Performance Modelling and Optimization of Memory Access on Cellular Computer Architecture Cyclops64

Yanwei Niu, Ziang Hu, Kenneth Barner, and Guang R. Gao

Department of ECE, University of Delaware, Newark, DE, 19711, USA
{niu, hu, barner, ggao}@ee.udel.edu

Abstract. This paper focuses on the Cyclops64 computer architecture and presents an analytical model and performance simulation results for the preloading and loop unrolling approaches to optimize the performance of SVD (Singular Value Decomposition) benchmark. A performance model for dissecting the total execution cycles is presented. The data preloading using "memcpy" or hand optimized "inline" assembly code, and the loop unrolling approach are implemented and compared with each other in terms of the total number of memory access cycles. The key idea is to preload data from offchip to onchip memory and store the data back after the computation. These approaches can reduce the total memory access cycles and can thus improve the benchmark performance significantly.

1 Introduction

The design concept of computer architecture over the last two decades has been mainly on the exploitation of the instruction level parallelism, such as pipelining,VLIW or superscalar architecture. For the next generation of computer architecture, hardware threading multiprocessor is becoming more and more popular. One approach of hardware multithreading is called CMP (Chip MultiProcessor) approach, which proposes a single chip design that uses a collection of independent processors with less resource sharing. An example of CMP architecture design is Cyclops64 [1,2,3,4,5], a new architecture for high performance parallel computers being developed at the IBM T. J. Watson Research Center and University of Delaware. More details of Cyclops64 architecture are described in Section 2.

This paper focuses on the Cyclops64 computer architecture and presented performance model and simulation results for the preloading and loop unrolling approach to optimize the performance of SVD benchmark. The key idea is to preload data from offchip to onchip memory and store the data back after the computation. The contributions include: (1) a performance model for dissecting the total execution cycles; (2) detailed analysis of the tradeoff of the data preloading approaches using "memcpy" or hand optimized "inline" assembly code, and the loop unrolling approach.

The remainder of this paper is organized as follows. The target platform Cyclops64 will be introduced in Section 2. The SVD benchmark and the GaoThomas algorithm are presented in Section 3. Different memory access approaches are introduced in Section 4 and detailed analysis of these approaches in Section 5. Simulation results and validation of the analysis are shown in Section 6. The conclusions are summarized in Section 7.

H. Jin, D. Reed, and W. Jiang (Eds.): NPC 2005, LNCS 3779, pp. 132–143, 2005.
© IFIP International Federation for Information Processing 2005

2 Cyclops64 Hardware and Software

Cyclops64(C64) is a petaflop supercomputer project under development at IBM research Laboratory. The Cyclops64 project is a renovative idea to explore the thread-level parallelism. Figure.1 shows the hardware architecture of a Cyclops64 chip, the main component of a Cyclops64 node. Each Cyclops64 chip has 80 processors, each consisting of two thread units, a floating-point unit and two SRAM memory banks of 32KB each. A 32KB instruction cache, not shown in the figure, is shared among five processors. In a Cyclops64 chip architecture there is no data cache. Instead a half of each SRAM bank can be configured as scratch-pad memory. Such a memory provides a fast temporary storage to exploit locality under software control. The latency of onchip scratch-pad memory is 2 cycles. Cyclops64 system also has offchip memory modules. The default offchip latency is 36 cycles. It could become larger when there is heavy load of memory accesses from many thread units. This parameter can be preset in the Cyclops64 simulator. In this paper, we preset the offchip latency to be 36 or 80.

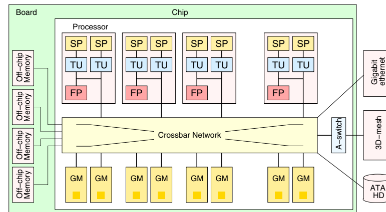

Fig. 1. Cyclops64 Chip

On the software side, one important part of the Cyclops64 system software is the Cyclops64 thread virtual machine. CThread is implemented directly on top of the hardware architecture as a micro-kernel/run-time system that fully takes advantage of the Cyclops64 hardware features. Cyclops64 thread virtual machine includes a thread model, a memory model and a synchronization model. The details of those models are explained in [6]. Suffice it to say that, the Cyclops64 chip hardware supports a shared address space model: all on chip SRAM and off-chip DRAM banks are addressable from all thread units/processors on the same chip.

3 SVD for Complex Matrices

In our implementation, we will focus on the one sided Jacobi SVD method since it is most suitable for parallel computing. The idea is to generate an orthogonal matrix V such that the transformed matrix $AV = W$ has orthogonal columns. Normalizing the Euclidean length of each nonnull column of W to unity, we will get the relation:

$$W = U\Sigma, \tag{1}$$

where the U is a matrix whose nonnull columns form an orthonormal set of vectors and Σ is a nonnegative diagonal matrix. Since $V^H V = I$, where I is the identity matrix, we have the SVD of A given by $A = U\Sigma V^H$.

Hestenes [7] uses plane rotations to construct V. He generates a sequence of matrices $\{A_k\}$ using the rotation

$$A_{k+1} = A_k Q_k \tag{2}$$

where the initial $A_1 = A$ and Q_k is a plane rotation matrix. The post-multiplication by Q_k affects only two columns, denoted by $\boldsymbol{u}$ and $\boldsymbol{v}$, for real matrices, we have:

$$(\boldsymbol{u}', \boldsymbol{v}') = (\boldsymbol{u}, \boldsymbol{v}) \begin{pmatrix} c & s \\ -s & c \end{pmatrix}. \tag{3}$$

For complex matrices, we have

$$(\boldsymbol{u}', \boldsymbol{v}') = (\boldsymbol{u}, \boldsymbol{v}) \begin{pmatrix} e^{j\beta} & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} e^{-j\beta} & 0 \\ 0 & 1 \end{pmatrix}. \tag{4}$$

where the angel β is from w: $w = |w|e^{j\beta}$, the formulas to get c and s are:

$$\alpha = \frac{y - x}{2|w|}, \qquad \tau = \frac{sign(\alpha)}{|\alpha| + \sqrt{1 + \alpha^2}}$$
$$c = \frac{1}{\sqrt{1 + \tau^2}}, \qquad s = \tau c. \tag{5}$$

We set $c = 1$ and $s = 0$ if $|w| = 0$. The peudocode of the one-sided Jacobi routine for complex matrices is show in Listing.1.1, which we refer to as “basic rotation routine”.

```
1   Rotation_of_two_column(colu, colv)
2   {
3       /*colu and colv are two columns of complex numbers */
4        w=inner_product(colu,colv);
5       if(|w| <= delta) {converged <- true; return;};
6       x=inner_product(colu, colu);
7       y=inner_product(colv, colv);
8
9       computer rotation parameter c,s from w, x, y according to Equation 5;
10      update colu, colv according to rotation Equation 4;
11  }
```

Listing 1.1. Rotation of two column of complex numbers

3.1 GaoThomas Algorithm

The plane rotations have to be applied to all column pairs exactly once in any sequence (a sweep) of $n(n-1)/2$ rotations. Several sweeps are required so that the matrix converges. A simple sweep can be a cyclic-by-rows ordering. For instance, let us consider a matrix with 4 columns, with the cyclic-by-rows order, the sequence of a sweep is:

$$(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4). \tag{6}$$

```
1  Rotation_of_two_column(colu, colv)
2  {
3
4    Allocate local_colu, local_colv
5    on the scratch-pad;
6
7    memcpy (local_colu <-colu);
8    memcpy (local_colv <-colv);
9
10   conduct three inner products and
11   column rotation on local_colu, local_colv
12   as in Listing.1.1
13
14   memcpy (colu <-local_colu);
15   memcpy (colv <-local_colv);
16 }
```

Listing 1.2. Basic rotation routine with preloading using "memcpy"

It is easy to see some pairs are independent and may be executed in parallel if we change the order in the sequence. Another possible sequence for a sweep can group independent pairs and executes them in parallel:

$$\{(1,2),(3,4)\},\{(1,4),(2,3)\},\{(1,3),(2,4)\}, \tag{7}$$

where the pairs in curly brackets are independent.We call each of these groups a step.

In this research, we implemented the GaoThomas algorithm. This algorithm computes the pairs of n elements on $n/2$ processors when n is a power of 2. A sweep is composed of $n-1$ steps, each step consisting of $n/2$ pairs of rotations. Therefore, one sweep consists of $n(n-1)/2$ rotations. In our shared memory implementation, the number of slave threads p can be set to be equal to the number of available processors. All the column pairs in one step can be treated as a work pool, the works in this work pool are shared among the p slave threads, where $1 \leq p \leq \frac{n}{2}$.

GaoThomas algorithm can compute $n(n-1)/2$ rotations of a matrix with n columns on $n/2$ processors. When the size of the matrix increases, group based GaoThomas algorithm can be adopted. For instance, when the matrix size is now $2n$ and we only have $n/2$ processors, we can group two columns together and treat them as one single unit. Generally speaking, for a matrix with n columns, if we group g columns together as a group, then we have n/g groups and can use the basic GaoThomas algorithm for n/g elements, except now each element is a group. For a matrix with n columns and group size g, one sweep contains $n/g-1$ steps, each step contains $n/2g$ instances of a rotation of two groups, which can run in parallel on maximum $n/2g$ processors. The rotation of two groups includes the rotation of all possible pairs of matrix columns in these two groups.

4 Optimization of Memory Access

4.1 Naive Approach

The default memory allocation using "malloc()" in the Cyclops64 simulator is from the offchip memory, while the local variables are allocated from the stack located on

the onchip scratch-pad memory. Assuming that the matrix data originally reside on the offchip memory, we implemented an SVD program where all the memory accesses are from the offchip memory. This implementation is referred to as the naive version in the following discussions. Also, the loop within the inner product computation of the rotation routine is implemented without any loop unrolling in the naive approach.

4.2 Preloading

In order to reduce the cycles spent on memory accesses, we can preload the data from the offchip memory to the onchip scratch-pad memory. Thus the data accesses in the computation part of the rotation routine are directly from the onchip memory. The updated data are then stored back to the offchip memory.

There are two ways to preload data. The simplest way is to use the "memcpy" function from the C library. The pseudo-code for the "memcpy" preloading in the two-column rotation routine is shown in Listing 1.2. We refer to the code segment from the line 10 to line 12 as the "computation core", which consists of the computation of three inner products and a column rotation. Preloading for the group based rotation routine is similar, except that two "groups" of columns are preloaded. The "memcpy" function based preloading has the problem of paying extra overhead of function calling. Additionally, the assembly code of the "memcpy" function is not fully optimized, which is shown with analysis in the next section.

To overcome these two problems, we implement preloading by using an optimized inline assembly code instead of a function call. We refer to this approach as the "inline" approach. For this approach, each "memcpy" function call is replaced with a segment of inline assembly code. The assembly code segment for the "memcpy" and "inline" preloading approaches (either group based rotation routine or basic rotation routine) are shown in Listing 1.4 and Listing 1.5. From the listings, we can see that memcpy and inline approaches have different instruction scheduling. The effect of different ways of instruction scheduling on the total memory access cycles is analyzed in Section 5.

4.3 Loop Unrolling of Inner Product Computation

There are three inner product function calls in the rotation routine. We implemented two versions of loop unrolling for the loop in the inner product computation: unrolling the loop body 4 times or 8 times. The idea is that loop unrolling makes it possible to schedule instructions from multiple iterations, thus facilitating the exploitation of instruction level parallelism.

5 Performance Model

5.1 Dissection of Execution Cycles

We begin with a simple execution trace example in Listing 1.3 to illustrate how to dissect total execution cycles into several parts. In the listing, the first column is the

current cycle number. We notice that at cycle 98472, there is a note "DLL = 1", which means that there is a one-cycle latency related to memory access. The reason is that at cycle 98472, the instruction needs the operand R9, which is not ready at cycle 98472 because the LDD instruction at cycle 98470 has two cycles of latency. Similarly, at cycle 98475, the FMULD instruction needs the input operand R8 generated by the FDIVD instruction at cycle 98469. R8 is not ready at cycle 98475 and needs an extra latency of 25 cycles since the FDIVD instruction has 30 cycles of latency from the float point unit. Counting the total number of cycles from cycle 98469 till cycle 98501, there are 33 cycles which include 7 instructions, 1 cycle of "DLL" and 25 cycles of "DLF". The integer unit may also cause certain latency called "DLI", which is similar to the "DLF" in the trace example. Therefore, we have the following equation:

$$\begin{aligned} Total \quad cycles = \quad & INST \\ & + \quad DLL + \quad DLF + \quad DLI, \end{aligned} \tag{8}$$

where the " INST" part stands for the total number of instructions, "DLL" represents the cycles spent on memory access, "DLF" represents the latency cycles related to floating point instructions, and "DLI" represents the latency cycles related to integer instructions.

```
98469 FDIVD   R8,R60,R8
98470 LDD     R9,R3,96
98471 ORI     R21,R0,0
98472 FDIVD   R20,R9,R62     DLL = 1
98474 LDD     R60,R3,104
98475 FMULD   R6,R61,R8      DLF = 25
98501 STD     R8,R3,160
```

Listing 1.3. Example of dissection of execution cycles

5.2 Analysis of Naive Approach

All memory accesses in the naive approach are from the offchip memory and the computation core part has a large number of "DLL" latency cycles. We denote the size of the matrix as $n \times n$. Each element of this matrix is a double complex number. We focus on one sweep that consists of $\binom{n}{2}$ basic rotations for either the non-group based approach or the group based approach. A basic rotation, as shown in Listing 1.1 consists of two different parts, the inner product part and the column rotation part. We analyze the total "DLL" latency cycles for both of them in this subsection.

First, there are three inner product function calls in the basic rotation routine. Each one of them consists of n iterations, each iteration producing a multiplication of two complex numbers and adding it to the sum. From the trace of the innermost iteration (the offchip latency is set to be 80 cycles), we see that the innermost iteration has a "DLL = 76". In general, if we preset the offchip latency to be L cycles, then the total number of "DLL" cycles in each iteration is $L - 4$. Therefore, in one sweep, the total number of "DLL" cycles within the inner product part is:

$$DLL_{innerproduct} = \binom{n}{2} \times 3 \times n \times (L - 4), \tag{9}$$

Second, for the column rotation part in the basic rotation routine, we conduct a similar analysis. The total number of "DLL" cycles of this part is:

$$DLL_{column_rotation} = \binom{n}{2} \times n \times (L - 4). \tag{10}$$

Therefore the total number of "DLL" cycles in the naive implementation of GaoThomas algorithm (either group based or non_group based, just one sweep) including both inner product and column rotation is:

$$\begin{aligned} DLL_{naive} &= DLL_{innerproduct} + DLL_{column_rotation} \\ &= \tbinom{n}{2} \times n \times (4L - 16). \end{aligned} \tag{11}$$

5.3 Analysis of "Memcpy" Approach

Using either the "memcpy" or "inline" preloading approach, the computation core accesses data from the onchip memory. The "DLL" part in the computation core is roughly zero due to the overlap of the short onchip memory access latency (2 cycles) with the float point unit latency. Therefore, from the program without preloading to the program with preloading, the decrease of the total number of "DLL" cycles in the computation core is DLL_{naive}, which is the cycles we save by using preloading, and thus the gain we expect to get.

Moving data from the offchip memory to the onchip memory results in an extra cost, which consists of two parts: the first part is the total "DLL" cycles in the code segment that is responsible for moving data, and the second part is the extra instructions incurred. The equation for the first part is derived as follows.

First, we derive the total number of "memcpy" function calls (which are responsible for loading data "in"). For the basic non-group-based GaoThomas algorithm, there are totally $\tbinom{n}{2}$ basic rotations (shown in Listing 1.1) in one sweep. A basic rotation needs to load in two columns, each of length n. Loading a double complex number needs two "LDD" instructions. Therefore, the total number of "LDD"s for preloading data is:

$$\begin{aligned} LDD_{memcpy_no_group} &= \tbinom{n}{2} \times 2 \times n \times 2 \\ &= \tbinom{n}{2} \times 4n, \end{aligned} \tag{12}$$

where the first "2" stands for loading "two" columns, n is that the length of the column, and the second "2" means that loading a double complex number needs two LDDs.

For the group based algorithm, if the group size is g, there are totally $\tbinom{n/g}{2}$ group based rotations. At the beginning of each group based rotation, we need to load in two groups of columns (i.e, $2 \times g$ columns) and each column needs $n \times 2$ LDDs. Therefore, the total number of LDDs for preloading data during one sweep is:

$$\begin{aligned} LDD_{memcpy} &= \tbinom{n/g}{2} \times 2g \times n \times 2 \\ &= \tbinom{n/g}{2} \times g \times 4n. \end{aligned} \tag{13}$$

If we treat the non-group-based GaoThomas algorithm as a group-based algorithm with group size one, then we can use (13) for either the group based algorithm or non-group-based algorithm.

Second, we compute the latency incurred by the LDDs. The execution trace segment of the assembly code for the "memcpy" function is shown in Listing 1.4, with the offchip latency set to be 80. From the Listing 1.4, we observe that each LDD instruction causes a long latency of 80 cycles, which is reflected where the "STD" instructions exist. If we preset the offchip latency to be L, then each "LDD" causes a latency of L cycles. So the total number of "DLL" cycles for preloading data using "memcpy" is:

$$\begin{aligned} DLL_{memcpy} &= LDD_{memcpy} \times L \\ &= \binom{n/g}{2} \times g \times 4n \times L. \end{aligned} \tag{14}$$

In addition to the change in the total "DLL"s, we also observe the increase in the total instruction count as:

$$Total \quad INST \quad increase = \binom{n/g}{2} \times g \times 4n \times 2 \times 2, \tag{15}$$

where the first part $\binom{n/g}{2} \times g \times 4n$ is the total number of "LDD"s for preloading data. We need a same amount of "STD", thus a multiplication by 2. Also we need to use "LDD" and "STD" to store data back, thus another multiplication by 2.

5.4 Analysis of "Inline" Approach

The total amount of data preloaded for the "inline" preloading approach is the same as the "memcpy" approach. Therefore the total number of "LDD"s of the inline approach is the same as the "memcpy" approach:

$$LDD_{inline} = \binom{n/g}{2} \times 2g \times n \times 2 \tag{16}$$

In the "inline" approach, 8 LDDs in a row are followed by 8 STDs in a row, as shown in Listing 1.5. From the trace we can see that we will have one "DLL=73" every 8 LDDs if we preset the offchip latency to be 80. If the offchip latency is L cycles, there is a "DLL=$L - 7$" every 8 "LDD" instructions. Therefore, the total number of "DLL" cycles for preloading data using the "inline" approach is:

$$\begin{aligned} DLL_{inline} &= LDD_{inline}/8 \times (L - 7) \\ &= \frac{1}{8} \times \binom{n/g}{2} \times g \times 4n \times (L - 7). \end{aligned} \tag{17}$$

From (17), we can see very clearly that preloading data using the "inline" approach is better than using the "memcpy" approach because DLL_{inline} is approximately $1/8$ of DLL_{memcpy}.

5.5 Analysis of the Loop Unrolling

The loop unrolling method only affects the inner product routine. For unrolling 4 times, eaczzzh inner product routine now contains $n/4$ iterations, each iteration consisting of computation of the sum of 4 multiplications of complex number. Based on the trace

```
105375  LDD   R6,R9,0
105376  STD   R6,R7,0      DLL = 80
105457  ADDI  R9,R9,8
105458  ADDI  R7,R7,8
105459  LDD   R6,R9,0
105460  STD   R6,R7,0      DLL = 80
105541  ADDI  R9,R9,8
105542  ADDI  R7,R7,8
105543  LDD   R6,R9,0
105544  STD   R6,R7,0      DLL = 80
105625  ADDI  R9,R9,8
105626  ADDI  R7,R7,8
105627  LDD   R6,R9,0
105628  STD   R6,R7,0      DLL = 80
```

Listing 1.4. Trace of the memcpy approach

```
112688  LDD   R16,R9,0
112689  LDD   R17,R9,8
112690  LDD   R18,R9,16
112691  LDD   R19,R9,24
112692  LDD   R20,R9,32
112693  LDD   R21,R9,40
112694  LDD   R22,R9,48
112695  LDD   R28,R9,56
112696  STD   R16,R6,0      DLL = 73
112770  STD   R17,R6,8
112771  STD   R18,R6,16
112772  STD   R19,R6,24
112773  STD   R20,R6,32
112774  STD   R21,R6,40
112775  STD   R22,R6,48
112776  STD   R28,R6,56
```

Listing 1.5. Trace of the “inline” approach

of the innermost iteration, the “DLL” incurred inside the inner product part can be summarized in (18):

$$DLL_{innerproduct_unroll4} = \binom{n}{2} \times 3 \times \frac{n}{4} \times (L - 8). \tag{18}$$

Similar analysis of unrolling 8 times can give us:

$$DLL_{innerproduct_unroll8} = \binom{n}{2} \times 3 \times \frac{n}{8} \times (L - 13). \tag{19}$$

6 Simulation Result

6.1 Cyclops64 Simulation Environment

The software tool chain of Cyclops64 platform currently provides a compiler, linker and simulator for users. A number of optimization levels are supported by the compiler. A multi-chip multi-threading functional accurate simulator (FAST) is also provided. We

Table 1. Model validation

		Latency=36		Latency=80	
		STD related DLL Latency	Computation core DLL Latency	STD related DLL Latency	Computation core DLL Latency
naive	Measured	52416	16646112	52416	39354336
memcpy	Measured	19664064	2016	42372288	2016
	Change from Naive	19611648	16644096	42319872	39354336
	Predicted change	18579456	16515072	41287680	39223296
	Diff percentage	5.41%	0.78%	2.47%	0.33%
inline	Measured	1943424	2016	4781952	2016
	Change from Naive	1891008	16644096	4729536	39354336
	Predicted change	1870848	16515072	4709376	39223296
	Diff percentage	1.08%	0.78%	0.43%	0.33%
unroll 4	Measured	46368	6711264	46368	16646112
	Change from Naive	6048	9934848	6048	22708224
	Predicted change	-	9676800	-	22450176
	Diff percentage	-	2.63%	-	1.14%
unroll 8	Measured	46368	5114592	46368	12920544
	change from Naive	6048	11531580	6048	26433792
	Predicted change	-	11273472	-	26175744
	Diff percentage	-	2.26%	-	0.98%

developed a Trace Analyzer that can take the output trace from the simulator and generate the dissection of execution cycles and analysis of the code simulated. The analyzer can generate statistics about the total "DLL" related to a certain instruction. For instance, in the example shown in Listing 1.5, the "DLL" latencies caused by the "LDD" instruction are reflected in the STD instruction. we call such latencies "related/associated" to the STD instruction.

6.2 Model Validation

Table 1 shows the change of total "DLL"s for different approaches with the group size set to be one. In the table, for the preloading based approaches (memcpy or inline), the change of the "STD associated DLL latency" is the cost we pay for preloading, as shown in the third and fifth column of this table. The predicted value of this part is computed using (14) for the memcpy approach, and (17) for the "inline" approach. The change of the total "DLL"s in the computation core (inner product and column rotation) is the gain we achieved. Without preloading the equation for this part is (11), with preloading, the number of total "DLL" cycles in this part is approximately zero. Therefore, for two preloading approaches, the equation for the cycles saved in the computation core is (11).

The difference percentage between the measured value from the simulation trace and the predicted value from the equations is computed using the following equation:

$$Diff.Percentage = \frac{|Measurement - Prediction|}{(Measurement + Prediction)/2}. \quad (20)$$

From the table, we can see the predicted value is very close to the measured value and the difference percentage is quite small. The prediction for the "memcpy" approach has a relatively bigger difference percentage since the extra overhead of function calling is not accounted for in our simplified model.

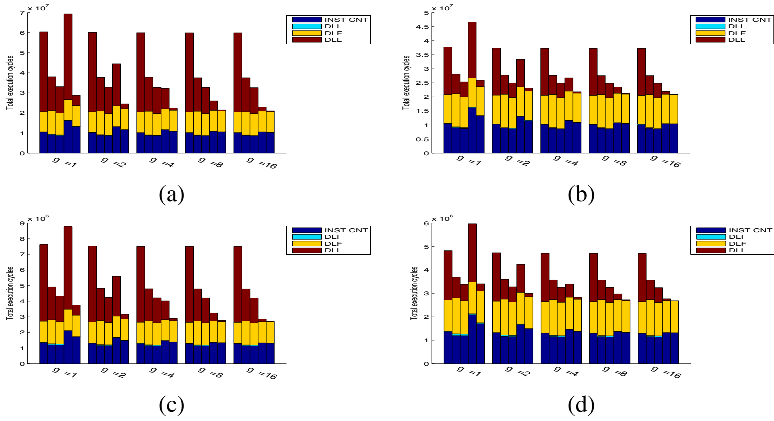

Fig. 2. comparison of different approaches (a)Problem size 64 by 64, L=80 (b) Problem size 64 by 64, L=36, (c) Problem size 32 by 32, L=80 (d) Problem size 32 by 32, L=36

6.3 Comparison of Different Approaches

Figure. 2 shows the comparison of total execution cycles and the dissection to four parts as in (8). Each figure is composed of five clusters of stacked bars. Within each cluster, the leftmost stacked bar is the microlevel breakdown of the naive approach, the second from the left shows the four times unrolling approach, the third one is the eight times unrolling approach, the fourth one is the “memcpy” approach, the fifth one is the “inline” approach. The first cluster shows the five approaches when group size equals one, the second cluster has group size 2, so on so forth. Within each stacked bar, the brown bar (the top bar) shows the total “DLL” cycles, the deep blue bar (the bottom bar) shows the total number of instructions, the light blue bar shows the total “DLI” latency, the yellow bar (in the middle) shows the total “DLF” float point unit latency.

There are several observations from the figures. (1) All the proposed approaches have performance improvement over the naive approach except the “memcpy” method (for group size 1). (2) The figure also shows how the “DLL” cycles change with the increase of the group size. For preloading based approaches (“memcpy” and “inline”), as the group size doubles, the “DLL” will reduce to one half. The loop unrolling based approach does not change with the change of the group size because the loop unrolling based approach only change the inner product routine of the basic rotation routine of two columns and the total number of basic rotations within one sweep is not changed when the group size changes. (3) This figure also shows the total instructions change for different approaches. It can be seen that for preloading based approaches, the total number of instructions increases from the naive approach due to the extra instructions for preloading. On the other hand, the loop unrolling approach can reduce the total instruction count from the naive approach since the loop unrolling reduces the total numbers that the loop control statement are executed. (4) The “DLF” part in the figure roughly does not change no matter what approach we are using. This is true because the “DLF” is related to the floating point instructions in the computation core, which is

kept unchanged. (5) It can be seen the "inline" preloading approach performs the best out of all five approaches.

7 Conclusions

This paper focus on the Cyclops64 computer architecture and presented an analytical model and performance simulation results for the preloading and loop unrolling approach to optimize the performance of SVD benchmark. The major contributions include: (1), We developed a performance model and trace analyzer to dissect the total execution cycles. This model allows us to study the application performance tradeoff for different algorithm or architectural design ideas. (2), We presented a clear understanding of SVD benchmark. (3), We used cycle accurate simulator to validate the model and compare the effect of four approaches on the "DLL" part and the total execution cycle. We find the hand optimized "inline" method can improve the performance significantly and performs best among several approaches. We would like to thank Juan B. del Cuvillo, Fei Chen, Weirong Zhu, and other members in the CAPSL (Computer Architecture and Parallel Systems Laboratory) group for their help.

References

1. C. Cascaval, J. G. C. nos, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J. E. Moreira, K. Strauss, and H. S. W. Jr., "Evaluation of a multithreaded architecture for cellular computing," in *HPCA*, 2002, pp. 311–322.
2. G. Almái, C. Cascaval, J. G. Castaños, M. Denneau, D. Lieber, José E. Moreira, and J. Henry S. Warren, "Dissecting cyclops: a detailed analysis of a multithreaded architecture," *SPECIAL ISSUE: MEDEA workshop*, vol. 31, pp. 26 – 38, 2003.
3. G. S. Almasi, C. Caşcaval, J. E. Moreira, M. Denneau, W. Donath, M. Eleftheriou, M. Giampapa, H. Ho, D. Lieber, D. Newns, M. Snir, and J. Henry S. Warren, "Demonstrating the scalability of a molecular dynamics application on a petaflop computer," in *ICS '01: Proceedings of the 15th international conference on Supercomputing*. New York, NY, USA: ACM Press, 2001, pp. 393–406.
4. J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture," in *Workshop on Modeling, Benchmarking and Simulation (MoBS), held in conjunction with the 32nd Annual Interantional Symposium on Computer Architecture (ISCA'05)*, Madison, Wisconsin, June 4 2005.
5. ——, "Tiny threads: a thread virtual machine for the cyclops64 cellular architecture," in *Fifth Workshop on Massively Parallel Processing (WMPP), held in conjunction with the 19th International Parallel and Distributed Processing System*, Denver, Colorado, April 3 - 8 2005.
6. J. B. del Cuvillo, Z. Hu, W. Zhu, F. Chen, and G. R. Gao, "Toward a software infrastructure for the cyclops64 cellular architecture," 2004, CAPSL Memo 55, Department of ECE, Universisty of Delaware.
7. M. R. Hestenes, "Inversion of matrices by biorthogonalization and related results," *J. Soc. Induct. Appl. Math.*, vol. 6, pp. 51–90, 1958.