Automating and Optimizing Data Transfers for Many-core Coprocessors

Bin Ren*, Nishkam Ravi†, Yi Yang†, Min Feng†, Gagan Agrawal*, Srimat Chakradhar†
*Dept. of Computer Science and Engineering, The Ohio State University
†NEC Laboratories America
{ren, agrawal}@cse.ohio-state.edu, {nravi, yyang, mfeng, chak}@nec-labs.com

ABSTRACT

Orchestrating data transfers between CPUs and a coprocessor manually is cumbersome, particularly for multi-dimensional arrays and other data structures with multi-level pointers, which are common in scientific computations. This work describes a system that includes both compile-time and runtime solutions for this problem, with the overarching goal of improving programmer productivity while maintaining performance.

We implemented our best compile-time solution, *partial linearization* with *pointer reset*, as a source-to-source transformation, and evaluated our work by multiple C benchmarks. Our experiment results demonstrate that our best compile-time solution can perform 2.5x-5x faster than original runtime solution, and the CPU-Coprocessor code with it can achieve 1.5x-2.5x speedup over the 16-thread CPU version.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: [Concurrent Programming — Parallel Programming]

Keywords

Coprocessors, Static Analysis, Runtime Analysis, Offloading

1. TECHNICAL DESCRIPTION

Accelerating parallel computation using many-core coprocessors requires specification of code regions that can be profitably offloaded to the coprocessor and executed as independent tasks. These code regions have been specified by the developer using low-level APIs till recently. The software available with Xeon Phi, as well as the emerging *directive-based* models for GPU programming, are providing much higher-level APIs for using accelerators. However, even with such high-level APIs, there are many challenging issues. Particularly, orchestrating data transfers for multi-level pointers using `in/out` or equivalent clauses is cumbersome and error-prone.

With the goal of further improving productivity of HPC programmers while also maintaining performance, we focus on easing data transfer related efforts, considering both compile-time and runtime solutions. While such data transfers for static arrays can be handled by ICC compiler[1] today, and solutions proposed previously by the literature [2, 1] can handle dynamically allocated one-dimensional arrays, the open problem is handling dynamically allocated multi-dimensional arrays or other structures with multi-level pointers.

It turns out that the problem is quite complex, particularly because the choice of the mechanism used for automatically inserting data transfer clauses impacts memory layouts and access functions (subscripts) on the coprocessor. Because of the nature of the accelerators and complex interactions between the resulting source-code and the native compiler on the accelerator, the performance can be impacted in multiple ways. Overall, our work considers four metrics: 1) Minimization of redundant data transfers, 2) Utilization of Direct Memory Accesses (DMA), 3) Minimization of memory allocation overheads on the accelerator (or even the host), and 4) Preservation of aggressive memory-related compiler optimizations (e.g., vectorization and prefetching) by proper memory layout and accesses for the accelerator.

Our work describes an automated framework that uses both compile time and runtime solutions to address this problem. This system includes a simple but effective compile-time solution, where we linearize the heap without having to modify the memory accesses (subscripts), by using a *pointer reset* approach. This method scores well on all of our metrics and maintains code readability.

For the cases where our compile-time approach cannot apply, we also explore runtime solutions. The background is that a system like Xeon Phi also has shared memory implementations available between the main processor and accelerator. We also investigate and optimize the performance of the runtime memory management approach, by providing certain improvements to the existing coherence protocol. The best compile-time solution consistently performs better than the optimized runtime scheme, but is not as generally applicable. In order to combine performance with generality, we describe a mechanism for integrating the two disjoint approaches using a simple source-to-source transformation.

We have implemented our compile-time solution as a source-to-source transformation using the Apricot framework and evaluated it within the context of application execution on Xeon Phi coprocessor.

ICS'14, June 10–13, 2014, Muenchen, Germany
ACM 978-1-4503-2642-1/14/06.
http://dx.doi.org/10.1145/2597652.2600114

2. REFERENCES

[1] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *SC*, 2010.

[2] N. Ravi, Y. Yang, T. Bao, and S. Chakradhar. Apricot: an Optimizing Compiler and Productivity Tool for x86-Compatible Many-Core Coprocessors. In *ICS*, pages 47–58, 2012.

[1] Intel C++ Compiler. http://www.intel.com/Compilers.