Dynamic Percolation: A Case of Study on the Shortcomings of Traditional Optimization in Many-core Architectures

Elkin Garcia
University of Delaware
egarcia@udel.edu

Daniel Orozco
University of Delaware
orozco@udel.edu

Rishi Khan
ET International
rishi@etinternational.com

Ioannis E. Venetis
University of Patras
venetis@ceid.upatras.gr

Kelly Livingston
University of Delaware
kelly@udel.edu

Guang Gao
University of Delaware
ggao@capsl.udel.edu

ABSTRACT

This paper provides a discussion on the shortcomings of traditional static optimization techniques when used in the context of many-core architectures. We argue that these shortcomings are a result of the significantly different environment found in many-cores. We analyze previous attempts at optimization of Dense Matrix Multiplication (DMM) that failed to achieve high performance despite extensive efforts towards optimization.

We have found that percolation (prefetching data) and scheduling play a central role in the performance of applications. To overcome those difficulties, we have (1) fused dynamic scheduling and percolation into a *dynamic percolation* approach and (2) we have added additional percolation operations. Our new techniques enabled us to increase the performance of the application in our study from 44 GFLOPS (out of 80 GFLOPS possible) to 70.0 GFLOPS (operands in SRAM) or 65.6 GFLOPS (operands in DRAM).

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: [Modeling techniques]; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; G.1.0 [**Numerical Analysis**]: General—*Parallel algorithms*

Keywords

Many-cores, Dynamic Scheduling, Percolation, Cyclops64

1. INTRODUCTION

This paper presents a comprehensive case of study that shows how to obtain high performance in modern many-core processors. This study is important because it addresses situations not previously encountered in multi-core architectures, other shared memory processors or distributed memory systems. Many-cores provide an environment where hardware resources are uncomplicated and abundant. Large numbers of thread units are present, on-chip memory is user-managed, no automatic data cache is present and hardware support for synchronization is available. The environment is different and requires new optimization paradigms.

Even in the simple case of Dense Matrix Multiplication (DMM) running on IBM's Cyclops-64 processor (C64) [2], extensive efforts toward optimization using a broad range of static optimization strategies such as multiple levels of tiling, instruction scheduling, register allocation, manual instruction selection, optimized synchronization and several other only resulted in disappointing performance of 44.12 GFLOPS [4] (out of 80 GFLOPS possible).

These results ultimately show that peak performance could not be achieved by static techniques alone, even for simple, highly parallel and regular programs such as DMM. Mainly, this happens because it is not possible to statically create a plan that efficiently schedules data prefetching (percolation) and computation at the right times. The reason is that small variations in the execution of tasks voids the possibility of making optimal scheduling decisions a-priory.

To solve the difficulties in percolation and scheduling, we propose to take advantage of the fine-grain synchronization primitives available in many-core architectures. Percolation and dynamic scheduling can be fused together in what we call *dynamic percolation* which dynamically schedules data prefetching at an appropriate time so that (1) data is available when the computation needs it and (2) the percolation operation is done when enough memory bandwidth is available. We apply this technique for the optimization of DMM.

2. BACKGROUND

Cyclops-64: C64 is a homogeneous many-core system on a chip architecture designed by IBM [2]. A C64 chip is an aggregation of 160 simple MIMD Thread Units (TU). It has 80 floating point units (FPU), 5 MB of user-managed on-chip memory (SRAM) with total bandwidth of 320 GB/s and 1 GB of DRAM with a total bandwidth of 16 GB/s. There is no automatic data cache. C64 has a total performance of 80 GFLOPS per chip when running at 500MHz. In addition, C64 incorporates efficient support for hardware barriers and atomic in-memory operations. Each memory controller has an ALU that allows it to execute atomic operations directly inside the memory controller (both SRAM and DRAM), without help from a thread unit.

Tiling: Bandwidth is the bottleneck for most naively-implemented algorithms in C64. On-chip memory is frequently used to perform partial computations (tilings) [1, 4, 8] decreasing the amount of bandwidth required.

Static Scheduling and Data Partitioning: Scheduling is an important optimization for programs once the bottleneck of memory bandwidth has been removed through tiling. Scheduling presents challenges in itself since it requires assignment of work to processors at the appropriate time, taking into account issues such as availability of resources and availability of data. The scheduling problem is complicated by the fact that the tasks scheduled to each processor are not necessarily identical. The problem seems simpler for regular and embarrassingly parallel applications, where the amount of data can be distributed uniformly between TUs, expecting similar execution times.

Two main factors, under the scenario imposed by many-core architectures, decrease the expected performance of this static approach to the point of making it impractical even for regular applications: 1) The imbalance, due to competition for shared resources, produced by shared resources even with tasks that perform similar amounts of work. 2) Partitioning the problem statically (in equal amounts of work per TU) may result in non-optimal tile sizes with poor performance.

Percolation: Uninterrupted computation by the processing units in a many-core chip requires data to be available continuously. Percolation is the process by which data is moved across the levels of memory hierarchy to meet the necessities of locality for computation. Percolation is related to data prefetching in that both achieve the same objective. As opposed to conventional data prefetching, percolation operations are expressed as tasks on their own, with precedence relationships with other computational tasks and with restrictions to available resources such as bandwidth or on-chip memory space.

It is difficult to know *a priori* when percolation should be done. As explained before, not all tiles are of the same size, and not all tiles take the same amount of time, even when they perform similar amounts of computations.

3. DYNAMIC SCHEDULING: SCALABILITY AND PERCOLATION

Our target operation is DMM ($C = A \times B$) for matrices with size $m \times m$. We propose a separation of the problem into two orthogonal subproblems: 1) optimizing DMM in SRAM moving operands between SRAM and Registers and 2) moving data between DRAM and SRAM. To extend the matrices to DRAM we simply partition the matrices A, B and C into $n \times n$ blocks $A_{i,k}$, $B_{k,j}$ and $C_{i,j}$ that fit in SRAM. This is similar to the blocking performed in traditional cache hierarchies, resulting in a trade off between computation and data movement. Each block of C is calculated by

$$C_{i,j} = \sum_{k=0}^{\frac{m}{n}-1} A_{i,k} \cdot B_{k,j} \qquad (1)$$

Considering the limitation of bandwidth in the crossbar and the unpredictable effects of resource sharing, we must devise a schedule that considers both computation and data movement efficiently. DMM with operands on DRAM will require two kind of tasks: Data movement tasks and computation tasks. Our analysis will follow a bottom-up approach:

1. We will analyze how to optimize MM in SRAM. Two major aspects are studied and solutions are given in each case: A load balanced scheduler with low overhead and an optimized computation task with proper *percolation* of operands between SRAM and Registers.

2. We will analyze the MM in DRAM. The main aspect studied here is a load-balanced scheduler that effectively overlaps data movement and computation tasks using *dynamic percolation*

Dynamic Scheduling for Computation Tasks: Static Scheduling (SS) is suboptimal because it does not consider two main sources of imbalance in a many-core environment: 1) The amount of work is a function of how the block is tiled and what fraction of tiles are not of optimum size. 2) Possible stalls due to arbitration of shared resources.

The unpredictable effects of resource sharing are a formidable challenge for SS. A static block partition exacerbates problems, especially when the number of processors (P) is increased. Despite the simplicity and regular behavior in computation and data access of DMM, static techniques cannot overcome these problems. At that point, Dynamic Scheduling (DS) arises as a feasible solution able to alleviate the overhead and scalability problems of SS.

We propose a work-stealing approach where the computation of optimum size tiles in matrix C are scheduled dynamically using atomic in-memory operations. The advantages of this technique are low overhead and improved load-balance in the presence of stalls.

Percolation in the Computation Tasks: Most of the time is spent computing tiles. Therefore, computation deserves special attention. Previous Instruction Scheduling [4], partially hides the latency incurred while fetching operands from SRAM to registers. The remaining stalls due to latencies in memory movements from SRAM are avoided with a combination of loop unrolling and percolation.

Dynamic Percolation: A DMM algorithm, that has been highly optimized through SRAM percolation, is severely limited in the size of the matrices that can fit in SRAM (i.e. 500×500). We extend DMM into DRAM by blocking at the SRAM level and using our percolated MM algorithm in SRAM. We assume that the target many-core architecture has no hardware mechanisms for block transfers (e.g. caches or DMA engines), forcing us to use TUs for memory movement. The computational TUs must be orchestrated with data movement TUs to enforce data dependencies: work cannot be done before a matrix is loaded and a matrix cannot be unloaded until work using it is completed. Further, TUs performing data movement should help with computation if there is no data to move. The straightforward static scheduling approach using barriers will waste resources while TUs are waiting on barriers. Also, it is inefficient for all TUs to copy data at the same time given the limited DRAM bandwidth. A dynamic scheduling approach replaces the barriers with finer-grained signals while still enforcing data dependencies.

We introduced *Dynamic Percolation*, where data movement tasks and computation tasks were assigned dynamically. Helper Threads (HT) are in charge of the data movement tasks and Computation Threads (CT) are in charge of the computation tasks. Computation and data movement tasks are overlapped by a pipelined schema using a double buffer in SRAM. Moreover, the distribution of computation

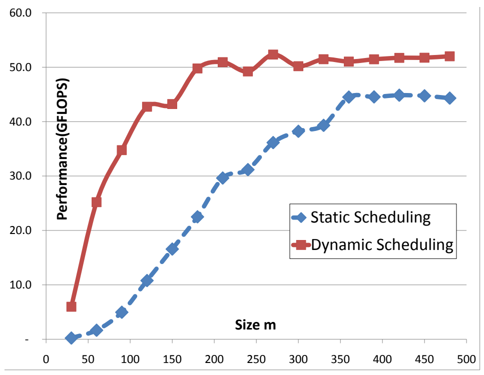

Figure 1: Scalability vs Matrix Size with 156 TUs

tasks and data movement tasks will vary in the course of Dynamic Percolation.

The Dynamic Scheduler for each set of tasks is implemented using atomic in-memory additions. The main advantage of this implementation is the low overhead given by the low latency of in-memory operations. They avoid unnecessary roundtrips to memory and they provide the necessary synchronization due to the atomicity supported by the hardware.

There is also a hierarchy of tasks. Tasks related with data movement of the $C_{i,j}$ blocks are at the highest level, data movement of the blocks $A_{i,k}$ and $B_{k,j}$ is next with computation tasks at the lower level.

4. EXPERIMENTAL EVALUATION

This section describes the experimental evaluation based on the analysis done in section 3 using the C64 architecture described in section 2. Our baseline parallel DMM implementation uses Static Scheduling (SS) and the set of optimizations described in [4].

Using the DMM in on-chip SRAM, we compared the scalability of SS and DS with several matrix sizes. Figure 1 shows that SS not only has a lower performance than DS, but also its performance is affected drastically for smaller matrices. This is critical to extend our algorithm to off-chip DRAM because smaller block sizes are required to hide the latency of data movement tasks.

Figure 2 compares three implementations. (1) A fully parallel DMM that uses off-chip DRAM without overlapping computation and data movement. This implementation is based in the already optimized version of DMM that uses on-chip SRAM. (2) A version with the proposed Dynamic Percolation with optimized computation tasks, the optimum number of HTs is 24. (3) A version that uses Dynamic Percolation, optimized computation tasks, and communication that was optimized for the on-chip block sizes and the required transposition of matrix A.

In the best implementation, only 8 HTs were needed, increasing the performance due to the larger number of TUs available for computation, and ultimately achieving 65.63 GFLOPS with matrices of 6336×6336 using 156 TUs: 82.02% of the theoretical peak performance of C64.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a Dynamic Percolation technique using two types of tasks – computation tasks and

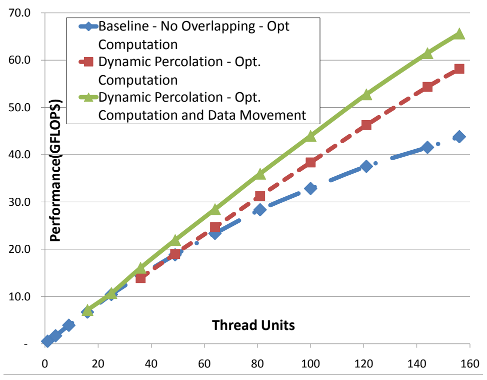

Figure 2: Performance of MM in off-chip DRAM

data movement tasks. The distribution of computation tasks and data movement tasks will vary in the course of dynamic percolation. Therefore, our method allows runtime redistribution between computational threads and data movement threads to achieve better utilization of thread unit resources. We have shown several advantages of the method proposed over well-known static techniques for many-core architectures in terms of scalability as a function of the matrix size for DMM. Our method also load-balances tasks across the machine because it handles well the unpredictable effects of resource-sharing, drastically improving performance. We report experimental results of our methods on a real C64 chip achieving 70.0 and 65.6 GFLOPS (out of 80 GFLOPS) for DMM with operands in SRAM and DRAM respectively.

Future work will apply this techniques to a broader set of applications, extending the work on Dynamic Scheduling [3], High Throughput Algorithms [5, 6] and Efficient Task Representation [7] for many-cores.

6. REFERENCES

[1] Chen, L., Gao, G.R.: Performance Analysis of Cooley-Tukey FFT Algorithms for a Many-core Architecture. In: HPC 2010 (2010)

[2] Denneau, M., Warren Jr., H.S.: 64-bit Cyclops: Principles of Operation. Tech. rep., IBM Watson Research Center, Yorktown Heights, NY (April 2005)

[3] Garcia, E., Orozco, D., Pavel, R., Gao, G.R.: A discussion in favor of Dynamic Scheduling for regular applications in Many-core Architectures. In: MTAAP'12. IEEE, Shanghai, China (May 2012)

[4] Garcia, E., Venetis, I.E., Khan, R., Gao, G.: Optimized dense matrix multiplication on a many-core architecture. In: Euro-Par'10. Ischia, Italy (2010)

[5] Orozco, D., Garcia, E., Khan, R., Livingston, K., Gao, G.: Toward high-throughput algorithms on many-core architectures. TACO 8(4), 49:1–21 (January 2012)

[6] Orozco, D., Garcia, E., Khan, R., Livingston, K., Gao, G.R.: High throughput queue algorithms. CAPSL Technical Memo 103 (January, 2011)

[7] Orozco, D., Garcia, E., Pavel, R., Gao, G.: TIDeFlow: The Time Iterated Dependency Flow Execution Model. In: DFM 2011. Galveston Island, TX, USA (October 2011)

[8] Orozco, D.A., Gao, G.R.: Mapping the fdtd application to many-core chip architectures. In: ICPP'09. pp. 309–316. IEEE Computer Society, Washington, DC, USA (2009)