Embedded Systems Laboratory (2016/17)

Lab. Assignment $1 : Group12$

Matrix Multiplication on Heterogeneous Processors

Abbas Jhabuawala - 4621883,
Anant Semwal - 4630734,
Anand-Haridas-4619153,
Anurag Kulkarni- 4627342,
Xiaoming Wen-4608682,
Minfeng Li-4608348

May 8, 2017

1. Introduction

Heterogeneous processors are nowadays widely used in embedded applications. In this report, an efficient matrix multiplication is introduced. Rather than solely running the computation on ARM, a heterogeneous programming idea is applied.

2. Problem Statement

This report is to optimize the matrix multiplication cost. Instead of using ARM to compute the product of two matrices, an on board DSP which can perform the computation much faster in combination with the ARM, is used. By resolving an appropriate communication scheme and achieving parallel computation, a speed up can be achieved.

3. Alternative Solution

The problem was approached by use of different kinds of ControlMsg

3.1 Approach 1

The matrix multiplication application is implemented on the DSP, with the matrices being generated in the ARM processor and send via queues to the DSP. The approach involves transferring a single row of first matrix and a single column of the second matrix in the same message using a vector of twice the maximum row/column size (i.e. 256). The message is passed via the message queue. The vectors are separated and multiplied in the DSP, and the result is send via the same message queue using a different argument of the same message. This approach results a maximum transfer of 128 * 128, as each row and column are multiplied and send back, and is highly inefficient.

3.2 Approach 2

In this approach, we send COLS0 number of rows from MATRIX 1 of N- order, and store the elements on DSP. After MATRIX 1 contents are loaded on DSP, ack message is sent from DSP after which COLS0 number of columns are sent to DSP till the entire MATRIX 2 is covered. Each time MATRIX 2 contents are sent to DSP, result of COLS0 order is sent in response from DSP which is mapped to result matrix of order N in ARM core. Above process is executed N/COLS0 times if N%COLS0 == 0, otherwise, (N/COLS0) +1 times. This covers entire matrix of N-order. Maximum N can be provided as 128. There was no check in code to check if the input argument was more than 128 and thus user co-operation was assumed during the programming. Uint16 mat[128][COLS] was configured in Control Message and Uint32 result was mapped on Uint16 matrix and extracted accordingly using row-major algorithm.

3.3 Approach 3

This approach contains three arguments in the control message, all three arguments are one dimensional array. In argument one 16 rows of matrix 1 are sent in each transfer and argument two is assigned with the whole matrix 2. Argument three sends back of the result. In dsp side, with matrix 2 and 16 rows of matrix 1 per transfer dsp could obtain 16 rows of result and send back in argument 3.

3.4 Approach 4

DSP imposes stringent memory restrictions (stack size being 4kB). Observing the formulae used to generate the two matrices to be multiplied, it was found that these two source matrices could be fit into arrays of data types Uint16. But the product was found to require an array of data type Uint32. This mismatch, and also the finding that the ControlMsg structure could not accommodate the double dimensional arrays of the required size and data types, an approach was sought which would keep three vectors of lengths 4096 and data type Uint32 as members of the ControlMsg structure. The rationale behind this is explained below.

The source arrays can be up to size 128X128. The product can also hence be up to the same size. This necessitates a partitioning strategy. After experimenting with various combinations of the numbers of rows and columns, a single dimensional array of size 4096 and was decided as the partitioning unit, and also a member of the ControlMsg structure. Thus, the GPP would send a chunk of the first source matrix - with the column size same as the original matrix and the row size being one-fourth. Inside the same message structure, a similar column-wise chunk of the second source matrix would be sent. This is convenient for matrix multiplication on the DSP, which, after receiving a row-wise chunk of the first source matrix, and all corresponding required column-wise chunks of the second matrix, would generate a chunk of the product with the row and column size being one-fourth of the final product matrix, and pass it immediately to the GPP. Thus, between the ARM and the DSP, a total of 17 iterations take place - including the message indicating the active status of the DSP. The main intention behind this approach was to reduce the number of iterations (respecting memory limitations) and maximally utilizing the DSP for matrices of larger sizes.

3.5 Approach 5

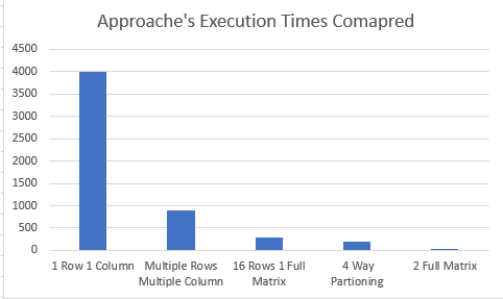
This approach is also a partitioning strategy. within the fixed length of message transfered,which is defined as input parameter, it sends a part of rows of matrix 1 and a part of columns of matrix 2 to DSP. After receiving a single message, the DSP can compute the part of the multiplication result and send it back to ARM without storing any data. To minimal the transfer times, the number of rows and columns sending to DSP should be as equal as possible.

3.6 Approach 6

In order to send the complete matrix in one transfer, we define the data for multiplication as type "Uint16" and the multipication result as type "Uint32". In this approach, the receiced data is also stored on DSP side and we only need 4 iterations for data transferring("Uint32" is two times bigger than "Uint16",so the multiplication result needs two transfers).

3.7 Comparison of Approaches

Approach No.	Execution Time (~ms)
Approach 1	4000
Approach 2	900
Approach 3	280
Approach 4	250
Approach 5	86
Approach 6	27



4. Optimal Solution

128*128 elements in "Uint32" format can not be passed in one transfer, so We use two arrays in "Uint16" format to store the data and a array in "Uint32" to store the multiplication result. In this way, 128*128 elements in "Uint16" can be transfer in one itration, which greatly reduces the transfer time. After receicing the data, the DSP will store it locally and implement the mulplication once the all the data(tow matrixs for multiplication) is prepared. Since the result is in "Uint32" format, two transter is required to pass it from DSP to ARM. By the way, to solve this different format conflict, the type of "union" is used in "ControlMsg" structure for storing different data with different formats.

5. Analysis & Results

5.1 Profiling

GProf was used to profile the given application to analyze the program function calls and time spent in each function. With this knowledge optimizations were applied in order to speedup the execution time.The Gprof output shows a Flat profile which displays shows how much time the program spent in each function, and how many times that function was called. It also outputs a Call graph which shows, for each function, which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function. As the output images were too big we have attached them with the source code in the zip file.

5.2 Optimization

It was observed that DSP consumed a high proportion of the execution time. -o3 optimization flag was added for DSP to select a high compiler optimization level. For the approach selected, it resulted in above 2x speedup. The multiplication is done in three loops, Also, the MUST_ITERATE # pragma was used to inform the compiler about the maximum iterations of the loop in advance. This results in further speed up on execution time on the DSP side as multiple for loops were used. Furthermore loop unrolling for also implemented in inner loop which significantly improve the performance. The disadvantage was that this increased the code size but we were willing to make this trade off for improvement in speed.

5.3 Execution Time

The execution time acquired for the optimal solution was 27.466ms for a square matrix multiplication of size 128. The code was tested from 1-127 sizes with the aid of a script to verify the correctness of the application

5.4 Memory Limitation

Owing to the restricted size of DSP memory, most of our approaches failed to generate an optimal code. However in the final approach, we made use of $union$ which simplified memory management as we were able to increase number of elements sent to DSP as Uint16 and in the same matrix we were able to retrieve product which was Uint32.

5.5 Speed-Up

A speedup of approximately 4.5x was achieved while using the optimal solution when the matrix size was 128. All sizes from $1-128$ were also tested but the speed up fluctuated. Due to loop unrolling factor of 16, when ever the value n = $MATRIX\_SIZE/16$ increases, there is an increase in completion time, as inner loop is executed $n+1$ times. However processing time is capped under $26.466ms$.

6. Conclusion

We learned several lessons while solving the problem of matrix multiplication on a heterogeneous core. Tackling memory challenges which arise due to memory size limitation on child processor (DSP), communication between master processor and child processor for result and processing data, along with control messages was also dealt with. We encountered problem where DSP would crash due to insufficient memory and program would deadlock in execute state waiting for an acknowledgement message from DSP core. Nevertheless we managed to provide a speedup of 4.5x.