Optimization of JPEG Decoder on CompSoC Platform

Weibo Qiu - 1033389, Apurv Ashish - 1036883, Mykola Zaitsev - 1034560

I. Introduction

A. JPEG decoder structure

The JPEG decoder takes a jpeg image (compressed data) and reconstructs a bitmap (uncompressed data) using a baseline decoding process. In general, JPEG algorithm is inherently sequential. Although, along with the sequential part, especially in decoding part of the image, there are CC and Raster function which can easily be parallelized. They present the parallel part of the JPEG algorithm and therefore can be more efficiently mapped on multi-core environment than the sequential part.

B. CompSoC platform and communication

Next, we shortly introduce the CompSoC platform on which the JPEG decoder is to be ported. In terms of hardware CompSoC platform consists of Daelite Network-On-Chip (NoC), four microBlaze tiles with local memories and DMAs and global shared DRAM memory. There are two means of communication in CompSoC: Memory Mapped I/O (MMIO) and Direct Memory Access (DMA). MMIO allows the CPU to control hardware by reading and writing specific memory addresses whereas DMA allows hardware to directly read and write memory without involving the CPU. The comparison between the two in terms of bandwidth for writing to and reading from DRAM by core 1 are shown in figures 1 and 2 respectively.

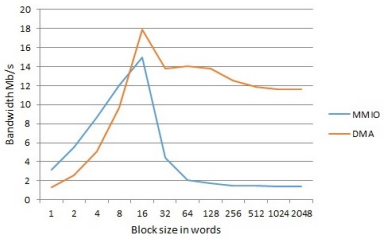

Fig. 1: Core 1 writing to DRAM memory

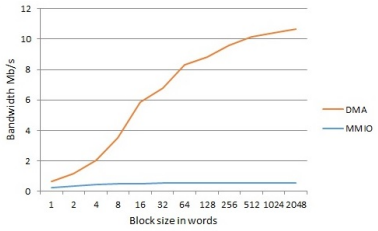

Fig. 2: Core 1 reading from DRAM

We can see that MMIO is faster only when transferring relatively small data blocks up to 16 words. For larger sizes DMA displays a much higher bandwidth. The reason for it is a relatively large execution overhead of DMA when transferring small data blocks. Thus we can conclude that MMIO should be typically used for low-bandwidth operations such as changing control bits and DMA for high-bandwidth operations which can take an advantage of its burst transfer behavior. From implementation view point, MMIO is completely transparent for a designer. DMA, on the contrary, requires extra adjustments such as specifying destination, source, block size and persistence type. A downside of DMA is that non-blocking DMA transfers may cause faulty behavior by transferring incorrect data, whereas blocking transfers may cause excessive delays. Moreover, substantial block transfers may cause excessive overhead of redundant data. Non-blocking scheme contributes in processing the JPEG decoder more efficiently compared to blocking mode. However the blocking scheme is safer in practice since it needs to check the status of the previous data and it helps to avoid hazard in DMA communication. In addition to the reliability, after measuring the execution time under these two schemes, we found that the non-blocking DMA is slightly faster than blocking DMA (no more than 2%). In this case, all the versions we did with DMA blocking to trade off efficiency for safety.

C. Functions divisions

Before dividing the whole JPEG decoder, we need to have a general idea on execution time of each function. Figure 3 illustrates execution times of JPEG functions on cores 1/2 and 3/4 with MMIO communication for Alps.jpg picture. We can see that CC function executes equally well on both cores. VLD and Raster functions are faster on cores 3/4 whereas the IDCT function is faster on cores 1/2. Faster execution of VLD and Raster functions on cores 3/4 is due to their higher frequency. On the contrary, faster execution of IDCT on cores 1/2 is explained by presence of accelerators that speed up complex operations this function includes.

Nevertheless, after benchmarking all the the 9 pictures, it is found that single execution time of each function depends on the picture used, except IQZZ and IDCT which take approximately the same amount of time in every case. But the function divisions of Alps.jpg already provides us with general optimization idea to port IQZZ+IDCT on cores 1/2. Furthermore, VLD should be mapped on cores 3/4.

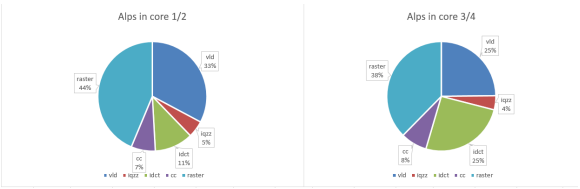

Fig. 3: Two cores function parallel

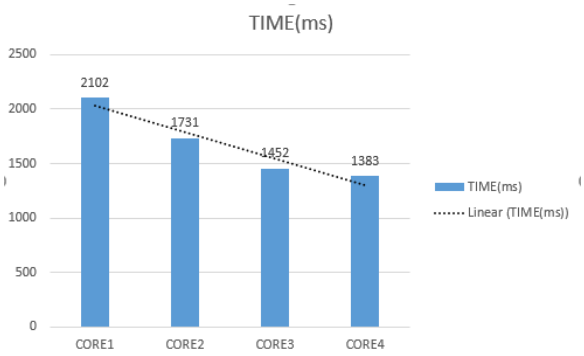

Fig. 4: Three cores function parallel

II. Different Versions of the JPEG decoder

A. Data parallel

For the data parallel version of the JPEG decoder, we experimented with different versions of data parallel codes. Obviously, major part of the JPEG algorithm is inherently sequential, hence, our approach has been largely focused on reducing the execution time by minimizing the redundant workloads on all the 4 cores. The computation power of cores also differs, the same amount of computation, takes 1.2 times more on core 3,4 in comparison to core 1,2. The workload division, if chosen naively, will only lead to an un-even execution time of four cores. Hence, even- banding of 25% workload on each core will result in considerable difference in the execution time of the available four cores. As evident from figure 3 (Effect of Even Banding) above, the core-4 finishes well in advance than core 1&2.

The main reason behind the difference in the execution time of different cores is basically due to two reasons:

- The difference in the computation power of core 1,2 and core 3,4.[minor reason]
- The inherent sequential nature of the unpacking part of JPEG.[major reason]

In JPEG, the decoding part of the image is inherently sequential, hence, it cannot be parallelized.The processing of each MCU requires the component of previously processed MCU, hence, even if the core is processing latter half of the image, still, it must execute the VLD headers from the beginning. This creates an extra overhead on the cores. However, the redundant part can be skipped by using communication between cores to transfer the last processed MCU, but only at the cost of huge communication overhead, as each core(which doesnt start execution from start of image) has to wait for previous core to finish its decoding part. In the parallel version using banding, we implement 3:3:2:2 banding scheme for core 1,2,3,4 respectively. Even with un-even banding, we assume that a same version cannot be suitable for all different type of images. The execution time of the image will also depend on the complexity of the image.

Considering these factors, the modulo-split scheme appears as one of the most desirable data parallel scheme. There are two reasons behind this statement:

- It distributes the entire image evenly on all the cores. Hence, the problem discussed above gets mitigated to a large extent.
- It is more consistent with memory alignment.

But, the choice of modulo scheme also counts. In the Huffman function, each symbol is taken and coded into a variable length code (1-16 bits). If the symbol is frequent then it takes only couple of bits for encoding but random/rarely used symbols take more bits to encode. We assume that the modulo-scheme takes more time to finish execution in such images.

B. Function parallel

The JPEG decoder in our project can be divided into 6 stages as mentioned before. In this section, IQZZ and IDCT are regarded as a whole part, because they both need ten tokens per execution from VLD which makes it hard to separate them in different cores. The benefits of applying function parallel is that each core is able to execute these functions in pipeline so as to decrease single core execution time. In this section, the naive (base-line) version for our experiment is the single core implementation of the whole JPEG process exclude paint part.

We performed 6 function parallel versions in total including 2 versions on two cores (FP1 and FP2) , 2 versions on 3 cores (FP3 and FP4) , 1 best version with DMA communication (FP5) as well as 1 version on four cores (FP6). Based on the execution time of each function we measured before, the execution time of 2 core versions (FP1 and FP2), 3 core versions (FP3 and FP4) will be compared respectively. The memory usage and energy will be further discussed in benchmark section. The energy is obviously dependent on the execution time of each core.

First, we tried two versions on 2 cores. The first version (FP1) is that Init, VLD and IQZZ+IDCT are executed in core 1, while CC and Raster are executed in core 3. The second version (FP2) on two cores improved the FP1 version by assigning Init and VLD in core 3; while IQZZ+IDCT, CC and Raster are executed in core 1. Theoretically, we assume FP2 may have

optimized execution time compared to FP1, because it assigns VLD in core 3 which could be accelerated. However, the Figure 4 indicates a strange result that in all benchmark pictures excluding noise.jpg, FP2 is a bit more time consuming. Only in noise.jpg, FP2 contributes to decrease the total time. Then we measured each function time of the 9 benchmark pictures and found that in noise.jpg the execution time of Init+VLD is close to the remaining functions, while other pictures execution time of IQZZ+IDCT, CC and Raster is almost 2 times more than Init+VLD. In case of FP2, even if Init and VLD would be accelerated in core 3, the remaining functions in core 1 will take much more time to execute. Hence after some time the FIFO buffer will be full and core 3 will be blocked to wait for core 1 to finish. However, the FP2 inspired us to use data parallel in core 1 and core 2 which would be further discussed in hybrid section.

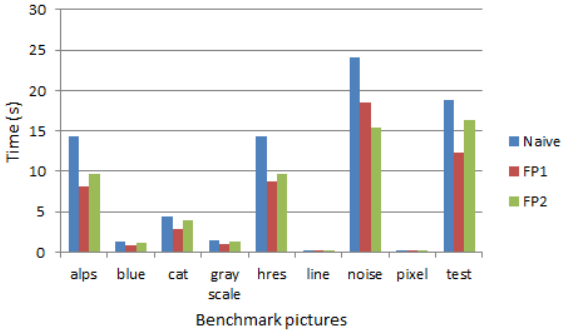

Fig. 5: Two cores function parallel

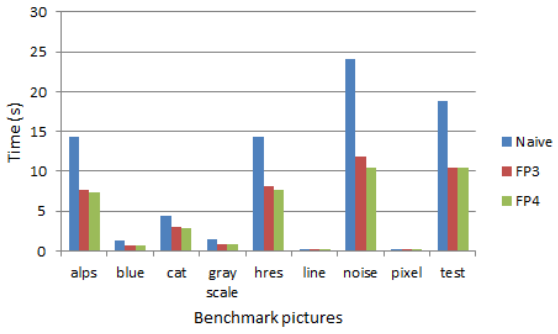

Fig. 6: Three cores function parallel

The third (FP3) and fourth (FP4) versions utilize 3 out of 4 cores so that different functions are assigned to three cores. Specifically, core 3 executes Init and VLD, core 2 executes IQZZ+IDCT, and core 1 executes CC and Raster in FP3. In FP4, core 1 executes Init and VLD, core 2 executes IQZZ+IDCT, and core 3 executes CC and Raster. These two different assignments are also attributed to the time difference of VLD in core 1/2 and core 3/4. We assume FP4 would have shorter time, because VLD would be accelerated much in core 3/4 and the execution time of 3 cores are close as well. The Figure 5 above illustrates the total time in nave, FP3 and FP4 version and FP4 achieves better optimization compared to FP3 which verifies our assumption. Additionally, we applied DMA in the best performing version FP4 and we get an even more optimized version in terms of execution time (FP5). The block size we set is in accord with MCU size.

Furthermore, we intend to divide the whole JPEG process into 4 cores as well. Specifically, core 1 executes Init and VLD, core 2 is responsible for IQZZ+IDCT, core 3 performs CC and core 4 conducts Raster. However, this version of function parallel is not regarded better than FP4, because the four cores execution time could be remarkably different or uneven which would cause blocking. For example, in these 4 pipeline stages, core 2 may spend more time on IQZZ+IDCT compared to core 1 on Init and VLD, therefore core 1 may get blocked to wait for core 2. In general we assume the total execution time might be only slightly improved or even remains similar as FP4. However, applying 4 cores cost much more in energy and memory. It needs 3 FIFOs that greatly increase memory usage and core 4 is used which would cause increase in the energy consumption. This 4 core version will be compared with 2 core and 3 core version in benchmark.

All the 5 versions of function parallel utilize the synchronization protocol FIFO buffers to transfer data between each cores based on the C-Heap tutorial. The FIFO size (in tokens) are all set as 2, while the token size (multiple of 4 bytes) depends on the struct we established in communication between each core. In the benchmarking chapter, we will compare these versions from energy and memory usage aspects.

C. Hybrid parallel

The hybrid method is to make full use of the advantages derived from data and function parallel and avoid the shortcomings as much as possible. In data parallel, the bottleneck is some functions, such as VLD, has to be repeated redundantly in each core and the complexity in different parts of the picture also influence the total time considerably. In function parallel, the communication overhead cannot be ignored. Besides, the memory usage is also largely increased by using FFIO. To make a trade off on these restrictions, we figured out two hybrid versions.

The first hybrid version, utilizing core 1, 2 and 3, is inspired from the function parallel FP2 as mentioned before. This version executes Init and VLD in core 3 first and transfers data to core 1 or core 2. Then core 1 and core 2 perform IQZZ, IDCT, CC and Raster as data parallel. This kind of assignments of 3 cores could be attributed that VLD is the only part that could be accelerated in core 3/4 compared to core 1/2. The other functions, especially IQZZ+IDCT, could be slower in core 3/4. In addition, the remaining functions execution time is 2 times (or more) of Init+VLD, therefore to put the rest functions in parallel would be able to make each core work evenly so as to avoid blocking as much as possible. Unfortunately, the Cmemout address of core 3 is not enough for FIFO size of 2, therefore we changed the FIFO size to 1 in this hybrid version.

The second hybrid version was implemented to investigate the effect of using all the four cores. In this version, core 1 executes the Init+VLD functions, and transfers the tokens required by core 2 executing the IDCT+IQZZ function. The core 3

and 4 uses the data parallelism to parallelize the CC+Raster function. However on implementation, the execution time of the JPEG increased. This mainly can be attributed to the execution time of the core 2, since core 2 has to execute the IDCT+IQZZ and it also has to transfer the data to core 3 and core 4. The setting up of 3 FIFO buffers only, creates a communication overhead of around 10% (depending on the image) of the total execution time of core 2. Hence, this version is sub-optimal in comparison to hybrid version 1.

III. BENCHMARKING

In previous chapters, we have assumed and raised many design choices in data parallel, function parallel and hybrid as well as some tests from execution time perspective. The benchmarking part is mainly focused on implementing and assessing the performance (execution time, energy and memory usage) of the various optimized versions on 9 benchmark pictures. The baseline version for this section is the naive version using core 1 to process JPEG decoder with MMIO. As for energy calculation, we assume the power of core 1 and 2 is 10mW, the power of core 3 and 4 is 60mW when they are working Otherwise the power consumed is assumed 2mW.

A. Data parallel

Data parallel version of Jpeg, undoubtedly is easiest to implement but also requires the proper distribution of the workload on all 4 cores. We will be testing our assumption of the working of the JPEG on multi-core, using three different versions of data parallel on all benchmarking images. We will further refine our choices by observing the performance of the best version based on MMIO and DMA.

In the first version, the whole image was divided on the 4 cores using the header-leftover of the image in an uneven banding scheme as described in above table. Whereas in the second version, the image was divided based on header-MCU Row. This version is also implemented with work-load division in 3:3:2:2 scheme.

The modulo split scheme is presented here in 1:1:1:1 scheme. In this version of data parallel, each core executes 1 block and skips the next 3 blocks. The basic reason behind this assumption is to investigate if this version of data parallel implementation is largely free from the varying unpredictability of JPEG images. The images with noise(noise.jpeg) or varying quality(CAT.jpeg) or abrupt changes(TEST.jpeg) will introduce varying execution time even with uneven banding. Since, this version doesnt utilizes the proper workload division, hence it delivers the worst-case execution time and energy performance. But the desirable feature with this version is its least dependency on the type of image.

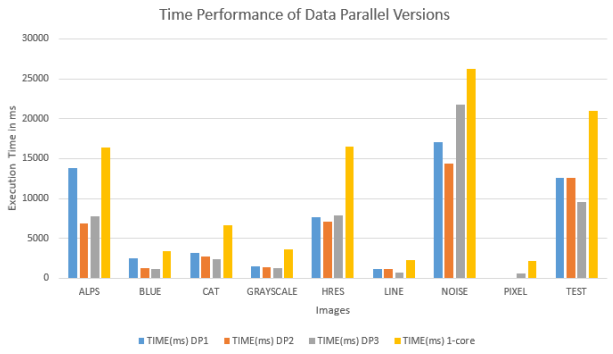

Fig. 7: Time performance of all Data parallel versions

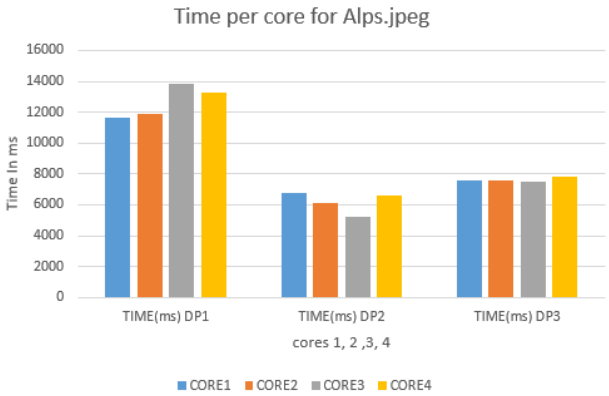

Fig. 8: Time performance for Alps.jpeg

As we can see from the graph(Fig.6), the execution time of the data parallel versions improved by more than 50% in almost all the cases (except for some images). It is also evident from the graph, that the execution time of 1:1:1:1 modulo is worse than the other two data parallel version except for the TEST.jpeg and cat.jpeg, and the data parallel version 1 takes almost twice the time in comparison to other two versions for alps image. We will investigate the reasons behind this observation individually.

The alps.jpeg image contains the restart marker which breaks the whole image in parts and hence is useful for reducing the bottleneck of running redundant blocks. The first version doesnt exploit this and hence must run all the redundant parts of the image. Hence, the core 3,4 running the last half of the image, takes more time in comparison to core 1,2. This increases the overall execution time of the image. However, in MCU-rows version, the image is dissected and this helps to reduce the sequential bottleneck of decoding the image. In the modulo split scheme, all the cores take nearly same amount of time to execute the code, which shows even distribution of image on all the cores. These reasons can be justified with graph below.

In the cat.jpeg, we can see the quality increases from left to right and somewhat from top to bottom. Hence, the cores (core 4) executing the lower quality part of the image takes lesser time than the core which executes most of the lower right corner

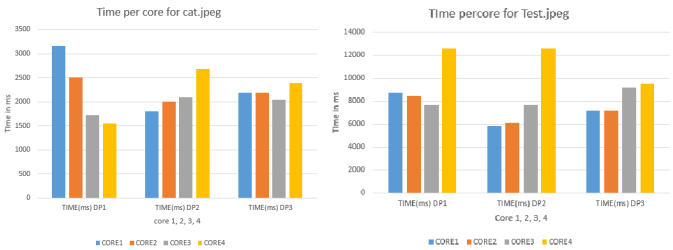

Fig. 9: Time performance of all Data parallel versions

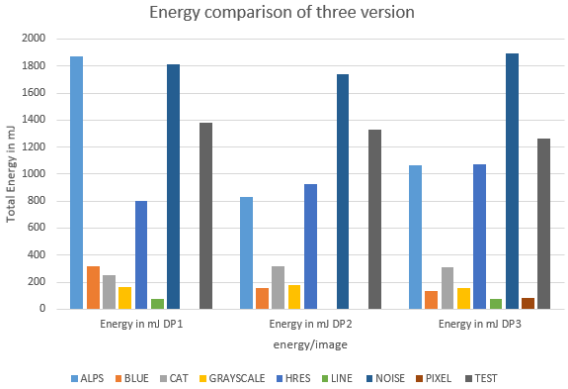

Fig. 10: Time performance for Alps.jpeg

of the image. This, un-evenness in execution time is absent in the modulo version since entire image is split evenly on all the cores.

The same reasoning can be made for the test image, since this image has abrupt changes block-wise, hence the Huffman encoding for some parts of the image takes more time than other cores. But due to equal distribution, the modulo version takes less time to execute the image. It is worth noticing that in such cases, modulo version provides an equal utilisation of all the cores, which is absent in other two versions.

Energy is also an important criterion for benchmarking. Previously, our assumption was to divide the workload on different core to achieve minimum time execution as well as proper utilisation of the cores. The even distribution of modulo split scheme should make it energy efficient, as all the cores will be properly utilised. But, still there can be few conditions which can make the cores finish at un-even time. The best example is the noise image, where core 4 and core 3 take more time than core 1 and core2 even in modulo split scheme. Hence, the overall energy for this image worsens in comparison to DP1, DP2. This is because of the high-frequency components of the Noise.jpeg, which takes more time in encoding.

Of course, by now we have analysed that in terms of workload distribution and energy consideration, the modulo split version is better even in its nave implementation. The utilisation factor of the all the cores in this modulo scheme is below 90%, which indicates that with a better scheme of modulo-split, there still a space for improvement.

B. Function parallel

In this section, three versions of FP1, FP4 and FP6 function parallel, which utilize 2, 3 and 4 cores respectively, are chosen and compared from execution time, energy and memory usage aspects. From the Figure 11 and 12 below, it indicates that applying 4 cores only slightly decrease the total execution time but consumes tremendously more energy. Specifically for noise.jpg, FP4 optimization with 3 cores is even better than 4 cores. It verifies our assumption raised in Function parallel that utilizing 4 cores will only slightly optimize the execution time or even worse. As for the memory usage, it includes data, instruction memory and CMI/O memory. We calculated total data and instruction memory usage of 2 cores, 3 cores and 4 cores respectively. Core 4 utilize more FIFOs compared to 3 core version, hence it would also consume more CMI/O memory. The performance of these three versions comparison is clear in Table 2. Therefore it is not worthwhile to use 4 cores on function parallel.

Table 2 Multiplex cores comparison

Function parallel	Memory usage KB	Energy consumption (mJ)	Execution time (s)
FP1 (2 cores)	61.5	247.4 mJ	12.37 s
FP4 (3 cores)	93.1	835.2 mJ	10.44s
FP6 (4 cores)	132.4	1480.8 mJ	10.58s

Furthermore in the benchmark section, different FIFO size (in tokens) are also measured to identify its influence on execution time. Specifically, we applied 1, 2, 3, 4, 8, 16 (tokens) FIFO size in FP1 version, because this version simply utilized FIFO to transfer data from core 1 to core 3. The total execution time of different FIFO size are dependent on different pictures as well, but the results are very close, approximately no more than 5%, which means the FIFO size contributes little in total execution time. It could be attributed to the uneven processing speed of the two cores. In the case of FP1, core 1 executes Init, VLD, IQZZ+IDCT and core 3 process CC and Raster. In each pipeline execution, the time spent on core 1 is much more than core 3 so that the FIFO buffer will be full after a short period. As soon as the FIFO buffer is full, the execution time will be the same no matter how big the FIFO size is. However, increasing the FIFO size contributes much in occupying the memory address, which would cost much more memory usage. In this case, what we use for FIFO size is 2 in general, or 1 in hybrid version 1 (HY1) due to the memory limits in core 3.

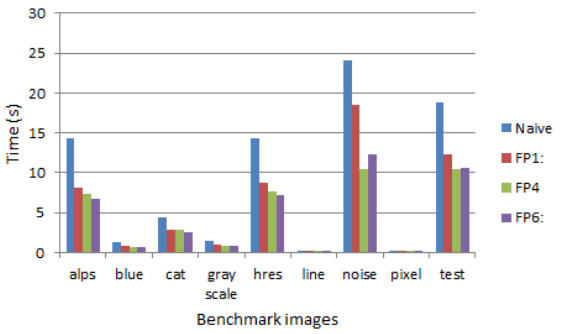

Fig. 11: 4 cores execution time

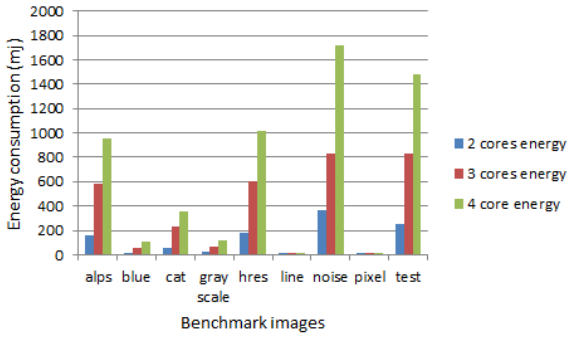

Fig. 12: Total energy consumption

C. Hybrid

The hybrid version 1 (HY1) has better time performance and consumes less memory and energy on benchmark pictures compared to HY2. Due to pages limit, the figures are not displayed here. The are various reasons for this result. First, CC and Raster would not be executed much faster in core 3 ore 4, but VLD would. Furthermore, the HY2 version utilize 3 FIFOs, which would cause much more communication overhead, compared to 2 FIFOs in HY1. As for energy, HY1 utilizes 3 cores and has shorter execution time that contributes much in saving energy compared to HY2 (utilizes 4 cores). Furthermore, HY1 applies 2 FIFOs which also cost less in memory in contras to HY2 using 3 FIFOs. Regarding the superiority of HY1, DMA is also applied in this version and the result of execution time is even better with DMA communication.

IV. Conclusion

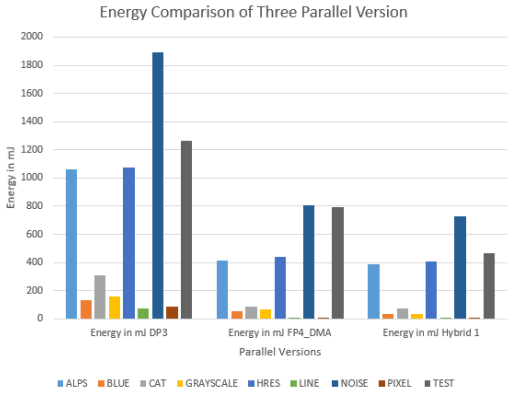

Fig. 13: 4 cores execution time

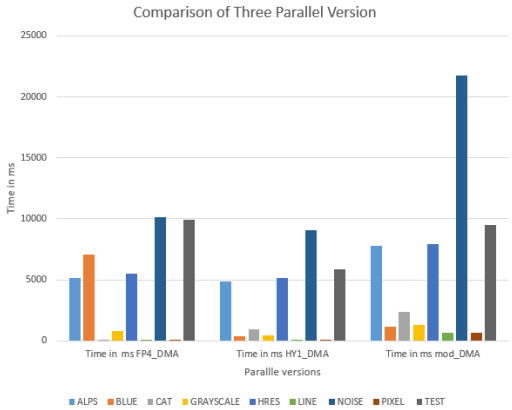

Fig. 14: Total energy consumption

During the whole process of JPEG optimization, we explored different optimization methods in data and function parallel. Particularly, we figured out the advantages as well as the bottlenecks of these two optimizations and tried to combine them to get the best hybrid version. However, the optimized version also depends on our requirements, whether we are optimizing on the basis of Execution time, Energy, or Memory. On the whole, considering time and energy parameters, they hybrid version 1 outperforms the other two parallel implementations. But still, we assume that a enhance modulo split scheme (7:7:5:5) for data parallel can provide a better time and Energy performance, as we investigated its implementation and it was giving better time performance. Although, since it didn't gave correct results for all benchmark images, hence, we didn't include it in the overall analysis. We deliver the Data parallel using, Function parallel (FP5 applies 3 cores) using DMA, Hybrid version (HY1) using DMA.

V. Reference

[1] http://www.es.ele.tue.nl/ kgoossens/2012-wese.pdf
[2] http://www.es.ele.tue.nl/ kgoossens/2012-crts.pdf