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 5, 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.

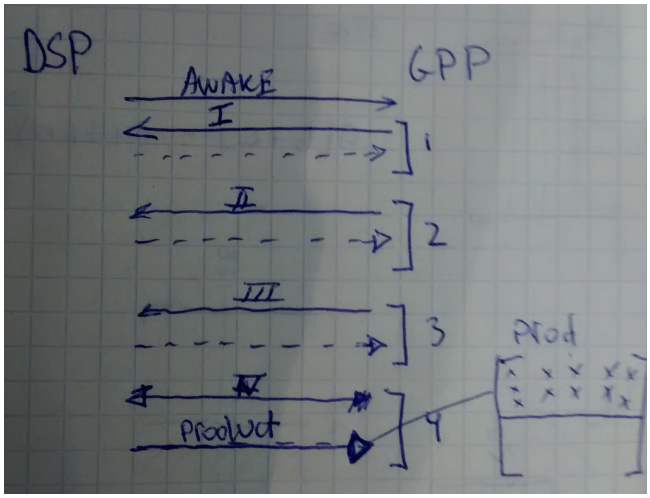

Figure 1: Describing the communication process between the cores over time

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 two matrices. 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. We thus decided to send a quarter of each matrix in each message.

```
#define ARG_SIZE 256

typedef struct ControlMsg
{
    MSGQ_MsgHeader header;        // 20 bytes
    Uint16 command;                    // 2 bytes
    Char8 arg1[ARG_SIZE];         // 256 bytes
    int mat1[SIZE][SIZE];         // 4 * SIZE * SIZE bytes
    int mat2[SIZE][SIZE];         // 4 * SIZE * SIZE bytes

} ControlMsg;
```

Figure 2: The message structure we used

3 NEON

Discuss the advantages of using the NEON core here.

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.