

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be “used for any purpose other than private study, scholarship, or research.” If a user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of “fair use” that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select “Pages from: first page # to: last page #” on the print dialog screen

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

HIGH-PERFORMANCE MATRIX MULTIPLICATION ON INTEL AND FPGA PLATFORMS

by
Gang Li

Matrix multiplication is at the core of high-performance numerical computation. Software methods of accelerating matrix multiplication fall into two categories. One is based on calculation simplification. The other one is based on increasing the memory access efficiency. Also matrix multiplication can be accelerated using vector processors. In this investigation, various matrix multiplication algorithms and the vector-based hardware acceleration method are analyzed and compared in terms of performance and memory requirements. Results are shown for Intel and Xilinx FPGA platforms. They show that when the CPU is fast, Goto's algorithm runs faster than Strassen's algorithm because the data access speed is the bottleneck in this case. On the contrary, when the CPU is slow, Strassen's algorithm runs faster because the computation complexity becomes the key factor in this case. Also, the results show that SIMD platforms, such as Intel Xeon and SIMD extensions and an in-house developed VP (Vector co-Processor), for an FPGA, can accelerate matrix multiplication substantially. It is even shown that the VP runs faster than MKL (Intel's optimized Math Kernel Library). This is because not only can the VP take advantage of larger vector lengths but it also minimizes inherent hardware overheads.

**HIGH-PERFORMANCE MATRIX MULTIPLICATION
ON INTEL AND FPGA PLATFORMS**

**by
Gang Li**

**A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering**

Department of Electrical and Computer Engineering

May 2012

Blank Page

APPROVAL PAGE

**HIGH-PERFORMANCE MATRIX MULTIPLICATION ON
INTEL AND FPGA PLATFORMS**

Gang Li

Dr. Sotirios G. Ziavras, Thesis Advisor Date
Professor of Electrical and Computer Engineering, NJIT

Dr. Roberto Rojas-Cessa, Committee Member Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Edwin Hou, Committee Member Date
Associate Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Gang Li
Degree: Master of Science
Date: May 2012

Undergraduate and Graduate Education:

- Master of Science in Computer Engineering,
New Jersey Institute of Technology, Newark, NJ, 2012
- Bachelor of Science in Software Engineering,
Wuhan University of Technology, Wuhan, P. R. China, 2006

Major: Computer Engineering

To my parents and Yayun.

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my advisor, Dr. Sotirios G. Ziavras, for the opportunity to explore the field of high-performance computation. His encouragement, guidance, and support are invaluable in my work. I would also like to thank Dr. Roberto Rojas-Cessa and Dr. Edwin Hou for their support as my professors and committee members. And many thanks to my fellow student, Spiridon F. Beldianu, for his support and inspiration.

TABLE OF CONTENTS

Chapter	Page
1 INTRODUCTION.....	1
2 BRUTE-FORCE IMPLEMENTATION	3
2.1 Introduction	3
2.2 Analysis	4
2.2.1 Time Complexity	4
2.2.2 Cache Performance	4
2.2.3 Memory Consumption	5
2.3 Results	6
2.3.1 Intel Xeon Platform	6
2.3.2 Xilinx ML501 FPGA Platform	7
3 BASIC BLOCK-BASED IMPLEMENTATION	9
3.1 Introduction	8
3.1.1 Calculating the First Three Columns of C	10
3.1.2 Calculating the Other Three Columns of C	11
3.2 Analysis	12
3.2.1 Time Complexity	12
3.2.2 Cache Performance	12
3.2.3 Memory Consumption	13
3.3 Results	13

TABLE OF CONTENTS
(Continued)

Chapter	Page
3.3.1 Intel Xeon Platform	13
3.3.2 Xilinx ML501 FPGA Platform	15
4 GOTO’S IMPLEMENTATION.....	15
4.1 Introduction	16
4.1.1 Goto’s Block-based Method	16
4.1.2 Calculation Process for Goto’s Algorithm	18
4.2 Analysis	20
4.2.1 Time Complexity	20
4.2.2 Cache Performance	20
4.2.3 Memory Consumption	21
4.3 Results	22
4.3.1 Intel Xeon Platform	22
4.3.2 Xilinx ML501 FPGA Platform	23
5 STRASSEN’S IMPLEMENTATION	24
5.1 Introduction	24
5.2 Analysis	27
5.2.1 Time Complexity	27
5.2.2 Cache Performance	27
5.2.3 Memory Consumption	27
5.2.4 Disadvantages	28

TABLE OF CONTENTS
(Continued)

Chapter	Page
5.3 Results	28
5.3.1 Intel Xeon Platform	28
5.3.2 Xilinx ML501 FPGA Platform	30
6 MKL'S IMPLEMENTATION.....	31
6.1 Introduction	31
6.2 Results	32
7 ACCELERATION USING VECTOR CO-PROCESSOR.....	33
7.1 Hardware Architecture.....	33
7.2 Calculation Process.....	34
7.3 Analysis.....	36
7.3.1 Time Complexity.....	36
7.3.2 Memory Consumption.....	36
7.4 Results.....	36
8 PERFORMANCE COMPARISONS	38
8.1 Intel Xeon Platform	38
8.1.1 Optimization Disabled (OD)	38
8.1.2 Full Optimization (O3)	39
8.2 Xilinx ML501 FPGA Platform	39
8.3 MKL vs. VP.....	40
9 CONCLUSIONS	41

Chapter	Page
APPENDIX C SOURCE CODE	42
REFERENCES	75

LIST OF TABLES

Table	Page
2.1 Cache Miss Numbers of Brute-force Implementation for Various Cache Sizes (in Number of Elements).....	5
2.2 The Specifications of the Intel Xeon Platform	6
2.3 Execution Time (in Seconds) of the Brute-force Implementation on the Intel Xeon Platform.....	6
2.4 The Specifications of the Xilinx FPGA Platform.....	7
2.5 Execution Time (in Seconds) of the Brute-force Implementation on the Xilinx FPGA Platform.....	8
3.1 Cache Miss Numbers of Basic Block-based Implementation for Various Cache Sizes (in Number of Elements).....	13
3.2 Execution Time (in Seconds) of the Basic Block-based Implementation on the Intel Xeon Platform.....	14
3.3 Execution Time (in Seconds) of the Basic Block-based Implementation on the Intel Xeon for K=32.....	14
3.4 Execution Time (in Seconds) of the Basic Block-based Implementation on a Xilinx FPGA Platform.....	15
3.5 Execution Time (in Seconds) of the Basic Block-based Implementation on the Xilinx FPGA Platform for K =8.....	15
4.1 Cache Miss Numbers of Goto’s Implementation for Various Cache Sizes (in Number of Elements).....	21
4.2 Execution Time (in Seconds) of the Goto’s Implementation on the Intel Xeon Platform.....	22
4.3 Execution Time (in Seconds) of the Goto’s Implementation on the Intel Xeon for K=32.....	22
4.4 Execution Time (in Seconds) of Goto’s Implementation on the Xilinx FPGA Platform.....	23

LIST OF TABLES
(Continued)

Table	Page
4.5 Execution Time (in Seconds) of Goto’s Implementation on the Xilinx FPGA Platform for $K = 8$	23
5.1 Execution Time (in Seconds) of the Strassen’s Implementation on the Intel Xeon Platform.....	28
5.2 Execution Time (in Seconds) of the Strassen’s Implementation on the Intel Xeon for $K=32$	29
5.3 Execution Time (in Seconds) of Strassen’s Implementation on the Xilinx FPGA Platform.....	30
5.4 Execution Time (in Seconds) of Strassen’s Implementation on the Xilinx FPGA Platform for $K=16$	30
6.1 Execution Time (in Seconds) of the MKL’s MM Implementation on the Intel Xeon Platform.....	32
7.1 Execution Time (in seconds) of Matrix Multiplication on the VP Platform.....	36
7.2 Execution Time (in million clock cycles) of Matrix Multiplication on the VP Platform.....	37
8.1 Execution Time (in Seconds) of All the Implementations on the Intel Xeon Platform with Compiling Optimization Disabled.....	38
8.2 Execution Time (in Seconds) of All the Implementations on the Intel Xeon Platform with Full Optimization (O3) [11].....	39
8.3 Execution Time (in Seconds) of All the Implementations on the Xilinx FPGA Platform with and without the VP	40
8.4 Execution Time (in million clock cycles) of MM using MKL and the VP.....	40

LIST OF FIGURES

Figure	Page
2.1 Process of MM calculation with the Brute-force implementation.....	4
3.1 Calculating a part of the first three elements of the first row of matrix C.....	10
3.2 The first iteration of calculating first three columns of matrix C	10
3.3 Calculating another part of the first three elements of the first row of matrix C ...	11
3.4 The second iteration of calculating the first three columns of matrix C	11
3.5 The first iteration of calculating the remaining three columns of matrix C	11
3.6 The second iteration of calculating the remaining three columns of matrix C	12
4.1 All possible shapes of matrix multiplication [taken from 2].....	16
4.2 All possible methods to break down matrix multiplication [taken from 2].....	17
4.3 Block-based decomposition of matrices A and B	18
4.4 Blocking B1 to calculate C1	19
4.5 Calculate C11	19
4.6 Calculate C12	19
4.7 Adding each layer of C	20
5.1 Sub-matrices of A and B	26
5.2 Calculate intermediate sub-matrices P1 to P7	26
5.3 Calculate matrix C from sub-matrices P1 to P7	26
7.1 Vector processor computing platform architecture [taken from 12].....	33
7.2 Partitioning matrices A and C for VP-based MM.....	34
7.3 C1 is calculated as $A_1 * B$	34

Figure	Page
7.4 Partitioning A1 and B.....	35
7.5 Partitioning of C1.....	35
7.6 Partitioning C11.....	35

CHAPTER 1

INTRODUCTION

The objective of this thesis is to present high-performance matrix multiplication algorithms and a relevant hardware acceleration method. Software methods of accelerating matrix multiplication fall into two categories. One is based on calculation simplification. The other one is based on increasing memory access efficiency. The hardware acceleration is done by using an in-house built vector co-processor for FPGAs.

Strassen's algorithm is a typical algorithm based on calculation simplification. Strassen's algorithm has complexity $O(n^{2.807})$ [1] [5] for $n * n$ matrices. It is a recursive algorithm. First, the input matrix is divided into four sub-matrices for independent multiplications, then recursively into sixteen sub-matrices, etc. But this by itself does not reduce the time complexity which is still $O(n^3)$. However, Strassen found a way to also reduce the complexity of single sub-matrix multiplication. Thus, the time complexity is reduced to $O(n^{2.807})$. The Coppersmith-Winograd algorithm has a time complexity of $O(n^{2.3737})$ [4]. However, this algorithm has a very large constant, so it is only useful for the multiplication of extremely large matrices. The lower bound is $O(n^2)$, i.e., the same as the number of elements in the product.

A block-based matrix multiplication method is based on increasing memory access efficiency. It calculates the resulting matrix block by block instead of line by line (row or column), most of the time, in order to keep the data needed small enough to fit in the cache and thus take advantage of cache hits.

Besides elaborately identifying blocks in the matrix, the Goto's method [2] increases the memory access efficiency further by copying the most frequently used data into contiguous memory locations in order to reduce the TLB misses.

MKL is the Math Kernel Library developed by Intel [10]. It heavily uses the Intel architecture's SSE instruction extensions to do the computations in parallel in the SIMD (Single Instruction Multi Data) mode.

The vector processor is an efficient implementation of an SIMD architecture for array operations. It can simultaneously execute the same operation, e.g. single-precision floating-point multiplication, on all the elements in an array.

The rest of the thesis introduces the details of the studied algorithms or methods and presents their implementations on Intel and FPGA platforms. Then, it compares the results.

CHAPTER 2

BRUTE-FORCE IMPLEMENTATION

Brute-force matrix multiplication (MM) is implemented exactly according to the matrix multiplication definition. It is simple and straight forward and provides a baseline in order to facilitate comparisons with other MM algorithms.

2.1 Introduction

The definition of matrix multiplication is: for $N * N$ matrices A and B, the result of their multiplication is matrix C whose elements are:

$$C[i][j] = \sum_{k=0}^{N-1} A[i][k] * B[k][j]$$

for $i, j = 0, \dots, N-1$

It could be easily implemented using there nested for loops as follows:

```
for( i = 0; i < N; i++)
  for( j = 0; j < N; j++ ) {
    sum=0;
    for( k = 0; k < N; k++ )
      sum += A[i][k] * B[k][j];
    C[i][j] = sum;
  }
```

The process of calculating the result is shown in Figure 2.1 for 6 by 6 matrices. Each square represents an element of a matrix. Each element of C is calculated by multiplying corresponding elements from one row of A with one column of B.

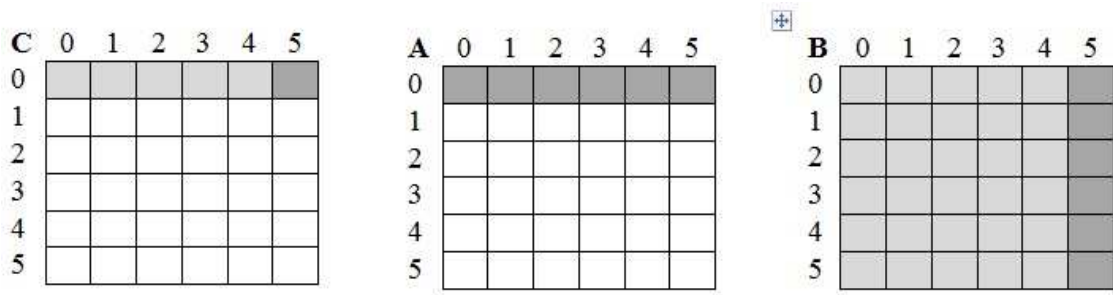


Figure 2.1 Process of MM calculation with the Brute-force implementation. White means the data has not been accessed; light gray means older accesses; and dark gray means current accesses.

2.2 Analysis

2.2.1 Time Complexity

To calculate one element of C, there are N multiplications and N additions. In total, there are N^3 multiplications and N^3 additions. So the time complexity of calculating matrix C is $O(N^3)$.

2.2.2 Cache Performance

Consider only cache capacity misses and compulsory misses for simplifying the analysis (i.e. ignore conflict misses). Capacity misses are those misses that occur regardless of the associativity or the block size, solely due to the finite size of the cache. Compulsory misses are those misses caused by the first reference to a datum. Conflict misses are those misses that could have been avoided, had the cache not evicted an entry earlier.

For the Brute-force implementation, Table 2.1 shows the absolute cache miss numbers for various cache sizes. An explanation for cache size $> (L+1)N$ follows as an example. In this case, one row of A and L columns of B could be held in the cache. In

order to calculate one row of C, a row of A is repeatedly accessed and there are N/L misses. So to calculate the whole matrix C, there will be N^2/L misses. At the same time, when calculating one row of C, N^2/L misses will occur for scanning the whole matrix B. So to calculate the whole matrix C, there will be N^3/L misses.

As for the cache organization, if it is directly mapped, there will be more cache conflict misses than in the case of set associative.

Table 2.1 Cache Miss Numbers of Brute-force Implementation for Various Cache Sizes (in Number of Elements)

Cache Size \ Matrix Size	$> (N+1)N$	$> (L+1)N$	$> (N+L)$	$> 2L$	0
A	N^2/L	N^2/L	N^2/L	N^3/L	N^3
B	N^2/L	N^3/L	N^3	N^3	N^3
C	N^2	N^2	N^2	N^2	N^2
Subtotal	$N^2 + 2N^2/L$	$N^2 + N^3/L + N^2/L$	$N^3 + N^2 + N^2/L$	$N^3 + N^2 + N^3/L$	$2N^3 + N^2$

2.2.3 Memory Consumption

In terms of memory consumption, the Brute-force implementation does not need extra memory but just memory to store the three matrices; this requires the storage of $3N^3$ elements.

2.3 Results

2.3.1 Intel Xeon Platform

The specifications of the Intel platform are shown in Table 2.2. It has a dual-core CPU and each core has two threads. In order to analyze the algorithms, only one thread is used.

The running time of the implementation for various matrix sizes is shown in Table 2.3. This shows how much more time is needed for the calculation when N doubles. The time complexity is $O(N^3)$, but Table 2.3 shows that the slowdown is not always 8 when N doubles. This is because the cache performance and the constants in the complexity affect the time spent.

Table 2.2 The Specifications of the Intel Xeon Platform

CPU	Xeon 3.20G Hz * 2
Memory	3GB
L1 cache	16KB, 8-way 64-byte line size
L2 cache	1024KB, 8-way 64-byte line size
Compiler	Intel c/c++ compiler
Compile option	-OD(optimization disabled)

Table 2.3 Execution Time (in Seconds) of the Brute-force Implementation on the Intel Xeon Platform

Brute-force Implementation						
Matrix Size	64	128	256	512	1024	2048
Time Spent (sec)	0.002336	0.029708	0.253991	2.684	63.753	537.389
Slowdown	NA	12.717	8.549	10.608	23.752	8.429

When $N \leq 64$, matrix B could fully fit in the L1 cache (16KB). The cache misses become $N^2 + 2N^2/L$. If N is larger than 64, the cache misses are $N^2 + N^2/L + N^3/L$. That is why the slowdown for $N = 128$ is larger than 8.

When $N=128$ or 256 , matrix B could fit in L2 cache, therefore the cache miss rate for $N=128$ or 256 are similar. In this case the slowdown is depended on computation complexity, thus the slowdown for $N=256$ is close to 8.

When $N=512$, the L2 cache could exactly hold matrix B (no more place for one row of matrix A), this makes the L2 cache miss rate higher than in the case of $N=256$. So the slowdown is slightly higher than 8.

When $N \geq 1024$, the L2 cache (1024KB) is not large enough to hold matrix B, so the cache miss rate increases. This causes the slowdown (23.752) for $N = 1024$ to be larger than 8. And because the speed gap between the L2 cache and main memory is wide, the slowdown (23.752) is so large.

2.3.2 Xilinx ML501 FPGA Platform

The platform's specifications are shown in Table 2.4. The Xilinx MicroBlaze processor was used. It contains the XC5VLX50 FPGA and runs at 125 MHz.

Table 2.4 The Specifications of the Xilinx FPGA Platform

CPU	Microblaze 125MHz
Memory	256MB
L1 cache	8KB, 1-way 32-byte line size
Compiler	GNU c compiler
Compile option	-OD(optimization disabled)

The running time of the implementation for various matrix sizes is shown in

Table 2.5.

Table 2.5 Execution Time (in Seconds) of the Brute-force Implementation on the Xilinx FPGA Platform

Brute-force Implementation					
Matrix Size	64	128	256	512	1024
Time Spent (sec)	0.105	0.841	6.717	53.664	429.12
Slowdown	NA	8.009	7.986	7.989	7.996

It is shown that the slowdowns are all close to 8. This is because the cache (8KB) is too small to hold even the 64 by 64 matrix, which means that for all the cases the cache performance is quite similar. Thus, the slowdown is depended on the computation complexity.

CHAPTER 3

BASIC BLOCK-BASED IMPLEMENTATION

The Brute-force implementation repeatedly accesses the whole matrix B, column by column. If matrix B cannot fit in the cache, the cache miss rate increases. The cost of the cache is much higher than that of the memory, so it cannot be too large. The basic Block-based method provides a way to access matrix B block by block instead of scanning the whole matrix. In this way, a small cache could hold all the needed data in each iteration, therefore the cache miss rate decreases even when the matrices are large.

3.1 Introduction

The chosen Block-based implementation calculates one row of C part by part. The process is rather complicated and it will be shown in pictures in the following discussion.

The basic Block-based algorithm implementation [1] in the C language is:

```
for (jj=0; jj<N; jj=jj+K)
  for (kk=0; kk<N; kk=kk+K)
    for (i=0; i<N; i=i+1)
      for (j=jj; j<min(jj+K,N); j++) {
        sum=0.0;
        for (k=kk; k<min(kk+K,N); k++)
          sum+=A[i][k] * B[k][j];
        C[i][j] += sum;
      }
```

There are five “for loops” assuming blocks of size $K \times K$. The following pictures illustrate the process for 6×6 matrices.

3.1.1 Calculating the First Three Columns of C

In the first iteration:

$$C[i][j] = A[i][0] * B[0][j] + A[i][1] * B[1][j] + A[i][2] * B[2][j]$$

Calculating a part of the first row of C is shown in Figure 3.1. A light shade means an older access and a dark shade means a current access.

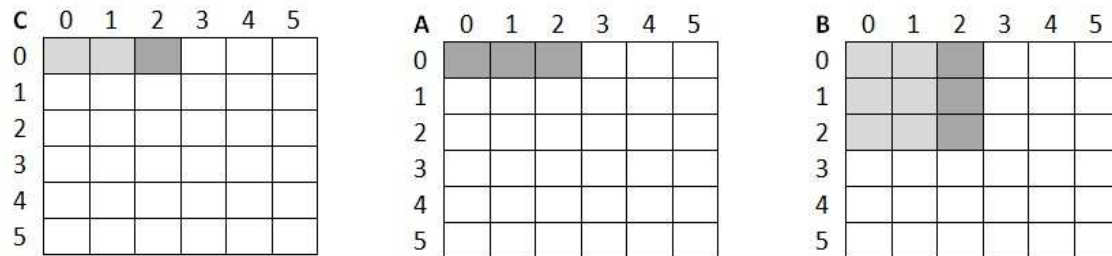


Figure 3.1 Calculating a part of the first three elements of the first row of matrix C.

The elements accessed in the whole iteration are shown in Figure 3.2.

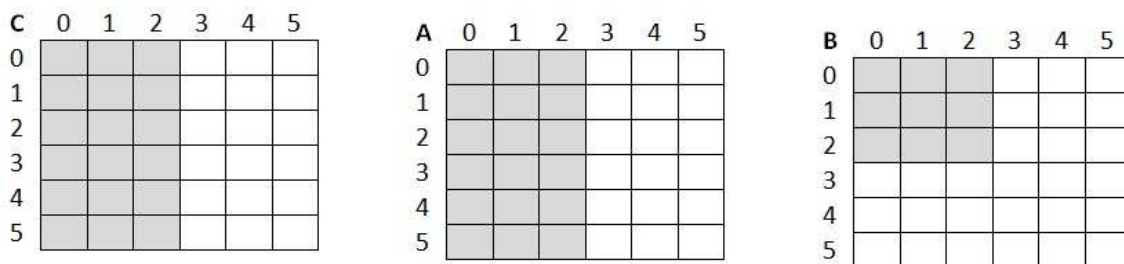


Figure 3.2 The first iteration of calculating the first three columns of matrix C. White means the data has not been accessed; light gray means completed accesses.

After the shown iteration, the calculation of the first 3 columns of C is not completed yet. Only one “layer” of the calculation is finished, which means that only some summations have been completed.

In the second iteration:

$$C[i][j] = A[i][3] * B[3][j] + A[i][4] * B[4][j] + A[i][5] * B[5][j]$$

Calculating a part of one row of C is shown in Figure 3.3.

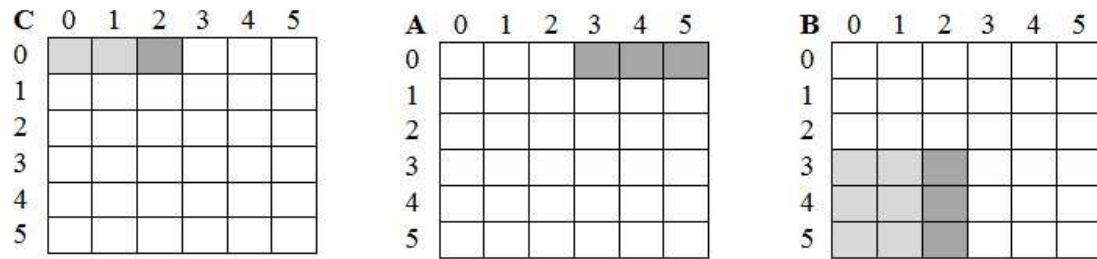


Figure 3.3 Calculating another part of the first three elements of the first row of matrix C.

The data accessed in the whole iteration is shown in Figure 3.4.

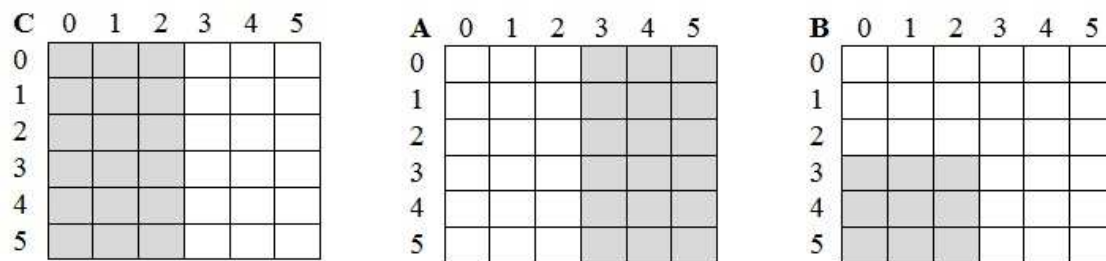


Figure 3.4 The second iteration of calculating the first three columns of matrix C. White means the data has not been accessed; light gray means completed accesses.

3.1.2 Calculating the Other Three Columns of C

In the first iteration:

$$C[i][j] = A[i][0] * B[0][j] + A[i][1] * B[1][j] + A[i][2] * B[2][j]$$

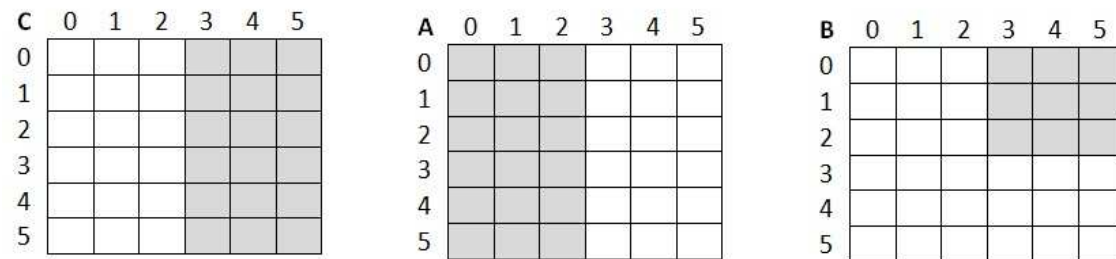


Figure 3.5 The first iteration of calculating the remaining three columns of matrix C. White means the data has not been accessed; light gray means completed accesses.

In the second iteration:

$$C[i][j] = A[i][3] * B[3][j] + A[i][4] * B[4][j] + A[i][5] * B[5][j]$$

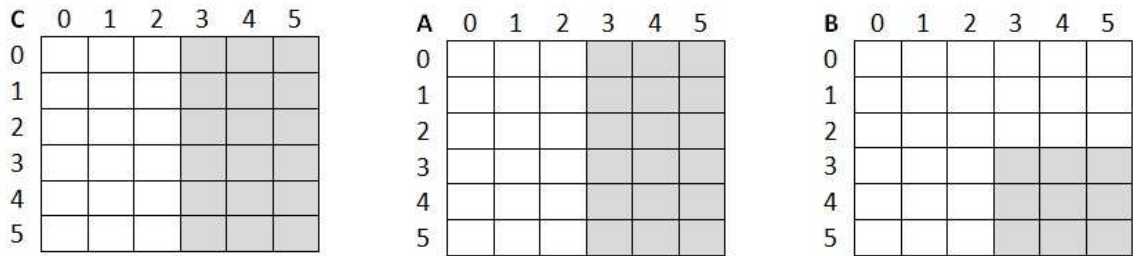


Figure 3.6 The second iteration of calculating the remaining three columns of matrix C. White means the data has not been accessed; light gray means completed accesses.

3.2 Analysis

3.2.1 Time Complexity

There are five “for” loops in this implementation.

$$\begin{aligned} \text{Total number of multiplications} &= N/K * N/K * N * K * K \\ &= N^3 \end{aligned}$$

where $K * K$ is the block size in matrix B used in each iteration.

So the time complexity of the basic Block-based implementation is $O(N^3)$.

3.2.2 Cache Performance

When $K < L$ (cache line size), the data stored in the cache will not be fully used, which is not efficient and will not be discussed here.

When $K \geq L$, Table 3.1 shows the cache misses for different scenarios.

Take the case of cache size $> (2NK + K^2)$ as an example to explain the cache misses. In this case, K columns of matrices C and A as well as K^2 elements of matrix B can be in the cache. To calculate K columns of matrix C, the cache misses are

$$(K/L) * N = NK/L$$

There are N/K calculations of this type to produce the whole matrix C , so the cache misses are

$$(NK/L) * N/K = N^2/L$$

Table 3.1 Cache Miss Numbers of Basic Block-based Implementation for Various Cache Sizes (in Number of Elements)

Cache Size \ Matrix Size	$>(2NK+K^2)$	$> K^2+2K$	$> K^2+K$	0
A	N^2/L	N^3/KL	N^3/KL	N^3
B	N^2/L	N^2/L	N^2/L	N^3
C	N^2/L	N^3/KL	N^3/K	N^3/K
Subtotal	$3N^2/L$	$2N^3/KL$ $+N^2/L$	$N^3/K+N^3/KL$ $+N^2/L$	$2N^3$ $+N^3/K$

3.2.3 Memory Consumption

No extra memory is needed other than storing matrices A , B and C , so the storage needed is $3N^3$.

3.3 Results

3.3.1 Intel Xeon Platform

Table 3.2 shows the time needed for calculating matrices of various sizes for the basic Block-based implementation.

The cache line size is 64 bytes. One cache line stores 16 floating point numbers. As discussed above, when $K < L$ memory accesses are less efficient. This is verified in Table 3.2.

Table 3.2 Execution Time (in Seconds) of the Basic Block-based Implementation on the Intel Xeon Platform

Basic Block-based Implementation						
Matrix Size Block Size	64	128	256	512	1024	2048
4	0.004130	0.051724	0.298041	2.895	36.908	297.644
8	0.004972	0.029490	0.253050	2.102	19.530	157.101
16	0.002773	0.040463	0.236596	1.817	15.030	120.654
32	0.002677	0.043091	0.227063	1.683	13.470	108.300
64	0.002651	0.037837	0.206537	1.572	15.737	126.023

As shown in Table 3.2, for large matrices the calculation is the most efficient for block sizes 32*32. Table 3.3 takes this case as an example to further illustrate the effect of the cache as a function of the matrix size.

Table 3.3 Execution Time (in Seconds) of the Basic Block-based Implementation on the Intel Xeon for K=32

Basic Block-based Implementation						
Matrix Size	64	128	256	512	1024	2048
Time Spent (sec)	0.002677	0.043091	0.227063	1.683	13.470	108.300
Slowdown	NA	16.096	5.269	7.412	8.003	8.040

When N increases, less data can fit in the cache. As shown in Table 3.1, when cache size $< (2NK + K^2)$, the cache misses will increase. For K=32 and $N > 64$

$$16K < 2 * N * 32 * 4 + 32 * 32 * 4$$

This is verified in Table 3.3 for N = 128, where the slowdown is much larger than 8. Compared to the Brute-force implementation for $N > 256$, the L1 cache could not hold $(N+1)L$ data and cache misses increased. But in the basic Block-based implementation it is easy to hold $K^2 + 2K$ data, and the cache miss rate is kept at a low level. It is verified

from Table 2.3 and Table 3.3 that, for $N > 256$, the basic Block-based implementation takes less time.

3.3.2 Xilinx ML501 FPGA Platform

Table 3.4 shows the time needed for calculating matrices of various sizes for the basic Block-based implementation on Xilinx FPGAs.

Table 3.4 Execution Time (in Seconds) of the Basic Block-based Implementation on a Xilinx FPGA Platform

Basic Block-Based Implementation					
Matrix Size Block Size	64	128	256	512	1024
4	0.066	0.549	4.771	38.322	318.234
8	0.054	0.457	4.029	32.648	269.376
16	0.050	0.425	7.126	58.740	481.562
32	0.049	0.866	6.914	56.025	475.860
64	0.107	0.852	6.809	55.855	440.815

Take $K = 8$ as an example. Table 3.5 shows a comparison of execution times.

Table 3.5 Execution Time (in Seconds) of the Basic Block-based Implementation on the Xilinx FPGA Platform for $K = 8$

Basic Block-Based Implementation					
Matrix Size	64	128	256	512	1024
Time Spent (sec)	0.054	0.457	4.029	32.648	269.376
Time Ratio	NA	8.462	8.816	8.103	8.250

The slowdown converges to 8. This is because, as analyzed before, the basic block-based implementation's time complexity is $O(N^3)$. So when N doubles, the execution time becomes about 8 times as much.

CHAPTER 4

GOTO'S IMPLEMENTATION

The Goto's implementation[3] not only decomposes the matrices into blocks in order to reduce the cache misses but also takes into account TLB misses. The results show that the Goto's implementation has better performance than the basic Blocked-based implementation.

4.1 Introduction

4.1.1 Goto's Block-based Method

Figure 4.1 shows all possible cases of matrix multiplication for matrices A and B having sizes $m*k$ and $k*n$, respectively, according to the Goto's classification.

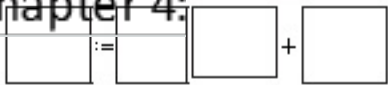

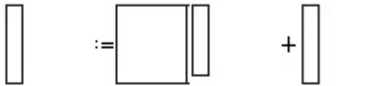
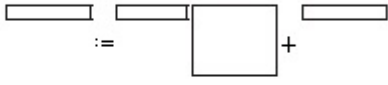




m	n	k	Illustration	Label
large	large	large		GEMM
large	large	small		GEPP
large	small	large		GEMP
small	large	large		GEPM
small	large	small		GEPP
large	small	small		GEPE
small	small	large		GEPDOT
small	small	small		GEBE

Figure 4.1 All possible shapes of matrix multiplication [taken from 2].

Goto's algorithm tries to find the best way to divide the matrices into blocks. All possible block-based approaches are shown in Figure 4.2.

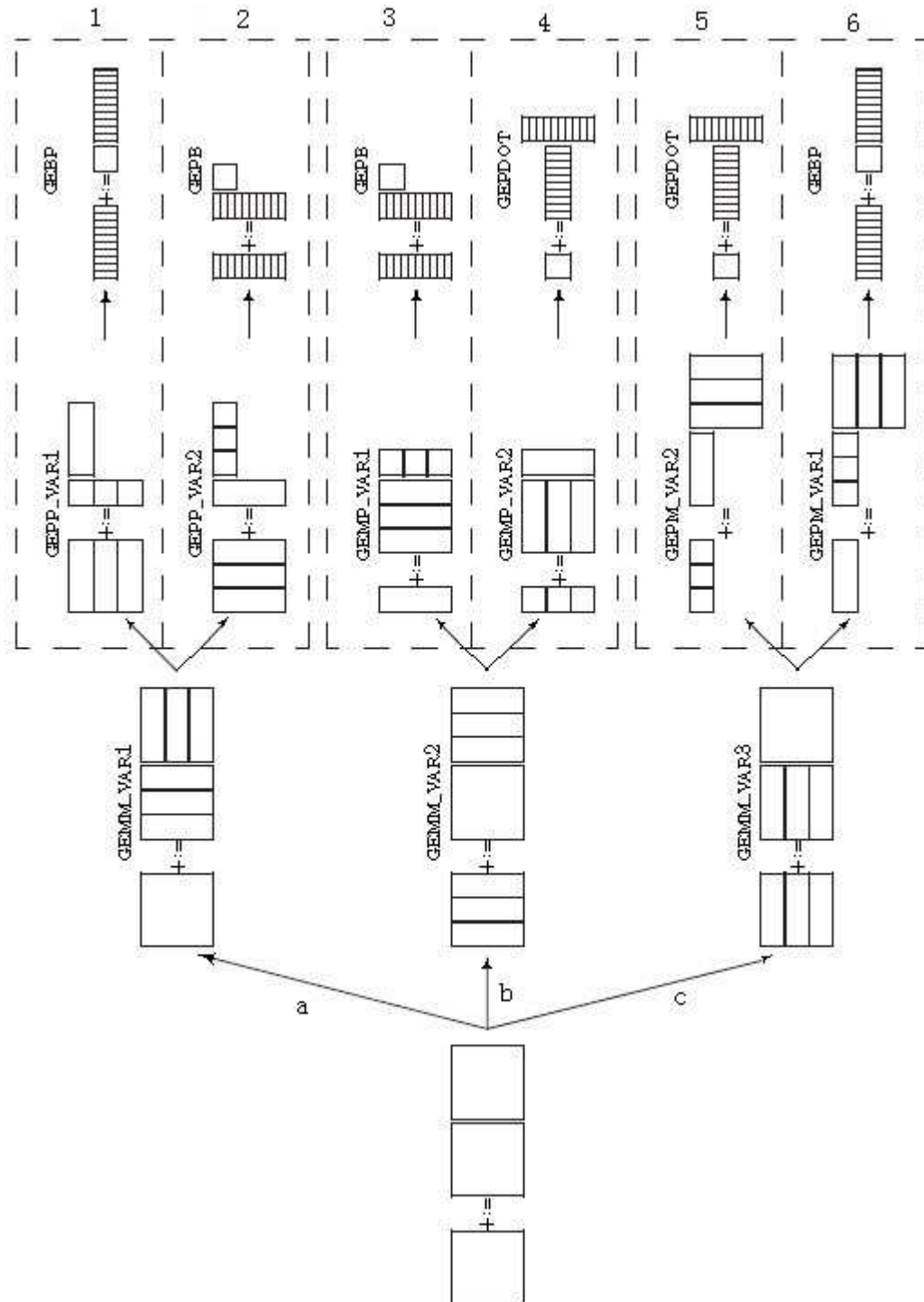


Figure 4.2 All possible methods to break down matrix multiplication [taken from 2].

Goto's algorithm chooses the number 2 method to implement the matrix multiplication if the matrix is stored in the row-major order.

In Figure 4.2, cases 1, 4, 5 and 6 are not TLB friendly in that there are horizontal panels (rectangle shape matrix). Every two adjacent accesses of the elements of a horizontal panel have a gap of N elements in the memory. This means, when N is large every access will cause a TLB miss if there is a cache miss first.

Now case 2 and case 3 will be compared. It is observed that, for case 2 in order to calculate a layer of C , K columns of matrix A are repeatedly accessed. This gives better cache performance especially when K columns of A could fit in the L2 cache. For case 3, the whole matrix A is accessed in each outer loop, so the chances of reducing the cache miss rate for accessing matrix A is relatively low.

4.1.2 Calculation Process for Goto's Algorithm

First, assume the block-based decomposition of matrices A and B as shown in Figure 4.3,

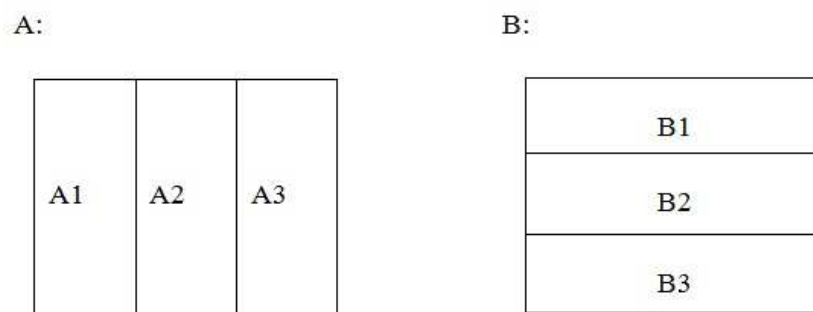


Figure 4.3 Block-based decomposition of matrices A and B .

Second, calculate $C1$ which is the first layer of summations of each element in matrix C (shown in Figure 4.4).

C1:

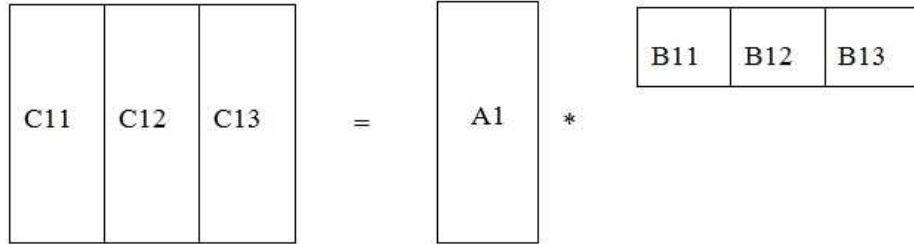


Figure 4.4 Blocking B1 to Calculate C1.

Third, calculate C1 block by block (shown in Figure 4.5 and Figure 4.6),



Figure 4.5 Calculate C11.

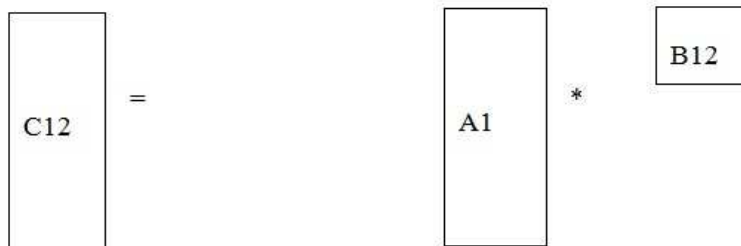


Figure 4.6 Calculate C12.

It is noticed that B11 is not stored in contiguous memory. By adjusting the block size, the cache miss rate could be reduced, but cache misses could not be avoided completely because of conflict and capacity misses. When a miss happens in this case, the system will access the TLB table. Because B11 is not in contiguous memory, the

possibility of having a TLB miss is high. And the cost of a TLB miss is high. Therefore, B11 is copied into contiguous memory in order to reduce cache and TLB misses.

Finally, every layer of C is accumulated to produce the result matrix C (shown in Figure 4.7).

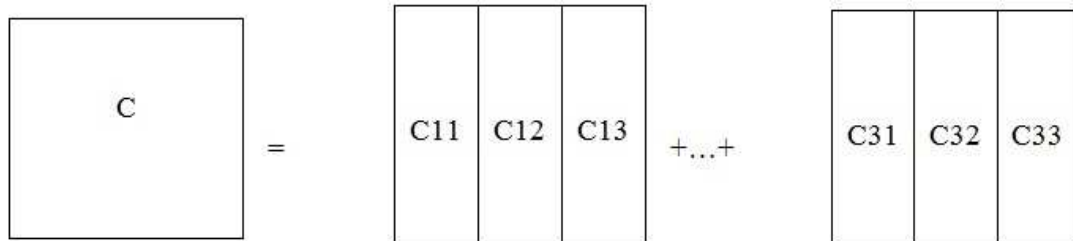


Figure 4.7 Adding each layer of C.

4.2 Analysis

4.2.1 Time Complexity

It is observed that the number of element multiplications is not reduced. Goto's algorithm only changes the order of multiplications. So the time complexity of Goto's implementation is $O(N^3)$.

4.2.2 Cache Performance

Table 4.1 shows the cache misses for various scenarios.

Table 4.1 Cache Miss Numbers of Goto's Implementation for Various Cache Sizes (in Number of Elements)

Cache Size / Matrix Size	$>(N^2+NK+K^2)$	$>NK+K^2$	$> K^2+2K$	$> K^2+K$	0
A	N^2/L	N^2/L	N^3/KL	N^3/KL	N^3
B	N^2/L	N^2/L	N^2/L	N^2/L	N^3
C	N^2/L	N^3/KL	N^3/KL	N^3/K	N^3/K
Subtotal	$3N^2/L$	$N^3/KL+$ $2N^2/L$	$2N^3/KL$ N^2/L	$N^3/K+$ $N^3/KL+N^2/L$	$2N^3$ $+N^3/K$

Take the case of cache size $> (K^2 + 2K)$ as an example to explain the cache miss calculation. In this case, K elements of C , K elements of A and $K * K$ elements of B are in the cache. To calculate one layer of C , the cache misses for matrix B are:

$$(K^2/L) * N/K = NK/L$$

There are N/K layers of C to be calculated, so the total number of cache misses for matrix B is:

$$(NK/L) * N/K = N^2/L$$

4.2.3 Memory Consumption

No extra memory is needed other than storing matrices A , B and C , so the storage needed is $3N^3$.

4.3 Results

4.3.1 Intel Xeon Platform

Table 4.2 shows the time needed for calculating matrices of various sizes for Goto's implementation.

Table 4.2 Execution Time (in Seconds) of the Goto's Implementation on the Intel Xeon Platform

Goto's Implementation						
Matrix Size Block Size	64	128	256	512	1024	2048
4	0.002912	0.031643	0.196508	1.492234	11.565	90.610
8	0.003932	0.022385	0.191102	1.453994	11.486	90.126
16	0.003226	0.025004	0.200335	1.466154	11.504	90.230
32	0.002824	0.022765	0.196912	1.457312	11.495	90.709
64	0.002831	0.027742	0.190376	1.445694	11.480	90.626

The performance is overall stable and better than that for the basic Block-based implementation. Table 4.3 shows the slowdown as a function of the matrix size for 32*32 blocks.

Table 4.3 Execution Time (in Seconds) of the Goto's Implementation on the Intel Xeon for K=32

Goto's Implementation						
Matrix Size	64	128	256	512	1024	2048
Time Spent (sec)	0.002824	0.022765	0.196912	1.457312	11.495	90.709
Slowdown	NA	8.061	8.649	7.400	7.887	7.891

The slowdown is always around 8. This is because the time complexity is $O(N^3)$ which is analyzed in Section 4.2.1.

4.3.2 Xilinx ML501 FPGA Platform

Table 4.4 shows results for the Goto's implementation on a MicroBlaze processor embedded in a Xilinx FPGA.

Table 4.4 Execution Time (in Seconds) of Goto's Implementation on the Xilinx FPGA Platform

Goto's Implementation					
Matrix Size Block Size	64	128	256	512	1024
4	0.055	0.476	4.410	34.452	270.092
8	0.054	0.472	4.335	32.975	264.281
16	0.054	0.468	4.639	36.827	295.163
32	0.054	0.517	4.625	38.321	317.821
64	0.075	0.516	4.618	37.385	300.296

The performance is relatively stable and better than that of the basic Block-based implementation. Take block size = 8 as an example to examine the slowdowns in Table 4.5.

Table 4.5 Execution Time (in Seconds) of Goto's Implementation on the Xilinx FPGA Platform for K =8

Goto's Implementation					
Matrix Size	64	128	256	512	1024
Time Spent (sec)	0.054	0.472	4.335	32.975	264.281
Slowdown	NA	8.740	9.184	7.606	8.014

The slowdown is close to 8.

CHAPTER 5

STRASSEN'S IMPLEMENTATION

The previous algorithms all have time complexity of $O(N^3)$. Strassen's algorithm has time complexity of $O(N^{2.807})$. It is a recursive algorithm and in each iteration it divides each matrix into four sub-matrices. The result will be calculated by sub-matrix multiplications.

5.1 Introduction

First, matrix multiplication could be implemented recursively. For example, A, B and C are $N \times N$ matrixes and $C = A * B$.

$$C = \begin{pmatrix} r & s \\ t & u \end{pmatrix}, A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

The sub-matrices of C could be calculated using the sub-matrices of A and B as follows:

$$r = ae + bg$$

$$s = af + bh$$

$$t = ce + dg$$

$$u = cf + dh$$

There are 8 sub-matrix multiplications. Each multiplication is done in the same way until the sub-matrix contains only one element.

The time complexity is:

$$T(n) = 8T(N/2) + O(N^2)$$

Resolving the recurrence, it gives us:

$$T(N) = O(N^3)$$

The time complexity of the recursive version of matrix multiplication is still $O(N^3)$. However, Strassen found a way to reduce one sub-matrix multiplication in each iteration. The process is as follows:

1) Calculate s:

$$\text{let } P_1 = a(f - h) = af - ah$$

$$\text{let } P_2 = (a + b)h = ah + bh$$

$$s = P_1 + P_2 = af + bh$$

2) Calculate t:

$$\text{let } P_3 = (c + d)e = ce + de$$

$$\text{let } P_4 = d(g - e) = dg - de$$

$$t = P_3 + P_4 = ce + dg$$

3) Calculate r:

$$\text{let } P_5 = (a + d)(e + h) = ae + ah + de + dh$$

$$\text{let } P_6 = (b - d)(g + h) = bg + bh - dg - dh$$

$$r = P_5 + P_4 - P_2 + P_6 = ae + bg$$

4) Calculate u:

$$\text{let } P_7 = (a - c)(e + f) = ae + af - ce - cf$$

$$u = P_5 + P_1 - P_3 - P_7 = cf + dh$$

P1 to P7 are intermediate sub-matrices. They are produced by 7 sub-matrix multiplications.

The process is shown in Figure 5.1 to Figure 5.3.

A:

a	b
c	d

B:

e	f
g	h

Figure 5.1 Sub-matrices of A and B.

Seven intermediate sub-matrices are produced:

P1	P2	P3	P4	P5	P6	P7
a(f-h)	(a+b)h	(c+d)e	d(g-e)	(a+d)(e+h)	(b-d)(g+h)	(a-c)(e+f)

Figure 5.2 Calculate intermediate sub-matrices P1 to P7.

To calculate the result matrix C:

C:

$P_5 + P_4 - P_2 + P_6$	$P_1 + P_2$
$P_3 + P_4$	$P_5 + P_1 - P_3 - P_7$

Figure 5.3 Calculate matrix C from sub-matrices P1 to P7.

5.2 Analysis

5.2.1 Time Complexity

There are seven multiplications of sub-matrices in each iteration, so

$$T(N) = 7T(N/2) + O(N^2)$$

Resolving the recurrence, we get

$$T(N) = O(N^{\lg_2 7}) = O(N^{2.807})$$

5.2.2 Cache Performance

It is observed from the process followed by Strassen's algorithm that the memory accesses are quite scattered, so the cache performance is not good.

5.2.3 Memory Consumption

Strassen's is a recursive algorithm. In iteration i , except the last one, it needs 17 intermediate $N/2^i$ by $N/2^i$ matrices. When the function returns, the intermediate memory will be freed. There are $\log_2^{N/K}$ iterations. Therefore, the memory needed could be calculated as follows:

$$\begin{aligned} \text{Memory} &= 3N^2 + 17((N/2)^2 + (N/4)^2 + \dots + K^2) \\ &= 3N^2 + (17N^2/3)(1-(K/N)^2) \\ &\quad \text{(in elements)} \end{aligned}$$

When K/N is small:

$$\begin{aligned} \text{Memory} &= 3N^2 + (17N^2/3) \\ &= 8.7 * N^2 \end{aligned}$$

So it needs 2.9 times the memory of the previous algorithms.

5.2.4 Disadvantages

Strassen's algorithm does not have stable performance. If N is not a power of 2, matrices A , B and C will be padded to make their sizes powers of 2. This means extra memory and computing time. In the worst case, N increases by 1, the computing complexity increases six times and the memory consumption increases three times.

5.3 Results

In the actual implementation, it was found that it is inefficient for the algorithm to go recursively down to a sub-matrix with one element. So a minimum block size is defined. If the sub-matrix is smaller than the minimum block, the matrix multiplication is implemented using the Brute-force algorithm. Various minimum block sizes were tried and the performance of the algorithm is shown in the following sections.

5.3.1 Intel Xeon platform

Table 5.1 shows the time needed for calculating matrices of various sizes for Strassen's algorithm.

Table 5.1 Execution Time (in Seconds) of the Strassen's Implementation on the Intel Xeon Platform

Strassen's Implementation						
Matrix Size Block Size	64	128	256	512	1024	2048
4	0.007765	0.058314	0.389617	2.702897	19.372	132.834
8	0.004404	0.038448	0.251321	1.708744	11.963	82.893
16	0.003736	0.027557	0.213323	1.461209	10.102	70.350
32	0.005010	0.037459	0.202927	1.416484	10.004	69.778
64	0.004127	0.028823	0.208687	1.499320	10.429	72.877

The performance is better than Goto's algorithm. Take block size = 32 as an example to examine the figures for slowdowns and memory consumptions, as shown in Table 5.2.

Table 5.2 Execution Time (in Seconds) of the Strassen's Implementation on the Intel Xeon for K=32

Strassen's Implementation						
Matrix Size	64	128	256	512	1024	2048
Time Spent (sec)	0.005010	0.037459	0.202927	1.416484	10.004	69.778
Slowdown	-	7.476	5.417	6.980	7.062	6.975
Memory Consumption	-	-	1.084M	6.876M	31.120M	121.8M
Size of three Matrices	-	-	0.768M	3M	12M	48M
Memory Expansion	-	-	1.41	2.29	2.59	2.54

The slowdown is close to 7 independent of the matrix size. It is consistent with the time complexity of $O(N^{2.807})$.

The memory expansion is defined as follows:

$$\text{Memory expansion} = \frac{\text{Memory consumption}}{\text{Size of three matrices}}$$

The memory expansions observed in Table 5.2 have values close to but less than the theoretical 2.9. This is because the calculation in the previous section does count other memory consumptions, like local variables.

5.3.2 Xilinx ML501 FPGA Platform

Table 5.3 shows the time needed for calculating matrices of various sizes with Strassen's algorithm.

Table 5.3 Execution Time (in Seconds) of Strassen's Implementation on the Xilinx FPGA Platform

Strassen's Implementation					
Matrix Size Block Size	64	128	256	512	1024
4	0.074	0.560	4.069	29.075	210
8	0.056	0.436	3.202	23.011	170
16	0.053	0.410	3.038	21.855	155
32	0.052	0.441	3.234	23.224	165
64	0.127	0.935	6.691	51	334

The performance is better than Goto's algorithm.

When the block size is 16*16, the slowdowns are shown in Table 5.4.

Table 5.4 Execution Time (in Seconds) of Strassen's Implementation on the Xilinx FPGA Platform for K=16

Strassen's Implementation					
Matrix Size	64	128	256	512	1024
Time Spent (sec)	0.053	0.410	3.038	21.855	155
Slowdown	NA	7.735	7.409	7.193	7.092

The slowdown is always close to 7. It is consistent with the theoretical analysis.

CHAPTER 6

MKL IMPLEMENTATION

6.1 Introduction

MKL is Intel's Math Kernel Library [10]. It is an optimized library for math. There are several aspects of optimization.

- 1) Multithreading. MKL puts emphasis on multithreaded optimization for multicores.
- 2) SIMD instructions. Execute in parallel using Intel's SIMD instruction extensions (SSE) which operate on eight 128-bit vector registers.
- 3) Assembly. Writing kernel functions in assembly. Carefully arrange instructions to reduce stalls.
- 4) Cache. Increase cache performance by blocking in order to improve both the spatial and temporal localities for better data accesses.

6.2 Results

Table 6.1 shows the execution results for MKL's MM implementation

Table 6.1 Execution Time (in Seconds) of the MKL's MM Implementation on the Intel Xeon Platform

MKL's MM Implementation							
Matrix Size	64	128	256	512	1024	2048	4096
Time Spent (sec)	0.027051	0.001231	0.004310	0.030382	0.328	2.516	20.00
Slowdown	NA	0.045	3.501	7.049	10.7	7.7	7.9
Memory Consumption (MB)	14.500	14.548	14.748	14.992	15.000	64.2	212.2
Size of the 3 Matrices (MB)	0.048	0.192	0.768	3	12	48	192
Memory Expansion	302.08	75.77	19.2	5.0	1.25	1.34	1.11

The slowdown keeps getting closer to 8 with increases in the matrix size. And the memory expansion becomes close to 1. This implies that MKL is not using Strassen's algorithm but a block-based algorithm, otherwise the memory expansion will not be close to 1.

CHAPTER 7

ACCELERATION USING VECTOR CO-PROCESSOR

7.1 Hardware Architecture

Figure 7.1 shows an in-house developed (at CAPPL laboratory) vector co-processor (VP) computing platform [12] [13]. The scalar CPU is a Xilinx MicroBlaze (125MHz). The CPU issues vector instructions to the VP. The VP loads data from the vector memory (VM) into the VP vector register(s), carries out computations and then, stores the results back into the VM. The CPU is responsible to transfer data from the off-chip DDR memory to the vector memory through DMA transfers before the computations, and from the vector memory to the DDR memory after the computations.

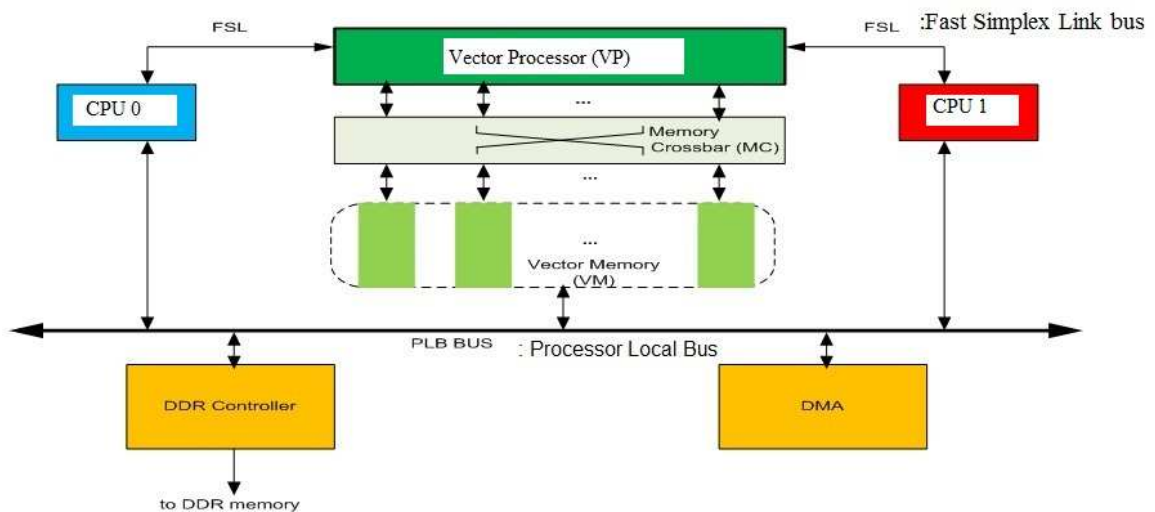


Figure 7.1 Vector processor computing platform architecture [taken from 12].

7.2 Calculation Process

The fundamental operation used is SAXPY (Single-precision real Alpha X Plus Y: $z=\alpha x+y$), which is a combination of scalar multiplication and vector addition in computations with vector processors. In order to use vector instructions, the matrix multiplication operation needs to be conducted in a different way than the traditional one. The calculation process is as follows.

Figure 7.2 shows that C1 (sub-matrix of C) is calculated from A1 (sub-matrix of A) and B, C2 from A2 and B, and so on.

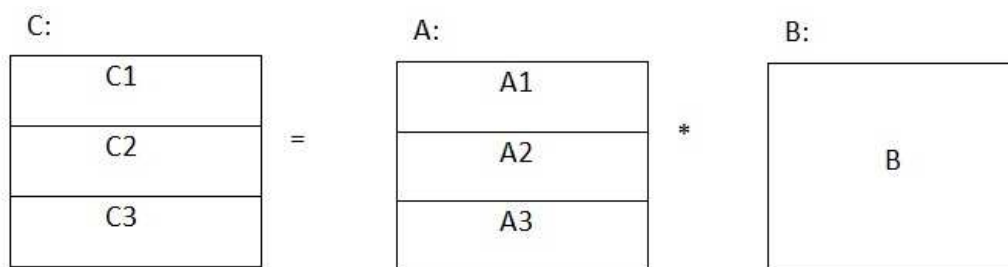


Figure 7.2 Partitioning matrices A and C for VP-based MM.

Figure 7.3 shows that C1 is actually calculated as $A1 * B$.

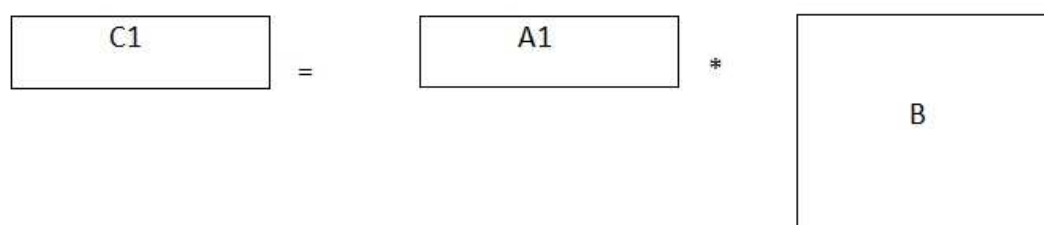


Figure 7.3 C1 is calculated as $A1 * B$.

Figure 7.4 shows how the columns of A1 are multiplied with the rows in matrix B. The first column of A1 is multiplied with the first row from B, to produce one layer of C1. The second column of A1 is multiplied with the second row from B, and the results are accumulated to C1. This procedure repeats until the final C1 is produced.

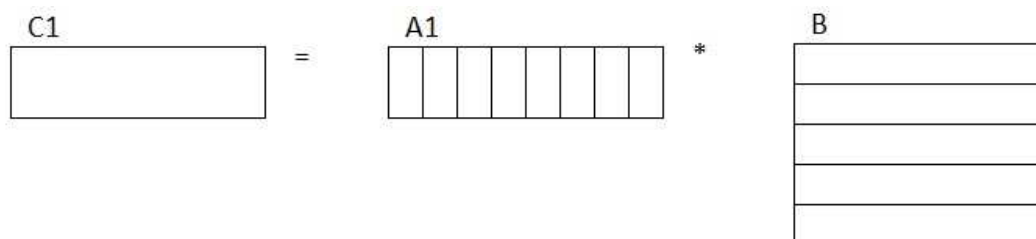


Figure 7.4 Partitioning A1 and B.

Figure 7.5 shows that how one column of elements of A1 is multiplied with one row of elements of B. One row of elements of B is divided in to several sections. The section size is the chosen vector length. The vector length is the number of elements that can be processed by one vector instruction. Before the calculation, B1 is transferred from the DDR memory to vector memory. To overlap computations with data transfers, when the computation happens on B1, B2 is being transferred to the vector memory.

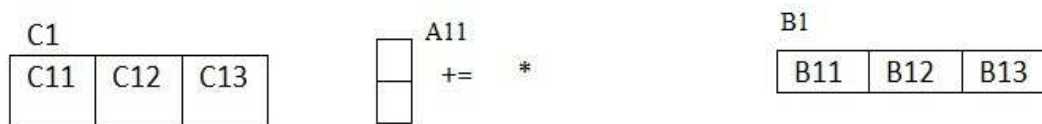


Figure 7.5 Partitioning of C1.

Figure 7.6 shows how C11 is produced. The first element of A11 is multiplied with B11 to produce the first row of C11. The second element of A11 is multiplied with B11 to produce the second row of C11, and so on.

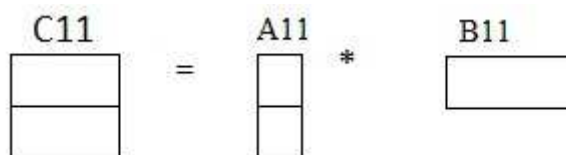


Figure 7.6 Partitioning C11.

7.3 Analysis

7.3.1 Time Complexity

The number of multiplications of elements is not reduced. The time complexity is still $O(N^3)$. However, in a vector processor, all lanes (processing units) in the VP can conduct element multiplication simultaneously. Thus, the speedup depends on the number of lanes. In this experiment, the number of lanes is eight. So the expected speedup is 8.

7.3.2 Memory Consumption

No extra memory in the DDR is needed, so the memory consumption is still $3 \cdot N^2$ elements.

7.4 Results

Tables 7.1 and 7.2 show the performance of matrix multiplication on the VP platform (125MHz). The vector length determines how many elements can be loaded into the VP at one time. Compared to other methods that were tested previously, the speedup is substantial.

Table 7.1 Execution Time (in seconds) of Matrix Multiplication on the VP Platform

Matrix Size	1024
Vector Length	
32	6.325
128	3.141

Table 7.2 Execution Time (in million clock cycles) of Matrix Multiplication on the VP Platform

Matrix Size	1024
Vector Length	
32	809.6
128	402.0

CHAPTER 8

PERFORMANCE COMPARISONS

The previous chapters presented the algorithms' performance individually and the implementations were compiled by disabling the optimizations. This chapter presents thorough performance results in various scenarios.

8.1 Intel Xeon Platform

8.1.1 Optimization Disabled (OD)

The comparison is shown in Table 8.1. $K \times K$ is the block size in number of elements.

Table 8.1 Execution Time (in Seconds) of All the Implementations on the Intel Xeon Platform with Compiling Optimization Disabled

Matrix Size Algorithm	64	128	256	512	1024	2048
Brute-force	0.002336	0.029708	0.253991	2.684	63.753	537.389
Basic Block-based (K=32)	0.002677	0.043091	0.227063	1.683	13.470	108.300
Goto's (K=8)	0.003932	0.022385	0.191102	1.453994	11.486	90.126
Strassen's (K=32)	0.005010	0.037459	0.202927	1.416484	10.004	69.778
MKL	0.027051	0.001231	0.004310	0.030382	0.328	2.516

Compiler optimizations are disabled to provide a baseline reference. In reality, some degree of optimization will be specified. Table 8.1 shows that Strassen's implementation runs slightly faster than Goto's for large matrix multiplications.

8.1.2 Full Optimization (O3)

The comparison is shown in Table 8.2. $K \times K$ is the block size in number of elements.

Table 8.2 Execution Time (in Seconds) of All the Implementations on the Intel Xeon Platform with Full Optimization (O3) [11]

Matrix Size \ Algorithm	64	128	256	512	1024	2048
Brute-force	0.000416	0.006999	0.080045	1.246206	61.403	491.611
Basic Block-based (K=32)	0.000658	0.006737	0.044589	0.395463	3.054	24.686
Goto's (K=64)	0.000350	0.004166	0.015871	0.116762	0.870	7.477
Strassen's (K=16)	0.001003	0.008999	0.065826	0.391564	2.661	18.565
MKL	0.050691	0.001343	0.004440	0.030711	0.200172	1.523986

Table 8.2 shows that when compiling with the O3 option, Goto's implementation runs faster than Strassen's algorithm and produces results even close to those of MKL. This shows that when the computation becomes faster, the bottleneck results from memory accesses.

8.2 Xilinx FPGA Platform

The comparison is shown in Table 8.1. K is the block size in number of elements.

Table 8.3 shows that the vector processor speeds up the computation drastically. It also shows that Strassen's implementation runs faster than the Goto's implementation. This is because when the processor is slow (125MHz for our FPGA implementation), the algorithm's time complexity is more influential than the memory access efficiency.

Table 8.3 Execution Time (in Seconds) of All the Implementations on the Xilinx FPGA Platform with and without the VP

Matrix Size Algorithm	64	128	256	512	1024
Brute-force	0.105	0.841	6.717	53.664	429.12
Basic Block-based (K=8)	0.054	0.457	4.029	32.648	269.376
Goto's (K=8)	0.054	0.472	4.335	32.975	264.281
Strassen's (K=16)	0.053	0.410	3.038	21.855	155
Vector Processor	-	-	-	-	3.141

8.3 MKL vs. VP

MKL was tested on the Intel Xeon platform which has a much higher clock frequency than the VP platform. In order to compare the performance of MKL and VP, the execution time is recorded in clock cycles. The result is shown in Table 8.4.

Table 8.4 Execution Time (in million clock cycles) of MM using MKL and the VP

Matrix Size Method	1024
MKL	640.5
VP	402.0

The result shows that the VP consumes fewer clock cycles than MKL.

CHAPTER 9

CONCLUSIONS

In terms of time complexity: Strassen's matrix multiplication algorithm has time complexity of $O(N^{2.807})$. The Brute-force, basic Block-based, Goto's algorithm and VP implementation all have time complexity of $O(N^3)$. In terms of memory accesses: the basic Block-based and Goto's algorithm improve the cache performance by blocking, which improves data access locality. Other than that, Goto's algorithm improves the TLB performance by copying kernel blocks into contiguous memory. The Brute-force and Strassen's algorithms have inferior cache performance due to poor data locality. The results show that when the CPU is fast, Goto's algorithm runs faster than Strassen's algorithm because the data access speed is the bottleneck in this case. On the contrary, when the CPU is slow, Strassen's algorithm runs faster because the computation complexity becomes the key factor in this case. Finally, the results show that SIMD platforms, such as the Intel Xeon with instruction extensions and the in-house developed VP (Vector Processor) for FPGA prototyping, matrix multiplication is accelerated substantially. In fact, the results show that the VP runs much faster than MKL (Intel's optimized Math Kernel Library) because the VP has can take advantage of much larger vector lengths while its overheads are negligible.

APPENDIX

C SOURCE CODE

Here is all the source code implemented.

```
//config.h
#ifndef __CONFIG_H__
#define __CONFIG_H__

#define EN_THR
#define EN_BLK
#define EN_GOTO
#define EN_STRSN
#define EN_MKL

/* define DEBUG to use simpler initialized matrix value */
#define DEBUG
//#define PRT_MALLOC

//#define PRT_MTX
#define PRT_LIGHT

#ifdef PRT_LIGHT
    #define PRT_SIZE 4
#else
    #define PRT_SIZE mtx_sz
#endif

//#define EN_CHK
//#define CTN_ERR
```

```
#define MTX_SIZE 64

#define B_SZ 4

#define GOTO_BL_SZ 4

#define STRSN_BL_SZ 4

#define MAX_MTX_SIZE 2048

#define MAX_B_SZ 64

#define MAX_GOTO_BL_SZ 64

#define MAX_STRSN_BL_SZ 64

#define TEST_SIZE 2

//Strassen's
//#define MY_MALLOC

#define EN_FREE

#define MALLOC_BASE (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x01000000)

#define A_MTX_BASE XPAR_DDR2_SDRAM_MPMC_BASEADDR
#define B_MTX_BASE (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x00400000)
#define C_MTX_BASE (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x00800000)
#define TST_MTX_BASE (XPAR_DDR2_SDRAM_MPMC_BASEADDR + 0x00C00000)

/* if define "CLOCK", it will use millisecond clock, otherwise high precision.*/
//#define CLOCK

#endif

//misc.h
```

```

#ifndef __MISC_H__
#define __MISC_H__
#include <malloc.h>
#include "config.h"

#define min(a,b) ((a)<(b)?(a):(b))
#define max(a,b) ((a)>(b)?(a):(b))

char* malloc_li(unsigned int size);

int free_li(char *p, unsigned int size);

void init_mtxs(float *DDR_A_mtx, float *DDR_B_mtx, float *DDR_C_mtx, unsigned int mtx_sz);

void blocking_mm(float *DDR_A_mtx, float *DDR_B_mtx, \
                 float *DDR_C_mtx, unsigned int mtx_sz, \
                 unsigned int B);

int cmp_mtx(float *A, float *B, unsigned int mtx_sz);

void printm(float *A, int lda, int n);

void resetm(float *A, unsigned int mtx_sz);

void goto_sgemm(float *A, int lda, \
               float *B, int ldb, \
               float *C, int ldc, \
               int Msz, int blk_size);

void stra_sgemm( float *A, int lda, float *B, int ldb, float *C, int ldc, \

```

```
int n, unsigned int strsn_blsz);

double clock_it(void);

void thr_for_loop(float *DDR_A_mtx, float *DDR_B_mtx, float *DDR_C_mtx, unsigned int mtx_sz);

#endif

//goto_blas.c
#include "config.h"
#include "misc.h"

/*
 * Block multiply Panel.
 */
static void sgebp(float *A, int lda,\
                  float *B, int ldb,\
                  float *C, int ldc, \
                  int Msz, int blk_size)
{
    float *a;
    unsigned int bs = blk_size * blk_size;
    int m, k, n, lixa, lixb, lixc, lixA;

    //copy A to continuous memory
#ifdef MY_MALLOC
```

```

a = (float*)malloc_li(bs*sizeof(float));
#else
a = (float*)malloc(bs*sizeof(float));
#endif

for(m=0; m < blk_size; m++){
    lixA = m*lda;
    lixA = m*blk_size;
    for(k=0; k < blk_size; k++){
        *(a + lixA + k) = *(A + lixA + k);
    }
}

//normal MM mutiplication
for(k=0; k < blk_size; k++){
    for(m=0; m < blk_size; m++){
        lixA=m*blk_size;
        lixB=k*ldb;
        lixC=m*ldc; //line index
        for(n=0; n < Msz; n++)
            *(C+ lixC +n) += *(a + lixA + k) * *(B + lixB + n);
    }
}

#ifdef MY_MALLOC
free_li((char *)a, bs*sizeof(float));
#else
free(a);
#endif

```

```
}

/*
 * Panel multiply Panel.
 */
static void sgepp(float *A, int lda,\
                 float *B, int ldb,\
                 float *C, int ldc,\
                 int Msz, int blk_size)
{
    int N = Msz/blk_size;
    int i = 0;
    float *Ax = A;
    float *Cx = C;
    int idxGapA = lda * blk_size;
    int idxGapC = ldc * blk_size;
    for( i = 0; i < N; i++) {
        sgebp(Ax, lda,\
             B, ldb,\
             Cx, ldc,\
             Msz, blk_size);
        Ax += idxGapA;
        Cx += idxGapC;
    }
}

/*
 * Matrix multiply Matrix.
```



```
*/  
void goto_sgemm(float *A, int lda,\br/>               float *B, int ldb,\br/>               float *C, int ldc,\br/>               int Msz, int blk_size)  
{  
    int N, i;  
    int idxGapA, idxGapB;  
    float *Ax, *Bx;  
    blk_size = blk_size < Msz ? blk_size : Msz;  
    N = Msz/blk_size;  
    Ax = A;  
    Bx = B;  
    idxGapA = blk_size;  
    idxGapB = ldb * blk_size;  
    for( i = 0; i < N; i++) {  
        sgepp(Ax, lda,\br/>             Bx, ldb,\br/>             C, ldc,\br/>             Msz, blk_size);  
        Ax += idxGapA;  
        Bx += idxGapB;  
    }  
}
```

```
//main.c
```

```
/*
```

```
* Xilinx EDK 12.3 EDK_MS3.70d
```

```
*  
* This file is a sample test application  
*  
* This application is intended to test and/or illustrate some  
* functionality of your system. The contents of this file may  
* vary depending on the IP in your system and may use existing  
* IP driver functions. These drivers will be generated in your  
* XPS project when you run the "Generate Libraries" menu item  
* in XPS.  
*  
* Your XPS project directory is at:  
* D:\Programs\Xilinx\FALL_11\mb_board_test_v01\  
*/
```

```
// Located in: microblaze_0/include/xparameters.h
```

```
#include <stdio.h>
```

```
#include<malloc.h>
```

```
#include <time.h>
```

```
#include <mkl_blas.h>
```

```
#include <windows.h>
```

```
#include "config.h"
```

```
#include "misc.h"
```

```
#ifdef MY_MALLOC
```

```
    extern unsigned int malloc_current ;
```

```
    extern unsigned int malloc_base;
```

```
    extern unsigned int malloc_high;
```

```
#endif
```

```
//=====
```

```
int main (void) {  
    unsigned int mtx_sz;  
  
#ifdef EN_BLK  
    unsigned int B;  
#endif  
  
#ifdef EN_GOTO  
    unsigned int goto_blsz;  
#endif  
  
#ifdef EN_STRSN  
    unsigned int strsn_blsz;  
#endif  
  
#ifdef EN_MKL  
    const float alpha = 1;  
    const float beta = 0;  
    const char transa='t';  
    const char transb='t';  
#endif  
  
    int re;  
    float* DDR_A_mtx;  
    float* DDR_B_mtx;  
    float* DDR_C_mtx;  
    float* DDR_T_mtx;
```

```
double execTime;

#ifdef CLOCK
    double startTime, endTime;
#else
    LARGE_INTEGER nFreq;
    LARGE_INTEGER nBeginTime;
    LARGE_INTEGER nEndTime;
    double nCycles;
    QueryPerformanceFrequency(&nFreq);
#endif

#ifdef MY_MALLOC
    malloc_base = (unsigned int)malloc(256*1024*1024);//256M memory
    malloc_current = malloc_base;
    malloc_high = malloc_base + 256*1024*1024 - 1;
#endif

for(mtx_sz = MTX_SIZE; mtx_sz <= MAX_MTX_SIZE; mtx_sz *= 2)
{
    DDR_A_mtx = (float *)malloc(sizeof(float) * mtx_sz * mtx_sz);
    DDR_B_mtx = (float *)malloc(sizeof(float) * mtx_sz * mtx_sz);
    DDR_C_mtx = (float *)malloc(sizeof(float) * mtx_sz * mtx_sz);
    DDR_T_mtx = (float *)malloc(sizeof(float) * mtx_sz * mtx_sz);

    init_mtxs(DDR_A_mtx, DDR_B_mtx, DDR_C_mtx, mtx_sz);
    resetm(DDR_T_mtx, mtx_sz);
}

#ifdef PRT_MTX
```

```

printf("A:\r\n");
printm(DDR_A_mtx,mtx_sz ,PRT_SIZE);
printf("B:\r\n");
printm(DDR_B_mtx,mtx_sz ,PRT_SIZE);
printf("C:\r\n");
printm(DDR_C_mtx,mtx_sz ,PRT_SIZE);
#endif

#ifdef EN_THR
    { //Algorithm 1
        printf("\n----- 3-for-loop:\r\n");
        printf("Matrix size = %d\r\n", mtx_sz);
        printf("START 3-for-loop implementation\r\n");

#ifdef CLOCK
            startTime = clock_it();

            // START PERFORMANCE ROUTINE
            thr_for_loop(DDR_A_mtx, DDR_B_mtx, DDR_T_mtx, mtx_sz);
            // END PERFORMANCE ROUTINE

            endTime = clock_it();
            execTime = endTime - startTime;
            printf("Execution time is %3.4f seconds\n", execTime);
#endif
    }
#else
    QueryPerformanceCounter(&nBeginTime);
    // START PERFORMANCE ROUTINE
    thr_for_loop(DDR_A_mtx, DDR_B_mtx, DDR_T_mtx, mtx_sz);
    // END PERFORMANCE ROUTINE
    QueryPerformanceCounter(&nEndTime);

```

```

nCycles = (double)(nEndTime.QuadPart-nBeginTime.QuadPart);
execTime =nCycles /((double)nFreq.QuadPart);

printf("The cpu's frequency is: %.0f Hz\n", (double)nFreq.QuadPart);
printf("Execution takes %.0f cycles\n", nCycles);
printf("Execution takes %.9f seconds\n", execTime);
#endif

#ifdef PRT_MTX
    printf("T:\r\n");
    printm(DDR_T_mtx,mtx_sz ,PRT_SIZE);
#endif
printf("END 3-for-loop implementation\r\n");
}
#endif

#ifdef EN_BLK
    for(B = B_SZ; B <= MAX_B_SZ; B*=2){
        //Algorithm 2
        resetm(DDR_C_mtx, mtx_sz);
        printf("\n----- Blocking algorithm:\r\n");
        printf("Matrix size = %d\r\n", mtx_sz);
        printf("Block size = %d\r\n", B);
        printf("START blocking implementation\r\n");
    }
#endif

#ifdef CLOCK
    startTime = clock_it();

```

```

// START PERFORMANCE ROUTINE
blocking_mm(DDR_A_mtx, DDR_B_mtx, DDR_C_mtx, mtx_sz, B);
// END PERFORMANCE ROUTINE

endTime = clock_it();
execTime = endTime - startTime;
printf("Execution time is %3.4f seconds\n", execTime);
#else
QueryPerformanceCounter(&nBeginTime);
// START PERFORMANCE ROUTINE
blocking_mm(DDR_A_mtx, DDR_B_mtx, DDR_C_mtx, mtx_sz, B);
// END PERFORMANCE ROUTINE
QueryPerformanceCounter(&nEndTime);

nCycles = (double)(nEndTime.QuadPart-nBeginTime.QuadPart);
execTime =nCycles /((double)nFreq.QuadPart);

printf("The cpu's frequency is: %.0f Hz\n", (double)nFreq.QuadPart);
printf("Execution takes %.0f cycles\n", nCycles);
printf("Execution takes %.9f seconds\n", execTime);
#endif

#ifdef PRT_MTX
printf("C:\r\n");
printm(DDR_C_mtx, mtx_sz,PRT_SIZE);
#endif

#ifdef EN_CHK

```

```

if((re = cmp_mtx(DDR_C_mtx, DDR_T_mtx, mtx_sz)) != -1)
{
    printf("Calculation wrong at row %d, column %d\r\n",re/mtx_sz, re%mtx_sz);
#ifdef CTN_ERR
    printf("Abort.\r\n");
    return -1;
#endif
}
else
{
    printf("Result correct!\r\n");
}
#endif

printf("END blocking implementation\r\n");

} //for
#endif

#ifdef EN_GOTO
for(goto_blsz = GOTO_BL_SZ; goto_blsz <= MAX_GOTO_BL_SZ; goto_blsz*=2){
//Algorithm 3
resetm(DDR_C_mtx, mtx_sz);
printf("----- GotoBLAS algorithm:\r\n");
printf("Matrix size = %d\r\n", mtx_sz);
printf("Block size = %d\r\n", goto_blsz);
printf("START GotoBLAS implementation\r\n");
#endif CLOCK
startTime = clock_it();

// START PERFORMANCE ROUTINE

```



```

        goto_sgemm(DDR_A_mtx, mtx_sz, DDR_B_mtx, mtx_sz,DDR_C_mtx, mtx_sz, mtx_sz,
goto_blsz);

        // END PERFORMANCE ROUTINE

        endTime = clock_it();
        execTime = endTime - startTime;
        printf("Execution time is %3.4f seconds\n", execTime);
#else
        QueryPerformanceCounter(&nBeginTime);

        // START PERFORMANCE ROUTINE
        goto_sgemm(DDR_A_mtx, mtx_sz, DDR_B_mtx, mtx_sz,DDR_C_mtx, mtx_sz, mtx_sz,
goto_blsz);

        // END PERFORMANCE ROUTINE
        QueryPerformanceCounter(&nEndTime);

        nCycles = (double)(nEndTime.QuadPart-nBeginTime.QuadPart);
        execTime =nCycles /((double)nFreq.QuadPart);

        printf("The cpu's frequency is: %.0f Hz\n", (double)nFreq.QuadPart);
        printf("Execution takes %.0f cycles\n", nCycles);
        printf("Execution takes %.9f seconds\n", execTime);
#endif

#ifdef PRT_MTX
        printf("C:\r\n");
        printm(DDR_C_mtx,mtx_sz,PRT_SIZE);
#endif

#ifdef EN_CHK

```

```

if((re = cmp_mtx(DDR_C_mtx, DDR_T_mtx, mtx_sz)) != -1)
{
    printf("Calculation wrong at row %d, column %d\r\n",re/mtx_sz, re%mtx_sz);
#ifdef CTN_ERR
    printf("Abort.\r\n");
    return -1;
#endif
}
else
{
    printf("Result correct!\r\n");
}
#endif

printf("END GotoBLAS implementation\r\n");
}

#endif

#ifdef EN_STRSN
for(strsn_blsz = STRSN_BL_SZ; strsn_blsz <= MAX_STRSN_BL_SZ; strsn_blsz*=2){
resetm(DDR_C_mtx, mtx_sz);

//Algorithm 4
printf("----- Strassen:\r\n");
printf("Matrix size = %d\r\n", mtx_sz);
printf("Block size = %d\r\n", strsn_blsz);
printf("START STRASSEN\r\n");

#endif

#ifdef CLOCK
startTime = clock_it();

// START PERFORMANCE ROUTINE

```

```

stra_sgemm(DDR_A_mtx, mtx_sz, DDR_B_mtx, mtx_sz, DDR_C_mtx, mtx_sz, \
           mtx_sz, strsn_blsz);
// END PERFORMANCE ROUTINE

endTime = clock_it();
execTime = endTime - startTime;
printf("Execution time is %3.4f seconds\n", execTime);
#else
QueryPerformanceCounter(&nBeginTime);
// START PERFORMANCE ROUTINE
stra_sgemm(DDR_A_mtx, mtx_sz, DDR_B_mtx, mtx_sz, DDR_C_mtx, mtx_sz, \
           mtx_sz, strsn_blsz);
// END PERFORMANCE ROUTINE
QueryPerformanceCounter(&nEndTime);

nCycles = (double)(nEndTime.QuadPart-nBeginTime.QuadPart);
execTime =nCycles /((double)nFreq.QuadPart);

printf("The cpu's frequency is: %.0f Hz\n", (double)nFreq.QuadPart);
printf("Execution takes %.0f cycles\n", nCycles);
printf("Execution takes %.9f seconds\n", execTime);
#endif

#ifdef PRT_MTX
printf("C:\r\n");
printm(DDR_C_mtx,mtx_sz,PRT_SIZE);
#endif

```

```

#ifdef EN_CHK
    if((re = cmp_mtx(DDR_C_mtx, DDR_T_mtx, mtx_sz)) != -1)
    {
        printf("Calculation wrong at row %d, column %d\r\n",re/mtx_sz, re%mtx_sz);
#ifdef CTN_ERR
        printf("Abort.\r\n");
        return -1;
#endif
    }
    else
    {
        printf("Result correct!\r\n");
    }
#endif

    printf("END STRASSEN\r\n");
}

#endif

#ifdef EN_MKL
    //Algorithm 3
    resetm(DDR_C_mtx, mtx_sz);
    printf("----- MKL library:\r\n");
    printf("Matrix size = %d\r\n", mtx_sz);
    printf("START MKL implementation\r\n");

    //using function from MKL.

    //void sgemm(const char *transa, const char *transb, const MKL_INT *m, const MKL_INT *n,
const MKL_INT *k,
        //const float *alpha, const float *a, const MKL_INT *lda, const float *b, const MKL_INT *ldb,
        //const float *beta, float *c, const MKL_INT *ldc);
#endif
#ifdef CLOCK

```

```

startTime = clock_it();

//result = alpha * A * B + beta * C

sgemm(&transa, &transb, &mtx_sz, &mtx_sz, &mtx_sz,
      &alpha, DDR_A_mtx, &mtx_sz, DDR_B_mtx, &mtx_sz,
      &beta, DDR_C_mtx, &mtx_sz);

endTime = clock_it();

execTime = endTime - startTime;

printf("Execution takes %3.4f seconds\n", execTime);

#else

QueryPerformanceCounter(&nBeginTime);

//result = alpha * A * B + beta * C

sgemm(&transa, &transb, &mtx_sz, &mtx_sz, &mtx_sz,
      &alpha, DDR_A_mtx, &mtx_sz, DDR_B_mtx, &mtx_sz,
      &beta, DDR_C_mtx, &mtx_sz);

QueryPerformanceCounter(&nEndTime);

nCycles = (double)(nEndTime.QuadPart-nBeginTime.QuadPart);

execTime =nCycles /((double)nFreq.QuadPart);

printf("The cpu's frequency is: %.0f Hz\n", (double)nFreq.QuadPart);

printf("Execution takes %.0f cycles\n", nCycles);

printf("Execution takes %.9f seconds\n", execTime);

#endif

#ifdef PRT_MTX
    printf("C:\r\n");
    printm(DDR_C_mtx,mtx_sz,PRT_SIZE);
#endif

#ifdef EN_CHK

```

```
    if((re = cmp_mtx(DDR_C_mtx, DDR_T_mtx, mtx_sz)) != -1)
    {
        printf("Calculation wrong at row %d, column %d\r\n",re/mtx_sz, re%mtx_sz);
#ifdef CTN_ERR
        printf("Abort.\r\n");
        return -1;
#endif
    }
    else
    {
        printf("Result correct!\r\n");
    }
#endif

    printf("END MKL implementation\r\n");
#endif

    free(DDR_A_mtx);
    free(DDR_B_mtx);
    free(DDR_C_mtx);
    free(DDR_T_mtx);
} //outer most "for"

    printf("-- Exiting main()--\r\n");

    return 0;
}

//strassen.c
#include <stdio.h>
```

```

#include "misc.h"

#include "config.h"

//single precision general matrix-matrix addition
//ld' is leading dimension, for example, for submatrix in A[m][n], their leading dimension is 'm'.
void sgema(float *A, int lda, float *B, int ldb, float *C, int ldc, int n)
{
    int i,j;
    for(i=0; i < n; i++)
    {
        for(j=0; j < n; j++)
        {
            *(C + i*ldc + j) = *(A + i*lda + j) + *(B + i*ldb + j);
        }
    }
}

//single precision general matrix-matrix subtraction
void sgems(float *A, int lda, float *B, int ldb, float *C, int ldc, int n)
{
    int i,j;
    for(i=0; i < n; i++)
    {
        for(j=0; j < n; j++)
        {
            *(C + i*ldc + j) = *(A + i*lda + j) - *(B + i*ldb + j);
        }
    }
}

```

```

//matrix-matrix multiplication
void stra_sgemm( float *A, int lda, float *B, int ldb, float *C, int ldc, \
                int n, unsigned int strsn_blsz)
{
    //print("Entering sgemm.\r\n");
    if( n <= strsn_blsz)
    {
        int i,j,k;
        for( i=0; i<n; i++)
        {
            for( j=0; j<n; j++)
            {
                *(C + i*ldc +j) = 0;
                for( k=0; k<n; k++ )
                {
                    *(C + i*ldc +j) += *(A + i*lda +k) * *(B + k*ldb + j);
                }
            }
        }
    }
    else
    {
        int ldm = n/2;
        float *a,*b,*c,*d;
        float *e,*f,*g,*h;
        float *r,*s,*t,*u;

#ifdef MY_MALLOC
        float *p1 = (float *)malloc_li(sizeof(float) * ldm * ldm);
        float *p2 = (float *)malloc_li(sizeof(float) * ldm * ldm);
        float *p3 = (float *)malloc_li(sizeof(float) * ldm * ldm);
#endif
    }
}

```



```
float *p4 = (float *)malloc_li(sizeof(float) * ldm * ldm);  
float *p5 = (float *)malloc_li(sizeof(float) * ldm * ldm);  
float *p6 = (float *)malloc_li(sizeof(float) * ldm * ldm);  
float *p7 = (float *)malloc_li(sizeof(float) * ldm * ldm);
```

```
float *A1 = (float *)malloc_li(sizeof(float) * ldm * ldm);  
float *A2 = (float *)malloc_li(sizeof(float) * ldm * ldm);  
float *A3 = (float *)malloc_li(sizeof(float) * ldm * ldm);  
float *A4 = (float *)malloc_li(sizeof(float) * ldm * ldm);  
float *A5 = (float *)malloc_li(sizeof(float) * ldm * ldm);  
float *B5 = (float *)malloc_li(sizeof(float) * ldm * ldm);  
float *A6 = (float *)malloc_li(sizeof(float) * ldm * ldm);  
float *B6 = (float *)malloc_li(sizeof(float) * ldm * ldm);  
float *A7 = (float *)malloc_li(sizeof(float) * ldm * ldm);  
float *B7 = (float *)malloc_li(sizeof(float) * ldm * ldm);
```

#else

```
float *p1 = (float *)malloc(sizeof(float) * ldm * ldm);  
float *p2 = (float *)malloc(sizeof(float) * ldm * ldm);  
float *p3 = (float *)malloc(sizeof(float) * ldm * ldm);  
float *p4 = (float *)malloc(sizeof(float) * ldm * ldm);  
float *p5 = (float *)malloc(sizeof(float) * ldm * ldm);  
float *p6 = (float *)malloc(sizeof(float) * ldm * ldm);  
float *p7 = (float *)malloc(sizeof(float) * ldm * ldm);
```

```
float *A1 = (float *)malloc(sizeof(float) * ldm * ldm);  
float *A2 = (float *)malloc(sizeof(float) * ldm * ldm);  
float *A3 = (float *)malloc(sizeof(float) * ldm * ldm);  
float *A4 = (float *)malloc(sizeof(float) * ldm * ldm);  
float *A5 = (float *)malloc(sizeof(float) * ldm * ldm);  
float *B5 = (float *)malloc(sizeof(float) * ldm * ldm);
```

```
float *A6 = (float *)malloc(sizeof(float) * ldm * ldm);
float *B6 = (float *)malloc(sizeof(float) * ldm * ldm);
float *A7 = (float *)malloc(sizeof(float) * ldm * ldm);
float *B7 = (float *)malloc(sizeof(float) * ldm * ldm);

#endif

a = A;
b = A + ldm;
c = A + lda * ldm;
d = c + ldm;

e = B;
f = B + ldm;
g = B + ldb * ldm;
h = g + ldm;

r = C;
s = C + ldm;
t = C + ldc * ldm;
u = t + ldm;

//p1 = a * (f - h);
sgems(f, ldb, h, ldb, A1, ldm, ldm);
stra_sgemm(a, lda, A1, ldm, p1, ldm, ldm, strsn_blsz);

//p2 = (a + b) * h;
sgema(a, lda, b, lda, A2, ldm, ldm);
stra_sgemm(A2, ldm, h, ldb, p2, ldm, ldm, strsn_blsz);

//p3 = (c + d) * e;
```

```

sgema(c, lda, d, lda, A3, ldm, ldm);
stra_sgemm(A3, ldm, e, ldb, p3, ldm, ldm, strsn_blsz);

//p4 = d * (g - e);
sgems(g, ldb, e, ldb, A4, ldm, ldm);
stra_sgemm(d, lda, A4, ldm, p4, ldm, ldm, strsn_blsz);

//p5 = (a + d) * (e + h);
sgema(a, lda, d, lda, A5, ldm, ldm);
sgema(e, ldb, h, ldb, B5, ldm, ldm);
stra_sgemm(A5, ldm, B5, ldm, p5, ldm, ldm, strsn_blsz);

//p6 = (b - d) * (g + h);
sgems(b, lda, d, lda, A6, ldm, ldm);
sgema(g, ldb, h, ldb, B6, ldm, ldm);
stra_sgemm(A6, ldm, B6, ldm, p6, ldm, ldm, strsn_blsz);

//p7 = (a - c) * (e + f);
sgems(a, lda, c, lda, A7, ldm, ldm);
sgema(e, ldb, f, ldb, B7, ldm, ldm);
stra_sgemm(A7, ldm, B7, ldm, p7, ldm, ldm, strsn_blsz);

//r = p5 + p4 - p2 + p6;
sgema(p5, ldm, p4, ldm, A1, ldm, ldm);
sgems(A1, ldm, p2, ldm, A2, ldm, ldm);
sgema(A2, ldm, p6, ldm, r, ldc, ldm);

//s = p1 + p2;
sgema(p1, ldm, p2, ldm, s, ldc, ldm);

```

```

//t = p3 + p4;
sgema(p3, ldm, p4, ldm, t, ldc, ldm);

//u = p5 + p1 - p3 - p7;
sgema(p5, ldm, p1, ldm, A1, ldm, ldm);
sgems(A1, ldm, p3, ldm, A2, ldm, ldm);
sgems(A2, ldm, p7, ldm, u, ldc, ldm);

```

```

#ifdef MY_MALLOC

```

```

#ifdef EN_FREE //free space

```

```

//free space
free_li((char*)B7, sizeof(float) * ldm * ldm);
free_li((char*)A7, sizeof(float) * ldm * ldm);
free_li((char*)B6, sizeof(float) * ldm * ldm);
free_li((char*)A6, sizeof(float) * ldm * ldm);
free_li((char*)B5, sizeof(float) * ldm * ldm);
free_li((char*)A5, sizeof(float) * ldm * ldm);
free_li((char*)A4, sizeof(float) * ldm * ldm);
free_li((char*)A3, sizeof(float) * ldm * ldm);
free_li((char*)A2, sizeof(float) * ldm * ldm);
free_li((char*)A1, sizeof(float) * ldm * ldm);

free_li((char*)p7, sizeof(float) * ldm * ldm);
free_li((char*)p6, sizeof(float) * ldm * ldm);
free_li((char*)p5, sizeof(float) * ldm * ldm);
free_li((char*)p4, sizeof(float) * ldm * ldm);
free_li((char*)p3, sizeof(float) * ldm * ldm);
free_li((char*)p2, sizeof(float) * ldm * ldm);

```

```
        free_li((char*)p1, sizeof(float) * ldm * ldm);
    #endif

    #else

        free(p1);
        free(p2);
        free(p3);
        free(p4);
        free(p5);
        free(p6);
        free(p7);

        free(A1);
        free(A2);
        free(A3);
        free(A4);
        free(A5);
        free(B5);
        free(A6);
        free(B6);
        free(A7);
        free(B7);

    #endif
    }
}

//utility.c
#include <stdio.h>
#include <time.h>
```

```
#include "config.h"

#include "misc.h"

#ifdef MY_MALLOC

unsigned int malloc_current = 0;
unsigned int malloc_base=0;
unsigned int malloc_high=0;

char* malloc_li(unsigned int size)
{
    char *ret;
    ret = (char*)malloc_current;
    if((malloc_current + size) > malloc_high)
    {
        printf("Error: Malloc(), not enough memory.\r\n");
        printf("size: %d, current: 0x%x \r\n", size, malloc_current);
        return 0;
    }
    else
    {
        malloc_current += size;
#ifdef PRT_MALLOC
        xil_printf("malloced: %d, current: 0x%x \r\n", size, malloc_current);
#endif
    }
    return ret;
}

int free_li(char *p, unsigned int size)
```

```

{
    if( (malloc_current - size) < malloc_base)
    {
        printf("Error: free_li(), reached bottom.\r\n");
        printf("size: %d, current: 0x%x \r\n", size, malloc_current);
        return -1;
    }
    else
    {
        malloc_current -= size;
        p = 0;
#ifdef PRT_MALLOC
        xil_printf("freed: %d, current: 0x%x \r\n", size, malloc_current);
#endif
        return 0;
    }
}
#endif

void init_mtxs(float *DDR_A_mtx, float *DDR_B_mtx, float *DDR_C_mtx, unsigned int mtx_sz)
{
    unsigned int i, j;

    printf("START Initialize DDRAM\r\n");
    // Initialize DDRAM
    for (i=0; i<mtx_sz; i++) {
        for (j=0; j<mtx_sz; j++) {
#ifdef DEBUG
            DDR_A_mtx[i*mtx_sz+j] = (float)(i*j+1)/(float)23;
            DDR_B_mtx[i*mtx_sz+j] = (float)(i*j+3)/(float)31;
#endif
        }
    }
}

```

```

        DDR_C_mtx[i*mtx_sz+j] = (float)0.0;
#else
        DDR_A_mtx[i*mtx_sz+j] = (float)((i*j+1)%2)/(float)10.0;
        DDR_B_mtx[i*mtx_sz+j] = (float)((i*j+3)%3)/(float)10.0;
        DDR_C_mtx[i*mtx_sz+j] = (float)0.0;
#endif
    }
}

printf("END Initialize DDRAM\r\n");
}

//get the current time in seconds
double clock_it(void)
{
    clock_t start;
    double timeInSec;

    start = clock();
    timeInSec = (double)(start) / CLOCKS_PER_SEC;
    return timeInSec;
}

//three for loops implementation of Matrix-Matrix Multiplication
void thr_for_loop(float *DDR_A_mtx, float *DDR_B_mtx, float *DDR_C_mtx, unsigned int mtx_sz)
{
    unsigned int i,j,k;
    float sum;
    for (i=0; i<mtx_sz; i++) {

```



```

        for (j=0; j<mtx_sz; j++) {
            sum=0.0;
            for (k=0; k<mtx_sz; k++) {
                sum+= DDR_A_mtx[i*mtx_sz+k] * DDR_B_mtx[k*mtx_sz+j];
            }
            DDR_C_mtx[i*mtx_sz+j]=sum;
        }
    }

/*
 * Blocking implementation of MM Multiplication.
 * Caculate block by bock to increase cache hit rate.
 */
void blocking_mm(float *DDR_A_mtx, float *DDR_B_mtx, \
                float *DDR_C_mtx, unsigned int mtx_sz,\
                unsigned int B)
{
    unsigned int i, j, k, jj, kk;
    float sum;
    B = B < mtx_sz ? B : mtx_sz;
    for (jj=0; jj<mtx_sz; jj=jj+B) {
        for (kk=0; kk<mtx_sz; kk=kk+B) {
            for (i=0; i<mtx_sz; i=i+1) {
                for (j=jj; j<min(jj+B,mtx_sz); j++) {
                    sum=0.0;
                    for (k=kk; k<min(kk+B,mtx_sz); k++) {
                        sum+=DDR_A_mtx[i*mtx_sz+k] *
DDR_B_mtx[k*mtx_sz+j];
                    }
                    DDR_C_mtx[i*mtx_sz+j]+=sum;
                }
            }
        }
    }
}

```

```

    }
}
}
}

int cmp_mtx(float *A, float *B, unsigned int mtx_sz)
{
    int i,j;
    int test_size;
    test_size = TEST_SIZE < mtx_sz ? TEST_SIZE : mtx_sz;
    for (i=0; i<test_size; i=i+1) {
        for (j=0; j<test_size; j=j+1)
        {
            if((A[i*mtx_sz+j] - B[i*mtx_sz+j]) < 1)
                continue;
            else
                return i*mtx_sz+j;
        }
    }
    return -1;
}

```

//print matrix

```
void printm(float *A, int lda, int n)
```

```

{
    float x;
    int i,j;
    for(i=0; i < n; i++)
    {

```

```
        for(j=0; j < n; j++)
        {
            x = *(A + i*lda + j);
            printf("\t%.2f",x);
        }
        printf("\n");
    }
    printf("\n");
}

void resetm(float *A, unsigned int mtx_sz)
{
    unsigned int i,j;
    for (i=0; i<mtx_sz; i=i+1) {
        for (j=0; j<mtx_sz; j=j+1)
        {
            A[i*mtx_sz+j] = 0;
        }
    }
}
```

REFERENCE

- [1] John L. Hennessy and David A. Patterson , *Computer Architecture: A Quantitative Approach*, 4th ed. San Francisco, CA: Morgan Kaufmann, 2006, pp. 288-305.
- [2] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, Article 12, 2008
- [3] Thomas H. Cormen, Charles E. Leiserson and Clifford Stein, *Introduction To Algorithms*, 2nd ed. Cambridge, MA: MIT Press, 2001, pp. 632-646.
- [4] Don Coppersmith and Shmuel Winograd, "Matrix multiplication via arithmetic progressions", *Journal of Symbolic Computation*, vol. 9, no. 3, pp. 251-280, 1990
- [5] Volker Strassen, "Gaussian Elimination is not Optimal," *Numer. Math.*, vol. 13, pp. 354-356, 1969
- [6] Mengqi Jiang, Yunquan Zhang, Gang Song and Yucheng Li. "Research on High Performance Implementation Mechanism of GOTOBLAS General Matrix-matrix Multiplication," *Computer Engineering*, vol. 34, no. 7, pp. 84-86, 2008
- [7] Zhonglong Lu, Cheng Zhong and Hualin Huang, "Non-recursive Parallel Computation for Matrix Multiplication on Multi-core Computers," *Journal of Chinese Computer Systems*, vol. 32, no. 5, pp. 860-866, 2011
- [8] (2012, March 23). [Online]. Available: <http://en.wikipedia.org/wiki/OpenMP>
- [9] (2012, March 29). [Online]. Available: http://en.wikipedia.org/wiki/Automatically_Tuned_Linear_Algebra_Software
- [10] (2012, March 29). *Intel® Math Kernel Library Reference Manual* [Online]. Available: <http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/index.htm>

[11] (2012, April 01). Quick-Reference Guide to Optimization with Intel® Compilers version 12 [Online]. Available:

http://software.intel.com/sites/products/collateral/hpc/compilers/compiler_qrg12.pdf

[12] S. F. Beldianu and S. G. Ziavras, "Multicore-based Vector Coprocessor Sharing for Performance and Energy Gains," *ACM Transactions on Embedded Computing Systems*, Accepted for publication.

[13] S. F. Beldianu, "Vector Coprocessor Sharing Techniques for Multicores: Performance and Energy Gains", PhD Dissertation, New Jersey Institute of Technology, May 2012