Embedded Lab Assignment 1

Haji Akhundov
h.akhundov@student.tudelft.nl

Misael Hernandez Leal
m.a.hernandezleal@student.tudelft.nl

Fei Tan
f.tan@student.tudelft.nl

Koray Yanik
k.i.m.yanik@student.tudelft.nl

Muneeb Yousaf
m.m.yousaf@student.tudelft.nl

May 8, 2015

Abstract

I am an empty abstract. Please fill me.

1 Introduction

The given reference implementation is that of a matrix multiplication program. This process consists of a large number of multiplications and additions that can be highly parallelised.

2 DSP

The *OMAP3530* general purpose processor (from now on referred to as the *GPP*) contains a *TMS320* digital signal processor (*DSP*).This core is optimised for using multiple calculation channels simultaneously: it is a *VLIW* (Very Long Instruction Word) processor that can utilise eight functional units at the same time. More specifically, we can use six ALU's and two multipliers per instruction. Another interesting feature is that each multiplier can perform two 16 x 16-bit multiplications as well, giving us potentially four multiplications in parallel.

The GPP unit has to first start up the DSP core, give it an executable to run and afterwards detach the core as well. Furthermore, the requirements specified that the matrices to multiply are generated on the GPP core as well (which is of course not a bad idea because the GPP is way more flexible: it can easily be changed to read the matrices from a file or standard input, and we can offload a part of the work to the GPP itself and also the *NEON* as we will discuss later as well).

2.1 Communication

This however brings us to one complication: the matrices have to be moved to the DSP core, while the (partial) end result has to be sent back to the GPP. Our first goal is thus to set up communications and send matrices back and forth between the two cores.

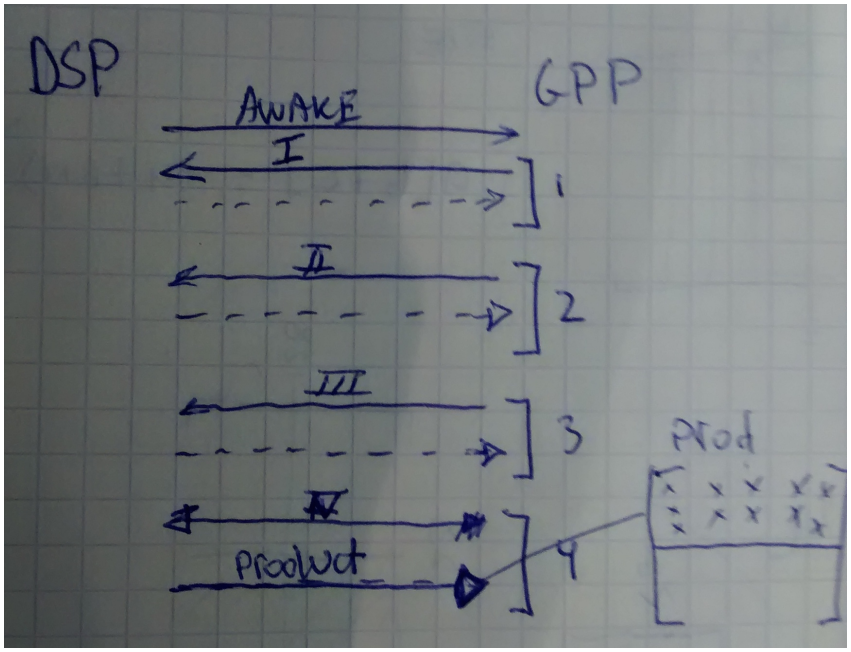

Figure 1: Describing the communication process between the cores over time

Unfortunately the communication API is slightly verbose and hard to handle. For example, the sender has to allocate memory for a message, sends it and the receiver has the possibility to send it back, or free the message. If we opted for the latter and re-allocate every message we would always crash the DSP core. The former option would mean that if we wanted to send more than one message directly from the GPP, the DSP would have to send the entire message back first. We however observed that the communication overhead is negligible, so we decided to always re-use the same message. The general communication time-line is described in figure 1. One observation in this time-line is that we send data four times. The reason why will be discussed in the next section.

2.1.1 Message Format

We used the existing reference communication implementation and it's message structure, but we extended it with our own matrices. The source matrices are 16 bit, but the result should be at least 32 bits. To accomadate for both situations in the same control message structure, we opted to use a *union* structure so we can either send two 16bit matrices or one 32bit matrix at a time. The structure is given in figure 2. One thing we had to do was determine the maximum matrix size we could assign, such that

it would fit in one message. This message would be stored inside a section of *APP_BUFFER_SIZE* large, thus our message structure has to fit in there. *TODO: Add calculation for maximum size here.* Because the application has to support matrices up to 128*128 big, we have to send up to $4 * 128 * 128 = 65.536$ bytes to the DSP. This means we need to send at least four messages in this case.

```
#define ARG_SIZE 256

struct mat2x16 {
  int16_t mat1[SIZE][SIZE];
  int16_t mat2[SIZE][SIZE];
};

struct mat32 {
  int32_t mat1[SIZE][SIZE];
};

typedef union {
  struct mat2x16 m16;
  struct mat32 m32;
} mat_t;

typedef struct ControlMsg
{
    MSGQ_MsgHeader header;  // 20 bytes
    Uint16 command;       // 2 bytes
    Char arg1[ARG_SIZE];  // 256 bytes
    mat_t mat;        // 4 * SIZE * SIZE bytes
} ControlMsg;
```

Figure 2: The message structure we used

3 NEON

To improve the performance of media and signal processing, *NEON* SIMD technology is implemented in *Cortex-A8* core of *OMAP 3530* GPP. *NEON* SIMD technology, which is also known as Advanced SIMD extension, takes the advantage of parallel operation to achieve the speed up.

3.1 SIMD

To understand NEON technology, the idea of SIMD is introduced at first. SIMD (Single Instruction Multiple Data) describes a way to perform the same operation on multiple data with same type and size in a single instruction. The idea of parallel operation comes from the fact that most of multimedia data are 16-bit or 8-bit wide, while the general purpose registers are 32-bit wide. To effectively utilize the space of registers, simultaneous computation is developed.

3.2 NEON Technology

NEON technology, as the Advanced SIMD extension in *Cortex-A8*, performs SIMD operations in group. NEON instructions operate on vectors stored in 64-bit or 128-bit registers, then vectors of elements with same type can perform the same operation on multiple items at the same time. The figure below shows how multiple items are computed simultaneously.

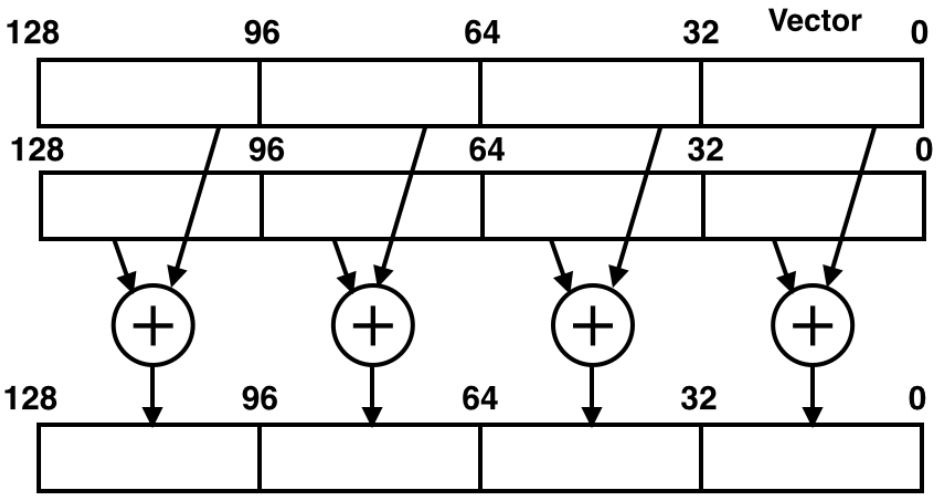

Figure 3: Parallel computing based on NEON

3.3 Hardware Features

NEON architecture has the following features:

1. *16-Entry instruction queue*
2. *32 x 64-bit general purpose registers in register file* These registers can alternatively be viewed as 16 x 128-bit registers
3. *6-stage execution pipeline* NEON supports either integer of single precision floating point execute pipeline.
4. *Load/store and permute pipeline*
5. *12Entry load data queue*

3.4 Implementation

To enable the build-in intrinsics of NEON, *mfpuñeon* is used during compiling time. Also, header file *arm_neon.h* is included to support NEON intrinsics in the c file. In our case, the incoming message contains the matrix with data of 16-bit wide, and then after calculation, when the final outcome is sending back to GPP, data size is 32-bit to avoid overflow. In the following, NEON intrinsics that are used for parallel computing in our experiment are explained.

3.4.1 Vector Data Type

Neon defined its own data type for multiple data operation, the format is given as: **<type ><size >x <number of lanes >_t**

For example, the data type we are going to use in NEON is *uint32x4_t*, which means the vector has four lanes, with each of the them containing an unsigned 32-bit integer.

3.4.2 NEON Intrinsics

NEON intrisics provide groups of functions for operation. In our case, functions related to load, multiplication and addition are used.

1. *uint32x4_t vmovq_n_u32(uint32_t value)*

   This intrinsic loads all lanes of vector to the same input value. The input value is an unsigned 32-bit integer, while the four lanes being loaded each contains an unsigned 32-bit as well.

2. *int32x4_t vld1q_s32(__transfersize(4) int32_t const * ptr)*

   This intrinsic loads a single value from memory to all lanes. The data stored in memory is signed 32-bit integer, while the four lanes each contains a signed 32-bit integer.

3. *int32x4_t vmlaq_s32(int32x4_t a, int32x4_t b, int32x4_t c)*

   This intrinsic multiplies b by c, and accumulates the result with a in all four lanes. The final results are then stored in four lanes as well.

In order to fully utilize the Neon resources, we have used following algorithm to compute the matrix multiplication. The following illustration shows one rows of results. The input matrices are following:

$$\begin{pmatrix} x1 & x2 & x3 & x4 \\ x5 & x6 & x7 & x8 \\ x9 & x10 & x11 & x12 \\ x13 & x14 & x15 & x16 \end{pmatrix} \times \begin{pmatrix} y1 & y2 & y3 & y4 \\ y5 & y6 & y7 & y8 \\ y9 & y10 & y11 & y12 \\ y13 & y14 & y15 & y16 \end{pmatrix}$$

The output matrix would be : =

$$\begin{pmatrix} x1y1 + x2y5 + x3y9 + x4y13 & x1y2 + x2y6 + x3y10 + x4y14 & x1y3 + x2y7 + x3y11 + x4y15 & x1y4 + x2y8 + x3y12 + \\ - & - & - & - \\ - & - & - & - \\ - & - & - & - \end{pmatrix}$$

In the output matrix , the elements with the same color are multiplied by the Neon at the same time. In this way, Neon produces the results of 4 multiplications at the same time. In the above output row, x1y1, x1y2, x1y3 and x1y4 are produces by the neon at the same time. But these are just the partial results for the first row of output matrix. In order to get the complete results for the first row these partial results should be added to the other partial results of the same row. This complete task of multiplication and addition is done by using Multiplication and addition functional unit of neon. The conceptual diagram of this multiplication and addition is shown in the following diagram:

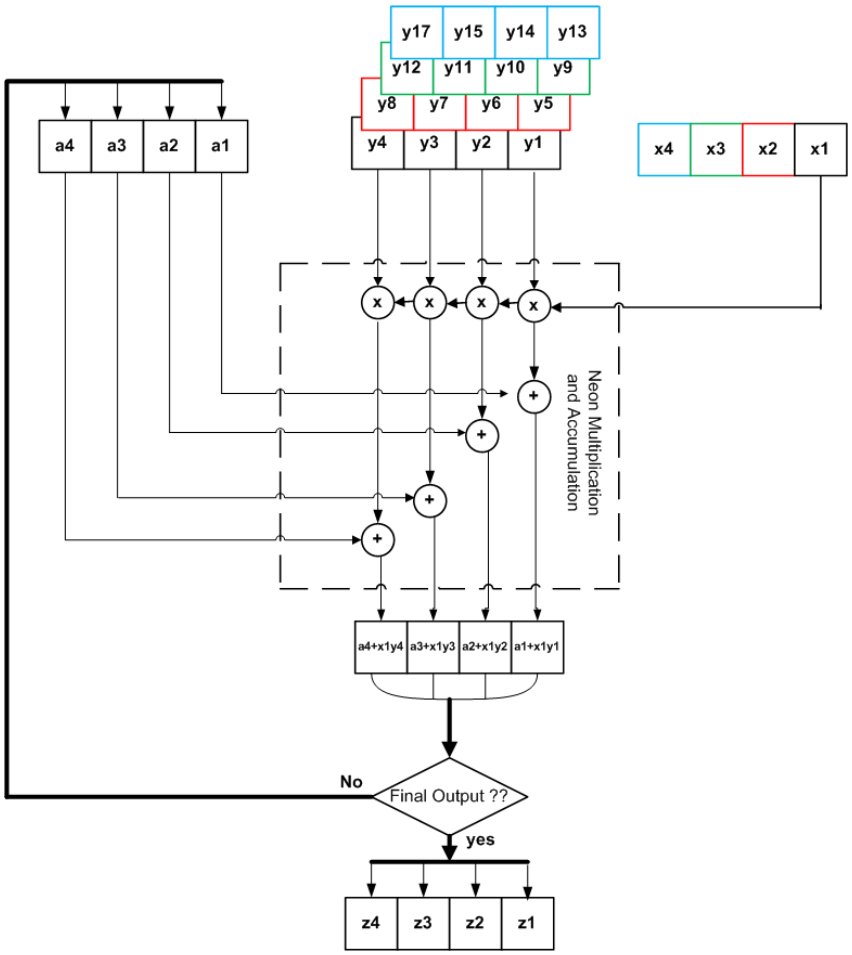

Figure 4: Parallel Multiplication and Addition with Neon

4 Integration

Discuss how we split and balance the workload here.

5 Conclusion

Let's rephrase a general conclusion here.

6 Discussion

Discuss possible improvements and pitfalls here.

References

[1] Example reference, please delete me. `http://www.example.com`. Accessed: 2015-05-08.