Mapping the LU Decomposition on a Many-Core Architecture: Challenges and Solutions

Ioannis E. Venetis
Dept. of Computer Engineering & Informatics
University of Patras, Rion 26500, Greece
venetis@ceid.upatras.gr

Guang R. Gao
Computer Architecture & Parallel Systems Lab
Dept. of Electrical & Computer Engineering
University of Delaware, Newark 19716, U.S.A.
ggao@capsl.udel.edu

ABSTRACT

Recently, multi-core architectures with alternative memory subsystem designs have emerged. Instead of using hardware-managed cache hierarchies, they employ software-managed embedded memory. An open question is what programming and compiling methods are effective to exploit the performance potential of this new class of architectures. Using the LU decomposition as a case study, we propose three techniques that combined achieve a 27 times speedup on the IBM Cyclops-64 many-core architecture, compared to the parallel LU implementation from the SPLASH-2 benchmarks suite. Our first method allows adaptive load distribution, which maximizes load-balance among cores – this is important to leverage the potential of the next two methods. Secondly, we developed a method for register tiling that determines the optimal data tile parameters and maximizes data reuse according to register file size constraints. We demonstrate that our method is inherently general and that it should have a much broader applicability beyond Cyclops-64. Thirdly, we present a register allocation method for register tiled loop bodies. We evaluate its effect through hand-tuned Cyclops-64 assembly code and observe a 6-fold reduction in load/store operations. We achieve a performance of 19.17 and 27.50 GFlops with double-precision floating point numbers, for a 700×700 and a 1000×1000 matrix respectively.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; G.1.0 [**Numerical Analysis**]: General—*Parallel algorithms*

General Terms

Performance

Keywords

Multi-core, local memory, LU decomposition, register tiling, load balancing

1. INTRODUCTION

The design of contemporary multi-core architectures has progressively diversified from more conventional architectures. Instead of simply "gluing" together a number of slightly modified existing uniprocessor cores, a new class of multi-core architectures is emerging, which is the result of a more extended exploration of the multiprocessor architecture design space. An important feature of these new architectures is the integration of a large number of simple cores with software-managed embedded memory (or local-storage), in place of a hardware-managed cache hierarchy. Examples of such architectures are the IBM/Sony Cell Broadband Engine [5], the 188-core Cisco Metro chip [11], the 80-core Intel Terascale chip [21], the 96-core Clearspeed CSX600 chip [6] and the 160-core IBM Cyclops-64 chip [8, 24].

Although it is not certain what the final impact of these new designs will be, they are considered to be interesting alternatives. A recent study at the University of California, Berkeley [2] suggests that it will soon be possible to fit more than 1000 cores on a single chip. Implementing a cache-coherency protocol for such a number of cores will require a significant percentage of the chips die area. This might limit the number of other functional units, which might leverage the performance potential of these designs more effectively, e.g., more floating-point units or more on-chip memory. Moreover, the efficiency of current cache-coherence protocols is questionable for that many cores.

On the other hand, offering these new architectures as general-purpose computation platforms creates a number of new problems, the most obvious one being programmability. Cache-based architectures have been studied thoroughly for years and, despite their differences, share similar characteristics at almost every level. For example, they feature a two- or three-level hardware-managed cache hierarchy and each node of the system has its own memory, creating either a distributed or distributed-shared memory address space. This lead to the development of well known programming methodologies for these systems, allowing a programmer to easily optimize code for them. On the other hand, multi-core architectures are relatively new and such general directions for application development do not exist yet.

Several segments in the computer industry are traditionally using explicit local-storage, for example, embedded systems. However, applying programming methodologies from this area to multi-core systems is not straightforward. The main concern in an embedded system is the amount of memory that the program and data will use, as the size of memory largely determines the cost of these systems. As a result, performance is usually a secondary concern. Multi-core architectures, on the other hand, are developed for problems where competitive execution performance is very important, in order to produce results fast enough or to achieve the desired accuracy. In a sense, the problem is the opposite one, compared to embedded systems: Having our data in local-storage, how do we achieve the best performance that our system can provide?

An important factor, with respect to programmability, is that the dominance of cache-based systems created a "cache aware" programming consensus, i.e., algorithms and applications even implicitly assume the existence of a cache. A typical example are linear algebra algorithms, where data distribution is largely determined by this assumption. Furthermore, such algorithms might also be transformed to exploit the fact that a cache miss will move a whole cache-line from main memory. The BLAS routines [9, 10], which are building blocks for high-performance linear algebra applications, are built on these assumptions.

Another issue that must be addressed is the diversity of multi-core architectures with local-storage. Local-storage might be visible only to the core that owns it, as in Cell, or it might be globally visible, as in Cyclops-64. One more important aspect for applications is when to move data between levels of memory [12]. In this case, the method provided by the hardware must be taken into account. For example, Cell provides asynchronous transfers through DMA, whereas Cyclops-64 only provides synchronous transfers. In the latter case, however, some of the cores can be used to run threads that only transfer data, effectively simulating asynchronous transfers.

Nevertheless, the question of how to program multi-core architectures with local-storage remains. We believe that one way to answer it is to implement, evaluate and optimize as many applications as possible for these systems. Only then will we be able to identify inefficiencies and provide solutions for them. Finally, it will be possible to have a high-level overview of problems and associated solutions, providing programmers with a tool-box from where they can draw ready-to-use solutions, as it is the case with cache-based architectures today. We believe that a good starting point are linear algebra problems. Most of them have regular access patterns, which are easy to analyze. Furthermore, they have been studied thoroughly on cache-based architectures, which makes it straight-forward to compare them with algorithms that are developed for local-storage based architectures.

In this paper we analyze our implementation of the LU decomposition for the Cyclops-64 architecture, as a case study in our effort to provide more general programming methodologies for multi-core architectures with local-storage. Our main contributions are:

- We propose a method that adjusts the load and data distribution during each successive elimination step of the LU computation. This is essential to keep all cores usefully busy, thus maximizing the register tiling performance potential of the following two techniques.

- We developed a method for register tiling that determines the optimal data tile parameters and maximizes data reuse according to register file size constraints. We demonstrate that our method is inherently general and that it should have a much broader applicability beyond the Cyclops-64 architecture.

- We present a register allocation method for register tiled loop bodies. We evaluate its effect through hand-tuned Cyclops-64 assembly code and observe a 6-fold reduction in load/store instructions.

- Our optimizations allow us to achieve an extremely high performance on the chip. The 19.17 and 27.50 GFlops achieved for a 700×700 and a more typical 1000×1000 matrix are, to our knowledge, the highest reported GFlops per chip rates so far.

To conduct our experiments, we used the FAST simulator [7], which is a functionally-accurate simulator of Cyclops-64. It is, however, worth to mention that our code has also been run on a first test implementation of the real chip. Results are within 15% up to 100 thread units. After that point, the general trend in performance is still followed quite accurately by the simulator.

The rest of this paper is organized as follows. In Section 2, we describe the Cyclops-64 architecture. In Section 3 we give a short overview on the current status of blocking algorithms for LU. In Section 4 we introduce our dynamic repartitioning algorithm. In Section 5 we present our register tiling method and in Section 6 we present the results of our experimental evaluation. In Section 7 we present related work and we conclude our paper in Section 8.

2. THE CYCLOPS-64 ARCHITECTURE

The Cyclops-64 (C64) chip is based on a multi-core-on-a-chip design, featuring 80 processors, each with two thread units (TUs). The chip is currently being developed by IBM and its design can be seen in Figure 1. Each processor is further equipped with a floating point unit and two SRAM memory banks of 32KB each. A 32KB instruction cache, not shown in the figure, is shared among five processors. The C64 chip has no data cache. Instead a portion of each SRAM bank can be configured as scratchpad memory (SP). The remaining sections are combined together, to form the global memory, which is uniformly addressable from all TUs. All TUs and SRAM banks are connected through a 96-port crossbar network, which provides a bandwidth of 4GB/s per port. This accounts to a total of 384GB/s on each direction. This huge bandwidth supports both the intra-chip communication as well as the six routing ports that connect each C64 chip to its neighbors. 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. This system incarnates the next generation of the Cyclops cellular architecture, which is designed to serve as a dedicated petaflop compute engine for running high performance applications.

The C64 architecture represents a major departure from mainstream microprocessor design in several aspects. The C64 chip integrates processing logic, embedded memory and communication hardware in the same piece of silicon. However, it provides no resource virtualization mechanisms. For instance, execution is non-preemptive and there is no hardware virtual memory manager. The former means that the C64 microkernel 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. From the processing core standpoint, a thread

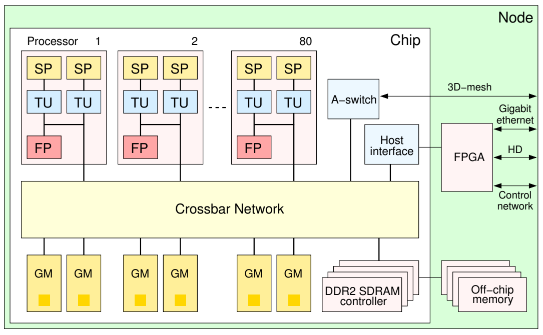

Figure 1: The architecture of a Cyclops-64 node.

unit is a simple 64-bit, single issue, in-order RISC processor with a small instruction set architecture, operating at a moderate clock rate (500MHz). Nonetheless, it incorporates efficient support for thread level execution. For instance, a thread can stop executing instructions for a number of cycles or indefinitely and can be woken up in a few tens of cycles by another thread through a hardware interrupt. C64 also provides an extremely fast hardware implementation of the barrier synchronization primitive.

3. CLASSIC BLOCK LU ALGORITHMS

The LU decomposition is in itself quite simple, as it only decomposes the matrix that describes a linear system into a product of a lower and an upper triangular matrix. Solving the linear systems described by these new matrices is then a trivial task. Due to its importance to scientific computing, it comes as no surprise that the LU decomposition is a well studied algorithm and many variations to it have been proposed, both for uni- and multi-processor systems. The significance of this application is further underlined by the fact that it is used as the primary means to characterize the performance of high-end parallel systems and determine their rank in the Top 500 list [20]. Variations to the LU algorithm include recursive methods [13], pipelining and hyperplane solutions [16], as well as blocking algorithms [14, 23], on which we will focus in this paper. Known implementations of the algorithm include the Linpack benchmark [1, 10] and its parallelized version *High-Performance Linpack* (HPL) [14]. The latter has been developed mainly for distributed-memory architectures, although it can also run on shared-memory systems. Another implementation is the one included in the SPLASH-2 benchmarks suite [23], which specifically targets shared-memory systems.

Although each of these implementations targets a different set of architectures, the algorithms used at the highest level are similar. The main concept is to partition the matrix into smaller blocks with a fixed size, each one being processed by one processor. The SPLASH-2 implementation divides the matrix into blocks that fit into the L1 data cache of a processor. HPL can be thought of having one more level of block creation. The matrix is initially divided into fixed size blocks that are distributed among nodes on the system, creating a first level of memory locality. Within each node, these blocks are further divided into smaller blocks, again of a fixed size, in order to take advantage of the cache hierarchy.

Figure 2 is an example of applying such a blocking algorithm to a matrix. On a shared-memory architecture, the whole matrix is assumed to be in the globally accessible memory address space. On a distributed system, each block is on a different node. Blocks with the same color can be executed in parallel. The algorithm starts by processing the diagonal block on one processor, while all other processors wait on a barrier. After this block finishes, the blocks on the same row and column with the diagonal block can be processed in parallel. Each processor will update a few of these blocks and will then wait again on a barrier. Finally, all other (inner) blocks can be processed, which completes the first step of the algorithm. The second step starts by moving to the next diagonal block and the whole procedure is repeated.

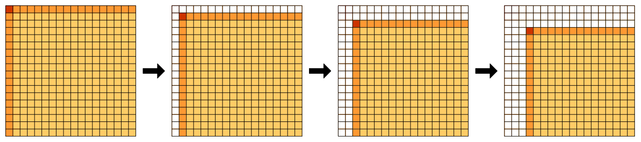

Figure 2: How the classic algorithms define blocks and progress in each step.

Despite its simplicity, the algorithm at the block-level has an important drawback. The number of available blocks in each parallel phase is usually not exactly divisible by the number of processors. This creates a load imbalance at each step, with the associated overhead accumulating over time. However, the exploitation of the cache on a per processor basis compensates to a large degree for this loss.

Nonetheless, LU implementations differ in a very important aspect, e.g., how they process each block. HPL uses the Level 3 *Basic Linear Algebra Subprograms* (BLAS-3) [9]. These routines perform operations between matrices. SPLASH-2 uses only the `DAXPY()` BLAS-1 routine, which updates a vector with a second vector, the latter having been multiplied with a scalar value, i.e., $\overrightarrow{y} \leftarrow \alpha \cdot \overrightarrow{x} + \overrightarrow{y}$. As already mentioned, all these routines are specifically optimized to exploit the data cache hierarchy of contemporary systems.

The last issue regarding LU is the stability of the computation. In order to achieve better accuracy, usually *partial pivoting* is employed, i.e., the largest coefficient in the same column with the current diagonal element is found in the remaining equations and the whole row that contains it is exchanged with the row that contains the diagonal element. In the rest of this paper, however, we do not employ this method for several reasons. Firstly, our focus is how to achieve the best possible performance on multi-core architectures with local-storage and not the stability of a specific algorithm. Secondly, pivoting is a completely independent pass in the LU decomposition and it could be easily added and optimized at any time. Moreover, the algorithms we propose are independent of pivoting. Therefore, the implementation of the rest of the application is not affected. Finally, for some common classes of scientific computing applications, pivoting is not required in practice because standard discretizations lead to diagonally dominant matrices.

4. DYNAMIC BLOCK LU ALGORITHM

In the previous section we identified how two features of applying a blocking algorithm for LU have been implicitly influenced by the fact that the algorithms were to be executed on cache-based systems. These are the use of a fixed

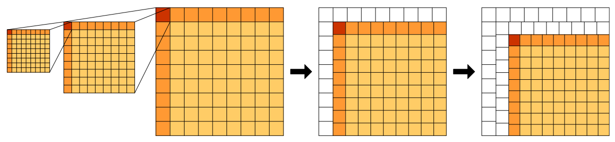

Figure 3: How the dynamic algorithm defines blocks and progresses in each step.

size for each block and the use of the BLAS routines to process each block. In this section, we argue that these decisions are poor choices for multi-core architectures with local-storage. Therefore, we introduce a new blocking algorithm, which focuses on load balancing, rather than memory access patterns.

Our blocking algorithm, which we will refer to as *Dynamic Repartitioning (DR)*, calculates the number of blocks and the size of each block based on the number of processors that are used to execute the algorithm, rather than the parameters of memory. More formally, we assume that we have a $N \times N$ matrix and that the number of available processors is P. In this case the matrix is divided into $B = (P/2) + 1$ blocks on each dimension. To make it easier to follow the description of our algorithm, we include Figure 3, which provides an example for 16 processors. As B might not exactly divide N, we define the first block, which will be the diagonal block in that step, to have a size equal to $\lfloor N/B \rfloor$. Since processing blocks in parallel can start only after the diagonal block has been processed, the latter should finish as soon as possible. Therefore, using the smallest size that also leads to a load balanced execution in the parallel phases benefits our algorithm. If N_R is the remainder of N/B, then $0 \leq N_R < B$. We define the size of the next N_R blocks to be $\lfloor N/B \rfloor + 1$. The remaining $B - N_R - 1$ blocks will have again a size of $\lfloor N/B \rfloor$. This assures that the size of all blocks does not differ more than one on each dimension. After applying the above algorithm to create the blocks and processing the diagonal block, we are left with $P/2$ blocks on the same row and $P/2$ blocks on the same column with the diagonal block. Hence, all P processors have exactly one block to process. In the second parallel phase, where all inner blocks are processed, we have a total of $(P/2) \cdot (P/2) = P^2/4$ blocks. Therefore each processor has to be assigned $P/4$ blocks in this phase. From this result, we conclude that our algorithm can be applied when P can be exactly divided by four. With a simple extension, we managed to apply the algorithm for every even number of processors. However, this is not the focus of our paper.

Although we managed to balance the work load in the first step of our algorithm, there remains an important issue. If we continue the execution of the algorithm with the current distribution of elements among blocks, we will have a load imbalance in all the following steps of the algorithm. The solution we adopted can be seen in Figure 3. Before each step of the algorithm begins, we repartition the remaining part of the matrix according to the same algorithm. Obviously, this creates at each step the same number of blocks, which results in good load balance, and only assigns less elements to each block.

In the first steps of the algorithm, each block might be quite large, which is an important drawback for the diagonal block. Using 16 processors on a 1000×1000 matrix, the diagonal block would have a size of $\lfloor 1000/((16/2)+1) \rfloor = 111$. The SPLASH-2 LU, in contrast, would create a much smaller diagonal block, typically 16×16. Since processing the diagonal block is performed by a single processor, this becomes a significant bottleneck. However, the operation performed on the diagonal block is actually a serial version of LU. Therefore, the bottleneck can be easily removed by recursively applying DR to the diagonal block. This is depicted in the left most part of Figure 3. The recursion stops when the number of elements in the last created diagonal block becomes smaller than $(P/2) + 1$, which would result in zero-sized blocks at the next level of recursion. After returning from each level of recursion, we can use all processors to solve the current diagonal block. Finally, after the last recursion completes, we can continue to the parallel phases in the initial matrix. During the next step of our algorithm, when we repartition the matrix, if the new diagonal block is large enough, we can apply the same procedure of recursions.

The opposite problem appears as we move towards the lower-right end of the matrix. Step after step, the part of the matrix that has to be processed becomes smaller. In the first implementation of our algorithm, this last part was calculated serially, after its size became less than $(P/2) + 1$. However, using a large number of processors would leave a large portion of the matrix to be updated serially. In order to overcome this problem, we modified our algorithm. After reaching the last part of the matrix we simply halve the number of processors we use. This allows each block to have a size larger than zero in the next step. Hence, we continue using the parallel algorithm. As we move further and the last part becomes again small enough, we again halve the number of processors and continue, until only one processor is left. This allows us to process in parallel the whole matrix. We refer to this optimization as *Processor Adaptation*. Notice that we also apply this optimization when recursively calling our parallel algorithm for the diagonal block. In this case, we just have to restore the number of processors each time we leave a level of the recursion.

One more optimization that can be applied, relies on the fact that the $P/4$ inner blocks that each processor has to update can be statically assigned to it. Therefore, we can choose the blocks to be continuous, combine them into a larger block and call only once the function that is used to update the inner blocks. As we will see in a later section, the best way to combine inner blocks in our algorithm is to use blocks that are on the same column. In other implementations this is not possible and blocks have to be assigned dynamically to processors. Moreover, the total number of blocks is much larger, which has two important consequences. The absolute function calling overhead per processor is much higher. The same holds for the percentage of function calling overhead compared to the processing time of each block, since the latter ones are smaller.

HPL is considered to be one of the best LU implementations and using it as a baseline to compare our algorithm to would be ideal. However, it requires an implementation of the MPI [18] message passing interface, even on shared-memory systems, which is not yet available on C64. Therefore, we decided to port the *Non-Contiguous* implementation of LU found in the SPLASH-2 benchmarks suite to C64. In the rest of this paper, we will refer to this port as the *Base Implementation (BI)*. We did not use the *Contiguous* version, as it accesses elements in each block through a jump table, which in turn contains the starting address

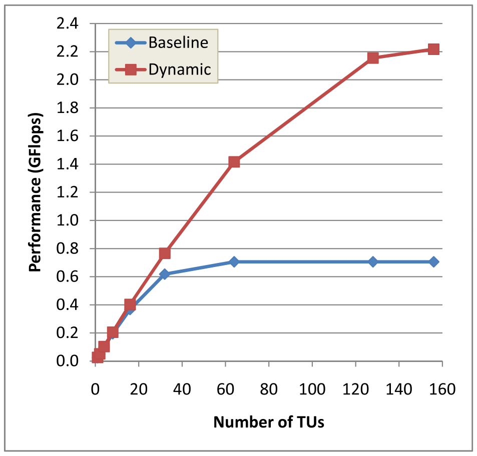

Figure 4: Performance of both algorithms on Cyclops-64 (700×700 matrix).

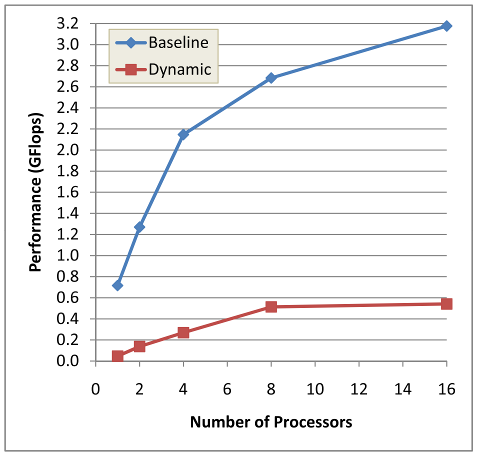

Figure 5: Performance of both algorithms on an Intel based system (4000×4000 matrix).

of each block. Thus, it requires two memory accesses per element. Although this is acceptable in the presence of a cache, it is not in our case, as each memory access has to go through the interconnection network to main memory. Specifically for C64, loading data from SRAM requires 20 cycles, whereas contemporary Intel based systems have a cache access time of 10 to 12 cycles.

Figure 4 presents the results of executing DR and BI on C64. For this experiment we used a 700×700 matrix, which is the maximum size that fits into SRAM. In order to only compare the blocking algorithms, we still use the `DAXPY()` routine to update each block. For BI we used the default block size of 16×16, which also proved to be the best choice. BI manages to closely follow DR up to 16 TUs. However, its performance remains almost unchanged for more TUs. In contrast, DR scales much better after that point, reaching 2.22 GFlops on 156 TUs, about 3.2 times better than BI.

Figure 5 presents the results of executing the algorithms on a 4 processor system. Each processor is a 3.0 GHz Intel Xeon with two cores. Each core supports HyperThreading and features a 32KB L1 and a 2MB L2 data cache. In order to be able to measure the execution time for this case accurately, we used a 4000×4000 matrix. As expected, the results are inverted. BI achieves a much better performance and scales far better than DR. These results strongly indicate that our approach better suits multi-core architectures with local-storage and that it can be used as a solid base for further improvements.

5. OPTIMIZING OPERATIONS WITHIN EACH BLOCK

5.1 Minimizing Memory Operations

In this section, we propose an alternative to `DAXPY()`, in order to process the elements of each block. `DAXPY()` takes advantage of the fact that a cache miss will move a whole cache line from main memory. On systems with local-storage, however, each request for data has to go through the interconnection network to main memory. Our goal, therefore, is to minimize the loads and stores that have to reach main memory. For C64, we can use SP and the register file of each TU for this purpose. In our implementation, we did not use the first one for several reasons. Firstly, as we will see in Section 6, the percentage of lost cycles that correspond to memory delays is extremely low, after optimizing register usage. Hence, using the synchronous data transfer method available on C64 would only add more overhead. If, on the other hand, we simulated asynchronous transfers, we would loose the performance of some TUs for actual calculations.

`DAXPY()` updates each row, using the rows above it, which is a good choice for cache-based architectures. For reasons that will become clear in Subsection 5.2, we changed this behavior and process the matrix by columns. Due to this change, we now have to use the stride of the matrix, to access the correct elements in each column. Initially, this slightly reduced performance, but was extremely important to finally improve it.

A well known method to maximize data reusage and minimize accesses to main memory is *Register Tiling* [4, 17], which has been successfully applied to matrix multiplication on C64 [15]. In this paper, we rely on this method to further improve performance, but extend it in a significant way. Register tiling is usually implemented as a compiler optimization and does not have a global, high-level knowledge of the algorithm that is used to solve a specific problem. It rather relies on the static semantics of the loop under consideration. In contrast, we took advantage of the global knowledge about our algorithm and especially data dependencies. This allows us to find an optimal solution for two problems, i.e., the optimal size for each tile and the optimal sequence in which tiles have to be traversed. We solved the first problem by taking into account the number of registers for a given architecture and the data dependencies between both, tiles and blocks created from our blocking algorithm. The second problem was solved by exhaustively analyzing all possible schemes to traverse tiles.

In order to clarify the above discussion, we describe this procedure through the example of Figure 6. In this example we assume that the inner block `C` has to be updated. The algorithm also requires data from two more blocks, one that is on the same row (`A`) and one that is on the same column (`B`) with the diagonal block. Furthermore, we assume that block `C` has a size of $N \times M$ and the diagonal block a size of $K \times K$. Block `B` has been subdivided into tiles of size $L_3 \times L_1$, whereas block `C` is subdivided into tiles

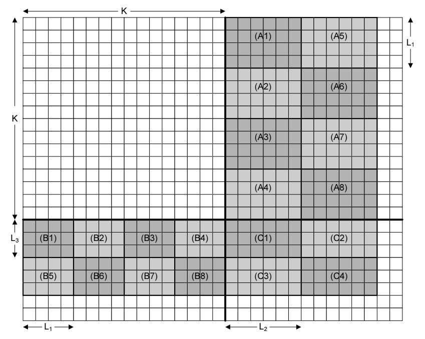

Figure 6: Dividing into tiles for `ProcessInnerBlock()`.

of size $L_3 \times L_2$. Due to data dependencies, block `A` has to be divided into tiles of size $L_1 \times L_2$. This can be derived from Figure 7, which depicts the source code of the `DAXPY()` function and the function that updates the inner block `C`, which we call `ProcessInnerBlock()`. For the rest of this section, we assume that the size of a block can be exactly divided with the size of each tile. This has no effect on the general results of our analysis, but makes it simpler to follow. Our implementation, however, takes into account these remainders and can be used to process matrices of any size. It can be derived that there are a total of 6 cases to create and traverse tiles when working with the inner blocks and all of them have been analyzed [22]. In the following paragraphs we present only the method to create tiles and how to traverse them that proves to be the optimal one.

We start by loading and keeping in registers tiles of size $L_1 \times L_2$, updating all possible tiles. Using the example of Figure 6, we would load `(A1)` and use it to update `(C1)`, in conjunction with `(B1)`. While still having `(A1)` in registers, we would load `(C3)` and `(B5)`, in order to update `(C3)` and we would repeat the same process for all other tiles under `(C3)`. This gives the following results:

$$\begin{aligned} Loads &= L_1 \cdot L_2 + (L_1 \cdot L_3 + L_2 \cdot L_3) \cdot (N/L_3) \\ Stores &= L_2 \cdot L_3 \cdot (N/L_3) \end{aligned}$$

At this point, `(A1)` is not needed anymore and we can load the next tile. This can be done either by moving horizontally to tile `(A5)` or vertically to tile `(A2)`. For this case, it does not matter which way we choose to continue and the result will be exactly the same, with respect to the total number of loads and stores required. This is because either way, there are a total of $K/L_1 \cdot M/L_2$ tiles that have to be loaded and in each case we need the aforementioned number of loads and stores. We chose to move vertically, as it allows for a more efficient implementation of index calculations. Hence, our final equations are:

$$\begin{aligned} Loads &= \left[L_1 \cdot L_2 + (L_1 \cdot L_3 + L_2 \cdot L_3) \cdot \frac{N}{L_3}\right] \cdot \frac{K}{L_1} \cdot \frac{M}{L_2} = \\ &= \frac{K \cdot M \cdot N}{L_1} + \frac{K \cdot M \cdot N}{L_2} + K \cdot M \\ Stores &= \left[L_2 \cdot L_3 \cdot \frac{N}{L_3}\right] \cdot \frac{K}{L_1} \cdot \frac{M}{L_2} = \frac{K \cdot M \cdot N}{L_1} \end{aligned}$$

According to the last equations, the number of loads and stores does not depend on L_3. Moreover, the bigger the

```
void ProcessInnerBlock(double **A, double **B,
            double **C, long K, long N, long M) {
  long i, j;

  for (i = 0; i < K; i++) {
    for (j = 0; j < M; j++) {
      DAXPY(&C[0][j], &B[0][i], -A[i][j], N);
    }
  }
}

void DAXPY(double *A, double *B, double X, long N) {
  long i;

  for (i = 0; i < N; i++) {
    A[i * Stride] += X * B[i * Stride];
  }
}
```

Figure 7: The initial source code of the `ProcessInnerBlock()` and `DAXPY()` functions in C.

remaining two parameters are, the smaller the number of loads required. Hence, we set $L_3 = 1$. If our architecture has R registers available, then one tile of each block has to fit into these registers:

$$\begin{aligned} & L_1 \cdot L_2 + L_1 \cdot L_3 + L_2 \cdot L_3 = R \Rightarrow L_1 \cdot L_2 + L_1 + L_2 = R \Rightarrow \\ & L_2 = (R - L_1)/(L_1 + 1) \end{aligned}$$

By replacing the value of L_2 in the equation that determines the number of loads and setting the derivative of that equation to zero, we can calculate the value of L_1 that minimizes the total number of loads. In order to keep the analysis short, we just present the final results:

$$\begin{aligned} & dLoads/dL_1 = 0 \Rightarrow R \cdot L_1^2 + 2 \cdot R \cdot L_1 - R^2 = 0 \Rightarrow \\ & L_1 = -1 \pm \sqrt{R+1} \Rightarrow L_2 = -1 \pm \sqrt{R+1} \end{aligned}$$

Notice that up to this point we did not refer to C64 at all, i.e., our method is general enough to be applied on any architecture. Applying it specifically to C64, the largest value of R that will also give an integer result is $R = 48$. Hence, $L_1 = L_2 = 6$ and our final equations are:

$$\begin{aligned} Loads &= (K \cdot M \cdot N)/3 + K \cdot M \\ Stores &= (K \cdot M \cdot N)/6 \end{aligned}$$

Other cases for processing the block would be to keep constantly in registers tile `(B1)` or `(C1)`. We also have to make a similar analysis for the diagonal block, as well as the blocks on the same row and column with the diagonal block, which leads to a total of 15 cases. Again, all of them have been thoroughly analyzed and only the best method for each case was used in our implementation. Table 1 summarizes the results of our analysis. For comparison, we include results when using `DAXPY()`. We remind the reader that the results for the optimized versions are not completely accurate, since we assumed that there are no remainders when we divide a block into tiles. Despite this fact, they can be used as a good estimate. From these results, we conclude that the number of loads and stores is reduced 4 times for the diagonal block and 6 times for all other blocks. It is important to notice that these numbers correspond to the optimal size of the tiles in each case. Therefore, the reduction would be larger for architectures with more registers. We realize,

Block Type	Loads (DAXPY)	Loads (Optimized)	Stores (DAXPY)	Stores (Optimized)
Diagonal	$\frac{2\cdot N\cdot(N^2-1)}{3}$	$\frac{N\cdot(N-1)\cdot(N+4)}{6}$	$\frac{N\cdot(N^2-1)}{3}$	$\frac{N\cdot(N+2)\cdot(N+4)}{6}$
Row	$M\cdot N\cdot(N+1)$	$\frac{M\cdot N^2}{6}+\frac{N\cdot(N+1)}{2}$	$\frac{M\cdot N\cdot(N+1)}{2}$	$\frac{M\cdot N\cdot(N+6)}{12}$
Column	$\frac{M\cdot(2\cdot N+1)\cdot(M-1)}{2}$	$\frac{M^2\cdot N}{6}+\frac{M\cdot(M-1)}{2}$	$\frac{M\cdot N\cdot(M-1)}{2}$	$\frac{M\cdot N\cdot(M+6)}{12}$
Inner	$2\cdot K\cdot M\cdot N+K\cdot M$	$\frac{K\cdot M\cdot N}{3}+K\cdot M$	$K\cdot M\cdot N$	$\frac{K\cdot M\cdot N}{6}$

Table 1: Comparing the number of loads and stores.

however, that applying this methodology manually to each application is a time consuming and error-prone procedure. Therefore, we believe that compilers should be made aware of an application's behavior at this level. To our knowledge, no compiler is currently able to identify all cases, make a detailed analysis as above and use the optimal case.

At this point, it is worth to mention that some recent studies [3, 19] propose a different approach to solve LU on multi-core systems with local-storage. They acknowledge the load-balancing problem when using fixed-sized blocks and try to solve it by simply reducing the size of each block. A dependence-based, dynamic scheduling algorithm is then employed to process all blocks. This procedure seems suitable for systems where local-storage can be accessed only by the core that owns it, as is the case in the aforementioned studies where Cell is used. We believe, however, that it introduces unnecessary complexity in the case of shared local-storage. Our static assignment of blocks to processing units is much simpler to implement. Moreover, the authors of these studies suggest that smaller blocks can better exploit the large register file of Cell. Not only do we agree with this statement, but we took it one step further by calculating the optimal size of each tile, instead of using classical square tiles. As a conclusion, these studies try to solve both problems by reducing block size, whereas we make a distinction between the load-balancing and optimal register file exploitation problems and propose a different, optimal solution for each one of them.

5.2 Architecture Specific Optimizations

Although the above results are already very important and promising, we can optimize our implementation even more, due to special load and store instructions provided by the C64 architecture. Specifically, our architecture provides the assembly instruction *'ldm RT, RA, RB'*, which combines several loads of data from memory into only one instruction. The register *RA* contains an address in memory. Starting from this address, consecutive 64-bit values in memory are loaded into consecutive registers, starting from *RT* through and including *RB*. Since each block is divided into smaller tiles that have either 4 or 6 elements consecutively in memory, we can take advantage of this instruction. This is also the reason why we changed our algorithm to perform calculations on columns, instead of rows, as described in Section 4. Notice that the number of elements that have to be loaded and operated on will still be the same in the optimized version, no matter if we use normal load operations or *ldm*. However, the number of load instructions that have to be issued, if we use *ldm*, will be less. Moreover, when using the normal load instructions, one request for data transfer is issued per element. If we use *ldm*, there is only one request for every 4 or 6 values. This reduces contention on the crossbar, allowing us to better exploit the

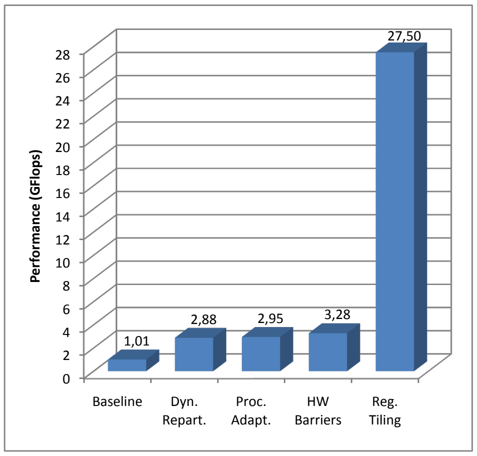

Figure 8: Evaluating the impact of each optimization on the performance of the application.

available bandwidth. Similarly to *ldm*, our architecture also provides the optimized store instruction *'stm RT, RA, RB'*, which we also used in our implementation.

Another optimization that has been employed for our purpose is the hardware implemented barrier, provided by the C64 architecture. It allows TUs to synchronize extremely fast and since barriers are the only synchronization operations required in our algorithm, it is important to use an efficient implementation.

6. EXPERIMENTAL EVALUATION

In this section we present the experimental evaluation of LU. Since the C64 chip is not yet physically available, all experiments had to be conducted using the FAST simulator [7]. FAST is an execution-driven and binary-compatible simulator of a multi-chip C64 system. It accurately represents the hardware components and reproduces the functional behavior of C64. It models in high detail all key components, such as the memory subsystem, the crossbar and other functional units. FAST has been extensively used by the C64 architecture design team at IBM for the purpose of chip design verification, and many developers for early application development. The development tool-chain for C64 includes version 4.1.1 of the gcc compiler. During compilation, we used the highest available optimization level (`-O3`).

First we evaluate the effect of each optimization on the performance of LU. During our experiments we used 156 TUs and a 1000×1000 matrix. Although only a 700×700 matrix fits into SRAM in the default chip configuration, it is possible to redefine the size of SRAM in the simulator. As will be explained in more detail, all matrices behave well for all optimizations that were described in Section 4. However, applying register tiling only shows its full potential for larger matrices. Figure 8 shows the results of our experiments. After applying all optimizations, we reach 27.50 GFlops, a 27 times speedup compared to BI.

In order to better explain these results, we include Figure 9, which is a break-down of the instruction mix executed by our algorithm. We use two cases to explain this large increase in performance. The first one is the C code before using register tiling (fourth column in Figure 8) and the second one is our assembly code, after register tiling (fifth

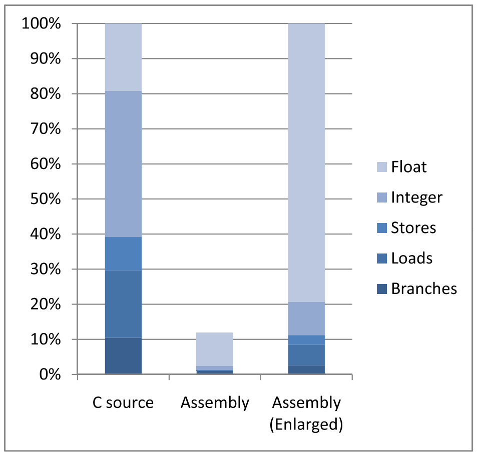

Figure 9: Break-down of instruction mix.

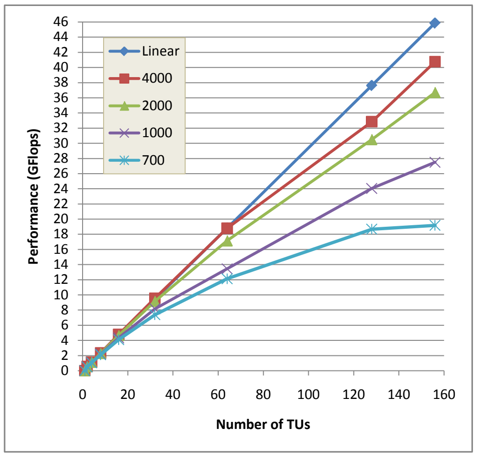

Figure 10: Performance achieved for different combinations of matrix sizes and number of TUs.

column in Figure 8). The first two columns in Figure 9 represent the relative number of instructions executed in each case. The third column is an enlarged version of the second one, to high-light the differences between the two cases. Our optimized code requires only 12% of the number of instructions that the unoptimized code executes. Since both versions solve the same problem, the absolute number of floating-point instructions does not change. However, we were able to reduce 28 times the number of load and store instructions issued. We remind the reader that in our analysis the number of load and store instructions is shown to be reduced approximately 6 times due to register tiling and 6 more times due to the use of *ldm* and *stm* instructions. A pleasant by-product, is the 36 times reduction of integer instructions, mainly because index calculations are reduced for the same reasons. As we can see, the experimental and theoretical results are fairly close in this case. Due to these improvements, the total time lost while waiting for data from memory dropped from 31.4% to 4.7%.

From these results, it is obvious that there are two optimizations that contribute the most in achieving the maximum performance. The first one is the introduction of DR. Although the performance is not very high, compared to the final numbers we achieve, it marks the first leap from the extremely slow BI. Effectively, it provides a solid basis, on which we can build our more advanced optimizations. Moreover, it proves that our departure from algorithms that implicitly rely on a cache hierarchy was a correct decision. The second important optimization is register tiling. This proves that our exhaustive and analytical approach for this optimization was driven by correct assumptions about its importance on our architecture. Additionally, it confirms that the introduction of an application-aware implementation of register tiling can lead to better exploitation of the available hardware.

As previously mentioned, only a 700 × 700 matrix can fit into SRAM on the default configuration of C64. The performance of our algorithm for this matrix and different numbers of TUs is depicted in Figure 10. Although the total performance using 156 TUs is quite high, our algorithm does not scale very well. After thoroughly studying the execution traces of several experiments, we concluded that the main problem is the current size of the available SRAM, which limits the number of elements that have to be updated by each TU. As the efficiency of register tiling depends on the amount of data that is being reused, smaller data sets are not fully exploiting this optimization. In order to verify our explanation, we run another set of experiments. Fortunately, FAST allows us to redefine the size of SRAM. Using this feature, we run our application for larger matrix sizes, ranging from 1000 × 1000 up to 4000 × 4000. The results of these experiments are also depicted in Figure 10. As can be seen, the performance improves dramatically as the size of the matrix increases. These experiments confirm that our algorithm is capable of achieving good performance and that the current limit is the size of SRAM. In turn, the size of the SRAM is currently limited only by the size of the chip, as it must co-exist with 80 cores, the crossbar and all other functional units. As manufacturing processes improve and the integration of chips increases, the size of SRAM will eventually become larger. Therefore, our experiments with larger matrices provide a useful insight on the performance that eventually can be achieved by C64.

Finally, we compare our results with the results reported for other architectures. Table 2 summarizes the performance of HPL for a variety of systems, on a per chip basis. The results for Cell were obtained for a 1000 × 1000 matrix. For all other systems, the results are the ones reported in the Top500 list [20] and the matrix size was the largest possible for the available memory. This gives these systems an edge over C64, as larger matrices allow better utilization of the hardware and higher performance metrics. Although our implementation of LU is not based on HPL, this comparison can still give some interesting insights.

Even for the smallest matrix we used, the 19.17 GFlops of C64 outperform the next best architecture, which is Cell with 9.46 GFlops. The difference rises much higher, if we use the matching 1000×1000 matrix on C64. All other architectures achieve an even lower total performance, even though they operate on larger matrices. A significant drawback of C64, is the fact that it achieves a much lower efficiency. The peak performance that can be achieved on 156 TUs is 78 GFlops, therefore the efficiency is 24.6% for the 700 × 700 matrix, 35.3% for the 1000 × 1000 matrix and reaches only 52.3% for the largest matrix we used. This can be explained by the fact that we use a much higher number of TUs, re-

	Peak GFlops	Actual GFlops	Efficiency
Cell BE 3.2 GHz	14.63	9.46	64.66%
Power5 1.9 GHz	7.60	6.21	81.71%
Itanium2 1.6 GHz	6.40	5.30	82.83%
Pentium 4 Xeon 3.6 GHz	7.20	5.20	72.22%
Cray XT3 2.6 GHz	5.60	4.17	74.48%
Opteron 2.8 GHz	5.60	3.87	69.10%
Blue Gene/L 700 MHz	2.80	2.14	76.46%

Table 2: Maximum performance with one processor.

ducing the data set that each one operates on. In contrast, Cell has only 8 Synergistic Processing Units (SPUs), each one operating at a much higher clock rate. Using 8 TUs on C64 for the 1000×1000 matrix gives an efficiency rate of 57.5%, which is much closer to the one achieved by Cell.

7. RELATED WORK

Apart from the HPL [14] and SPLASH-2 [23] implementations of LU, which we analyzed and compared to our own algorithms, others also exist. A recursive algorithm for LU has been proposed for uni-processor systems [13]. The main characteristic of this algorithm is that it tries to reduce the working set at each level of recursion, to better exploit the cache. In contrast, we applied recursion on a parallel algorithm and with a totally different goal, i.e., to improve load-balancing among processors.

Pipelined and hyperplane (or wavefront) algorithms have also been studied for shared-memory systems [16]. The first category divides the matrix into horizontal stripes. As soon as a processor finishes with the first column in its stripe, it moves to the second column and informs the next processor that it can calculate its own first column. This creates a pipelined execution, which is again specifically designed to take advantage of the cache. Wavefront algorithms, on the other side, fail to exploit effectively the cache, but allow more parallelism. Their drawback is that they require much more fine-grained synchronization. However, they are extremely suitable for specific parallel architectures, e.g., systolic arrays.

Currently, we are trying to apply our methodology to other linear algebra problems. A good candidate seems to be matrix multiplication. Although this application has already been studied on C64 [15], the approach is not as systematic as ours and preliminary results indicate that we are able to achieve a much higher performance.

8. FUTURE WORK AND CONCLUSIONS

In this paper we present a methodology to design algorithms for multi-core architectures with local-storage and apply it to implement the LU decomposition for C64. Initially, we established that the design of current algorithms for linear algebra problems relies, either explicitly or implicitly, on the fact that they are executed on cache-based architectures. We show that this approach does not scale well for the architectures under consideration and propose a data distribution scheme that favors load-balancing, instead of memory access patterns. If required by the application, we redistribute data at appropriate points, to ensure that load-balancing is maintained throughout execution. For the same reason, we recursively apply our parallel algorithm to parts that traditionally have been solved serially.

Currently, register tiling depends on the ability of the compiler to discover data dependencies among elements that are accessed in loops. However, the current status of compilers only allows them to have a narrow view of these dependencies, within the limits of the loop itself. We minimized the number of memory accesses required to load data by applying register tiling in an application aware manner and mathematically proved the best way that it can be applied on a given architecture.

Regarding our *Dynamic Repartitioning* algorithm, we believe that it is an important departure from more conventional solutions. This algorithm has been developed due to the realization that traditional blocking algorithms trade the performance gain of load balancing with the larger gain due to the exploitation of the cache. Since this cannot hold in the architectures under consideration, we believe that algorithms targeting multi-core architectures with local-storage should focus more on load balancing. Similar arguments can be made for the register tiling method we propose. The fact that a cache miss will move a whole cache line from main memory has a definite impact on the methodology followed when applying register tiling, as it should be taken into account when loading data into registers, if maximum performance is to be achieved. Again, this fact does not hold in our case and which values are loaded into registers should be adapted accordingly.

It is important to mention that in this paper we focused mainly on blocking algorithms. However, it remains an open question whether other LU decomposition algorithms would perform better on our architecture, especially after optimizing them as carefully as we did for the proposed blocking algorithm. Currently, we consider the possibility to implement other algorithms on C64. Certainly, however, we can argue that the implemented combination of algorithms and methodologies does provide an extremely high performance.

An important aspect is the fact that the C64 architecture actually includes a large number of nodes. Currently, however, our application only runs on one node. Our work up to this point remains relevant, even if we decide to expand our algorithm to run on more nodes. As previously mentioned, the update of the diagonal block is in itself an LU decomposition of smaller size. Therefore, the code that we developed would be reused to process the diagonal block on just one node. However, routines that would handle all other kinds of blocks would have to be written in this case. Finally, some care would also be required if pivoting is desired. During this independent phase of the algorithm, some communication between nodes would be necessary.

9. ACKNOWLEDGEMENTS

This work was supported by the NSF (Projects CNS-0509332, CSR-0720531, CCF-0833166 and CCF-0702244) and other government sponsors. We would like to thank all members of the CAPSL group at the University of Delaware for their help, as well as everyone else who helped at ETI and IBM.

10. REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 3^{rd} edition, 1999.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Dec. 2006.

[3] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. LAPACK Working Note 194, Nov. 2007.

[4] D. Callahan, S. Carr, and K. Kennedy. Improving Register Allocation for subscripted Variables. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 53–65, White Plains, NY, June 1990.

[5] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and its First Implementation: A Performance View. http://www-128.ibm.com/developerworks/power/library/pa-cellperf.

[6] Clearspeed White Paper: CSX Processor Architecture. http://www.clearspeed.com/newsevents/presskit.

[7] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. FAST: A Functionally Accurate Simulation Toolset for the Cyclops-64 Cellular Architecture. In *Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation (MoBS 2005)*, Madison, Wisconsin, June 2005.

[8] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. TiNy Threads: a Thread Virtual Machine for the Cyclops64 Cellular Architecture. In *Proceedings of the 5th Workshop on Massively Parallel Processing*, Denver, Apr. 2005.

[9] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[10] J. J. Dongarra and D. W. Walker. Software Libraries for Linear Algebra Computations on High Performance Computers. *SIAM Review*, 37(2):151–180, 1995.

[11] W. Eatherton. The Push of Network Processing to the Top of the Pyramid. Keynote at the Symposium on Architectures for Networking and Communication Systems, Princeton, NJ.

[12] K. Gallivan, W. Jalby, and D. Gannon. On the problem of optimizing data transfers for complex memory systems. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 238–253, St. Malo, France, 1988.

[13] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–753, Nov. 1997.

[14] HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl, 2004.

[15] 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)*, pages 134–144, Dresden, Germany, Aug. 2006.

[16] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report nas-99-011, NASA Ames Research Center, 1999.

[17] K. S. McKinley, S. Carr, and C. W. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, 1996.

[18] Message Passing Interface Forum. *MPI-2:Extensions to the Message-Passing Interface*, 2003.

[19] G. Quintana-Orti, E. S. Quintana-Orti, E. Chan, R. A. van de Geijn, and F. G. V. Zee. Design and Scheduling of an Algorithm-by-Blocks for the LU Factorization on Multithreaded Architectures. FLAME Working Note 26, Sept. 2007.

[20] The Top500 List. http://www.top500.org.

[21] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *Proceedings of the 2007 IEEE International Solid-State Circuits Conference*, pages 5–7, San Francisco Marriott, CA, USA, Feb. 2007.

[22] I. E. Venetis and G. R. Gao. Optimizing the LU Benchmark for the Cyclops-64 Architecture. Technical Memo 75, Computer Architecture and Parallel Systems Laboratory, University of Delaware, Feb. 2007. http://www.capsl.udel.edu/publications.shtml.

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[24] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao. Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 35–45, San Diego, California, USA, June 2007.