Image Upscaling on UPmem using Convolutional Neural Networks

Dominik Ochs, dominik.ochs@stud.uni-heidelberg.de
Maximilian Mielke, maximilian.mielke@stud.uni-heidelberg.de
Leandro Borzyk, leandro.borzyk@stud.uni-heidelberg.de

August 15, 2024

Abstract

The rescaling of images represents a crucial undertaking in a multitude of applications. Although deep learning methods such as SRCNN have markedly enhanced the quality of image upscaling, they frequently necessitate substantial computational resources. This project is focused on the implementation of SRCNN inference on the UPmem platform, with the objective of leveraging its efficiency in memory-intensive applications. By harnessing the high memory bandwidth and parallel processing capabilities of UPmem, the aim is to accelerate the image upscaling process while maintaining high quality. Results show improvements over single-threaded CPU, but fail to perform as well as accelerators such as GPUs.

1 Introduction and Application Description

The following will detail the chosen application and the concrete implemented model.

1.1 Chosen Application – Image Super-resolution

The application chosen for implementation on UPmem is image upscaling using the SRCNN deep convolutional neural network [1]. This application is crucial in several domains, including image processing, web development and computer vision, where high quality image upscaling is essential. SRCNN, a relatively lightweight yet powerful model, enhances images by combining classical bicubic interpolation with deep learning techniques. By implementing this model on the UPmem platform, we can leverage UPmem's efficiency in handling memory-intensive, data-centric tasks, making it well suited for the parallel and memory-intensive operations required by CNN-based image upscaling.

1.2 The SRCNN Modell

The Super-Resolution Convolutional Neural Network (SRCNN) [1] represents a significant advancement in the field of image upscaling, offering a highly effective and streamlined approach to single-image super-resolution. The SRCNN model, introduced by Dong et al. [1], is designed to enhance low-resolution images by recovering high-resolution details.

The model's architecture comprises three lightweight convolutional layers, with layer one and two having a Rectified Linear Unit (ReLU) activation function and layer three not having any activation after the convolutions. To facilitate the process, the input image is initially upscaled using bicubic interpolation and subsequently converted to the YCbCr colour space. SRCNN is designed to process solely the luminance (Y) channel, which significantly reduces the computational complexity while maintaining visual quality. A simplified version of the overall process is shown in figure 1.

This strategic design choice addresses the need for efficiency in resource-constrained environments, as exemplified by the UpMem platform. SRCNN's performance exceeds that of traditional methods such as sparse coding and bicubic interpolation, rendering it a practical tool for a range of image processing tasks.

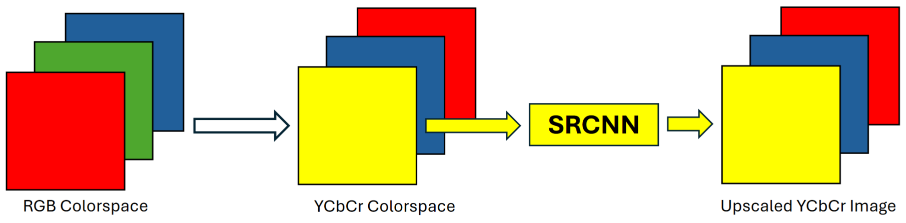

Figure 1: SRCNN Pipeline without bicubic upscaling

For the SRCNN model, we used a pretrained network to focus solely on inference due to the complexity of training on UPmem. The model was entirely rewritten in C from scratch without relying on libraries to facilitate easier implementation on UPmem. We performed inference using the Set14 dataset [2]. The results of the AI image upscaling are displayed in figure 2.

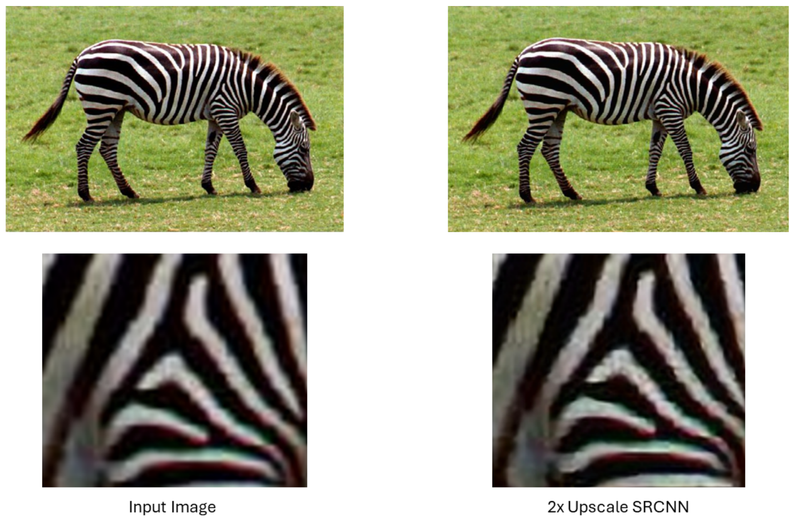

Figure 2: Left: Input image; Right: After bicubic upscaling and SRCNN processing

2 DPU Implementation

2.1 Summary

We have first implemented the inference of SRCNN in C using pretrained weights loaded from disk. We then ported the convolutional layer to the DPU accelerators and made incremental improvements to the DPU and host code.

The input image is pre- and post-processed on the host, as this code is very efficient and fast. The performance bottleneck is given by the convolution layers, which require hundreds of thousands of multiplications on more or less the same data. This is why we ported the convolution layer code to the accelerator Upmem PIM DIMM. Since computation of a single pixel on the input image is independent of computations on the other pixels, we split the input image into several tiles, where each tile is computed by one DPU within the PIM DIMM (more on that in the following section 2.2). Within each DPU, multiple tasklets share the work to compute the result of applying different independent filter kernels to the image. The DPUs also perform ReLU activation in the applicable layers. After all tiles are computed by the DPUs, the output is joined, tiled again and passed to the next convolution layer. This is necessary, as the next convolution layer can not be simply computed on the same DPU and tile without knowing the result of computation of other DPUs from the previous layer because some pixel (in the border regions of the tile) depend on computation done in another DPU, so the outputs are joined together and re-distributed again. The data has to be transferred back to the host which does the joining and tiling because DPUs can not communicate with each other except through the host.

2.2 Work Division

As explained in the summary, the work to process an image with the SRCNN is divided by tiling the input image (and following feature maps) into several patches. These tiles have a small overlap according to the filter kernel size of the layer. The tiles are generated row-wise.

Currently, the input image is tiled as given by the numbers of DPUs in use. So if 4 DPUs are available, 4 tiles are generated. Each DPU then processes one tile. The execution in the host code starts all DPUs asynchronously and subsequently waits for them to finish their respective work. Each DPU recieves its tile and all weights and biases of the layer. This is necessary since each pixel has to be convolved by every filter, therefore requiring all weights. However, since the network is so small, the largest layer contains only about 2 MB of weights.

Since each DPU has 64 MB of MRAM storage, the size of the image that can be processed depends on the number of DPUs in use. This is suboptimal in the current implementation. To alleviate this problem, the tiling process could be independent of the number of DPUs and create tiles of a preset size, so they fit into DPU MRAM memory of 64MB. This way, a single DPU could process images of any size. However, this approach can lead to a performance bottleneck. As an example, given there are 2 DPUs, but the input image is divided according to a fixed size to 3 tiles. The DPUs will each compute one tile, but the last tile will be computed by one DPU alone, meaning the other DPU idles. If the image is divided according to DPU number (our current approach), this image is divided into 2 tiles, therefore utilizing all DPUs evenly and maximally. Since we aim for maximum efficiency, we stay with the approach of dividing by DPUs. We will further discuss this in section 6 *Scalability*. Our current approach ensures that the execution time of the convolution layers is approximately sped up by the number of DPUs in use: $t_{total} \approx \frac{t_{sequentiel}}{\#DPUs}$.

Within each DPU, we use tasklets to compute the various filters of the layer. Tasklets are used to fill the DPU pipeline more efficiently and can result in better DPU utilization. However, they are not computed in true-parallel and their usefulness saturates between 11 and 16 tasklets [3]. Increasing the number further will likely *not* improve performance. In our code, the tasklets run completely independent of each other, so no synchronization overhead is required (except before starting the convolutions to initialize cache). In each layer, there are many filters. For example, layer one features 64 filters. So each tasklet computes multiple filters. The i-th tasklet computes all filters were $\#filters \mod \#tasklets = i$, so if there are 16 tasklets, each tasklets computes 4 filters. This means that each tasklet convolves each of its filters across the entire image. This operation is independent of any other filters, requiring no synchronization.

3 Handling Floating Points

Since floating point operations are emulated in software on Upmem DPUs [4], the frequent floating point multiplication and addition to the accumulator in the innermost loop is a severe performance bottleneck. We try to alleviate this problem by quantizing the floating point inputs and weights into integers, and then perform integer mulitplication and addition. The addition is going to be very fast as compared to floating point addition, but the integer multiplication is also emulated in software [4]. However, the integer multiplication is likely still going to be much more efficient compared to the floating point multiplication, as the latter requires at least one integer multiplication along with many other operations.

This quantization step is the first optimization step that is an approximation. All the other steps do not alter the outcome. Additionally, performance could be further improved by performing an approximate integer multiplication as opposed to an exact one. However, we did not pursue this, as the approximation caused by the quantization already impacted image quality visibly. This is because we could not use a large quantization factor for our simple constant scaling quantization approach because the accumulator would overflow. Since the integers are multiplied, the quantization factor is practically squared, leading to large numbers very quickly.

4 Data Transfers

The convolutions require the following data to be readily available: the input image or feature map (hereafter *input*), the filter kernel weights, and the filter kernel biases. Furthermore, the data needs to be written to an output buffer since the input might be required by another tasklet and can not be overwritten. Since the biases for each layer are very small (the largest has 64 elements), the biases are written to WRAM by the main tasklet (index = 0) before the convolutions start. The input and weights, however, are much larger and can not fit into the 64 kB of WRAM. Therefore, they need to be incrementally written to a cache. Since each tasklet operates on different inputs and weights at a time, they respectively receive their own WRAM cache for input and weights. For each output pixel, the current filter kernel is applied to the corresponding input image area. This operation in itself is three-dimensional because the filters have a certain height, width and depth, where the depth corresponds to the input channels from the previous layer. Due to filter and input being three-dimensional, the input and filter are in memory only contiguous in rows. This is best explained by an example: Let's consider a 1000x1000 pixel input image and disregard channels. The image is organized in memory as follows: First comes the first row of 1000 elements, then the second row of 1000 elements and so on, until all 1000 rows with 1000 elements each are listed. When we now apply a 3x3 filter kernel to the image and want to calculate for input pixel (20, 43), then we need (among others), the contests of memory addresses 20043, 20044, 20045 for one row but also 21043, 21044, 21045 for another row. This means, only 3 pixels are contiguous in memory. Therefore, it only makes sense to copy rows of filter width to WRAM using DMA. The filter widths for layer 1, 2, 3 are 9, 5, 5, respectively. So not a lot of data can be copied with one DMA access. However, it should still speed up memory accesses dramatically, because an MRAM access is now only required every ninth or fifth multiplication as opposed to every multiplication.

Due to the same reason, the output buffer is written directly and not through a cache. Here, the output elements are not contiguous in memory at all (a write needs to be done every 1 element) and resemble random access. This could potentially be fixed by restructuring the output. However, this would add immense complexity as the output buffer needs to be restructured again when it is passed to the next convolution layer in order for the rows to be contiguous. Additionally, compared to reads from input and weights, writes to the output are relatively rare (it is only in the third of 6 nested loops), so the performance impact is negligible.

Due to the mentioned reasons, the number of bytes read from MRAM to WRAM can not be controlled in the convolutions. However, this number should be increased where possible to use the cache and speedup through batched DMA to the best extent.

5 Implementation Process

One challenge was to work with the memory layout of the DPUs and the differences between MRAM and WRAM. Specifically, that MRAM has to be 8-byte-aligned while WRAM is 4-byte-aligned. When working with 4-byte integers, this can cause issues in copying of the data between them. For example, we faced a hard-to-find bug when we were copying from MRAM to WRAM with a non-8-byte-aligned, i.e. an uneven, address. The DPU SDK silently rounded down the uneven address to the next even address, thereby shifting the intended contents by one 4-byte integer. We think it would be better if the SDK threw an error in this case or at least documented this behavior. The documentation for `mram_read` and `mram_write` only states that the number of bytes to copy must be a multiple of 8, not the source or destination addresses. We fixed this issue by adjusting the indexing to read from to $idx - 1$ if idx is uneven. Here, the indexing was based on 4-byte integers. With this offset, we could correct the problem.

Additionally, a challenge was posed by the limited main (MRAM) and cache (WRAM) memory available to each DPU. This way, we had to calculate how much memory each DPU can take and not assume endless memory, which is the default in today's programming. To circumvent the low memory of only 64MB per DPU, we split the input image into independent tiles. Also, compared to CUDA programming where any thread can access the entire memory, it is more complicated to program DPUs, but we also acknowledge the benefits this can bring. As the memory is more local for a DPU (even the MRAM), through clever, memory-aware coding, more efficient code can be written.

6 Scalability

As already elaborated in section 2.2, we tile the input image to handle scalability. In our current approach, the work is split equally between all DPUs, leading to the highest efficiency, but the input image size is limited by how many DPUs are available on the system. In the other approach we described in section 2.2, the efficiency would be lower, but even one DPU could handle any size of input images. In that approach, the input image would be tiled into chunks of a fixed size that fit into a single memory bank of 64 MB belonging to one DPU. However, this would not cause the most efficient speed-up.

Probably the best possible solution would be to combine both approaches by introducing a thread pool, or rather *DPU pool*. In this approach, the image would be divided into as many patches as there are DPUs as long as these patches do not exceed the permissible size. If the image is so large that the NB_DPU patches would exceed the permissble size, the patches are capped at the maximal permissible size, so there are more patches being generated than there are DPUs. These remaining patches would then be processed in a thread pool approach. So the first DPU that finishes its initial patch will move on to another patch. This repeats until all patches are processed. If the image is very large, so the number of patches is a multiple of the number of DPUs, this DPU pool approach ensures maximal utilization of the given DPUs. However, the implementation of such a pool and the described tiling strategy is rather complex. The host code needs to repeatedly check each DPU for its status to find free DPUs that can be started with another patch until all patches are processed. There is no such pool in the existing host SDK. A naive implementation could result in very high host utilization and thus power consumption, when in the other approaches, the host is mostly idling and waiting for the DPUs to finish, therefore not requiring much energy.

Another bottleneck for scalability could be the size of the weights. We already somewhat support scalability for this as larger CNNs only introduce more layers, and we already split the forward pass by layers, therefore a DPU only needs to hold weights of one layer. For the small CNN we implemented, all required weights of a layer fit easily into DPU memory, with the largest layer only requiring 2 MB. However, if layers get much larger, for example with more filters, at some point, which filter kernels a DPU calculates must also be split in addition to splitting the input. In this case, for example, 2 different DPUs could work on the same input patch, where one DPU calculates the first half of the filters, while the other DPU calculates the second half of the filters. This would further make the code more complex, but is achievable to support any CNN.

In terms of scaling hardware, more ranks and PIM DIMMs can be added to the system to have access to more DPUs which can share the work. This can easily be achieved by slightly modifying the current approach to share the generated patches across DPUs on multiple PIM DIMMs as compared to on only one PIM DIMM.

7 Performance Results

The CNN used for upscaling was run on both the traditional CPU and Upmem's DPUs that were provided for this project. The CPU code was compiled with two optimization levels: `O0` (no optimization) and `O3` (full optimization). For the DPU implementations, various configurations were tested with different numbers of tasklets and different numbers of DPUs. The image used for testing the upscaling had a resolution of 400x266 pixels.
The experiments measured the execution time of the upscaling process in different scenarios. The following execution times were observed:

- **CPU Execution:**
  - `O0` optimization: 38.77 seconds
  - `O3` optimization: 7.47 seconds
- **DPU Execution for Integer Multiplication:**
  - 12 tasklets, 64 DPUs: 1.39 seconds
  - 16 tasklets, 64 DPUs: 1.37 seconds
  - 24 tasklets, 64 DPUs: 1.41 seconds
  - 08 tasklets, 64 DPUs: 1.44 seconds
  - 04 tasklets, 64 DPUs: 1.54 seconds
  - 12 tasklets, 32 DPUs: 647.90 seconds
- **DPU Execution for Floating Point Multiplication:**
  - 12 tasklets, 64 DPUs: 1.78 seconds

The results reveal that the Upmem DPUs significantly outperformed the traditional CPU, particularly when compared to the CPU code compiled with `O0` optimization. Even when the CPU code was optimized at `O3`, the DPUs provided a substantial performance boost, with execution times ranging from 1.37 to 1.54 seconds for most configurations. This demonstrates the efficiency of in-memory processing for this specific task. However, a notable exception was the configuration with 12 tasklets and 32 DPUs, which resulted in an unexpectedly high execution time of 647.90 seconds. This suggests that the performance scaling may be sensitive to the number of DPUs used, and that the system may require a minimum threshold of DPUs to operate efficiently. Another explanation could be that there is something wrong with the code. However, the results produced by 32 DPUs and 64 DPUs are exactly the same, which makes it unlikely that there is a bug.

The float multiplication task on DPUs also performed better than expected, considering that float multiplication is emulated on DPUs, with a time of 1.78 seconds, further highlighting the capabilities of DPUs in handling computational tasks. Nevertheless, employing quantization yielded a notable performance improvement from 1.78 s to 1.39 s.

The number of tasklets used does not majorly impact the performance once a certain threshold is exceeded. In our testing, the difference between 12, 16, and 24 tasklets was minor and likely due to natural variance. However, using a low number of tasklets, such as 4, impacted the efficiency negatively.

The project demonstrates the potential of Upmem's PIM DIMM technology. However, it has to be noted that convolutional neural networks might not be the best application for the DPUs since CNNs are rather compute-heavy but the DPUs are designed for simple data-centric arithmetic (supporting no hardware integer multiplication or floating point operations). This is illustrated by the following proposition: The CPU code was only single-threaded and reached 7,47 seconds. If all 32 cores of the test system were utilized, the CPU would have presumably outpaced the DPUs due to its hardware arithmetic capabilities. Similarly, a low-end graphics card takes only a few milliseconds to perform the task (measured via the python implementation also submitted with the code).

8 Conclusion

As a conclusion, the results suggest that in-memory processing on DPUs could be a promising approach for tasks requiring high throughput and low latency. Future work should explore optimizing the use of DPUs for different neural network architectures and investigating the reasons behind the performance drop observed with lower DPU counts.

References

[1] C. Dong, C. C. Loy, K. He, and X. Tang, "Image super-resolution using deep convolutional networks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 2, pp. 295–307, 2015.

[2] J.-B. Huang, A. Singh, and N. Ahuja, "Single image super-resolution from transformed self-exemplars," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5197–5206, 2015.

[3] J. Gómez Luna, et al., "Understanding a modern processing-in-memory architecture: Benchmarking and experimental characterization." `https://people.inf.ethz.ch/omutlu/pub/PrIM-UPMEM-Tutorial-Analysis-Benchmarking-20min-2021-07-04-talk.pdf`. Accessed: 2024-08-12.

[4] UPMEM SAS, "Upmem coding tips." `https://sdk.upmem.com/2024.1.0/fff_CodingTips.html`. Accessed: 2024-08-16.