Design Space Exploration for Multi-Core JPEG Decoding

Mohit Agnihotri
Email: m.k.agnihotri@student.tue.nl

Nayan Singh Ravindra
Email: n.singh.ravindra@student.tue.nl

Pavan Kumar Shankara
Email: p.k.shankara@student.tue.nl

Cao Xu
Email: x.cao.1@student.tue.nl

Rajibul Alam
Email: r.alam@student.tue.nl

Abstract*— *This report is a technical account of the porting, optimization and design space exploration using multiple architectures for implementing JPEG decoder on a multi-core Micro-blaze Platform. The work is a practical assignment for the course Embedded Systems laboratory (5KK03) at Eindhoven University of Technology. Literature survey was done to get acquainted with JPEG decoder and various design space and optimization opportunities were identified, implemented and compared to conclude on possible trade-off points. Design space explored included two different software architectures. First focused on utilizing partial data independent nature of JPEG decoder by splitting data processing over available cores while second focused on utilizing discrete step based architecture of JPEG by splitting each step onto available core. Performance of data transfer through Network-on-Chip (NoC) using DMA and C-heap was analysed and benchmarked using multiple set of experiments with above identified architectures. Finally applications were extended on COMIK micro-kernel architecture, allowing integrating multiple real-time applications. Design space for COMIK included combination of fractal, Data-parallel and functional-parallel implementation executing in parallel.

Keywords ***JPEG, Data-parallelism, Functional-parallelism, Micro-blaze, DMA, C-Heap, COMIK, Circular-Buffer.***

June 11, 2014

I. Introduction

JPEG [1] [2] is discrete cosine transform (DCT) based still image compression standard. Wide acceptance of coding system has motivated study of optimization schemes as it allows efficient transfer and storage of images. Report describes implementation of decoding algorithm on a multi-core architecture realized using Xilinx FPGA board. The platform consists of three Micro-blaze processors interconnected by network on chip(NoC) to provide inter-core communication.

The paper is structured as follows. Single core Memory Mapped Input Output (MMIO) implementation of JPEG decoder application is described in II while III describes the architecture, design considerations and optimizations done for a single core implementation using Direct Memory Access (DMA). Data parallel architecture is discussed in IV highlighting decision points, trade-off considered with mention of COMIK version. Functional parallel architecture's implementation and corresponding COMIK design is discussed in V. Details of benchmarking environment and strategy can be referred in VII and VIII concludes the report.

II. Single Core MMIO Design space

First step involved porting generic JPEG decoder application onto hardware platform with given limitations. It required data structure relocation to local memory and images data could be access via DRAM, also referred as shared memory. Library functions were required to be replaced with custom code implemented within hardware limitation as limited support is available from standard libraries. In the embedded cores functions such as malloc, free, fprintf, fopen, fclose, exit etcetera are not available.

File operation require special mention as platform libraries provided no support and during porting code is required to be adapted to access data from shared memory directly. Ported application access image data from DDR as flat stream reading single or multiple byte based upon the operation mode. In order to optimize and minimize communication overhead, data was read as integer instead character.

Once the image is decoded, output bitmap image is required to be written onto DDR. This part of the code was modified as underlying hardware platform imposed specific limitation on the format and location (frame-buffer), dictating how and where finally output could be written. The frame buffer has a resolution of 1024x768 which has to be written in ARGB format. The 8 most significant bits are the alpha channel which is ignored, followed by 8 bits of red channel, 8 bits of green channel, and lastly 8 bits of blue channel. After the color conversion process, data in the color buffer is moved to the output frame-buffer so that the final image can be completed.

III. Single-Core DMA Design space

1) Implementation of input circular buffer: Underlying hardware design allows to loading input jpeg files into shared memory (DDR). Given the limitation of exclusive availability of DMA or MMIO mode, image must be transferred into local memory using DMA before any operation can proceed in DMA mode. The maximum burst size of single DMA command is constrained by size of either source or destination memory locations and requires all operation i.e. write and read operations should be 32 bits wide. In current platform maximum burst size is constrained by the size of communication memory size (cmemout: 0x1000 bytes). There are design level trade-offs while considering optimal values for DMA burst size(reading from DDR), memory usage and execution time. In our implementation decision was taken to optimize communication memory footprint for minimum value and thereby the input DMA burst size(data transferred in single DMA call) was set to 128x4 bytes using a tunable parameter.

The input circular buffer is located in communication memory (cmemout) of a tile. It has four blocks with each of size 128x4 bytes which are consumed in circular fashion leading to minimum DMA delay. Initially all the four blocks of the input circular buffer are loaded with data from the DDR. Buffers are tracked for consumption and reloaded from DDR once fully consumed. The condition to refill a consumed block of input circular buffer is checked after each byte is consumed from the input circular buffer. The block size of 128 words was chosen to strike balance between communication memory usage and execution time spent. The design of circular buffer enables the application to decode input jpeg images with size greater than 4KB and also to reduce the memory footprint.

Circular buffer implementation enabled jpeg decoder to decode images with sizes larger than 4KB but it have associated overhead of book keeping. Images with the application markers and comment section which require to jump to an offset in the bits stream suffers from greater overhead as the stream in input circular buffer has to be appropriately handled along with the book keeping variables as depicted in figure 1. The problem faced during this implementation was to ensure that the source of DMA read commands are always at word boundaries (alignment). This aspect of aligning DMA to word boundaries added extra code to appropriately set the bit stream book keeping variables both at DDR (producer) and input circular buffer at communication memory (consumer). After resetting the stream to appropriate location in DDR, the next burst of DMA receive command loads the appropriate block of input data from DDR to the next block of input circular buffer.

2) Implementation of output frame buffer: Decision to keep circular and output display buffer on different communication memories (cmem out) was influenced by consideration for COMIK architecture. This decision helped during COMIK implementation by allowing efficient communication memory management between applications. Output write to display frame buffer is handled in the MCU loop, thereby reducing the communication memory usage for output write (equal to one MCU size, i.e. (8 X 8) etc.). The redundant copy of frame buffer in local to communication memory was removed and after each MCU is decoded, the output pixels were written to communication memory directly in ARGB format, later to be get pushed out by DMA at appropriate locations in display frame buffer. This posed few problems for MCU at edges that have partial sizes. This was solved by restricting DMA burst to column size and choosing not to write the extra rows of MCU than required for MCUs at the edges. Since display frame buffer is of size 4MB, the maximum resolution of output image is HD (720p). For monochrome images each pixel value is copied into RGB bytes, with transparency character (A) as zero. For partial MCUs at the edges, DMA burst was chosen to restrict the display write to actual column size; thereby writing only required rows to the display frame buffer.

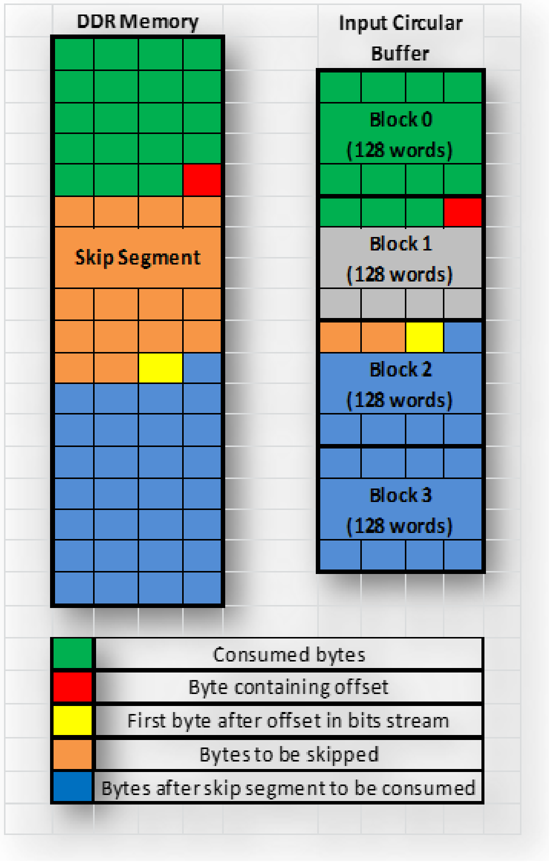


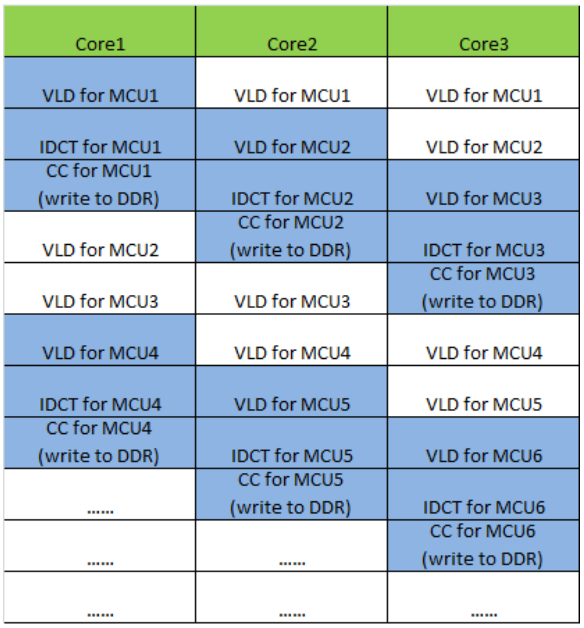
Fig. 1. Input Circular buffer Design for JPEG decoder

IV. Multi-Core Data Parallel Design Space

The DC coefficient of each MCU in JPEG decoder application is coded using difference encoding and the AC coefficients are first run length encoded followed by Huffman coding. Due to such coding the Variable Length Coding is serial and hence cannot be parallelized by executing on multiple cores. Zig-Zag conversion, IDCT and Color Conversion do not have dependencies and hence these functionalities can execute in parallel. The team implemented two data parallel architectures based on restart markers.

Architecture: JPEG Without Restart Markers: In this architecture, each core performs VLD for each of the MCU. IDCT and Color Conversion on each core are executed for MCUs with offset of 3 as shown in figure 2. For example, core 1 executes MCUs 1, 4, 7...., 3n+1, core 2 executes MCUs 2, 5, 8...., 3n+2, and core 3 executes MCUs 3, 6, 9,...., 3n+3. Disadvantage of this approach is the rework performed by each core which is also processed by other cores which leads to less energy efficient architecture. Major advantage of this approach is that unlike Functional parallel implementation, operation of each core is independent of each other and saves time of initiating other cores and data transfer between cores. Another noticeable feature is that all 3 cores have approximately equal distribution of work.

The data parallel clocks in at 2.22 seconds for execution time for the reference.jpg image from benchmarking dataset.



Core1	Core2	Core3
VLD for MCU1	VLD for MCU1	VLD for MCU1
IDCT for MCU1	VLD for MCU2	VLD for MCU2
CC for MCU1 (write to DDR)	IDCT for MCU2	VLD for MCU3
VLD for MCU2	CC for MCU2 (write to DDR)	IDCT for MCU3
VLD for MCU3	VLD for MCU3	CC for MCU3 (write to DDR)
VLD for MCU4	VLD for MCU4	VLD for MCU4
IDCT for MCU4	VLD for MCU5	VLD for MCU5
CC for MCU4 (write to DDR)	IDCT for MCU5	VLD for MCU6
......	CC for MCU5 (write to DDR)	IDCT for MCU6
......	......	CC for MCU6 (write to DDR)
......	......	......

Fig. 2. Data Split of JPEG Decoder over 3-Processor tiles

Hence this architecture provided 43.19% improvement over the single core DMA implementation.

Architecture: JPEG With Restart Markers: From Amdahl's law, the speed-up of a program using parallel architecture is limited by the execution time for sequential fraction of the program. For an image comprising restart markers, the DC components of MCUs after a restart interval are independent of DC components of the preceding MCUs. Image is broken into restart intervals and each core works on restart intervals with offsets of 3 with similar logic for images without restart markers. In JPEG, a restart interval is defined with the FFDD marker as a 2-byte number and the restart marker values will increment from FFD0 to FFD7 and then start again at FFD0.

In our architecture, Core 1 processes the restart intervals marked by first interval and the intervals marked by FFD2,FFD5 and FFD7; Core 2 processes the restart intervals marked by FFD0,FFD3 and FFD6; and Core 3 processes the restart intervals marked by FFD1 and FFD4. The decision to hardcode the restart marker was taken to eliminate the need of communication between core. Hardcoding of restart markers eliminates the interdependency between the cores and reduces the processing overhead for each cores. Additionally hardcoding of markers facilitated in debugging of the code as it is allowed to pinpoint the restart interval which is not processed effectively, leading to the core which is causing the problem. Assuming a JPEG with restart interval after every 8 MCUs, figure 3 illustrates the distribution of work between Core1, Core2 and Core3. It is important to note that Core3 processes only 2 restart intervals for every 8 intervals as opposed to Core1 and Core2. This decision was based on the observation that Core3 has to match markers for more number of MCUs during which Core1 and Core2 would already be processing other MCUs. Another noticeable feature is that, unlike data parallel implementation without restart markers, each core performs VLD for only the MCUs in the respective MCUs leading to much power efficient architecture.

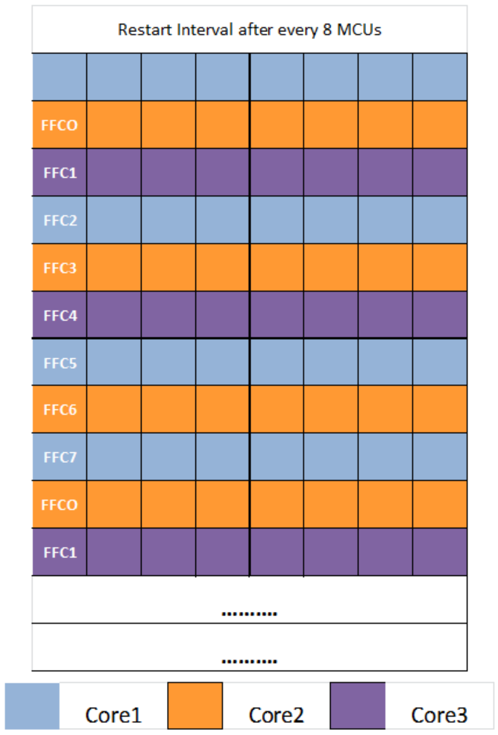


Fig. 3. Data Split of JPEG Decoder with Restart Marker over 3-Processor tiles

The Data parallel architecture was implemented to optimally utilize both the DMA units available to each core. After processing each MCU in the restart interval, the processed output from the MCU was split in two parts in the Y axis of the image and the respective data was written to DDR using two DMA. This reduces number of DMA cycles necessary for writing one MCU by half by distributing the load between the DMAs. This architecture provided 31% improvement over the normal data split using a JPEG image with restart makers.

Data Parallel with Fractal using COMIK: COMIK microkernel creates composable virtual processors that are dedicated application resources. Using COMIK it is possible to execute multiple applications in TDM slots which make the architecture composable. The team exploited this composability by running an instance of Data parallel JPEG on COMIK microkernel platform. All 3 cores run an instance of COMIK which schedules the code in TDMA slots.

During the implementation of this architecture, the VGA Frame Grabber functionality was enabled in the makefile and the feed from the server was used to analyse the runtime processing of the images. This architecture was experimented with different combinations of COMIK Slot durations. The processing was observed using the VGA Grabber. Due to overhead involved in context switch of application, the execution time increases. The execution time increased from 2.22 sec to 2.8 sec due to switching overhead. This overhead can be justified. Once COMIK architecture was in place, fractal application was made to run in along with Data parallel code on all 3 cores. It was observed that fractal implementation consumes more execution time compared to the JPEG images. This observation allowed make the decision to hard-coded the placement of mk_mon_debug_tile_finished() at the end of the fractal implementation. Each core processes 1/3rd of the JPEG image and 1/3rd of the Fractal, leading to equal distribution of work. This combination requires 4.86 seconds to execute.

JPEG TWINs - Two time multiplexed Data parallel implementation using CoMik: Data split architecture was further extended to decode two jpeg images in parallel using CoMik virtual processors. In order to achieve, frame-buffer was divided into two section each of 512x768 pixels while multiple JPEGs were uploaded and accessed using pre-defined infrastructure provided by platform. Each virtual processor has a pre-defined static region on memory to facilitate compartmentalization of applications leading to zero contention while access DDR.

One significant design consideration during the implementation of this architecture was that each data parallel implementation of each core should use only one out of two DMAs available for that core i.e. one instance data parallel code uses DMA0 on all cores and the other instance uses DMA1 on all cores. This decision allowed placing data parallel implementations in two COMIK slots, whereby each slot uses a particular DMA and the corresponding DMA is responsible for transferring all the processed pixel data for one particular image. This allowed to spatially segregate the transfer of data and facilitated during the debugging process. Since the two data parallel versions are completely decoupled from each other, it is possible to execute two images with different image sizes, different MCU dimensions and images with and without restart markers in parallel. The architecture was tested and verified with one image of 8 x 8 MCU size and another image of 16 x 16 MCU size running in time interleaved fashion, justifying the complete decoupling. While this architecture can be extended to implement picture In picture functionality for two images, due to time limitation this architecture consideration was not pursued further.

V. Multi-Core Functional Parallel Design Space

JPEG is computationally demanding consisting of Huffman decompression, de-quantization, IDCT, image sampling and colour conversion stages. Among all them, Huffman decompression is strictly sequential due to the variable length of code words and context where new code words starts is known to last decoded code-word, forming majority of sequential part of the decoding process.

Exploring alternate design option with aim to optimize execution time, memory usage and proportionate load balancing, functionalities of JPEG decoder were split into 3 task units. Each unit was chosen such that they could perform desired function independently with minimum inter-core communication requirement. Core 1 was assigned to perform Huffman decompression followed by de-quantization producing frequency block. It is then transferred to core 2 which performs IDCT transformation resulting in pixel block which is later transmitted to core 3 as described in figure 4. Finally it is color converted and written to DDR.

One of the biggest problems encountered was synchronization of data among the tiles as each of them waited in infinite loop. It was not possible without explicit communication among cores availability of new data. With this information, communication protocol as illustrated in figure 5 was established.

Protocol performed reasonably but erratic behaviors were observed. Analysis suggested communication protocol suffered from producer-consumer problem. Time difference between the consumption and production was not accounted in the protocol leading to events where data at tile was over-written by incoming data. Adding extra buffers at consumer's end is a probable solution but not full-proof as it fails to circumvent problem when producer is faster by factor greater than buffer size. Results and experiments indicated necessity to have full-duplex communication to avoid inter-core issues.

C-Heap: C-Heap communication protocol provides an efficient FIFO based method to transfer data across as described in [3]. It is designed to safely transfer data between cores by providing synchronization, which is rigorously tested and proven to work. It can use MMIO or DMAs for doing the actual transfer. In C-heap ideology, there is a producer and a consumer who communicates via a buffer. (FIFO can be mapped to any memory) and unit of communication is called a token. Token is user defined and can be an integer, a video frame, etc. Producer and consumer have FIFO administration information i.e. variables tracking location of FIFO, available data, space, etc. and double administration avoids reading in remote memory. DMA can be seen as a dumb hardware unit that can only move data around and C-Heap takes care of synchronization, tracks available memory making it more efficient, safer and easier to be used than standalone DMA.

Functional Split used libraries/source code provided to circumvent inter-core synchronization issues. Architecture provides a flexible design and parameters can be tuned for a given hardware constraints enabling best performance at the cost of execution over-head.

FIFO Buffer Size: It is used to determine maximum number of in-flight communication units between producer and consumer. Higher numbers allow greater degree of pipeline at the cost of inter-core communication memory. Thus a trade is required to derive optimum value for the given platform. Experiment performed with 9 different images with varying FIFO size between 2-22 symmetrically i.e. size of FIFO buffer between core 1-core 2 and core 2-core 3 is always same, revealed 6 as the optimum value minimizing execution time as reflected in figure 6.

Architecture derived above is limited to single application. Extending the design for multiple applications requires extensive effort in porting new application and efficient resource management as embedded systems are highly constrained especially in regards to hardware resources. Comik micro-kernel is an elegant method which allows programmer to focus on single application by providing sand-box environment while efficiently handling under-lying resources in user defined TDM schedule each serving a virtual processor [4]. Design when extend with CoMik kernel presented degradation in execution time when compared with bare-bones version. Experimentation shows maximum cost of 0.103 sec with single application i.e. JPEG decoder. Figure 7 presents details.

Experiments were carried out to determine extent of degradation observed when JPEG decoder was run simultaneously with Fractal application. Three different tests where one, two

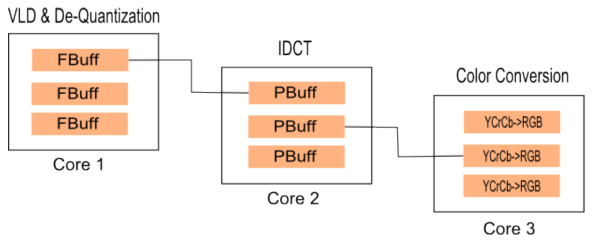


Fig. 4. Functional Split of JPEG Decoder over 3-Processor tiles

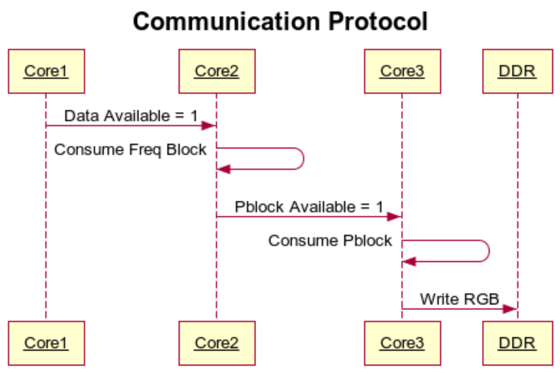


Fig. 5. Inter-Core Communication protocol

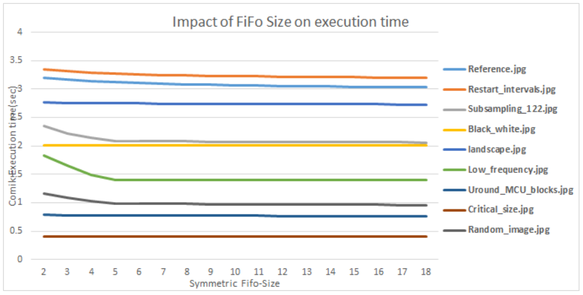


Fig. 6. Impact of Fifo buffer on decoding time.

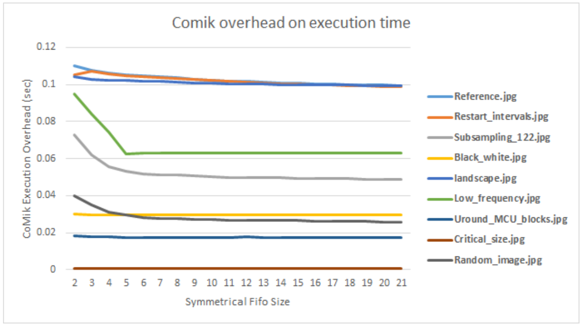


Fig. 7. Impact of CoMik Micro-kernel on decoding time.

and three instance of fractal were run in parallel to decoder. Figure 8 suggest minimal overhead.

Experiments also presented few functional limitations of CoMik kernel which makes integration of multiple applications on CoMik virtual processor tedious. First limitation arises from the fact that developer must be aware of relative timings of the applications to introduce marker *mk_mon_debug_tile_finished* to mark end of processing. Marker is required to signal end of processing and should be indicated by application which is last to finish as COMIK waits for single marker. Once marker is detected, COMIK starts reading TFT buffer and dumps final output. This can be a tedious task as determining exact running time of each application before-hand and modifying code every time estimate changes. It is proposed to extend functionality by allowing waiting for reception of end makers from all application to conclude mark end of processing.

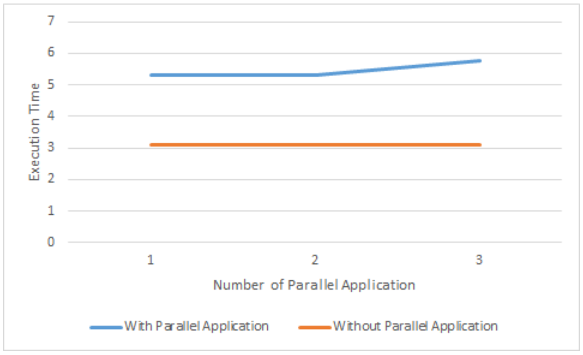


Fig. 8. Multiple Application overhead on Execution time

Second limitation arises from the fact that current CoMik uses monolithic compilations for each processor tiles. When multiple applications attached to different virtual processor are compiled, linking issues maybe experienced due to colliding names for global variables across application.

VI. JPEG DUO - Time multiplexed Data parallel and Functional parallel Implementation using CoMik

The CoMik architecture allows creating composable virtual processors on the existing processor cores allowing for execution of two completely decoupled applications on each processor. The team decided to take advantage of this option by implementing architecture in which one data and function parallel implementation execute in time interleaved fashion on all 3 cores, demonstrating and validating the complete composability feature of CoMik.

Design Considerations: Current architecture is designed considering hardware limitations on the Xilinx ML-605 FPGA board. Platform has two independent DMA per core available for inter-core and shared memory communication. Design necessitated to ensure data and function parallel implementations use different DMA channels on each core to guarantee no overlap of resources during context switching. It was decided to use DMA0 for data and DMA1 for function parallel implementation on all cores.

Communication memory is other resource which has to be shared. Each core has two cmemout buffers of size *0x1000 Bytes*. This necessitates that two implementations use different cmemout for data transfer. It is also important to note that both the implementations use input circular buffer functionality explained in previous section to read the image data from the DDR. Hence during the design essential steps were taken to handle the circular buffer functionality and output DMA transfer for each core. This included reducing the size of input

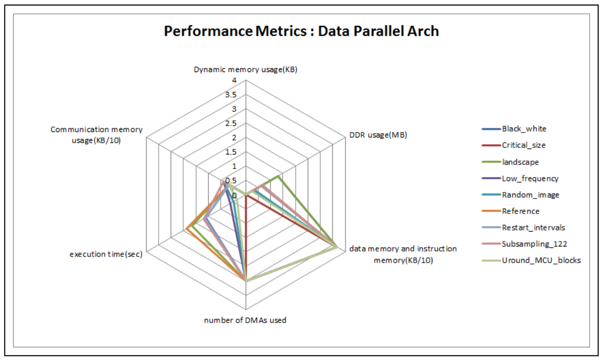


Fig. 9. Data Parallel Implementation Spider diagram

circular buffer blocks to 2 and number of initial buffer loads to 2.

Function parallel implementation uses C-Heap functionality to communicate image data between the cores with dedicated DMA, cmemout and cmemin resources whereas in data parallel implementation there is no inter-core communicate except the flags used to indicate the completion of processing. This helped to ensure guaranteed non-overlapping usage of cmemin's for each core.

Implementation: During the implementation it was observed that after one context switch from data to functional parallel implementation and back (or vice versa), output image was uncorrelated with original jpeg image. By enabling VGA Grabber functionality during debugging, presence of shared global variables between two implementations were discovered. CoMik architecture enforces applications to have unique name for global variables. Inability to ensure this condition may lead to undesirable results. Upon fixing this issue, the architecture was completely functional whereby it allowed running two images of different dimensions and MCU sizes on data and function parallel architectures.

Execution time of function parallel implementation is higher than the data parallel implementation for images with similar dimensions and MCU sizes, it was decided to hardcode the placement of end of processing marker i.e. mk_mon_debug_tile_finished() at the end of function parallel implementation. This architecture was further extended to include picture in picture functionality in which the image processed in the function parallel implementation is encloses by data parallel image.

VII. Benchmarking Strategy and Results

In each of the following graphs, we compare the performance of our different versions for each benchmarking image. These images are compared on the basis of 6 parameters decided by the benchmarking comity namely

1) Dynamic Memory usage (in KB)
2) Data and Instruction memory (in KB/10)
3) DDR Usage (MB)
4) Execution Itme (in Seconds)
5) Number of physical DMA channels used (#)
6) Communication memory usage (in KB/10)

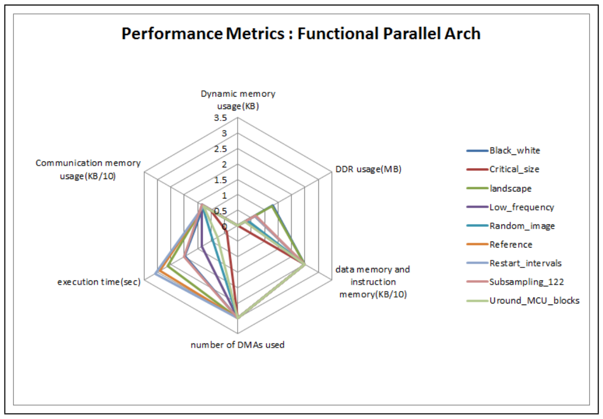


Fig. 10. Function Parallel Implementation Spider diagram

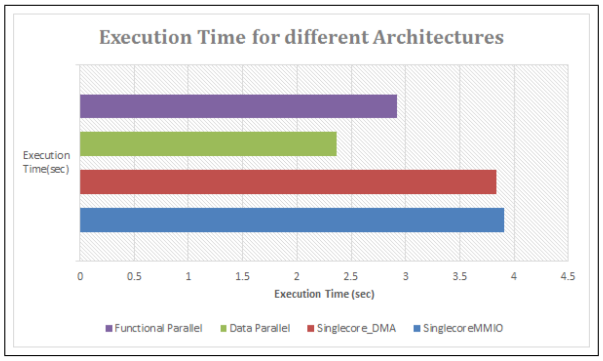


Fig. 11. Execution Time for Different Architecture

Figure 9 depicts the performance of data parallel architecture with respect to the benchmarking parameters for all the reference images. The amount of communication memory and the execution time of the JPEG decoder are largely influenced by the size of the image as seen from the graph. On the other hand the number of DMA channels used is irrespective of the size and type of the image. As it can be observed from the Spider graph in figure 9, execution time of Data parallel execution is the least of all the architectures considered so far. This is due to the equal and effective distribution of the computation among all 3 cores resulting in effective utilization of hardware resources and hence better throughput and lower execution time. Similiar results for Functional parallel architecture can be explored in figure 10.

VIII. Conclusion

In this course work project various designs performing JPEG decoding operation were explored and implemented on Micro-blaze platform. An effort was made to explore most of the interesting designs in the design space of multi-core JPEG decoder implementation on this platform. Various trade-offs and analysis was carried out the chosen design to achieve optimal balance between various parameters to name few of them total execution time, usage of communication memory, usage of DMA resources, data memory (both static and dynamic memory). It was decided to reduce the communication memory footprint in all our implementations, this aspect assisted us to effortlessly port one or more applications on COMIK micro kernel and execute them in time multiplexed form. Even though there are eight partitions per MB core tile which can ideally support different applications, the main constraint in DMA mode was the number of DMA channels available per MB core tile.

Experiments show data parallel architecture is best among the various explored. Benefits of this architecture are even more prominent when image contain reasonable amount of restart markers. This observation can be attributed to equal distribution of the work among the core which is not in other architecture. In functional architecture core performing VLD & Zig-Zag transform takes approximately 50% of processing and additional cost is introduced for inter-core communication. MMIO based architecture is outperformed by DMA based as CPU is stalled for the communication. Figure 11 gives an evolution of explored architectures on execution time.

Theoretically, in MMIO mode of operation eight different applications can be executed in parallel in eight different partitions per MB core tile. The major limitation is posed by the communication memory usage in each MB core tile. This aspect was stressed upon while implementing and designing the JPEG decoder and Fractal application on the platform.

Acknowledgement

The authors would like to thank Professor Kees Goossens, Andrew Nelson, Valeriu Codreanu, Martijn Koedam and Rachana Arun Kumar for their guidance and feedback throughout work.

References

[1] International Organization for Standardization, *ISO/IEC 10918-1:1994: Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines*. Geneva, Switzerland: International Organization for Standardization. [Online]. Available: http://www.iso.ch/cate/d18902.html

[2] G. K. Wallace, "The jpeg still picture compression standard," *Commun. ACM*, vol. 34, no. 4, pp. 30–44, Apr. 1991. [Online]. Available: http://doi.acm.org/10.1145/103085.103089

[3] A. Nieuwland, J. Kang, O. Gangwal, R. Sethuraman, N. Bus, K. Goossens, R. Peset Llopis, and P. Lippens, "C-heap: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems," *Design Automation for Embedded Systems*, vol. 7, no. 3, pp. 233–270, 2002. [Online]. Available: http://dx.doi.org/10.1023/A%3A1019782306621

[4] A. Nelson, A. Nejad, A. Molnos, M. Koedam, and K. Goossens, "Comik: A predictable and cycle-accurately composable real-time microkernel," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, pp. 1–4.