Optimizing the LU Benchmark for the Cyclops-64 Architecture

Ioannis E. Venetis
Guang R. Gao

CAPSL Technical Memo 84
Original publication date: February 14, 2007
Date of revision: July 8, 2009

An abbreviated version of this Technical Memo is included in the Proceedings of the 2009 ACM International Conference on Computing Frontiers, Ischia, Italy, May 2009.

Abstract

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 results of a more radical 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, in place of a hardware managed cache hierarchy. These two subsystems communicate through a powerful on-chip interconnection network, which is capable of providing a very high bandwidth. However, what remains an open question is what the programming model of this new class of multi-core architectures should be. In this report we present an implementation of the LU application for Cyclops-64, an architecture that fits into the above category. Through this experience, we identified a number of program developing methodologies that are extensively used on cache-based parallel systems to improve performance, but behave poorly on Cyclops-64. These include algorithmic design, the interaction between the high-level algorithm and the architecture and architecture specific optimizations. Moreover, we identified methodologies that improve performance on both kind of systems. Along with the description of our algorithm for LU and the experimental evaluation, we analyze and explore the impact of those methodologies on the performance of LU and provide alternatives whenever they fail on our architecture. As a result, we are able to achieve a performance of 11.19 GFlops with double-precision floating point numbers, even for a small matrix of size 512×512. To our knowledge, this is the highest GFlops per chip rate reported so far for this application.

Contents

List of Figures

List of Tables

1 Introduction

Large scale parallel systems are nowadays increasingly used to solve important scientific and engineering problems. Therefore, the performance that a specific architecture can deliver for solving these problems can be of great importance. In many of these applications the problem of quickly solving dense linear systems arises. Examples include the design of airplane wings, the flow of gases or liquids around solid objects, diffusion of solid objects in liquids and diffusion of light through small particles. Although for some problems the linear systems that emerge are symmetric and more efficient methods can be used to solve them, in the general case the LU decomposition algorithm is employed to perform that task. The algorithm itself is quite simple, as it only decomposes the matrix that describes the 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. These include recursive methods [10], blocking algorithms [11, 19], pipelining and hyperplane (or wavefront) solutions [13]. 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 [17].

Naturally, all of the above algorithms are designed to exploit the characteristics of the underlying architecture in the best possible way. Despite their differences in their implementation, contemporary parallel architectures 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 it's own memory, creating either a distributed or distributed-shared memory address space. As a result, algorithms for these systems have to be explicitly designed to take advantage of memory locality, in order to achieve high performance.

Lately, a new trend in parallel computer architecture is gaining importance and starts to show promising results. Multi-core-on-a-chip systems have significantly different characteristics, compared to previous architectures. They integrate a large number of processing cores in a single chip and instead of a hardware-managed cache hierarchy, they usually include a software-managed memory hierarchy, visible to the programmer. Cyclops-64 [7] and the recently announced 80-core Intel processor [18] are examples of such architectures. The challenge for these architectures is to show that they can outperform previous designs for problems of immediate interest to the computing community. If this can be achieved, they will have a chance to become the computing platforms of the near future.

These platforms however have currently a significant disadvantage, compared to more traditional architectures. The latter have been thoroughly studied for many years and techniques that allow applications to take full advantage of the underlying hardware have been developed. On the other hand, multi-core architectures are relatively new and such general directions for application development do not exist yet.

In this report we present an implementation of the LU decomposition algorithm that targets a single Cyclops-64 chip, a multi-core architecture with a software-managed memory hierarchy. Through this implementation, we identified a number of programming methodologies that are extensively used on cache-based parallel systems to improve performance, but behave poorly on multi-core architectures. Moreover, we identified methodologies that improve performance on both kind of systems. Specifically:

- We show how partitioning the original matrix into blocks, in order to assign them to processors, is influenced by the assumption that the underlying architecture has a cache. This scheme does not fit our case and we provide an alternative *Dynamic Repartitioning Algorithm*, in conjunction with *Recursion on the Diagonal Block* and *Processor Adaptation*.
- The methodology to update the elements within each block also relies on the existence of a cache. Due to the lack of a cache on Cyclops-64, we use instead the next level of high-speed storage, which is the register file. We show that applying register tiling in an application-aware manner, instead of relying on the static semantics of a loop, can provide a huge increase in performance.
- We verify that instruction scheduling on Cyclops-64 is extremely important, due to the in-order execution engine of the chip and can further improve performance.
- The result of applying all of our optimizations allows us to achieve an extremely high performance on the chip. The 11.19 and 21.92 GFlops achieved for a 512×512 and 1024×1024 matrix respectively are, to our knowledge, the highest reported GFlops per chip rates so far.

The rest of this report is organized as follows: In Section 2, we describe the Cyclops-64 architecture. In Section 3 we describe the high-level algorithm that is used in most blocking implementations of LU. In Section 4 we introduce our dynamic repartitioning algorithm for defining the number and size of each block. In Section 6 we present how register tiling can significantly improve performance. In Section 7 we further improve performance by better scheduling instructions at the application level. In Section 8 we present the results of our experimental evaluation, whereas in Section 9 we present related work. Finally, Section 10 concludes this report and gives some directions for future work.

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 design of the processor can be seen in Figure 1. Each processor is further equipped with one 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 of SRAM are combined together, to form the global memory (GM), 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 nearest neighbors in the system. 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

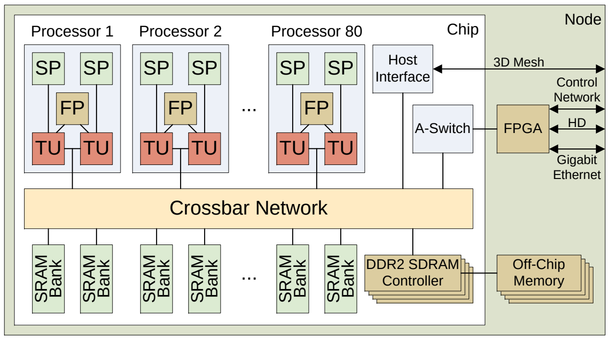

Figure 1: The architecture of a Cyclops-64 node.

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 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 The Classic Block-LU Algorithms

In this paragraph, we give a short overview on how most classic blocking algorithms for LU work. During the initial phase of our research on the topic, we considered using one of the well known LU algorithms and modify it, so that it better maps to the C64 architecture. As we will see, however, this would not be as easy as initially thought. The first algorithm that we considered was the Linpack benchmark [1, 9]. Since this implementation is not parallelized, we focused our attention on *High-Performance Linpack* (HPL) [11]. This implementation has been developed mainly for distributed-memory architectures, although it can also run on shared-memory systems. However, this benchmark requires an implementation of MPI [16], which is currently not available on C64. Moreover, since our initial implementation targets only one chip, even if the MPI interface would be available, the overhead of using it in our case would be significant.

We finally opted for a parallel implementation that specifically targets shared-memory systems, which is the LU in the SPLASH-2 benchmarks suite [19]. This implementation directly uses threads to express parallelism, which is extremely convenient in our case. The C64 tool-chain provides the TiNy-Threads run-time library [7], which provides an API to manage threads that execute on a C64 chip. The library directly maps each thread that is created to a TU, hence the associated cost of managing parallelism is kept extremely low. Creating a first version of the application for C64 was quite easy, since it only required the modification of a few predefined macros that create and join threads, as well as those that handle barriers.

Although each of the above implementations targets a different set of architectures, the algorithms used at the highest level are quite 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

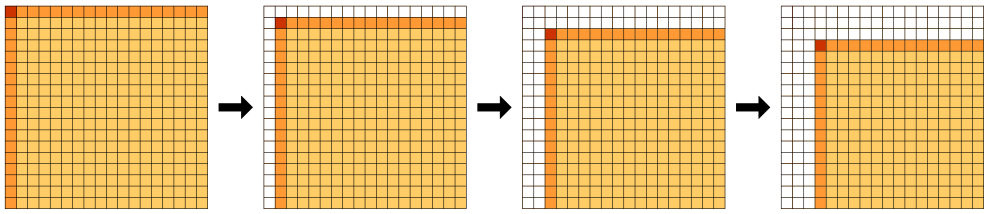
Figure 2: How the classic algorithms define blocks and progress in each step.

that fit into the L1 data cache of a processor. A typical block size that gives good results is 16×16. The HPL algorithm can be thought of having one more level of block creation. Since it targets mainly distributed-memory systems, the matrix is initially divided into a number of blocks, equal to the number of nodes on the system. Each node receives a block, which creates a first level of memory locality. Similarly to the previous case, these blocks have a constant size. 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.

There is, however, one important difference between the implementation of HPL and SPLASH-2. The first one uses the Level 3 *Basic Linear Algebra Subprograms* (BLAS-3) [8, 14], in order to update the elements within each block. These routines perform operations between vectors and matrices. The latter uses only the `DAXPY()` Level 1 BLAS routine, which updates a vector with a second vector, the latter having been multiplied with a scalar value, i.e., $y \leftarrow \alpha \cdot x + y$, where x, y are vectors and α is a scalar. The drawback of these implementations, for the architecture under consideration in this report, is the fact that they both use BLAS routines to perform the basic operations within each block. These routines have been specifically designed to take advantage of the data cache hierarchy, since this is the dominant computer system design nowadays. However, multi-core architectures like C64, Cell, ClearSpeed CSX and others employ an explicitly programmable local memory storage. BLAS routines map poorly on such multi-core chips, resulting in very low performance, as we will see later.

To continue our discussion and compare the above algorithms with the one we propose in the next section, we include Figure 2 as an example. We assume that the initial matrix is divided into smaller blocks. 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 (dark red color) 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 (dark orange color). Each processor will update a few of these blocks and when it has no more blocks to process in this phase, it waits again on a barrier. Finally, all other (inner) blocks can be processed (light orange color), which completes the first step of the algorithm. The second step starts by moving to the next diagonal block to update it. Notice that the first row and column of blocks are not processed now. The blocks on the same row and column with the new diagonal block are the next ones to be processed. The second step completes with the processing of all other colored blocks. The algorithm continues by moving each time to the next diagonal block, until it processes the last one. After each step, one more row and column of blocks is discarded and not processed in the next step.

Despite its simplicity, the algorithm at the block-level has an important drawback. On a shared-memory system, 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. On a distributed system, the processors that processed the diagonal block and the blocks that share the same rows and columns with it, are left idle on all the following steps. However, the exploitation of the cache on a per processor basis compensates at a large degree for this loss in both of the above cases. For the rest of this report, we will refer to the SPLASH-2 implementation of LU as the *Base Implementation*, since it is the one we started with.

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 report, 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 always needed in practice because standard discretizations lead to diagonally dominant matrices.

4 The 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 architectures. The first one is the use of a fixed size for each block, in order for the block to fit into local memory and/or into the cache of each processor. The second one is the use of the BLAS routines to process each block, which are specifically designed to take advantage of the cache.

In this section, we argue that both these decisions are a poor choice for modern multi-core architectures that lack a cache-hierarchy. With respect to the BLAS routines, they rely on the fact that accessing one element in memory, fetches more elements into the cache of a processor. Consequently, these elements can be accessed faster, if immediately used. This is however not true for the architectures we examine, where access to each element has to reach main memory through the interconnection network. As a result, every access to an element is more expensive and we should try to find alternative ways to minimize them. This is also the reason why we chose to use the *Non-Contiguous* version of the SPLASH-2 implementation. The second alternative (*Contiguous*) accesses elements in each block through an auxiliary table, requiring two memory accesses to load each element. On the other hand, using a fixed size for each block creates imbalance in the distribution of work. Combined with the fact that accessing elements in main memory is much slower, the imbalance caused at the end of each step of the algorithm has a greater effect on the total execution time.

In the rest of this section, we will present an alternative method to divide the matrix into the blocks that will be distributed among the available processors, which we will refer to as *Dynamic Repartitioning*. For the time being, we will continue using the BLAS-1 `DAXPY()` routine, in order to update the elements of each block. According to our previous analysis, our main goal should be to improve load balance among processors. In order to achieve this goal, we will return to the description of the blocking algorithm in the previous section and we will assume that we use 16 processors. The algorithm starts by processing the diagonal block. After it finishes, it processes all blocks on the same row and column with the diagonal block. These blocks can all be processed in parallel. Therefore, we should ideally have 16 blocks of equal size, in order to achieve the best possible load

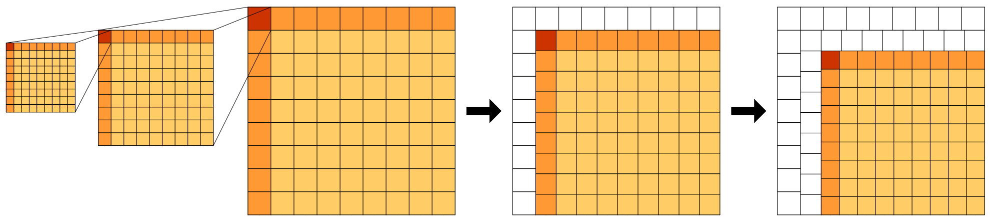
Figure 3: How the dynamic algorithm defines blocks and progresses in each step.

balance in this example. This can be achieved if we divide the initial matrix into 9 equally sized blocks on each side. This is depicted in Figure 3. After we process the diagonal block, 8 blocks remain on the same row and 8 blocks on the same column with that block (dark orange color). Hence, all 16 processors can be kept busy. This algorithm of partitioning the matrix into blocks has one more advantage. As can be seen in the same figure, the number of blocks that remain (light orange color) after we process all the above blocks are $8 \times 8 = 64$, which can be exactly divided by the number of available processors. This means that we can statically define which of these blocks will be executed on each processor and we can achieve again excellent load balance.

As a result, the number of blocks and the size of each block are not defined by the parameters of the memory, as on a cache-based system, but rather by the number of processors that are used to execute the algorithm. In order to express the above description more formally, we assume that we have a matrix of size $N \times N$ 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. As a result, the size of each block on each dimension should be $N_B = N/B$. Obviously, the last result might not be an integer. The way we handle this case is as follows. We define the first block, which will be the diagonal block in that step, to have a size equal to the floor of the above division ($\lfloor N/B \rfloor$). 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 the above division ($N_R = N \bmod B$), it can have a value ranging from zero up to $B - 1$. We define the size of the next N_R blocks to be equal to the ceiling of the above division ($\lceil N/B \rceil$). The rest of the blocks will have again a size equal to the floor of the same division. As a result, each block can have a size of $\lfloor N/B \rfloor$ or $\lfloor N/B \rfloor + 1$ on each dimension. This means that the total size of each block is almost equal to the size of each other block.

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 and all of them have almost the same size. 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. This number can always be exactly divided with P, 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 currently not the focus of this report, therefore we will not describe further this extension.

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, since there will not be enough blocks for all processors. The solution we adopted for this problem is simple and 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. For the same example we used above, we would divide the remaining part of the matrix again into 9×9 blocks. 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. This is also depicted in the above figure.

In the first steps of the algorithm, each block might be quite large. This is not a problem for the blocks that are processed during the parallel phases, since all processors will be busy. However, it is an important drawback for the diagonal block. The larger the diagonal block, the more time it requires to be processed and the later the parallel phases can be invoked. To explain the problem better, assume that we have a 1000×1000 matrix and 16 processors. This would mean that we have to create 9 blocks on each direction and the diagonal block would have a size of $\lfloor 1000/9 \rfloor = 111$. The classic algorithm would create a much smaller diagonal block, typically 16×16, which is about 7 times less. Since processing the diagonal block is performed by only one processor, this becomes a significant bottleneck in our algorithm. Fortunately, there is a simple solution to this problem too. The operation performed in the diagonal block is actually a serial version of LU. To remove the bottleneck, we can recursively apply our parallel algorithm on the diagonal block. This is also depicted in the left most part of Figure 3, where the diagonal block is magnified and the same algorithm is applied. Using the same example as above, we would apply our parallel algorithm to the 111×111 diagonal block of the initial matrix and divide it into 9×9 blocks. This would give a new diagonal block of $\lfloor 111/9 \rfloor = 12$. The recursion stops when the number of elements in the last created diagonal block becomes smaller than the value $(P/2) + 1$, because this means that if we applied one more level of recursion, some of the newly created blocks would have a size of zero. In the example we used, we can apply one more level of recursion. The new diagonal block would have a size of $\lfloor 12/9 \rfloor = 1$, which is trivial to calculate. Returning from this level of recursion, we can use all processors to solve the 12×12 block. Finally, we return to the last level and again using all processors, we calculate the 111×111 block. After this step 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 divide the number of available processors by two. 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 divide again the number of available processors by two 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 this optimization can also be applied when recursively calling our parallel algorithm for the diagonal block, as long as the number of processors is restored, 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 and combine them into a larger block. As a result, we need to call only once the function that is used to update the inner blocks, instead of $P/4$ times. In the base implementation this is not possible. Since the number of blocks that have to be processed changes after each step, they have to be assigned dynamically to the processors. Moreover, the total number of blocks is much larger, since the block size is chosen to be small. As a result, the overhead of

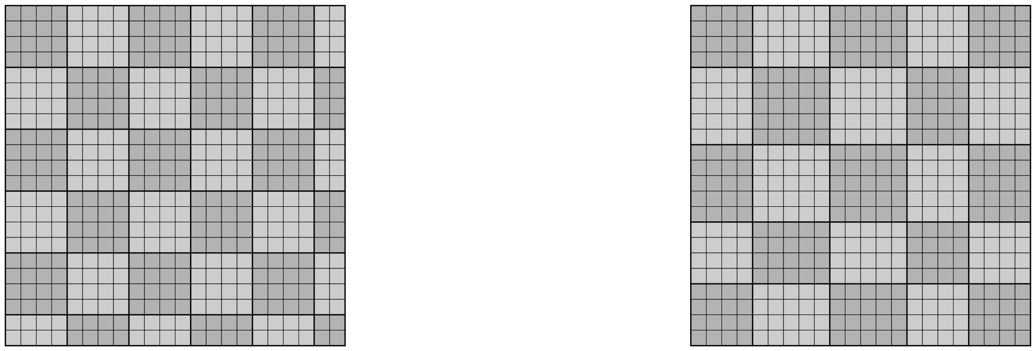

Figure 4: The number of elements in each block for the classic and dynamic algorithm.

calling functions is considerably higher in this case. 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.

A final and more detailed example of how we divide a matrix into blocks and how many elements each block contains can be seen in Figure 4. For this example, we assume that we have a matrix with a size of 22×22 and 8 processors. On the left part of the figure, we have applied blocking with a size of 4×4. As can be seen, there is a remainder at the end of both dimensions, which leads to the creation of smaller blocks. After the first diagonal block has been processed, all 8 processors will be used to update blocks of the first row and column in the matrix. However, the two smaller blocks on the same row and column will have to be updated in a second pass and only two processors can be used for this. The remaining six processors have no useful work to do at this point. On the right part of the figure, we demonstrate how our dynamic repartitioning algorithm would divide the matrix. It will create $(P/2) + 1 = (8/2) + 1 = 5$ blocks. The diagonal block will have a size of $\lfloor 22/5 \rfloor = 4$. The remainder of this division is $22 \bmod 5 = 2$, which means that the following two blocks will have a size of $\lceil 22/5 \rceil = 5$. Finally, the remaining blocks will have again a size of 4. As can be seen, there are blocks with dimensions 4×4, 4×5, 5×4 and 5×5. Although these differences seem significant, proportionally to the size of each block in this example, their importance diminishes as the size of the matrix grows, which explains why we can achieve better load balance.

Figure 5 presents the results of applying the dynamic repartitioning algorithm. Since one C64 chip has in the default configuration only 2.5MB of SRAM available, only small matrices can fit into it. Therefore, we used a matrix of size 512×512. For the base implementation we used the default block size of 16×16, which also proved to be the best choice. This implementation manages to closely follow the dynamic algorithm up to about 16 TUs. However, its performance remains unchanged after using 32 or more TUs. In contrast, the dynamic repartitioning algorithm scales much better after that point, obtaining 1.81 GFlops on 128 TUs, which is about 3.5 times better, compared to the base implementation. This result verifies that our approach up to this point is valid and can be used as a solid base for further improvements.

5 Analysis of Memory Operations in the Dynamic Block-LU Algorithm

Despite the differences between the classic and the dynamic LU algorithms, in defining the number and the size of blocks, the methods used to update the elements of each block remain the same in both cases. In this section

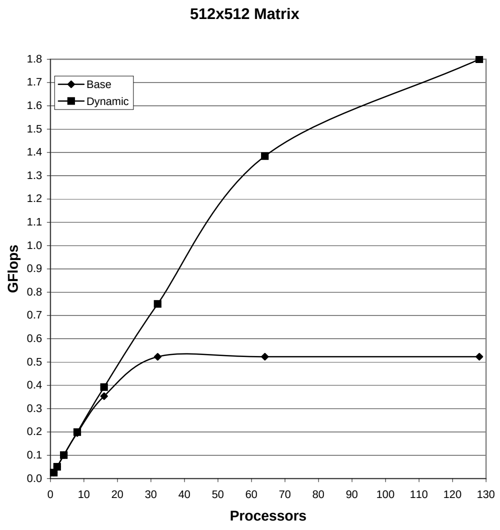

Figure 5: The performance of the base implementation and the dynamic repartitioning algorithm.

we analyze the behavior of these methods, with respect to the number of loads and stores required to implement the algorithm. This is quite important, since the Cyclops-64 architecture has a characteristic memory hierarchy, which elevates the importance of data transfers to an extremely high level. Moreover, this will prove very useful later (Section 6), when we will introduce alternative methods to update each block and compare them to the original ones. As we will see, reusing as much as possible data that has been already loaded into registers, can lead to reduced data transfers and boost performance by an order of magnitude.

It is also worth to mention at this point, that the base implementation operates on each row of the matrix. More specifically, the `DAXPY()` routine updates each row, using the rows above it, which is a good choice for the targeted architectures. The C language stores matrices by row and accessing an element will automatically load more elements of the same row into the cache. Initially, we used the same convention in our algorithm. However, for reasons that will become clear in Section 7, we decided to change this behavior and instead process the matrix by columns. If the matrix is available from another application, we only have to load it transposed into memory, in order to get the same results with the base implementation. Moreover, when using `DAXPY()`, 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.

It is important to clarify that the size of each type of block in the following paragraphs is not necessarily equal to the size of any other type of block, although we use the same symbols to represent each dimension. For example, we assume that a block processed with the function `ProcessBlockOnRow()` has N rows and M columns. Although we represent the dimensions of a block processed with the function `ProcessInnerBlock()` with the same symbols, this does not mean that these two blocks have the same size. We merely use the same symbols for uniformity reasons.

```
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 6: The source code of the `DAXPY()` function in C.

```
 1  void ProcessDiagonalBlock(double **A, long N)
 2  {
 3          long    i, j;
 4
 5          for (i = 0; i < N; i++) {
 6                  for (j = i + 1; j < N; j++) {
 7                          A[i][j] /= A[i][i];
 8                          DAXPY(&A[i+1][j], &A[i+1][i], -A[i][j], N-i-1);
 9                  }
10          }
11  }
```

Figure 7: The source code of the `ProcessDiagonalBlock()` function in C.

5.1 The `ProcessDiagonalBlock()` Function

The diagonal block A in a given repetition is processed by the function `ProcessDiagonalBlock()`. Let us assume that the diagonal block is of size $N \times N$. Recall that with the current implementation, the diagonal block always has the same length on both dimensions. The function uses each column i to update columns $i + 1$ to N. The code of the function is depicted in Figure 7. It consists of two nested loops, with the main computation being performed by the function `DAXPY()`. The general operation of the `DAXPY()` function is to update a vector with a second vector, the latter having been multiplied with a scalar value, i.e., $y \leftarrow \alpha \cdot x + y$, where x and y are vectors and α is a scalar. The code for the `DAXPY()` function is depicted in Figure 6.

In order to calculate the number of loads and stores performed in this code, we start with a single repetition of the inner loop in Figure 7 (lines 6-9). At Line 7 the algorithm has to load elements `A[i][j]` and `A[i][i]`, in order to divide them. An obvious optimization is to keep the new value of `A[i][j]` in a register, so that it can be used in the call to the `DAXPY()` function. We verified that the compiler in the current tool-chain for Cyclops-64, actually performs this optimization. Eventually, however, this value has to be written back to memory, hence a store instruction is required. Within the `DAXPY()` function, each element of the first vector and the corresponding element of the second vector have to be loaded into registers, so as to perform the required calculations on them. The scalar value, however, is already in a register, as previously mentioned. Therefore, no load instruction has to be issued for it. After all calculations are performed, the new value has to be stored in the first vector. For each repetition of the inner loop, `DAXPY()` has to process $N - i - 1$ elements (Line 8). As a result, the total load and store operations for one iteration of the inner loop are:

$$\begin{aligned}
&Loads\ per\ inner\ loop\ repetition = 2 + 2\cdot(N-i-1) = 2\cdot(N-i)\\
&Stores\ per\ inner\ loop\ repetition = 1 + (N-i-1) = (N-i)
\end{aligned}$$

However, this loop is executed $N-i-1$ times for each iteration of the outer loop. As a result, the total load and store operations for an iteration of the outer loop are:

$$\begin{aligned}
&Loads\ per\ outer\ loop\ repetition = 2\cdot(N-i)\cdot(N-i-1)\\
&Stores\ per\ outer\ loop\ repetition = (N-i)\cdot(N-i-1)
\end{aligned}$$

Finally, the total number of loads and stores in the function can be obtained, if we sum up the number of loads and stores per iteration of the outer loop. Notice that in the following sum, the upper bound for i is equal to $N-2$, since for $N-1$, the inner loop will not be executed:

$$\begin{aligned}
Loads &= \sum_{i=0}^{N-2} 2\cdot(N-i)\cdot(N-i-1) = 2\cdot\sum_{i=0}^{N-2}(N-i)\cdot(N-i-1)\\
Stores &= \sum_{i=0}^{N-2}(N-i)\cdot(N-i-1)
\end{aligned}$$

Hence, it is enough to find a closed form for the expression:

$$\begin{aligned}
&\sum_{i=0}^{N-2}(N-i)\cdot(N-i-1) = \sum_{i=0}^{N-2}(N^2 - i\cdot N - N - i\cdot N + i^2 + i) =\\
=&\sum_{i=0}^{N-2}\left[(N^2-N) - (2\cdot N-1)\cdot i + i^2\right] = \sum_{i=0}^{N-2}(N^2-N) - \sum_{i=0}^{N-2}(2\cdot N-1)\cdot i + \sum_{i=0}^{N-2} i^2 =\\
=&(N^2-N)\cdot(N-1) - (2\cdot N-1)\cdot\frac{(N-1)\cdot(N-2)}{2} + \frac{(N-1)\cdot(N-2)\cdot(2\cdot N-3)}{6} =\\
=&N\cdot(N-1)\cdot(N-1) - (N-1)\cdot(N-2)\cdot\left(\frac{2\cdot N-1}{2} - \frac{2\cdot N-3}{6}\right) =\\
=&N\cdot(N-1)\cdot(N-1) - (N-1)\cdot(N-2)\cdot\left(\frac{6\cdot N-3}{6} - \frac{2\cdot N-3}{6}\right) =\\
=&N\cdot(N-1)\cdot(N-1) - (N-1)\cdot(N-2)\cdot\left(\frac{4\cdot N}{6}\right) =\\
=&N\cdot(N-1)\cdot\left[(N-1) - \frac{2}{3}\cdot(N-2)\right] = N\cdot(N-1)\cdot\left[\frac{3\cdot N-3}{3} - \frac{2\cdot N-4}{3}\right] =\\
=&N\cdot(N-1)\cdot\frac{N+1}{3} =\\
=&\frac{N\cdot(N^2-1)}{3}
\end{aligned}$$

```
void ProcessBlockOnRow(double **A, double **D, long N, long M)
{
        long    i, j;

        for (i = 0; i < N; i++) {
                for (j = 0; j < M; j++) {
                        A[i][j] /= D[i][i];
                        DAXPY(&A[i+1][j], &D[i+1][i], -A[i][j], N-i-1);
                }
        }
}
```

Figure 8: The source code of the `ProcessBlockOnRow()` function in C.

Which gives the final result:

$$Loads = \frac{2 \cdot N \cdot (N^2 - 1)}{3}$$
$$Stores = \frac{N \cdot (N^2 - 1)}{3}$$

5.2 The `ProcessBlockOnRow()` Function

The function `ProcessBlockOnRow()` is used to handle blocks that share the same rows with the diagonal block. This function is similar to the `ProcessDiagonalBlock()` function, with the exception that each block might not have the same number of rows and columns. Figure 8 depicts the code for the function, for a block that has N rows and M columns. Variable A refers to the block being processed in this function and D refers to the diagonal block, which has previously been processed by `ProcessDiagonalBlock()`. Notice that the difference between the two functions are actually the boundaries of the inner loop.

Consequently, the number of loads and stores required in this function can be calculated in the same manner. However, each iteration of the outer loop executes now M iterations of the inner loop, instead of $N - i - 1$. As a result, the equations to calculate the number of loads and stores now become:

$$Loads = \sum_{i=0}^{N-1} 2 \cdot (N - i) \cdot M = 2 \cdot M \cdot \sum_{i=0}^{N-1} (N - i)$$
$$Stores = \sum_{i=0}^{N-1} (N - i) \cdot M = M \cdot \sum_{i=0}^{N-1} (N - i)$$

Notice that due to the different boundaries of the inner loop, the latter is executed even when i reaches the value $N - 1$, in contrast to the function `ProcessDiagonalBlock()`. Therefore, the upper limit in the above

```
 1  void ProcessBlockOnColumn(double **A, double **D, long N, long M)
 2  {
 3          long    i, j;
 4
 5          for (i = 0; i < M; i++) {
 6                  for (j = i + 1; j < M; j++) {
 7                          DAXPY(&A[0][j], &A[0][i], -D[i][j], N);
 8                  }
 9          }
10  }
```

Figure 9: The source code of the `ProcessBlockOnColumn()` function in C.

sums must be corrected accordingly. The above expression can be easily simplified:

$$\sum_{i=0}^{N-1}(N-i) = \sum_{i=0}^{N-1} N - \sum_{i=0}^{N-1} i = N^2 - \frac{N\cdot(N-1)}{2} = \frac{2\cdot N^2 - N^2 + N}{2} = \frac{N\cdot(N+1)}{2}$$

Which gives the final result:

$$Loads = \frac{2\cdot M\cdot N\cdot(N+1)}{2} = M\cdot N\cdot(N+1)$$
$$Stores = \frac{M\cdot N\cdot(N+1)}{2}$$

5.3 The `ProcessBlockOnColumn()` Function

The function `ProcessBlockOnColumn()` is used to handle blocks that share the same columns with the diagonal block. Similarly to the function `ProcessBlockOnRow()`, each block might not have the same number of rows and columns. Figure 9 depicts the code for the function, for a block that has N rows and M columns. Again, A refers to the block being processed in this function and D refers to the diagonal block.

Each column i is used to update columns $i+1$ to M. Two differences exist, with respect to the previous two functions. The first one is that no division between elements is required. This, however, affects the call to `DAXPY()`, since the third parameter (Line 7) must now be loaded from memory. The second difference is that the range of elements that the `DAXPY()` function operates on does not depend on the index of the outer loop.

In order to calculate the number of loads and stores for this function, we start again with one iteration of the inner loop. As can be seen in Line 7, it is necessary to load each element of the two vectors and the scalar value. Finally, the result of each computation has to be stored back into the first vector. As a result, the total load and store operations for one iteration of the inner loop are:

$$Loads\ per\ inner\ loop\ repetition = 2\cdot N + 1$$
$$Stores\ per\ inner\ loop\ repetition = N$$

However, the inner loop is executed $M-i-1$ times for each iteration of the outer loop. As a result, the total load and store operations for an iteration of the outer loop are:

$$\begin{aligned}
\mathit{Loads\ per\ outer\ loop\ repetition} &= (2\cdot N+1)\cdot(M-i-1)\\
\mathit{Stores\ per\ outer\ loop\ repetition} &= N\cdot(M-i-1)
\end{aligned}$$

Finally, the total number of loads and stores in the function can be obtained, if we sum up the number of loads and stores per iteration of the outer loop. Notice that in the following sum, the upper bound for i is equal to $M-2$, since for $M-1$, the inner loop will not be executed:

$$\begin{aligned}
\mathit{Loads} &= \sum_{i=0}^{M-2}(2\cdot N+1)\cdot(M-i-1) = (2\cdot N+1)\cdot\sum_{i=0}^{M-2}(M-i-1)\\
\mathit{Stores} &= \sum_{i=0}^{M-2} N\cdot(M-i-1) = N\cdot\sum_{i=0}^{M-2}(M-i-1)
\end{aligned}$$

Hence, it is enough to find a closed form for the expression:

$$\begin{aligned}
&\sum_{i=0}^{M-2}(M-i-1) = \sum_{i=0}^{M-2}(M-1) - \sum_{i=0}^{M-2} i = (M-1)^2 - \frac{(M-1)\cdot(M-2)}{2} =\\
=&(M-1)\cdot\left[(M-1)-\frac{M-2}{2}\right] = (M-1)\cdot\frac{2\cdot(M-1)-(M-2)}{2} =\\
=&\frac{M\cdot(M-1)}{2}
\end{aligned}$$

Which gives the final result:

$$\begin{aligned}
\mathit{Loads} &= \frac{M\cdot(2\cdot N+1)\cdot(M-1)}{2}\\
\mathit{Stores} &= \frac{M\cdot N\cdot(M-1)}{2}
\end{aligned}$$

5.4 The `ProcessInnerBlock()` Function

The last function that processes blocks of the matrix is `ProcessInnerBlock()`. It is used to process blocks that don't belong to the previous classes. These blocks share neither rows nor columns with the diagonal block. The code of the function is presented in Figure 10. Variable C refers to the block that must be processed by this function, which consists of N rows and M columns. Variable A refers to the block that shares the same columns with C and has previously been processed by `ProcessBlockOnRow()`. Similarly, B refers to the block that

```
1  void ProcessInnerBlock(double **A, double **B, double **C, long K, long N, long M)
2  {
3          long    i, j;
4
5          for (i = 0; i < K; i++) {
6                  for (j = 0; j < M; j++) {
7                          DAXPY(&C[0][j], &B[0][i], -A[i][j], N);
8                  }
9          }
10 }
```

Figure 10: The source code of the `ProcessInnerBlock()` function in C.

shares the same rows with C and has previously been processed by `ProcessBlockOnColumn()`. The size of the diagonal block is given by K. This value is obviously equal to the number of rows for A and the number of columns for B.

As with all previous cases, we will calculate the required number of loads and stores for this code, starting with one iteration of the innermost loop. As can be seen in Line 7, it is necessary to load one column of block C, one column of block B and one scalar value. The result is stored back in the same column of block C. As a result, the total load and store operations for one iteration of the inner loop are:

$$\begin{aligned} Loads\ per\ inner\ loop\ repetition &= 2 \cdot N + 1 \\ Stores\ per\ inner\ loop\ repetition &= N \end{aligned}$$

However, the inner loop is executed M times for each iteration of the outer loop. As a result, the total load and store operations for an iteration of the outer loop are:

$$\begin{aligned} Loads\ per\ outer\ loop\ repetition &= (2 \cdot N + 1) \cdot M \\ Stores\ per\ outer\ loop\ repetition &= N \cdot M \end{aligned}$$

Obviously, the above result does not depend at all on the loop variable of the outer loop. As a result, the total number of loads and stores in the function are:

$$\begin{aligned} Loads &= (2 \cdot N + 1) \cdot M \cdot K \\ Stores &= N \cdot M \cdot K \end{aligned}$$

5.5 Validating the Results

In order to verify the previous results, we ran the LU benchmark on the Cyclops-64 simulator with different combinations of matrix sizes and number of thread units. During these experiments, we exploited the `StatsOn` and

`StatsOff` macros, which are provided by the tool-chain for Cyclops-64. These macros enable the programmer to collect statistics only in the parts of a program that are of immediate interest. Among the statistics that are collected, the number of loads and stores in the designated part of the program are included.

Initially, we added `StatsOn` to the beginning of `ProcessDiagonalBlock()` and `StatsOff` to the end of the same function. For each combination of matrix size and number of thread units, we executed the benchmark on the Cyclops-64 simulator, but allowed it to complete only the first step of the algorithm. We noted the size of the diagonal block and the number of loads and stores, as reported by the simulator. Subsequently, we executed the benchmark again with the same parameters, allowing it this time to complete two steps. As a result, the reported number of loads and stores were the sum of the first two steps, for the same function. Subtracting the numbers from the first step gives us the number of loads and stores for the second step. We repeated the procedure, this time allowing the benchmark to complete three steps. Subtracting the corresponding numbers from the previous runs, gives us the number of loads and stores for the third step.

We repeated the procedure for all other functions of the algorithm, i.e., `ProcessBlockOnColumn()`, `ProcessBlockOnRow()` and `ProcessInnerBlock()`. In all cases, the theoretically calculated number of loads and stores exactly matches the measured ones. As an example, we include Table 1 to Table 4. These measurements were obtained using 16 thread units and a matrix with a size of 512×512. This verifies that our theoretical approach is valid and accurate. In these tables, the column labeled "Thread Unit" corresponds to a range of thread units to which blocks of the same size have been assigned to, as required by our dynamic repartitioning algorithm . Columns labeled "M/N" and "K/M/N" provide the exact size of the blocks that have been assigned to these thread units. We remind the reader that each block can have a difference of at most one on each dimension with every other block.

		Theoretical		Measured	
Repetition	N	Loads	Stores	Loads	Stores
1	56	117040	58520	117040	58520
2	50	83300	41650	83300	41650
3	45	60720	30360	60720	30360

Table 1: The number of loads and stores in `ProcessDiagonalBlock()`.

			Theoretical		Measured	
Repetition	Thread Unit	M/N	Loads	Stores	Loads	Stores
1	8-15	57/56	181944	90972	181944	90972
2	8-13	51/50	130050	65025	130050	65025
	14-15	50/50	127500	63750	127500	63750
3	8	46/45	95220	47610	95220	47610
	9-15	45/45	93150	46575	93150	46575

Table 2: The number of loads and stores in `ProcessBlockOnRow()`.

			Theoretical		Measured	
Repetition	Thread Unit	M/N	Loads	Stores	Loads	Stores
1	0-7	56/57	177100	87780	177100	87780
2	0-5	50/51	126175	62475	126175	62475
	6-7	50/50	123725	61250	123725	61250
3	0	45/46	92070	45540	92070	45540
	1-7	45/45	90090	44550	90090	44550

Table 3: The number of loads and stores in `ProcessBlockOnColumn()`.

			Theoretical		Measured	
Repetition	Thread Unit	K/M/N	Loads	Stores	Loads	Stores
1	0-15	56/57/228	1458744	727776	1458744	727776
2	0-5	50/51/204	1042950	520200	1042950	520200
	6-7	50/50/204	1022500	510000	1022500	510000
	8-13	50/51/202	1032750	515100	1032750	515100
	14-15	50/50/202	1012500	505000	1012500	505000
3	0	45/46/181	751410	374670	751410	374670
	1-7	45/45/181	735075	366525	735075	366525
	8	45/46/180	747270	372600	747270	372600
	9-15	45/45/180	731025	364500	731025	364500

Table 4: The number of loads and stores in `ProcessInnerBlock()`.

6 Minimizing the Number of Memory Operations in the Dynamic Block-LU Algorithm

Having the promising results of the previous paragraphs for the high-level algorithm, we are ready to continue optimizing our application at the next level. As previously mentioned, we still use the `DAXPY()` BLAS-1 routine to update the elements within each block. In this section we propose an alternative way to perform the same operations, since the above routine does not map well to our architecture. Since there is no data cache to speed up the access to required elements of the matrix, we will have to rely on the next level of high-speed storage, which is the register file in each TU. Each TU has a total of 64 registers, but some of them cannot be used in user-level applications, like R3 (Stack Pointer) and R0 (Permanent Zero). Moreover, parameters to functions are passed through registers and are required throughout the function, for example the block that has to be processed and it's dimensions. Finally some registers are required as loop indices in our code. After optimizing usage of these registers we are left with only 48 registers for loading data. However, this number proves to be crucial for our implementation.

In order to use such a limited number of registers, each block will have to be further subdivided into smaller blocks. This widely used technique is known as *register tiling* [3, 4, 15] and applying it in the case of a matrix multiplication for C64 yielded excellent results [12]. However, 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 order to overcome this shortcoming in current compiler technology and get the highest possible performance out of this technique, we decided to manually apply register tiling in an application-aware manner. More specifically, after loading the elements of a sub-block into registers, we examined how it is possible to perform the largest number of floating point operations, before writing them back into memory. Since updating these elements depends on elements from other blocks, we have to take into account the fact that we also need registers for these elements. The advantage that we have over an automated, compiler-driven analysis, is that we can understand in a more complete way the data dependencies between these elements and further optimize our code. We believe that making compilers aware of an application's behavior at this level is a challenge to be met and that results will be very important for the architectures under consideration. One more advantage we have over current compiler technology, is that we can exactly calculate the number of loads and stores required in each case, which makes it possible to determine the case that minimizes memory operations. Finally, taking into account the number of available registers, it is possible to determine the optimal size for each sub-block. We realize, however, that applying this methodology manually to each application is a time consuming and error-prone procedure, which is why we already commented on the features that we would like to see in future compilers.

In the following paragraphs, we identify possible ways to subdivide each block. Although we did our best, we do not argue that this is an exhaustive list of possibilities. However, among the ones we identified, we mathematically prove which is the best one, with respect to the number of loads and stores that are required. We used the optimal cases in our implementation and, as we will see through our experiments, the whole process was worth it.

For the rest of this section, we will assume that the size of a block or sub-block can always be exactly divided with the size of another block or sub-block, i.e., we will not take into account any remainders. This has no effect on the general results of our analysis and makes it simpler to follow.

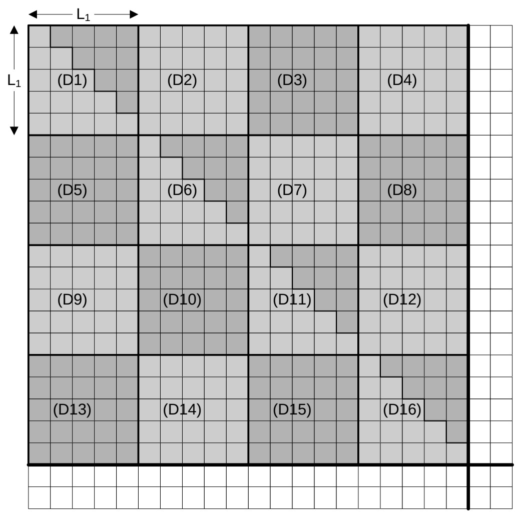

Figure 11: Dividing into sub-blocks for `ProcessDiagonalBlock()`.

6.1 Optimizing the `ProcessDiagonalBlock()` Function

The `ProcessDiagonalBlock()` function operates at each step of our algorithm on the first block of data. In order to update the elements of the block, it exclusively uses elements from the same block. As mentioned in Section 5.1, each block is assumed to have N rows and columns. Each column i of the block is used to update all subsequent columns, i.e., columns $i+1$ to N. For this function, we identified only one possible way to apply register tiling. We assume that the block has been divided into sub-blocks of size $L_1 \times L_1$. Figure 11 shows an example of dividing a block into sub-blocks, which we will use throughout this paragraph. From the same figure, it should also be clear why we decided to divide the block into sub-blocks that have the same size on both dimensions. As can be seen, the sub-blocks that are now on the diagonal have to update elements differently, depending on whether they are on the upper or lower triangular part of the whole block (see Figure 7). Choosing the same size for both dimensions of each sub-block makes them symmetrical in this aspect and the whole process is far easier to be implemented.

We start with the first sub-block of size $L_1 \times L_1$ and update it. This corresponds to sub-block `(D1)` in Figure 11. Then, we load successively each sub-block of size $L_1 \times L_1$ that is to the right of the first sub-block and make the necessary updates. Using again Figure 11, while we still have `(D1)` in registers, we consecutively load `(D2)`, `(D3)` and `(D4)` and update them. Then we move to the next row of sub-blocks to update `(D5)`. We load `(D1)` and `(D5)` and update the latter. While still having `(D5)` in registers, we load `(D2)` and `(D6)` to update `(D6)`, `(D3)` and `(D7)` to update `(D7)` and `(D4)` and `(D8)` to update `(D8)`. Then we load `(D6)` into registers, update it and also use it to update `(D7)` and `(D8)`. With this we finished updating the second row of sub-blocks and can move to the third one. We load `(D1)` and `(D9)` and update the latter. While still having `(D9)` in registers, we load `(D2)` and `(D10)` to update `(D10)`, `(D3)` and `(D11)` to update `(D11)` and `(D4)` and `(D12)` to update `(D12)`. Then we load `(D6)` and `(D10)` and update the latter. While still having `(D10)` in registers, we load `(D7)` and `(D11)` and update `(D11)` and `(D8)` and `(D12)` to update `(D12)`. Finally, we load `(D11)` into registers, update it and also use it to update `(D12)`. Finally, we move to the fourth row of sub-blocks. We load `(D1)` and `(D13)` and update the latter. While still having `(D13)` in registers, we load `(D2)` and `(D14)` to update `(D14)`, `(D3)` and `(D15)` to update `(D15)` and `(D4)` and `(D16)` to update `(D16)`. Then we load `(D6)` and `(D14)` and update the latter. While still having `(D14)` in registers, we load `(D7)` and `(D15)` and update `(D15)` and `(D8)` and `(D16)` to update `(D16)`. Then we load `(D11)` and `(D15)` and update the latter. While still having `(D15)` in registers, we load `(D12)` and `(D16)` and update `(D16)`. Finally, we load `(D16)` and update it.

In order to calculate the number of loads and stores required, we start with the first row of sub-blocks. We load `(D1)` into registers, update it and also use it to update `(D2)`, `(D3)` and `(D4)`:

$$\begin{aligned} Loads &= \frac{N}{L_1} \cdot L_1^2 \\ Stores &= \frac{N}{L_1} \cdot L_1^2 = N \cdot L_1 \end{aligned}$$

On the second row of sub-blocks, we load `(D1)` and `(D5)` to update the latter and then we load `(D2)` and `(D6)` to update `(D6)`, `(D3)` and `(D7)` to update `(D7)` and `(D4)` and `(D8)` to update `(D8)`. Then we load `(D6)` into registers, update it and also use it to update `(D7)` and `(D8)`:

$$\begin{aligned} Loads &= \frac{N}{L_1} \cdot 2 \cdot L_1^2 + \left(\frac{N}{L_1} - 1\right) \cdot L_1^2 \\ Stores &= 2 \cdot L_1^2 + \left(2 \cdot \frac{N}{L_1} - 3\right) \cdot L_1^2 = 2 \cdot \frac{N}{L_1} \cdot L_1^2 - L_1^2 \end{aligned}$$

On the third row of sub-blocks, we load `(D1)` and `(D9)` to update the latter and then we load `(D2)` and `(D10)` to update `(D10)`, `(D3)` and `(D11)` to update `(D11)` and `(D4)` and `(D12)` to update `(D12)`. Then we load `(D6)` and `(D10)` and update the latter. While still having `(D10)` in registers, we load `(D7)` and `(D11)` and update `(D11)` and `(D8)` and `(D12)` to update `(D12)`. Finally, we load `(D11)` into registers, update it and also use it to update `(D12)`:

$$\begin{aligned} Loads &= \frac{N}{L_1} \cdot 2 \cdot L_1^2 + \left(\frac{N}{L_1} - 1\right) \cdot 2 \cdot L_1^2 + \left(\frac{N}{L_1} - 2\right) \cdot L_1^2 = 2 \cdot \frac{N}{L_1} \cdot 2 \cdot L_1^2 - 2 \cdot L_1^2 + \left(\frac{N}{L_1} - 2\right) \cdot L_1^2 \\ Stores &= 3 \cdot L_1^2 + \left(3 \cdot \frac{N}{L_1} - 6\right) \cdot L_1^2 = 3 \cdot \frac{N}{L_1} \cdot L_1^2 - 3 \cdot L_1^2 \end{aligned}$$

Following the same procedure for all rows of sub-blocks, leads us to the more general equations:

$$
\begin{aligned}
Loads &= \sum_{i=0}^{\frac{N}{L_1}-1} \left[2 \cdot L_1^2 \cdot \frac{N}{L_1} \cdot i - \sum_{j=1}^{i} \left[2 \cdot L_1^2 \cdot (j-1) \right] + \left(\frac{N}{L_1} - i \right) \cdot L_1^2 \right] = \\
&= \sum_{i=0}^{\frac{N}{L_1}-1} \left[2 \cdot N \cdot L_1 \cdot i + \sum_{j=1}^{i} (2 \cdot L_1^2) - \sum_{j=1}^{i} (2 \cdot L_1^2 \cdot j) + N \cdot L_1 - i \cdot L_1^2 \right] = \\
&= \sum_{i=0}^{\frac{N}{L_1}-1} \left[2 \cdot N \cdot L_1 \cdot i + 2 \cdot L_1^2 \cdot i - 2 \cdot L_1^2 \cdot \frac{i \cdot (i+1)}{2} + N \cdot L_1 - i \cdot L_1^2 \right] = \\
&= \sum_{i=0}^{\frac{N}{L_1}-1} \left[2 \cdot N \cdot L_1 \cdot i + N \cdot L_1 - L_1^2 \cdot i^2 \right] = \\
&= 2 \cdot N \cdot L_1 \cdot \frac{\frac{N}{L_1} \cdot (\frac{N}{L_1} - 1)}{2} + N \cdot L_1 \cdot \frac{N}{L_1} - L_1^2 \cdot \frac{\frac{N}{L_1} \cdot (\frac{N}{L_1} - 1) \cdot (2 \cdot \frac{N}{L_1} - 1)}{6} = \\
&= \frac{N^2 \cdot (N - L_1)}{L_1} + N^2 - \frac{N \cdot (N - L_1) \cdot (2 \cdot N - L_1)}{6 \cdot L_1} = \\
&= \frac{N \cdot (4 \cdot N - L_1) \cdot (N + L_1)}{6 \cdot L_1}
\end{aligned}
$$

$$
\begin{aligned}
Stores &= \sum_{i=1}^{\frac{N}{L_1}} \left[N \cdot L_1 \cdot i - \sum_{j=1}^{i-1} j \cdot L_1^2 \right] = \sum_{i=1}^{\frac{N}{L_1}} \left[N \cdot L_1 \cdot i - L_1^2 \cdot \frac{i \cdot (i-1)}{2} \right] = \\
&= \sum_{i=1}^{\frac{N}{L_1}} \left[N \cdot L_1 \cdot i - \frac{L_1^2}{2} \cdot i^2 + \frac{L_1^2}{2} \cdot i \right] = \sum_{i=1}^{\frac{N}{L_1}} \left[\left(N \cdot L_1 + \frac{L_1^2}{2} \right) \cdot i - \frac{L_1^2}{2} \cdot i^2 \right] = \\
&= \left(N \cdot L_1 + \frac{L_1^2}{2} \right) \cdot \frac{\frac{N}{L_1} \cdot (\frac{N}{L_1} + 1)}{2} - \frac{L_1^2}{2} \cdot \frac{\frac{N}{L_1} \cdot (\frac{N}{L_1} + 1) \cdot (2 \cdot \frac{N}{L_1} + 1)}{6} = \\
&= \left(N + \frac{L_1}{2} \right) \cdot \frac{N \cdot (N + L_1)}{2 \cdot L_1} - \frac{N \cdot (N + L_1) \cdot (2 \cdot N + L_1)}{12 \cdot L_1} = \\
&= \frac{N \cdot (N + L_1) \cdot (2 \cdot N + L_1)}{3 \cdot L_1}
\end{aligned}
$$

In order to minimize the number of loads and stores, we have to calculate the parameter L_1. For that, we have to take into account the fact that the number of available registers in our architecture is fixed. If we assume that we have R registers available, then at any point three $L_1 \times L_1$ sub-blocks have to fit into the available registers, which means that:

$$3 \cdot L_1^2 = R \Rightarrow L_1 = \sqrt{\frac{R}{3}}$$

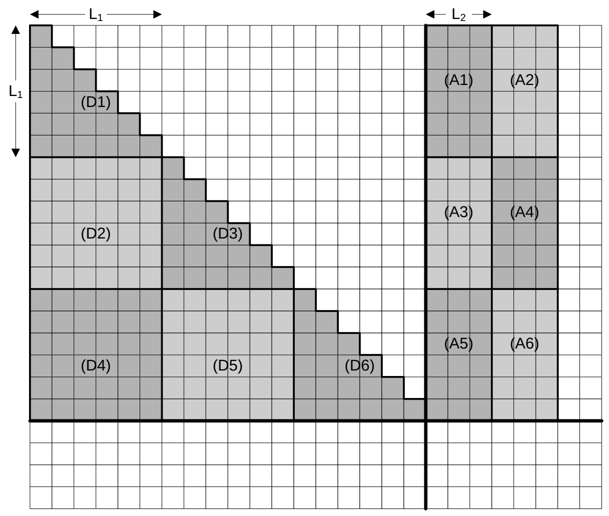

Figure 12: Dividing into sub-blocks for `ProcessBlockOnRow()`.

For our architecture we can use $R = 48$. As a result, $L_1 = 4$ and the final equations for the number of loads and stores become:

$$Loads = \frac{N \cdot (N-1) \cdot (N+4)}{6}$$
$$Stores = \frac{N \cdot (N+2) \cdot (N+4)}{6}$$

6.2 Optimizing the `ProcessBlockOnRow()` Function

The `ProcessBlockOnRow()` function operates on blocks of data that share the same rows with the diagonal block. In order to update these elements, it also requires data from the diagonal block. As mentioned in Section 5.2, each block is assumed to have N rows and M columns. Each column from the diagonal block is used to update all columns in the block that is being processed. For this function, we identified four possible ways to apply register tiling. We assume that each block that has to be processed by the function has been divided into sub-blocks of size $L_1 \times L_2$. Due to data dependencies (see Figure 8), this means that the required data in the diagonal block has to be divided into sub-blocks of size $L_1 \times L_1$. Moreover, it is worth to notice that not all the data from the diagonal block is required, but only the data from the lower triangular part. Figure 12 shows an example of dividing a block into sub-blocks, which we will use throughout this paragraph. For the readers who would like to skip the detailed analysis, we mention that the optimal method to load sub-blocks proves to be the last one, which we also used in our implementation.

Case 1: In the first case, we start with the first sub-block of size $L_1 \times L_2$ and update it, using the corresponding triangular-shaped data sub-block from the diagonal block. These correspond to sub-blocks `(A1)` and `(D1)` in Figure 12. Then, we load successively each sub-block of size $L_1 \times L_2$ that is to the bottom of the first sub-block and the corresponding sub-block of size $L_1 \times L_1$ from the diagonal block and make the necessary updates. Using again Figure 12, while we still have `(A1)` in registers, we load `(A3)` and `(D2)` and update `(A3)`. Then we load `(A5)` and `(D4)` and update `(A5)`. Moving on, we load `(A3)` and `(D3)` and update `(A3)`. While we still have `(A3)` in registers, we load `(A5)` and `(D5)` and update `(A5)`. Finally, we load `(A5)` and `(D6)` and update `(A5)`. This completes the first column of sub-blocks and we can move to the next column, containing sub-blocks `(A2)`, `(A4)` and `(A6)`, in order to repeat the process.

Before calculating the total number of loads and stores required for this case, we have to calculate the number of loads required for the triangular sub-blocks of the diagonal block, i.e., blocks `(D1)`, `(D3)` and `(D6)` in the example of Figure 12.

$$Loads\ in\ triangular\ sub\text{-}block = 1 + 2 + 3 + \ldots + L_1 = \frac{L_1 \cdot (L_1 + 1)}{2}$$

Having this result, we can move on to calculate the total number of loads and stores required. As mentioned earlier, we start by updating sub-block `(A1)` and then using this sub-block, we update `(A3)` and `(A5)`:

$$Loads = \left[L_1 \cdot L_2 + \frac{L_1 \cdot (L_1 + 1)}{2}\right] + \left[2 \cdot L_1 \cdot L_2 + 2 \cdot L_1^2\right]$$
$$Stores = 3 \cdot L_1 \cdot L_2$$

Then we update sub-block `(A3)` and then using this sub-block, we update `(A5)`:

$$Loads = \left[L_1 \cdot L_2 + \frac{L_1 \cdot (L_1 + 1)}{2}\right] + \left[L_1 \cdot L_2 + L_1^2\right]$$
$$Stores = 2 \cdot L_1 \cdot L_2$$

Finally, we update sub-block `(A5)`:

$$Loads = L_1 \cdot L_2 + \frac{L_1 \cdot (L_1 + 1)}{2}$$
$$Stores = L_1 \cdot L_2$$

Which leads us to the more general equations for the total number of loads and stores per column of sub-blocks:

$$Loads = \sum_{i=1}^{\frac{N}{L_1}} \left[i \cdot L_1 \cdot L_2 + (i-1) \cdot L_1^2 + \frac{L_1 \cdot (L_1 + 1)}{2}\right]$$
$$Stores = \sum_{i=1}^{\frac{N}{L_1}} i \cdot L_1 \cdot L_2$$

However, the above procedure has to be repeated for every column of sub-blocks, which gives the final equations:

$$\begin{aligned}
Loads &= \frac{M}{L_2} \cdot \sum_{i=1}^{\frac{N}{L_1}} \left[i \cdot L_1 \cdot L_2 + (i-1) \cdot L_1^2 + \frac{L_1 \cdot (L_1+1)}{2} \right] = \\
&= \frac{M}{L_2} \cdot \sum_{i=1}^{\frac{N}{L_1}} \left[i \cdot (L_1 \cdot L_2 + L_1^2) - L_1^2 + \frac{L_1 \cdot (L_1+1)}{2} \right] = \\
&= \frac{M}{L_2} \cdot \sum_{i=1}^{\frac{N}{L_1}} \left[i \cdot (L_1 \cdot L_2 + L_1^2) - \frac{L_1 \cdot (L_1-1)}{2} \right] = \\
&= \frac{M}{L_2} \cdot (L_1 \cdot L_2 + L_1^2) \cdot \sum_{i=1}^{\frac{N}{L_1}} i - \frac{M}{L_2} \cdot \frac{N}{L_1} \cdot \frac{L_1 \cdot (L_1-1)}{2} = \\
&= \frac{M}{L_2} \cdot L_1 \cdot (L_1 + L_2) \cdot \frac{\frac{N}{L_1} \cdot (\frac{N}{L_1}+1)}{2} - \frac{M \cdot N \cdot L_1 \cdot (L_1-1)}{2 \cdot L_1 \cdot L_2} = \\
&= \frac{M}{L_2} \cdot (L_1 + L_2) \cdot \frac{N \cdot (N + L_1)}{2 \cdot L_1} - \frac{M \cdot N \cdot L_1 \cdot (L_1-1)}{2 \cdot L_1 \cdot L_2} = \\
&= \frac{M \cdot N \cdot (L_1 + L_2) \cdot (N + L_1) - M \cdot N \cdot L_1 \cdot (L_1 - 1)}{2 \cdot L_1 \cdot L_2} = \\
&= \frac{M \cdot N \cdot (N \cdot L_1 + N \cdot L_2 + L_1 \cdot L_2 + L_1)}{2 \cdot L_1 \cdot L_2}
\end{aligned}$$

$$\begin{aligned}
Stores &= \frac{M}{L_2} \cdot \sum_{i=1}^{\frac{N}{L_1}} i \cdot L_1 \cdot L_2 = \frac{M}{L_2} \cdot L_1 \cdot L_2 \cdot \sum_{i=1}^{\frac{N}{L_1}} i = M \cdot L_1 \cdot \frac{\frac{N}{L_1} \cdot (\frac{N}{L_1}+1)}{2} = \\
&= \frac{M \cdot N \cdot (N + L_1)}{2 \cdot L_1}
\end{aligned}$$

In order to minimize the number of loads and stores, we have to calculate the parameters L_1 and L_2. For that, we have to take into account the fact that the number of available registers in our architecture is fixed. If we assume that we have R registers available, then at any point one $L_1 \times L_1$ sub-block from the diagonal block and two $L_2 \times L_1$ sub-blocks from the block that is being updated have to fit into the available registers, which means that:

$$2 \cdot L_1 \cdot L_2 + L_1^2 = R \Rightarrow L_2 = \frac{R - L_1^2}{2 \cdot L_1}$$

By replacing L_2 in the equation for the total number of loads and setting the derivative of that equation equal 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:

$$\frac{dLoads}{dL_1} = 0 \Rightarrow (N+2) \cdot L_1^4 + (4 \cdot N \cdot R + 2 \cdot R) \cdot L_1^2 - N \cdot R^2 = 0 \Rightarrow$$
$$L_1^2 = R \cdot \frac{-(2 \cdot N + 1) \pm \sqrt{5 \cdot N^2 + 6 \cdot N + 1}}{N+2}$$

By letting N move from a small value $(N = 1)$ to infinity in the above equation, we can conclude that L_1^2 should move between the values:

$$0.154700 \cdot R \leq L_1^2 \leq 0.236068 \cdot R$$

Since R in Cyclops-64 can have a value of up to 48, we conclude that:

$$7.43 \leq L_1^2 \leq 11.33$$

The only integer value for L_1 that gives a value in the above range is $L_1 = 3$. In this case, however, the largest R that we can use is 45, because this gives an integer value for L_2 of 6. This gives the final values for the total number of loads and stores:

$$Loads = \frac{M \cdot N \cdot (3 \cdot N + 7)}{12}$$
$$Stores = \frac{M \cdot N \cdot (N + 3)}{6}$$

Case 2: The second case can be seen as the reverse of the first one. Keeping one sub-block of size $L_1 \times L_2$ constant, we successively load every sub-block that is on top of it, together with the corresponding sub-block of size $L_1 \times L_1$ from the diagonal block. Referring again to Figure 12, we start with sub-blocks `(A1)` and `(D1)` and update the first. Then we load sub-block `(A3)` and keep it constantly in registers. We also load `(A1)` and `(D2)` and update `(A3)`. Finally, we load `(D3)` and update again `(A3)`. Now we finished updating `(A3)` and we load `(A5)` into registers and keep it constantly there. Now we load `(A1)` and `(D4)` and update `(A5)`. Then we load `(A3)` and `(D5)` and update again `(A5)`. Finally, we load `(D6)` and update `(A5)`. This completes the first column of sub-blocks and we can move to the next column.

In order to calculate the total number of loads and stores in this case, we start with the update of sub-block `(A1)`:

$$Loads = L_1 \cdot L_2 + \frac{L_1 \cdot (L_1 + 1)}{2}$$
$$Stores = L_1 \cdot L_2$$

In order to update sub-block `(A3)`, we require sub-blocks `(A1)`, `(A3)`, `(D2)` and `(D3)`:

$$\begin{aligned}
Loads &= \left[L_1 \cdot L_2 + \frac{L_1 \cdot (L_1 + 1)}{2}\right] + \left[L_1 \cdot L_2 + L_1^2\right] \\
Stores &= 2 \cdot L_1 \cdot L_2
\end{aligned}$$

Finally, in order to update sub-block `(A5)`, we require sub-blocks `(A1)`, `(A3)`, `(A5)`, `(D4)`, `(D5)` and `(D6)`:

$$\begin{aligned}
Loads &= \left[L_1 \cdot L_2 + \frac{L_1 \cdot (L_1 + 1)}{2}\right] + \left[2 \cdot L_1 \cdot L_2 + 2 \cdot L_1^2\right] \\
Stores &= 3 \cdot L_1 \cdot L_2
\end{aligned}$$

Which leads us to the more general equations for the total number of loads and stores per column of sub-blocks:

$$\begin{aligned}
Loads &= \sum_{i=1}^{\frac{N}{L_1}} \left[i \cdot L_1 \cdot L_2 + (i-1) \cdot L_1^2 + \frac{L_1 \cdot (L_1 + 1)}{2}\right] \\
Stores &= \sum_{i=1}^{\frac{N}{L_1}} i \cdot L_1 \cdot L_2
\end{aligned}$$

However, the above procedure has to be repeated for every column of sub-blocks, which gives the final equations:

$$Loads = \frac{M}{L_2} \cdot \sum_{i=1}^{\frac{N}{L_1}} \left[i \cdot L_1 \cdot L_2 + (i-1) \cdot L_1^2 + \frac{L_1 \cdot (L_1 + 1)}{2}\right]$$

$$Stores = \frac{M}{L_2} \cdot \sum_{i=1}^{\frac{N}{L_1}} i \cdot L_1 \cdot L_2$$

Obviously, the above equations are exactly the same with the ones from the previous case, which means that the optimum number of loads and stores in this case is again:

$$\begin{aligned}
Loads &= \frac{M \cdot N \cdot (3 \cdot N + 7)}{12} \\
Stores &= \frac{M \cdot N \cdot (N + 3)}{6}
\end{aligned}$$

Case 3: The third case is different from the two previous ones. After updating each sub-block of size $L_1 \times L_2$ we move to the sub-block to the right. This leads to reusing data from the diagonal block, whereas in the previous cases, we were reusing data from the block that is being updated. Using again the example of Figure 12, we start by loading sub-blocks `(A1)` and `(D1)` and update `(A1)`. Keeping the elements from `(D1)`, we load `(A2)` and update it. We continue until we reach the end of the block. Then we move to the next row of sub-blocks. We load `(D2)` and keep it constantly in registers. We also load `(A1)` and `(A3)` and update `(A3)`. Then we load `(A2)` and `(A4)` and update `(A4)`. Again, we continue until the end of the block and then move to the next row of sub-blocks. We load `(D4)` and keep it constantly in registers. We also load `(A1)` and `(A5)` and update `(A5)`. Then we load `(A2)` and `(A6)` and update `(A6)`. After we finish with the whole row of sub-blocks, we load `(D3)` and keep it constantly in registers. We load `(A3)` and update it, then `(A4)` and update it and we continue to the end of the row of sub-blocks. Then we load `(D5)` and keep it constantly in registers. We also load `(A3)` and `(A5)` and update `(A5)`. Then we load `(A4)` and `(A6)` and update `(A6)`. Again, we continue until we finish with the whole row of sub-blocks. Finally, we load `(D6)` and keep it constantly in registers. Then we load subsequently `(A5)` and update it, then `(A6)` and update it, until we reach again the end of the block.

In order to calculate the total number of loads and stores in this case, we start with the updates in the first row of sub-blocks. In the example of Figure 12 we need to load `(D1)` and then successively all of the M/L_2 sub-blocks that have to be updated in the column, i.e., `(A1)`, `(A2)`, etc. This leads to the equations:

$$
\begin{aligned}
Loads &= L_1 \cdot L_2 \cdot \frac{M}{L_2} + \frac{L_1 \cdot (L_1 + 1)}{2} \\
Stores &= \frac{M}{L_2} \cdot L_1 \cdot L_2
\end{aligned}
$$

After that, we load `(D2)`, `(A1)` and `(A3)`, in order to update `(A3)`. We repeat the whole process for the whole row of sub-blocks to the right of `(A3)`. We also repeat the process for `(D4)` and sub-block `(A5)` and all sub-blocks to the right of it. Combined with the above result, this gives a total of:

$$
\begin{aligned}
Loads &= 2 \cdot \left[L_1^2 + (L_1 \cdot L_2 + L_1 \cdot L_2) \cdot \frac{M}{L_2} \right] + L_1 \cdot L_2 \cdot \frac{M}{L_2} + \frac{L_1 \cdot (L_1 + 1)}{2} \\
Stores &= 2 \cdot \frac{M}{L_2} \cdot L_1 \cdot L_2 + \frac{M}{L_2} \cdot L_1 \cdot L_2 = 3 \cdot \frac{M}{L_2} \cdot L_1 \cdot L_2
\end{aligned}
$$

Subsequently, we repeat the whole process starting with `(D3)` and updating `(A3)` and all blocks that are to the right it. This time, however, we have to update only one row of sub-blocks more, by loading `(D4)` and updating `(A5)` and all sub-blocks to the right it. Hence:

$$
\begin{aligned}
Loads &= \left[L_1^2 + (L_1 \cdot L_2 + L_1 \cdot L_2) \cdot \frac{M}{L_2} \right] + L_1 \cdot L_2 \cdot \frac{M}{L_2} + \frac{L_1 \cdot (L_1 + 1)}{2} \\
Stores &= \frac{M}{L_2} \cdot L_1 \cdot L_2 + \frac{M}{L_2} \cdot L_1 \cdot L_2 = 2 \cdot \frac{M}{L_2} \cdot L_1 \cdot L_2
\end{aligned}
$$

Finally, we have to load `(D6)` and update `(A5)` and all sub-blocks to the right of it:

$$\begin{aligned}
Loads &= L_1 \cdot L_2 \cdot \frac{M}{L_2} + \frac{L_1 \cdot (L_1 + 1)}{2} \\
Stores &= \frac{M}{L_2} \cdot L_1 \cdot L_2
\end{aligned}$$

By adding all previous equations and generalizing, we conclude that the total number of loads and stores is in this case:

$$\begin{aligned}
Loads &= \sum_{i=1}^{\frac{N}{L_1}} \left[(i-1) \cdot \left[L_1^2 + (L_1 \cdot L_2 + L_1 \cdot L_2) \cdot \frac{M}{L_2} \right] + L_1 \cdot L_2 \cdot \frac{M}{L_2} + \frac{L_1 \cdot (L_1 + 1)}{2} \right] = \\
&= \sum_{i=1}^{\frac{N}{L_1}} \left[(i-1) \cdot (L_1^2 + 2 \cdot L_1 \cdot M) + M \cdot L_1 + \frac{L_1 \cdot (L_1 + 1)}{2} \right] = \\
&= (L_1^2 + 2 \cdot L_1 \cdot M) \cdot \frac{\frac{N}{L_1} \cdot (\frac{N}{L_1} - 1)}{2} + \frac{N}{L_1} \cdot M \cdot L_1 + \frac{N}{L_1} \cdot \frac{L_1 \cdot (L_1 + 1)}{2} = \\
&= (L_1 + 2 \cdot M) \cdot \frac{N \cdot (N - L_1)}{2 \cdot L_1} + \frac{2 \cdot M \cdot N \cdot L_1}{2 \cdot L_1} + \frac{N \cdot L_1 \cdot (L_1 + 1)}{2 \cdot L_1} = \\
&= \frac{N^2 \cdot L_1 + 2 \cdot M \cdot N^2 + N \cdot L_1}{2 \cdot L_1} = \\
&= \frac{M \cdot N^2}{L_1} + \frac{N \cdot (N + 1)}{2}
\end{aligned}$$

$$\begin{aligned}
Stores &= \sum_{i=1}^{\frac{N}{L_1}} i \cdot L_1 \cdot L_2 \cdot \frac{M}{L_2} = M \cdot L_1 \cdot \sum_{i=1}^{\frac{N}{L_1}} i = M \cdot L_1 \cdot \frac{\frac{N}{L_1} \cdot (\frac{N}{L_1} + 1)}{2} = \\
&= \frac{M \cdot N \cdot (N + L_1)}{2 \cdot L_1}
\end{aligned}$$

As can be seen, the above results do not depend on L_2 at all and it is also obvious that the bigger L_1 is, the less the total number of loads and stores. Hence, we can set $L_2 = 1$. If we further assume that the number of available registers in our architecture is R, then at each point in time, the maximum number of elements loaded into registers cannot be more than R:

$$2 \cdot L_1 \cdot L_2 + L_1^2 = R \Rightarrow 2 \cdot L_1 \cdot 1 + L_1^2 = R \Rightarrow L_1^2 + 2 \cdot L_1 - R = 0 \Rightarrow L_1 = -1 \pm \sqrt{R + 1}$$

As mentioned earlier, $R = 48$ for our architecture, which will give us a value of $L_1 = 6$. As a result, the optimum number of loads and stores in this case is:

$$Loads = \frac{M \cdot N^2}{6} + \frac{N \cdot (N+1)}{2}$$
$$Stores = \frac{M \cdot N \cdot (N+6)}{12}$$

Case 4: The last case is similar to the previous one. The only difference is that we load the sub-blocks in the diagonal block row-wise, whereas in the previous case we were loading them column-wise. Using again the example of Figure 12, we start by loading sub-blocks `(A1)` and `(D1)` and update `(A1)`. Keeping the elements from `(D1)`, we load `(A2)` and update it. We continue until we reach the end of the block. Then we move to the next row of sub-blocks. We load `(D2)` and keep it constantly in registers. We also load `(A1)` and `(A3)` and update `(A3)`. Then we load `(A2)` and `(A4)` and update `(A4)`. Again, we continue until the end of the block. In contrast to the previous case, we now load `(D3)` and keep it constantly in registers. We load `(A3)` and update it, then we load `(A4)` and update it and we continue to the end of the block. After we finish with the whole row of sub-blocks, we load `(D4)` and keep it constantly in registers. We also load `(A1)` and `(A5)` and update `(A5)`. Then we load `(A2)` and `(A6)` and update `(A6)` and we continue to the end of the row of sub-blocks. Then we load `(D5)` and keep it constantly in registers. We also load `(A1)` and `(A5)` and update `(A5)`. Then we load `(A2)` and `(A6)` and update `(A6)`. Again, we continue until we finish with the whole row of sub-blocks. Finally, we load `(D6)` and keep it constantly in registers. Then we load subsequently `(A5)` and update it, then `(A6)` and update it, until we reach again the end of the block.

In order to calculate the total number of loads and stores in this case, we start with the updates in the first row of sub-blocks. In the example of Figure 12 we need to load `(D1)` and then successively all of the M/L_2 sub-blocks that have to be updated in the column, i.e., `(A1)`, `(A2)`, etc. This leads to the equations:

$$Loads = L_1 \cdot L_2 \cdot \frac{M}{L_2} + \frac{L_1 \cdot (L_1+1)}{2}$$
$$Stores = \frac{M}{L_2} \cdot L_1 \cdot L_2$$

After that, we load `(D2)`, `(A1)` and `(A3)`, in order to update `(A3)`. We repeat the whole process for the whole row of sub-blocks to the right of `(A3)`. Then we load `(D3)` and update again sub-block `(A3)` and all sub-blocks to the right of it. This gives a total of:

$$Loads = \left[L_1^2 + (L_1 \cdot L_2 + L_1 \cdot L_2) \cdot \frac{M}{L_2}\right] + L_1 \cdot L_2 \cdot \frac{M}{L_2} + \frac{L_1 \cdot (L_1+1)}{2}$$
$$Stores = \frac{M}{L_2} \cdot L_1 \cdot L_2 + \frac{M}{L_2} \cdot L_1 \cdot L_2 = 2 \cdot \frac{M}{L_2} \cdot L_1 \cdot L_2$$

Subsequently, we repeat the whole process starting with `(D4)` and updating `(A5)` and all blocks that are to the right of it. The same has to be done for `(D5)` and `(A5)` and all blocks that are to the right of it. Finally, we load `(D6)` and update again `(A5)` and all blocks that are to the right of it. Hence:

$$Loads = 2 \cdot \left[L_1^2 + (L_1 \cdot L_2 + L_1 \cdot L_2) \cdot \frac{M}{L_2}\right] + L_1 \cdot L_2 \cdot \frac{M}{L_2} + \frac{L_1 \cdot (L_1 + 1)}{2}$$
$$Stores = 2 \cdot \frac{M}{L_2} \cdot L_1 \cdot L_2 + \frac{M}{L_2} \cdot L_1 \cdot L_2 = 3 \cdot \frac{M}{L_2} \cdot L_1 \cdot L_2$$

By adding all previous equations and generalizing, we conclude that the total number of loads and stores is in this case:

$$Loads = \sum_{i=1}^{\frac{N}{L_1}} \left[(i-1) \cdot \left[L_1^2 + (L_1 \cdot L_2 + L_1 \cdot L_2) \cdot \frac{M}{L_2}\right] + L_1 \cdot L_2 \cdot \frac{M}{L_2} + \frac{L_1 \cdot (L_1 + 1)}{2}\right]$$

$$Stores = \sum_{i=1}^{\frac{N}{L_1}} i \cdot L_1 \cdot L_2 \cdot \frac{M}{L_2}$$

Obviously, the above equations are exactly the same with the ones from the previous case, which means that the optimum number of loads and stores in this case is again:

$$Loads = \frac{M \cdot N^2}{6} + \frac{N \cdot (N + 1)}{2}$$
$$Stores = \frac{M \cdot N \cdot (N + 6)}{12}$$

The last step in our analysis is to compare the above results with the results obtained for all previous cases. In order to simplify this procedure, we remind the reader that the value of M can be either equal to N or $N + 1$, due to the way each block is defined. Setting the value of M to these values, it is very easy to verify that the last case we analyzed is each time the best one, in terms of both the total number of loads and stores required. Although the third case gives the same results as the last case, we chose to use the latter in our implementation. Our decision was driven by the fact that in a real implementation, moving to the next sub-blocks that have to be updated is easier in the last case.

6.3 Optimizing the `ProcessBlockOnColumn()` Function

The `ProcessBlockOnColumn()` function operates on blocks of data that share the same columns with the diagonal block. In order to update these elements, it also requires data from the diagonal block. As mentioned in Section 5.3, each block is assumed to have N rows and M columns. Each column i is used to update all subsequent columns, i.e., columns $i + 1$ to M. For this function, we identified four possible ways to apply register tiling. We assume that each block that has to be processed by the function has been divided into sub-blocks of size $L_2 \times L_1$. Due to data dependencies (see Figure 9), this means that the required data in the diagonal block has to be divided into sub-blocks of size $L_1 \times L_1$. Moreover, it is worth to notice that not all the data from

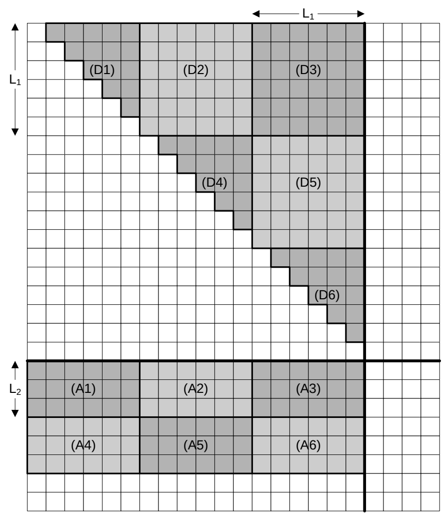

Figure 13: Dividing into sub-blocks for `ProcessBlockOnColumn()`.

the diagonal block is required, but only the data from the upper triangular part. Figure 13 shows an example of dividing a block into sub-blocks, which we will use throughout this paragraph. For the readers who would like to skip the detailed analysis, we mention that the optimal method to load sub-blocks proves to be the last one, which we also used in our implementation.

Case 1: In the first case, we start with the first sub-block of size $L_2 \times L_1$ and update it, using the corresponding triangular-shaped data sub-block from the diagonal block. These correspond to sub-blocks `(A1)` and `(D1)` in Figure 13. Then, we load successively each sub-block of size $L_2 \times L_1$ that is to the right of the first sub-block and the corresponding sub-block of size $L_1 \times L_1$ from the diagonal block and make the necessary updates. Using again Figure 13, while we still have `(A1)` in registers, we load `(A2)` and `(D2)` and update `(A2)`. Then we load `(A3)` and `(D3)` and update `(A3)`. Moving on, we load `(A2)` and `(D4)` and update `(A2)`. While we still have `(A2)` in registers, we load `(A3)` and `(D5)` and update `(A3)`. Finally, we load `(A3)` and `(D6)` and update `(A3)`. This completes the first row of sub-blocks and we can move to the next row, containing sub-blocks `(A4)`, `(A5)` and `(A6)`, in order to repeat the process.

Before calculating the total number of loads and stores required for this case, we have to calculate the number of loads required for the triangular sub-blocks of the diagonal block, i.e., blocks `(D1)`, `(D4)` and `(D6)` in the example of Figure 13.

$$\mathit{Loads\ in\ triangular\ sub\text{-}block} = 1 + 2 + 3 + \ldots + (L_1 - 1) = \frac{L_1 \cdot (L_1 - 1)}{2}$$

Having this result, we can move on to calculate the total number of loads and stores required. As mentioned earlier, we start by updating sub-block `(A1)` and then using this sub-block, we update `(A2)` and `(A3)`:

$$\mathit{Loads} = \left[L_1 \cdot L_2 + \frac{L_1 \cdot (L_1 - 1)}{2}\right] + \left[2 \cdot L_1 \cdot L_2 + 2 \cdot L_1^2\right]$$
$$\mathit{Stores} = 3 \cdot L_1 \cdot L_2$$

Then we update sub-block `(A2)` and then using this sub-block, we update `(A3)`:

$$
\begin{aligned}
Loads &= \left[L_1 \cdot L_2 + \frac{L_1 \cdot (L_1 - 1)}{2}\right] + \left[L_1 \cdot L_2 + L_1^2\right] \\
Stores &= 2 \cdot L_1 \cdot L_2
\end{aligned}
$$

Finally, we update sub-block `(A3)`:

$$
\begin{aligned}
Loads &= L_1 \cdot L_2 + \frac{L_1 \cdot (L_1 - 1)}{2} \\
Stores &= L_1 \cdot L_2
\end{aligned}
$$

Which leads us to the more general equations for the total number of loads and stores per row of sub-blocks:

$$
\begin{aligned}
Loads &= \sum_{i=1}^{\frac{M}{L_1}} \left[i \cdot L_1 \cdot L_2 + (i-1) \cdot L_1^2 + \frac{L_1 \cdot (L_1 - 1)}{2}\right] \\
Stores &= \sum_{i=1}^{\frac{M}{L_1}} i \cdot L_1 \cdot L_2
\end{aligned}
$$

However, the above procedure has to be repeated for every row of sub-blocks, which gives the final equations:

$$
\begin{aligned}
Loads &= \frac{N}{L_2} \cdot \sum_{i=1}^{\frac{M}{L_1}} \left[i \cdot L_1 \cdot L_2 + (i-1) \cdot L_1^2 + \frac{L_1 \cdot (L_1 - 1)}{2}\right] = \\
&= \frac{N}{L_2} \cdot \sum_{i=1}^{\frac{M}{L_1}} \left[i \cdot (L_1 \cdot L_2 + L_1^2) - L_1^2 + \frac{L_1 \cdot (L_1 - 1)}{2}\right] = \\
&= \frac{N}{L_2} \cdot \sum_{i=1}^{\frac{M}{L_1}} \left[i \cdot (L_1 \cdot L_2 + L_1^2) - \frac{L_1 \cdot (L_1 + 1)}{2}\right] = \\
&= \frac{N}{L_2} \cdot (L_1 \cdot L_2 + L_1^2) \cdot \sum_{i=1}^{\frac{M}{L_1}} i - \frac{N}{L_2} \cdot \frac{M}{L_1} \cdot \frac{L_1 \cdot (L_1 + 1)}{2} = \\
&= \frac{N}{L_2} \cdot L_1 \cdot (L_1 + L_2) \cdot \frac{\frac{M}{L_1} \cdot (\frac{M}{L_1} + 1)}{2} - \frac{M \cdot N \cdot L_1 \cdot (L_1 + 1)}{2 \cdot L_1 \cdot L_2} = \\
&= \frac{N}{L_2} \cdot (L_1 + L_2) \cdot \frac{M \cdot (M + L_1)}{2 \cdot L_1} - \frac{M \cdot N \cdot L_1 \cdot (L_1 + 1)}{2 \cdot L_1 \cdot L_2} = \\
&= \frac{M \cdot N \cdot (L_1 + L_2) \cdot (M + L_1) - M \cdot N \cdot L_1 \cdot (L_1 + 1)}{2 \cdot L_1 \cdot L_2} = \\
&= \frac{M \cdot N \cdot (M \cdot L_1 + M \cdot L_2 + L_1 \cdot L_2 - L_1)}{2 \cdot L_1 \cdot L_2}
\end{aligned}
$$

$$Stores = \frac{N}{L_2} \cdot \sum_{i=1}^{\frac{M}{L_1}} i \cdot L_1 \cdot L_2 = \frac{N}{L_2} \cdot L_1 \cdot L_2 \cdot \sum_{i=1}^{\frac{M}{L_1}} i = N \cdot L_1 \cdot \frac{\frac{M}{L_1} \cdot (\frac{M}{L_1} + 1)}{2} =$$
$$= \frac{M \cdot N \cdot (M + L_1)}{2 \cdot L_1}$$

In order to minimize the number of loads and stores, we have to calculate the parameters L_1 and L_2. For that, we have to take into account the fact that the number of available registers in our architecture is fixed. If we assume that we have R registers available, then at any point one $L_1 \times L_1$ sub-block from the diagonal block and two $L_2 \times L_1$ sub-blocks from the block that is being updated have to fit into the available registers, which means that:

$$2 \cdot L_1 \cdot L_2 + L_1^2 = R \Rightarrow L_2 = \frac{R - L_1^2}{2 \cdot L_1}$$

By replacing L_2 in the equation for the total number of loads and setting the derivative of that equation equal 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:

$$\frac{dLoads}{dL_1} = 0 \Rightarrow (M - 2) \cdot L_1^4 + (4 \cdot M \cdot R - 2 \cdot R) \cdot L_1^2 - M \cdot R^2 = 0 \Rightarrow$$
$$L_1^2 = R \cdot \frac{(1 - 2 \cdot M) \pm \sqrt{5 \cdot M^2 - 6 \cdot M + 1}}{M - 2}$$

By letting M move from a small value ($M = 3$) to infinity in the above equation, we can conclude that L_1^2 should move between the values:

$$0.236068 \cdot R \leq L_1^2 \leq 0.291503 \cdot R$$

Since R in Cyclops-64 can have a value of up to 48, we conclude that:

$$11.33 \leq L_1^2 \leq 13.99$$

Although these values are theoretically minimizing the total number of loads, they do not include an exact square, which is required in a real implementation of the algorithm, since L_1 should be an integer. Hence, we have to compromise and choose between the closest values for L_1, which are 3 and 4. If we choose $L_1 = 3$, then the largest R that we can use is 45, because this gives an integer value for L_2 of 6. If we choose $L_1 = 4$, then $R = 48$ and $L_2 = 4$. Between these two sets of values for L_1 and L_2, the one that gives the smallest number of loads is $L_1 = 4$ and $L_2 = 4$, which gives the final values for the total number of loads and stores:

$$Loads = \frac{M \cdot N \cdot (2 \cdot M + 3)}{8}$$
$$Stores = \frac{M \cdot N \cdot (M + 4)}{8}$$

Case 2: The second case can be seen as the reverse of the first one. Keeping one sub-block of size $L_2 \times L_1$ constant, we successively load every sub-block that is to the left of it, together with the corresponding sub-block of size $L_1 \times L_1$ from the diagonal block. Referring again to Figure 13, we start with sub-blocks `(A1)` and `(D1)` and update the first. Then we load sub-block `(A2)` and keep it constantly in registers. We also load `(A1)` and `(D2)` and update `(A2)`. Finally, we load `(D4)` and update again `(A2)`. Now we finished updating `(A2)` and we load `(A3)` into registers and keep it constantly there. Now we load `(A1)` and `(D3)` and update `(A3)`. Then we load `(A2)` and `(D5)` and update again `(A3)`. Finally, we load `(D6)` and update `(A3)`. This completes the first row of sub-blocks and we can move to the next row.

In order to calculate the total number of loads and stores in this case, we start with the update of sub-block `(A1)`:

$$Loads = L_1 \cdot L_2 + \frac{L_1 \cdot (L_1 - 1)}{2}$$
$$Stores = L_1 \cdot L_2$$

In order to update sub-block `(A2)`, we require sub-blocks `(A1)`, `(A2)`, `(D2)` and `(D4)`:

$$Loads = \left[L_1 \cdot L_2 + \frac{L_1 \cdot (L_1 - 1)}{2}\right] + \left[L_1 \cdot L_2 + L_1^2\right]$$
$$Stores = 2 \cdot L_1 \cdot L_2$$

Finally, in order to update sub-block `(A3)`, we require sub-blocks `(A1)`, `(A2)`, `(A3)`, `(D3)`, `(D5)` and `(D6)`:

$$Loads = \left[L_1 \cdot L_2 + \frac{L_1 \cdot (L_1 - 1)}{2}\right] + \left[2 \cdot L_1 \cdot L_2 + 2 \cdot L_1^2\right]$$
$$Stores = 3 \cdot L_1 \cdot L_2$$

Which leads us to the more general equations for the total number of loads and stores per row of sub-blocks:

$$Loads = \sum_{i=1}^{\frac{M}{L_1}} \left[i \cdot L_1 \cdot L_2 + (i-1) \cdot L_1^2 + \frac{L_1 \cdot (L_1 - 1)}{2}\right]$$
$$Stores = \sum_{i=1}^{\frac{M}{L_1}} i \cdot L_1 \cdot L_2$$

However, the above procedure has to be repeated for every row of sub-blocks, which gives the final equations:

$$Loads = \frac{N}{L_2} \cdot \sum_{i=1}^{\frac{M}{L_1}} \left[i \cdot L_1 \cdot L_2 + (i-1) \cdot L_1^2 + \frac{L_1 \cdot (L_1 - 1)}{2} \right]$$

$$Stores = \frac{N}{L_2} \cdot \sum_{i=1}^{\frac{M}{L_1}} i \cdot L_1 \cdot L_2$$

Obviously, the above equations are exactly the same with the ones from the previous case, which means that the optimum number of loads and stores in this case is again:

$$Loads = \frac{M \cdot N \cdot (2 \cdot M + 3)}{8}$$
$$Stores = \frac{M \cdot N \cdot (M + 4)}{8}$$

Case 3: The third case is different from the two previous ones. After updating each sub-block of size $L_2 \times L_1$ we move to the sub-block downwards. This leads to reusing data from the diagonal block, whereas in the previous cases, we were reusing data from the block that is being updated. Using again the example of Figure 13, we start by loading sub-blocks `(A1)` and `(D1)` and update `(A1)`. Keeping the elements from `(D1)`, we load `(A4)` and update it. We continue until we reach the end of the block. Then we move to the next column of sub-blocks. We load `(D2)` and keep it constantly in registers. We also load `(A1)` and `(A2)` and update `(A2)`. Then we load `(A4)` and `(A5)` and update `(A5)`. Again, we continue until the end of the block and then move to the next column of sub-blocks. We load `(D3)` and keep it constantly in registers. We also load `(A1)` and `(A3)` and update `(A3)`. Then we load `(A4)` and `(A6)` and update `(A6)`. After we finish with the whole column of sub-blocks, we load `(D4)` and keep it constantly in registers. We load `(A2)` and update it, then `(A5)` and update it and we continue to the end of the column of sub-blocks. Then we load `(D5)` and keep it constantly in registers. We also load `(A2)` and `(A3)` and update `(A3)`. Then we load `(A5)` and `(A6)` and update `(A6)`. Again, we continue until we finish with the whole column of sub-blocks. Finally, we load `(D6)` and keep it constantly in registers. Then we load subsequently `(A3)` and update it, then `(A6)` and update it, until we reach again the end of the block.

In order to calculate the total number of loads and stores in this case, we start with the updates in the first column of sub-blocks. In the example of Figure 13 we need to load `(D1)` and then successively all of the N/L_2 sub-blocks that have to be updated in the column, i.e., `(A1)`, `(A4)`, etc. This leads to the equations:

$$Loads = L_1 \cdot L_2 \cdot \frac{N}{L_2} + \frac{L_1 \cdot (L_1 - 1)}{2}$$
$$Stores = \frac{N}{L_2} \cdot L_1 \cdot L_2$$

After that, we load `(D2)`, `(A1)` and `(A2)`, in order to update `(A2)`. We repeat the whole process for the whole column of sub-blocks under `(A2)`. We also repeat the process for `(D3)` and sub-block `(A3)` and all sub-blocks below it. Combined with the above result, this gives a total of:

$$
\begin{aligned}
Loads &= 2 \cdot \left[L_1^2 + (L_1 \cdot L_2 + L_1 \cdot L_2) \cdot \frac{N}{L_2} \right] + L_1 \cdot L_2 \cdot \frac{N}{L_2} + \frac{L_1 \cdot (L_1 - 1)}{2} \\
Stores &= 2 \cdot \frac{N}{L_2} \cdot L_1 \cdot L_2 + \frac{N}{L_2} \cdot L_1 \cdot L_2 = 3 \cdot \frac{N}{L_2} \cdot L_1 \cdot L_2
\end{aligned}
$$

Subsequently, we repeat the whole process starting with `(D4)` and updating `(A2)` and all blocks that are below it. This time, however, we have to update only one column of sub-blocks more, by loading `(D5)` and updating `(A3)` and all sub-blocks below it. Hence:

$$
\begin{aligned}
Loads &= \left[L_1^2 + (L_1 \cdot L_2 + L_1 \cdot L_2) \cdot \frac{N}{L_2} \right] + L_1 \cdot L_2 \cdot \frac{N}{L_2} + \frac{L_1 \cdot (L_1 - 1)}{2} \\
Stores &= \frac{N}{L_2} \cdot L_1 \cdot L_2 + \frac{N}{L_2} \cdot L_1 \cdot L_2 = 2 \cdot \frac{N}{L_2} \cdot L_1 \cdot L_2
\end{aligned}
$$

Finally, we have to load `(D6)` and update `(A3)` and all sub-blocks below it:

$$
\begin{aligned}
Loads &= L_1 \cdot L_2 \cdot \frac{N}{L_2} + \frac{L_1 \cdot (L_1 - 1)}{2} \\
Stores &= \frac{N}{L_2} \cdot L_1 \cdot L_2
\end{aligned}
$$

By adding all previous equations and generalizing, we conclude that the total number of loads and stores is in this case:

$$
\begin{aligned}
Loads &= \sum_{i=1}^{\frac{M}{L_1}} \left[(i-1) \cdot \left[L_1^2 + (L_1 \cdot L_2 + L_1 \cdot L_2) \cdot \frac{N}{L_2} \right] + L_1 \cdot L_2 \cdot \frac{N}{L_2} + \frac{L_1 \cdot (L_1 - 1)}{2} \right] = \\
&= \sum_{i=1}^{\frac{M}{L_1}} \left[(i-1) \cdot (L_1^2 + 2 \cdot L_1 \cdot N) + N \cdot L_1 + \frac{L_1 \cdot (L_1 - 1)}{2} \right] = \\
&= (L_1^2 + 2 \cdot L_1 \cdot N) \cdot \frac{\frac{M}{L_1} \cdot (\frac{M}{L_1} - 1)}{2} + \frac{M}{L_1} \cdot N \cdot L_1 + \frac{M}{L_1} \cdot \frac{L_1 \cdot (L_1 - 1)}{2} = \\
&= (L_1 + 2 \cdot N) \cdot \frac{M \cdot (M - L_1)}{2 \cdot L_1} + \frac{2 \cdot M \cdot N \cdot L_1}{2 \cdot L_1} + \frac{M \cdot L_1 \cdot (L_1 - 1)}{2 \cdot L_1} = \\
&= \frac{M^2 \cdot L_1 + 2 \cdot N \cdot M^2 - M \cdot L_1}{2 \cdot L_1} = \\
&= \frac{M^2 \cdot N}{L_1} + \frac{M \cdot (M - 1)}{2}
\end{aligned}
$$

$$Stores = \sum_{i=1}^{\frac{M}{L_1}} i \cdot L_1 \cdot L_2 \cdot \frac{N}{L_2} = N \cdot L_1 \cdot \sum_{i=1}^{\frac{M}{L_1}} i = N \cdot L_1 \cdot \frac{\frac{M}{L_1} \cdot (\frac{M}{L_1} + 1)}{2} =$$
$$= \frac{M \cdot N \cdot (M + L_1)}{2 \cdot L_1}$$

As can be seen, the above results do not depend on L_2 at all and it is also obvious that the bigger L_1 is, the less the total number of loads and stores. Hence, we can set $L_2 = 1$. If we further assume that the number of available registers in our architecture is R, then at each point in time, the maximum number of elements loaded into registers cannot be more than R:

$$2 \cdot L_1 \cdot L_2 + L_1^2 = R \Rightarrow 2 \cdot L_1 \cdot 1 + L_1^2 = R \Rightarrow L_1^2 + 2 \cdot L_1 - R = 0 \Rightarrow L_1 = -1 \pm \sqrt{R+1}$$

As mentioned earlier, $R = 48$ for our architecture, which will give us a value of $L_1 = 6$. As a result, the optimum number of loads and stores in this case is:

$$Loads = \frac{M^2 \cdot N}{6} + \frac{M \cdot (M-1)}{2}$$
$$Stores = \frac{M \cdot N \cdot (M+6)}{12}$$

Case 4: The last case is similar to the previous one. The only difference is that we load the sub-blocks in the diagonal block column-wise, whereas in the previous case we were loading them row-wise. Using again the example of Figure 13, we start by loading sub-blocks `(A1)` and `(D1)` and update `(A1)`. Keeping the elements from `(D1)`, we load `(A4)` and update it. We continue until we reach the end of the block. Then we move to the next column of sub-blocks. We load `(D2)` and keep it constantly in registers. We also load `(A1)` and `(A2)` and update `(A2)`. Then we load `(A4)` and `(A5)` and update `(A5)`. Again, we continue until the end of the block. In contrast to the previous case, we now load `(D4)` and keep it constantly in registers. We also load `(A2)` and update it, then we load `(A5)` and update it and we continue to the end of the block. After we finish with the whole column of sub-blocks, we load `(D3)` and keep it constantly in registers. We also load `(A1)` and `(A3)` and update `(A3)`. Then we load `(A4)` and `(A6)` and update `(A6)` and we continue to the end of the column of sub-blocks. Then we load `(D5)` and keep it constantly in registers. We also load `(A2)` and `(A3)` and update `(A3)`. Then we load `(A5)` and `(A6)` and update `(A6)`. Again, we continue until we finish with the whole column of sub-blocks. Finally, we load `(D6)` and keep it constantly in registers. Then we load subsequently `(A3)` and update it, then `(A6)` and update it, until we reach again the end of the block.

In order to calculate the total number of loads and stores in this case, we start with the updates in the first column of sub-blocks. In the example of Figure 13 we need to load `(D1)` and then successively all of the N/L_2 sub-blocks that have to be updated in the column, i.e., `(A1)`, `(A4)`, etc. This leads to the equations:

$$
\begin{aligned}
Loads &= L_1 \cdot L_2 \cdot \frac{N}{L_2} + \frac{L_1 \cdot (L_1 - 1)}{2} \\
Stores &= \frac{N}{L_2} \cdot L_1 \cdot L_2
\end{aligned}
$$

After that, we load `(D2)`, `(A1)` and `(A2)`, in order to update `(A2)`. We repeat the whole process for the whole column of sub-blocks under `(A2)`. Then we load `(D4)` and update again sub-block `(A2)` and all sub-blocks below it. This gives a total of:

$$
\begin{aligned}
Loads &= \left[L_1^2 + (L_1 \cdot L_2 + L_1 \cdot L_2) \cdot \frac{N}{L_2}\right] + L_1 \cdot L_2 \cdot \frac{N}{L_2} + \frac{L_1 \cdot (L_1 - 1)}{2} \\
Stores &= \frac{N}{L_2} \cdot L_1 \cdot L_2 + \frac{N}{L_2} \cdot L_1 \cdot L_2 = 2 \cdot \frac{N}{L_2} \cdot L_1 \cdot L_2
\end{aligned}
$$

Subsequently, we repeat the whole process starting with `(D3)` and updating `(A3)` and all blocks that are below it. The same has to be done for `(D5)` and `(A3)` and all blocks that are below it. Finally, we load `(D6)` and update again `(A3)` and all blocks that are below it. Hence:

$$
\begin{aligned}
Loads &= 2 \cdot \left[L_1^2 + (L_1 \cdot L_2 + L_1 \cdot L_2) \cdot \frac{N}{L_2}\right] + L_1 \cdot L_2 \cdot \frac{N}{L_2} + \frac{L_1 \cdot (L_1 - 1)}{2} \\
Stores &= 2 \cdot \frac{N}{L_2} \cdot L_1 \cdot L_2 + \frac{N}{L_2} \cdot L_1 \cdot L_2 = 3 \cdot \frac{N}{L_2} \cdot L_1 \cdot L_2
\end{aligned}
$$

By adding all previous equations and generalizing, we conclude that the total number of loads and stores is in this case:

$$
Loads = \sum_{i=1}^{\frac{M}{L_1}} \left[(i-1) \cdot \left[L_1^2 + (L_1 \cdot L_2 + L_1 \cdot L_2) \cdot \frac{N}{L_2}\right] + L_1 \cdot L_2 \cdot \frac{N}{L_2} + \frac{L_1 \cdot (L_1 - 1)}{2}\right]
$$

$$
Stores = \sum_{i=1}^{\frac{M}{L_1}} i \cdot L_1 \cdot L_2 \cdot \frac{N}{L_2}
$$

Obviously, the above equations are exactly the same with the ones from the previous case, which means that the optimum number of loads and stores in this case is again:

$$
\begin{aligned}
Loads &= \frac{M^2 \cdot N}{6} + \frac{M \cdot (M - 1)}{2} \\
Stores &= \frac{M \cdot N \cdot (M + 6)}{12}
\end{aligned}
$$

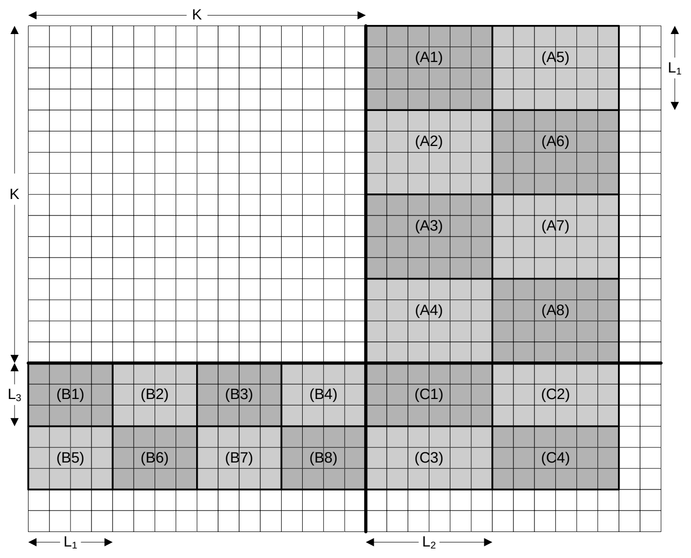

Figure 14: Dividing into sub-blocks for `ProcessInnerBlock()`.

The last step in our analysis is to compare the above results with the results obtained for all previous cases. In order to simplify this procedure, we remind the reader that the value of N can be either equal to M or $M + 1$, due to the way each block is defined. Setting the value of N to these values, it is very easy to verify that the last case we analyzed is each time the best one, in terms of both the total number of loads and stores required. Although the third case gives the same results as the last case, we chose to use the latter in our implementation. Our decision was driven by the fact that in a real implementation, moving to the next sub-blocks that have to be updated is easier in the last case.

6.4 Optimizing the `ProcessInnerBlock()` Function

The `ProcessInnerBlock()` function operates on blocks of data that share neither columns nor rows with the diagonal block. In order to update the elements in these blocks, it exclusively uses data from blocks that have been previously updated through the functions `ProcessBlockOnRow()` and `ProcessBlockOnColumn()`, i.e., no elements from the block being updated are used to update other elements in the same block. As mentioned in Section 5.4, each block is assumed to have N rows and M columns, whereas the diagonal block is assumed to be of size $K \times K$. For this function, we identified 6 possible ways to apply register tiling. Although all of them traverse in a different way all sub-blocks, they give in pairs the same results, with respect to the total number of loads and stores required. As a result, only three distinct cases have actually to be analyzed. During the analysis of each case, we will mention both cases that give the same results. We assume that each block that has previously been processed by `ProcessBlockOnColumn()`, is subdivided into sub-blocks of size $L_3 \times L_1$, whereas the block that has to be updated is subdivided into sub-blocks of size $L_3 \times L_2$. Due to data dependencies (see Figure 10), the block that has previously been processed by `ProcessBlockOnRow()` has to be divided into sub-blocks of size $L_1 \times L_2$. Figure 14 shows an example of dividing a block into sub-blocks, which we will use throughout this paragraph. For the readers who would like to skip the detailed analysis, we mention that the optimal method to load sub-blocks proves to be the last one, which we also used in our implementation.

Case 1: In the first case, we start with the first sub-block of size $L_2 \times L_3$ and update it, using the corresponding sub-blocks of size $L_3 \times L_1$ and $L_1 \times L_2$. Using the example of Figure 14, we would load `(C1)` and update it using `(A1)` and `(B1)`. While still having `(C1)` in registers, we would then update it again using `(A2)` and `(B2)`. Finally, we would repeat the process using the pairs of sub-blocks `(A3)` and `(B3)` and then `(A4)` and `(B4)`. This would give the following results:

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

At this point, we can move on to update the next sub-block, using the exact same procedure. This can be done either by moving horizontally to sub-block `(C2)` or vertically to sub-block `(C3)`. No matter which way we choose to continue, 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 $M/L_2 \cdot N/L_3$ sub-blocks that have to be updated and each one requires the aforementioned number of loads and stores. Hence, our final equations are:

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

Case 2: In the second case, we subsequently load and keep in registers sub-blocks of size $L_3 \times L_1$, updating all possible sub-blocks. Using again the example of Figure 14, we would load `(B1)` and use it to update `(C1)`, in conjunction with `(A1)`. While still having `(B1)` in registers, we would load `(C2)` and `(A5)`, in order to update `(C2)` and we would repeat the same process for all other sub-blocks to the right of `(C2)`. This would give the following results:

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

At this point, `(B1)` is not needed anymore and we can load the next sub-block. This can be done either by moving horizontally to sub-block `(B2)` or vertically to sub-block `(B5)`. No matter which way we choose to continue, 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 $K/L_1 \cdot N/L_3$ sub-blocks that have to be loaded and in each case we need the aforementioned number of loads and stores. Hence, our final equations are:

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

Case 3: In the third case, we subsequently load and keep in registers sub-blocks of size $L_1 \times L_2$, updating all possible sub-blocks. Using again the example of Figure 14, 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 sub-blocks under `(C3)`. This would give the following results:

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

At this point, `(A1)` is not needed anymore and we can load the next sub-block. This can be done either by moving horizontally to sub-block `(A5)` or vertically to sub-block `(A2)`. No matter which way we choose to continue, 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 $K/L_1 \cdot M/L_2$ sub-blocks that have to be loaded and in each case we need the aforementioned number of loads and stores. Hence, our final equations are:

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

Obviously, all three cases above yield quite similar results. Firstly, the number of loads does in each case depend only on two of the three parameters L_1, L_2 and L_3. Moreover, the bigger the remaining two parameters are, the smaller the number of loads required. Hence, the unused parameter should be set equal to one in each case, i.e., $L_1 = 1$ in the first case, $L_2 = 1$ in the second case and $L_3 = 1$ in the third case. Moreover, in each case the total number of elements that has to fit into the register file at each step is the same. If we assume that our architecture has R registers available, then:

$$L_1 \cdot L_2 + L_1 \cdot L_3 + L_2 \cdot L_3 = R$$

However, in each case one of the parameters L_1, L_2 and L_3 is equal to one:

$$\begin{aligned} &Case\ 1: \ L_2 + L_3 + L_2 \cdot L_3 = R \\ &Case\ 2: \ L_1 + L_3 + L_1 \cdot L_3 = R \\ &Case\ 3: \ L_1 + L_2 + L_1 \cdot L_2 = R \end{aligned}$$

Combining these last equations with the ones that give the number of loads in each case, reveals that the values of the parameters L_2 and L_3 that minimize the number of loads in the first case, will be exactly the same as the values for L_1 and L_3 in the second case, which will be equal to the values of L_1 and L_2 for the third case. Hence, the best case is actually determined by the factors that do not depend on L_1, L_2 and L_3 in the equations that give the number of loads, i.e., $M \cdot N$ in the first case, $K \cdot N$ in the second case and $K \cdot M$ in the third case. However, each block processed by this function is actually a set of smaller blocks, as defined by the dynamic algorithm. Moreover, blocks that share the same columns are connected and processed together in this function, which means that the number of rows is larger than the number of columns, i.e., $N > M$. Moreover, due to the way blocks are defined in our algorithm, $M \geq K$. This means that the smallest of the above factors is actually $K \cdot M$, which in turn means that the third case is the best one, in terms of the total number of loads and stores required to complete the algorithm.

Having reached this conclusion, we will determine now the exact values of L_1 and L_2 for our specific architecture. As determined above:

$$L_1 + L_2 + L_1 \cdot L_2 = R \Rightarrow L_2 = \frac{R - L_1}{L_1 + 1}$$

Replacing the value of L_2 in the equation that determines the number of loads for the third case and setting the derivative of that equation equal 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:

$$\frac{dLoads}{dL_1} = 0 \Rightarrow R \cdot L_1^2 + 2 \cdot R \cdot L_1 - R^2 = 0 \Rightarrow L_1 = -1 \pm \sqrt{R+1}$$

As mentioned earlier, $R = 48$ for our architecture, which will give us a value of $L_1 = 6$. This, in turn, leads to the conclusion that L_2 should also be equal to 6. Hence, the final equations for the total number of loads and stores required are:

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

6.5 Applying Architecture Specific Optimizations

We start this section by comparing the results we obtained from our analysis on the required number of loads and stores for both, the original versions to process each block and the optimized versions. We have to remind the reader at this point, that the results obtained for the optimized versions are not completely accurate, since we assumed that there are no remainders when we divide a block into smaller sub-blocks. Despite this fact, our results are still useful and can be used as a good estimate. The results are summarized in Table 5. As can be seen, all equations are of third order. However, the dominating term for all optimized versions is each time divided by 6, which is the optimal size for each sub-block, as calculated in the previous sections. The exceptions are the number of loads and stores for the diagonal block, since the optimal sub-block size is in this case 4 and the dominating factor is divided by this value. Hence, we expect the optimized versions to require far less loads and stores, in order to complete the calculations. Moreover, this result shows that a larger register file would benefit even more our application. The conclusion is that implementing the optimized versions in our algorithm would greatly reduce execution time.

	Loads (DAXPY)	Loads (Optimized)	Stores (DAXPY)	Stores (Optimized)
`ProcessDiagonalBlock()`	$\frac{2\cdot N\cdot(N^2-1)}{3}$	$\frac{N\cdot(N-1)\cdot(N+4)}{6}$	$\frac{N\cdot(N^2-1)}{3}$	$\frac{N\cdot(N+2)\cdot(N+4)}{6}$
`ProcessBlockOnRow()`	$M\cdot N\cdot(N+1)$	$\frac{M\cdot N^2}{6}+\frac{N\cdot(N+1)}{2}$	$\frac{M\cdot N\cdot(N+1)}{2}$	$\frac{M\cdot N\cdot(N+6)}{12}$
`ProcessBlockOnColumn()`	$\frac{M\cdot(2\cdot N+1)\cdot(M-1)}{2}$	$\frac{M^2\cdot N}{6}+\frac{M\cdot(M-1)}{2}$	$\frac{M\cdot N\cdot(M-1)}{2}$	$\frac{M\cdot N\cdot(M+6)}{12}$
`ProcessInnerBlock()`	$2\cdot K\cdot M\cdot N+K\cdot M$	$\frac{K\cdot M\cdot N}{3}+K\cdot M$	$K\cdot M\cdot N$	$\frac{K\cdot M\cdot N}{6}$

Table 5: Comparing the number of loads and stores.

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 sub-blocks 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 5. 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 4 or 6 times 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 available bandwidth. Similarly to the optimized load instruction, our architecture also provides the optimized store instruction *'stm RT, RA, RB'*. Naturally, we used it in our implementation to reduce the number of store instructions issued.

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.

7 Improving Instruction Scheduling

Having all the above information from our theoretical analysis and the multiple load and store instructions of our architecture, we finally implemented a first optimized version of the algorithm. Obviously, the whole process of dividing blocks into smaller sub-blocks and loading them into registers to perform the calculations, exceed the capability of modern compilers to automatically perform this task, as already discussed. Therefore, the whole procedure had to be written in assembly code. Although the performance increased about four times, compared to the version that used the `DAXPY()` function, it was still less than expected. For a matrix of size 512×512, the implementation that uses register tiling reached a performance of 7.09 GFlops, using 128 TUs, whereas the implementation before applying register tiling managed to reach only 1.81 GFlops.

In order to explain why the total performance was still quite low, we took advantage of another functionality that the simulator provides. Specifically, it is possible to create execution traces, which are text files that include detailed information about the execution of every assembly instruction on every thread unit. Among this information, the number of cycles that execution of an instruction stalls is reported. This may happen for several reasons, as for example, delays to transfer data from memory or data dependencies between instructions. This last remark is very important, since Cyclops-64 is an in-order processor and data dependencies can incur significant delays.

Our findings can be made clear by studying the assembly code of our initial implementation. The left part of Figure 15 shows the initial version of the main loop that performs the loads, calculations and stores for the `ProcessInnerBlock()` function. We present the code for this function, because it uses only square sub-blocks and is easier to understand. Nonetheless, the code for all other functions is very similar. We start by loading a square sub-block of size 6×6 (sub-blocks `A` in Figure 14) into registers `r28` to `r63`. Then we load one by one all the rows from the corresponding sub-blocks `B` and `C` into registers `r16` to `r21` and `r22` to `r27`. Finally, we update the elements from each row in `C` and store them back to memory. The assembly instruction *'fmsd' (Fused Multiply and Subtract Double)* maps directly to the dominant operation in LU, which updates an element by subtracting the product of two other elements. Using the first such instruction in the example of Figure 15, the operation that is performed is $r22 \leftarrow r22 - r16 \cdot r28$. From the trace files, we found out that this instruction requires 5 cycles to complete in Cyclops-64, after all required data has been loaded into registers. Since we update the same register six times consecutively, this means that there is a data dependence between these instructions and each one must wait for 5 cycles, until the previous one completes.

Fortunately, no data dependencies exist between different elements of the sub-block `C`. In our example, this means that registers `r22` to `r27` have no dependencies among them. As a result, updates between different registers can be performed in any order, although updates to each register have to be performed in a specific order. Taking advantage of this observation, we decided to interleave updates of different registers. This leads to our optimized version of the same code, which is shown in the right part of Figure 15. As can be seen, two subsequent updates of the same register are separated by updates of all other registers. Effectively, the five instructions that separate updates to the same register, exactly hide the latency of the first *'fmsd'* instruction to that register. By applying this simple optimization, the performance finally reached 11.19 GFlops.

We believe that compilers could easily perform this kind of reordering, especially on the C64 architecture, which executes instructions in-order. However, it was not possible for us to test whether this is true in the current tool-chain for C64. The reason is that in order to apply this kind of instruction scheduling, the compiler would have to perform first the register tiling optimization. Despite this fact, our experiment shows that instruction scheduling remains relevant on our architecture.

```
             /* Load elements from block A */
             ldm     r28, r8, r33
             add      r8, r8, r14
             ldm     r34, r8, r39
             add      r8, r8, r14
             ldm     r40, r8, r45
             add      r8, r8, r14
             ldm     r46, r8, r51
             add      r8, r8, r14
             ldm     r52, r8, r57
             add      r8, r8, r14
             ldm     r58, r8, r63

             /* Start loop over 'N' */
             beq   r12, LoopN_End
             mov     r7, r12

             /*
              * Load elements from
              * blocks C and B.
              */
LoopN_Start: ldm     r22, r10, r27
             ldm     r16,  r9, r21
             add      r9,  r9, r14

             /* Update values in C  */
             fmsd    r22, r16, r28
             fmsd    r22, r17, r34
             fmsd    r22, r18, r40
             fmsd    r22, r19, r46
             fmsd    r22, r20, r52
             fmsd    r22, r21, r58

             fmsd    r23, r16, r29
             fmsd    r23, r17, r35
             fmsd    r23, r18, r41
             fmsd    r23, r19, r47
             fmsd    r23, r20, r53
             fmsd    r23, r21, r59
             .
             .
             .
             fmsd    r27, r16, r33
             fmsd    r27, r17, r39
             fmsd    r27, r18, r45
             fmsd    r27, r19, r51
             fmsd    r27, r20, r57
             fmsd    r27, r21, r63

             /* Store values in C   */
             stm     r22, r10, r27
             add     r10, r10, r14

             /* Exit from 'N' loop  */
             addi     r7, r7, -1
             bne      r7, LoopN_Start
LoopN_End:
```

```
             /* Load elements from block A */
             ldm     r28, r8, r33
             add      r8, r8, r14
             ldm     r34, r8, r39
             add      r8, r8, r14
             ldm     r40, r8, r45
             add      r8, r8, r14
             ldm     r46, r8, r51
             add      r8, r8, r14
             ldm     r52, r8, r57
             add      r8, r8, r14
             ldm     r58, r8, r63

             /* Start loop over 'N' */
             beq   r12, LoopN_End
             mov     r7, r12

             /*
              * Load elements from
              * blocks C and B.
              */
LoopN_Start: ldm     r22, r10, r27
             ldm     r16,  r9, r21
             add      r9,  r9, r14

             /* Update values in C  */
             fmsd    r22, r16, r28
             fmsd    r23, r16, r29
             fmsd    r24, r16, r30
             fmsd    r25, r16, r31
             fmsd    r26, r16, r32
             fmsd    r27, r16, r33

             fmsd    r22, r17, r34
             fmsd    r23, r17, r35
             fmsd    r24, r17, r36
             fmsd    r25, r17, r37
             fmsd    r26, r17, r38
             fmsd    r27, r17, r39
             .
             .
             .
             fmsd    r22, r21, r58
             fmsd    r23, r21, r59
             fmsd    r24, r21, r60
             fmsd    r25, r21, r61
             fmsd    r26, r21, r62
             fmsd    r27, r21, r63

             /* Store values in C   */
             stm     r22, r10, r27
             add     r10, r10, r14

             /* Exit from 'N' loop  */
             addi     r7, r7, -1
             bne      r7, LoopN_Start
LoopN_End:
```

Figure 15: Assembly code before and after instruction scheduling.

8 Experimental Evaluation

In this section we present the experimental evaluation of LU, which focuses on the following key points.

- The effect of each optimization on the total performance of the application.
- The effect of the memory size on the performance and the efficiency of the optimal implementation.
- Comparison of the achieved performance with other architectures.

Since the actual C64 chip is not yet physically available, all experiments had to be conducted using the FAST simulator [6]. 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 the key components of the system, 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 dozens of developers for early application development.

The development tool-chain for C64 also includes version 4.1.1 of the gcc compiler. During compilation, we used the highest available optimization level (`-O3`), although the effect of this is quite limited. The functions that perform the bulk of the data transfer from memory and the calculations have been hand-written in assembly. The total percentage of execution time that corresponds to functions written in C, and therefore optimized by the compiler, is actually very low.

8.1 The Effect of Optimizations on the Performance

In this paragraph, we provide the necessary data to evaluate the effect that each optimization on the performance of LU has. During our experiments we used 128 TUs and a matrix with a size of 1024×1024. Although only a 512×512 matrix fits into the fast SRAM, in the default configuration of a chip, it is possible to redefine the size of the SRAM in the simulator. As will be explained in more detail in the next section, all matrices behave well for all optimizations that were described in Section 4. However, applying register tiling and instruction scheduling only show their full potential for larger matrices. Therefore, we used the above matrix to better highlight the differences in the last optimizations we applied. Figure 16 shows the results of our experiments. The first column in the graph represents the performance for the base implementation, which reaches only about 76 MFlops. The second column represents the performance after applying the dynamic repartitioning algorithm and recursion on the diagonal block. The 2.39 GFlops achieved represent an improvement of about 31.5 times, compared to the base implementation. As can be seen, the results for these two cases closely follow the ones presented at the end of Section 4. Introducing processor adaptation yields a marginal performance improvement of about 25 MFlops. The use of the hardware provided barriers, instead of our initial software-based solution, improves the performance by 129 MFlops, reaching a total of 2.54 GFlops. The largest improvements however, appear with the introduction of register tiling and instruction scheduling. The performance almost quadruples after applying the first optimization, reaching 9.90 GFlops. After applying the second optimization we finally reach our maximum of 21.92 GFlops.

From the above results, it is obvious that there are three optimizations that contribute the most in achieving the maximum performance. The first one is the introduction of dynamic repartitioning. Although the performance is

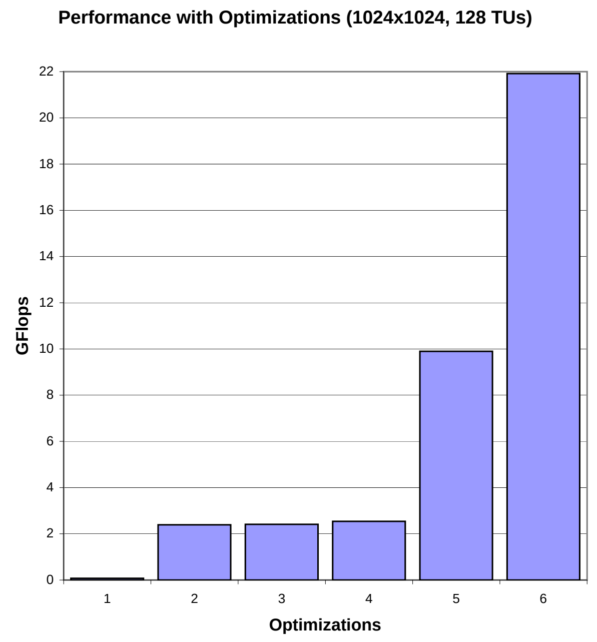

Figure 16: Evaluating the impact of each optimization on the performance of the application.

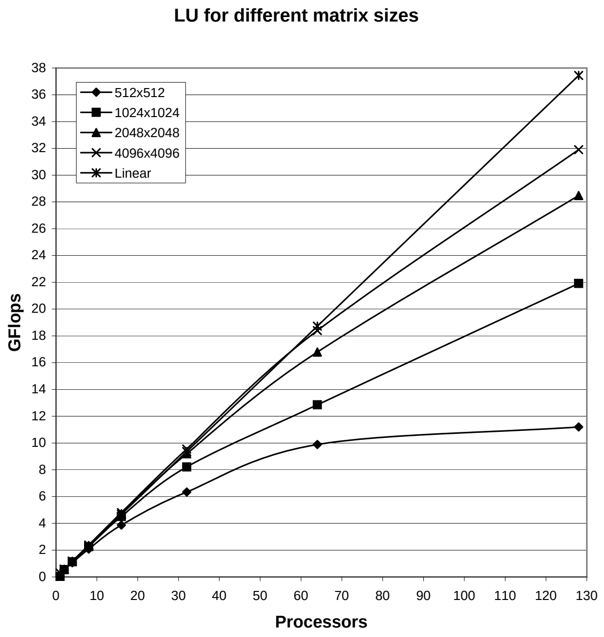

Figure 17: Performance achieved for different matrix sizes and number of TUs.

not very high, compared to the final numbers that we achieve, it marks the first leap from the extremely slow base implementation. 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, as expected, 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. The last optimization worth to be discussed is instruction scheduling. Although it is common knowledge that this is an important aspect on every architecture, it becomes even more important on C64. Due to its in-order execution engine, hiding latencies of instructions with data dependencies has to be performed and carefully tuned at the application or compiler level.

8.2 The Efficiency of the Dynamic Repartitioning Algorithm

As previously mentioned, only a 512×512 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 17. Although the total performance using 128 TUs is quite high, at 11.19 GFlops, it is obvious that our algorithm does not scale well, if more than 16 TUs are used. Especially the difference between 64 and 128 TUs is negligible, compared to the difference in the number of TUs. This behavior enabled us to further investigate why this is the case. Our main concern was whether this is a limitation of our algorithm and of the optimizations we applied or whether the reason was something totally different. After thoroughly studying the execution traces of several experiments, we concluded that the main problem is the current size of the available SRAM. Having a matrix of only 512×512 in SRAM, significantly limits the number of elements that have to be updated by each TU, especially as the number of TUs rises. As the efficiency of register tiling depends on the amount of data that is being reused, smaller data

	Peak GFlops	Actual GFlops	Efficiency
Cell Broadband Engine 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 6: Maximum performance achieved with one processor.

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 1024×1024 up to 4096×4096. The results of these experiments are also depicted in Figure 17. As can be seen, the performance improves dramatically as the size of the matrix increases. The performance for 1, 2 and 4 TUs is in all cases almost the same. Using this information, we extrapolated how our application would scale linearly. Doubling the size of the matrix, allows our algorithm to achieve linear speedup, up to twice the number of TUs of the previous matrix. For example, the 1024×1024 matrix scales linearly up to 16 TUs, the 2048×2048 up to 32 TUs and the 4096×4096 up to 64 TUs. 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. Being able to fit a 1024×1024 matrix in SRAM seems to be sufficient to exceed the performance of most current architectures and is probably an achievable goal. Nevertheless, real world applications require even larger matrices to be processed. In this case, the matrix can be stored in DRAM and parts of it can be moved into SRAM when they need to be processed. Moving data can even be overlapped with useful calculations, if we use a few of the available TUs specifically for this purpose. In this case, the maximum performance to process such a large matrix would equal the performance achieved for a matrix that can fit into SRAM.

8.3 Comparison with other Architectures

In this paragraph, we compare the results obtained on C64 with the results reported for other architectures. Table 6 summarizes the performance of the Linpack benchmark for a variety of systems, on a per chip basis. The results for the Cell Broadband Engine [5] were obtained for a matrix of size 1000×1000. For all other systems, the results are the ones reported in the Top500 list [17] 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 the Linpack benchmark, this comparison can still give some interesting insights.

Firstly, it is obvious that even for the smallest matrix size that we used (512×512), the 11.19 GFlops of C64 outperform the next best architecture, which is the Cell Broadband Engine with 9.46 GFlops. The difference rises much higher, if we use the matching 1024×1024 matrix on C64. All other architectures achieve an even lower total performance, even though they operate on much larger matrices. A significant drawback of C64, is the fact that it achieves a much lower efficiency, compared to all other architectures. The peak performance that can be achieved on 128 TUs is 64 GFlops, therefore the efficiency is 17.48% for the 512×512 matrix, 34.25% for the 1024×1024 matrix and reaches only 49.84% for the largest matrix we used. This can be explained by the fact that on C64 we use a much higher number of TUs, meaning that every TU has a much smaller data set to operate 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 1024×1024 matrix gives an efficiency rate of 59.50%, which is much closer to the one achieved by Cell. Moreover, Cell has a total of 128 registers, each one 128 bits wide, which means that two double-precision floating point numbers can simultaneously be loaded into each register and processed with the *Fused Multiply and Add* instruction, which is also supported on Cell. Hence, Cell has effectively four times more registers than C64. Nonetheless, we are confident that it is possible to further optimize our algorithm for C64 and improve the efficiency rate. Our optimism on this matter stems from the fact that the execution traces of our experiments reveal that a significant percentage of cycles is lost while waiting for data to be read from memory. Our first analysis shows that it is possible to overlap part of these lost cycles with useful calculations.

9 Related Work

Probably the most used implementation of LU is the one provided by HPL [11]. It is a blocking algorithm, originally developed to run on distributed-memory systems. Over the years, it has become the standard benchmark for ranking high-end parallel systems in the Top 500 list [17]. Although this implementation includes an algorithm to solve the two triangular systems that result from the LU algorithm, the largest percentage of the computation is actually performed in the latter. The usage of BLAS-3 routines to process the elements within each block provides the base to achieve high-performance on cache based systems.

The SPLASH-2 [19] implementation of the algorithm targets shared-memory systems. It uses small blocks and BLAS-1 routines to achieve a good balance between load-balancing and memory locality. Although this works well for cache based systems, architectures that lack a cache seem not to benefit from this approach.

A recursive algorithm for LU has been proposed for uni-processor systems [10]. The main characteristic of this algorithm is that it tries to reduce the working set at each level of recursion, exploiting in a better way the cache. In our case, however, 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 [13]. The first category divides the matrix into horizontal stripes. As soon as the first processor finishes with the first column in its stripe, it moves to the second column and informs the second processor that it can start calculating its own first column. After the second processor finishes, the third can start, etc. This creates a pipelined execution, which is again specifically designed to take advantage of the cache hierarchy. 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, i.e., systolic arrays.

10 Future Work and Conclusions

In this report we presented an implementation of the LU application for C64. The design of our algorithm significantly differs from other algorithms that solve the same problem. These differences stem from the fact that the design of our target architecture is decoupled from other contemporary and more conventional architectures. The key differences between these architectures is the lack of a hardware controlled cache-hierarchy in C64, which has been replaced with a software-controlled memory hierarchy, and the large number of cores that have been integrated into a single chip.

Our first conclusion during development was the fact that the design of most algorithms that solve the problem under consideration, implicitly assume that most architectures include a cache-hierarchy. As a result, many decisions are driven by this assumption. However, we have shown in this report that these assumptions are not adequate for multi-core architectures that lack a cache. Our point of view is that algorithms have to be rethought and redesigned, in order to comply with the idiosyncracies of the underlying architecture, if high performance is to be achieved. Our *Dynamic Repartitioning* algorithm, enhanced with *Recursion on the Diagonal Block* and *Processor Adaptation*, highlights some important points that have to be taken into consideration in this case. We believe that similar inefficiencies occur in many algorithm, dye to these implicit assumptions. Other optimization techniques used to improve performance on conventional architectures may give excellent results on multi-core architectures, if properly modified. 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. Starting, however, from both ends, the high-level algorithm and the multi-core architecture, allows register tiling to connect them in a much more efficient way. Finally, other aspects of application optimization, such as instruction scheduling, retain their importance across architectures.

Our next step in the development of the LU application is to further improve the low level integration with the underlying architecture. We believe that it is possible to further improve instruction scheduling, according to the execution traces of our experiments. Another important aspect is the fact that the C64 architecture actually includes a large number of nodes, creating a distributed-memory system across nodes and a shared-memory system within each node. 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, in a version of our code that would be able to run on several nodes, the code that has been developed up to this point would be reused to process the diagonal block on just one node. However, routines that would handle all other kind 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.

We believe that our methodology can be applied to other linear algebra problems too. A good candidate seems to be matrix multiplication. Although this application has already been studied on C64 [12], the approach is not as systematic as ours and preliminary results indicate that we are able to achieve a much higher performance. Cholesky factorization is also an interesting application where our methodology could be applied. In general, it seems that almost any application based on calculations on matrices is a potential candidate for applying our methodology. This enables us to explore the possibility to re-implement BLAS routines using our methodology. Although some work has been done towards this goal for other multi-core architectures [2], we believe that our methodology provides an alternative approach that is worth investigating. This work would allow us to have a common base, with respect to other multi-core architectures, in order to better evaluate the performance achieved by C64.

References

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

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

[3] 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.

[4] S. Carr, K. S. McKinley, and C. W. Tseng. Compiler Optimizations for Improving Data Locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, California, 1994.

[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, November 2005.

[6] 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.

[7] 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, Colorado, April 2005.

[8] J. J. Dongarra, J. Du 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.

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

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

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

[12] 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, August 2006.

[13] Haoqiang Jin, Michael Frumkin, and Jerry Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report nas-99-011, NASA Ames Research Center, 1999.

[14] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[15] 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, July 1996.

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

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

[18] 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, February 2007.

[19] S. C. Woo, M. Ohara, E. Torrie, J. Pal 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.