University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Dynamic Percolation - Mapping Dense Matrix Multiplication on a Many-Core Architecture

Elkin Garcia§
Rishi Khan‡
Kelly Livingston§
Ioannis E. Venetis†
Guang R. Gao§

CAPSL Technical Memo 098
June, 2010

Copyright © 2010 CAPSL at the University of Delaware

§ University of Delaware
{egarcia, kelly, ggao}@capsl.udel.edu
‡ ET International
rishi@etinternational.com
† University of Patras
venetis@ceid.upatras.gr

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Contents

1 Introduction **1**

2 Background and Motivation **1**
2.1 The IBM Cyclops-64 Architecture 1
2.2 Off-chip Memory Bandwidth Bottleneck 2
2.3 Static Scheduling and Percolation for MM 3

3 Problem Statement **4**

4 Dynamic Scheduling: Scalability, Performance Tuning and Percolation **5**
4.1 Scalability and Performance Tuning of MM in SRAM 5
4.2 Dynamic Percolation 7

5 Experimental Evaluation **10**

6 Related Work **13**

7 Conclusions and Future Work **13**

List of Figures

1 C64 Architecture details 2
2 Bandwidth bottleneck for MM 3
3 Partition Schemes for a matrix C of 15×15 with tiles of 3×3 4
4 Algorithm for computing a tile of C with size $L_1 \times L_2$ 7
5 Tasks for computing one block $C_{i,j} \in C$ 8
6 Dynamic Percolation 9
7 Dynamic Percolation for MM 10
8 Performance and Scalability of MM using on-chip SRAM 11
9 Performance and Scalability of MM using off-chip DRAM 12

Abstract

This paper continues the line of work reported in [1] where a series of steps using static analysis and scheduling techniques were presented on the IBM Cyclops-64 (C64) many-core-on-a-chip architecture where they showed a performance of 44.12 GFLOPS. An open question is: Can the performance and scalability be drastically improved again, and if so, how? The answer (to be detailed in this paper) is that it can. However, much to our surprise, it cannot be achieved by using static techniques alone.

In this paper, two types of tasks are employed: computation tasks and data movement tasks. A computation task is *dynamically* enabled and assigned to a thread unit to compute a tile (of the result matrix). The size of the tile is determined by a number of machine parameters: e.g. size of available registers and crossbar bandwidth. A data movement task is *dynamically* enabled and assigned to a helper thread unit to move inputs and results between on-chip SRAM memory and off-chip DRAM memory. This process is called dynamic percolation. Furthermore, the distribution of computation tasks and data movement tasks will vary during the course of dynamic percolation. Therefore, our method allows runtime redistribution between helper threads and computational threads to achieve better utilization of thread unit resources. Our dynamic percolation method mitigates the unpredictable effects of resource sharing (e.g. arbitration of crossbar network ports and limited off-chip bandwidth) and the runtime redistribution of the available threads between the two types of tasks, a formidable challenge for a static scheduling method. The overhead of dynamic percolation is reduced effectively by a carefully designed lock-free queue implementation.

We report experimental results of our methods on a real C64 chip achieving 70.0 and 65.6 GFLOPs for MM with operands in SRAM and DRAM 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.

1 Introduction

This paper continues the line of work reported in [1]. They optimized the Dense Matrix Multiplication (MM) on the IBM Cyclops-64 (C64) many-core-on-a-chip architecture using static analysis and static scheduling techniques. They reached slightly more than one half of the peak performance after several hand-coded optimizations when the matrices are in on-chip (SRAM) memory. Furthermore, our findings show that dynamic scheduling (DS) has several advantages over static scheduling (SS) in spite of the regularity in computation and memory accesses of MM.

Two types of tasks are employed for the MM, computation tasks and data movement tasks. A computation task is a very small fine grained unit of work which is dynamically enabled and assigned to a computation thread (CT). The tile size is parameterized by the size of available registers and optimized for reuse and bandwidth. The computation task is also tuned by well known static techniques. A data movement task, which is also optimized for bandwidth, is dynamically enabled and assigned to a Helper Thread (HT) to move operands between SRAM and DRAM. The orchestration between these two types of tasks is called Dynamic Percolation. Dynamic Percolation allows runtime redistribution between HTs and CTs to achieve better utilization of Thread Unit resources. Moreover, our method manages the unpredictable effects of resource sharing (e.g. arbitration of crossbar network ports and limited off-chip bandwidth). The potential overhead of a dynamic centralized scheduler is mitigated by atomic in-memory task assignments supported by hardware. In contrast, SS can not properly manage these unpredictable effects given the amount of shared resources, the increasing number of threads and the complex interaction of all these components at runtime on a many-core architecture.

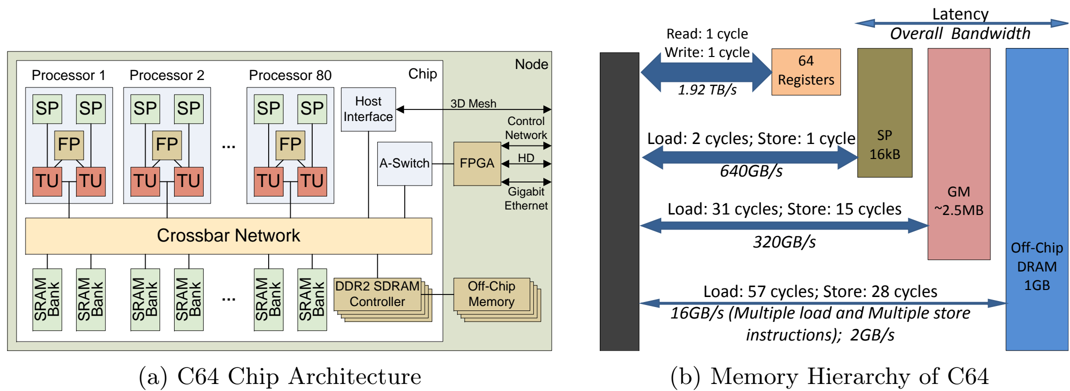
We report experimental results of our methods on a real C64 chip achieving 70.0 and 65.6 GFLOPS for MM with operands in SRAM and DRAM respectively: 87.5% and 82% of the theoretical peak performance of 80 GFLOPs. Also we provide evidence of the power efficiency of our implementation, reaching 1.09 GFLOPS/W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

The paper is organized as follows: Section 2 presents background and motivation on MM and their optimizations. Section 3 defines the problem addressed in this paper. Section 4 shows the optimizations for tasks and scheduling we propose. Section 5 presents our results and talks about the effectiveness of our approach. Section 6 presents other related work in the field. Finally, Section 7 presents our conclusions and possible directions for our future work.

2 Background and Motivation

2.1 The IBM Cyclops-64 Architecture

As previously mentioned, Cyclops-64 (C64) is a homogeneous many-core system on a chip architecture designed by IBM. A full system consists of a back end of C64 nodes connected to a linux front end for system monitoring and file I/O. Each node consists of 1 GB off-chip DRAM mem-



(a) C64 Chip Architecture (b) Memory Hierarchy of C64

Figure 1: C64 Architecture details

ory, a C64 chip, and an FPGA to aid in boot operations and to possibly extend the functionality of a blade. A C64 chip is an aggregation of 160 simple MIMD execution units, 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, which is specified as having 2 Thread Units(TU), a floating point unit (FPU), two 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 processors is an I-cache of 32 KB for storing code. There are 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 SRAM banks [2].

A C64 chip has an explicit three-level memory hierarchy: scratchpad memory, on-chip SRAM, off-chip DRAM. 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. The latency ratios of the memory hierarchy are listed above. 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.

2.2 Off-chip Memory Bandwidth Bottleneck

The most obvious and embarrassingly parallel way to distribute the MM workload on any multi/many-core processor is to divide the C matrix into subproblems and assign these problems to a core. However, with C64 you quickly reach a problem. Given that you have 16 GB/s of bandwidth to off-chip DRAM and 80 GFLOPS of compute power, the minimum block size to overcome this gap is very large, as Figure 2 indicates. As you can see, the smallest possible block

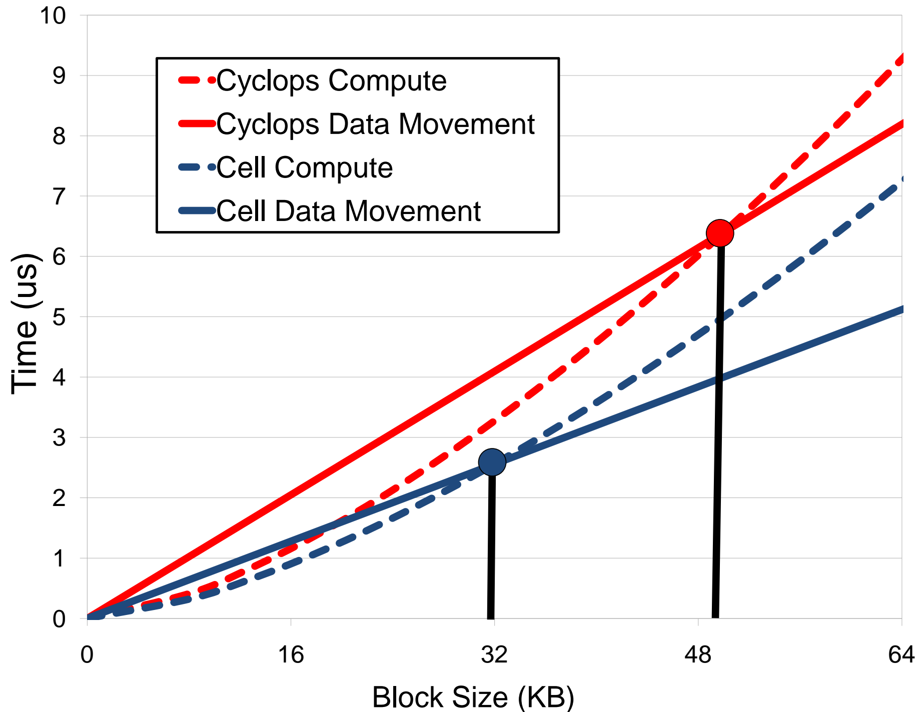


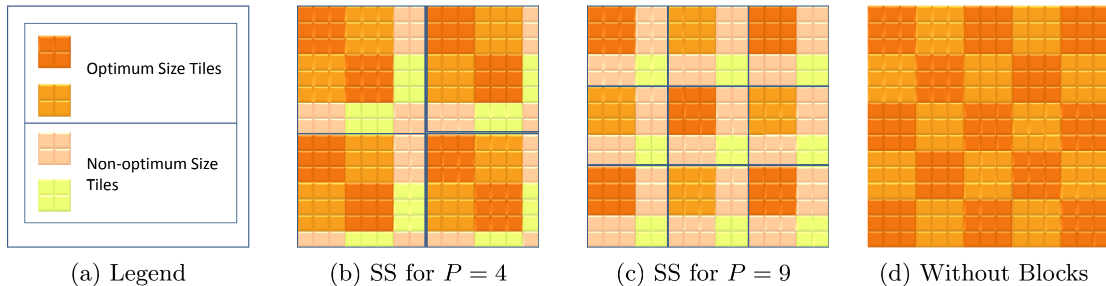
Figure 2: Bandwidth bottleneck for MM

size C64 can not fit in any configuration that the SPM could be configured, the total opposite situation of most processors such as Cell which can easily fit the blocks in its local storage, making it embarrassingly parallel to block for MM. Unlike Cell, TUs on C64 must find a way to share memory on the chip to overcome the high FLOPS/Bandwidth ratio of 5 flops/ byte / second, the largest ratio we have seen to date on any general purpose processor.

2.3 Static Scheduling and Percolation for MM

Since we resolve that the blocking level must be higher than at the SPM level, we exam work already accomplished at the SRAM level. A series of steps using static analysis on the C64 many-core-on-a-chip architecture studied in [1] reached more than a half of the theoretical peak performance of the chip. Their main contribution was to alleviate the impact of the memory latencies and limited bandwidth with an optimal register tiling that minimizes the number of memory operations in SRAM. Work was scheduled statically in equal sized blocks among all cores. Blocks were further divided into tile-sized computational tasks.

Because the blocks are not necessarily multiples of the Optimal Tile Size (OTS), there is exposure to non-optimal sized tiles at the trailing boundaries of each block. These smaller tiles incur an extra overhead due to the lack of register reuse and insufficient latency hiding. Additionally, when many processors share a small fixed amount of on-chip memory, the block size decreases and the proportion of non-optimal work increases. Avoiding the partition in blocks,



(a) Legend (b) SS for $P = 4$ (c) SS for $P = 9$ (d) Without Blocks

Figure 3: Partition Schemes for a matrix C of 15×15 with tiles of 3×3

and taking a tile as the computation task unit minimizes the impact of non-optimum size tiles. Figure 3 shows an example for the amount of non-optimum tiles (higlighted). The worst scenario is when number of processors P is increased and the best one is when blocks are not used.

Yet, a non-optimal decomposition of the work is not the only impediment to performance. SS assumes that TUs doing the same amount of work will complete at the same time. However, in a complex many-core system such as C64, this is not the case. Many-core architectures have plenty of shared resources such as FPUs, crossbar, memory, and I-Caches that can produce unexpected stalls. Additionally, every TU has the ability to move data and calculate tiles unlike traditional architectures where DMA logic or cache logic could not perform floating point calculations when idle. Thus, for peak performance, we should never have these TUs idle. And scheduling work for all TUs, whether data movement or computation, cannot be known statically in the face of these runtime variables.

3 Problem Statement

Our target operation is the dense MM $C = A \times B$ for matrices with size $m \times m$. We propose a separation of the problem into two orthogonal subproblems: optimizing MM in SRAM and moving data between DRAM and SRAM. We are interested in answering the following questions:

1. How do we efficiently schedule and compute tasks in SRAM? This question can be subdivided into 2 parts:
   (a) How can we load balance tasks across the machine (e.g. computation and memory access) by solving the task scheduling problem and the task assignment problem?
   (b) What is a desirable task partition that fully exploits data reuse and how can we compute that task efficiently given the constraints of register file size and crossbar bandwidth?
2. How do we extend MM in SRAM to solve larger MM problems in DRAM?

To extend the matrices to DRAM 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 SRAM. This is similar to the blocking performed in traditional cache hierarchies. Yet, C64 has no automation of data movement by caches or DMA engines but instead must use TUs that could compute instead. This means we have a trade off between compute 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.

4 Dynamic Scheduling: Scalability, Performance Tuning and Percolation

This section will explain our findings and solution to the questions raised on Section 3. As explained before, a MM 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 given in each case: A load balanced scheduler with low overhead and and optimized and tuned computation task

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.

4.1 Scalability and Performance Tuning of MM in SRAM

As mentioned before, this paper continues the line of work reported in [1] where they applied a series of steps using static analysis and static scheduling. We already have pointed out the disadvantages of SS from the many-core perspective, and we will explain how DS can improve performance over SS using a register tile as a computational unit of work. After that, we will explore several techniques for tuning the performance of this task.

4.1.1 Dynamic Scheduling for Computation Tasks

SS is suboptimal because it does not consider 2 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 unpredictable effects of resource sharing are a formidable challenge for SS and a static block partition exacerbates problems especially when the P is increased. Despite the simplicity and regular behavior in computation and data access of MM, static techniques cannot overcome these problems. At that point, 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 C are scheduled dynamically using a lock-free queue. The proposed DS has the following advantages over SS:

1. A carefully designed queue can be managed with low overhead using atomic in-memory operations, specifically atomic increment/decrement.
2. The dynamic approach load balances optimally in the presence of stalls due arbitration of shared resources, increasing the efficiency by keeping all threads working.
3. Since the work unit is the optimal tile size, then 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 as long as there is not contention in the queue which is satisfied by the first advantage. In other words, DS would have better scalability than SS for m and P.

4.1.2 Tuning of the computation task

The majority of the computation is computing tiles. Therefore, computation deserves special attention. The Instruction Scheduling made by Garcia et. al. [1], only partially hides the latency incurred while fetching the operands a and b from SRAM to registers.

To eliminate stalls due to latency, we prefetch operands a and b into registers using loop unrolling in the calculation of the tile. Figure 4 shows the pseudo-code for calculating one tile C of size $L_1 \times L_2$ with and without Data Prefetching (DP). Uppercase variables are arrays in share memory and lowercase variables are arrays allocated in registers.

Unrolling the loop once and carefully using instruction scheduling, we can significantly increase the time between the issue of loads for operands a and b and computations where they are required. In [1] we showed that $L_1 = L_2 = 6$ consumes a bandwidth below crossbar saturation. 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 DP to $2(L_1 + L_2) + L_1 \cdot L_2$ with DP.

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

```
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]
```

(a) Without data prefetching

```
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]
```

(b) With data prefetching

Figure 4: Algorithm for computing a tile of C with size $L_1 \times L_2$

and b vectors in one iteration can be reused in another one. Therefore, we were able to retain a 6×6 tile without spilling register.

With respected to I-Caches. Instruction misses will produce costly stalls in the execution while instruction are accessed from main memory. The case for many-core architectures imposes additional constrains because I-caches are shared and executing the same code by the processors that share the I-cache is desirable. For the particular case of MM, the most used code is the DS and the code for computing a tile. Two well know strategies can be applied for minimizing the I-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 Prefetching (IP), it can be done executing the DS and Tile computation code prior to the execution of the whole MM, allocating that code in the shared I-Caches reducing the excessive number of I-misses on the first iterations.

4.2 Dynamic Percolation

A well tuned MM algorithm in SRAM is severely limited in size (i.e. 500x500). Here we extend MM into DRAM by blocking at the SRAM level and using our tuned MM algorithm in SRAM. However, because C64 has no hardware mechanisms for block transfer (e.g. caches or DMA engines), we must use TUs to transfer the data. These computational TUs must be scheduled with data movement TUs to enforce the 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.

An straight forward static schedule for the MM algorithm detailed in section 3 is to synchronize each task needed using barriers and parallelize each task. To compute the whole matrix C, the tasks detailed in Figure 5 are executed $\frac{m^2}{n^2}$ times.

Although task 2b is implemented efficiently as described in section 4.1, a direct implemen-

```
1 :  Initialize C_{i,j} to 0 on SRAM
2 :  Compute the block C_{i,j} = \sum_{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 result of C_{i,j} and accumulate.
3 :  Copy Back the block C_{i,j} calculated
```

Figure 5: Tasks for computing one block $C_{i,j} \in C$

tation 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 at the same time given limited DRAM bandwidth. A dynamic scheduling approach replaces the barriers with finer-grained signals enforcing data dependencies.

We introduce Dynamic Percolation, where data movement tasks and computation tasks are 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 (e.g. buffers X and Y). Moreover, the distribution of computation tasks and data movement tasks will vary in the course of Dynamic Percolation. A set of simple rules for creation (based on data dependencies) and issue of tasks helps the dynamic scheduler keep threads working efficiently on a computation task or a data movement task:

1. Task Creation rules:
   - (a) A set of computation tasks on buffer X is created and ready to be fired when all the data movement tasks for buffer X are complete. The same is true of buffer Y.
   - (b) A set of data movement tasks for buffer X are created and ready to be fired when computation is complete for the data buffer X. The same is true of buffer Y.
2. Task Issue rules:
   - (a) A set of tasks (computation or data movement) are scheduled dynamically between the threads that belong to set of that type of task (CT or HT).
   - (b) When a HT has finished and all data movement tasks of a buffer, it 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 DRAM bandwidth.
   - (d) When a CT has finished and all tasks of that set (e.g. on buffer X) have been issued, it becomes a HT for the set of data movement tasks on that buffer when that set is created presuming that the maximum number of HT has not been reached. Otherwise it becomes a CT of the next set of computation tasks (e.g. on buffer Y).

The Dynamic Scheduling for each set of tasks is implemented by a carefully designed lock-free queue. For C64, we implement this lock-free queue using hardware provided atomic increment

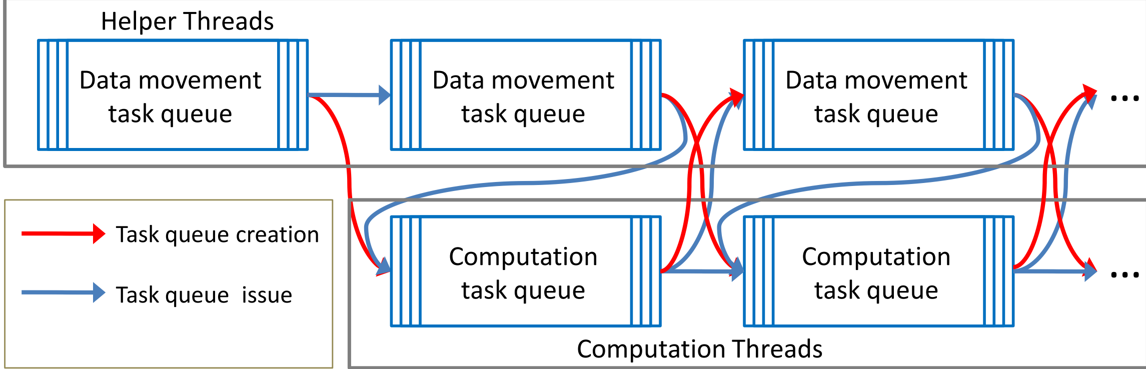


Figure 6: Dynamic Percolation

and decrement operations. Figure 6 shows the rules for the creation and the issue of task queues. 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 Y) but buffer Y's set of data movement tasks 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. A rough starting point estimation is given by (2), where N_{HT} is the maximum number of helper threads:

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

The tasks in Figure 5 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 Figure 6. Task 2a maps to the data movement task queue and tasks 2b maps to the computation task queue. 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 is a subset of the percolation model for tasks 1 - 3 as shown in Figure 7. 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

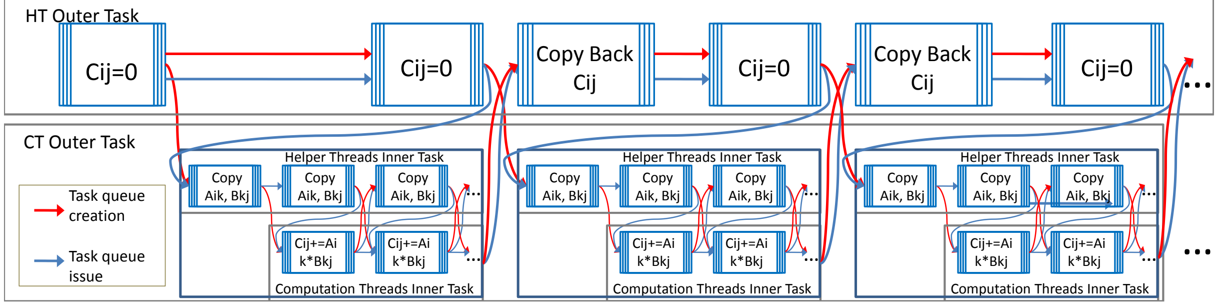


Figure 7: Dynamic Percolation for MM

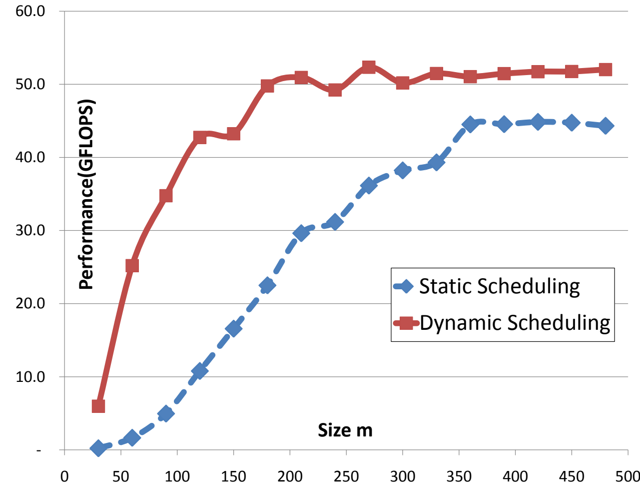
of the current iteration. In the initialization step, we only initialize $C_{i,j}$ and do not copy it back until the first computation task is completed.

Under the assumption that the number of HTs at both levels have been chosen properly for doing 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 DRAM MM with respect to the SRAM MM 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.

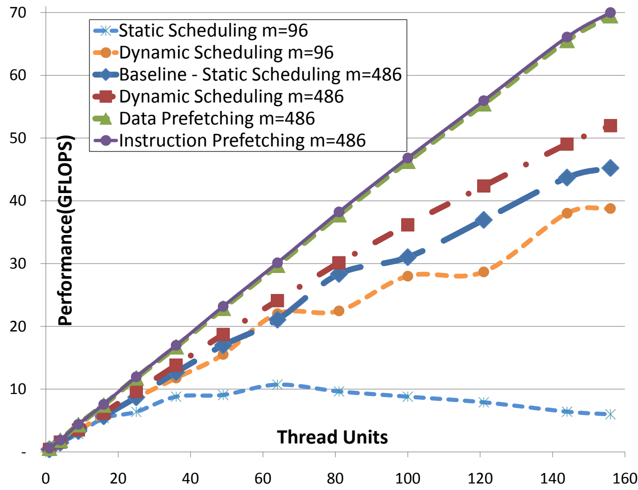
5 Experimental Evaluation

This section describes the experimental evaluation based on the analysis done in section 4 using the C64 architecture. Our baseline parallel MM implementation uses the SS and the set of optimizations described in [1]. Using the MM in on-chip SRAM, we compare the scalability of SS and DS with the matrix size. Figure 8a shows that SS not only has a lower performance than DS, but also its performance is affected drastically for smaller matrices. It is critical for extending our algorithm to off-chip DRAM because the use of buffers for hiding the data movement tasks requires smaller block sizes. Figure 8b shows the scalability of SS and DS in terms of number of TUs. While DS scale near to linear for big matrices (e.g. $m = 486$) and sustain an increasing performance even when the size is drastically reduced (e.g. $m = 96$), SS can not follow the same rate of improvement and decrease its performance if the matrix size is small and the number of threads increases. Also, Figure 8b shows the results of the progressive improvements made for the computation task unit (the register tile), again hiding the latency of memory operations avoiding register spilling gives 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.



(a) Scalability for Matrix Size with 156 TUs

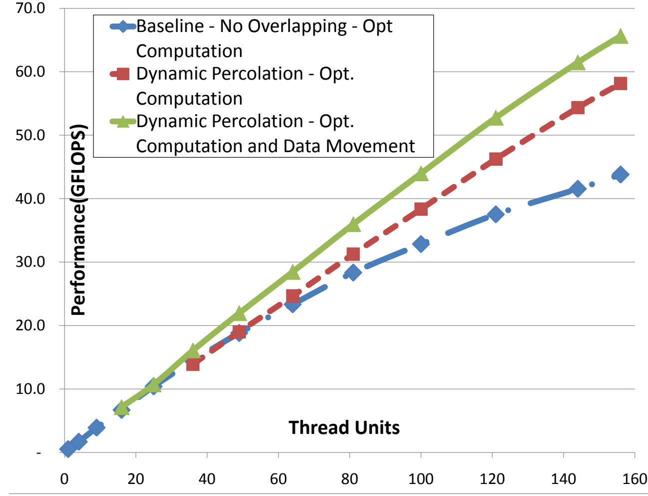


(b) Performance of MM in on-chip SRAM

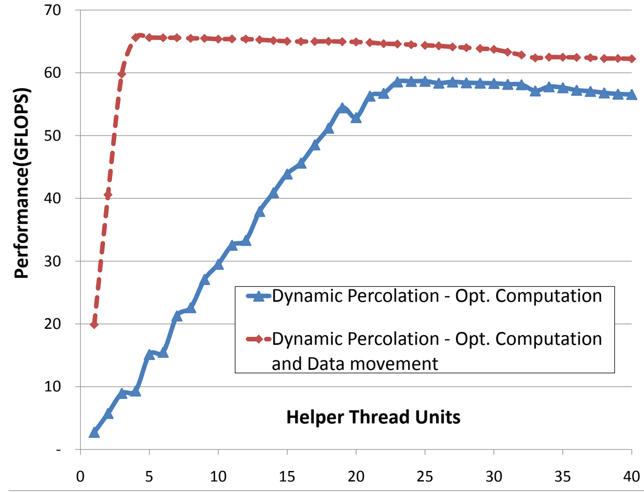
Figure 8: Performance and Scalability of MM using on-chip SRAM

Using the already optimized MM in on-chip SRAM, a fully parallel MM using 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 9a. 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 are needed and the performance was increased even more. 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 9b, 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 and when the CTs finish they will stall. In addition, if the HTs increases too much, the bandwith saturation will decrease sligthly the performance.



(a) Performance of MM in off-chip DRAM for $m = 5280$ and block size $n = 264$



(b) Impact of the maximum number of Helper Threads using 156 TUs and $m = 5280$

Figure 9: Performance and Scalability of MM using off-chip DRAM

6 Related Work

Directly related work on MM has been detailed in section 4. From the many-core architecture side, static approaches have been proposed for regular applications on linear algebra and signal processing, they have shown significant speed up but they are far from the peak performance [3, 4, 5, 6]. Also, percolation has been been studied and applied on irregular application [7, 8]. Other studies have been focused on semi-automatic mechanisms for doing percolation at tile-level [9].

For other parallel architectures (e.g. CellBE, GPGPU, Intel), extensive work has been done. They have exploited the interconnection pattern of processors, like Cannon's matrix multiply algorithm [10], or the reduced number of operations like Strassen's algorithm [11, 12]. Most of the implementations have used tuned versions of specialized libraries/subroutines provided by the vendors (e.g. GEMM/BLAS, CUBLAS, MKL) [13, 14, 15].

Other studies have been focused on models that captures performance-relevant aspects of the hierarchical nature of computer memory like the Uniform Memory Hierarchy (UMH) model or the Parallel Memory Hierarchy (PMH) model [16, 17] or the principles that underlie the high-performance implementation of the MM [18].

7 Conclusions and Future Work

In this paper we have proposed a Dynamic Percolation for MM using two types of tasks – computation tasks and 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 with the number of threads and the matrix size. Our method also load balances tasks across the machine because it handles well the unpredictable effects of resource sharing and a carefully designed lock-free queue set reduces effectively the overhead.In addition, each kind of task was also optimized for maximizing the performance, subject to the constrains of the architecture: number of registers available, crossbar bandwidth and limited shared resources.

We report experimental results of our methods on a real C64 chip achieving 70.0 and 65.6 GFLOPS for MM with operands in SRAM and DRAM 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 includes to extend the MM to multiple chips and apply these techniques for the implementation of LINPACK benchmark on C64.

References

[1] E. Garcia, I. E. Venetis, R. Khan, and G. Gao, "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.

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

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

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

[5] I. E. Venetis and G. R. Gao, "Mapping the LU Decomposition on a Many-Core Architecture: Challenges and Solutions," in *Proceedings of the 6th ACM Conference on Computing Frontiers (CF '09)*, Ischia, Italy, May 2009, pp. 71–80.

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

[7] G. Tan, D. Fan, J. Zhang, A. Russo, and G. R. Gao, "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*. New York, NY, USA: ACM, 2008, pp. 279–280.

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

[9] G. Gan, X. Wang, J. Manzano, and G. R. Gao, "Tile percolation: An openmp tile aware parallelization technique for the cyclops-64 multicore processor," in *Euro-Par*, 2009, pp. 839–850.

[10] 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)*. Vienna, Austria: ACM, 1997, pp. 44–51.

[11] D. H. Bailey, K. Lee, and H. D. Simon, "Using Strassen's Algorithm to Accelerate the Solution of Linear Systems," *Journal of Supercomputing*, vol. 4, pp. 357–371, 1991.

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

[13] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation-a performance view," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.

[14] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 1990.

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

[16] B. Alpern, L. Carter, E. Feig, and T. Selker, "The uniform memory hierarchy model of computation," *Algorithmica*, vol. 12, pp. 72–109, 1992.

[17] B. Alpern, L. Carter, and J. Ferrante, "Modeling parallel computers as memory hierarchies," in *In Proceedings Programming Models for Massively Parallel Computers*. IEEE Computer Society Press, 1993, pp. 116–123.

[18] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 1–25, 2008.