomputability across our benchmarks is about 51.1%.
- It presents a practical approach to computing with near data (CND) that employs recomputation in a “data locality-aware” fashion. This approach relies on compiler for both code analysis and code transformation, e.g., replacing a data reference (access) with its “recomputed version”. Our experimental data indicate that the proposed compiler-based approach is able to exploit 79.3% of the recomputation opportunities presented by our workloads. The paper also discusses the reasons behind the untapped potential.
- It presents a technique to improve the value of the recomputability metric. This technique is based on the duplication/copy of a variable’s value that is critical for recomputation. Our experimental evaluation of this technique shows that it increases the value of the recomputability metric significantly. As a result, now,

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SIGMETRICS '19 Abstracts, June 24–28, 2019, Phoenix, AZ, USA

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-6678-6/19/06.

<https://doi.org/10.1145/3309697.3331487>

our compiler-based approach brings an average execution time improvement of 40.1%.

- It compares CND against three “ideal” incarnations of NDC, and shows that CND outperforms all three NDC approaches for all the benchmark programs tested.

2 HIGH LEVEL VIEW OF CND

Figure 1: (a) Traditional computing vs (b) near data computing (NDC) vs (c) computing with near data (CND).

In this section, we contrast CND with NDC and explain the similarities and differences between the two. We chose NDC for this comparison, as it is a relatively recent paradigm and, like CND, it also tries to reduce data access latencies and data movement costs. A conceptual comparison of the traditional mode of computation, NDC, and CND is given in Figure 1, assuming that a computation in the core shown originally wants to access data from far memory locations. In this figure, (a) illustrates the traditional computing paradigm, where a computation scheduled on a core that needs a data (say, with a long access latency) issues a request for it and the data is returned, from its far memory location, to the requesting core. This traditional option can be summarized as one in which we bring data to (the core that is to perform) computation. Clearly, in this option, we incur the full cost of the long-latency data access. NDC is illustrated in (b); it has several incarnations such as [2, 3, 7, 9]. NDC works by bringing computation to data (usually through static scheduling), as opposed to bringing data to computation, which was the case in option in (a). Consequently, the distance between computation and data gets reduced, which in turn reduces data access latencies. In comparison, our proposed CND, illustrated in (c), eliminates the costly data access altogether and replaces it with recomputation, which involves a number of accesses to less costly data (in nearby memory locations) and some extra computation.

3 EVALUATION

The first bar in Figure 2 plots the execution time of the best version (as defined above) of each application program tested. For each application, the execution time is *normalized* with respect to the execution time of the “original version” of that application (the latter is set to 100). In the same graph, the second bar for each application gives the result of the most aggressively recomputed version. As can be observed, using aggressive recomputation can lead to disastrous results for many benchmarks. For each application, the third bar in Figure 2 gives the normalized execution time for the *compiler-optimized* version. It can be seen that, the compiler version generates performance improvements, ranging between 0.9% and 66.9%, averaging on 36.9%. The results given as the last bar for each benchmark in Figure 2 indicate that, this optimization improves performance by 40.1% on average, and as much as 67.4% and 65.8% in benchmarks *soplex* and *calculix*.

Figure 2: The normalized execution time of the best version of each application program (lower, better).

4 CONCLUSIONS AND FUTURE WORK

This work makes four main contributions. First, targeting emerging manycore systems, it proposes a novel computation paradigm called CND (which is based on recomputing the values of costly data elements), and quantifies its potential using a diverse set of sequential and parallel application programs. Second, it presents a practical compiler-driven strategy to take advantage of CND. Third, it proposes a strategy to increase the chances for employing CND. And finally, it compares CND with another recent paradigm, near data computing (NDC), in both qualitative and quantitative terms. Our experimental evaluations on a state-of-the-art many-core system reveal that CND can be very effective in reducing data access/movement costs, and improves, on average, the execution time by 40.1% across our programs including both single-threaded and multi-threaded applications. Further, our simulation based experiments indicate that CND outperforms various (idealistic) implementations of NDC.

ACKNOWLEDGMENT

We thank Evgenia Smirni for shepherding our paper. We also thank the anonymous reviewers for their constructive feedback. Myoungsoo Jung is in part supported by NRF 2016R1C1B2015312, DOE DEAC02-05CH11231, IITP-2018-2017-0-01015, NRF 2015M3C4A7065645, Yonsei Future Research Grant (2017-22-0105) and Samsung grant (2018). This research is supported in part by NSF grants #1526750, #1763681, #1439057, #1439021, #1629129, #1409095, #1626251, #1629915, and a grant from Intel.

REFERENCES

- [1] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoun Choi. 2015. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *ISCA*.
- [2] Reetuparna Das, Rachata Ausavarungrun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. 2012. Application-to-core Mapping Policies to Reduce Memory Interference in Multi-core Systems. In *PACT*.
- [3] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in Memory: the Terasys Massively Parallel PIM Array. *IEEE Computer* (1995).
- [4] Orhan Kislal, Jagadish Kotra, Xulong Tang, Mahmut Taylan Kandemir, and Myoungsoo Jung. 2018. Enhancing Computation-to-core Assignment with Physical Location Information. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
- [5] Gabriel H. Loh. 2008. 3D-Stacked Memory Architectures for Multi-core Processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*.
- [6] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. 2016. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *PACT*.
- [7] Erik Riedel, Christos Faloutsos, and David Nagle. 2000. *Active disk architecture for databases*. Technical Report. DTIC Document.
- [8] Jihyun Ryoo, Orhan Kislal, Xulong Tang, and Mahmut Taylan Kandemir. 2018. Quantifying and Optimizing Data Access Parallelism on Manycores. In *MASCOTS*.
- [9] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. 2017. Data Movement Aware Computation Partitioning. In *MICRO*.