Mapping a JPEG decoding algorithm on the CompSOC platform

N. B. Student ID: -------, Dragoș P., Student ID: -------, K. Z., Student ID: -------

Group 14, Date: June 18, 2018

I. Introduction

In this report, a JPEG decoding algorithm is mapped on the CompSOC platform and optimization strategies are applied to find an optimal mapping in terms of execution time, power and memory usage.

The CompSOC platform consists of four processor tiles that use a 32-bit MicroBlaze processor. These four tiles consists of two pairs of identical tiles. The differences between these tiles can be found in Table I. It is clear that there are important differences between the two types of tiles, which will result in heterogeneous performance among the various mappings possible on the tiles. In order to converge to optimal performance, the execution time of the functions are evaluated on both types of tiles during various stages of the design process. These results are taken into account when distributing the workload of the decoder among the processor tiles in the various stages of the design.

TABLE I: An overview of the differences between the processor tiles of the CompSOC platform.

	Tiles 1 and 2	Tiles 3 and 4
Clock frequency	60 MHz	120 MHz
Pipeline	5 stages	3 stages
Accelerators	Barrel shifter, integer divide & multiply	None
Architecture	Harvard	von Neumann
Memory size (I and D)	2x 128 kB	1x 64 kB
Communication memory	2x 32 kB	4x 8 kB
	1 DMAMEM, 1 CMEM	2 DMAMEM, 2 CMEM
DMAs	1	2
Energy consumption active (idle)	9.3 mW (2.1 mW)	60 mW (2.1 mW)

The JPEG decoding algorithm works in five stages. In the first stage, the input stream is read and decoded using a variable length Hoffmann decoder (VLD). After decoding, a number of Minimal Coded Units (MCUs) are obtained. These MCUs contain at most ten blocks of frequency space values that are dequantized and zig zag scanned (IQZZ). The resulting frequency space values are converted in pixel space values by applying the inverse discrete cosine transformation (IDCT). The pixels are converted to the RGB color space in the color conversion (CC) stage. Finally, in the RASTER stage, the RGB colors are written to the correct DRAM address.

The functional correctness of the mapping is determined by running the algorithm for a set of nine JPEG images ranging from a single pixel image, to images with a resolution of 1024 by 768 pixels. The binary output of these images is compared to the binary output of the same image that is generated by a JPEG decoding algorithm running on a PC that is known to produce the correct result. If and only if the binary output of the decoder running on the CompSOC platform is equal to the binary output of the PC version for all nine images, the mapping is considered to be functionally correct.

Direct Memory Access (DMA) and Memory Mapped I/O (MMIO) are the two main options for inter-tile and DRAM communication. An advantage of DMA over MMIO is that the processor can work in parallel with the DMA and therefore does not have to wait for the data transfer to be finished. Next to that, DMA supports moving data in bursts. This means that the command and address of the data transfer need to be programmed only once when a number of consecutive words are to be send. For large burst sizes, this results in a high throughput of up to one word per cycle. Because the data that is to be send between the various cores can become quite large, as is shown in Table II, using a DMA for inter-tile and DRAM communication is preferred to get the best performance.

TABLE II: Sizes of the structures and arrays that are transferred between processor tiles

Structure name	Size (bytes)
FValue [10]	3200
FBlock [10]	2560
PBlock [10]	640
Subheader 1	25
Subheader 2	52

The problem solved in this report is related to mapping the JPEG algorithm to the CompSOC platform. Because the algorithm can be split in various stages and the processor tiles have different characteristics in terms of performance, power and the communication, the design space that can be explored is very large. However, in this paper an effort is made to find the best possible mapping in terms of execution time, power consumption and memory usage. In order to investigate the various optimization options, three types of parallelization are explored.

First, a strictly data parallel version is created that makes sure the various tiles work separately on different data without any inter-tile communication such that each tile must process the input stream independently. After that, a functional parallel is made that splits the decoding of the JPEG image between the various tiles such that in each tile a different stage of the algorithm is performed. Finally, a hybrid version is created that combines the notions of data and functional parallelism.

II. Different Versions of the JPEG decoder

In this section, the different versions of the JPEG decoder are described and both advantageseous and disadvantages are discussed. For each implementation, the assumptions that are made are described and a validation of the design decisions is presented. First, the data parallel version is described, followed by the functional parallel and hybrid versions.

A. *Data parallel*

Firstly, a data parallel implementation is created from the provided code for the CompSoC platform. The goal of this implementation is to make sure that each core processes a different and independent section of the output image. This can easily be implemented for JPEG decoding, because the full encoded image consists of multiple MCUs that corresponds to a block of x by x pixels of the output image, where x can vary. As MCUs are independent units, workload distribution naturally becomes a distribution of MCUs among tiles.

As mentioned before, MCUs are obtained by decoding the input stream. The size of the MCUs can vary and is decoded in the header of the JPEG input stream. It is therefore unknown at design time at what address each MCU starts in the input stream. In a pure data parallel implementation there is no communication between the processor tiles such that each core must process the entire input stream and pick out the MCUs that are to be processed.

*1) **MMIO 2 cores:*** First, a simple 2-core data parallel implementation is created that runs on cores 1 and 2, where core 1 processes the even MCUs and core 2 processes the odd MCUs. In this way, the workload is equally distributed among the identically performing cores - cores 3 and 4 not being used. As both cores still need to process the entire input stream, the execution time is expected to be more than half of the execution time of the original non-data parallel implementation. The power is increased from *15.6 mW* to *22.8 mW*, but because the execution time is expected to be less than half as on one core, the energy consumption is expected to be roughly the same. The memory usage is expected to be double, because the code that originally ran on a single tile is mostly duplicated to the other tile to obtain a data parallel implementation.

*2) **MMIO 4 cores:*** Similarly, an implementation is created that utilizes all four cores and distributes the workload evenly among the cores in terms of MCUs. As it is found, by benchmarking the initial code that executing the algorithm takes 20.4% longer on cores 3 and 4 than on 1 and 2, a lower execution time can be obtained by unevenly distributing the workload among the cores such that cores 1 and 2 have to process more MCUs than cores 3 and 4.

However, for simplicity it is chosen to distribute the workload evenly among all cores by means of a local counter in each core that checks whether the MCU that it has decoded from the input stream is actually the n-th MCU according to modulo 4 counting, where n is the id of the tile. It is expected that the 4-core implementation is therefore roughly 20.4% faster than the 2-core implementation. The power consumption on this implementation is equal to $138.6\ mW$, which is six times more than the 2-core implementation. The total execution time is obviously expected to decrease further than the 2-core version, as the workload is split for 4 cores, but the speedup should not be as much as from 1-core to 2-core version. However, regardless of speedup, the energy consumption of this implementation is expected to be much higher than the 2-core implementation, because of the six-times increase in power.

*3) **DMA on a single core:*** All previous implementations of the JPEG decoding algorithm used the MMIO to transmit to and from the DRAM. However, because DMA can load and store multiple consecutive words in the DRAM at once, it is expected to be advantageous to use the DMA to communicate data due to its high throughput at large block sizes. In order to use the DMA, mainly $raster.c$ and $5kk03.c$ are adjusted. In the first version, the RGB values of each pixel in $raster.c$ are now being stored in the DMA memory, rather than in communication memory. After these values are stored, the DMA is programmed to send the pixel information from the DMA memory to the correct location in DRAM. In $5kk03.c$, the word in DRAM that contains the byte that is requested by the JPEG algorithm is loaded into the memory of the DMA. The byte is then extracted from the word and returned by the function.

*4) **DMA prefetching:*** In the previous simple version, the advantages of the DMA are not fully exploited - merely the DMA is only switched from MMIO. In this current, more optimized version, the DMA stores the multiple pixels' information of an MCU in DRAM only when it has processed an entire row. In this way, the block size in a DMA transaction is increased to the number of pixels that are in a single column of a MCU. To reduce the number of DMA load transactions, a buffer is created in the $FGETC$ function of $5kk03.c$ that locally stores 32 consecutive words in the memory of the processor tile. The buffer is updated with 32 new words when the byte that is requested is not present in this buffer. This number is chosen to be 32, because it is equal to the maximum burst size of a single DMA transaction. Further increasing this value would split the data transfer in multiple transactions, whereas decreasing the number would underutilize the slots that are available in the transaction, leading to a lower throughput.

*5) **DMA prefetching with pingpong buffer:*** In order to make sure the processor does not have to wait before it has received data using the DMA, another local buffer is added to optimize the previously implemented prefetching. With this additional pingpong buffer, the 32 words that are expected to be needed after the first buffer is depleted are loaded in the background. Once all data in the first buffer is read and the processor starts to use the second buffer, the DMA is programmed to load the next 32 words from DRAM into the first buffer, in the background. In this way, the processor can do work that contributes to the end result during a transaction and does not have to wait before the data from DRAM has become available. Blocking calls are therefore also replaced by non-blocking ones. After switching between the two buffers it is checked whether the second buffer is loaded with the data from the correct position in memory.

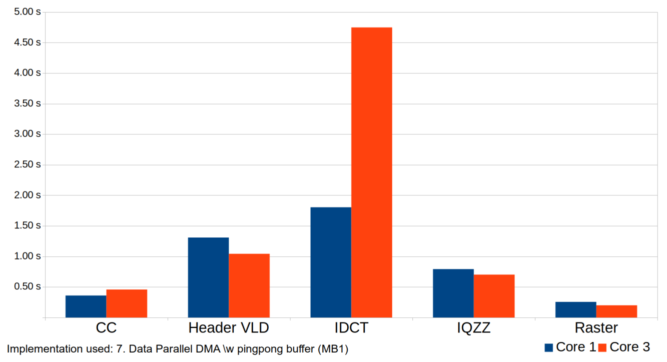
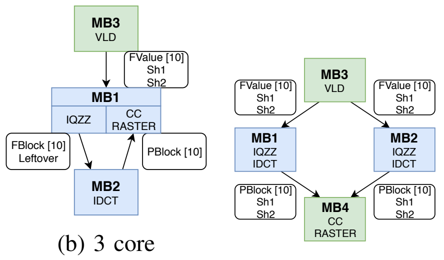


Fig. 1: Execution times of all functions benchmarked on small and large tiles



(a) Energy efficient functional.
(b) 3 core functional.
(c) Hybrid design.

Fig. 2: Schematic representation of the distribution of the workload among the various tiles.

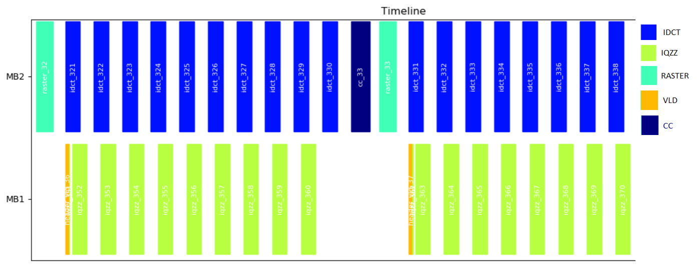


Fig. 3: Timeline of energy efficient functional parallel cores

The blocking method of programming the DMA in the *raster* function is also replaced by non-blocking. This can be done because the data that is sent in this function is the final output and other functions do not depend on this output data. Additionally, another check is added at the beginning of the *raster* function to make sure that the previous values in the DMA memory are not overwritten before they are transferred to DRAM. After the check, new values - being pixel values of the decoded image - can be placed in the DMA memory again. In order to make sure the program does not terminate before all data is correctly written to DRAM, it is checked whether the DMA is finished at the end of the the *main* function. In this way, it is made sure that the DMA has completed all its transactions before the program terminates.

*6) **DMA final, optimized version** :* It is also possible to make sure that a single tile processes multiple consecutive MCUs to increase the possibility that the correct data is prefetched in the $FGETC$ function. Note that this optimization does not come at the cost of increased memory usage, since the allocated memory is only used by one MCU at a time per core. This optimization is performed in the final version of the data parallel design.

B. Functional parallel

Another way to speed up the JPEG decoder is by implementing a functional parallel solution. The goal is to evenly divide the functions over the available cores, resulting in a reduction in overall computation time. The JPEG decoder's functions have different execution times on the small and large tiles. Therefore, before implementing a solution, it is essential to determine what computation times the different decoder's functions have, on the different cores. During the determination of what functionality should be run on what core, the difference in energy consumption on the small or large tiles will also be taken into consideration.

*1) **Benchmarking the functions**:* From the benchmarking Figure 1 it can be observed that certain functions take less time on the smaller tile than the big tiles. However, this result does not seem to be large enough to justify the additional cost in power. Therefore in the power-efficient implementation, only the large, power-efficient cores, will be used. It can also be observed that the IDCT takes a lot less time on the slower, larger cores than the small cores, which is most likely due to the accelerators present in the large tiles. It is obvious that those functions should be scheduled on those cores in almost all circumstances as it is both faster and costs less power, thus more energy efficient.

*2) **Energy efficient core 1 & 2**:* When only using the two identical large tiles, the best solution is to try and distribute the workload evenly the workload on both cores. However, all functions have to be executed in a specific order. This results in the fact that the perfect distribution, namely both cores have the same workload, of the functions requires exchanging data between the cores multiple times creating overhead. Therefore, it was decided to take the approach which introduces the least amount of overhead, by only transferring data once, as is shown in Figure 2a.

In an ideal case, the speedup factor compared to the single core version should be two, while the energy consumed stays the same. This is obviously not going to be the case as the functions are not completely evenly divided between the cores and the communication between the cores introduces an overhead. As DMA is used for the communication between the cores, the processor can do other instructions during the communication, hence the overhead is reduced to the minimum. To get a better view of the division of workload between the various cores and verify that the benchmark measurements were correct, a timeline was created using the time-stamps at the beginning and end of the functions as is shown in Figure 3.

When taking a look at the steady state of an iteration (buffer is filled), it can be observed that the next iteration of MB1 starts after the completion of the RASTER function on MB2.

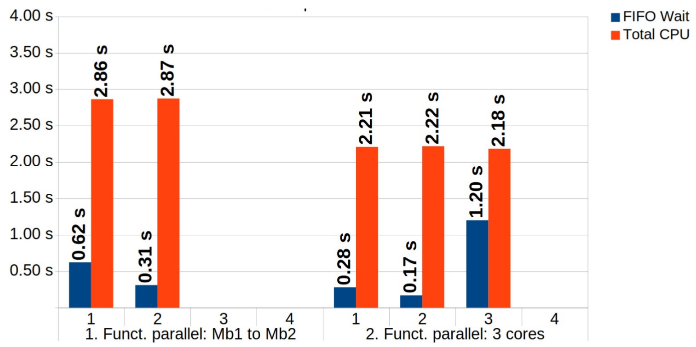


Fig. 4: C-Heap executing vs waiting graph functional parallel implementations for average case

This is due to the fact that MB2 releases the token after it finishes the RASTER function. This shows that MB1 has to wait for MB2 to finish with its code before getting a new MCU. It can be concluded that for this program, the execution time of an iteration on MB2 is longer than on MB1.

*3) **Fast functional parallel with switching**:* The other goal was to create a functional parallel version that was designed with execution time as its priority. When taking another look at the benchmark it can be observed that specific functions take less time on the small tiles. In case that energy consumption comes secondary to speed, it becomes interesting to use the small tiles for decreasing the overall computation time. When looking closely at the benchmarking for the average case, it shows that the VLD is faster on small tiles than on large tiles. Therefore it seems a logical choice to start with the VLD on a small tile. When taking a closer look at the remaining functions, it shows that the IDCT takes longer than the other 3 functions combined. It can also be observed that for the average case, it makes no sense to add a 4th core to do functionalities as well, as the IQZZ, CC and RASTER can be executed before the IDCT will be finished. As it will be seen in an example later, in practice this is not always the case. However the IDCT function has to be executed between the IQZZ and the RASTER functions, which creates a complication. To overcome this problem, a solution was created as is shown in Figure 2b.

As the DMA memory of the small tiles is not that large and the token they have to send are quite large, it can only hold two tokens. To save some DMA memory, the FGETC function used by VLD that also uses DMA is transferred to mb3_dmamem1 instead of dmamem0. Although not much, this does give us an extra 64 bytes of memory to use, prevents any conflicts from happening and distributes the workload between two DMA's. The second and larger problem that has to be addressed is that core MB1 has to alternate between two different tasks. Namely the execution of IQZZ and the execution of CC and RASTER. In the case that one buffer is full or empty, MB1 should not wait until the buffer issue is resolved, but should instead move on to the other functions that it can executed. It should also not execute the CC and RASTER more times than it has executed the IQZZ, as there won't be any data to process or it will repeat the processing of old data, adding redundancy. It is also not allowed to execute the IQZZ function more times than the buffer between MB1 and MB2 allows to store, as data will be lost. Therefore MB1 has to keep an internal counter of how many of both tasks it has completed to prevent the premature or late execution of those tasks.

When designing this functional parallel, the choice was made to place the IDCT on a separate core as it was the slower than all the other functionalities on MB1. When carefully inspecting the CHEAP wait time Figure 4 , it shows that MB2 in some cases does have to wait for the CHEAP. In other words, MB1 is limiting the throughput of MB2, which is the opposite of what was concluded before implementing this design. When taking a closer look at the actual benchmark results specific for Alps it indeed shows that IQZZ, CC and RASTER together have a longer execution time than IDCT. This shows that, despite this solution providing the fastest execution time for the average picture, it does not hold that it has the fastest execution time for all the pictures separately. Thus implementing a functional parallel solution with 4 cores does improve the execution speed for some pictures. When taking a look at the average FIFO wait time regarding the energy efficient solution in Figure 4, it shows that both cores have some FIFO wait time. Thus in this version the throughput is limited by a different tile for different pictures. When using a functional parallel solution, the efficiency of the solution depends a lot on the input image, which is in some cases a disadvantage. The overhead on the other hand, is kept to minimum by only DMA transfers and internal counters, as no redundant work is done such as executing the VLD more than necessary. This is represented in the low energy usage of the energy efficient solution shown in Figure 7. The disadvantage of functional parallelism can be overcome in a hybrid solution, as it is possible to implement a data parallel solution for the functions that vary in execution time.

C. *Hybrid*

When data parallelism is combined with functional parallelism, a hybrid design can be obtained. A well designed hybrid design, exploits the benefits of both functional and data parallelism. In order to create a good design, an effort is made to make sure that the execution time of various stages in the design is equally distributed. An example of

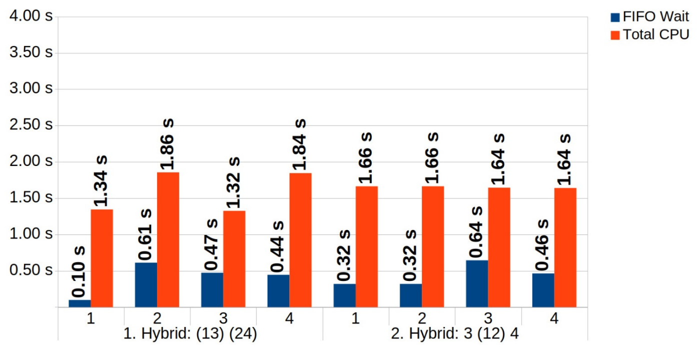


Fig. 5: C-Heap executing vs waiting graph hybrid implementations for average case

this is to apply data parallelism to stages that have a long and variable execution time. Also, opposed to a fully data parallel and functional parallel version, it is now possible to execute the VLD on a single core and distribute the workload of processing the resulting MCU's on different cores in a data parallel fashion. In order to make the design decisions, the performance of functions on both large and small tiles, as is shown in Figure 1 is taking into account.

1) ***Simple Hybrid (MB3 to MB1 and MB4 to MB2):*** At first, a simple hybrid design is created that uses the functional parallel design where the algorithm is executed on tiles 1 and 3. Because tiles 2 and 4 offer the same performance as tiles 1 and 3, the MCUs can be easily split between the two sets of tiles. The functionality is therefore copied to tiles 2 and 4. Both tiles 3 and 4 only forward half of the total amount of MCUs that need to be processed in a way such that together they decode the entire image. A very rough estimation in the improved execution time of this simple design is that it is equal to half of the execution time of the functional design on which it is based. However, because at least one of the tiles must apply VLD to the entire input stream, the actual speedup is expected to be a less than 2. In terms of memory usage, the code from the functional parallel design is mostly copied to the other cores. The memory usage is therefore expected to be twice as high as the functional parallel design based on tiles 1 and 3.

2) ***Hybrid (MB3 to (MB1 and MB2) to MB4):*** After that, a hybrid design is designed from the ground up. In this more optimal design, the VLD is performed on tile 3. The resulting frequency values are then equally distributed among tiles 1 and 2. These tiles perform IQZZ and IDCT in parallel to each other. The resulting blocks of pixel-space values are then sent to tile 4. This tile performs CC and RASTER in order to get the final output image. A schematic representation of this design can be found in Figure 2c. When taking a look at the benchmarking results in Figures 7 and 6, it shows that the energy usage of this implementation is relatively high in comparison to the functional parallel implementations. Due to using both the small tiles, the energy consumption is expected to be relatively high. As it utilizes all 4 cores, the execution time is relatively low.

3) ***Improvements on hybrid design:*** Finally, some improvements on the previously mentioned design are proposed. The previous design sends the subheaders that are extracted during VLD to tiles 1 and 2. However, these tile do not need the subheaders and only forward them to tile 4. Instead of sending them to tile 1 and 2, the subheaders could also be send from tile 3 to tile 4 directly. This reduces the execution time by a factor that is proportional to the time that is wasted copying the subheaders in tiles 1 and 2, as the throughput is limited by the execution time of cores 1 and 2. Doing this will also slightly reduce the amount of memory usage. As some of the token sizes can be reduced by 77 bytes according to Table II.

4) ***Task pool:*** In order to obtain the lowest possible execution time, the workload must be optimally distributed among the various tiles. In order to obtain this optimal distribution it is considered to create a task pool. Such a pool consists of all MCUs that still need to be processed. Once a tile is ready to start to work on a new MCU it asks the pool which MCU it should work on. The pool will respond with the appropriate MCU such that no tiles work on the same MCU simultaneously and that all MCUs are eventually processed. The overhead of creating this pool in terms of latency and memory usage is not expected to outweigh the performance increase in terms of execution time. It is therefore chosen not to implement this optimization strategy.

III. Benchmarking

Benchmarking has been automated. As the JPEG decoder is going through many versions in order to be optimized, it is obviously advantageous to create a benchmark that automated some tasks, for faster testing and easy identification of the best version implemented in terms of execution time and energy.

The benchmark itself automatically injects function wrappers and caters to the whole compilation chain of each version of the JPEG decoder implemented, in order to obtain $.csv$ files that can be imported and studied, these files are obtained from the outputted debug prints. The resulting binary images are also automatically compared to reference ones for correctness. Two benchmarks are created: a benchmark testing execution time of functions and cores (for cores, see Fig. 6), as well as a benchmark creating timelines to help with a visual organization of the functions, mapped differently on cores, as seen in Figure 3).

The benchmarks created are taking care of the finer details, such as: substracting times taken to print debug information from benchmarked timings, making sure all $Makefile$-s have the same $OPTFLAGS$ (imposing $-O3$) and giving correct image resolutions to $Makefile$-s as parameters.

The benchmark's main goal is obtaining the execution times of all implementations, over all cores and images, for the average, best and worst case scenarios. In this way, the JPEG decoder functions could also be ranked by their time spent on the running system, to help with their mapping on cores, for functional parallelism and hybrid implementations. For graphs, it is chosen to depict the average time of execution of the whole system over all images (instead of worst case, for example), as the 9 images given for benchmarking are of various complexities, as was found after benchmarking the

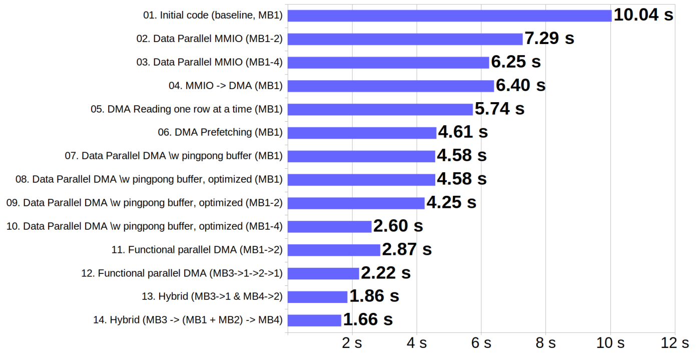


Fig. 6: Average execution time on all implementations for the average case

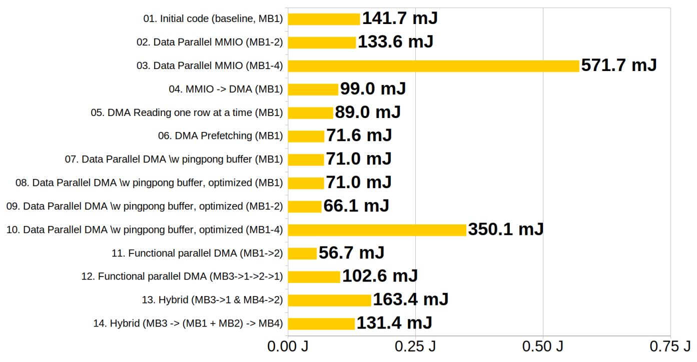


Fig. 7: Energy usage for all implementations for the average case

results, are considered as depicting real-life scenarios of a JPEG decoder.

Having obtained the execution times among all cores, the total execution time could then be considered as being the maximum execution time among all cores. The energy was then easily obtained (see Figure 7), any core finishing faster than the slowest core being considered as still idling until all of the cores are done. This challenged some initial assumptions, as some implementations running on one core are not as energy efficient as some implementations running on two cores. As a result, some multi-core implementations finish faster and the inactive cores spend less time idling, therefore wasting less energy than their 1-core counterparts.

The *C-HEAP FIFO wait times* are of interest, as it is suspected that functional parallel and hybrid versions could be open for further optimizations, by minimizing the difference in workloads between the cores, as full or empty FIFOs are causing waiting times for different cores. As an example, one remarkable result that is to be noticed after such a benchmark is observed in Figure 4. The 3rd core of the 3-core functional parallel version experiences wait times for more than half of its own execution time, as the token buffer between MB3 and MB1 is often full. Such wait times issues are depicted on all cores of the functional and hybrid versions, in Figures 4 and 5. Optimizations over this issue are not implemented as they are not very trivial to make, since specific images create a different balance between the workloads of the cores.

Last but not least, the benchmarks produced also helped in making other decisions and visualizing statistics not depicted in this report (e.g.: time of execution per image and execution time per core).

Note: For the timeline version of the benchmark, code was provided by E. Wouters, a member of another team, that, when provided a correct *.csv* file, the script would strictly only draw the timeline. It was a matter of interpreting the output prints to turn them into the *.csv* file, that was needed to graph the timeline, that was hard and left open to interpretation.

IV. Conclusion

It can be seen that the most optimal version in terms of memory usage and energy is based on a single core. This is mainly due to the overhead that is introduced when extending the decoding algorithm to multiple cores. This overhead consists of the fact that VLD must be executed on every core in a strictly data parallel version and that implementing FIFO buffers introduces latency as data need to be copied between tiles in functionally parallel and hybrid designs. Due to his additional latency, an implementation on N cores can never achieve a speedup of N. The energy consumption however, can be decreased slightly, due to idling cores also consuming power. In terms of memory usage, there are always sections of code that are the same in data parallel and functional parallel designs. A single tile design is therefore best if only memory usage is considered.

From the Pareto graph in Figure 8, it can be concluded that there are 4 Pareto optimal designs. If the execution time must be as low as possible at the cost of energy consumption and memory usage, the Hybrid (MB3 to (MB1 and MB2) to MB4) is considered to be best. If the lowest possible energy consumption is required, the Functional Parallel DMA (MB1 to MB2) is preferred. This is due to the fact that it only runs on power efficient tiles 1 and 2 and has a low execution time. If the amount of memory that the design requires is very limited, it might be considered to use the Initial code (baseline, MB1) design. However, for a small increase in memory usage of less than 14%, both the speedup and energy consumption can be decreased by a factor of roughly 3. The last Pareto optimal design Functional parallel DMA (MB3 to MB1 to MB2 to MB1) finds a tradeoff between energy consumption and execution time and is therefore part of the Pareto dominant designs.

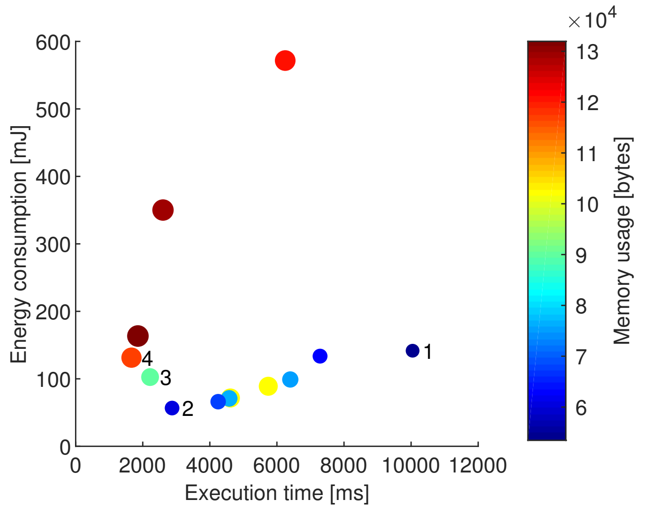


Fig. 8: Pareto graph of the various JPEG decoder designs where the numbers indicate the pareto dominant designs: (1. Initial code (baseline, MB1), 2. Functional parallel DMA (MB1 to MB2), 3. Functional parallel DMA (MB3 to MB1 to MB2 to MB1), 4. Hybrid (MB3 to (MB1 and MB2) to MB4))