Compiler Support for Near-Cache Computing in Kobold

COLIN MCDONALD AND JENNIFER BRANA OCTOBER 2023
colinmcd@cs.cmu.edu, jbrana@cs.cmu.edu
GitHub: https://github.com/JenniferBrana/15745-project.git

1 URL

https://jenniferbrana.github.io/15745-project/

2 Project Description

Near-cache computing (NCC) is a recent paradigm which offloads tasks to programmable accelerators inside the cache hierarchy to avoid the costly transfer of data to the core. Unfortunately, existing NCC systems are targeted at expert programmers with deep knowledge of both the application and hardware [1]. The primary goal of this project is to enable transparent offload by building a compiler which automatically extracts tasks for execution on near-cache engines.

In the Kobold system, programmers manually specify regions of code for offload to engines, requiring them to reason about sending work to the cache hierarchy. For example, performing a reduce operation at the near-cache engine requires the programmer to write a function specifying the reduce operation to be performed at the end and, from within the main code, use the Kobold API to pass a pointer to the function along with data to the engine. Additionally, where (locally on the main core or on a near-cache engine) to execute operations such as reduction is highly dependent on the data size and cache size as well as the reuse characteristics of the data. For example, if a reduction is performed over a large set of data which does not fit in the L1D and is never reused, offloading to the engine ensures the cores working set is not evicted by data used in the reduction.

Given these challenges, we plan to implement a compiler which can identify candidate code regions for offload and determine candidates which will benefit from being executed on a near-cache engine. We will generate code which offload selected candidates to a and compare against baseline benchmarks without offload. Specifically, our compiler will target reduction operations which have affine, indirect, or pointer chasing memory patterns, similar to previous work [4].

We will evaluate our compiler on a set of benchmarks from [4] which perform many reduction operations.

75 percent goal: Implement compiler phases to recognize code region candidates and select candidates qualified for offloading. Implement pass to detect if there is aliasing within the callback.

100 percent goal: Generate code to perform callbacks for qualified code regions.

125 percent goal: Write code to statically detect if a callbacks memory footprint (inferred from the memory access pattern and length) cannot fit into the private cache. Implement a static method to detect if data from callback will be reused. We will use these two methods to determine whether to offload a callback to the engine or execute it locally.

3 Plan of Attack and Schedule

Week 1: Literature review. Review existing work on near-cache acceleration and compiling for near-cache accelerators.

Week 2: Implement compiler phase to recognize code region candidates.
Week 3: Implement compiler phase to select code region candidates with memory access patterns qualified for offloading.
Week 4: Implement compiler pass to detect aliasing loads and stores in candidate code regions (aliasing disqualifies code regions).
Week 5: Generate code to perform callbacks for qualified code regions.
Week 6: Test code with automatically generated callbacks vs baseline benchmarks in gem5. Work on poster.

4 Milestone

By November 20th we hope to have compiler phases complete to recognize code regions that contain reductions and select candidates qualified for offloading.

5 Literature Search

We have literature on near-cache computing [2, 1] and literature on stream-based memory optimizations which includes sections on compiler optimizations [3, 5].

6 Resources Needed

We will use the gem5 simulator to simulate near-cache accelerators. Jennifer has previously implemented user-level interrupts in gem5 which allows processor cores to send lightweight interrupts to "near-cache engines" which trigger the engines to execute a callback. This system allows us to simulate near-cache accelerators as desired. We will use LLVM for implementing the compiler transformations.

7 Getting Started

We have started our literature search.

References

[1] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. Livia: Data-centric computing throughout the memory hierarchy.

[2] Brian C. Schwedock, Piratach Yoovidhya, Jennifer Seibert, and Nathan Beckmann. täkō: A polymorphic cache hierarchy for general-purpose optimization of data movement.

[3] Zhengrong Wang and Tony Nowatzki. Stream-based memory access specialization for general purpose processors.

[4] Zhengrong Wang, Jian Weng, Sihao Liu, and Tony Nowatzki. Near-stream computing: General and transparent near-cache acceleration.

[5] Zhengrong Wang, Jian Weng, Jason Lowe-Power, Jayesh Gaur, and Tony Nowatzki. Stream floating: Enabling proactive and decentralized cache optimizations.