Optimizing the LU Factorization for Energy Efficiency on a Many-Core Architecture

Elkin Garcia, Jaime Arteaga, Robert Pavel, and Guang R. Gao

Computer Architecture and Parallel Systems Laboratory (CAPSL)
Department of Electrical and Computer Engineering
University of Delaware, Newark 19716, U.S.A.
`{egarcia@,jaime@,rspavel@,ggao@capsl.}udel.edu`

Abstract. Power consumption and energy efficiency have become a major bottleneck in the design of new systems for high performance computing. The path to exa-scale computing requires new strategies that decrease the energy consumption of modern many-core architectures without sacrificing scalability or performance. The development of these strategies demands the use of scalable models for energy consumption and the reorientation of optimization techniques to focus on energy efficiency, evaluating their trade-offs with respect to performance.
In this paper, we investigate several optimization techniques to reduce the energy consumption on many-core architectures with a software-managed memory hierarchy. We study the impact of these techniques on the Static Energy and the Dynamic Energy of the LU factorization benchmark using a scalable energy consumption model. The main contributions of this paper are: (1) The modeling and analysis of energy consumption and energy efficiency for LU factorization; (2) the study and design of instruction-level and task-level optimizations for the reduction of the Static and Dynamic Energy; (3) the design and implementation of an energy aware tiling that decreases the Dynamic Energy of power hungry instructions in the LU factorization benchmark; and (4) the experimental evaluation of the scalability and improvement in terms of energy consumption and power efficiency of the proposed optimizations using the IBM Cyclops-64 many-core architecture. We study the trade-offs between performance and power efficiency for the proposed optimizations. Our results for the LU factorization benchmark, using 156 hardware thread units, show an improvement in power efficiency between 1.68X and 4.87X for different matrix sizes. In addition, we point out examples of optimizations that scale in performance but not necessarily in power efficiency.

1 Introduction

The many-core revolution brought forward by recent advances in computer architecture has made feasible the integration of hundreds of processing elements on a single chip. With these new architectures, several challenges have arisen.

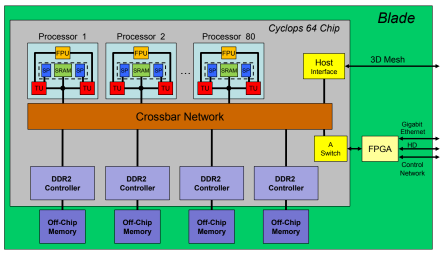
Major efforts and progress have been made in order to achieve high performance on these many-core chips. In particular, optimizations have been developed to improve the number of Floating Point Operations per Second. However, recent developments have shifted the focus to other constraints [1]. The design of the new generation of exa-scale supercomputers is restricted by power requirements [2, 3]. As a result, Energy efficiency and power consumption have become an imperative.

Energy efficiency is limited by many factors. From the point of view of semiconductor manufacturing processes, the integration of hundreds of independent processors on a single chip within a given area results in an increase in temperature and leakage current. This, in turn, results in more energy and transistors dedicated toward cooling and a deep rethinking of traditional architectures. A feasible alternative is a many-core with a software-managed memory hierarchy where the programmer controls data movement. This can free area previously used for cache controllers and over-sized caches while providing more opportunities to improve performance and energy efficiency at the cost of a higher complexity with respect to programmability.

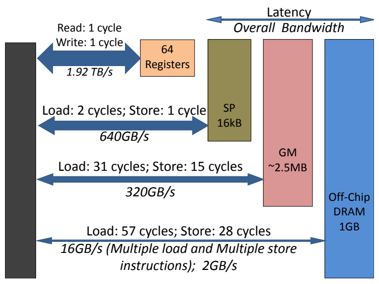
An interesting case study is the IBM Cyclops-64 many-core architecture [4] with 160 Thread Units able to run independent pieces of code and a software managed memory hierarchy. Extensive studies on performance for the Cyclops-64 have been performed in the past [5–7], energy efficiency has only recently been studied with early efforts resulting in a scalable energy consumption model for Cyclops-64 [8]. A deep understanding of this model can allow for the design of specific optimizations to decrease energy consumption.

In this paper, we study and implement several techniques to target energy efficiency on many-core architectures with software managed memory hierarchies. We study the impact of these techniques on the Static Energy and the Dynamic Energy of LU factorization using a scalable energy consumption model described by Garcia et. al. [8]. The main contributions of this paper are: First, the modeling and analysis of energy consumption and energy efficiency for LU factorization; second, the study and design of instruction-level and task-level optimizations for the reduction of Static and Dynamic energy; third, the design and implementation of an energy aware tiling for the LU factorization benchmark; and fourth, the experimental evaluation of the scalability and improvement in energy consumption and energy efficiency of the proposed optimizations using the IBM Cyclops-64 many-core. The proposed optimizations for energy efficiency increase the power efficiency of the LU factorization benchmark by 1.68X to 4.87X, depending on the problem size, with respect to a highly optimized version designed for performance.

The rest of this paper is organized as follows. In Section 2, we discuss the Cyclops-64 architecture, the energy consumption model used and the basics of a parallel LU factorization algorithm. In Section 3, we study the impact of several optimizations in the Static and Dynamic Energy. In Section 4, we present the experimental evaluation of the proposed optimizations. Section 5 examines related work. Finally, we conclude and present future work in Section 6.



(a) C64 Chip Architecture


Latency
Overall Bandwidth
Read: 1 cycle
Write: 1 cycle
1.92 TB/s
64 Registers
Load: 2 cycles; Store: 1 cycle
640GB/s
SP 16kB
Load: 31 cycles; Store: 15 cycles
320GB/s
GM ~2.5MB
Load: 57 cycles; Store: 28 cycles
16GB/s (Multiple load and Multiple store instructions); 2GB/s
Off-Chip DRAM 1GB

(b) C64 Memory Hierarchy

Fig. 1: C64 Architecture details

2 Background

2.1 A many-core Architecture: The IBM Cyclops-64

The IBM Cyclops-64 (C64) is a homogeneous many-core architecture designed by IBM for High Performance Computing. A C64 chip consists of 160 single-issue Thread Units (TUs) running at 500 MHz (see Figure 1a). A pair of TUs share a single 64-bit Floating-Point Unit (FPU). An FPU can execute a floating-point *Multiply and Add* instruction in one cycle, for a total performance of 80 GFLOPS. C64 features a three-level software-managed memory hierarchy (completely visible to the programmer) instead of a hardware and automatic data cache. This hierarchy consists of an On-Chip Scratch-Pad Memory Level (SP), an On-Chip Global SRAM Memory Level (GM), and an External DRAM Memory Level. Each TU has a 32KB memory bank, with half of that assigned, by default, as its SP. The SP can be accessed with low latency by the TU that owns it. The remaining halves of all 160 TUs banks form the GM with an approximate size of 2.5MB that is available to all the TUs. The External DRAM Memory has a size of 1GB divided into 4 memory banks and connected to the C64 chip through a crossbar network. Figure 1b presents the sizes, latencies, and bandwidth of each level of the Memory Hierarchy.

A C64 processing node needs a 1.2V regulated power supply for the C64 chip and a 1.8V regulated power supply for the external DRAM and other glue logic.

2.2 Energy Consumption model

The model proposed by Garcia et al. is a conceptually simple model that allows scalability with high accuracy for the estimation of energy consumption [8]. This is accomplished by dividing energy consumption into two components: Static Energy and Dynamic Energy. The total energy consumed by a program, Λ, with K different types of instructions, I, can be expressed as:

$$E_T(\Lambda) = E_s(t) + \sum_{j=1}^{K} E_d\left(I_j\right) \tag{1}$$

Table 1: Energy Coefficients e

Instruction	**e[pJ/Operation]**	**Instruction**	**e[pJ/Operation]**
load dram	48924.10	**store dram**	51488.99
load sram	964.65	**store sram**	548.31
double mult. and add	245.27	**double add**	178.30
double mult.	210.15	**integer mult.**	225.43
integer add	127.65	**and**	126.69
move	105.48	**load inmediate**	86.01

Static Energy, E_s, is the sum total of energy lost due to leakage currents in addition to the energy consumed by hardware units that operate continuously and consume energy even when the system as a whole is idle (e.g. the clock). E_s is proportional to the execution time t, and an architecture dependent coefficient e_0.

Dynamic Energy, E_d, is the energy consumed during the execution of an instruction, minus the leakage component. This is related to the power consumption of all active transistors, registers, and logic. E_d is a function of the number of executed instructions of each type I_j and its energy coefficient associated e_j.

This model has been successfully tested on the Cyclops-64 chip. For this particular architecture, the static coefficient is $e_0 = 63.11W$ and a representative subset of Dynamic Energy coefficients can be found in Table 1. A more detailed explanation of the model can be found in Garcia et al. publication [8].

2.3 LU Factorization

The LU factorization is a matrix factorization which represents the product of two matrices; a lower triangular matrix, L, and an upper triangular matrix, U. This algorithm is often used in linear systems in order to solve linear equations. Assuming A to be a square matrix, it can be represented as $A = L \times U$. This type of LU factorization is called *without pivoting* and is the one presented in this document. An LU factorization with pivoting performs a permutation of the rows or columns of the matrix A using one of several strategies such as Partial Pivoting, Partial Scaled Pivoting, Total Pivoting, or Total Scaled Pivoting. A comprehensive study of different pivoting strategies for LU factorization can be found in [9].

Because the LU factorization is a well studied algorithm, there are many variations such as the Linpack benchmark [10], High Performance Linpack (a parallel version of Linpack) [11], and the SPLASH-2 suite [12].

The classical approach for parallel LU factorization in cache-based systems uses fixed-size blocks that fit into cache to distribute the workload among threads. As shown in Figure 2, in the first step of the algorithm the matrix A is divided into one *Diagonal* block and several *Column*, *Row*, and *Inner* blocks. Each block is assigned to one processing element, which further divides the block into tiles in order to improve data reuse and locality. At this point, the *Diagonal* block is computed individually by one processing element, followed by a concurrent computation of the *Column* and *Row* blocks. Once all the *Column* and *Row*

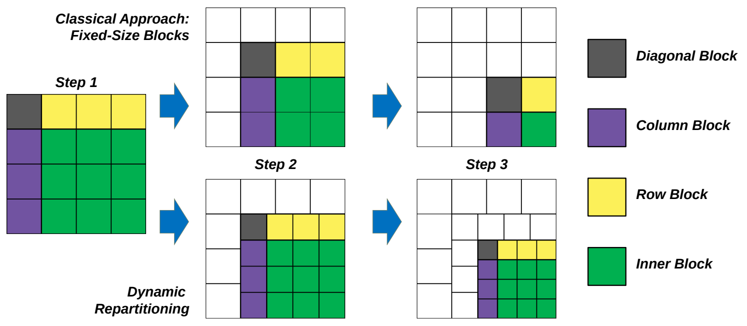


Fig. 2: Progress in each step of LU Factorization

blocks have been computed, the *Inner* blocks are processed. In the second step of the algorithm, the *Inner* blocks of the previous step are grouped again into one *Diagonal* block and several *Column*, *Row*, and *Inner* blocks, which are computed following the rules previously mentioned. This is repeated until there is only one *Inner* block, which is processed as a *Diagonal* block in the last step. The progression of steps following this classical approach is illustrated at the top of Figure 2. As can be seen, the number of blocks (i.e. the number of tasks assigned to the processing elements) decreases as the algorithm moves forward. This is translated into an increasing number of processing elements becoming idle, which lowers the performance of the application.

The *Dynamic Repartitioning* technique proposed by Venetis and Gao [13] uses varying-size blocks in each step of the algorithm in order to optimize the distribution of work among processing elements. As shown at the bottom of Figure 2, the size of the blocks is calculated at the beginning of each iteration of the LU factorization. This size is calculated as a function of the number of processing elements, so each processing element has at least one assigned task (i.e. one block to process). This optimization has been proved to increase the overall performance up to $2.8X$ in systems with a software managed memory hierarchy [13].

3 Energy Optimizations

In this section we will study the impact of several optimizations on the energy consumption of the LU factorization algorithm targeting systems with software managed memory hierarchy such as C64. The impact of these optimizations can affect the two sources of energy consumption described in Section 2.2: Static Energy E_s and Dynamic Energy E_d. Our baseline implementation is the LU factorization without pivoting by Venetis and Gao [13]. They used the *Dynamic Repartitioning* technique described in Section 2.3 and implemented a carefully designed register tiling. All their optimizations were targeting high performance.

While the increase in performance obtained by Venetis and Gao is reflected in savings of Static Energy, this high performance LU implementation has some

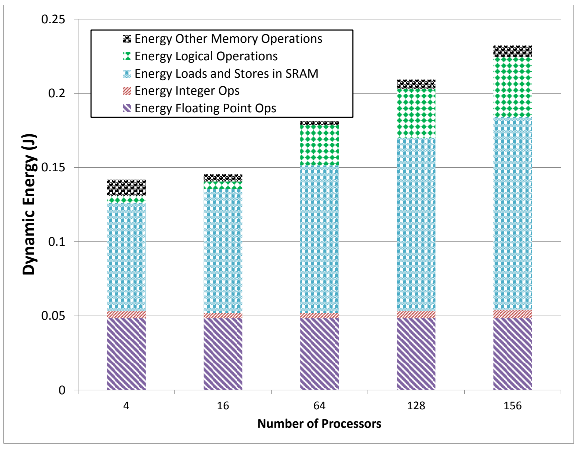


Fig. 3: Dynamic Energy Distribution for LU factorization of 840×840

drawbacks from the Energy consumption point of view: First, its register tiling focuses on increasing locality and it is not aware of the energy consumption of each instruction. Second, the static distribution of work does not consider the variance in completion time of processing similar tasks in presence of shared resources such as memory, crossbar interconnections, and FPUs. And finally, the hierarchical division into blocks and further into tiles, produces an increasing amount of smaller tiles in the borders of each block, which can hurt not just the performance but also the energy consumption.

3.1 Energy Aware Tiling design

To reduce the Dynamic Energy consumption of the LU factorization, we will focus on the instructions that contribute the most to it. Using the Energy consumption model described in Section 2.2, we characterized the Dynamic Energy of the LU Factorization implementation optimized for performance by Venetis and Gao [13] using the traces generated during the simulation of the application on a C64 architecture and a matrix of 840×840 allocated in on-chip memory.

Figure 3 shows how the Dynamic Energy of the LU factorization increases with the number of processors. As can be seen, Loads and Stores on the on-chip memory (SRAM) are the instructions with the largest contribution to the Dynamic Energy; this contribution also increases with the number of processors. On the other hand, the Energy of Floating point operations remains constant and the contribution of integer, logical, and other memory operations is not significant.

In order to minimize the Dynamic Energy E_d for a particular algorithm Λ, we propose to minimize the energy contribution of the most power hungry operations, in this case Loads LD and Stores ST with energy coefficients e_1 and e_2. The minimization is done on a set of possible tilings T with parameters S and L (e.g. shape and tile size). The optimization problem is shown in Eq.(2).

$$\begin{aligned} \min_{T(L,S)} E_d\left(\Lambda, T\right) \approx \quad & e_1\left|\mathrm{LD}\right| + e_2\left|\mathrm{ST}\right| \\ \text{subject to} \quad & R\left(\Lambda, T\right) \leq R_{\max}, \quad T \text{ is parallel} \end{aligned} \tag{2}$$

There are two constraints in the optimization problem: The registers used by the tiling ($R(\Lambda,T)$) need to fit in the available registers R_{max} and the tiling has to allow parallel execution. The former avoids unnecessary energy consumption produced by register spilling and the later prevents solutions with low performance due to increasing execution time produced by inability to exploit task parallelism.

In order to solve this problem for LU factorization, we analyze the energy consumption of each type of block (*Diagonal*, *Row*, *Column* and *Inner*) with sizes $M_0 \times M_0$, $M_0 \times M_1$, $M_2 \times M_0$ and $M_2 \times M_1$ respectively. Each block is assigned to a processor and further divided into tiles. There are 3 cases of sequences to traverse the tiles (e.g. S_0, S_1 and S_2) for each type of block. A detailed explanation of the procedure to find the optimum tiling for the *Inner* block and a summary of the results for the other type of blocks are presented in the next paragraphs.

Inner Blocks: For the computation of an *Inner* block, a *Row* block and a *Column* block are required. *Row*, *Column* and *Inner* blocks are divided into tiles of $L_0 \times L_1$, $L_2 \times L_0$ and $L_2 \times L_1$ respectively. The three possible sequences of traversing tiles reuse tiles on a different operand: The *Row* block (case S_0), the *Column* block (case S_1) and the *Inner* block (case S_2). The problem formulation for the Dynamic Energy is shown in Eq. 3.

$$\begin{aligned} \min_{\substack{L\in\{L_0,L_1,L_2\},\\ S\in\{S_0,S_1,S_2\}}} \quad & f\left(L,S\right) = \begin{cases} e_1 M_0 M_1 \left(\frac{M_2}{L_0} + \frac{M_2}{L_1} + 1\right) + \frac{e_2 M_0 M_1 M_2}{L_0} & \text{if } S = S_0 \\ e_1 M_0 M_2 \left(\frac{M_1}{L_0} + \frac{M_1}{L_2} + 1\right) + \frac{e_2 M_0 M_1 M_2}{L_0} & \text{if } S = S_1 \\ e_1 M_1 M_2 \left(\frac{M_0}{L_1} + \frac{M_0}{L_2} + 1\right) + e_2 M_1 M_2 & \text{if } S = S_2 \end{cases} \\ \text{s.t.} \quad & L_0 L_1 + L_0 L_2 + L_1 L_2 \leq R_{\max}, \quad L_0, L_1, L_2 \in \mathbb{Z}^+ \end{aligned} \tag{3}$$

The the non-linear optimization problem was solved using the Karush Kuhn Tucker conditions. We assumed all the variables being positive and M_0, M_1 and M_2 being bigger or equal than L_0, L_1 and L_2. In addition, we used the fact that M_1 and M_2 are equal to M_0 or $M_0 + 1$. We found that the best solution was to reuse the *Inner* tile (case S_2) with parameters $L_0 = 1$, $L_1 = N$ and $L_2 = N$, with $N^2 + 2N \leq R_{\max}$. In this case, an *Inner* block is computed by dividing it into tiles of $N \times N$ elements and loading each *Inner* tile into the registers, which act as accumulators for the partial results. Each partial result is calculated from a pair composed of one tile of $N \times 1$ elements of the corresponding *Column* block and one tile of $1 \times N$ elements of the corresponding *Row* block. The registers used as accumulators are stored back into memory only when there are no more

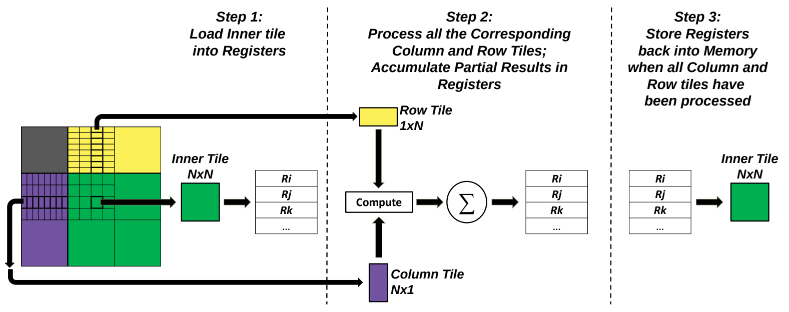

Step 1:
Load Inner tile into Registers
Step 2:
Process all the Corresponding Column and Row Tiles; Accumulate Partial Results in Registers
Step 3:
Store Registers back into Memory when all Column and Row tiles have been processed
Row Tile 1xN
Inner Tile NxN
Ri
Rj
Rk
...
Compute
Σ
Column Tile Nx1
Inner Tile NxN

Fig. 4: Optimum Energy-Aware Tiling for an Inner Block

pairs of *Column* and *Row* tiles to process. An example of this process is shown in Figure 4

Row Blocks: To compute a *Row* block, this is divided into tiles of $N \times N$ elements (with N being the same as for the *Inner* block). The process followed to compute each *Row* tile is similar to the one used for an *Inner* tile. The main difference is that the computation of a *Row* tile requires tiles of $N \times 1$ elements of the corresponding *Diagonal* block and tiles of $1 \times N$ elements that have been previously processed in the current *Row* block. Each *Row* tile to be processed is loaded into the registers, which are used as accumulators for the partial results of the computation of each pair of *Diagonal* and *Row* tiles. These registers are stored back into memory when there are no more pairs to process.

Column Blocks: To compute a *Column* block, this is also divided into tiles of $N \times N$ elements. Each *Column* tile is computed using tiles of $1 \times N$ elements of the corresponding *Diagonal* block and tiles of $N \times 1$ elements that have been previously processed in the current *Column* block. In order to minimize the Dynamic Energy of loads and stores, each *Column* tile to be processed is firstly loaded into registers. Then, these registers are used as accumulators for the partial results computed for each pair of *Diagonal* and *Column* tiles. When there are no more pairs to process, the content of the registers used as accumulators is stored back into memory.

Diagonal Block: A *Diagonal* block can be seen as another matrix A' that needs to be LU-factorized. Consequently, the *Diagonal* block can be divided into tiles of $N \times N$ elements, labeled as *Diagonal*, *Column*, *Row*, and *Inner* tiles. They can be latter processed following the same rules used in the computation of the matrix A and the same traversing of tiles previously described for the *Column*, *Row*, and *Inner* blocks.

3.2 Minimizing Static Energy using Pipelining

The design of specific tilings for energy consumption already targets Dynamic Energy. However, the long latency of memory operations with respect to the latency of arithmetic operations can produce stalls, where each processor is waiting for data required for computation. This scenario becomes worse if hundreds of threads, starvation of shared resources and bandwidth limitations are considered. This behavior can increase the Static Energy consumption due to increasing latency produced by contention.

In order to successfully minimize the impact of Static Energy, further optimizations were done to the implementation of the tilings described in Section 3.1. Each *for* loop was software-pipelined and unrolled twice, using different registers for each unrolled iteration if possible and sharing registers when necessary.

Following Figure 4, a *for* loop iteration computes a partial result for an *Inner* tile of $N \times N$ elements using a *Row* tile of $1 \times N$ elements and a *Column* tile of $N \times 1$ elements; the next iteration uses a different *Row* tile and a different *Column* tile to compute the next cumulative partial result of the same *Inner* tile. Consequently, a *for* loop that has been unrolled twice requires at least $N^2 + 4N$ registers. Since additional registers are required in the loop iterations for loop control and pointers (a pointer for the *Row* tiles and a pointer for the *Column* tiles; no pointer is necessary inside the loop for the *Inner* tile since this tile is the same for all the iterations), some registers were shared between iterations in order to decrease the requirement in the number of registers.

To diminish the impact of this register-sharing, the instructions of the loop were later properly interleaved to ensure that memory-related instructions (i.e. *loads* and *stores*) were already completed at the moment the registers involved in such operations were used in a arithmetic instruction, decreasing the execution time to directly impact the static energy.

3.3 Dynamic Task Scheduling for Energy Reduction

At this point, the fine-grain tasks have been optimized in order to decrease energy consumption while using the performance-oriented Static scheduling proposed by Venetis and Gao [13]. Even though the *Dynamic Repartition* technique is meant to perform an optimized distribution of work among processing elements, it does not take into account the undesirable delays produced by the competition of access to shared resources (e.g. competition for memory bandwidth on shared memory). This results in variations in the completion time between tasks of the same size. As a consequence, the energy consumption per task will not be uniform. This variation will be most significant with fine-grained tasks, such as the tiles described for LU factorization. In the end, a static distribution of limited work, even for cases of very regular tasks, will result in scenarios where the unbalanced distribution of work will have a negative impact on the Static Energy consumption. In addition, division of blocks into tiles produces a set of smaller border tiles per block that are suboptimal in terms of energy consumption.

In order to overcome these problems, a Dynamic Scheduling of tasks was used in the LU factorization, using the tile as a unit of work assigned to each processing element, instead of a block. First, the matrix is divided into tiles of $N \times N$ elements, which are processed following the LU factorization algorithm, that is, first the *Diagonal* tile, then all the *Column* and *Row* tiles, and finally all the *Inner* tiles. However, in this case, the assignment of tiles is not made statically (as in Venetis and Gao [13]) but in a first-come first-served basis: A tile is assigned to a processing element as soon as the processing element becomes available (i.e. as soon as the processing element finishes the computation of the previous assigned tile) and the tile dependencies are satisfied.

Dividing the matrix in tiles of $N \times N$ leads to a significant amount of tasks, which could increase the overhead of the implementation and reduce the data reuse. Nevertheless, the Dynamic Scheduling of tasks has ultimately a positive impact in the Static Energy consumption of the application since it ensures a better workload balance by keeping the number of idle processors low. This is ultimately translated in a reduction of the execution time of the application. In addition to this, the overhead associated with Dynamic Scheduling is diminished thanks to the support of in-memory atomic operations in the C64 [14]. Using an in-memory atomic operation such as L_ADD, a Dynamic Scheduler can be easily implemented with a counter for the number of tasks. Every time a processor is available, it asks for a new task and increments the counter. Since this increment is performed atomically in memory, additional round trips are avoided increasing the throughput of this counter.

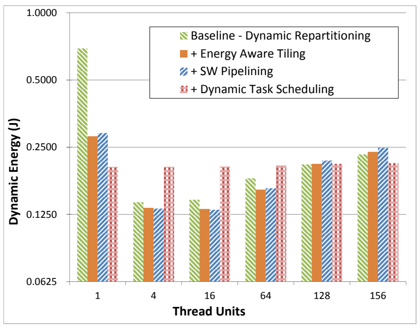
To increase the data reuse with Dynamic Scheduling and to avoid that a *Diagonal* tile of $N \times N$ becomes a bottleneck for the whole algorithm (since no tile can be processed until that tile is computed), the size of the *Diagonal* tile can be increased to $bN \times bN$ with $b \in \mathbb{N}$ and $b \geq 2$, while the sizes of other tiles remain as $N \times N$. This reduces by b the number of steps required to compute the LU factorization. The use of a tile as a unit of work for the Dynamic Scheduling, instead of a block, decreases significantly the number of suboptimal border tiles, decreasing the *Dynamic Energy* too.

4 Experimental Evaluation

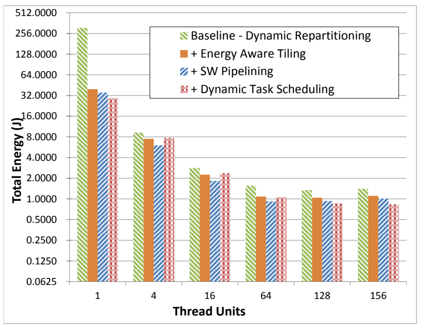
This section describes the experimental evaluation of the proposed optimizations targeting energy consumption and power efficiency described in Section 3. We have used the IBM C64 platform described in Section 2.1 and the energy estimations using the model described in Section 2.2. All benchmarks were written in C with hand-tuned assembly for the register tiling. Benchmarks were compiled with ET International's C64 C compiler with compilation flags `-O3`. We ran all of our experiments using FAST [15], a highly accurate C64 simulator.

We implemented several versions of LU factorization using on-chip shared memory. The power-aware tiling proposed in Section 3.1 uses $N = 6$ given the 64 registers per Thread Unit (TU) available in Cyclops-64. Also, for the Dynamic Task Scheduling described in Section 3.3, we used $b = 2$ so the *Diagonal* tile is 12×12. The Static Energy coefficient e_0 was computed using measurements on a real chip and the number of TUs used, having in mind that 4 additional TUs are reserved: 1 for executing the runtime system and other 3 for managing the communication with other chips using a 3D mesh.

Our first set of experiments uses a matrix of 840×840, the maximum size that fit in on-chip memory. We study the scalability of Dynamic Energy (Figure 5a) and Total Energy (Figure 5b) using different number of TUs. As expected, our



(a) Dynamic Energy vs. Thread Units for a matrix of 840 × 840



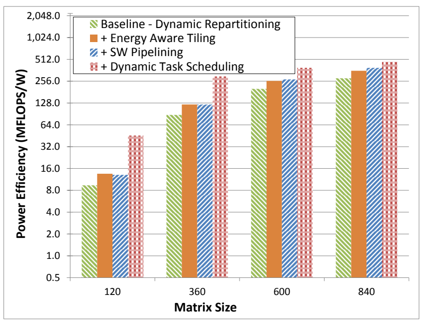
(b) Total Energy vs. Thread Units for a matrix of 840 × 840

Fig. 5: Scalability of Energy Consumption with the number of TUs

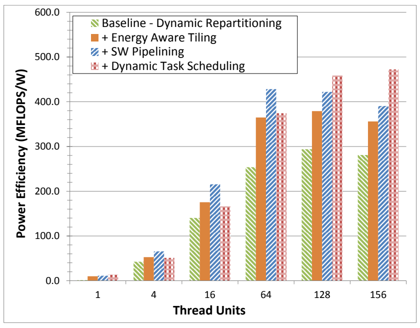
Energy Aware tiling decreases the Total Energy with respect to the baseline version that uses Dynamic Repartitioning. This is also true for the Dynamic Energy up to 128 TUs. The software pipelining do not significantly impact the Dynamic Energy because the instructions executed are practically the same but this technique decreases Total Energy because the total execution time and the Static Energy decreases. In addition, we noticed that the Dynamic Energy consumption of our Dynamic Task Scheduling does not vary with the number of TUs. The reason is that the size of the basic unit of work, the tile, is function of architectural parameters such as the number of registers but it is not function of the number of TUs like the blocks used in Dynamic Repartitioning. Our approach using Dynamic Scheduling seems useful for decreasing dynamic energy and total energy when the number of TUs surpasses 128. In addition, we noticed that total energy and dynamic energy of the baseline implementation using 1 TU are particularly high, compared with higher number of threads. The reason is that the *Diagonal* register tiling used in the *Diagonal* block calculation is highly inefficient compared with the other tilings; a serial execution computes an LU Factorization as a single *Diagonal* block and exposing this fact.

We also study the impact of the optimizations proposed in terms of Power Efficiency (the ratio between performance and power consumption) in order to examine the trade offs between performance and power consumption. Figure 6a shows the scalability of the Power Efficiency with respect to the matrix size using the maximum number of TUs available, while Figure 6b shows the scalability of the Power Efficiency with respect to the number of TUs for the biggest matrix that fits on SRAM.

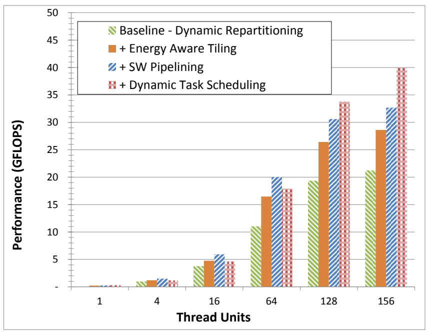
For different matrix sizes on Figure 6a, all the proposed optimizations increase the power efficiency. The increase in power efficiency for the LU factorization varies between 1.68X and 4.87X with respect to a highly optimized version that targets performance (Our baseline that uses Dynamic Repartitioning). The major returns of the techniques proposed are reached with small matrices. The



(a) Power Eff. vs. Matrix Size for $TU = 156$



(b) Power Eff. vs. TUs for Matrix Size 840×840



(c) Performance vs. TUs for Matrix Size 840×840

Fig. 6: Power Efficiency and Performance for LU factorization

optimization with the higher impact is the Dynamic Task Scheduling: between 1.2X and 3.5X to the power efficiency.

A careful comparison of the behavior between Power efficiency (Figure 6b) and Performance (Figure 6c) shows similarities when few threads are used. For the baseline implementation, as well as for the Energy-aware tiling and the Software Pipelining optimizations, the power efficiency drops after 128 TUs. This is related to the fact that even though the execution time and Static Energy decreases for an increasing number of TUs in all three implementations, the Dynamic Energy increases because these optimizations schedule tasks based on blocks. In contrast, the Power Efficiency of the Dynamic Task Scheduling optimization increases properly with the number of TUs because this type of scheduling does not only scales in terms of performance and Static Energy but also because it keeps the Dynamic Energy constant with the number of TUs.

For the C64 architecture there is a big correlation between the performance and the energy efficiency using few TUs given the high contribution of the static energy to the total energy budget. However, this scenario changes when more TUs are used. While all the techniques proposed improve the performance (as seen in Figure 6c), the power efficiency decreases after 64 TUs or 128 TUS for the Static scheduling techniques (as seen in Figure 6b). On the other hand, the Dynamic Task scheduling scales in Performance and Power Efficiency.

5 Related Work

As previously mentioned, the modeling of and optimization for energy consumption is a well researched topic. Many models focus on scheduling and are based on the overall amount of work per unit time [16] or energy [17]. These approaches yield a simplified model that is comparatively easy to use. However, the options and optimizations are limited by the coarse-grained approach.

In contrast, fine-grain approaches [18], like our own, exchange complexity for the potential optimizations that can be applied. Previous works utilized highly accurate, but highly complex, techniques to reduce energy consumption on uniprocessor architectures. These required precise information about the underlying hardware and are based on a sturdy foundation of instruction scheduling techniques [19]. This focus on the individual core worked well for uniprocessor architectures but it is unclear how well it will scale for multi-cores. Additionally, these models do not fit with the comparatively recent worldwide pursuit of energy efficiency on multiprocessors: the development and analysis of hardware features such as energy efficient off-chip memory and dynamic voltage selection [20].

6 Conclusions and Future Work

In this paper, we studied and implemented several optimizations to target energy efficiency on many-core architectures with software managed memory hierarchies using LU factorization. Our starting point was a highly optimized LU factorization designed for high performance [13]. We analyzed the impact of these optimizations on the Static Energy E_s, Dynamic Energy E_d, Total Energy E_T and Power Efficiency. To facilitate this, we used a scalable energy consumption model [8]. We designed and applied further optimizations strategies at the instruction-level and task-level to directly target the reduction of Static and Dynamic Energy and indirectly increase the Power Efficiency. We designed and implemented an energy aware tiling to decrease the Dynamic Energy. The tiling proposed minimizes the energy contribution of the most power hungry instructions. Our experimental evaluation of the scalability and improvement in energy consumption and energy efficiency of the proposed optimizations was made using the FAST simulator for the IBM Cyclops-64 many-core architecture. The proposed optimizations for energy efficiency increase the power efficiency of the LU factorization benchmark by 1.68X to 4.87X, depending on the problem size, with respect to a highly optimized version designed for performance. In addition, we point out examples of optimizations that scale in performance but not necessarily in power efficiency.

Future work includes the implementation and energy analysis of a DRAM-version of the LU factorization algorithm, the extension of the model and methodology to other algorithms (e.g. Linear Algebra and Graphs) and a study of the impact on the energy consumption and power efficiency of the task size with dynamic scheduling techniques. We are also interested in the relation between optimum tiling for increasing performance and optimum tiling for energy efficiency. Additionally, a hybrid approach combining the advantages of static and dynamic scheduling [21] will be investigated.

7 Acknowledgements

This material is based upon work supported by the Department of Energy [Office of Science] under Award Number DE-SC0008717. This work was partly supported by European FP7 project TERAFLUX, id. 249013. We also thank ET International, Inc. for its support during the course of experiments. Finally, we thank the reviewers for their valuable suggestions.

References

1. E. Garcia, D. Orozco, R. Khan, I. Venetis, K. Livingston, and G. R. Gao, "Dynamic Percolation: A case of study on the shortcomings of traditional optimization in Many-core Architectures," in *Proceedings of 2012 ACM International Conference on Computer Frontiers (CF 2012)*, (Cagliari, Italy), ACM, May 2012.
2. K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," *DARPA Information Processing Techniques Office (IPTO) sponsored study*, 2008.
3. J. Torrellas, "Architectures for extreme-scale computing," *Computer*, vol. 42, pp. 28 –35, Nov. 2009.
4. M. Denneau, "Cyclops," in *Encyclopedia of Parallel Computing: SpringerReference (www. springerreference. com)* (D. Padua, ed.), Springer-Verlag Berlin Heidelberg, 2011.
5. 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), Part II*, vol. 6272 of *Lecture Notes in Computer Science*, (Ischia, Italy), pp. 316–327, Springer, Aug. 2010.
6. L. Chen and G. R. Gao, "Performance analysis of cooley-tukey fft algorithms for a many-core architecture," in *Proceedings of the 2010 Spring Simulation Multiconference*, SpringSim '10, (San Diego, CA, USA), pp. 81:1–81:8, Society for Computer Simulation International, 2010.
7. D. Orozco, E. Garcia, and G. Gao, "Locality optimization of stencil applications using data dependency graphs," in *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, LCPC'10, (Berlin, Heidelberg), pp. 77–91, Springer-Verlag, 2011.
8. E. Garcia, D. Orozco, and G. Gao, "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)*, (Heraklion, Greece), pp. 53–66, Jan. 2011.

9. O. Y. Chen, "A comparison of pivoting strategies for the direct lu factorization," *Electronic Proceedings of the Eighth Annual International Conference on Technology in Collegiate Mathematics Houston, Texas, November 16-19, 1995*, Nov. 1995.
10. J. J. Dongarra and D. W. Walker, "Software libraries for linear algebra computations on high performance computers," *SIAM Rev.*, vol. 37, pp. 151–180, June 1995.
11. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, pp. 803–820, 2003.
12. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," *SIGARCH Comput. Archit. News*, vol. 23, pp. 24–36, May 1995.
13. 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), pp. 71–80, May 2009.
14. E. Garcia, D. Orozco, R. Pavel, and G. R. Gao, "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 and Distributed Processing Symposium (IPDPS 2012)*, (Shanghai, China), IEEE, May 2012.
15. J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "FAST: A Functionally Accurate Simulation Toolset for the Cyclops-64 Cellular Architecture," in *Workshop on Modeling, Benchmarking, and Simulation (MoBS '05). In conjunction with the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, pp. 11–20, 2005.
16. F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," in *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pp. 374 –382, Oct. 1995.
17. M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced cpu energy," *Mobile Computing*, pp. 449–471, 1996.
18. S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel, "An accurate and fine grain instruction-level energy model supporting software optimizations," in *Proc. of PATMOS*, Citeseer, 2001.
19. S. Lee, A. Ermedahl, and S. L. Min, "An accurate instruction-level energy consumption model for embedded risc processors," in *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, Compilers and Tools for Embedded Systems*, (New York, NY, USA), pp. 1–10, ACM, 2001.
20. A. Andrei, P. Eles, Z. Peng, M. Schmitz, and B. Hashimi, "Energy optimization of multiprocessor systems on chip by voltage selection," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 15, pp. 262 –275, Mar. 2007.
21. S. Donfack, L. Grigori, W. Gropp, and V. Kale, "Hybrid static/dynamic scheduling for already optimized dense matrix factorization," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 496–507, 2012.