A Framework for Resource Aware Multithreading

Sunil Shrestha
and Guang R. Gao
University of Delaware
Newark, DE 19716
Email: sunil@udel.edu
ggao@capsl.udel.edu

Joseph Manzano
and Andres Marquez
Pacific Northwest National Laboratory
Richland, WA 99354
Email: joseph.manzano@pnnl.gov
andres.marquez@pnnl.gov

***Abstract*—In this poster, we present a novel methodology that takes into consideration multithreaded many-core designs to increase both intra and inter tile parallelism as well as memory residence on tilable applications. It partly takes advantage of polyhedral analysis and transformation, combined with a highly optimized fine grain tile runtime to exploit parallelism at multiple levels of the memory hierarchy. The main contributions include (1) a framework for multi-hierarchical tiling techniques that takes advantage of intra-tile parallel-start parallelism, (2) a data-flow inspired runtime library that allows the expression of parallelism with an efficient synchronization registry (3) and an efficient memory reuse strategy. Our current implementation shows significant performance improvements on an Intel Xeon Phi board against instances produced by state-of-the-art compiler frameworks for selected loop nests.**

I. Introduction

Harvesting the immense potential of current multithreaded many-core architectures is not an easy task. Inefficiencies inherent from previous generation techniques can lead to serious performance bottlenecks due to unpredicted interference or unnecessary hardware artifacts.

Classically, for microarchitectures with small number of threads or cores, compilers have used tiling techniques to take advantage of the parallelism in order to reduce synchronization and improve locality [1]. With the advent of multithreaded many-core architectures, new opportunities arise to tap efficiently into finer levels of parallelism.

Another major contributor to performance and power is the data movement that the application must do to achieve its goals. Loop transformation to maximize reuse in low latency memory such as caches and scratchpad SRAMs as proposed for Exascale architectures - have been widely studied [2], [3]. However, these techniques still operate under the paradigm that memory agnostic coarse grain parallelism is the norm. Even when the inner tiles are assigned to fine grain tasks, their memory agnostic placement will still incur heavy penalties when resources are contended. Such contention can happen at different levels of memory hierarchy, memory banks and memory pages. Memory centric approaches [4]–[6] although has been studied, it has yet to encompass the idea of collective approach for multiple threads working together. Thus, we introduce our novel framework to better tap into the vast potential offered by this new generation of computers.

II. Group Locality

Group Locality(GL) is the concept in which group of threads work together as a unit, are aware of each others execution pattern and hence can collaborate at a very fine-grain level. Using careful orchestration of memory accesses, data movement and data restructuring, interference in memory is reduced. In order to achieve this, we developed an end-to-end framework that uses static information gained from the compiler to improve locality of multiple threads working in parallel. Such framework uses: (1) Efficient and novel tiling strategies with a focus on intra-tile parallelism. (2) A Fine grained parallelism runtime where threads perform micro dataflow execution using bit mask matrices to represent their dependencies graphs. Finally, (3) runtime techniques designed to optimize data movement, by reshaping key data structures and the careful mapping of the such structures to physical resources to reduce interference in memory access.

A. Efficient Tiling Techniques

Our framework uses the information generated by the state-of-the-art polyhedral tool PLUTO [7] to generate tiles for the closest-to-the-processor memory level[1]. These tiles are created to minimize the communication volume, as well as the temporal distance of such communication. When using PLUTO for multi-hierarchical tiling, these schedules are reapplied to the different levels of the memory hierarchy as-they-are, i.e. without considering possible interference from multiple threads or other external factors. This may lead to restricted parallelism between the L1 tiles[2]. In our framework, we solve this by constructing outer tiles such that at least one face of the polytope has a truly concurrent start. Given 'm' hyperplanes, we create at least one parallel hyperplane and use $m-1$ hyperplanes used for L1 tiling.

Using such technique, we create jagged tiles such that inner tiles have concurrent start. We show two different flavors of such tiling strategy (1) for a pipeline start application and (2) for a parallel start application which we refer to as Jagged Diamond Tiling.

B. Fine-Grain Execution

Codes that are designed to run at a very fine grain level suffer from communication overhead, reflected in its performance. With jagged tiling we have created a highly parallel code that is capable of running multiple level of tiles in parallel. In order to exploit available parallelism, we use a data-flow inspired low overhead dependency and task update scheme, based on

[1] referred as L1 Tiles for the rest of the discussion

[2] such as imposing a pipeline parallel execution schedule when it might not be required

bit masks that represents the data dependency graphs among the set of tiles. Using these masks to enable fine grained tasks, multiple threads can work synchronously within a tile.

Each group of threads grab a L2 task and initial dependency mask associated with it. Such masks are repeatable across different L2 tiles for regular applications. Dependencies among the lowest level tiles are represented by different sets of bits which are collectively updated by a group of threads working together inside the highest level tile. Each thread perform atomic bit-wise operations on the dependency masks and creates a task mask that represents all the tasks ready to execute. If the task mask is non-zero, thread finds the task, execute, update dependencies and update the task mask. This happens in a highly parallel fashion such that every thread is aware of the status of the tasks within the assigned tile. The implementation of this approach of synchronization between threads is done solely using atomic operations to keep the overhead low. Moreover, this approach can be used in upper level of the hierarchy to exploit even more parallelism.

C. Data Reshaping

Whilst exploiting the architecture parallelism, we still need to take advantage of the opportunities that can be extracted from the mapping and reshaping of data structures based on their access patterns. Reshaping and mapping are achieved through an invertible linear mapping function. This function is necessary to avoid element collisions to the same address. Indexes from iteration space i are mapped to a new iteration space j using transformation matrix T such that $\vec{j} = T\vec{i}$ [8].

Reshaping of data comes with two different overheads. First, it requires extra memory space that adds to the application memory footprint. Second, it adds to the overhead of synchronization between data transfer and usage. In order to reduce the space overhead, we divide threads into two restructuring usage groups: *use* and *reuse*. A group of threads enters a *use* phase in which data is brought into the restructure space and transformed by the same thread using the unstructured data. After this phase completes, the next phase, dubbed *reuse*, takes place. During this phase, participating threads will have access to the transformed space with the optimized access pattern. Such approach has multiple advantages. One of them is data restructured by one thread can be reused by the other threads improving the locality of an entire thread group. Another advantage of this scheme is that any synchronization needed between data use and reuse is localized to the group of threads working together.

III. Experiments

The experiments were done using an Intel Xeon Phi 7110P coprocessor. Each coprocessor is equipped with 61 cores running at 1.1 GHz connected. Each core can support up to 4 hyper-threads. Each core has 32KB L1 cache per thread and 512KB L2 cache shared by its 4 hyper-threads. On the software side, we chose a LU decomposition kernel, heat-3D and a Gauss Seidel-2D linear solver as our examples.

In our experiments, we created group of threads (4 per group) such that they work together within outer L2 tiles. Such approach is designed to exploit L2 cache locality and inner parallelism. The poster shows speedup graphs of our technique against PLUTO generated code for Gauss Seidel-2D and performance chart for heat-3D. This shows a significant improvement in performance between our fine grain code and the PLUTO generated code. Moreover, the poster compares the results of the fine-grain LU decomposition kernel previously used against a fine-grain plus data restructuring version of the same code. Using our restructuring technique, we align our data to minimize interference while accessing reuse data. Our results show that our performance improves significantly with minimal changes in power consumption. Since, energy is the function of power and time, we show that we improved the energy profile of this kernel.

IV. Conclusion

Several optimization techniques over the years have provided significant boost in performance by reducing memory access latency. These techniques, although very effective, are often unaware of the vast set of resources that massively parallel systems offer. This can lead to the resources sitting idle during critical performance phases of an application. Moreover, even when the resources are used, the communication patterns of each parallel entity must work in concert to prevent certain performance/power pathologies created by contention or overly restrictive actions taken by the runtime and/or hardware [3]. The framework presented in this poster aims to exploit parallelism, manage fine grain execution framework and steers data restructuring. We showed that when parallel threads collaborate, better utilization of both processor and memory resources in terms of power and performance is possible.

Acknowledgment

This research was supported in part by DOE ASCR XStack program under Awards DE-SC0008716, DE-SC0008717

References

[1] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, ser. PLDI '91. New York, NY, USA: ACM, 1991, pp. 30–44.

[2] D. Bacon, S. Graham, and O. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys (CSUR)*, vol. 26, no. 4, pp. 345–420, 1994.

[3] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, Los Alamitos, CA, USA, 2012, pp. 40:1–40:11.

[4] I. Kodukula, N. Ahmed, and d Keshav Pingali, "Data-centric multi-level blocking," 1997, pp. 346–357.

[5] M. M. Baskaran *et al.*, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 1–10.

[6] G. Bikshandi *et al.*, "Design and use of htalib: a library for hierarchically tiled arrays," in *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, ser. LCPC'06. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 17–32. [Online]. Available: http://dl.acm.org/citation.cfm?id=1757112.1757117

[7] U. Bondhugula and J. Ramanujam, "Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer," 2007.

[8] S.-T. Leung and J. Zahorjan, *Optimizing data locality by array restructuring*. Department of Computer Science and Engineering, University of Washington, 1995.

[3] i.e. unnecessary memory flushes, closing of needed memory pages, etc