Compiler Support for Near-Cache Computing in Kobold

Colin McDonald Carnegie Mellon University
colinmcd@cs.cmu.edu
Jennifer Brana Carnegie Mellon University
jbrana@cs.cmu.edu

I. Introduction

A. Background

Computer systems are increasingly bottlenecked by the rising cost of data movement and communication [4], [5], [6], [8]. To address data movement, near-data computing (NDC) has been proposed to add compute resources within memory hierarchies to move compute closer to data and avoid costly transfers of data. *Near-cache computing (NCC)* is a recent paradigm of NDC which enables fine-grain collaboration between cores and accelerators by offloading work to accelerators within the cache hierarchy [2], [3], [9], [12], [16], [17].

NCC enables computing near-data while allowing applications to fully exploit locality and allows fast communication between processors and accelerators [10]. NCC can accelerate operations that typically result in excessive data movement. For example, if data doesn't fit into a core's cache or an object, such as a synchronization variable, is highly contended, a lot of execution time is spent shuffling data in the cache hierarchy. By placing accelerators near caches, operations can be performed at whichever level of the hierarchy where a dataset fits best and, similarly, all operations on a synchronization variable can be performed at a common location in the shared cache, significantly reducing data movement.

Figure 1 shows Kobold [1], a near-cache computing system modeled after the architecture of täkō [14]. Kobold provides a task-based programming model similar to [10] that expresses applications as series of tasks which can be scheduled for execution on cores or near-cache engines. Kobold augments a baseline, cache-coherent multicore with an programmable engine (i.e., accelerator[1]) on each tile, allowing the engine to efficiently compute on data in the tile's L2 and LLC banks. Dataflow engines enable offloading arbitrarily large functions and are ideal for executing short, repetitive operations efficiently with support for data-parallel operations [14].

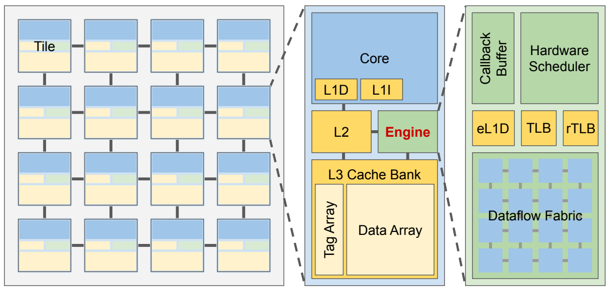

Fig. 1. Kobold [1] adds a reconfigurable engine to each tile of a CMP. Engines accelerate tasks for data that resides in the L2 or L3 bank on that tile. Each engine has a coherent eL1D cache.

To achieve general-purpose applicability, NCC systems should enable transparently offloading arbitrarily large operations to near-data. Placing reprogrammable engines near caches provides the ability to offload arbitrary operations to the cache hierarchy but presents a challenge to the programmer. NCC systems are notoriously difficult to program as they require programmers to reason about what to offload and to tailor applications to a specific architecture. This requires expert knowledge of the underlying architecture and limits code portability.

B. Our Approach

Determining *what to offload* to near-cache engines is important to exploit the potential of near-cache computing. Applications bottlenecked by data access often involve patterns of loading data, performing a short computation, then loading more

[1]We use the terms "accelerator" and "engine" interchangeably.

data. Therefore, we identify identifying this pattern within the compiler as an opportunity to enable transparent offload.

To address the programming challenges of prior NCC systems, we propose a compiler which detects reductions operations performed over a large data structure and generates code to offload such patterns to near-cache engines. Similar to prior work [18], we identify streams, or coarse-grain memory access patterns, as the ideal granularity for transparent offload. Streams are capable of capturing long-term per-data structure behavior, allowing all operations to a data structure to be offloaded together to reduce data movement [18]. We chose to offload reduction operations as they are common in many applications [18] and they cannot exploit the L1D cache (i.e. they are performed over large data sets that do not fit in the L1 cache or their data has long reuse distances that make the L1D ineffective) and therefore can benefit from near-cache execution. Additionally, the short and repetitive nature of reductions make them amenable to execution on dataflow engines [14].

Our project investigates the requirements to extract code regions for execution on near-cache engines. We present a method capable of extracting tasks for offload near-cache in using Kobold's task-based programming model. We detect common streaming access patterns, including affine, indirect, and pointer-chasing. We focus on offloading a limited subset of operations, reductions, which can benefit from near-cache acceleration and present mechanisms that can be applied to detect further operations.

C. Related Work

Past work has proposed placing reprogrammable engines in the cache hierarchy to provide the ability to execute operations near-data in the caches [1], [10], [14]. Livia [10] and Kobold [1] require programmers to break applications into short, simple tasks which are associated with memory locations. Livia transparently schedules tasks to execute where data lives in the memory hierarchy. These system require programmers to identify opportunities where near-cache acceleration can improve performance. Identification requires significant programmer effort along with a detailed understanding of the hardware tradeoffs between CPU cores and near-cache engines.

Near-data computing has been proposed for GPU architectures in which operations can be offloaded to execute near the LLC or to logic layers in DRAM. [13] presents a compiler which detects basic sequences of load-compute-store instructions for offload near-data to accelerators in the LLC. Their compiler identifies nine common instructions patterns beginning with a load and ending with a store or ALU operation that are amendable to offload. However, they provide limited support for offloading operations in applications with high control flow or irregular memory access patterns. [7] performs a statically identifies code blocks with maximum potential memory bandwidth savings from offloading to the near-data compute units in DRAM and decides whether to offload based on dynamic conditions of the system. Offload is performed at the block granularity, preventing the exploitation of long-term per-data structure behavior.

[18] is a recent near-cache system that uses streams as the abstraction for near-data offload. Programs express coarse-grain memory access patterns as streams for offload near caches and can associate limited operations with streams to be performed near-cache as data is streamed in. Last-level cache banks are augmented with stream engines which are essentially programmable prefetchers. Compute can be performed using a scalar PE (for simple computations). However, many workloads are data-parallel and must be executed on the local core, incurring significant overhead to spawn core threads. Compiler support is provided for detecting combinations of stream memory access patterns (affine, indirect, multi-operand) and computation types (loads, stores, reductions, atomics). Near-stream computing is limited due to its inability to execute data-parallel or complex operations near-data in the cache hierarchy. Additionally, its programming model restricts its ability to offload operations to near-data as computations are tied to streaming memory access patterns, preventing the offload of operations independent from memory streams such as remote-memory operations [15]. Kobold supports executing arbitrarily large functions near-data and is not tied to memory access pattern, enabling it to accelerate a wider variety of workloads without falling back to the core to perform complex operations.

D. Contributions

To address the programmability challenges of near-cache computing in Kobold, we present a methodology to extract tasks for offload near-data transparently to the programmer. Our compiler recognizes affine and pointer-chasing memory access patterns as candidates for offload. We extract tasks that execute reductions over these memory access patterns and generate code to automatically offload these tasks to engines within the cache hierarchy. Importantly, this is all done entirely transparently to the programmer. We perform demonstrate the functional correctness of our compiler by generating near-cache code for a suite of representative microbenchmarks. Furthermore, we perform a limited performance study on two representative benchmarks and find that offloading reduction operations over large data structures to near-cache compute units can yield performance improvements of 4x to 9.5x.

II. Design

The Kobold programming interface allows cores to offload tasks to near-cache engines. Our compiler enables transparent offload by automatically extracting tasks to be offloaded on near-cache engines. It does so by first analyzing programs to select code regions for offload, then extract those regions and packaging them as a task for execution on the near-cache engine, and finally replacing the extracted code with a Kobold API function call to sends the task to the engine. Below we detail the steps involved in this process.

A. Offload Task Selection

Using LLVM, we implement a pass that, for each loop, determines if it qualifies for offloading with several criteria:

- Single entry, single exit: in order for our offloading pass to work, the loop must have a single entry point and a single exit point. By preprocessing loops with the loop-rotate pass, LLVM inserts a preheader that is the single entry point for every loop. Our pass must then check that the loop has only one exit (i.e. no `break` or `goto` statements).
- Has induction variable(s): the loop must have an induction variable that changes with each loop iteration (e.g. `i++`). We check this by examining the basic block in the loop with a back edge to the loop header; in particular, we check that this block conditionally jumps depending on at least one such induction variable.
- Has reduction variable(s): the loop must also have some variable that it updates to with each loop iteration (e.g. `total += ...`). The difference between a reduction variable and an induction variable is that the loop condition does not depend on reduction variables. A qualifying loop must have at least one of each type of variable.
- Is not contained within other loops: we only offload the outermost loop if there are nested. Future work could include implementing offloading of nested loops, but this is beyond the scope of our project.
- Loop has either an affine or a pointer-chasing data access pattern, as shown in Figure 2 and described below.

We only offloads loops that satisfy all these criteria.

A loop is **affine** if it accesses into an array with the form

`r` = ... `r` ... `a[c*i+k]` ...

And updates `i` with the form

$$\texttt{i} = \texttt{i} \pm \texttt{b}$$

Where `i` is an induction variable and `r` is a reduction variable. That is, the reduction variable must be updated by the loop body to some expression containing both itself and an array access based off the induction variable.

A loop is **pointer-chasing** if it has an induction variable `i` that is updated with the form

`i = i->fieldA` ...

and updates the reduction variable with the form

`r` = ... `r` ... `i->fieldB` ...

This kind of loop represents a pointer-chasing data access pattern, where each iteration of the loop aggregates some data into the reduction variable and updates the induction variable to one of its own fields that is a pointer.

B. Offload Task Code Generation

To offload selected code regions to the near-cache engine, the compiler must generate code to setup

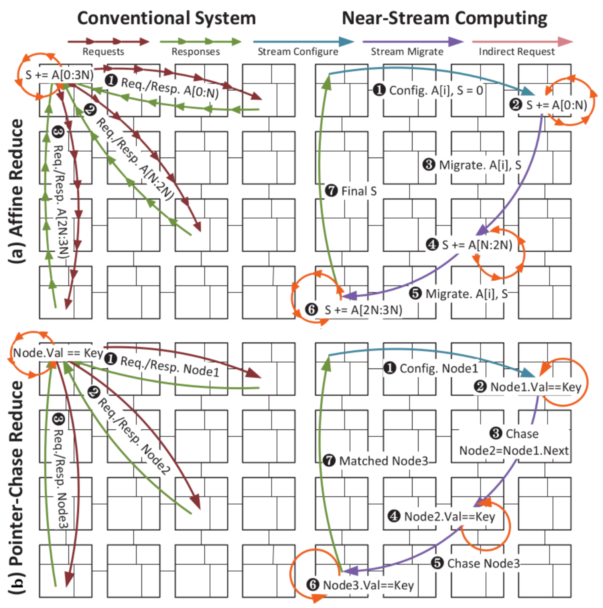

Fig. 2. Data Access Patterns. Figure adapted from [18].

```
int sum(int xlen, int xs[]) {
    int total = 0;
    for (int i = 0; i < xlen; i++) {
        total += xs[i];
    }
    return total;
}
```

Fig. 3. Example loop. `xs`, `total`, and `xlen` are upwards-exposed. `total` is also downwards-exposed.

necessary pointers and data structures and insert calls to the Kobold API which send requests to the engine. The Kobold API allows cores to offload tasks to the near-cache engine by passing a function pointer and a pointer to a data structure containing the function parameters and all necessary variables, as determined by the analysis pass *exposed variables*.

1) Analysis Pass: Exposed Variables: We define the iterative dataflow analysis pass *exposed variables* as follows:

- Direction: backwards
- Domain: variables
- Transfer function:
  $\text{in}[B] = (\text{out}[B] - \text{kill}[B]) \cup \text{gen}[B]$
- Meet operator ($\wedge$): union ($\cup$)
- Boundary condition (exit): $\varnothing$
- Initial inter points (out): $\varnothing$

Then let E be the loop entry basic block, and E' the loop exiting block (i.e. the block in the loop that exits). We can compute the upwards- and downwards-exposed variables as follows:

$$up = \text{in}[E] \ \cap \ \{v \mid v \text{ is defined outside loop}\}$$

$$down = \bigcup_{\text{loop succ. } B} \text{in}[B] \cap \{v \mid v \text{ is defined in loop}\}$$

Upwards-exposed variables are those whose prior values the loop depends on, and downwards-exposed variables are those whose values are determined by the loop execution and on which the remaining program depends. Consequently, our optimization stores upwards-exposed variables in a data structure to be passed to the function that runs the offloaded loop, and that function returns a data structure containing the values of the downwards-exposed variables back to the main thread.

After our pass identifies the for-loop in Figure 3 as a candidate for offloading, it transforms it into the code in Figure 4.

```
struct Data {
    int up_xs[];
    int up_total;
    int up_xlen;
    int down_total;
}

void engine_function(struct Data data) {
    int xs[] = data.up_xs;
    int total = data.up_total;
    int xlen = data.up_xlen;
    // BEGIN ORIGINAL LOOP
    for (int i = 0; i < xlen; i++) {
        total += xs[i];
    }
    // END ORIGINAL LOOP
    data.down_total = total;
}

int sum(int xlen, int xs[]) {
    int total = 0;
    // BEGIN GENERATED
    struct Data data;
    data.up_xs = xs;
    data.up_total = total;
    data.up_xlen = xlen;
    uli_send_req_fx_addr_data(1,
        engine_function, &data);
    total = data.down_total;
    // END GENERATED
    return total;
}
```

Fig. 4. Transformed code from Figure 3, where the for-loop has been offloaded to `engine_function`.

2) Kobold interface code: Task sending

Engine initialization Engines must be initialized at the beginning of every application. To do so, we must insert a call to pthread_create which is used to spawn an engine thread on every engine in the system and initialize the engine thread. During code generation, if the compiler has detected candidates for offload to near-cache engines, we insert the necessary code to initialize engine threads at the beginning of the application.

Engine handler function When a engine receive tasks from cores, the engine must call a handler function to access information from the task. The engine handler function is called when a task is received by the engine. Within the handler, the engine unpackages the task, recasts the function pointer sent with the task, and calls the function while passing it any necessary parameters sent in the task. During code generation, we must insert a header file into the original application to provide access to the engine handler function.

III. Experimental Setup

We evaluate of method in the Kobold system. Each tile in a chip multiprocessor is augmented with a near-cache engine. Tiles also contain a core, private L1D cache, private L2 cache, a slice of the shared last-level cache, and a engine L1D cache. Cache coherence is maintained using the Kobold coherence protocol, a protocol optimized for near-cache accelerators [1]. Cores can communicate directly with any engines through user-level interrupts (ULI), a light-weight hardware mechanism that enables cores to send tasks to near-cache engines. ULIs can package a pointer to a function for the engine to execute and a pointer to a data structure to pass to the function as a parameter. Near-cache engines can directly communicate results to cores as responses to ULIs or by passing results through shared memory. A back-pressure mechanism ensures cores only send tasks when engines have available capacity.

gem5 details We model the Kobold system using the gem5 cycle accurate simulator in syscall emulation mode [11]. The inter-processor ULI mechanism is implemented as specified by the RISC-V ISA [19]. The ULI network is modeled as an independent network with a 1 cycle channel latency. Each core is augmented with a unit to send ULIs and ULIs are sent by writing to a control register. Each core and engine has a control register to buffer incoming requests. Responses received from the ULI network are also buffered in a control register. We implement backpressure by limiting request and response buffering and issuing backpressure when buffers become full. Cores are modeled using an in-order core in gem5. Engines are presently modeled using an in-order core with modified statistics to emulate the performance of a dataflow engine. Engines and cores have 16kB L1D and L1I caches. Each core has a 128kB private L2 cache and each tile contains a 512kB bank of the shared L3 cache. Engines and cores use the RISC-V ISA.

IV. Experimental Evaluation

We verify the correctness of our compiler and evaluate its performance on representative benchmarks to demonstrate the potential of transparent offload for near-cache computing systems.

To demonstrate functional correctness of our compiler analysis and code generation, we run our compiler on a suite of microbenchmarks. We evaluate microbenchmarks that include a mix of pointer-chasing and affine memory access patterns which perform reduction operations. We demonstrate our compilers ability to extract multiple tasks for offload from a single applications by evaluating on microbenchmarks that contain multiple pointer-chasing and affine reductions. We compare the output results of our generated code against the unmodified, baseline benchmarks and verify that the output produced is correct.

We evaluate the performance of our compiler generated code by gathering performance results for two key benchmarks. The first benchmark we evaluate, index sum, performs an affine reduction over a large array of data. The second benchmark we evaluate performs a reduction over the elements in a large linked list. In both cases, we achieve significant speedups ranging from 4X to 9.5X speedup, as shown in Figure 5.

V. Surprises and Lessons Learned

There were many surprises, setbacks, and lessons learned during this project. We initially experience some difficulty identifying pointer-chasing loops on account of having to do so through the LLVM IR, but eventually worked out that we can just search

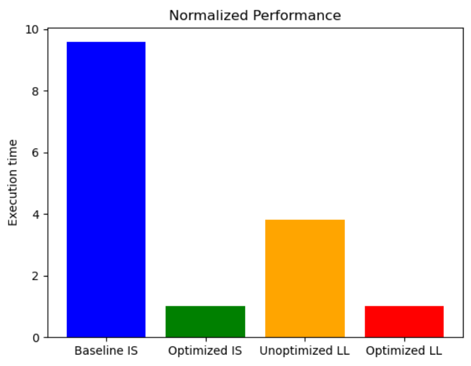

Fig. 5. Normalized performance results comparing optimized performance to baseline performance. IS = index sum, LL = linked list.

for an instruction where the left-hand side is set to an expression with some nested occurrence of itself and a `getelementptr` instruction (representing a struct field access). We also were stuck for multiple days due to a bug cross-compiling statically (gem5/Kobold is based on RISC-V); we ended up needing to compile files and run executables on the same computer even though they were static executables. We learned a substantial amount about cross-compiling, loops, linking, and writing LLVM passes in the process of implementing this optimization and troubleshooting these issues.

VI. Conclusions and Future Work

We have demonstrated the potential for transparent-offload in near-cache computing systems by implementing and evaluating a compiler that extracts tasks for near-cache execution. We present a functionally correct compiler that effectively extracts tasks for offload by analyzing memory patterns and finding affine and pointer chasing reduction operations. We demonstrate the potential of analyzing memory access patterns to find long running operations over data structures. Through our performance results, we demonstrate that finding long running operations over a common data structure and offloading these code regions to near-cache compute units can significantly improve performance by reducing data movement necessary.

In the future, we would like to expand our solution so that it may recognize and offload more operations besides reduction to enable further general-purpose applicability. Furthermore, we would like to implement the be able to schedule engine task sending to as early in the program as possible to increase parallelism available. However, the current Kobold interface currently prevents this as it requires applications to wait for results from task offload before executing later operations.

VII. Project Logistics

Work was evenly distributed between both members.

References

[1] J. Brana, B. C. Schwedock, Y. A. Manerkar, and N. Beckmann, "Kobold: Simplified cache coherence for cache-attached accelerators," *IEEE Computer Architecture Letters*, vol. 22, no. 1, pp. 41–44, 2023.

[2] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman, "Composable accelerator-rich microprocessor enhanced for adaptivity and longevity," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2013.

[3] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Charm: A composable heterogeneous accelerator-rich microprocessor," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, 2012.

[4] W. J. Dally, "GPU Computing: To Exascale and Beyond," in *Supercomputing '10, Plenary Talk*, 2010.

[5] J. Hennessy and D. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," in *Turing Award Lecture*, 2018.

[6] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *ISSCC*, 2014.

[7] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 204–216.

[8] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the top: What will drive computer performance after moore's law?" *Science*, vol. 368, no. 6495, 2020.

[9] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing soc accelerators for memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, 2013.

[10] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann, "Livia: Data-centric computing throughout the memory hierarchy."

[11] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News*, 2005.

[12] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "Snnap: Approximate computing on programmable socs via neural acceleration," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[13] A. Pattnaik, X. Tang, O. Kayiran, A. Jog, A. Mishra, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Opportunistic computing in gpu architectures," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 210–223. [Online]. Available: https://doi.org/10.1145/3307650.3322212

[14] B. C. Schwedock, P. Yoovidhya, J. Seibert, and N. Beckmann, "täkō: A polymorphic cache hierarchy for general-purpose optimization of data movement."

[15] V. Soria-Pardos, A. Armejach, T. Mück, D. Suárez-Gracia, J. Joao, A. Rico, and M. Moretó, "Dynamo: Improving parallelism through dynamic placement of atomic memory operations," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589065

[16] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV, 2010.

[17] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores."

[18] Z. Wang, J. Weng, S. Liu, and T. Nowatzki, "Near-stream computing: General and transparent near-cache acceleration."

[19] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual volume ii: Privileged architecture version 1.9.1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-161, Nov 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-161.html