IL22I2 – Embedded Software: Technical Lab Report

Sebin Shaji Philip
sebin@kth.se

Prasanna Balaji
pbalaji@kth.se

SPECIFICATION

.

1. Application

1. SDF graph

Synchronous Dataflow (SDF) has the advantage that algorithms expressed as consistent SDF graphs can always be converted into an implementation that is guaranteed to take finite-time to complete all tasks and use finite memory, provided that individual SDF actors take finite time and finite memory to execute. Thus, an SDF graph can be executed over and over again in a periodic fashion without requiring additional resources as it runs. SDF graph also tells us about the data rate or token consumption, which will be useful in estimating optimal buffer sizes between actors.

The two steps in scheduling an SDF graph are:

1. compute the repetitions vector which tells how many times to execute each actor to bring the graph back to its original state, and

2. find an order of the firings using the repetitions vector that satisfies the data dependencies in the graph.

The SDF diagram of the application is in Figure A.

controlSig (mooreSDF):

The actor controlSig is a moore SDF that'll be used to enhance the edge detection by using the min, max pixel values of current image and two previous images. The nextstate function will take the average of (max-min) of current image and two previous images (initially these values will be 255). Then as images comes through the values will be populated and stored for future reference.

The moore SDF diagram of controlSig actor is in Figure B.

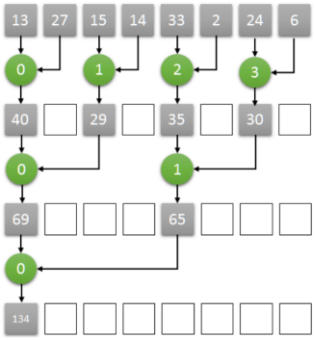
2. Parallel Constructs

The following parallel constructs are utilized in the project as detailed below:

Pipeline
Pipelining involves overlapping operations and executing different stages of them in parallel. In this project, a three-stage pipeline is employed across 3 cores, the stages being write image, process image and read image in this order of execution. The stages of the pipeline are mapped to different cores. CPU 0 handles the write image and read image operations, while CPU 1 and CPU 2 handle processing of different parts of the same image in parallel.

Map
Map refers to simultaneously executing the same operation on each part of the input. Parallelism through mapping is achieved in this project by having CPU 0 and CPU 1 perform the same operations (grayscale, resize, sobel etc.) on different images at the same time.

Reduce
Reduction is similar to interpolation where multiple values are gathered into one. This reduction, when simultaneously applied to different parts of the input can become a parallel construct with a positive effect on performance. In this project, 12 byte values (denoting 4 pixels) is reduced into 1 byte value denoting a single pixel (where converting the image to grayscale and resizing the image act as reduction operations). This happens in parallel on CPUs 1 and 2.



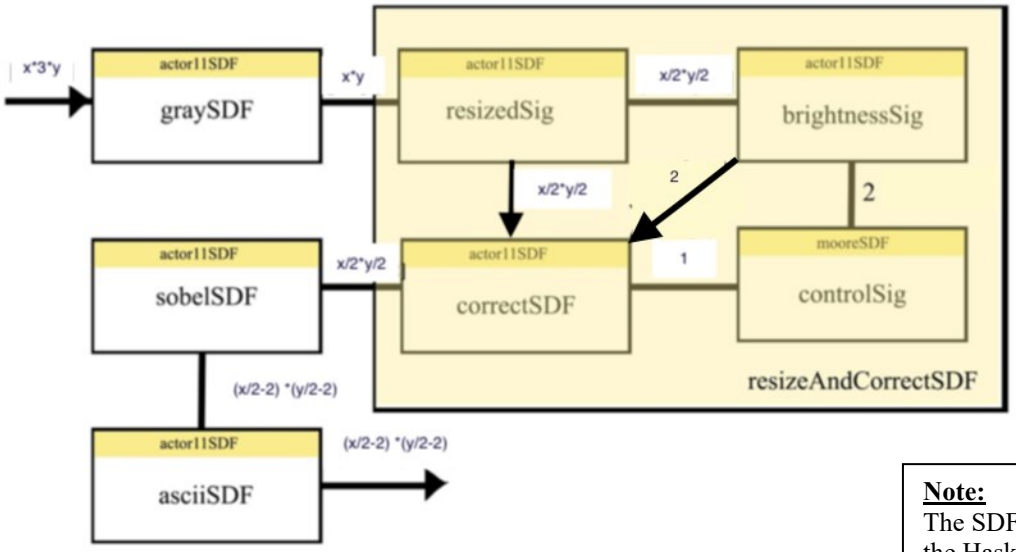


Fig A: SDF actors diagram

Note:
The SDF is intended to depict the Haskell code as is, which may possibly abstract from some internal details.

In figure B, tokens from nextstate are cloned to output and nextstate.

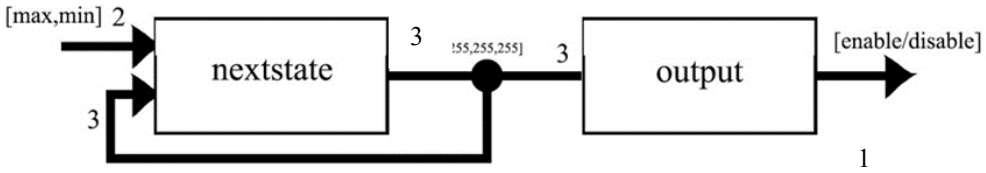


Fig B: ControlSig actor (mooreSDF)

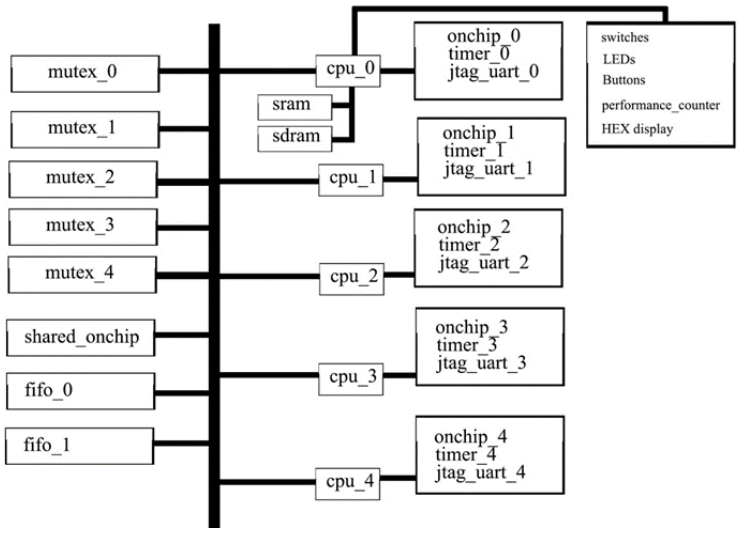


Fig C: Hardware interconnect overview

Stencil

A stencil pattern is a form of the map pattern where the output depends not only on the corresponding input value but also on the neighbors of the input at a particular offset. Here, the stencil pattern is utilized in the actual Sobel filter operation where the required gradients x and y are computed using convolution where a result at each position is determined by the input at that position as well as it's neighbors. In this case, only the neighbors at an offset of 1 are considered, and their weights are determined by the Sobel matrix.

G is the Sobel operator discussed above.

$$\mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A}$$

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$$

2. Altera Interconnection hardware layout:

The hardware architecture of the given de2_nios2_mpsoc.qsf has the given structure as shown above. The hardware has five shared mutex (mutex0 to mutex4) which can be used by any of the five CPU cores. There are five CPU cores (cpu_0 to cpu_4) connected common Avalon switch fabric. The Avalon switch fabric has DATA_IN width of 32 bits and DATA_OUT width of 32 bits (maximum data transfer widths).It is through this fabric all the data communication passes through.

Each CPU core has its own hardware timer (timer_i), on-chip memory and jtag units explicitly for itself as shown above. cpu_0 controls read and write to SRAM and SDRAM memory area and no other CPU cores can do read/write to theses locations. The shared_onchip can be accessed for read/write by all the CPU cores. There is also two shared hardware FIFO units FIFO_0 and FIFO_1 which can be used by any of the five CPU cores for message passing. The I/O interfacing components like switches, LEDs, Buttons, performance_counter, HEX display etc are controlled by cpu_0.

The Altera hardware interconnect overview is in Figure C.

IMPLEMENTATION

1. Performance Optimizations

Merging SDF Actors:

In an effort to reduce the total overall computation, the execution of the grey SDF actor is merged with the resized signal and brightness signal actors of the resize and correct SDF. Here resize happens first, followed by grayscale and brightness calculations. This enables a four-fold reduction in the total number of greyscale operations required.

Exploiting complete width of data bus:

Memory reads and writes are the bottlenecks of the system. In-order to reduce the execution time of these operations, the width of the data bus is exploited. Since the data bus is 32 bits or 4 bytes wide, data transfer is also performed 4 bytes at a time instead of in a byte-wise manner. The data bus is used at its maximum capacity, thus guaranteeing enhanced performance.

Shifting in place of floating point operations:

Conversion of an RGB pixel to a grayscale one involves floating point multiplications or fractional multiplications (with a division operation). Floating point multiplications have a negative impact on memory footprint as a larger library is required. Fractional multiplication which involve divisions negatively affect performance as they take much longer to execute. These issues are bypassed by employing bit shifts to achieve the same results.

Loop Optimizations:

Most of the program execution time is spent in loops. Due to this, loop optimizations significantly enhance performance. Here, multiple SDF actors and other operations are rearranged in a way that would guarantee the lowest possible number of loops to perform the intended functionality, thus requiring fewer iteration through the entire input. Furthermore, a few other optimizations are performed as well, such as approximation of the square root function from our initial implementation to require no looping. The nested loops performing convolution is also unrolled to improve performance.

The square root approximation was computed against 30 values and exhibited a deviation of +/- 11. The deviation is compensated by the improvement in performance. Having tested with multiple images, it was determined that the edges were detected with good accuracy. This bold approximation is found to be feasible because the output signal levels are just 16 values.

Predominant usage of shared memory over FIFO:

Hardware FIFO's serve as a safe and easy to use way for inter-core communication. However, they perform poorly with regards to execution time. Hence, in this project, the faster shared memory is used to communicate larger blocks of information between processors, while FIFO's are limited

to smaller pieces of information which need to have guaranteed safety to ensure proper execution, which include operations such as event signaling.

2. RTOS

The single core bare metal implementation is enhanced with the usage of a RTOS. Multiple actors are realized as separate tasks and synchronization is achieved with the use of semaphores.

3. Multi-core bare metal

Event Based Scheduling

In order to perform operations in a synchronized manner, this project utilizes a rudimentary event based scheduling approach where events are notified to corresponding cores using the available hardware fifo. For instance, CPU 2 informs CPU 0 that the image to be processed is read by updating the fifo 0 with a predetermined bit pattern which CPU 0 can read to be informed of this event. Similarly, cores 1 and 2 each update the same fifo (with different bit patterns) once they are done processing the image and writing their part into shared memory in order to intimate CPU 0 of these events as well. Hardware mutexes are used to ensure synchronized memory access.

Memory Footprint Optimizations:

In order to reduce the memory footprint, this project does not include large libraries such as the floating point library, the math library for square root and power calculations. Moreover, a lightweight version of the stdio library is used.

RESULTS AND DISCUSSION

The values of Throughput and Size of all implementations are in Table A.

Multicore – baremetal :

We have used an event based scheduling with the help of hardware FIFO to communicate b/w different CPU cores. The cpu0 is the master core that'll read and write the input and output images. cpu_1 and cpu_2 are slaves that'll do continuous processing of the input images and write back the results.

The execution details are denoted in the schedule in Figure D.

cpu0 will initially write the RGB image into shared memory location and waits on FIFO0 in a blocking manner to receive commands from cpu1 or cpu2 (if they finish processing).

The RGB image written over the shared memory will be read (R)out by the cpu2 for the first 3.3ms and continues its processing (complete operations till Sobel output, in next 5.5ms).
During this time the cpu1 will read the image (pipelined as data bus is free during this time for image read) and continue its own processing in parallel. After processing cpu2 will write back the Sobel image into another shared memory location(W) and informs the waiting cpu0 through FIFO. On receiving the signal cpu0 will read (R)the Sobel output and display the ascii image or write the output into SRAM. Also cpu0 will rewrite the input image into shared memory(W). cpu1 will do the same functionality as cpu2.

In all the operations the shared memory access is protected by mutex0 for synchronisation.

The Sobel edge detection is done correctly for 200 images/second requirement with total code footprint less than 45 kB.

Another approach :

The application can be implemented more efficiently if we would have done a task wise migration onto different cores. Like image resizing done by one core and then rest of the operations by different cores (dedicated functionalities for individual cores.).

When one of the core finishes its functionality it can process the incoming next image immediately and thus increasing the throughput.

Issues and solutions:

One of the major issue was the image corruption due to lack of synchronisation, which was solved by protecting all of the shared memory read/write with mutex0.
Also synchronisation/timing among different cores was another issue which was solved using the FIFO message passing to communicate b/w cores for synchronisation.

The mapping of SDF Actors to different cores are as in Figure E.

Optimal number of cores:

The shared memory read/write is the main bottle neck of the application that prevent us from utilising the complete parallel processing capability of independent cores. When one of the core uses the shared memory others have to wait for it to finish in order to access memory (since it is a shared resource). We have not utilised all extra 5-cores but instead only used 3 cores (cpu_0, cpu_1 and cpu_2). This is because the pipeline will be efficient with just three cores and increasing/utilising other cores will take-up the data bus of the Avalon fabric and this in-turn will reduce the overall performance (as other will be blocked for memory operations.)

	Single Core (RTOS)	Single Core (Bare metal)	Multi Core (Bare metal)
Throughput	332ms/image	29ms/image	217 images/ sec
SRAM [bytes]	135584 Bytes	13988 Bytes	19700 Bytes

Table A : Throughput and memory footprint for all implementations

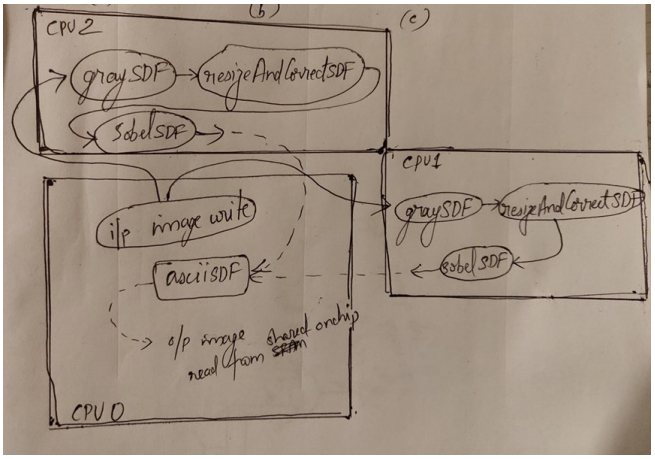


Figure E : Mapping of SDF Actors across cores

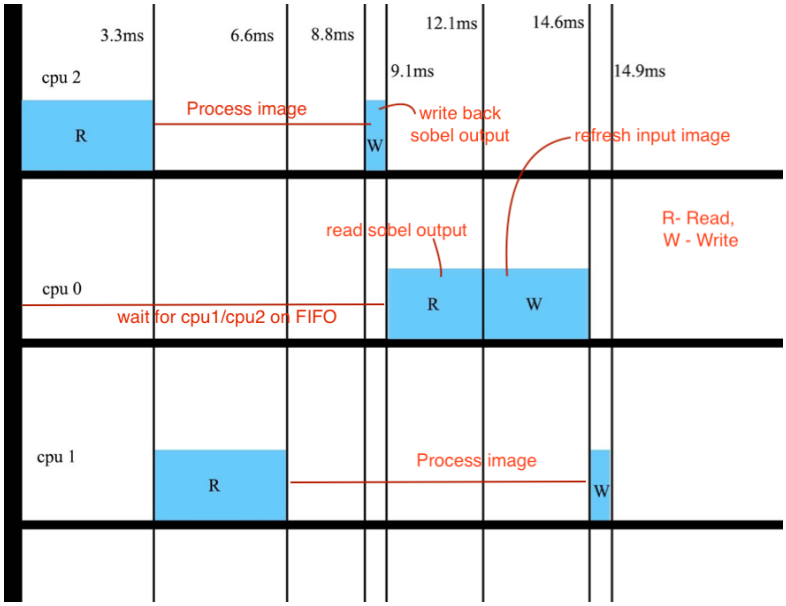


Figure D: Schedule for CPU 0 , CPU 1, and CPU 2

REFERENCES

[1] Edward A. Lee and Sanjit A. Seshia, *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*, Second Edition, MIT Press, ISBN 978-0-262-53381-2, 2017.

[2] Introduction to pipelining and multiprocessing, http://www.cse.ust.hk/~cktang/cs180/notes/topic09.pdf

[3] McCool, M., Robison, A., & Reinders, J. (2012). Structured parallel programming patterns for efficient computation. Amsterdam ; Boston, Mass.: Elsevier/Morgan Kaufmann.

[4] Altera Nios II documentation page, https://www.intel.com/content/www/us/en/programmable/products/processors/support.html

[5] Embedded Peripherals IP User Guide, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_embedded_ip.pdf