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

Optimized Dense Matrix Multiplication on a Many-Core Architecture

Elkin Garcia[§]
Ioannis E. Venetis[†]
Rishi Khan[‡]
Guang R. Gao[§]

CAPSL Technical Memo 095
February, 2010

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

Contents

List of Figures

List of Tables

Abstract

Traditional parallel programming methodologies for improving performance assume cache-based parallel systems. However, new architectures, like the IBM Cyclops-64 (C64), belong to a new set of many-core-on-a-chip systems with a software managed memory hierarchy. New programming and compiling methodologies are required to fully exploit the potential of this new class of architectures.

In this paper, we use dense matrix multiplication as a case of study to present a general methodology to map applications to these kinds of architectures. Our methodology exposes the following characteristics: (1) Balanced distribution of work among threads to fully exploit available resources. (2) Optimal register tiling and sequence of traversing tiles, calculated analytically and parametrized according to the register file size of the processor used. This results in minimal memory transfers and optimal register usage. (3) Implementation of architecture specific optimizations to further increase performance. Our experimental evaluation on a real C64 chip shows a performance of 44.12 GFLOPS, which corresponds to 55.2% of the peak performance of the chip. Additionally, measurements of power consumption prove that the C64 is very power efficient providing 530 MFLOPS/W for the problem under consideration.

1 Introduction

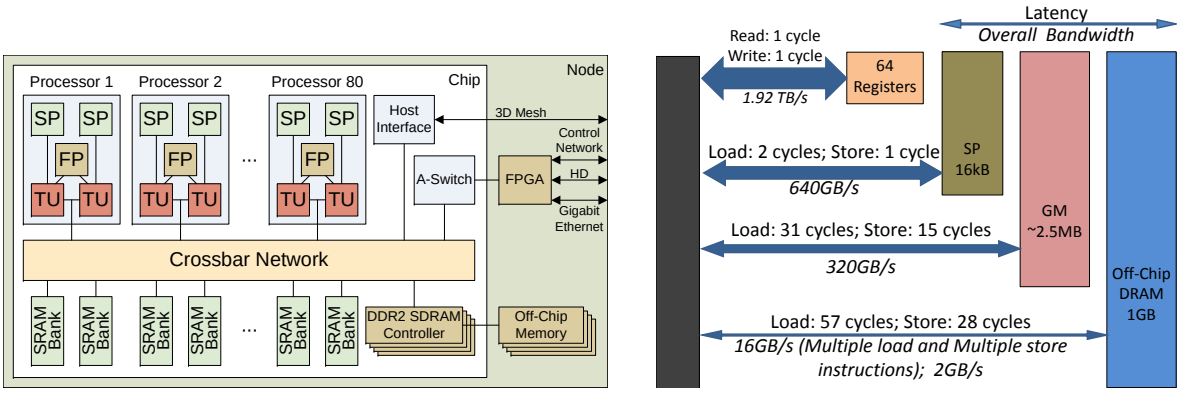
Traditional parallel programming methodologies for improving performance assume cache-based parallel systems. They exploit temporal locality making use of cache tiling techniques with tile size selection and padding [1, 2]. However, the data location and replacement in the cache is controlled by hardware making fine control of these parameters difficult. In addition, power consumption and chip die area constraints make increasing on-chip cache an untenable solution to the memory wall problem [3, 4].

As a result, new architectures like the IBM Cyclops-64 (C64) belong to a new set of many-core-on-a-chip systems with a software managed memory hierarchy. These new kinds of architectures hand the management of the memory hierarchy to the programmer and save the die area of hardware cache controllers and over-sized caches. Although this might complicate programming at their current stage, these systems provide more flexibility and opportunities to improve performance. Following this path, new alternatives for classical algorithmic problems, such as Dense Matrix Multiplication (MM), LU decomposition (LU) and Fast Fourier Transform (FFT) have been studied under these new many-core architectures [5, 6, 7]. The investigation of these new opportunities leads to two main conclusions: (1) The optimizations for improving performance on cache-based parallel system are not necessarily feasible or convenient on software managed memory hierarchy systems. (2) Memory access patterns reached by appropriate tiling substantially increase the performance of applications.

Based on these observations we can conclude that new programming and compiling methodologies are required to fully exploit the potential of these new classes of architectures. We believe that a good starting point for developing such methodologies are classical algorithms with known memory access and computation patterns. These applications provide realistic scenarios and have been studied thoroughly under cache-based parallel systems.

Following this idea, we present a general methodology that provides a mapping of applications to software managed memory hierarchies, using MM on C64 as a case of study. MM was chosen because it is simple to understand and analyze, but computationally and memory intensive. For the basic algorithm, the arithmetic complexity and the number of memory operations in multiplications of two matrices $m \times m$ are $O(m^3)$.

The methodology presented in this paper is composed of three strategies that result in a substantial increase in performance, by optimizing different aspects of the algorithm. The first one is a balanced distribution of work among threads. Providing the same amount of work to each thread guarantees minimization of the idle time of processing units waiting for others to finish. If a perfect distribution is not feasible, a mechanism to minimize the differences is proposed. The second strategy is an optimal register tiling and sequence of traversing tiles. Our register tiling and implementation of the sequence of traversing tiles are designed to maximize the reuse of data in registers and minimize the number of memory accesses to slower levels, avoiding unnecessary stalls in the processing units while waiting for data. The last strategy involves more specific characteristics of C64. The use of special instructions, optimized instruction scheduling and other techniques further boost the performance reached by the previous two strategies. The



(a) C64 Chip Architecture

(b) Memory Hierarchy of C64

Figure 1: C64 Architecture details

impact on performance can change according to the particular characteristics of the many-core processor used.

The experimental evaluation was performed using a real C64 chip. After the implementation of the three strategies proposed, the performance reached by the C64 chip is 44.12 GFLOPS, which corresponds to 55.2% of the peak performance. Additionally, measurements of power consumption prove that C64 is very power efficient, providing 530 MFLOPS/W for the problem under consideration. This value is comparable to the top of the Green500 list [8], which provides a ranking of the most energy-efficient supercomputers in the world.

The rest of this paper is organized as follows. In Section 2, we describe the C64 architecture. In Section 3, we give a short overview on the current status of MM Algorithms. In Section 4, we introduce our proposed MM Algorithm and optimizations. In Section 5, we present the experimental evaluation of our implementation. Finally, we conclude and present future work in Section 6.

2 The IBM Cyclops-64 Architecture

Cyclops-64 (C64) is an innovative architecture developed by IBM, designed to serve as a dedicated petaflop computing engine for running high performance applications. A C64 chip is an 80-processor many-core-on-a-chip design, as can be seen in Figure 1a. Each processor is equipped with two thread units (TUs), one 64-bit floating point unit (FP) and two SRAM memory banks of 30kB each. It can issue one double precision floating point "Multiply and Add" instruction per cycle, for a total performance of 80 GFLOPS per chip when running at 500MHz.

A 96-port crossbar network with a bandwidth of 4GB/s per port connects all TUs and SRAM banks. The total crossbar network bandwidth of 384GB/s supports both the intra-chip communication, as well as the six routing ports that connect each C64 chip to its neighbours [9]. The complete C64 system is built out of tens of thousands of C64 processing nodes arranged in

a 3-D mesh topology. Each processing node consists of a C64 chip, external DRAM, and a small amount of external interface logic.

A C64 chip has an explicit three-level memory hierarchy (scratchpad memory, on-chip SRAM, off-chip DRAM), 16 instruction caches of 32kB each (not shown in the figure) and no data cache. The scratchpad memory (SP) is a configured portion of each on-chip SRAM bank which can be accessed with very low latency by the TU it belongs to. The remaining sections of all on-chip SRAM banks consist the on-chip global memory (GM), which is uniformly addressable from all TUs. As a summary, Figure 1b reflects the current size, latency (when there is no contention) and bandwidth of each level of the memory hierarchy.

Execution on a C64 chip is non-preemptive and there is no hardware virtual memory manager. The former means that the C64 micro-kernel will not interrupt the execution of a user application unless an exception occurs. The latter means the three-level memory hierarchy of the C64 chip is visible to the programmer. In addition, the C64 instruction set architecture incorporates efficient support for thread level execution, hardware barriers and atomic in-memory operations.

3 Classic Matrix Multiplication Algorithms

MM algorithms have been studied extensively. These studies focused mainly on two areas: (1) Algorithms that decreases the naïve complexity of $O(m^3)$. (2) Implementations that take advantage of advanced features of computer architectures to achieve higher performance. This paper is oriented towards the second area.

In the first area, more efficient algorithms are developed. Strassen's algorithm [10] is based on the multiplication of two 2×2 matrices with 7 multiplications, instead of 8 that are required in the straightforward algorithm. The recursive application of this fact leads to a complexity of $O(m^{log7})$ [11]. Disadvantages, such as numerical instability and memory space required for sub-matrices in the recursion, have been discussed extensively [12,13]. The current best lower bound is $O(m^{2.376})$, given by the Coppersmith–Winograd algorithm [14]. However, this algorithm is not used in practice, due to its large constant term.

The second area focuses on efficient implementations. Although initially more emphasis was given towards implementations for single processors, parallel approaches quickly emerged. A common factor among most implementations is the decomposition of the computation into blocks. Blocking algorithms not only give opportunities for better use of specific architectural features (e.g., memory hierarchy) but also are a natural way of expressing parallelism. Parallel implementations have exploited the interconnection pattern of processors, like Cannon's matrix multiply algorithm [15, 16, 17], or the reduced number of operations like Strassen's algorithm [18, 19, 20]. These implementations have explored the design space along different directions, according to the targeted parallel architecture.

The many-core architecture design space has not yet been explored in detail, but existing studies already show their potential. A performance prediction model for Cannon's algorithm has

shown a huge performance potential for an architecture similar to C64 [21]. Previous research of MM on C64 showed that is possible to increase performance substantially by applying well known optimizations methods and adapting them to specific features of the chip [6]. More recent results on LU decomposition conclude that some optimizations that performs well for classical cached-based parallel system are not the best alternative for improving performance on software managed memory hierarchy systems [7].

4 Proposed Matrix Multiplication Algorithm

In this section we analyze the proposed MM algorithm and highlight our design choices. The methodology used is oriented towards exploiting the maximum benefit of features that are common across many-core architectures. Our target operation is the multiplication of dense square matrices $A \times B = C$, each of size $m \times m$ using algorithms of running time $O(m^3)$. Throughout the design process, we will use some specific features of C64 to illustrate the advantages of the proposed algorithm over different choices used in other MM algorithms.

Our methodology alleviates three related sources identified to cause poor performance in many-core architectures: (1) Inefficient or unnecessary synchronization. (2) Unbalanced work between threads. (3) Latency due to memory operations. Relation and impact in performance of these sources are architecture dependent and modeling their interactions has been an active research topic.

In our particular case of interest, the analysis of MM is easier than other algorithms not only for the simple way it can be described but also for the existence of parallel algorithms that do not required synchronizations. It simplifies the complexity of our design process because we only need to carefully analyze in two instead of the three causes of poor performance we have identified as long as the algorithm proposed does not require synchronizations. These challenges will be analyzed in the following subsections.

4.1 Work Distribution

The first challenge in our MM algorithm is to distribute work among P processors avoiding synchronization. It is well known that each element $c_{i,j} \in C$ can be calculated independently. Therefore, serial algorithms can be parallelized without requiring any synchronization for the computation of each element $c_{i,j}$, which immediately solves this requirement.

The second step is to break the $m \times m$ matrix C into blocks such that we minimize the maximum block size pursuing optimal resource utilization and trying to avoid overloading a processor. This is optimally done by breaking the problem into blocks of $\frac{m^2}{P}$ elements, but the blocks must be rectangular and fit into C.

One way to break C in P rectangular blocks is dividing rows and columns of C into q_1 and q_2 sets respectively, with $q_1 \cdot q_2 = P$. The optimal way to minimize the maximum block size is

to divide the m rows into q_1 sets of $\left\lfloor \frac{m}{q_1} \right\rfloor$ rows (with some having an extra row) and the same for columns. The maximum tile size is $\left\lceil \frac{m}{q_1} \right\rceil \cdot \left\lceil \frac{m}{q_2} \right\rceil$ and it is bounded by $\left(\frac{m}{q_1}+1\right) \cdot \left(\frac{m}{q_2}+1\right)$. The difference between this upper bound and the optimal tile size is $\frac{m}{q_1} + \frac{m}{q_2} + 1$ and this difference is minimized when $q_1 = q_2 = \sqrt{P}$. If P is not a square number, we find the q_1 that is a factor of P and closest but not larger than $\sqrt{P}$. To further optimize, we can turn off some processors if the maximum tile size could be decreased. In practice, this reduces to turning off processors if $q_2 - q_1$ is smaller and in general, this occurs if P is prime or one larger than a square number.

4.2 Minimization of High Cost Memory Operations

After addressing the synchronization and load-balancing problems for MM, the next major bottleneck is the impact of memory operations. Despite the high bandwidth of on-chip memory in many-core architectures (e.g. C64), bandwidth and size of memory are still bottlenecks for algorithms, producing stalls while processors are waiting for new data. As a result, implementations of MM, LU and FFT are still memory bound [5,6,7]. However, the flexibility of software-managed memory hierarchies provides new opportunities to the programmer for developing better techniques for tiling and data locality without the constraints imposed by cache parameters like line sizes or line associativity [22, 7]. It implies an analysis of the tile shapes, the tile size and the sequences in which tiles have to be traversed taking advantage of this new dimension in the design space.

While pursuing a better use of the memory hierarchy, our approach takes two levels of this hierarchy, one faster but smaller and the other slower but bigger. our objective is to minimize the number of slow memory operations, loads (LD) and stores (ST), that may are function of the problem (Λ), the number of processors (P), the tile parameters (L) and the sequence of traversing tiles (S), subject to the data used in the current computation (R) cannot exceed the size of the small memory ($R_{\max}$). This can be expressed as the optimization problem:

$$\min_{L,S} \quad LD\left(\Lambda, P, L, S\right) + ST\left(\Lambda, P, L, S\right), \quad s.t. \quad R\left(\Lambda, P, L, S\right) \leq R_{\max} \tag{1}$$

In our case, internal registers are the fast memory and Λ is the MM with the partitioning described in subsection 4.1. Our analysis assumes a perfect load-balanced case where each block $C' \in C$ of size $n \times n$ $\left(n = \frac{m}{\sqrt{P}}\right)$ computed by one processor is subdivided in tiles $C'_{i,j} \in C'$ of size $L_2 \times L_2$. Based on the data dependencies, the required blocks $A' \in A$ and $B' \in B$ of sizes $n \times m$ and $m \times n$ are subdivided in tiles $A'_{i,j} \in A'$ and $B'_{i,j} \in B'$ of sizes $L_2 \times L_1$ and $L_1 \times L_2$ respectively.

Each processor requires 3 nested loops for computing all the tiles of its block. Using loop interchange analysis, an exhaustive study of the 6 possible schemes to traverse tiles was conducted and two prototype sequences S_1 and S_2 were found. The algorithms that describe these sequences are in Figure 2.

```
S1:  for i = 1 to n/L2
S2:    for j = 1 to n/L2
S3:      Initialize C'_{i,j}
S4:      for k = 1 to m/L1
S5:        Load A'_{i,k}, B'_{k,j}
S6:        C'_{i,j} += A'_{i,k} · B'_{k,j}
S :      end for
S7:      Store C'_{i,j}
S :    end for
S : end for
```

(a) Algorithm using sequence S_1

```
S1:  for i = 1 to n/L2
S2:    for k = 1 to m/L1
S3:      Load A'_{i,k}
S4:      for j = 1 to n/L2
S5:        if k = 1 then Initialize C'_{i,j}
S6:        else Load C'_{i,j}
S7:        Load B'_{k,j}
S8:        C'_{i,j} += A'_{i,k} · B'_{k,j}
S9:        Store C'_{i,j}
S :      end for
S :    end for
S : end for
```

(b) Algorithm using sequence S_2

Figure 2: Implementation of sequences for traversing tiles in one block of C

Based on the data dependencies of this implementations, the general optimization problem described in (1) can be expressed for our case by Eq. (2).

$$\begin{aligned} \min_{\substack{L\in\{L_1,L_2\},\\ S\in\{S_1,S_2\}}} \quad & f\left(m,P,L,S\right) = \begin{cases} \frac{2}{L_2}m^3 + m^2 & \text{if } S = S_1 \\ \left(\frac{2}{L_2} + \frac{1}{L_1}\right)m^3 + \left(\sqrt{P}-1\right)m^2 & \text{if } S = S_2 \end{cases} \\ \text{s.t.} \quad & 2L_1L_2 + L_2^2 \leq R_{\max} \end{aligned} \tag{2}$$

Analyzing the piecewise function f, we notice that if $P \geq 4$ the objective function for $S = S_1$ is always smaller to the objective function for $S = S_2$. Since f only depends on L_2, we minimize f by maximizing L_2. Given the constraint, L_2 is maximized by minimizing L_1. Thus $L_1 = 1$, we solve the optimum L_2 in the boundary of the constraint. The solution of Eq. (2) if $P \geq 4$ is:

$$L_1 = 1,\ L_2 = \left\lfloor \sqrt{1 + R_{\max}} - 1 \right\rfloor \tag{3}$$

This result is not completely accurate, since we assumed that there are not remainders when we divide the matrices into blocks and subdivide the blocks in tiles. Despite this fact, they can be used as a good estimate.

For comparison purposes, C64 has 63 registers and we need to keep one register for the stack pointer, pointers to A, B, C matrices, m and stride parameters, then $R_{\max} = 63 - 6 = 57$ and the solution of Eq. (3) is $L_1 = 1$ and $L_2 = 6$. Table 1 summarizes the results in terms of the number of LD and ST for the tiling proposed and other 2 options that fully utilizes the registers and have been used in practical algorithms: inner product of vectors ($L_1 = 28$ and $L_2 = 1$) and square tiles ($L_1 = L_2 = 4$). As a consequence of using sequence S_1, the number of ST is equal in all tiling strategies. As expected, the tiling proposed has the minimum number of LD: 6 times less than the inner product tiling and 1.5 times less than the square tiling.

Table 1: Number of memory operation for different tiling strategies

Memory Operations	Inner Product	Square	Optimal
Loads	$2m^3$	$\frac{1}{2}m^3$	$\frac{1}{3}m^3$
Stores	m^2	m^2	m^2

4.3 Architecture Specific Optimizations

Although the general results of subsection 4.2 are of major importance, an implementation that properly exploits specific features of the architecture is also important for maximizing the performance. We will use our knowledge and experience for taking advantage of the specific features of C64 but the guidelines proposed here could be extended to similar architectures.

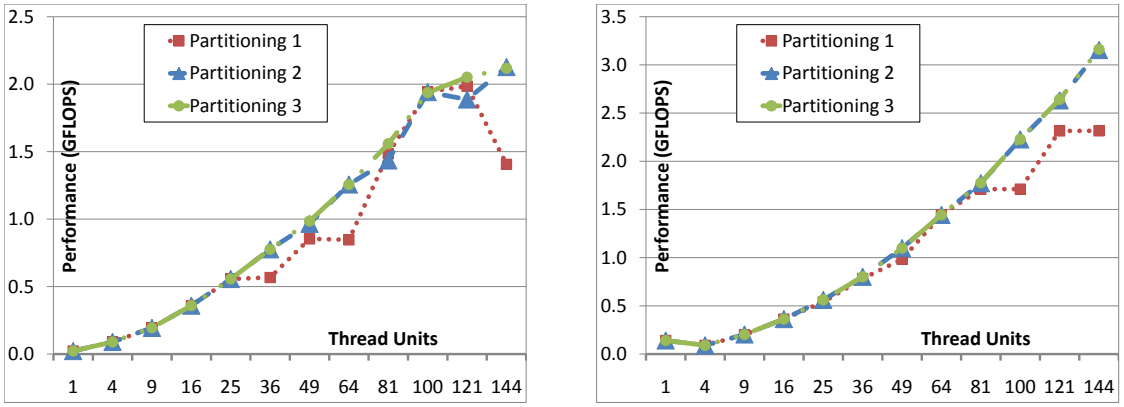
The first optimization is the use of special assembly functions for Load and Store. C64 provides the instructions multiple load (*ldm RT, RA, RB*) and multiple store (*stm RT, RA, RB*) that combine several memory operations into only one instruction. For the *ldm* instruction, starting from an address in memory contained in RA, consecutive 64-bit values in memory are loaded into consecutive registers, starting from RT through and including RB. Similarly, *stm* instruction stores 64-bit values in memory consecutively from RT through and including RB starting in the memory address contained in RA.

The advantage in the use of these instructions is that the normal load instruction issues one data transfer request per element while the special one issues one request each 64-byte boundary. Because our tiling is 6×1 in A and 1×6 in B, we need A in column-major order and B in row-major order as a requirement for exploiting this feature. If they are not in the required pattern, we transpose one matrix without affecting the complexity of the algorithms proposed because the running time of transposition is $O(m^2)$.

The second optimization applied is instruction scheduling: the correct interleaving of independent instructions to alleviate stalls. Data dependencies can stall the execution of the current instruction waiting for the result of one issued previously. We want to hide or amortize the cost of critical instructions that increase the total computation time executing other instructions that do not share variables or resources. The most common example involves interleaving memory instructions with data instructions but there are other cases: multiple integer operations can be executed while one floating point operation like multiplication is computed.

5 Experimental Evaluation

This section describes the experimental evaluation based on the analysis done in section 4 using the C64 architecture described in section 2. Our baseline parallel MM implementation works with square matrices $m \times m$ and it was written in C. The experiments were made up to $m = 488$ for placing matrices A and B in on-chip SRAM and matrix C in off-chip DRAM, the maximum number of TUs used is 144.



(a) Matrix Size 100×100

(b) Matrix Size 488×488

Figure 3: Different Partition Schemes vs. Number of Threads Units

To analyze the impact of the partitioning schema described in subsection 4.1 we compare it with other two partition schemes. Figure 3 shows the performance reached for two different matrix sizes. In *Partitioning 1*, the m rows are divided into q_1 sets, the first $q_1 - 1$ containing $\left\lfloor \frac{m}{q_1} \right\rfloor$ and the last set containing the remainder rows. The same partitioning is followed for columns. It has the worst performance of the three partitions because it does not minimize the maximum tile size. *Partitioning 2* has optimum maximum tile size of $\left\lceil \frac{m}{q_1} \right\rceil \cdot \left\lceil \frac{m}{q_2} \right\rceil$ but does not distribute the number of rows and columns uniformly between sets q_1 and q_2 respectively. Its performance is very close to our algorithm *Partitioning 3*, which has optimum maximum tile size and better distribution of rows and columns between sets q_1 and q_2 respectively. A disadvantage of *Partitioning 2* over *Partitioning 3* is that for small matrices ($n \leq 100$) and large number of TUs *Partitioning 2* may produce a significant lower performance as can be observed in Figure 3a. Our partitioning algorithm *Partitioning 3* performs always better, the maximum performance reached is 3.16 GFLOPS. The other one with optimum maximum tile size performs also well for large matrices, indicating that minimizing the maximum tile size is an appropriate target for optimizing the work load. In addition, our partition algorithm scales well with respect to the number of threads which is essential for many-core architectures.

The results of the progressive improvements made to our MM algorithm are shown in Figure 4 for the maximum size of matrices that fits on SRAM. The implementation of the tiling strategy proposed in subsection 4.2 for minimizing the number of memory operations, was made in assembly code using tiles of 6×1, 1×6 and 6×6 for blocks in A, B and C respectively. Because the size of blocks in C are not necessarily multiple of 6, all possible combinations of tiles with size less than 6×6 were implemented. The maximum performance reached was 30.42 GFLOPS, which is almost 10 times the maximum performance reached by the version that uses only the optimum partition. This big improvement shows the advantages of the correct tiling and sequence of traversing tiles that directly minimizes the time waiting for operands, substituting costly memory operations in SRAM with operations between registers. From another point of

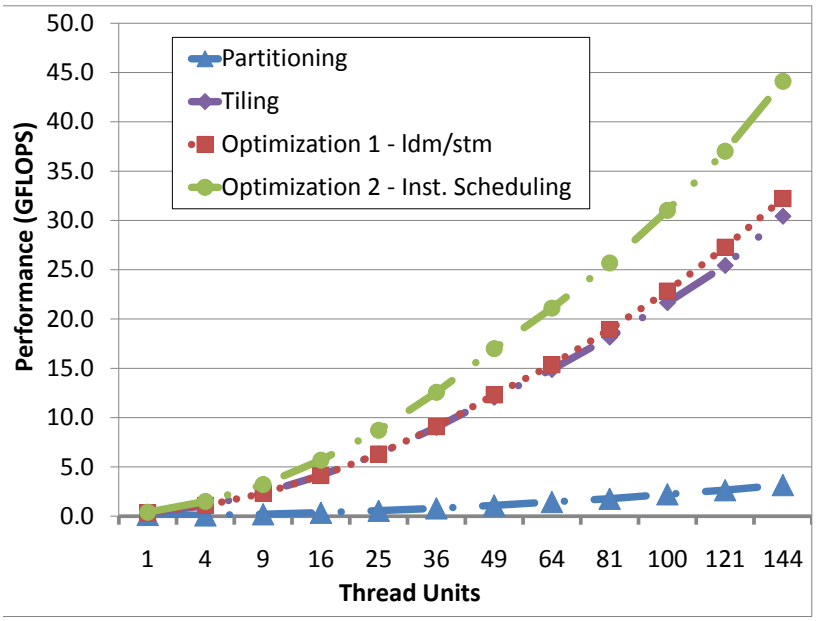


Figure 4: Impact of each optimization on the performance of MM using $m = 488$

view, our tiling increases the reuse of data in registers minimizing number of access to memory for a fixed number of computations.

The following optimizations related more with specific features of C64 also increased the performance. The use of multiple load and multiple store instructions (*ldm/stm*) diminishes the time spent transferring data addressed consecutively in memory. The new maximum performance is 32.22 GFLOPS: 6% better than the version without architecture specific optimizations. The potential of these features has not been completely exploted because transactions that cross a 64-byte boundary are divided and transactions in groups of 6 do not provide an optimum pattern for minimizing this division. Finally, the instruction scheduling applied for hiding the cost of some instructions doing other computations in the middle increases performance by 38%. The maximum performance of our MM algorithm is 44.12 GFLOPS which corresponds to 55.2% of the peak performance of a C64 chip. We also made measurements of power consumption using the current consumed by the two voltage sources of the C64 chip (1.2V and 1.8V) yielding a total of 83.22W or 530 MFLOPS/W. This demostrates the power efficiency of C64 for the problem under consideration. This value is similar to the top of the Green500 list, which provides a ranking of the most energy-efficient supercomputers in the world.

6 Conclusions and Future Work

In this paper we present a methodology to design algorithms for many-core architectures with a software managed memory hierarchy taking advantage of the flexibility these systems provide.

We apply it to design a Dense Matrix Multiplication (MM) mapping and we implement MM for C64. We propose three strategies for increasing performance and show their advantages under this kind of architecture. The first strategy is a balanced distribution of work amount threads: our partitioning strategy not only distributes the amount of computation as uniform as possible but also minimizes the maximum block size that belongs to each thread. Experimental results show that the partitioning proposed scales well with respect to the number of threads for different sizes of square matrices and performs better than other similar schemes.

The second strategy alleviates the total cost of memory accesses. We propose an optimal register tiling with an optimal sequence of traversing tiles that minimizes the number of memory operations and maximizes the reuse of data in registers. The implementation of the proposed tiling reached a maximum performance of 30.42 GFLOPS which is almost 10 times larger than the maximum performance reached by the optimum partition alone.

Finally, specific architecture optimizations were implemented. The use of multiple load and multiple store instructions (*ldm/stm*) diminishes the time spent transferring data that are consecutive stored/loaded in memory. It was combined with instruction scheduling, hiding or amortizing the cost of some memory operations and high cost floating point instructions doing other computations in the middle. After these optimizations, the maximum performance of our MM algorithm is 44.12 GFLOPS which corresponds to 55.2% of the peak performance of a C64 chip. We also provide evidence of the power efficiency of C64: power consumption measurements show a maximum efficiency of 530 MFLOPS/W for the problem under consideration. This value is comparable to the top of the Green500 list, which provides a ranking of the most energy-efficient supercomputers in the world.

Future work includes the study of other techniques like software pipelining and work-stealing that can further increase the performance of this algorithm. We also want to explore how to increase the size of the tiles beyond the maximum number of registers, using the stack and SPM. In addition, we desire to apply this methodology to other linear algebra algorithmic problems like matrix inversion.

References

[1] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation.* New York, NY, USA: ACM, 1995, pp. 279–290.

[2] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems.* New York, NY, USA: ACM, 1991, pp. 63–74.

[3] D. Callahan and A. Porterfield, "Data cache performance of supercomputer applications," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing.* Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 564–572.

[4] M. Kondo, H. Okawara, H. Nakamura, T. Boku, and S. Sakai, "Scima: a novel processor architecture for high performance computing," in *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, vol. 1, 2000, pp. 355–360 vol.1.

[5] 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.

[6] 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.

[7] 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.

[8] W.-C. Feng and T. Scogland, "The Green500 List: Year One," in *5th IEEE Workshop on High-Performance, Power-Aware Computing. In conjunction with the 23rd International Parallel & Distributed Processing Symposium)*, Rome, Italy, May 2009.

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

[10] V. Strassen, "Gaussian Elimination is not Optimal," *Numerische Mathematik*, vol. 14, no. 3, pp. 354–356, 1969.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[12] N. J. Higham, "Exploiting Fast Matrix Multiplication Within the Level 3 BLAS," *ACM Transactions on Mathematical Software*, vol. 16, no. 4, pp. 352–368, 1990.

[13] D. H. Bailey and H. R. P. Gerguson, "A Strassen-Newton Algorithm for High-Speed Parallelizable Matrix Inversion," in *Supercomputing '88: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Orlando, Florida, United States, 1988, pp. 419–424.

[14] D. Coppersmith and S. Winograd, "Matrix Multiplication via Arithmetic Progressions," in *Proceedings of the 19th Annual ACM symposium on Theory of Computing (STOC '87)*, New York, NY, USA, 1987, pp. 1–6.

[15] L. E. Cannon, "A Cellular Computer to Implement the Kalman Filter Algorithm," Ph.D. dissertation, Montana State University, Bozeman, MT, USA, 1969.

[16] C.-T. Ho, S. L. Johnsson, and A. Edelman, "Matrix Multiplication on Hypercubes Using Full Bandwidth and Constant Storage," in *Proceeding of the 6th Distributed Memory Computing Conference*. IEEE Computer Society Press, 1991, pp. 447–451.

[17] 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.

[18] 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.

[19] 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.

[20] Sascha Hunold and Thomas Rauber and Gudula Rünger, "Multilevel Hierarchical Matrix Multiplication on Clusters," in *Proceedings of the 18th Annual International Conference on Supercomputing (ICS '04)*, Malo, France, 2004, pp. 136–145.

[21] J. N. Amaral, G. R. Gao, P. Merkey, T. Sterling, Z. Ruiz, and S. Ryan, "Performance Prediction for the HTMT: A Programming Example," in *Proceedings of the Third PETAFLOP Workshop*, 1999.

[22] 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.