OVERVIEW OF EMBEDDED DSP DESIGN

Iain Hunter

Texas Instruments Limited
800 Pavilion Drive, Northampton, NN4 7YL, UK
phone: +44-1604-663181, email: i-hunter1@ti.com
www.ti.com

ABSTRACT

Many researchers are looking to take their work from a simulation environment and implement it on an embedded DSP platform. This is primarily driven by the need to demonstrate that the research is viable commercially. This paper will provide an overview of the different types of hardware and software development platforms that are available. It will then provide a summary of the software design techniques that are required to maximise the efficiency of code on current embedded DSP platforms. These will be demonstrated using an implementation of a Canny Edge Detection algorithm on a DM6437 Evaluation Module.

1. INTRODUCTION

Once the decision has been made to implement an algorithm on an embedded platform there are several major criteria that need to be evaluated. These include:

- What is the Input/Output format of the data? If it is an existing standard such as video then there should be no problem. If it is very high sample rate data or an unusual format then it will probably either need custom analogue interfacing hardware or the use of a board with an FPGA that can be programmed to do the pre-processing of the data.
- Is the DSP application standalone or does it need to be networked? If it needs to be networked then this drives towards a dual processor solution such as an ARM +DSP as the ARM will come with an OS to provide all the connectivity required.
- How much effort can be put into dealing with the software issues that occur on an embedded DSP? These include modifying drivers, optimising the software, using OS features such as semaphores, logging and solving the real time problems in a multi-tasking environment. If the answer is none, then the only choice is the use of an auto-code generation tool from a simulation environment such as Matlab-Simulink®. This implies the use of a hardware development platform that is supported by this tool using a Target Support Package.
- The final criteria is often the most critical one, overriding the technical issues described previously. What is the budget that can be spent on this tooling? Where this is the key criteria then the only realistic option is the use of a standard DSP Development Platform or Starter Kit that will be bundled with a software development environment.

2. AVAILABLE DEVELOPMENT PLATFORMS

This section will review in more detail the different types of development platforms that are available and the trade offs involved in their use.

2.1 Combined FPGA + DSP Platform with Automatic Code Generation

These fully featured development platforms are manufactured by companies such as Sundance[1] and Lyrtech[2]. In general they combine high speed data acquisition front ends (>100MSPS) with an FPGA and DSP and all the software tools to auto-generate code from a model based design environment. These manufacturers supply a variety of boards with different combinations of acquisition modules and processing resources targeting different end applications. For the researcher they allow the architecting and partitioning of the algorithm to suit the capabilities of the different processing devices. The FPGA provides the flexibility and parallelism to pre-process high bandwidth input signals down to a data rate that can be accommodated by the DSP. The DSP can then implement more complex processing on the pre-processed signal. This ability to pre-process large quantities of data makes these boards ideal for research in high performance areas such as Software Defined Radio or Medical Imaging.

The complexity of the hardware required for these applications means that buying a Commercial Off The Shelf (COTS) platform is the only practical option. On the software side the use of auto-generation of VHDL and C code allied to the use of Target Board Support Packages completely abstracts the user from low level programming.

This leaves researchers free to focus only on algorithm design problems with the main optimisation being on the top level partitioning between FPGA and DSP. The primary disadvantage is cost.

2.2 DSP Platforms with Automatic Code Generation

This type of environment uses standard DSP only development platforms from companies such as Texas Instruments (TI). This limits the standard interfacing options to the basics such as video (Standard or High Definition) or audio.

The software support comes from the Matlab® Target Support Package TC6 which supports all of TI's single core DSP

C6000 development platforms. The DM6446 and DM6467 ARM+DSP boards are not supported, as the Target Support Package tools support only the DSP/BIOS Real Time Operating System (RTOS) environment and not the Linux on the ARM9.
The software flow involves the auto-generated algorithm code and appropriate board drivers being compiled within TI's Code Composer environment. This step allows the developer to manually intervene in the process to optimise the code, call other optimised library functions, use specific DSP/BIOS features such as semaphores or change driver code. This is of course also possible on the combined DSP and FPGA platforms described previously. This flexibility allows the developer to invest as little or as much time into the final optimisation stages as required.
The cost of the auto-generation tools buys the time saved in getting the initial system working. If it provides an acceptable performance then the code generation can be fully automatic. The tools are also able to integrate some optimised libraries for standard functions. The trade off for the auto-generation tools is that as more time is spent on manual software optimisations, the benefit of the quick development cycle that is being paid for is lost. These platforms are ideal for proof of concept projects that require a fast turnaround.

2.3 Traditional DSP Platforms

This type of environment takes standard DSP development platforms and all the driver and example application software and then builds a system from the bottom up. The algorithm will be written from scratch in C with the translation from the initial simulation environment done manually.
As the automatic tool generation options do not cleanly support the ARM + DSP products such as DM6446/DM6467 or OMAP EValuation Modules (EVMs), this is the only real option on these platforms. The auto-generated DSP code could be packaged for use on these platforms but this would involve manual integration.
On some specific platforms such as TI's TAS3xxx Audio processors a free development environment called PurePathStudio integrates a Graphical interface for routing data between blocks and a traditional C code development environment for developing algorithms.
As developing code in C is the lowest cost approach, the reality is that this is the most common method used in research.
The rest of this paper will describe in detail the major optimisation techniques that are used on C64x+ devices and then provide example benchmarks using the Canny Edge Detection algorithm as an example. These techniques can be applied in the manual intervention stages in CCS on any of the platforms described here.

3. OPTIMISATION TECHNIQUES ON C6000 DSPS

This section will summarise the major areas of optimisation that will significantly improve the performance of the code execution and take advantage of the architecture of the devices. There are two main core variations available now. The C64x+ core is a fixed point core and is found in the 64xx products. The C67x core is fixed and floating point and available in the TMS320C674x products.

3.1 Optimising Code Execution

The primary aim here is to minimise the number of instructions that are actually executed and to maximise the use of, and minimise the disruption to, the deep pipeline on the C64x+ cores. The recommendations on coding techniques that follow are general guidelines that are applicable to any code that is written.

3.1.1 Choosing the correct data type

The majority of the platforms use DSPs with the fixed point C64x+ core. Frequently when algorithms are initially ported from a simulation environment the DSP code inherits the use of floating point variables. This is very inefficient as every mathematical operation has to be carried out with a function call to a floating point emulation library. In many cases the algorithm has no explicit requirements for floating point arithmetic, and in fact for video applications the raw data type is 8 bit integer. In these cases the simulation model should be coded in fixed point arithmetic so that all the fixed point implementation issues are managed before the code runs on an embedded DSP.
If an application really requires significant floating point operations it should be implemented on a C67x core.

3.1.2 Understand software pipelining

Understanding the principles of software pipelining is the key to writing code that makes the most of the C64x+ core. The C64x+ core consists of two banks of 4 processing units each with 32 registers as shown in Figure 1.

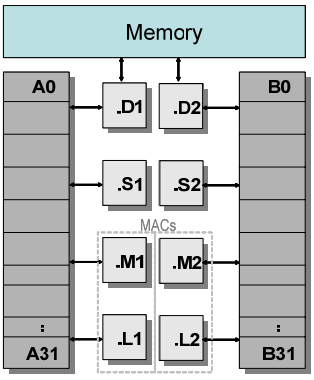

Figure 1 – C64x+ Core Architecture.

The D units are primarily responsible for Data moves to and from memory, the S units for program flow, the M units for multiplication and L units for arithmetic operations.
The optimal efficiency for code is achieved when all 8 units have work to do every cycle. The way this can be achieved on typical DSP code is via software pipelining. This is illustrated in Figure 2 which shows a typical for() loop to imple-

ment a Sum of Products. This figure also shows a pseudo code implementation of the loop in the software pipeline. Each iteration of the loop involves a load of data from memory, a multiply, an add and loop control with a branch. In pseudo code it takes 4 cycles to implement a single iteration. Software pipelining takes advantage of the fact that in cycle 2 the L units are free to read in the data for the second iteration. Similarly, in cycle 3 the M unit is able to multiply the second iteration's data while the D units are loading the data for the third iteration. The fourth cycle is the first time that the pipeline is fully loaded.

```
short mac(short *m, short *n, int count)
{
  for (i=0; i < count; i++)  {
   sum += m[i] * n[i]; }
```

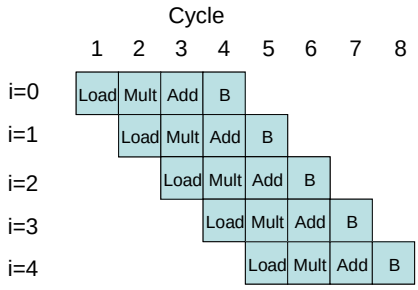

Figure 2 – Software Pipelining

The goal is to write code that is in this fully loaded pipeline state for as much of the time as possible. This can be achieved by following some basic rules [3].

- The algorithm must be written in the form that the processing takes place in for() loops.
- Give the compiler as much information as possible about the loop and the data alignment used for data.
  The following Compiler Directive can be used to give the compiler details about the possible number of iterations in the loop.
  #pragma MUST_ITERATE()
  In particular specifying the multiple parameter will enable the compiler to work out if it will be more efficient to unroll the loop so that each loop in the pipeline actually implements 2 or 4 etc iterations of the code.
  The _nassert() intrinsic can be used to tell the compiler the expected alignment of data pointers which allows it to optimise memory accesses.
- The loop should not contain function calls as these disable software pipelining. Note that the run time support library will implement a modulo (%) or divide (/) operation on fixed point variables and all floating point operations on a fixed point core with a function call.
- Use C instrinsics to access specific assembly language instructions cleanly from C. Instrinsics provide a method to take advantage of specific operations such as saturated arithmetic or Single Instruction Multiple Data (SIMD) packed instructions that have no equivalent in the C language.
- The basic unit of data size for an instruction is 32 bits which means that a single ADD instruction takes one cycle and three registers whether it is operating on two 8 bit, 16 bit or 32 bit values. The C64x+ core adds support for packed data which allow either 2 16 bit values or 4 8 bit values to be added in a single cycle with the ADD2 and ADD4 instructions respectively.
- Similarly a load of a single data operand takes one instruction whether it is 8 or 64 bits. So with 8 bit data the use of a 64 bit pointer in C will allow 8 consecutive 8 bit values to be read into registers in a single instruction. In many cases using the #pragma MUST_ITERATE() directive and _nassert() instrinsic will give the compiler enough information to optimise code with packed data operations.
- Where possible think laterally about how to implement an operation in a way that is easier for the processor to implement. As an example a modulo operation on a base that is a power of 2 such as,
  i = count % 256;
  can be replaced by the more efficient
  i = count & 0xFF;
  which will be implemented in a single instruction.

3.2 **Use Optimised libraries**

For many fundamental Signal Processing operations optimised libraries are available from TI that can be included in an algorithm. These include:

- DSPLIB [4] - this provides operations such as filtering, correlation and FFTs as well as general vector and matrix operations.
- IMGLIB [5]– This provides basic image processing blocks such as filtering, thresholding, correlation, convolution and morphology with both C64x+ libraries and Matlab Simulink blocks available.
- VLib [6] – This provides the basic image processing blocks required for Video Analytic applications.

3.3 **Optimising Data Accesses**

The other critical aspect to system design is optimising the algorithm's data memory accesses. On most C64x+ devices the core will be running two or three times faster than the large external DDR memory. As well as this difference in relative speed the memory interface and DDR designs introduce significant delays on the first data read in a sequence. This particularly impacts random data accesses as each read is then the first.
The C6000 devices design mitigates this performance impact through the use of a two level cache architecture. There will be a small L1 cache for data and programme that runs at the same speed as the core as well as a larger L2 cache that usually runs at half the core speed. Some of the L1 and L2 memory can also be configured as mapped memory which allows the core to directly access it as fast memory. The DM6437 can configure either 48 or 64kbytes of the 80kbytes

of L1 memory to be mapped and between 0 and 128kbytes of the 128kbytes of L2 memory to be mapped.

3.3.1 *Using the Cache*

On a DM6437 the default RTOS software initialisation will enable the maximum L1 and L2 caches. In this case the developer does not need to worry about memory access times as the caches hide the physical memory architecture. When the cache is being used the key optimisation is to ensure that data arrays are aligned on 128 byte boundaries to fit with the cache line lengths. This minimises the effect of cache misses.

As an example a Canny Edge Detection application was built using the VLib[6] library on a DM6437 EVM. It captured PAL D1 images from a camera, implemented the Canny Edge Detection Filter and then displayed a D1 black and white edge D1 map on a display. The algorithm shown in Figure 3 illustrates the pre and post processing steps required to deal with 16 bit YUV data in the capture and display drivers and the conversion to 8 bit Luma data for the Edge Detection.

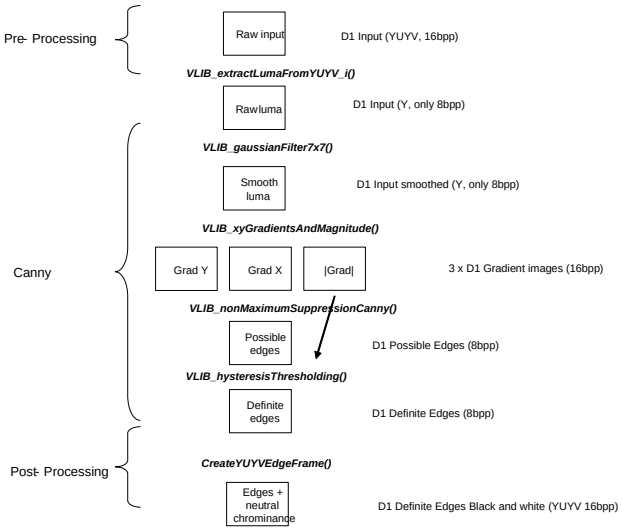

Figure 3 – Canny Edge Detection Algorithm

The timings of the individual blocks in the algorithm when using the cache to manage all data in external memory are given in Table 1. They show that with a total processing time of 46ms per frame for the Canny and 15ms per frame for the pre and post processing, it is possible to achieve a frame rate of 16 fps.

3.3.2 *Using the available Internal Memory*

Some applications can slice up the input data into smaller discrete blocks which can be operated on independently. This processing of the data in slices can in most cases be more efficient that just using the caches. The reasons for this is that the small slices of data can be put into the L1 mapped RAM from external memory in a single Direct Memory Access (DMA) copy and then accessed by the core with no penalties or cache misses. The overall algorithm needs to be split up into the functions that can operate upon sliced data and those that can't because they need access to the entire data set. For the Canny Edge detection algorithm the Gaussian Filtering, Gradient Calculations and non-Maximal Suppression blocks all operate upon discrete lines in the image and so can be sliced. Figure 4 shows the relationship between the number of lines in each of the data slices. In order to generate N lines of non-Maximally suppressed potential Canny edges, N+10 lines of raw image need to be used.

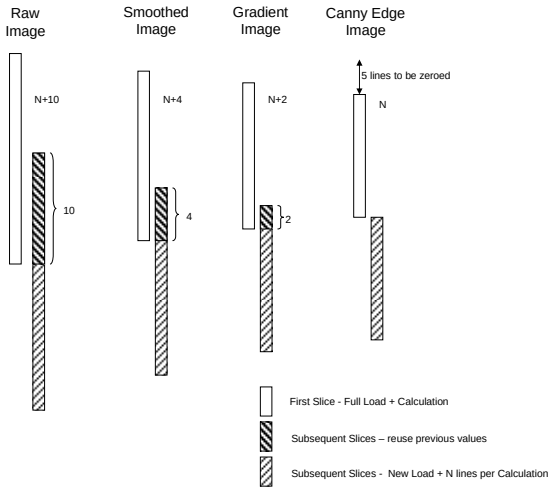

Figure 4 – Overlapping Slices in the Algorithm

On the DM6437 with 64kbytes of L1 mapped memory available the value of N=7 was used so that all the slices could exist in internal memory at the same time.

In order to use the slicing approach the algorithm is also responsible for using DMA channels to bring the N+10 raw image lines into memory, processing them and then storing the N lines of possible canny edges back to external memory. In addition it can optimise the use of DMA and core bandwidth by ensuring that where lines of data overlap between slices they are reused by the next slice. This ensures that each line of data is only fetched from external memory or calculated once.

The final stage of the Canny algorithm is Hysteresis thresholding. This function cannot be broken up into slices and so must use the cache to access full frames of data in external memory. As the sliced data is written back out to external memory the cache needs to be invalidated before the Hysteresis function is carried out to force the reloading of the valid data from external memory. The timings of each block using slicing are shown in Table 1.

	Slicing (ms)	Frame/Caching (ms)
Extract Luma from raw image	1.40	3.95
Fill Luma with Definite Edges	8.20	10.60
Total pre+post processing	**9.60**	**14.55**
Gaussian Filtering	5.00	6.75
Gradient Calculation	0.95	13.70
Non Maximal Edge Suppression	11.20	16.55
Edge Hysteresis	9.50	9.10
DMA+cache management	4.10	
Total Canny	**30.75**	**46.10**

Table 1 – Comparing Caching and Slicing Timings.

These results show that by using slicing the total frame rate that can be achieved increases to 24fps. The most significant improvement in performance is from 13.7ms to 0.95ms on the gradient calculation. This is because this function has the highest proportion of data accesses to instructions and most of them are writes. In the slicing method the time taken for the Edge Hysteresis function actually increases compared to the cached method. This is due to the fact that the cache has been invalidated and so there are no cache hits initially.
There is an overhead with slicing of 4ms per frame for DMA and cache management but this is in this case significantly outweighed by the performance improvement.

4. CONCLUSIONS

This paper has summarised the different approaches that are available for moving an algorithm from simulation to a real time DSP hardware implementation. The options are driven by the applications specific input/output requirements, the requirement to run an OS like Linux on an ARM core, the software effort required and the budget available.
All of the methods leave the option of optimising the code to improve performance by making better use of the DSP's instruction set and memory. The paper reviewed the major techniques for improving the efficiency of the code generated and the memory usage.

REFERENCES

[1] www.sundance.com
[2] www.lyrtech.com
[3] TMS320C6000 Programmer's Guide, SPRU198i, www.ti.com
[4] TMS320C64x+ DSP Little-Endian DSP Library Programmer's Reference, SPRUEB8B, www.ti.com
[5] TMS320C64x+ DSP Image/Video Processing Library (v2.0) Programmer's Reference, SPRUF30A, www.ti.com
[6] Vision Library (VLIB) Application Programming Interface, www.ti.com/vlibrequest