Group 6: Distributed Video Encoding using Dolphin PCI Express Networks

Skjalg Gustav Eriksen
University of Oslo
skjale@ifi.uio.no

ABSTRACT

In this paper we document our solution for a distributed c63 encoder. We split c63 encoder over two machines connected with PCIe interconnect, one machine will handle the I/O and the other handles the processing.

1 INTRODUCTION

This paper is the home exam 3 for group 6 in IN5050 at UiO. The assignment is to take advantage of distributed computing to accelerate the video encoding for the c63 encoder. We are going to distribute I/O and the encoding over two machines with PCIe interconnect. We will use a x86 machine to read and write and a Tegra machine to encode the frames. On the Tegra we have decided to use neon optimization we made in exam 1.

1.1 Sections

The remainder of this paper is organized in 5 sections. In section 2 we discuss the benchmarks to see what SISCI transfer is best suited to our needs. In section 3 We document and discuss our first implementation of a distributed c63 encoder. In 4 we document and discuss optimizing our first implementation. In 5 we evaluate our resutls. In 6 we conclude the paper.

2 TRANSPORTING DATA WITH SISCI

To get some idea of how fast the alternatives for data transfer with the SISCI API we did some benchmarks. With DMA (Direct Memory Access) we can expect transfer speeds on a foreman frame to be between 5571.19 MBytes/s and 5702.81 MBytes/s based on dma_bench. With PIO (programmable I/O) we can expect max 4770.67 MBytes/s shown in scibench2.

To synchronize we can either use PIO or interrupts. With Intr_bench we Benchmarked interrupts to an average round trip interrupt time to be 14.515 us. The PIO benchmark scipp shows PIO to have an avarage latency be between 2.318us-3.916us.

From these benchmarks we decided to use DMA to transfer image data and PIO for synchronization and small transfers. DMA offers greater transfer speeds when transferring bigger amounts of data while PIO offers low latency for synchronizing x86 and The Tegra.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

3 IMPLEMENTING A DISTRIBUTED SOLUTION FOR THE C63 ENCODER

3.1 Setting up PIO

The first thing we worked on was transferring some data used to synchronize with PIO. We used the SCICreateSegment(), SCIPrepareSegment(), SCISetSegmentAvailable() functions from the SISCI API on both the x86 and Tegra. This creates a memory segment that can be connected to with SCIConnectSegment(). We can then map this segment and and access the Tegras memory segment from x86 and x86s memory segment from the Tegra. For the mapped segment in tegra and x86 we used a struct comms we defined in sisci_variables.h, it contains a variable for command, width and height. The cmd variable is a uint8_t we use to send messages between x86 and the Tegra.

The first thing we needed was to transfer the width and height input parameter from x86 to the Tegra to make the cm data structure used in the c63 encoder. After the Tegra had connected to and mapped x86s memory segment we used a while loop that loops until x86 has set the cmd variable is set to CMD_DONE. Tegra can then read the width and height from the mapped segment.

3.2 Setting up DMA

To set up DMA we use the same first step PIO to create and setup a memory segment that can be mapped and connected to. We also needed to define a DMA queue SCICreateDMAQueue(), which can then be used to start a transfer with SCIStartDmaTransfer(). After starting a DMA transfer we can call SCIWaitForDMAQueue() to make the program wait until the transfer is complete.

With DMA we ran into more issues than we did with PIO. When we tried send a image over in a yuv_t struct which holds 3 uint8_t pointers that gets memory dynamically allocated to it with calloc() it would not work. The first fix we tried was adding the allocated size to the transfer size in the DMA queue with little success. We then defined a new struct similar to c63's yuv_t struct, but with normal arrays instead of pointers. Then we used Memcpy() to copy the contents of the image into our new struct that we then transferred with DMA, This worked.

We used the same strategy when making a struct to transfer the results after encoding, A struct with normal arrays for the residuals and macroblocks.

3.3 Main loop synchronization

After PIO communication and DMA transfer between x86 and Tegra is set up we move on to the main loop for both Tegra and x86. The overall structure of the x86 and Tegra main loop is shown in figure 1.

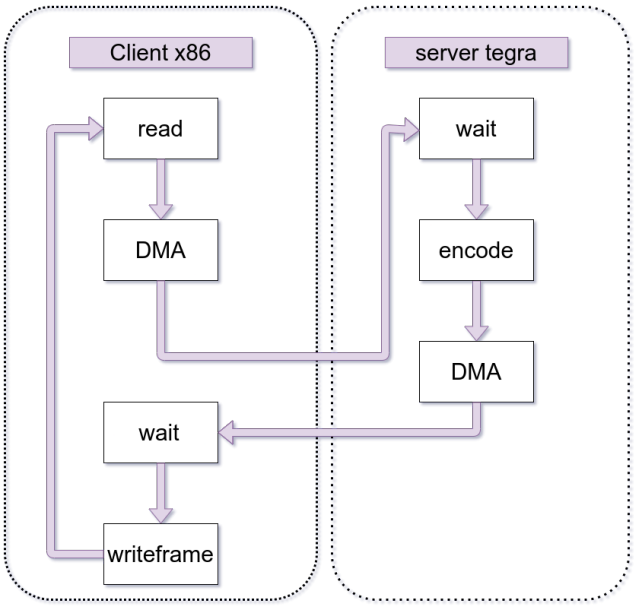

Figure 1: flowchart of program flow

```
while(1)
{
    local_cmd = invalid
    read_image()
    DMA transfer image          // transfer with DMA
    wait_DMA                    // wait for DMA to finish
    remote_cmd = done           // signal tegra
    while(local_cmd != done)    // busy wait for Tegra
    write_image()               // write frame
}
```

Figure 2: psudocode for x86 main loop

```
while(1)
{
    while(local_cmd == invalid) // busy wait for x86
    local_cmd = invalid
    encode()                    // encode
    DMA transfer image          // transfer with DMA
    wait_DMA                    // wait for DMA to finish
    remote_cmd = done           // signal x86 to continue
}
```

Figure 3: psudocode for tegra main loop

In x86 main loop the first thing we do is read the image, then send it with DMA. When the DMA transfer finishes we signal the Tegra to start encoding with remote_cmd=done. We then wait for the Tegra to set x86's cmd to done before continuing to write image. x86 main loop is shown with the psudocode in figure 2.

In The Tegra main loop we first wait for the ready signal from x86 with local_cmd. Once we finish encoding we send the results back and signal x86 with remote_cmd = done. The Tegra main loop is with psudocode in figure 3.

4 IMPLEMENTING A 3-FRAME PIPELINE

So far our solution only has 1 frame in flight, This means while the x86 reads, writes to file or transfers the Tegra is waiting in a busy wait loop. To investigate how much downtime the Tegra has we

```
while(1)
{
    // read if
    if(cmd_read)
    {
    read_image()    // read
    DMA()           // transfer image
    wait_DMA()      // wait for dma to finish
    cmd_read = 0    // signal frame ready to encode
    }
    // write if
    if(cmd_write)
    {
    DMA_pull()      // pull encoded image from Tegra
    wait_DMA()      // wait for dma to finish
    write_Frame()   // write frame to file
    cmd_read = 0    // signal x86 ready to pull frame
    }
}
```

Figure 4: psudocode for x86 main loop with 3 frames in flight

used time.h C library to time the busy wait loops. For the tractor video with each frame the Tegra has to wait for 0.016s and spends 0.007s transferring results to x86.

Ideally when efficiently transferring data between the x86 and the Tegra, the Tegra should be able to continuously encode frame after frame without downtime. To reduce this downtime we can have more frames in flight and transfer frames in advance. This way when its finished encoding the next frame is always ready. The assignment parameters allows us up to 3 frames in flight, with this we can make a pipeline where frame 1 is being read, frame 2 is being encoded and frame 3 is being written to file. And the last 0.007s drain the Tegra spends transferring can be done with using the x86 DMA engine to pull the results from the tegra. This way the Tegra can encode frame after frame without downtime.

At first we thought of using multithreading to have a read and a write thread on the x86 but we ran into some issues with passing mapped memory segment pointers as arguments to a thread, it would not work. We then decided it would not be necessary to complicate things with multithreading.

The solution we landed on was having two if statements in x86's main loop, one for read and one for write. These if statements are true/false checks on two int variables cmd_read and cmd_write. after the Tegra has read a frame and before its encoding it sets cmd_read to 1 which signals the x86 that it can transfer a new frame. Once encoding finishes it sets cmd_write to 1 signaling the x86 its ready to pull the encoded frame. To synchronize we still need busy wait statements in Tegras main loop. Before reading the next frame the Tegra will wait for cmd_read to be 0 and before overwriting the last result it will wait for cmd_write to be 0. However ideally the x86 is fast enough so the Tegra will not have to wait. Psudocode for x86 main loop is in figure 4 and for Tegra its in figure 5.

This solution worked nicely and the only downtime now remaining was the 0.016s for the very first frame to be transferred. To stress test the solution we tried to see how much downtime we would get when commenting out motion estimate/motion compensate in the encoder. There was still no meaningful downtime even with the heavy parts of the encoder commented out.

```
while(1)
{
    while(cmd_read)   // busy wait x86 to read
    cmd_read = 1      // signal ready for next frame
    encode()
    while(cmd_write)  // busy wait for x86 write
    cmd_write = 1     // signal frame ready to be pulled
}
```

Figure 5: psudocode for tegra main loop with 3 frames in flight

initial implementation	0m9,519s
3-frame implementation	0m8,983s
exam1 neon implementation	0m6,575s

Figure 6: foreman timed using linux time command

5 EVALUATION

Table 6 shows the final results, going from the initial implementation to the 3-frame in flight implementation we gained 0.536s. This speed up is very close to the total downtime we measured so its as expected.

Its a bit trickier comparing directly with the neon implementation from exam 1 since the run.sh script we use to launch our distributed solution will take a bit of time to set up and start the encoding. But overall the difference is not very large and i think it can be attributed to the run.sh script and the SISCI setup and connect the memory segments.

6 CONCLUSION

In this assignment we have learnt about the SISCI interconnect API and have practiced data transfers over PCIe with PIO and DMA. We have learnt about synchronizing distributed computing with busy wait. Finally we have learnt to efficiently use a 3-frame pipeline with a distributed program.