LU Decomposition on Cell Broadband Engine: An Empirical Study to Exploit Heterogeneous Chip Multiprocessors

Feng Mao, Xipeng Shen

LU Decomposition On Cell Broadband Engine: An Empirical Study to Exploit Heterogeneous Chip Multiprocessors

Feng Mao* Xipeng Shen

Computer Science Department
The College of William and Mary
Williamsburg, VA, USA 23185

Abstract. To meet the needs of high performance computing, the Cell Broadband Engine owns many features that differ from traditional processors, such as the large number of synergistic processor elements, large register files, the ability to hide main-storage latency with concurrent computation and DMA transfers. The exploitation of those features requires the programmer to carefully tailor programs and simutaneously deal with various performance factors, including locality, load balance, communication overhead, and multi-level parallelism. These factors, unfortunately, are dependent on each other; an optimization that enhances one factor may degrade another. This paper presents our experience on optimizing LU decomposition, one of the commonly used algebra kernels in scientific computing, on Cell Broadband Engine. The optimizations exploit task-level, data-level, and communication-level parallelism. We study the effects of different task distribution strategies, prefetch, and software cache, and explore the tradeoff among different performance factors, stressing the interactions between different optimizations. This work offers some insights in the optimizations on heterogenous multi-core processors, including the selection of programming models, considerations in task distribution, and the holistic perspective required in optimizations.

Keywords: Software cache, Heterogeneous architecture, LU decomposistion, CELL Broadband Engine

1 Introduction

Multi-core and heterogeneousness have been the recent trends in computer development. A typical example is the IBM Cell Broadband Engine (Cell B/E) [11], an asymmetric and heterogeneous multi-core architecture. It typically consists of one general-purpose IBM PowerPC processor element (PPE) and eight independent synergistic processor elements (SPEs). The SPEs have large register files and good ability to hide main-memory latency with concurrent computation and direct memory access (DMA) transfers. These features make this heterogeneous

* This work is done when Feng Mao was associated with College of William and Mary.

architecture suitable for accelerating computation-intensive applications, such as gaming, multimedia, and scientific applications.

The matching between software and hardware on such an architecture is more important and also, more challenging than on traditional homogeneous systems, mainly because the architecture is more complex and more flexible in control. For instance, the SPEs have no cache but local storages, whose management has to be explicit through schemes like DMA transfers. Therefore, a suboptimal matching may easily cause factors of performance degradation than the optimal matching. On the other hand, finding the good matching requires the consideration of multiple factors at the same time, in particular, how an optimization affects data locality, load balance, communication cost, and multi-level parallelism. It is typical that those factors are inter-dependent, sometimes even causing optimization conflicts. A good understanding to those performance factors and their interactions is important for effective uses of such processors.

This work concentrates on the exploration of the different performance factors and their interactions on LU decomposition. We choose LU decomposition as the focus because it is a fundamental kernel in many scientific applications, such as linear algebra and signal processing programs. The insights obtained from this work may directly benefit those applications. In addition, the computation of LU decomposition includes many data dependences, posing interesting challenges to the exploitation of parallelism, communication, and other optimizations. Although many studies have analyzed LU decomposition on traditional homogeneous systems, we are not aware of any systematic exploration to the optimizations of the problem on the Cell B/E architecture. (Existing implementations, such as [9], show no systematic explorations to the optimization space.) This work emphasizes the interactions between the different performance factors in the optimization, distinguishing it from many other case studies on Cell programming.

More specifically, this work makes the following contributions:

- Based on Cell SDK, we develop an extensible framework for flexibly experimenting different optimization components for LU decomposition on Cell B/E. The framework allows plugins of a set of optimization components, and reports various performance metrics, including numbers of DMA operations, software cache hit rates, branch hint hit rates, numbers of clock cycles and so forth.
- We exploit different levels of parallelism supported by Cell. Our implementation exposes parallel tasks through a status matrix, leverages data-level parallelism by manual vectorization, and enables parallel communications by the use of non-blocking mailboxes and DMA.
- We explore the effects of a spectrum of locality optimization techniques and four kinds of task distribution schemes for block LU decomposition. We concentrate on prefetching and software cache management to enhance the effective bandwidth and hide memory access latency. We adopt SPE-centric computation acceleration programming model, and construct three static

task distribution models and a dynamic distribution model to explore the tradeoff between locality and load balance.
- We conduct detailed analysis on the influence of the different techniques on matching LU decomposition with Cell B/E architecture. The analysis reports the influence of each optimization on individual performance factors, such as locality, load balance, communication overhead, and parallelism. More importantly, it reveals the interactions of those factors and produces insights into the holistic consideration of optimizations for heterogeneous multicore processors. For instance, the experiments show that although task distribution affects both locality and load balance, load balance should be the only consideration when a good prefetching scheme is included in block LU decomposition.

The rest of the paper is organized as follows. In section 2, we introduce the background on Cell B/E architecture and block LU decomposition algorithm. In section 3, we describe our implementation and optimizations of the algorithm on Cell B/E. Section 4 reports and analyzes the influence of the optimizations and their interactions. Section 5 discusses related work, followed by a short summary.

2 Background

2.1 Cell B/E Architecture

The Cell B/E is a heterogeneous architecture, designed for accelerating computationally intensive applications [7]. A Cell processor is a single-chip multi-core processor, including 1 PPE and 8 SPEs operating on a shared, coherent memory. Figure 1 shows an overview of the architecture.

The PPE is the main processor. It contains a 64-bit PowerPC Architecture core with multimedia extension unit to support vector operations. Typically it runs the operating system, manages system resources, and controls the allocation and management of SPE threads. The 8 SPEs are processors designed for single instruction multiple data (SIMD) computation. Each contains a 256-KB local store controllable by software and a large (128-bit, 128-entry) register file. It relies on asynchronous DMA for data and instruction transfer to and from the main memory. It supports a special SIMD instruction set and is optimized for data-rich operations.

PPE and SPEs are connected by the element interconnect bus. Each SPE has a memory flow controller (MFC) to communicate with main memory. It is the application's responsibilty to maintain coherence between main memory and distributed local stores. User programs explicitly issue DMA command to exchange data between local store and memory. The user program code running on SPEs typically are implemented as a group of threads.

2.2 Block LU Decompostion Algorithm

LU decomposition is to transform a matrix A into a product of a lower triangular matrix L and an upper triangular matrix U, expressed as $A = L * U$. (Pivoting

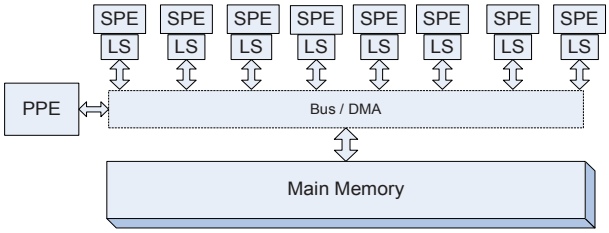


Fig. 1. Cell Broadband Engine archicture.

is not considered in this work.) Due to the importance of LU decomposition in numerical computing, many studies have explored the problem on the aspects of both algorithms and implementations. Block LU decompostion is a typical parallel algorithm to solve this problem in a divide and conquer stragtegy. Assume that we have a matrix A, expressed as a composition of 4 sub-matrices:

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix} * \begin{bmatrix} U_{00} & U_{01} \\ 0 & U_{11} \end{bmatrix} \tag{1}$$

The LU decomposition results can be derived as follows:

$$\begin{cases} L_{00}U_{00} = A_{00} \\ L_{10}U_{00} = A_{10} \\ L_{00}U_{01} = A_{01} \\ L_{10}U_{01} + L_{11}U_{11} = A_{11} \end{cases} \longrightarrow \begin{cases} L_{00}U_{00} = A_{00} \\ L_{10} = A_{10}/U_{00} \\ U_{01} = L_{00} \backslash A_{01} \\ L_{11}U_{11} = A_{11} - L_{10}U_{01} \end{cases} \tag{2}$$

L_{00} and U_{00} are respectively a lower and upper triangle matrix. Because A_{00} is usually small, L_{00} and U_{00} can be easily obtained through Gaussian elimination. The sub-matrices, L_{10} and U_{01}, can be computed subsequently. Notice that the final equation in 2 is another LU decomposition problem, with a smaller problem size. It can be reduced to an even smaller LU decomposition problem in the same manner as above. When the problem is reduced to a matrix with only 1 block, Gaussian elimination will produce the final result. This iterative strategy is the core of the block LU decomposition algorithm. We refer the reader to [4] for more details.

3 Implementation and Optimizations

The Cell B/E supports parallelism at different levels, such as task-level, data level, and communication level. In the first part of this section, we concentrate on the exploitation of various parallelisms in block LU decomposition, with the focus on task-level parallelism. We discuss the task-level dependences in block LU decomposition, and describe the use of a status matrix to help dependence analysis and expose parallel tasks. In the second part, we describe the programming model used in our implementation. Our focus in this part is on the different

strategies for distributing tasks to SPEs. The distribution strategies are important to load balance, locality, and communication cost. In the third part, we concentrate on the use of prefetch and software cache for locality improvement.

3.1 Dependence Analysis and Parallelism Exploitation

In a typical implementation of block LU decomposition, a matrix is partitioned into many small blocks. At the first iteration of the decomposition process, the whole matrix is treated as 4 regions as represented by different depths of grey (or colors) in the left bottom graph in Figure 2. The first iteration computes the final LU decomposition results corresponding to regions A_{00}, A_{01}, and A_{10} using the top 3 equations in Equation 2. It also computes the right hand side of the final equation in Equation 2 as the intermediate results corresponding to the region A_{11}. The next iteration conducts the same computation but only on the updated region A_{11}. As more iterations are executed, the working set becomes smaller and smaller until the final result of the last block is attained. The bottom row of graphs in Figure 2 illustrate the whole iterative process.

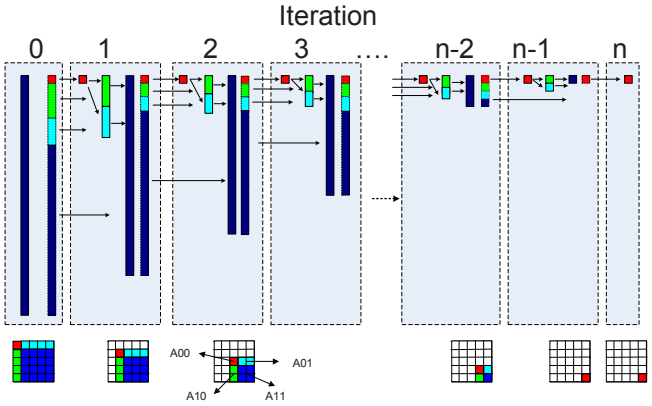


Fig. 2. Iterative computation in block LU algorithm. The top graph shows the dependences in the computation, represented by arrows. In that graph, the matrix is represented by a single column. The bottom graph shows the part of the matrix that is manipulated in each iteration.

The top row of graphs in Figure 2 shows the dependences in the computation. Consider the computation in iteration 0. The final results corresponding to region A_{00} depend on the current values of region A_{00}. And the computed results, along with the current values of a block in region A_{01}, determine the computation corresponding to block A_{01}; the blocks in the region A_{10} have the similar dependences. The computation corresponding to a block (i, j) in region A_{11} depends on the current values of block (i, j) and the results corresponding to blocks $(0, j)$ and $(i, 0)$.

We use a status matrix to help runtime dependence checking. Each block in the data matrix has one corresponding element in the status matrix, indicating the number of the iteration in which the computation corresponding to the block has just finished. During runtime, the PPE will use the status matrix to determine the tasks that are ready to run. For example, in the second iteration (i.e. iteration 1), if the PPE finds that the status corresponding to block (1, 3) and (2, 1) are both 1 (the row and column numbers are 0-based), it will immediately know that the computation corresponding to block (2, 3) is ready to run. This scheme exposes all task-level parallelism.

Besides task-level parallelism, we also exploit other levels of parallelism supported by Cell B/E. Cell B/E supports data-level parallelism mainly through vectorization. In our implementation, we manually vectorized the program to maximize the data-level parallelism. We unroll the loops and insert branch hints to increase instruction-level parallelism (details in Section 3.4.) We exploit communication parallelism by using mailbox as the main communication scheme between SPEs and the PPE. Communication through mailbox is non-blocking. When a task is assigned to a SPE, it is given a unique DMA tag, and then all the DMA_Get requests that the task needs are enqueued into the MFC. While the MFC is executing the DMA operations, the SPE is free to do computation on other requests. The status of the pending DMA operations under each tag is polled regularly. When the MFC indicates that some DMA tag has no more pending DMA operations, the corresponding task is marked "processable"; it will be processed when the processor becomes free. The similar non-blocking scheme is used for the store of computation results to the main memory.

3.2 Programming Model and Task Distribution

As a heterogeneous architecture, Cell permits two kinds of programming models: the PPE-centric, and the SPE-centric. In the PPE-centric model, the PPE runs the main application, and off-loads individual tasks to the SPEs. The PPE waits for, and coordinates, the results returned by the SPEs. This model has some variants like multistage pipeline model, parallel stage model, and services model [8]. In the SPE-centric model, most of the application code is distributed among the SPEs. The PPE is a centralized resource manager for the SPEs. Each SPE fetches its next work item from main storage when it completes its current work.

In this work, we choose SPE-centric model because it fits the property of the problem and may expose the maximum amount of task-level parallelism. Figure 3 depicts our programming model.

The instructions for the computation of a matrix block reside in the local store of every SPE. The SPEs conduct all the decomposition computation, and the PPE's job is to maintain the status matrix, find ready tasks and distribute them to the SPEs.

There are two queues associated with each SPE: the ready queue and the done queue. The PPE puts the tasks that are ready to run into the ready queues for SPE to dequeue and run. The SPEs put completed tasks and the results

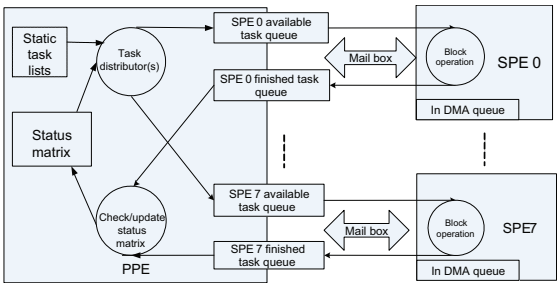


Fig. 3. Programming model for block LU decomposition on Cell B/E. The static task lists are used only in static task distributions.

into the done queues for PPE to dequeue and commit the changes to the status matrix and the output matrix. The communication between PPE and SPEs is through mailboxes, a scheme provided by Cell for exchanging 32-bit messages. Both parties use non-blocking operations (mailbox stating) to avoid unnecessary data stalls.

Task Distribution The scheme of task distribution determines which SPE will get the next ready task. It critically affects the load balance, locality, communication cost, and task-level parallelism. In this work, we implement 4 different distribution schemes.

The first is a balance-driven dynamic distribution scheme. When a task becomes ready, the PPE puts it into the ready queue that contains the fewest tasks. Although this scheme may produce good load balance among SPEs, it is locality oblivious, considering no data reuses across tasks.

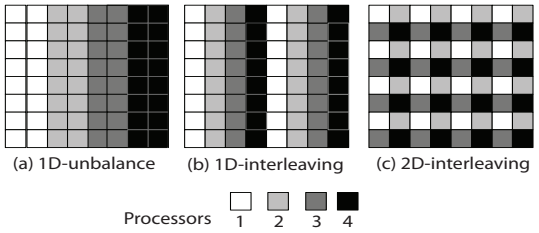


Fig. 4. Illustration of 3 static task distribution schemes on 4 processors.

The other 3 distribution schemes are static, with different tradeoff between locality and load balance. Each of the 3 schemes determines a static mapping from tasks to SPEs. All of mappings are embodied by a partition of the matrix blocks. The first static scheme evenly partitions the matrix into 8 sub-matrices as illustrated in Figure 4 (a) (the figure uses 4 processors for illustration); the second is a 1-dimension interleaving partition as shown in Figure 4 (b); the third

is a 2-dimension interleaving partition shown in Figure 4 (c). Each SPE executes only the tasks corresponding to those blocks that are mapped to it.

The first static partition has the worst balance: The SPE 0 has much less job to do than the other SPEs, because the blocks in the right and bottom regions have more computations than the blocks in the left and top regions, due to the iterative computation as shown in Figure 2. The second static scheme has much better balance, and the third one has the best. On the other hand, the first and the second scheme has better locality than the third one because a SPE in the two schemes is in charge of some whole columns and thus has more data reuse than the third scheme has. Better locality also suggests the need for fewer communications. Section 4 reports the quantitative measurement of the effects of these distribution schemes.

The dynamic distribution uses only the runtime status matrix to discover and distribute ready tasks. Whereas, the static distribution schemes use both the runtime status matrix and the static task mapping for task distribution. In our implementation of the static schemes, the PPE creates 8 threads, each of which dedicates itself to the task distribution for one SPE.

3.3 Locality Optimizations

For locality optimizations, we concentrate on the use of prefetch and software cache. Prefetch hides the latency in data transfer, and software cache reduces the required data fetches.

Prefetch Prefetch is an effective mechanism to hide the latency in data transfer if the data accesses are predictable. In our programming models, no matter with static or dynamic task distributions, a SPE can easily predict what data it is about to use by checking the tasks in its ready queue.

The prefetch in our implementation works in this way. When a SPE is about to process a task, it checks its ready queue and issues prefetch instructions for the data that the newly entered tasks may need (and not in the local store if software cache is used.) As prefetch is non-blocking, the SPE then can immediately start processing the next task in the ready queue. The capacity of the mailbox in Cell allows at most 4 tasks to be handled concurrently by the processors. So, we set the length of a ready queue to be 8.

The implementation uses double-buffering to overlap computation and communication. As shown in Figure 5, the pointer of the target location for prefetch moves continuously in the local store to avoid the conflicts between the data that are used or to be used soon in the computation and the data that are newly prefetched.

Software Cache As mentioned earlier, the SPEs in Cell have no cache but only local stores. Previous work has used the local store as a software cache, such as the general-purpose software cache in the single source compiler developed by

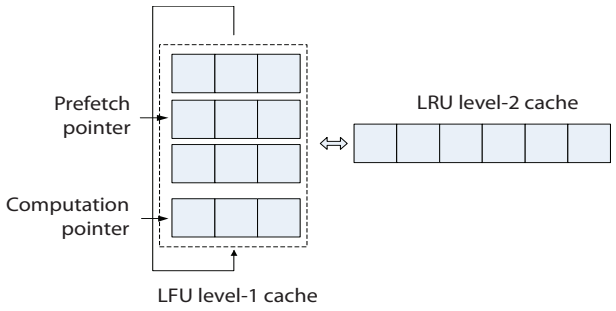


Fig. 5. Illustration of double-buffering for prefetch and 2-level software cache.

IBM [6]. In this work, we implement a software cache specific for LU decomposition. By tailoring itself to the data structure of the application, it is potentially more efficient.

In our implementation of the software cache, we use an index array to record which blocks are in the local store. Each element in the index array is a triple: $(row, column, iteration)$, where, $(row, column)$ identifies the block in the input matrix, *iteration* is the iteration in which the last update to this block occurs. At a request for data, the index is first searched; only when not found, a DMA is issued to fetch the data from the main memory. When a block is fetched into the local store, the index array is updated immediately. We use a set of counters to record the number of times each block has been accessed since the latest time it is brought into the local store. The counters are used to implement the least frequently used (LFU) replacement policy, in which, when the cache is saturated, the new block replaces the block that has the smallest counter value among all the blocks currently in the cache.

To explore the benefits from different replacement policies, we also implemented a second-level cache with least recently used (LRU) replacement policy. When the second-level cache is used, the blocks evicted from the LFU cache are put into it before being completely evicted from the local store.

The cache scheme is used for load operations only. For store operations, the data are directly written to the main memory via DMA to update the status matrix as soon as possible.

3.4 Other Optimizions

The SPEs in Cell B/E contain no hardware branch predictors, but support the use of branch hints. In our implementation, we insert 104 branch hints (51 *often* and 53 *seldom*) according to our understanding of the program behavior. These hints help the program speculate on branches and keep the instruction pipeline properly filled. In addition, we manually unroll the loops to gain more instruction-level parallelism and reduce loop control overhead. As these are standard optimizations, the details are skipped in this paper.

4 Evaluation

This section reports the effects of different task distributions and the locality optimizations. To help understand the effects of the various optimizations, we use the IBM Full System Simulator to collect detailed runtime information. The simulator supports both functional simulation and cycle-accurate simulation of full systems, including the PPE, SPEs, MFCs, PPE caches, bus, and memory controller. It can simulate and capture many levels of operational details on instruction execution, cache and memory subsystem, interrupt subsystem, communications, and other important system functions [8]. We configure the simulator to simulate a Cell B/E with 8 SPEs, each of which has 256K load storage, and 2 PPEs. It runs on a Linux operating system.

All the matrices used in the experiments are randomly generated; each element is a double precision floating-point number.

4.1 Single SPE Performance

This section measures the benefits from program code optimizations, which include vectorization, loop unrolling, and branch hints. We use a standalone mode to measure the computation to a 4-block matrix by a single SPE. All communication overhead is ignored. There are three versions of the program in our comparison: a scalar version with pure scalar operations, a simple vector version, and an optimized vector with loop unrolling and branch hints.

Table 1. SPE performance in standalone mode

	Clock cycles					Instructions Issued				
Block size	2x2	4x4	8x8	16x16	32x32	2x2	4x4	8x8	16x16	32x32
Scalar	20710	88596	149126	640344	3848292	9066	43332	63962	263830	1628488
Vector	23257	48131	56255	181201	1003435	11286	24080	25554	75290	412910
Opt Vector	8597	16254	15936	39248	194461	3788	7918	7296	17164	83600

Table 1 shows the total numbers of clock cycles and instructions by the three versions. When the block size is small, 2×2, the performance of the vector code without branch hints and unrolling is even worse than that of scalar code. It is because the block is too small to benefit from the vector registers. When the block size increases, the vector code finds enough elements to vectorize. However, the loop control statements and branches limit the speedup of the vector code: The SPE has to execute them in scalar operations. The loop unrolling reduces loop controls and increases instruction-level parallelism. The branch hints help the vector code to decrease branch miss predictions and remove unnecessary control instructions. Together, they bring speedup of a factor of 2.7 to 5.2 compared to the simple vector version. The optimized vector version outperforms the scalar version by a factor of 2.4 to 19.8 as showed in Figure 6. The optimizations cause

the code size to increase slightly, from 4K bytes in the scalar version to 4.8K in the vector version to 5.1K in the optimized vector version. The increase has negligible effects to the program performance.

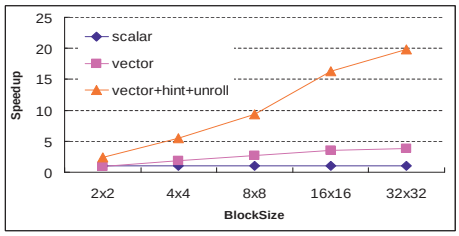


Fig. 6. SPU SIMD speedup on standalone mode

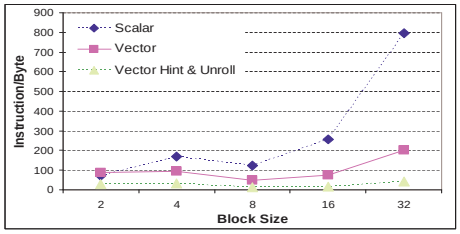


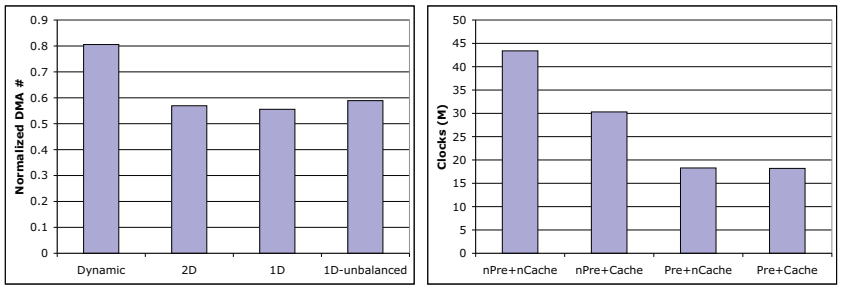
Fig. 7. The number of instructions required for processing one byte of a matrix element.

Figure 7 shows the efficiency of instructions in the three versions. The vectorized code uses far fewer instructions to process one matrix element because of the SIMD scheme provided by Cell. The unrolling and branch hints enhance the efficiency further by removing many scalar operations and reducing the penalty of branch miss prediction.

The following sections report the benefits from locality optimizations and different task distributions. All the experiments use 32x32 as the block size, with all code optimizations enabled.

4.2 Locality Optimizations

Software cache is effective in reducing the number of DMAs as shown in Figure 8 (a). The graph shows the required number of DMAs, normalized by the number when software cache is not used. As the 1D and 2D static task distribution have the best locality, the software cache reduces 45% DMA accesses. In contrast, the dynamic distribution has the worst locality, only showing 20% DMA reduction.



(a) (b)

Fig. 8. Reduced DMAs due to software cache (a), and the impact to overall performance from prefetch and software cache (b). ("nPre": no prefetch, "nCache": no software cache.)

Figure 8 (b) shows how the reduction of DMA helps the overall performance. The matrix size is 256x256 and we use 2D task distribution. If prefetch is disabled, the software cache improves the performance by 43%. However, when prefetch is enabled, the benefits from software cache become unnoticeable. The prefetch itself is enough to hide the latency in data transfer, improving the overall performance by 137%. This result suggests one of the interactions between different optimization techniques: An effective optimization technique, such as the software cache, becomes unnecessary when some other optimizations like prefetch is used. So by default, we enable prefetch and disable software cache in the following experiments.

4.3 Task Distribution

Different task distribution strategies cause different data locality and load balance. Good locality helps to reduce the required data transfers (i.e., the number of DMA operations.) In last section, Figure 8 (a) already shows the different effectiveness of software cache on different task distributions. Figure 9 further shows the average numbers of DMA operations per task when different task distributions are used when the block size is 32x32. When single-level cache is used only, the dynamic and the 2D interleaving distributions require much larger numbers of DMA operations than the 1D distributions. When two level cache is used, the 2D interleaving distribution requires no more DMA operations than the 1D distributions, whereas, the dynamic distribution still requires significantly more DMA operations. This result is intuitive as the dynamic distribution is locality-oblivious, having the worst locality among all the distributions.

On the other hand, the dynamic distribution and the 2D interleaving distribution have the best load balance. The poor load balance in the two 1D distributions causes up to orders of magnitude performance degradation compared to the performance by the other two distributions, even though they have better locality.

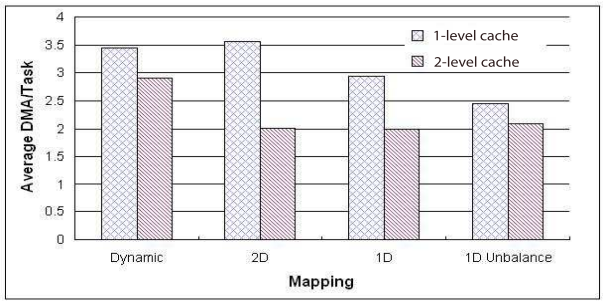
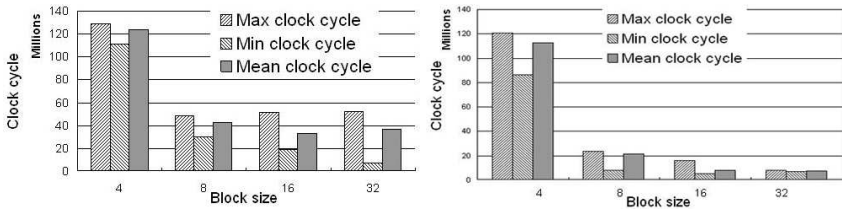


Fig. 9. Average numbers of DMA operations per task.

Besides load balance, the dynamic distribution has another advantage: It allows the full freedom for SPEs to explore task-level parallelism. While in the static distributions, a ready task may have to wait for a particular SPE to get free, even though some other SPEs may be idle. This advantage comes at the sacrifice of data locality. However, as shown in Section 3.3, prefetch is enough to hide data transfer latencies for LU decomposition. Therefore, the two advantages of the dynamic distribution make it a more appealing choice than the static distributions.

The comparison between the two graphs in Figure 10 verfies the above projection. The figures show the performance of solving a 128×128 matrix for various block sizes by using the 2D-interleaving distribution and the dynamic distribution. The three bars for a block size show the range and the mean of the performance of the 8 SPEs. The dynamic distribution shows smaller difference between the minimum and maximum clocks when the block size is 16 and 32 than the static distribution. It indicates better balance. The relatively larger difference when the block size is 8 is likely due to the randomness in dynamic distribution. Overall, the dynamic distribution outperforms the static distribution by a factor of 1 to 7 in terms of the maximum clocks.



(a) 2D interleaving distribution (b) dynamic distribution

Fig. 10. Overall running time with different task distributions.

5 Related Work

As an important linear algebra kernel, LU decomposition has been studied extensively on traditional parallel computers. In 1993, Buoni et al. have studied the different algorithms for static LU decomposition on traditional shared memory processors [4]. A scalable implementation is included in some widely used linear algebra packages, such as LAPACK [1] and ScaLAPACK [3]. However, the implementations assume the computing systems to be either a homogeneous parallel computer or a homogeneous network of workstations. Beaumont et al. studied the matrix partition schemes for LU decomposition on heterogeneous networks of workstations [2], rather than Cell B/E. We note that this current work is not the first work that implements LU decomposition on Cell B/E. But the existing implementations, such as the one in the Cell SDK library [9], although working well on a single SPE, have not systematically explored the design space and optimization issues for multiple SPEs. This work is not aimed to produce a universally applicable, fastest LU decomposition, but to use LU decomposition as an example problem to reveal the interactions among different optimizations on Cell B/E and obtain the insights in holistic optimizations for heterogeneous multicore architecture.

Locality optimization has been a focus in many previous studies, especially on traditional CPU and modern multicores [10,13]. For LU decomposition, an example is the automatic blocking for improving its locality on SMP [12]. On Cell B/E, the IBM project of Single Source Compiler (SSC Research Compiler) [5, 6] has included interesting explorations to locality optimizations. These explorations are particularly for automatic conversion of general OpenMP programs to Cell programs; the performance of the generated programs is often not as good as that of the programs directly developed from Cell SDK.

6 Conclusions

In this paper, we present an exploration to tailor block LU decomposition to Cell Broadband Engine processors. The implementation exploits different levels of parallelism supported by Cell. It exposes parallel tasks through a status matrix, leverages instruction-level parallelism by manual vectorization, and enables parallel communications by the use of non-blocking mailboxes. We study the effects of different task distribution strategies and a set of locality optimizations. The exploration reveals the interactions between those optimizations, and offers some insights into the optimization on heterogenous multi-core processors, such as the selection of programming models, considerations in task distribution, and holistic perspective required in optimizations.

Acknowledgments We thank Dimitrios Nikolopoulos, Zhengyu Wu, and Stephen McCamant for their help. This material is based upon work supported by the National Science Foundation under Grant No. 0720499 and 0811791. Any opinions, findings, and conclusions or recommendations expressed in this material

are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *IEEE Supercomputing*, pages 2–11, 1990.
2. Oliver Beaumont, Arnaud Legrand, Fabrice Rastello, and Yves Robert. Static LU decomposition on heterogeneous platforms. *The International Journal of High Performance Computing Applications*, 15(3):310–323, Fall 2001.
3. L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.
4. John J. Buoni, Paul A. Farrell, and Arden Ruttan. Algorithms for lu decomposition on a shared memory multiprocessor. *Parallel Comput.*, 19(8):925–937, 1993.
5. Tong Chen, Tao Zhang, Zehra Sura, and Mar Gonzales Tallada. Prefetching irregular references for software cache on cell. In *CGO*, pages 155–164, 2008.
6. A. E. Eichenberger et al. Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture. *IBM Syst. J.*, 45(1):59–84, 2006.
7. D. Pham et al. The design and implementation of a first-generation cell processor. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, 2005.
8. IBM. Cell be programming tutorial. http://www-01.ibm.com/chips/techlib/techlib.nsf/products /Cell_Broadband_Engine.
9. IBM. Cell broadband engine sdk libraries v3.0, 2008. http://www.ibm.com/developerworks/power/cell.
10. Y. Jiang, E. Zhang, K. Tian, and X. Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Proceedings of the International Conference on Compiler Construction*, 2010.
11. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.
12. Qing Yi, Ken Kennedy, Haihang You, Keith Seymour, and Jack Dongarra. Automatic blocking of qr and lu factorizations for locality. In *MSP '04: Proceedings of the 2004 workshop on Memory system performance*, pages 12–22, New York, NY, USA, 2004. ACM.
13. E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 203–212, 2010.