EPFL

CS-473: System programming for Systems on Chip

Practical work 6

DMA and compiler options

Version:
1.0

Contents

1 Introduction **1**
1.1 Prerequisites . . . 1
1.2 DMA and compiler options . . . 1

2 Exercises **2**
2.1 Prerequisites . . . 2
2.2 Compiler options that control optimization . . . 2
2.3 DMA instead of cache . . . 3
2.4 DMA-calculation overlap . . . 5

1 Introduction

1.1 Prerequisites

On moodle you find the file `pw6.zip`; it contains the complete source code of the brute force accelerated version of the Mandelbrot set and the software version. Download this file. This template contains a directory called `dma`, where the `makefile` can be found as for our previous PW.

1.2 DMA and compiler options

In last and this weeks theory we saw the DMA. In this practical work we are going to look into the influences of the different optimization levels, and how DMA-transfer in parallel to computation can help to speed-up our program. Please note that the VGA-controller *fetches* each line in a burst of `nrOfPixelsPerLine/2` 32-bit words.

2 Exercises

2.1 Prerequisites

The program that is contained in the zip-file is a conditional code that is dependent on the definition of `__REALLY_FAST__` (see `dma/src/main.c`). When `__REALLY_FAST__` is not defined the compiler will build the software-implementation of the fixed-point Mandelbrot algorithm. On the other hand, if `__REALLY_FAST__` is defined, the compiler will build the hardware-accelerated version of the Mandelbrot algorithm (the same we used in the last PW). The default is that `__REALLY_FAST__` is not defined, and hence when the program is compiled, the non-accelerated version is build.

To be able to build the accelerated version, there are two options:

- You can define `__REALLY_FAST__` in `main.c` as shown below. This method will only define `__REALLY_FAST__` in `main.c`, and not in all the other source files.

```
1 #include <defs.h>
  #include "fractal_fxpt.h"
3
  #define __REALLY_FAST__
5
  int main() {
```

- You can also define `__REALLY_FAST__` on a global basis by adding a compile option in line 7 of the `makefile` as shown below.

```
CFLAGS ?= -D__OR1300__
```

Before continuing with the tasks, build and test on your GECKO-board both versions. Please note that the profiling system is already set-up for the measurements to be done for the up coming experiments.

Important: for all the tasks we are not going to change the cache configurations, being:

- Instruction cache: 8kByte, 4-way set-associative.
- Data cache: 8kByte, direct-mapped, write-back.

2.2 Compiler options that control optimization

The compiler has several options that control the optimization of the given program. A complete list with explanations can be found here for gcc. In this task we are going to look into the influence of these options on size of the program and the execution speed.

The compile options that we are going to use are: `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Ofast`, and `-Oz`.

You can find the compile option on line 39 in the `makefile`:

```
1 BUILD = build-release
_CFLAGS += -DNDEBUG -Os
```

In this example the `-Os` option is used. For this experiment we are going to follow following work-flow:

1. Compile the program with the given compile option.
2. Upload the program to your virtual prototype.
3. Determine the size of the program by typing `*i` in the communication program.
4. Run the program by `**` and note the different values.
5. Reset the board, upload the program, and run the program again by `**` and note the different values.
6. Reset the board, upload the program, and run the program again by `**` and note the different values.

As we have 3 runs of the same program, we have a small "statistic".

Perform the measurements for both the accelerated and non-accelerated version of the Mandelbrot algorithm. Note the results in a spreadsheet (for example in excel or OpenOffice). What are the observations that you can make?

Hint: OpenOffice/Excel have the build-in function `HEX2DEC()` that can convert a hexadecimal string to a decimal value. By using the formula $= (HEX2DEC(A1) * 4)/1024$ in cell $B1$ will calculate the size in kBytes of the hex-string in cell $A1$.

2.3 DMA instead of cache

Our virtual prototype has a build-in DMA-controller in the scratch-pad-memory. As the scratch-pad-memory has a single-cycle access on the data-side, and our algorithm only writes to the frame-buffer (the pixels), it is perfect to use as intermediate buffer. By writing one line of pixels (hence 512 pixels) and then DMA this data towards the proper location in the framebuffer will, most probably, be way more efficient than using the data-cache.

The base address of the dma-controller is `0x50000040` with the following register-map:

Byte offset:	Word offset:	Function:	Description:
0	0	Memory address	Start address in the memory space
4	1	SPM-address	Start address in the SPM-space
8	2	Buffer size	Size in words (32-bit) of the buffer to transfer
12	3	Control/Status	Write: control register, read: status register

The control register (hence writing to word-offset 3) has following functionality:

bits31..10	bit9	bit8	bits7..0	Function:
don't care	0	0	burst size	Specify the bus burst size (0->1 word, 255->256 words)
	0	1	don't care	Start a DMA transfer from SPM to memory
	1	0		Start a DMA transfer from memory to SPM
	1	1		No operation

The Status register (hence reading from word-offset 3) contains following information:

Bit:	Information:
0	A 1 indicates that the DMA-controller is busy and cannot be programmed.
1	A 1 indicates that the DMA-controller faced a bus error.
2	A 1 indicates a memory address alignment error (lower 2 bits are not 0).
3	A 1 indicates a SPM address alignment error (lower 2 bits are not 0).
4	A 1 indicates an SPM-range error.

Some important information:

- If any of the bits 2..4 are 1, the DMA-controller will not react on the *Start a DMA transfer commands*. Hence no DMA-transfer is done.
- If bit 1 is 1, the DMA-controller aborted at a certain moment, as it faced a bus-error. There is no information at which address it faced this error.
- If bit 0 is 1, the DMA-controller will generate a bus-error in case of a write to any of it's registers.
- The SPM-range error is set in following cases:
  - The *start address in the SPM-space* is outside of the SPM-space.
  - The *SPM-address + Buffer size* is outside of the SPM-space.
- The DMA-controller will always generate an interrupt when it finished the transfer. For the following tasks we will not use the interrupt-driven approach, but the polling approach.

Remember: we have a 8kByte SPM located at `0xC0000000-0xC0001FFF`.

For your convenience, the file `dma/support/include/dma.h` already contains all required definitions.

For the following tasks we are going to use the compile option `-Os`, so please make sure that you modify the `makefile` accordingly!

We are going to start with the accelerated version of the Mandelbrot algorithm. So make sure that you specify `-D__REALLY_FAST__` in your `makefile`. As we are going to DMA on line-level (hence `SCREEN_WIDTH*2` bytes), define a buffer in the SPM-region for this. Note that the accelerated version of the Mandelbrot algorithm calculates 2 pixels at a time, and hence has an `uint32_t` access to the buffer. In this task we are going to use one single buffer in SPM, meaning that:

- We only have to specify one's the `Buffer size`.
- We only have to specify one's the `SPM-address`.

Extend `main.c` with all code required to:

- Use the SPM to write 1 line (512-pixels) in a SPM-buffer.
- Instantiate a DMA-transfer after 1 line has been written for bus-burst-sizes 255(1kByte), 63(256 bytes), and 15(32 bytes, D$ cache-line size).
- Wait through polling until the DMA-transfer has finished until processing the next line.

Run the program for the above stated bus-burst-sizes, and note:

- The run-time of the program.
- What can you observe in the graphical window where the image is presented?

An Interesting reflection before executing the program: What do you expect?

We can use the same method for the non-accelerated version of the Mandelbrot algorithm. Adding this is almost copying the changes made for the exercise above, with the difference that the non-accelerated version of the Mandelbrot algorithm uses an `uint16_t` access to the buffer.

Modify the function `draw_fractal` in the file `dma/task/src/fractal_fxpt.c` in such a way that it uses the DMA-transfer as described in the previous task. For this exercise use a bus-burst-size of 255(1kByte). Note the execution time. What can you observe compared to the accelerated version?

2.4 DMA-calculation overlap

In the previous experiment, we have seen that using DMA, and an SPM-buffer, can aid for this algorithm to:

- Improve performance.
- Reduce energy consumption.

Why it reduces energy consumption?.....

Hence when we do not wait by polling until the DMA-transfer is finished, but try to overlap calculations with data-transfer, this should give us even more benefit. This aspect we are going to evaluate in the third part of this exercise. To be able to exploit this technique, we need two buffers in SPM, namely:

- The line-buffer where the processor is writing the current line-pixels.
- The dma-buffer, that contains all the pixels that the processor already calculated for the previous line.

Hence we have the so-called double-buffer, or sometimes referred to as ping-pong-buffer, situation. The basic idea is that the previous line is dma'ed to the frame buffer, whilst a new line is calculated. This makes sense in the case that ..... (for you to determine :-)).

Perform all the changes on the accelerated and non-accelerated version of the Mandelbrot implementation to perform the experiments on this part.

Some help on this task:

1. Define 2 buffers, for example dma-buffer and write-buffer.
2. Write one line of pixels in the write-buffer.
3. Swap the pointers from the dma-buffer and write-buffer.
4. Wait until the dma-controller is not busy.
5. Instantiate a dma-transfer of the dma-buffer and go to point 2 above.