# Assignment 1
# BeagleBoard: Optimization of Matrix Calculation

Group 11
Michiel van der Vlag 4369327 Andrea Pallini 4519086
Atul Pandaravila Biju 4511077 Christina Kylili 4520203
Christos Kyprianou 4516222 Dirk Vos 4391470

May 3, 2016

## 1 Introduction

The TU Delft course IN4342 Embedded System Laboratory revolves around two projects which let us play the role of system engineers that received the task to implement a determined application on the OMAP 3530 system. The goal of this first lab-assignment is to optimize a matrix calculation program running in an heterogeneous core consisting of a GPP ARM, a NEON processor and an application specific DSP, using a BeagleBoard. This aims to make us learn how to compare and analyze different possible solutions, to improve the performance of a system, in terms of execution time.
The baseline program to which all optimizations will be compared is a plain matrix calculation program with no optimizations run only on the ARM. The matrices used in our test are all squared matrices. When we speak of a matrix of size 32, we mean a matrix of size 32x32. Our goal is to optimize the execution time of the application, by using the processing power of the DSP and NEON.
This document describes the different aspects we have looked into through the course of the project to optimize our solution, some of them described theoretically. In section 2 the Neon solution is described. Section 3 is dedicated to the interactions between ARM and DSP. In section 4 the general optimizations at compile time are described. The results are presented in section 5. The final conclusions are described in section 6.

## 2 ARM Optimizations

### 2.1 Neon Vectorized Instruction Set

#### 2.1.1 Vectorized calculations

Vectorization is a type of parallel computing where a single process performs multiple tasks at the same time. This is done by packing a set of elements together into an array and operating on them all at once. It's especially useful for multimedia purposes such as gaming, image processing, video encoding/decoding and much more. In order to vectorize our execution we used NEON's Intrinsics, which form an intermediate layer between assembly and C. They're a tad bit confusing, and make the C code more complex

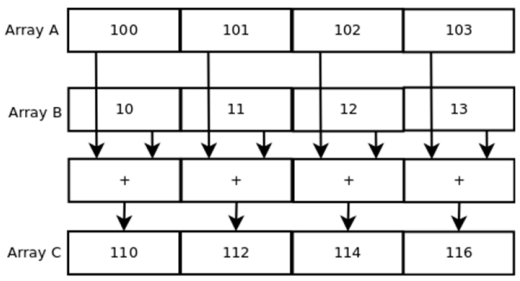



Figure 1: The picture shows how the vectorization is performed.

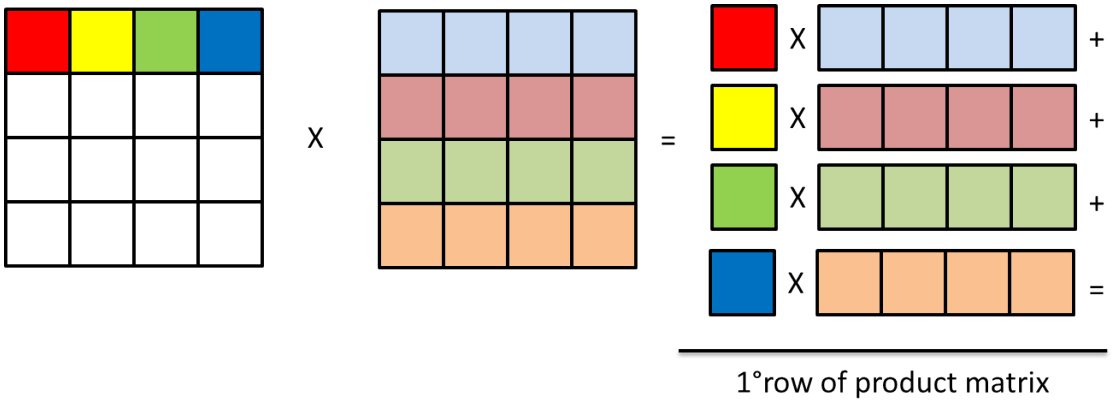



Figure 2: The picture shows the algorithm used to improve the performance of our execution. Each element of each row of the first matrix is multiplied by the "corresponded" row of the second matrix. The sum of the result vector corresponds to the product matrix final row.

but thanks to the vectorization, it is possible to execute 4 operations (on integer number) per time in parallel. This implementation gives an incredible speedup to the execution of our program, also because we were able to use a smart algorithm illustrated in "*http://www.fixstars.com/en/news/?p=125*". It is well known that the matrix multiplication is a typical example of possible parallel execution; taking advantage of this characteristic we multiplied every single element on the first row of matrix1 for the entire corresponding row of the matrix2 (dividing the row in few vectors composed by 4 elements). The second step was to sum all these "product" vectors. The result is the first row of the product result. Repeating the procedure for all the rows of matrix1, we were able to increase the speedup upto 6x times faster than the baseline. Figure1 illustrates the logic implemented with NEON's Intrinsics.

#### 2.1.2 Register use

There are 16 registers of 128 bits each to store four 32 bits vectors. We only used two of the available sixteen registers to store vectors of our optimized C code. The ideal execution should use all of these register in order to further increase the speedup. It could limit the access to the memory, because the write action will be performed only when all the registers are full. Unfortunately we were not able to implement this solution and check how much it could increase the speed of our code.

### 2.2 Cache

The BeagleBoard has some advantages when it runs powered by an external power supply, one of these being the ability to enable the Instruction Cache. The presence of the Instruction Cache helps the execution to store instructions and execute them faster. One other benefit is the speed of the processor, which runs at 720 mhz instead of 500 mhz when powered externally. We always kept the BeagleBoard connected to the power by the external cable. Data cache is enabled by modifying the boot file. The boot file is altered using the RS232 connection on the board in combination with an ftdi cable. In the next assignment we will try to get advantage of this resource, because till now we didn't have enough time to make it in practice.

### 2.3 ARM Compiler Optimization

Compiler flags could produce some speed up. The compiler flags next to -O is -O3 which turns on all the flags specified in -O and -O2 plus a lot more. The biggest gain will probably be upgrading the free compiler for the Compiler 6 from ARM itself, it is not free like the linux compiler but it fully exploits the architecture since ARM itself makes the chip, which means a –vectorize flag to optimise NEON use and a -Otime flag to optimise for fast execution time.

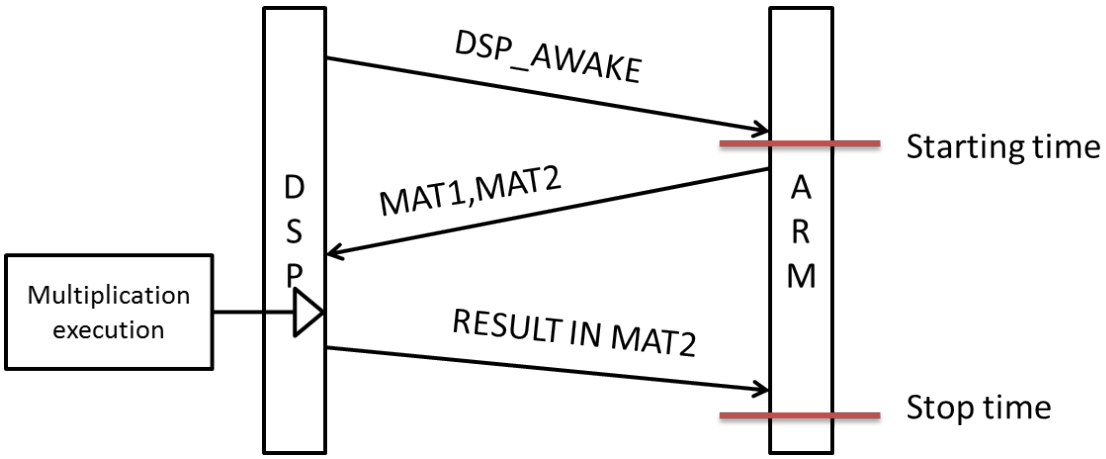



Figure 3: The figure shows how the communication between the ARM and DSP is done.

# 3 DSP optimizations

## 3.1 Matrix to DSP

Using the DSP , to perform the multiplication of the matrices, can improve the execution time, since DSP is a fast processor built specially for such processes. However, there is a need for communication between the ARM processor and the DSP, that will cause some overhead.
To use the DSP first we need to establish the communication between ARM and DSP. The two matrices are created in the ARM processor, and they are sent together, in a single message, in DSP. Initially we tried to sent the two matrices in two different messages. When doing this, we verified that the two matrices were sent correctly to the DSP, but the multiplication result that was sent back to the ARM , was wrong. We suspect that using two different messages, will not significantly improve the performance, because of the handling of creating a new message and the allocation of memory needed. After the DSP receives the two matrices, the multiplication is done, and the result is sent back to the ARM.
Section 5, shows that the execution time increases, when the size of the matrices increase. Unfortunately, we did not manage to do more optimizations on the DSP, due to lack of time. We would like to explore the DSP, and try to run it in parallel with ARM, by performing on part of the calculation in the DSP and some other part in the ARM.

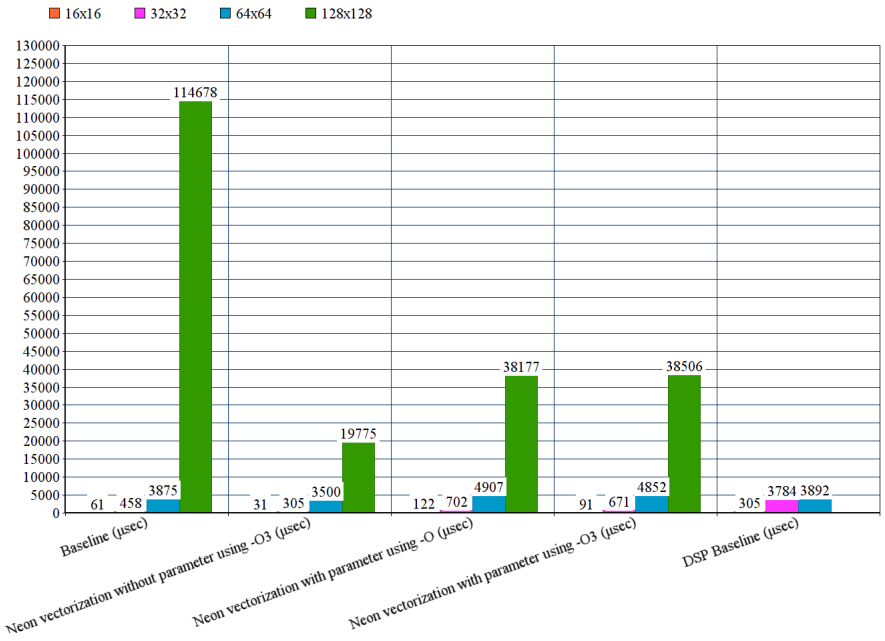
# 4 General optimizations

The BeagleBoard has an ARM v7 rev3 chip or v7l in short. It clocks at 720MHz according to the specification sheet and it was researched whether overclocking the board was an option to optimize the calculation of matrices. Although this is not a direct approach in optimizing the process, we looked into how feasible such a process would be. Overclocking is increasing the timing of the clock, making it run processes faster. The obvious advantage is a sudden surge in processing capabilities and faster response times. But all this would only be achieved only with disadvantages like voided manufacturer warranties and a risk of damaging the board. Also since there is nothing academic about achieving such optimizations, we decided not to use such an approach considering the amount of time at hand as well.

# 5 Results

The following results are measurements we did on different matrix sizes.

| Matrix Size | 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Runtime Baseline (msec) | 61 | 458 | 3875 | 114678 |
| Runtime Neon Vectorized without parameter -O3 (msec) | 31 | 305 | 3500 | 19775 |
| Runtime Neon Vectorized with parameter (msec) -O | 122 | 702 | 4907 | 38177 |
| Runtime Neon Vectorized with parameter (msec) -O3 | 91 | 671 | 4852 | 38506 |
| Runtime DSP baseline (msec) | 305 | 3784 | 3892 | — |




Speed up, under the minimal requirements, is not reached for smaller matrices. We see from the results that our optimization only have effect when the matrices become larger then 64.

We can see that the Neon vectorization of the C code has the best speed up. For the largest matrix, we see a speed up of 3x when we used the option of providing the dynamic matrix size. When the matrix size is not dynamic, the calculations are a lot faster. This has to do with the allocation of the size. When the size is fixed, the compiler at forehand allocates memory. With the dynamic size, the compiler needs to do this at runtime which consumes more time.

Using the compiler flag -O3 does not have any significant speed up. A lot of the options that are enabled do not have any effect on the small code we run for matrix computation. For example there are not a lot of functions to be inlined.

The results for DSP shows that no big speed up is reached when purely executing all the calculations on the DSP. As expected the communication between the ARM and DSP gives a lot of overhead such that the overall program is busy with handling message to and forth the DSP . Notice, that when using the max size of 128, the DSP is not working . The message containing 2 matrices of size 128 is to large. A solution could be to split up the matrices and send only a part of them. This option needs to be explored further. Also notice that a matrix of size 64 has a better runtime then a matrix of size 32.

# 6 Conclusions

Using a heterogeneous core is important when the goal is optimizing the execution time, since we can exploit each part of the core more effectively. As we have seen, the size of the input data plays a critical

role in the performance . As the size increases, using an hybrid solution with vectorization and DSP we get closer to the theoretical upper-bound as defined by Amdahl's Law.