A Dynamic Schema to increase performance in Many-core Architectures through Percolation operations

Elkin Garcia*, Daniel Orozco*, Rishi Khan†, Ioannis E. Venetis‡, Kelly Livingston* and Guang R. Gao*
*Computer Architecture and Parallel System Laboratory (CAPSL) - Electrical and Computer Engineering Department
University of Delaware. Newark, DE. 19716, U.S.A.
Email: {egarcia@, orozco@, kelly@, ggao@capsl.}udel.edu
†ET International
Newark, DE 19711, U.S.A.
Email: rishi@etinternational.com
‡Department of Computer Engineering and Informatics
University of Patras. Rion 26500, Greece
Email: venetis@ceid.upatras.gr

***Abstract*—Optimization of parallel applications under new many-core architectures is challenging even for regular applications. Successful strategies inherited from previous generations of parallel or serial architectures just return incremental gains in performance and further optimization and tuning are required. We argue that conservative static optimizations are not the best fit for modern many-core architectures. The limited advantages of static techniques come from the new scenarios present in many-cores: Plenty of thread units sharing several resources under different coordination mechanisms.**

We point out that scheduling and data movement across the memory hierarchy are extremely important in the performance of applications. In particular, we found that scheduling of data movement operations significantly impact performance.

To overcome those difficulties, we took advantage of the fine-grain synchronization primitives of many-cores to define percolation operations in order to schedule data movement properly. In addition, we have fused percolation operations with dynamic scheduling into a dynamic percolation approach.

We used Dense Matrix Multiplication on a modern many-core to illustrate how our proposed techniques are able to increase the performance under these new environments. In our study on the IBM Cyclops-64, we raised the performance from 44 GFLOPS (out of 80 GFLOPS possible) to 70.0 GFLOPS (operands in on-chip memory) and 65.6 GFLOPS (operands in off-chip memory). The success of our approach also resulted in excellent power efficiency: 1.09 GFLOPS/Watt and 993 MFLOPS/Watt when the input data resided in on-chip and off-chip memory respectively.

I. 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 a situation arising on many-core architectures and not previously encountered in multi-core architectures, or other systems such as clusters or shared memory processors. Many-cores provide an environment where hardware resources are uncomplicated and abundant. Large numbers of thread units are present, on-chip memory can be user-managed, automatic data cache may not be present and hardware support for synchronization is available. In summary, the environment is different, and it requires a new optimization paradigm.

We have observed that the use of traditional optimization techniques does not result in the best performance in many-core architectures. As an example, we take the simple case of dense matrix multiplication (DMM) running on a modern many-core architecture such as the IBM Cyclops-64 processor (C64) [8]; the extensive efforts toward optimization of this important kernel only resulted in a disappointing performance of 44.12 GFLOPS (out of 80 GFLOPS) possible [19], [17]. This far-from-optimal performance was not the result of lack of trying. The study presented by Garcia explored a broad range of optimization strategies: Multiple levels of tiling were employed, instruction scheduling, register allocation and instruction selection was done by hand, the code was written in assembly, pipelining was used, synchronization was optimized through the use of hand-written assembly primitives and so on.

The study presented by Garcia ultimately shows that peak performance could not be achieved by static techniques alone, even for simple, highly parallel and regular programs such as matrix multiply.

Being surprised by Garcia's early results in Matrix Multiply, we analyzed their experiments to find why their methodical approach failed to achieve peak performance. Through extensive profiling, we have seen that static plans are bound to fail to achieve peak performance in many-core architectures. Mainly, this happens because it is not possible to statically create a plan that efficiently schedules data movement and computation at the right times. The reason is that small variations in the execution of tasks (or

Early results of this research were published as a short paper in Computer Frontiers 2012 under the tile "Dynamic Percolation: A Case of Study on the Shortcomings of Traditional Optimization in Many-core Architectures". This paper extends the content of our previous publication

even individual instructions) voids the possibility of making optimal scheduling decisions a-priory. Even the best policies for scheduling frequently resulted in data movement being scheduled too late or when main memory bandwidth was not available because another data movement operation did not finish on time. In any case, the result is the same: time and resources were wasted, leading to poor performance.

To solve the difficulties in the data movement and scheduling, we took advantage of the fine-grain synchronization primitives available in many-core architectures. We propose *Percolation*; percolation is the process by which data is moved across the levels of the memory hierarchy to meet locality requirements for computation but as opposed to data prefetching, computation tasks are not scheduled until the percolation operation has not finished, e.g. there is explicit synchronization between data movement tasks and computation tasks. In addition, percolation operations consider restrictions to available resources such as bandwidth or on-chip memory space.

In addition, we found that Percolation and dynamic scheduling can be fused together in what we call *Dynamic Percolation* which dynamically schedules data percolation 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.

This paper describes the process that ultimately led to a successful optimization of parallel applications, we examine in detail a simple but challenging benchmark: the Dense Matrix Multiplication (DMM). We describe how we introduced percolation operations and how we used dynamic percolation to complete the optimization process. After the optimization of DMM on C64 using the proposed techniques, the performance increases to 70.0 GFLOPS (87.5% the theoretical peak performance) and 65.6 GFLOPs (82% of the theoretical peak performance) when the input data for DMM resided in on-chip and off-chip memory respectively. These results are far from any other benchmark implemented on the Cyclops-64 processor [17], [5], [23], [6].

In addition, our success with dynamic percolation also enabled us to achieve impressive results in power efficiency, reaching 1.09 GFLOPS/Watt (operands in SRAM) and 0.993 GFLOPS/Watt (operands in DRAM)

The paper is organized as follows: Section II presents relevant background information. Section III defines the problem addressed in this paper. Section IV shows the optimizations for tasks and scheduling we propose. Section V presents our results and talks about the effectiveness of our approach. Section VI presents other related work in the field. Finally, Section VII presents our conclusions and possible directions for our future work.

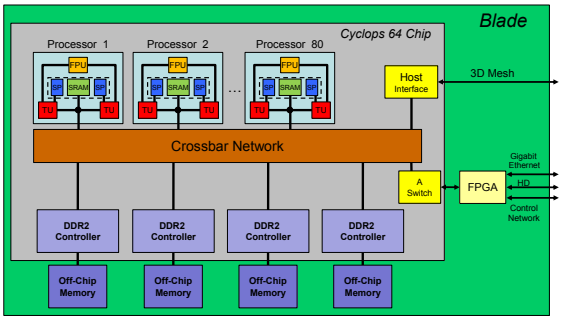


Figure 1: C64 Chip Architecture

II. Background

A. The IBM Cyclops-64 Architecture

Cyclops-64 (C64) [8], [17] is a homogeneous many-core system on a chip architecture designed by IBM. A C64 chip is an aggregation of 160 simple MIMD Thread Units (TU), a design that will share many features with future architectures. As such, it is an excellent and tangible testbed to both expose and provide insight into the future problems that many-core architectures must surmount in order to maintain high performance and power efficiency.

A C64 chip contains 80 processors, each one of them containing 2 Thread Units (TU), a floating point unit (FPU), two on-chip SRAM memory banks of 30KB each, and a port to the on-chip interconnect. The TUs must share both the on-chip network port and floating point unit, allowing for better utilization of these resources. Amongst every 5 processor cores is an I-cache of 32 KB for storing code. The off-chip DRAM memory, which is typically 1GB, can be accessed through 4 DDR2 DRAM controllers, each interfacing 64-bit channels running at 500MHz, providing a total bandwidth of 16 GB/s or 100 MB/s per TU in the system. A 96-port crossbar network (CBN) with a bandwidth of 4GB/s per port connects all TUs and on-chip memory banks [8]. A C64 node can be seen in Figure 1.

A C64 chip has an explicit three-level memory hierarchy: scratchpad memory, on-chip SRAM, off-chip DRAM. There is no automatic data cache. All on-chip memory is user-managed. The scratchpad memory (SP) is a configured portion of each on-chip SRAM bank which can be accessed with very low latency by its associated TU. The remaining sections of on-chip SRAM banks comprise the on-chip interleaved global memory (GM), which is uniformly accessible from all TUs. For floating-point performance, C64 can issue one double precision "Fused Multiply and Add" instruction per cycle per processor, for a total performance of 80 GFLOPS per chip when running at 500MHz. In addition, the C64 instruction set architecture incorporates efficient support for hardware barriers and atomic in-memory operations. Each memory controller has an ALU that allows it to execute atomic operations in 3 clock cycles directly inside the memory controller (both SRAM and DRAM), without help from a thread unit.

B. Limitations of modern many-cores

Bandwidth is the bottleneck for most naively-implemented algorithms in general purpose many-core architectures such as GPUs [4], [25], the new Intel Xeon Phi [24] and The IBM Cyclops-64 [8].

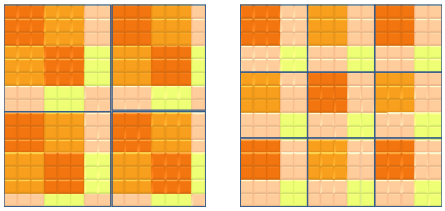
We use a simple observation on the C64 architecture to illustrate this problem of modern many-cores. Achieving 80 GFLOPS (the peak performance of C64) in a processor running at 500 MHz requires execution of 80 Fused-Multiply-Add instructions per cycle. A naive implementation that loads and stores the results of such operations from and to off-chip memory will require to load 160 operands and store 80 operands per cycle (assuming perfect pipelining of the operations to hide latency). Naive implementations will be likely limited by off-chip memory bandwidth given the limitations of off-chip memory controllers, each allowing a single memory operation per cycle (e.g. C64 only has 4 off-chip memory controllers).

On-chip memory can be used to reduce the requirements on off-chip memory bandwidth. Optimized algorithms usually rely on partial computations done in on-chip memory and registers. Tiling techniques exploit the locality of data, reusing data as much as possible. Studies in this area are extensive for GPUs [9], [30], Intel Xeon Phi [10] and IBM Cyclops-64 [17], [5], [23] to alleviate the problem of memory bandwidth. Such approaches frequently require tiling at all levels of the memory hierarchy inside the chip (*e.g.* Garcia et al. [17] showed the necessity of SRAM tiling and Register tiling for Matrix Multiply).

C. 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. These two factors are: 1) shared resources and 2) size and shape of tiles.

Shared resources such as function units or bandwidth are a source of imbalance even with tasks that perform similar computations over the same amount of data. As a result, tasks may have different execution times due to competition for shared resources. This factor is critical on many-cores, where shared resources are abundant at different levels with diverse arbitration schemes.



Partition for TU=4 Partition for TU=9

Figure 2: The figure illustrates the problem of partitioning. Tiles of 3×3 are optimum-sized and they result in the best performance. However, as the number of Thread Units (TUs) increase, the number of optimum-sized tiles decrease. In the Figure, a matrix of 15×15 results in 16 optimum-sized tiles when using 4 TUs, but only 9 optimum-sized tiles are available when using 9 TUs.

The size and shape of the tiles greatly influence the performance of an application. Numerous publications for all kind of many-cores have been devoted to the discussion of what is the optimal tile that must be used for particular problems [29], [9], [18], [26], [17], [5], [23]. Usually, criteria to select a good tile size is that which maximizes the ratio of computation to memory operations given some constraints such as available memory, the desired parallelism, or the number of processing units in a chip. Although tiling effectively improves the efficiency of the computation, it is not always possible to place all of a problem's data into tiles since it is frequent that the problem dimensions are not a multiple of the tile size. For that reason, in general, problems result in a combination of optimal-sized tiles and non-optimal-sized tiles.

For example, a previous study by Garcia et al. [17], showed that the best strategy to compute a matrix multiplication on many-cores with software managed memory hierarchies was to divide the computation uniformly between blocks according to the number of TUs and partition these blocks into optimal-sized tiles if possible, even if such a partition left some non-optimal-sized tiles. Although the idea of partitioning a problem into equal work for all the TUs works well, it may still result in some non-optimal-sized tiles left because the problem size is not necessarily a multiple of the tile size used. These remaining non-optimal-sized tiles result in poor performance during execution. Two factors exacerbate the presence of slow non-optimal tiles: 1) an increased number of TUs working in parallel and 2) a limited amount of on-chip shared memory available to host the data. These two conditions are evident on a many-core environment and they will ultimately limit the ability of an application to reach peak performance [16]. Figure 2 shows a simple example where the amount of data that belongs to non-optimum-sized tiles (in light yellow and red) increases when the number of TUs increases and the amount of data shared is limited.

III. Problem Statement

The objective of our study is to find the reasons for the failure of previous approaches in optimization of programs for many-core architectures. A blanket statement about what is exactly the problem in a general case is unfeasible. Instead, we decided to start by looking at a case of study that involves several features in common to a large family of problems in high performance computing. We think that the solutions proposed are broad enough to be applied to the optimization of a large class of programs.

We analyze in detail the case of Dense Matrix Multiplication (DMM). DMM is a highly regular and embarrassingly parallel, it is used extensively as a major kernel in HPC. Despite multiple effort for optimizing this kernel, the optimization of DMM under new many-cores is still an open question. As an example, after extensive optimizations in C64, the maximum performance reached is below 45 GFLOPS (out of 80 GFLOPS) [17].

The general questions that our inquiries address are (1) What scheduling and data movement strategy enables high performance under the constraints given by the available resources? (2) What is a good technique to achieve load-balancing? and (3) When should data movement be scheduled to prevent computations from stalling due to lack of data?

And in particular, we have undertaken activities that are representative of the problem that we want to solve. We have used Dense Matrix Multiply to understand the challenges posed by each one of the previous questions and we have worked on various issues found: (1) How can high performance be achieved for large problem sizes through the use of on-chip memory? (2) How and when should data movement be scheduled?

We used 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 Matrix Multiply in on-chip memory moving explicitly operands between on-chip memory and Registers and (2) moving explicitly data between off-chip memory and on-chip memory.

To extend the matrices to off-chip memory we simply partition matrices A, B and C into $n \times n$ blocks $A_{i,k}$, $B_{k,j}$ and $C_{i,j}$ that fit in on-chip memory. We assume explicit data movement on the memory hierarchy without DMA engines. Instead we will use Thread Units and the same mechanism for synchronization used for computation, it will allow simplicity between the synchronization of computation and data movement. Each block of C is calculated by

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

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.

IV. Dynamic Scheduling: Scalability and Percolation

This section will explain our findings and solution to the questions raised on Section III. As explained before, a DMM with operands on off-chip memory will require two kind of tasks: Data movement tasks and computation tasks. Our analysis will follow a bottom-up approach:

1) Optimization of DMM in on-chip memory. Two major aspects are studied and solutions proposed: an optimized computation task with proper *percolation* of operands between on-chip memory and Registers and a load balanced scheduler with low overhead.
2) Optimization of DMM in off-chip memory. The main aspect studied here is a load balanced scheduler that effectively overlaps data movement and computation tasks using *dynamic percolation*

A. Scalability and Percolation of DMM in on-chip memory

We already have pointed out the disadvantages of Static Scheduling (SS) from the many-core perspective, and we will explain how Dynamic Scheduling (DS) can improve performance over SS using a register tile as a computational unit of work. After that, we will explore the percolation process of operands between on-chip memory and Registers among other techniques for the optimization of the computation task.

1) Dynamic Scheduling for Computation Tasks: 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 does not have optimum size. 2) Possible stalls due to arbitration of shared resources.

The problem of predicting and modeling resource sharing is challenging for SS. A static block partition aggravates the problems, especially when the number of processors (P) is increased. Despite the simplicity and regular behavior in computation and data access of DMM, the use of static techniques is not enough to overcome these problems. DS is a feasible solution able to alleviate the scalability problems of SS if their overhead is managed properly. We propose a distributed scheduling system approach where the computation of optimum size tiles in matrix C are scheduled dynamically using atomic in-memory operations, particularly atomic addition. A pseudo-code is presented on Figure 3.

The proposed DS has the following advantages over SS:

1) A carefully designed DS can be managed with low overhead using atomic *in-memory* operations, specifically, atomic increment/decrement. In-memory operations can complete in very few cycles (e.g. 3 cycles on C64 if they are pipelined properly), allowing more

```
01: // Globals
02: int TotalNumTasks
03: int TaskIndex = 0
04: ...
05: // DS running on each Processor
06: int i
07: i = AtomicAdd(&TaskIndex, 1)
08: while i<TotalNumTasks do
09:   ComputeTile(i)
10:   i = AtomicAdd(&TaskIndex, 1)
11: end while
12: ...
```

Figure 3: Code Fragment for a DS implementation

requests to be completed per unit of time and avoiding unnecessary roundtrips to memory [21].

2) The dynamic approach load-balances optimally in the presence of stalls due to arbitration of shared resources, increasing the efficiency by keeping all threads working.
3) Since the work unit is the optimal size tile, the number of non-optimum size tiles is minimized and it does not depend on the number of processors P.

The first and second advantages suggest that DS will have a better performance than SS and the maximum performance will be reached when the amount of data is big enough to feed all processors in parallel.

The third advantage implies that DS will overcome SS especially when the size of matrix m is limited or the number of processors P increases. Modern many-core architectures include in-memory computation capabilities allowing little contention and overhead due to the use of DS as described in Figure 3.

Due to the advantages explained before, it is feasible to expect a better scalability when using DS for a broad range of values of m and P.

2) Percolation in the computation task: Most of the time is spent computing tiles. Therefore, computation deserves special attention. A high performance tiling for DMM used in many-cores with software managed memory hierarchies computes a single tile c of size $L_1 \times L_2$ using the outer product of tiles a and b of sizes $L_1 \times 1$ and $1 \times L_2$ respectively, allowing maximum reuse of elements in a and b tiles and fitting a, b and c in registers. Instruction scheduling could be used to partially hide the latencies incurred while moving the operands a and b from on-chip memory to registers but it is not enough [17].

A proper interleaving of task is required to avoid stalls due to data movement. We propose percolation as a feasible solution. Percolation is the process by which data is moved across the levels of the memory hierarchy to meet locality requirements for computation but as opposed to data prefetching, computation tasks are not scheduled until the percolation operation has not finished, e.g. there is explicit synchronization between data movement tasks and computation tasks.

```
S1: c[1..L1][1..L2] = 0
S2: for k = 1 to m, k++
S3:   a[1..L1][1] = A[i..i+L1][k]
S4:   b[1][1..L2] = B[k][j..j+L2]
S5:   c[1..L1][1..L2] += a[1..L1][1] × b[1][1..L2]
S :  end for
S6: C[i..i+L1][j..j+L2] = c[1..L1][1..L2]
```

Figure 4: Computation of a tile C with size $L_1 \times L_2$ without Percolation

```
S1 : c[1..L1][1..L2] = 0
S2 : a[1..L1][1] = A[i..i+L1][1]
S3 : b[1][1..L2] = B[1][j..j+L2]
S4 : for k = 1 to m, k++
S5 :   a[1..L1][2] = A[i..i+L1][k+1]
S6 :   b[2][1..L2] = B[k+1][j..j+L2]
S7 :   c[1..L1][1..L2] += a[1..L1][1] × b[1][1..L2]
S8 :   k++, if k == m then break
S9 :   a[1..L1][1] = A[i..i+L1][k+1]
S10:   b[1][1..L2] = B[k+1][j..j+L2]
S11:   c[1..L1][1..L2] += a[1..L1][2] × b[2][1..L2]
S : end for
S12: C[i..i+L1][j..j+L2] = c[1..L1][1..L2]
```

Figure 5: Computation of a tile C with size $L_1 \times L_2$ with Percolation

Figure 4 shows the pseudo-code for calculating one tile C of size $L_1 \times L_2$ without Percolation. Uppercase variables are arrays in shared on-chip memory and lowercase variables are arrays of the same operand allocated in registers. Inside the for loop, there are two tasks: Data movement $(S3 - S4)$ and computation $(S5)$. To eliminate stalls due to latency, we percolate operands a and b into registers using loop unrolling in the calculation of the tile. Figure 5 show the pseudo-code with Percolation (PC).

With proper percolation of data between on-chip memory and registers we can significantly increase the time between the issue of loads for operands a and b and computations where they are required. A basic analysis of the unrolled loop show us that the required number of registers increases from $L_1 + L_2 + L_1 \cdot L_2$ without percolation to $2(L_1 + L_2) + L_1 \cdot L_2$ with percolation. In the general case, the number of iterations unrolled depends on the latency of memory operations.

For the particular case of C64, the optimum tile is $L_1 = L_2 = 6$ [17]. It has been shown that this tiling consumes a bandwidth below crossbar saturation. Of the available 63 register of C64, 5 registers are used for pointer and indexes leaving 58 registers for computation. A careful live variable analysis shows that registers used to store a and b vectors in one iteration can be reused in another iteration. Therefore, we were able to retain a 6×6 tile without spilling registers.

Instruction-cache misses will produce costly stalls in the execution while instructions are accessed from main memory. The case for many-core architectures imposes additional constrains because I-caches are shared. Executing the same code by the processors that share the same I-cache is desirable. For the particular case of DMM, the most used

```
1 : Initialize C_{i,j} to 0 on SRAM
2 : Compute the block C_{i,j} = Σ_{k=0}^{m/n-1} A_{i,k} · B_{k,j}.
    This can be subdivided in 2 subtasks:
2a:    Copy A_{i,k} and B_{k,j} from DRAM to SRAM.
2b:    Compute a partial C_{i,j} and accumulate.
3 : Copy Back the block C_{i,j} calculated.
```

Figure 6: Tasks for computing one block $C_{i,j} \in C$

code is the dynamic scheduler and the code for computing a tile. Two well know strategies can be applied to minimize I-cache misses. The first one is to align the functions of the DS and Tile computation with the I-Cache block size, minimizing the number of cache blocks for that code. The second one is to apply Instruction Percolation (IP), which can be done by executing the DS and Tile computation code prior to the execution of the whole DMM, allocating that code in the shared I-Caches, and reducing the excessive number of I-misses on the first iterations.

B. Dynamic Percolation

A highly optimized DMM algorithm in on-chip memory is limited by the size of operands it can handle (e.g. matrices of 400×400). In this section, we extend DMM into off-chip memory by blocking at on-chip memory level and using our percolated DMM algorithm. We assume that the target many-core architecture has no hardware mechanisms for transferring data blocks (e.g. DMA engines or Caches). Then, we use Thread Units (TUs) to transfer the data. These data movement TUs must be orchestrated with the computational TUs in order to enforce data dependencies: the computation cannot be done before the matrices are loaded and a matrix result cannot be stored until work using it is completed. Further, TUs working on data movement need to help with computation if there is no data to move.

A straightforward static schedule for the DMM algorithm detailed in section III would synchronize tasks using barriers and would parallelize each task. To compute the whole matrix C, the tasks detailed in Figure 6 would be executed $\frac{m^2}{n^2}$ times.

Although task 2b is implemented efficiently, as described in section IV-A, a direct implementation of task 2, with barriers between tasks 2a and 2b, would waste resources while TUs are waiting on barriers. Further, it would be inefficient for all TUs to copy data concurrently given the limited off-chip memory bandwidth. A dynamic scheduling approach can replace the barriers with finer-grained signals while still enforcing data dependencies.

We define *Dynamic Percolation*, where the assignment of data movement tasks and computation tasks is done dynamically. There are two types of threads: Helper Threads and Computation Threads. 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 on-chip memory (e.g. buffers $F1$ and $F2$). Moreover, the distribution of computation tasks and data movement tasks changes dynamically. Dynamic Percolation follows a simple set of rules for creation and issue of tasks based on their dependencies. These rules help the dynamic scheduler to keep threads working efficiently on a computation task or a data movement task. The rules are as follow:

1) Task Creation rules:
   a) A set of computation tasks on buffer $F1$ is *created* and ready to be fired when all the data movement tasks for buffer $F1$ are complete. The same is true of buffer $F2$.
   b) A set of data movement tasks for buffer $F1$ is *created* and ready to be fired when computation is complete for the data buffer $F1$. The same is true of buffer $F2$.
2) Task Issue rules:
   a) A set of tasks (computation or data movement) is *scheduled* dynamically between the threads that belong to a set of that type of task (CT or HT).
   b) When a HT has finished and all data movement tasks of a buffer have been issued, the HT *becomes* a CT for the current actively computed buffer.
   c) There is a maximum number of HT that can run in parallel to avoid contention on off-chip memory.
   d) When a CT has finished and all tasks of that set (e.g. on buffer $F1$) have been issued, it *becomes* a HT for the set of data movement tasks on that buffer (e.g. on buffer $F1$) if the maximum number of HT has not been reached. Otherwise it becomes a CT for the next set of computation tasks (e.g. on buffer $F2$).

The Dynamic Scheduler for each set of tasks can be implemented efficiently by using atomic in-memory operations, specifically, in-memory atomic addition. The main advantage of this implementation is the low overhead given by the low latency of in-memory operations compared with an atomic operation that required to load the data, to perform the operation and to save back. Under normal conditions (e.g. no unexpected failures of any components in the chip) a possible scenario for stalls is given by rule 2d: a CT stalls when it becomes a CT of the next set of computation tasks (e.g. buffer $F2$) while the buffer's data movement tasks (e.g. buffer $F2$) have not finished. This condition can be easy solved if the size of the HT set is large enough to guarantee that the data movement tasks finish before their associated computation tasks. This parameter is architecture dependent and it is related to the compute/bandwidth ratio and the size of on-chip buffers. An optimistic estimation is given by eq. (2), where N_{HT} is the maximum number of helper threads:

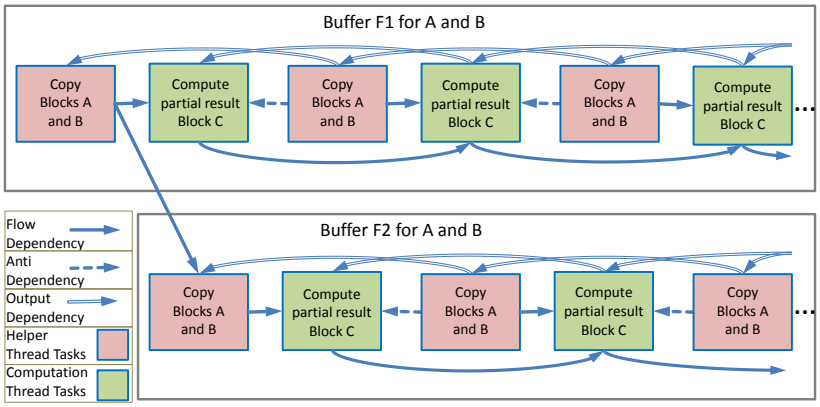


Figure 7: Dynamic Percolation for Computation of one block $C_{i,j}$

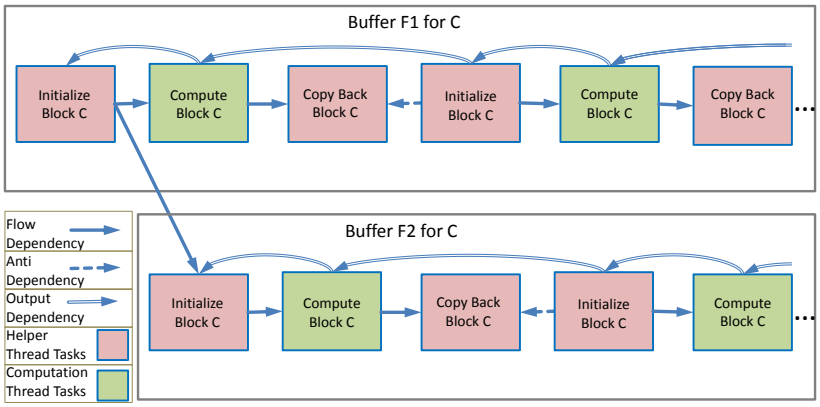


Figure 8: Dynamic Percolation for Computation of matrix C

$$\frac{DataMoved}{Bandwidth(N_{HT})} \leq \frac{FLOPS\ Computed}{Performance(P - N_{HT})} \quad (2)$$

There are two basic structures common to several algorithms where Dynamic Percolation can be applied. The first structure is *Copy - Compute* and the second one is *Copy - Compute - Copy Back*. These structures can be applied hierarchically without restriction. DMM is a good case to expose the advantage of Dynamic Percolation because the algorithm exposes both structures on a hierarchical fashion. The tasks in Figure 6 can be classified into two groups: 1) Computation tasks (2b) and 2) Data movement tasks (1, 2a, 3). Also, there is a hierarchy of tasks. At the highest level, tasks $1-3$ are related with blocks $C_{i,j}$ (Initialize, Compute, Copy Back) while at the next level down, tasks $2a$ and $2b$ are specific for computing one block $C_{i,j}$ using several blocks $A_{i,k}$ and $B_{k,j}$ (Copy, Compute). We will analyze each level separately, starting with the inner level: tasks for computing a block $C_{i,j}$, and continuing with the outer level: tasks for computing the whole matrix C.

Computation of one block $C_{i,j}$: Data is percolated as shown in Figure 7. Tasks 2a map to the data movement tasks and tasks 2b map to the computation tasks. In the initialization step, we create the first set of data movement tasks and create the second set when all data movement tasks in first set have been issued.

Computation of matrix C: Computing the whole matrix involves a *Hierarchical Dynamic Percolation*, where tasks 2 are a subset of the percolation model for tasks 1 - 3 as shown in Figure 8. However, at this level, all tasks in task 2 are considered computation tasks. There are two data movement tasks (1 and 3) where task 1 of the next outer loop iteration is dependent on task 3 of the current iteration. In the initialization step, we only initialize $C_{i,j}$ and do not copy it back until the first computation task is completed.

Under the assumption that the number of HTs at both levels have been chosen properly to do the data movement tasks in less time than the computation tasks, the Dynamic Percolation for MM not only allows runtime redistribution between helper threads and computational threads to achieve better utilization of TUs, but also its dynamic behavior can efficiently manage the unpredictable effects of resource sharing (e.g. arbitration of crossbar network ports and limited off-chip bandwidth). This is a challenging problem on many-core architectures that, as discussed previously, SS cannot overcome.

The performance of the off-chip memory DMM with

respect to the on-chip memory DMM is expected to be slightly lower because now some threads are not doing computation and the cost of data movement has to be included. This cost depends on the maximum number of HTs allowed at each task level, the bandwidth for memory transfers, and the size of blocks on SRAM.

V. Experimental Evaluation

We have implemented and tested the techniques exposed across this paper using the DMM as an example. The optimization of DMM exposes all the challenges and difficulties we addressed through Percolation and Dynamic Percolation in section IV. For our evaluation, we used the C64 architecture described in section II-A. The software managed memory hierarchy, hundreds of independent hardware TUs and plenty of shared resources are a good fit to test the advantages of the techniques proposed given the difficulties to optimize applications under these new type of many-cores.

First, we used the on-chip memory algorithm to compare the scalability of SS vs. DS without percolation operations. Figure 9 shows that the performance of SS is drastically decreased for smaller matrices with respect to DS. In general, SS always has lower performance than DS. The low performance of SS for small matrices is a major limitation to the off-chip memory algorithm because the use of extra buffers limits even more the size of them inside on-chip memory.

Figure 10 shows the scalability of SS and DS in terms of number of TUs. While DS scales near to linear for big matrices (e.g. the maximum size that fits in on-chip memory is $m = 486$) and sustains an increased performance even when the size is drastically reduced (e.g. $m = 100$), SS cannot follow the same rate of improvement and decreases its performance if the matrix size is small and the number of threads increases. Also, Figure 10 shows the results of the progressive improvements made for the computation task unit (the register tile). The latency of memory operations is hidden again, avoiding register spilling and resulting in an speedup of 25%. The maximum performance after the improvements is 70.00GFLOPS with matrices of 486×486 using 156 TUs: 87.50% of the theoretical peak performance of C64. 4 TUs are reserved for communication with other chips and the execution of the runtime.

Using the already optimized DMM in on-chip SRAM, a fully parallel DMM that uses off-chip DRAM was implemented without overlapping computation tasks and data movement tasks. Also, we implemented the Dynamic Percolation proposed using the optimized computation task and 24 HTs. Their performance for different number of threads is shown on Figure 11. Furthermore, the data movement tasks where also optimized for the known on-chip block size and the efficient transposition of matrix A required by the computation tasks. As a result, only 8 HTs were needed, increasing the performance due to the larger number of

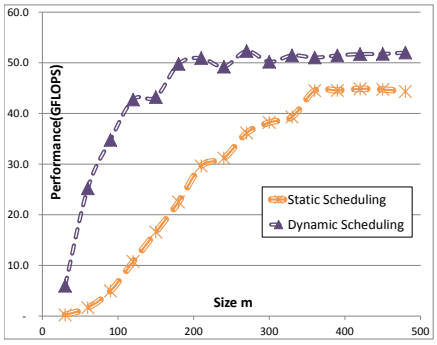


Figure 9: Scalability for Matrix Size with 156 TUs

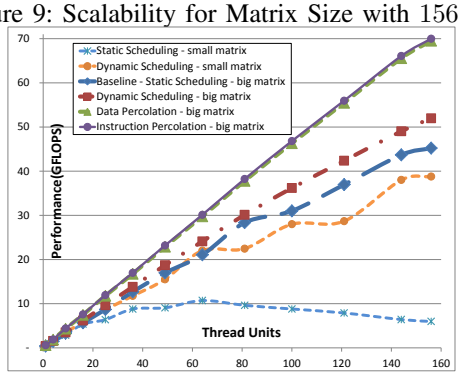


Figure 10: Performance of DMM in on-chip SRAM

TUs available for computation. The maximum performance provided by the dynamic percolation and the optimized computation tasks and data movement tasks is 65.63GFLOPS with matrices of 6336×6336 using 156 TUs: 82.02% of the theoretical peak performance of C64. Also we provide evidence of the power-efficiency of our implementation, reaching 1.0 GFLOPS/W for this case.

Finally, the impact of the maximum number of HTs is shown in Figure 12 for a matrix of size 5280×5280 using 156 TUs. Clearly, when the data movement tasks are not optimized, more HTs are required. Otherwise, the data movement task takes longer than its computational task associated causing the computation to stall. In addition, if the HTs increase too much, the bandwidth saturation will decrease slightly the performance.

VI. Related Work

Work that is directly related to this research has been detailed in section IV. From the many-core architecture side, static approaches have also been proposed for regular applications on linear algebra and signal processing. These applications have shown significant speedup but they are far from the peak performance [6], [23]. Also, percolation has been been studied and applied on irregular applications [27], [28]. Other studies have been focused on semi-automatic mechanisms for doing percolation at tile-level [13].

For other parallel architectures (e.g. CellBE, GPGPU, Intel), extensive work has been done for the optimization

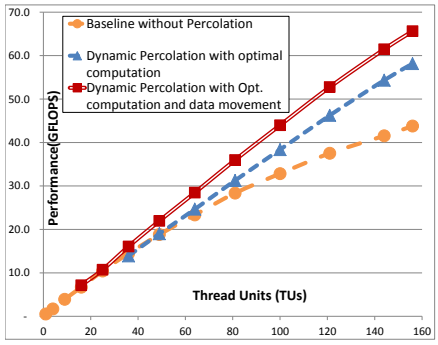


Figure 11: Scalability of MM in off-chip DRAM for $m = 6336$

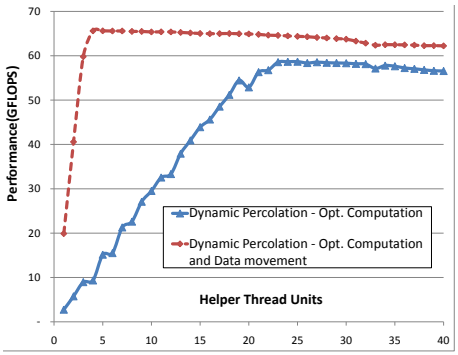


Figure 12: Impact of the maximum number of Helper Threads using 156 TUs

of linear algebra applications exploiting patterns in the algorithms but missing architectural parameters [20], [12]. Most of the implementations have used tuned versions of specialized libraries/subroutines provided by the vendors (e.g. GEMM/BLAS, CUBLAS, MKL) [7], [11], [3].

Other studies have been focused on models that capture performance-relevant aspects of the hierarchical nature of computer memory. Among those, we found the Uniform Memory Hierarchy (UMH) model or the Parallel Memory Hierarchy (PMH) model [1], [2], the increase of shared resources inside a single chip have made unfeasible to model many-cores with high accuracy using an scalable framework.

This publication does not study in deep the impact in energy efficiency but parallel efforts made at University of Delaware have shown an scalable and simple energy consumption model for modern many-cores [15] and how similar techniques can be used to improve energy efficiency [14].

VII. Conclusions and Future Work

We have analyzed some of the difficulties on modern many-core architectures to reach high performance using static techniques. To overcome those difficulties, we took advantage of the fine-grain synchronization primitives of many-cores to define percolation operations in order to schedule data movement properly. In addition we have fused percolation operations with dynamic scheduling into a dynamic percolation approach.

We have shown that a carefully designed Dynamic Percolation using in-memory operations effectively reduces the overhead during execution.

We report experimental results of our methods on a real C64 chip achieving 70.0 and 65.6 GFLOPS for DMM with operands in on-chip and off-chip memory respectively: 87.5% and 82% of the theoretical peak performance of 80 GFLOPs. Also, we provide evidence of the power efficiency of our implementation, reaching 1.09 GFLOPS/W and 993 MFLOPS/W when matrices are on SRAM and DRAM respectively.

Future work will extend percolation to multiple chips and other applications using a dataflow inspired execution model and runtime system [22].

VIII. Acknowledgements

This work has been made possible by the generous support of the NSF through research grants CCF-0833122, CCF-0925863, CCF-0937907, CNS-0720531, and OCI-0904534. It was also partly supported by European FP7 project TERAFLUX, id. 249013 and the Department of Energy [Office of Science] under Award Number DE-SC0008717.

We also want to thank ET International for their collaboration and support on the software and hardware infrastructure provided for the Cyclops-64 Architecture.

References

[1] Alpern, B., Carter, L., Feig, E., Selker, T.: The uniform memory hierarchy model of computation. Algorithmica 12, 72–109 (1992)

[2] Alpern, B., Carter, L., Ferrante, J.: Modeling parallel computers as memory hierarchies. In: In Proceedings Programming Models for Massively Parallel Computers. pp. 116–123. IEEE Computer Society Press (1993)

[3] Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S.: Evaluation and tuning of the level 3 cublas for graphics processors. In: IPDPS. pp. 1–8 (2008)

[4] Bauer, M., Cook, H., Khailany, B.: Cudadma: optimizing gpu memory bandwidth via warp specialization. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 12:1–12:11. SC '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2063384.2063400

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

[6] Chen, L., Hu, Z., Lin, J., Gao, G.R.: Optimizing the Fast Fourier Transform on a Multi-core Architecture. In: IEEE 2007 International Parallel and Distributed Processing Symposium (IPDPS '07). pp. 1–8 (Mar 2007)

[7] Chen, T., Raghavan, R., Dale, J.N., Iwata, E.: Cell broadband engine architecture and its first implementation-a performance view. IBM Journal of Research and Development 51(5), 559–572 (2007)

[8] Denneau, M.: Cyclops. In: Padua, D. (ed.) Encyclopedia of Parallel Computing: SpringerReference (www.springerreference.com). Springer-Verlag Berlin Heidelberg (2011), http://www.springerreference.com/docs/html/chapterdbid/311592.html

[9] Di, P., Xue, J.: Model-driven tile size selection for doacross loops on gpus. In: Euro-Par 2011 Parallel Processing, pp. 401–412. Springer (2011)

[10] Dongarra, J., Gates, M., Jia, Y., Kabir, K., Luszczek, P., Tomov, S.: Magma mic: Linear algebra library for intel xeon phi coprocessors. In: SC12 (2012)

[11] Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. 16(1), 1–17 (1990)

[12] Douglas, C.C., Heroux, M., Slishman, G., Smith, R.M.: GEMMW: A Portable Level 3 Blas Winograd Variant Of Strassen's Matrix-Matrix Multiply Algorithm (1994)

[13] Gan, G., Wang, X., Manzano, J., Gao, G.R.: Tile percolation: An openmp tile aware parallelization technique for the cyclops-64 multicore processor. In: Euro-Par. pp. 839–850 (2009)

[14] Garcia, E., Gao, G.R.: Strategies for improving Performance and Energy Efficiency on a Many-core. In: Proceedings of 2013 ACM International Conference on Computer Frontiers (CF 2013). pp. 9:1–4. ACM, Ischia, Italy (May 2013)

[15] Garcia, E., Orozco, D., Gao, G.R.: Energy efficient tiling on a Many-Core Architecture. In: Proceedings of 4th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG 2011); 6th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC) (2011)

[16] Garcia, E., Orozco, D., Pavel, R., Gao, G.R.: A discussion in favor of Dynamic Scheduling for regular applications in Many-core Architectures. In: Proceedings of 2012 Workshop on Multithreaded Architectures and Applications (MTAAP 2012); 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2012). pp. 1591–1600. ACM, Shanghai, China (May 2012)

[17] Garcia, E., Venetis, I.E., Khan, R., Gao, G.: Optimized dense matrix multiplication on a many-core architecture. In: Proceedings of the Sixteenth International Conference on Parallel Computing (Euro-Par 2010). Ischia, Italy (2010)

[18] Heinecke, A., Trinitis, C.: Cache-oblivious matrix algorithms in the age of multicores and many cores. Concurrency and Computation: Practice and Experience pp. n/a–n/a (2012), http://dx.doi.org/10.1002/cpe.2974

[19] Hu, Z., del Cuvillo, J., Zhu, W., Gao, G.R.: Optimization of Dense Matrix Multiplication on IBM Cyclops-64: Challenges and Experiences. In: 12th International European Conference on Parallel Processing (Euro-Par 2006). pp. 134–144. Dresden, Germany (Aug 2006)

[20] Hyuk-Jae Lee and James P. Robertson and José A. B. Fortes: Generalized Cannon's algorithm for parallel matrix multiplication. In: Proceedings of the 11th International Conference on Supercomputing (ICS '97). pp. 44–51. ACM, Vienna, Austria (1997)

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

[22] Orozco, D., Garcia, E., Pavel, R., Gao, G.: TIDeFlow: The Time Iterated Dependency Flow Execution Model. In: Proceedings of Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2011); 20th International Conference on Parallel Architectures and Compilation Techniques (PACT 2011). pp. 1–9. IEEE Computer Society, Galveston Island, TX, USA (October 2011)

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

[24] Reinders, J.: Parallel Programming and Optimization with Intel Xeon Phi coprocessors. Colfax International, 1st edn. (2013)

[25] Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.m.W.: Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. pp. 73–82. PPoPP '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1345206.1345220

[26] Stock, K., Pouchet, L.N., Sadayappan, P.: Automatic transformations for effective parallel execution on intel many integrated core. In: TACC-Intel Highly Parallel Computing Symp (2012)

[27] Tan, G., Fan, D., Zhang, J., Russo, A., Gao, G.R.: Experience on optimizing irregular computation for memory hierarchy in manycore architecture. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. pp. 279–280. ACM, New York, NY, USA (2008)

[28] Tan, G., Sreedhar, V.C., Gao, G.R.: Just-in-time locality and percolation for optimizing irregular applications on a manycore architecture. In: LCPC. pp. 331–342 (2008)

[29] Wei, L.Y.: Tile-based texture mapping on graphics hardware. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. pp. 55–63. HWWS '04, ACM, New York, NY, USA (2004), http://doi.acm.org/10.1145/1058129.1058138

[30] Xu, C., Kirk, S., Jenkins, S.: Tiling for performance tuning on different models of gpus. In: Information Science and Engineering (ISISE), 2009 Second International Symposium on. pp. 500–504 (2009)