IDEAL: Image DEnoising AcceLerator

Mostafa Mahmoud
University of Toronto
mostafa.mahmoud@mail.utoronto.ca

Bojian Zheng
University of Toronto
bojian.zheng@mail.utoronto.ca

Alberto Delmás Lascorz
University of Toronto
a.delmaslascorz@mail.utoronto.ca

Felix Heide
Stanford University/Algolux
fheide@stanford.edu

Jonathan Assouline
Algolux
jonathan.assouline@algolux.com

Paul Boucher
Algolux
paul.boucher@algolux.com

Emmanuel Onzon
Algolux
emmanuel.onzon@algolux.com

Andreas Moshovos
University of Toronto
moshovos@eecg.toronto.edu

ABSTRACT

Computational imaging pipelines (CIPs) convert the raw output of imaging sensors into the high-quality images that are used for further processing. This work studies how Block-Matching and 3D filtering (BM3D), a state-of-the-art denoising algorithm can be implemented to meet the demands of user-interactive (UI) applications. Denoising is the most computationally demanding stage of a CIP taking more than 95% of time on a highly-optimized software implementation [29]. We analyze the performance and energy consumption of optimized software implementations on three commodity platforms and find that their performance is inadequate.

Accordingly, we consider two alternatives: a dedicated accelerator, and running recently proposed Neural Network (NN) based approximations of BM3D [9, 27] on an NN accelerator. We develop *Image DEnoising AcceLerator*(*IDEAL*), a hardware BM3D accelerator which incorporates the following techniques: 1) a novel software-hardware optimization, *Matches Reuse* (MR), that exploits typical image content to reduce the computations needed by BM3D, 2) prefetching and judicious use of on-chip buffering to minimize execution stalls and off-chip bandwidth consumption, 3) a careful arrangement of specialized computing blocks, and 4) data type precision tuning. Over a dataset of images with resolutions ranging from 8 megapixel (MP) and up to 42MP, IDEAL is 11, 352× and 591× faster than high-end general-purpose (CPU) and graphics processor (GPU) software implementations with orders of magnitude better energy efficiency. Even when the NN approximations of BM3D are run on the DaDianNao [14] high-end hardware NN accelerator, IDEAL is 5.4× faster and 3.95× more energy efficient.

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123941

CCS CONCEPTS

• **Computer systems organization → Special purpose systems**; *Neural networks*; *Real-time system architecture*; Single instruction, multiple data;

KEYWORDS

Computational imaging, image denoising, accelerator, neural networks

ACM Reference format:
Mostafa Mahmoud, Bojian Zheng, Alberto Delmás Lascorz, Felix Heide, Jonathan Assouline, Paul Boucher, Emmanuel Onzon, and Andreas Moshovos. 2017. IDEAL: Image DEnoising AcceLerator. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 14 pages.
https://doi.org/10.1145/3123939.3123941

1 INTRODUCTION

Numerous applications, such as those in medical imaging, film production, automotive, and robotics, use imaging sensors (IS) to convert light to signals appropriate for further processing and storage by digital devices such as smartphones, desktop computers, digital cameras, and embedded systems. IS output is far from perfect and requires significant processing in the digital domain to yield acceptable results [39, 44]. For example, lens imperfections result in distorted output, while sensor imperfection such as non-uniform sensitivity may yield output that is underexposed or overexposed at places and thus missing crucial information or that contains other artifacts. Computational Imaging (CI) is the processing in the digital domain of IS output to compensate for these limitations. A Computational Imaging Pipeline (CIP) comprises a sequence of processing steps implementing CI.

What are the likely inputs to a CIP? On the commercial side, photos and videos dominate in the past few years with 1 trillion photos taken in 2015 compared to the 3.8 trillion photos that were taken in all human history until 2011 [32]. By the end of 2017 around 80% of all photos will be taken with a mobile phone, a platform with limited power and computational capabilities [51]. Similar resource limitations apply to many devices such as digital cameras or medical devices. Accordingly, it is desirable to support CI applications on platforms whose cost, power, energy, and potentially form factor are constrained. CI applications are not limited to consumer devices. For example, they are widely used for scientific applications such

as telescope imaging with images of up to 1.5 billion pixels [31, 34]. Thus, there are also applications where higher cost and energy are acceptable for better quality.

An essential step in practically all CI applications is *denoising*, a process that aims to improve the signal to noise ratio in the output image frame. The state-of-the-art denoising algorithm is Block-Matching and 3D filtering (*BM3D*) [17] as it delivers the highest known image quality compared to other techniques [7, 8, 17, 18, 24–26, 28, 29, 33, 43]. However, this superior quality comes at a significant computational power cost. Previous BM3D implementations on high-end general purpose and graphics processors found that it is impractical to use even after intensive software optimization efforts [29]. We corroborated this observation by profiling a typical front-end CIP, the process of converting the raw sensor signal into a typical image representation (e.g., RGB color channel), that was developed by Heide *et al.* [29]. Out of the 184 seconds that are needed to process a raw 2MP image on a high-end CPU, 95% is devoted to denoising. Further, we found that BM3D is compute bound.

User Interactive (UI) applications expect processing to take at most one second if not less [12, 37, 40, 49] while real-time computer vision (CV) applications expect much shorter processing times. Mobile phones and photo cameras incorporate a wide range of image capturing UI applications while *Advanced Driver-Assistance Systems* (ADAS) incorporate camera-driven real-time CV components for scene understanding and segmentation, and multi-object tracking. The camera's resolution dictates the distance at which a given object can be detected while the frame rate dictates system responsiveness for critical functions such as the vehicle's stop distance [38]. While current ADAS use 1MP cameras at 10-15 frames per second (FPS), next generation systems will use 2MP at 30 FPS and soon thereafter 8MP [3, 38]. Such systems require an accelerated, reliable and power efficient CI processing pipeline. Finally, video capturing applications need to denoise raw video frames in real-time before encoding. The denoised frames require substantially less compression and therefore lead to significant bandwidth savings as denoising itself can be seen as a compression mechanism [13]. Accordingly, the goal of this work is to explore how to implement BM3D so that it can be used for UI and real-time applications.

We consider the following options: 1) Optimized software implementations running on commodity hardware, including a high-end and an embedded general purpose processor or a graphics processor, 2) a dedicated hardware accelerator, and 3) running recently proposed NN-based approximations of BM3D on a high-end NN hardware accelerator.

Unfortunately, performance with the software implementations falls far short of the needs of UI applications. Accordingly, we present IDEAL a dedicated hardware BM3D accelerator that allows for the first time a BM3D variant to be used for UI applications. IDEAL incorporates the following techniques: 1) a novel software-hardware optimization, Matches Reuse (MR) that exploits typical image content to reduce the computations needed by BM3D without sacrificing quality, 2) prefetching and judicious use of on chip buffering to minimize execution stalls and off-chip bandwidth consumption, 3) a careful arrangement of specialized computing blocks, and 4) data type precision tuning.

Recent work has showed that it is possible to approximate BM3D using Deep Neural Networks (*DNNs*) [9, 27]. A DNN approach has the advantage of being less rigid whereas a DNN accelerator could be used for other applications as well. Unfortunately, we find that further execution time improvements are needed to meet the requirements of UI applications even when a high-end DNN accelerator is used.

In summary the contributions and findings of this work are:

- We develop and analyze the performance and energy consumption of highly optimized software implementations of BM3D for three commodity platforms: 1) a high-end general purpose processor with vector extensions (CPU), 2) a desktop graphics processor (GPU), and 3) a general purpose processor targeting the embedded systems market. We find that these BM3D implementations fall far short of the requirements of UI applications even for 1080p HD image frames (roughly 2MP resolution).
- We develop IDEAL, a dedicated hardware accelerator for BM3D. IDEAL incorporates a novel software optimization that takes advantage of the common case reducing the amount of computation performed. Experiments using cycle accurate simulation and synthesis results demonstrate that IDEAL makes it possible to process image frames of up to 42MP for UI applications. On average, IDEAL is 11, 352× and 591× faster than the CPU and the GPU implementations respectively. IDEAL is 4 and 3 orders of magnitude more energy efficient respectively as well. We also show that IDEAL can be modified with little effort to support another CI application, sharpening.
- We find that even on an appropriately configured high-end state-of-the-art DNN accelerator [14] the two NN alternatives while far faster than the software implementations still fall short of the UI processing requirements. Specifically, IDEAL is found to be at least 5.4× faster and 3.95× more energy efficient than the best of the two NN alternatives.

While this work focuses on using BM3D for denoising, BM3D has many more applications as it is an instance of the Similarity-Based Collaborative Filtering (*SBCF*) technique, a state-of-the-art filtering technique that can implement a wide variety of CI building blocks. For example, BM3D can be used for image sharpening, deblurring, upsampling, video denoising, and super-resolution by simply changing the filter it implements [16, 17, 19–22, 36]. Investigating the architectural support necessary for BM3D can be thus valuable for other CI building blocks. Moreover, the MR optimization, applied here to BM3D, can be applicable to all SBCF-based algorithms as illustrated in Section 5.1.

The rest of this paper is organized as follows: Section 2 reviews the BM3D denoising algorithm. Section 3 analyzes software implementations of BM3D. Section 4 presents a basic accelerator design IDEAL_{B}. Section 5 further refines the design into IDEAL_{MR} by presenting the MR technique and other optimizations. Section 6 compares the performance, energy, and where appropriate area of the BM3D implementations. Section 7 illustrates how IDEAL can be extended to implement additional functionality via an example. Finally, Section 8 reviews related work and Section 9 concludes.

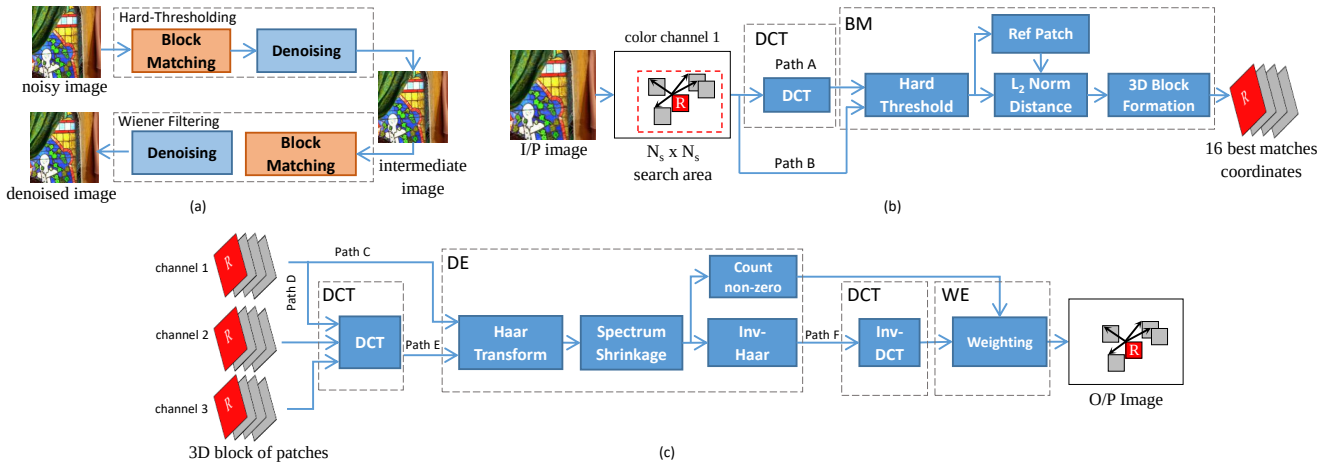


Figure 1: BM3D processing flow: (a) high level stages, (b)block-matching step, (c) denoising step.

2 BM3D DENOISING

BM3D treats the input 3-channel (Red-Green-Blue) image as a grid, with a stride of P_s pixels of potentially overlapping subimages, or *patches*. BM3D transforms the input image as follows: For each patch, BM3D searches through a surrounding window of $N_s \times N_s$ pixels and with a stride of S_s for the best set of P *similar* patches, given some similarity metric. The P matching patches including the reference patch are then *all* transformed and the results affect *all* corresponding output image patches. Accordingly, an output pixel might receive multiple updates from overlapping patches.

As Fig. 1a shows, BM3D denoising processes the input image in two stages: a) Hard-Thresholding, and b) Wiener Filtering. Each stage comprises two steps which are almost identical across the two stages: 1) Block Matching (Fig. 1b), and 2) Denoising (Fig. 1c). This results into four steps: Block Matching #1 (BM1), Denoising #1 (DE1), Block Matching #2 (BM2), and Denoising #2 (DE2). The input is a raw image from an IS. We first describe BM1 and DE1, the steps of the "Hard-Thresholding" stage, followed by the modifications needed by the "Wiener filtering" stage to implement BM2 and DE2. The section then comments on how easy it is to use BM3D for other applications and concludes by detailing the computations performed by the BM3D building blocks.

Block-Matching #1: As Fig. 1b shows, BM1 searches an area of $N_s \times N_s$ pixels centered at the reference patch for similar patches. Patches are typically 4×4 pixels wide. According to Heide *et al.* [29], the following configuration provides the best quality: a search stride S_s and a reference patches stride P_s of 1 and an N_s of 49 (39 for BM2). Block-matching uses only the first channel. Accordingly, for each reference patch, $49 \times 49 = 2401$ patches are processed in BM1.

In detail, BM1 operates as follows: a) For every reference patch P_{ref}, the search area is read patch by patch and transformed through Discrete Cosine Transform (DCT) (Path A in Fig. 1b). The search includes and starts with P_{ref}. b) The DCT patches are hard-thresholded, that is coefficients that are below a preset threshold T_{ht} are eliminated. c) The *l_2-Norm* (Euclidean distance) from P_{ref} is calculated. If the distance is below a certain threshold T_{match}, the patch is reported as a match to next module. d) The *3D Block Formation* step keeps the 16 patches with the least distance from P_{ref} including P_{ref} itself. At the end, a sorted according to distance 3D stack of the closest 16 patches is sent to DE1. The stack contains the patch coordinates only. BM3D uses 16 best matches as this many were found to be sufficient to compensate for additive white Gaussian noise with an up to 75 standard deviation (σ) [18].

Denoising Stage #1: As Fig. 1c shows, this step processes the input patch stack, and updates the corresponding output image patches. The denoising step processes each of the 3 color channels separately. The processing proceeds as follows: a) Saving the DCT of channel 1 patches in BM1 avoids recomputing this for the stack patches in DE1 (Path C in Fig. 1c). Only channel 2 and 3 stack patches are passed through DCT. b) The 3D stack in DCT domain is read in vectors along the depth dimension (z-dimension), as input to the Haar transform module. c) The Spectrum Shrinkage module uses hard-thresholding to eliminate those Haar coefficients that are below a preset threshold T_{hard}. d) The number of non-zero coefficients in the entire 3D block M is counted. e) The inverse Haar and inverse DCT restore the 3D block of patches to the color domain. f) Each restored patch is weighted by $1/M$ before being accumulated to its original location in the output image.

Wiener Filtering Stage: The differences in the Wiener filtering stage are as follows: a) The search window is smaller with N_s set to 39. b) In BM2, searching for the best matches is done in the color domain instead of the DCT domain (Path B in Fig. 1b). c) Since channel 1 patches bypass DCT in BM2, the matching channel 1 patches go through DCT in DE2 (Path D in Fig. 1c) like the other channels. d) Spectrum shrinkage in DE2 implements a "Wiener Filter" that attenuates the Haar coefficients instead of applying a hard threshold.

Other BM3D Applications: The Haar-transformed 3D stack exhibits the sparsity needed for further processing to achieve the targeted effect not only of denoising [17], but also of sharpening [36], deblurring [20] or upsampling [22]. A combined implementation can achieve more than one effect at once [19]. Such effects are achieved by replacing/adding different filters to the DE step with the rest of the pipeline remaining as-is.This class of algorithms has

also been extended beyond the imaging domain to video processing including denoising [16] and super-resolution [21].

2.1 Computational Blocks

The main computational blocks of BM3D are: DCT, Haar transform, l_2-Norm (distance), inverse Haar transform, and inverse DCT.

DCT and Inverse DCT: The DCT consists of a 1D DCT along the rows of the input patch, a transpose of the output followed by one more 1D DCT applied along the rows. The 1D DCT is a matrix multiplication of the patch by a matrix of constant coefficients. For a 4×4 patch, this matrix multiplication involves 64 multiplications and 64 additions. The computation of the DCT of a patch P follows this equation:

$$P_{\text{DCT}} = C(CP)^{\text{T}} \tag{1}$$

Where C is the transform coefficients matrix and $(.)^{\text{T}}$ is the transpose operator. The inverse DCT uses the same computation with the transpose of the transform coefficients matrix.

Haar and Inverse Haar Transforms: These are 1D transforms along the z-dimension of the stacked 16 best matches. The transform is a matrix-vector multiplication of a 16×16 matrix containing constant coefficients by each 16-element vector along the z-dimension. Each such multiplication entails 256 scalar multiplications and 256 scalar additions.

l_2-Norm: The l_2-Norm (distance) between two $M \times M$ patches P_1 and P_2 needs M^2 subtractions, M^2 multiplications and M^2 additions. The computation implements this equation:

$$distance = \sum_{i=0}^{i=M} \sum_{j=0}^{j=M} (P_1(i,j) - P_2(i,j))^2 \tag{2}$$

3 SOFTWARE IMPLEMENTATIONS

This section justifies the need for accelerating BM3D by analyzing the execution time of optimized software implementations on commodity hardware. The analysis shows the bottlenecks that should be targeted by a custom hardware accelerator.

3.1 CPU Implementation

The original pre-compiled reference implementation of BM3D is highly optimized using Intel's Thread Building Blocks and Intel's Math Kernel Library. On average, its Instructions Per Cycle (IPC) throughput is 2.45 on a processor with an ideal IPC of 4. Unfortunately, as the reference implementation is only available in binary form, reverse engineering would have been needed to associate performance events with code and data structures. Thus, we implemented BM3D in C++ exploiting the Intel AVX extended vector instruction set to match or exceed the performance of the reference implementation on a high-end Intel Xeon E5-2650 v2 (Section 6.1 describes our methodology). We also developed a vectorized implementation on the ARM Cortex-A15 mobile processor. Fig. 2 reports the processing time in *seconds* for images of different resolution up to 16MP for the reference ("Orig"), non-vectorized Xeon ("Basic"), vectorized Xeon ("AVX Vect") and ARM ("ARM Vect") implementations. The performance of AVX Vect almost matches that of Orig. The best runtime for moderate resolution images is far beyond the acceptable range; a 16MP image takes 1400 seconds to process on the Xeon, whereas the ARM implementation is 5.2× slower on average. In the interest of space, Fig. 3 (left y-axis) shows the processing time for a wider range of resolutions and only for the vectorized Xeon implementation. In conclusion, performance is far below that needed for UI applications even for low resolution images.

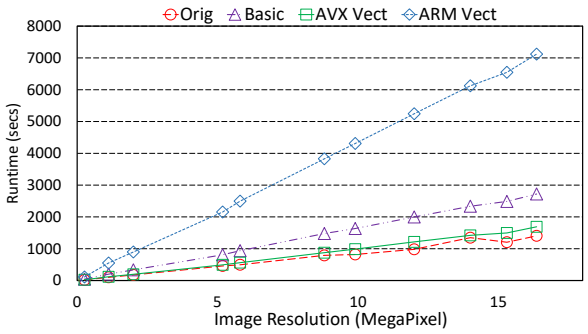


Figure 2: CPU runtime for images up to 16MP.

Table 1: Microarchitectural breakdown of CPU runtime.

Retiring cycles	62.4%
Front-end stalls	4.1%
Mispeculation stalls	5.4%
Back-end (Memory) stalls	5.5%
Back-end (Core) stalls	22.8%

3.1.1 Microarchitectural Analysis. Using Intel's VTune, we analyzed the microarchitectural behavior of our vectorized Xeon implementation [52]. Table 1 shows that instructions are successfully retired in 62.4% of the total processor cycles. Front-end stalls and mispeculation stalls are as low as 4.1% and 5.4% respectively. Back-end stalls contribute 28.3% of the total cycles but only 5.5% of these is due to memory. The 22.8% of core-related stalls suggest that computational resources are a bottleneck, and given that the implementation exhibits a high IPC of 2.7, these measurements demonstrate that BM3D running on commodity CPUs is compute-bound.

3.2 GPU Implementation

We implemented BM3D in CUDA and analyzed its runtime on a high-end NVIDIA GTX 980 with 4GB of memory. The implementation uses an accurate nearest neighbor search for block-matching and divides the image into tiles to fit the intermediate results on the 4GB memory. Tiling was also used to exploit the fast on-chip shared memory. Fig. 3 (right y-axis) shows the runtime for different resolution images. While the GPU implementation is much faster than the CPU one, its performance remains unsatisfactory. For example, processing a 16MP and a 42MP images takes 86 and 226 seconds respectively. Heide *et al.* [29] report that using the approximated-nearest-neighbor (ANN) method proposed by Tsai *et al.* [48] improves GPU performance by 4×. We did not implement this modification as performance would still be unsatisfactory.

3.3 Execution Time Breakdown

Fig. 4 shows a per algorithm step breakdown of the runtime on the Xeon and the GTX 980. Recall that BM3D entails two stages: Hard-threshold filtering followed by Wiener filtering. Each stage consists of the following steps: computing the DCT transformation of all possible patches (DCTx), block-matching (BMx) to find the best 16 matches of each reference patch, and finally the actual denoising (either hard threshold filter or Wiener filter – DEx). As Fig. 4 shows, the block-matching step is the bottleneck as it searches through 49×49 patches (39×39 for the Wiener stage) for every reference patch. This step accounts for 67% of the CPU runtime combined for both stages. On the GPU, the BM step also dominates instead taking 87% of the runtime combined for the two stages. The accurate nearest neighbor search requires some synchronization among GPU threads to exchange information about the best matches found so far. This limits the amount of concurrency that can be extracted on the GPU for these steps.

In order to reduce processing latency to acceptable levels, all steps need to be accelerated in hardware. However, the breakdown above can guide the design process to judiciously partition the power and area budgets among the pipeline stages.

4 IDEAL

We first describe a basic accelerator configuration ($IDEAL_B$) for BM3D denoising, which we later enhance to meet our performance requirement. Fig. 5 shows the main components of IDEAL_B: 1) the

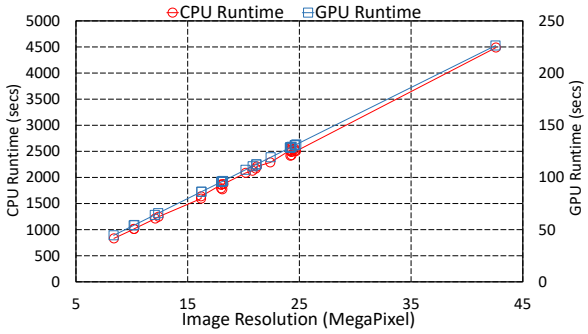


Figure 3: CPU and GPU runtime for images up to 42MP.

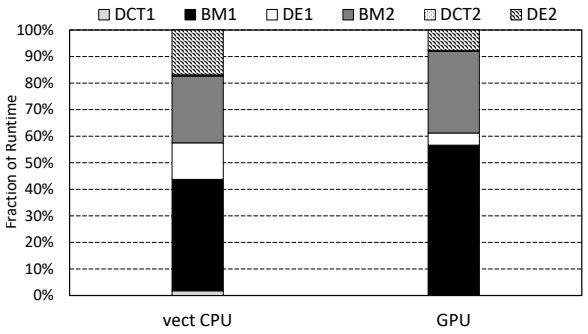


Figure 4: Runtime breakdown for the CPU and GPU implementations.

DCT engine (E_{DCT}), 2) the Patch Buffer (PB), 3) the 16 Block-Matching Engines (E_{BM}s), and 4) the Denoising Engine (E_{DE}). The dashed boxes in Fig. 1b and Fig. 1c provide the mapping of the algorithmic blocks to IDEAL_B components. Instead of directly mirroring the BM3D pipeline in hardware, IDEAL_B uses only as many resources as necessary to keep all units as busy as possible. Since the bulk of the computation happens in the E_{BM}s, a single E_{DCT} and a single E_{DE} are sufficient to sustain the 16 E_{BM}s. The PB stores channel 1 patches in the frequency domain (Path A for BM1) or in the color domain (Path B for BM2) exploiting reuse across adjacent search windows. The PB also feeds the E_{DE} with channel 1 DCT patches (Path C for DE1 and Path D through E_{DCT} for DE2). E_{DCT} performs both DCT and inverse DCT reusing internal resources for both. It accepts jobs from three queues: a) the BM Patch Queue (Q_{BMP}) (BM1, Path A Fig. 1b), b) the Denoiser DCT Queue (Q_D) (DE1 channels 2 and 3, DE2 channels 1 Path D, 2 and 3 Fig. 1c), and c) the Denoiser Inverse DCT Queue(Q_{iD}) (DE1 and DE2, Path F Fig. 1c).

Processing starts by fetching channel 1 patches from memory and onto Q_{BMP}. The patches either go through the E_{DCT} (BM1 Path A) or bypass it (BM2 Path B) before being stored in the PB. The 16 E_{BM}s process concurrently 16 adjacent reference patches. The PB, detailed in Section 4.3, holds all the channel 1 patches of an area covering the search windows for these 16 reference patches. Every cycle, one patch is read out of the PB and is broadcast to all the E_{BM}s. Each E_{BM} calculates the distance between this patch and its assigned reference patch. It builds a list of the 16 closest matching patches. Once the search window is exhausted, each E_{BM} enqueues the coordinates of the 16 best matches into the *Denoising Jobs Queue* (Q_{DJ}) which feeds the E_{DE}. The E_{DE} then processes these jobs one at a time. For each job, the three color channels are processed one at a time. As channel 1 patches are already buffered in the PB, the E_{DE} gets those needed for denoising from there (Path C for DE1 and Path D through Q_D, E_{DCT} then Path E for DE2). For the two other channels, the patches are read off-chip onto Q_D, passed through the E_{DCT} and then follow Path E. E_{DE} stacks the 16 best matching DCT patches, performs a Haar transform along the depth dimension, either hard-thresholds (DE1) the resulting coefficients or attenuates them with a Wiener Filter (DE2), and finally, performs an inverse Haar transform. The resulting patches are then sent back to the E_{DCT} through Q_{iD} to transform them back to the color domain (Path F). At the very end, the patches are weighted and accumulated back to the output image in the main memory. The rest of this section details the individual processing components, illustrates design optimizations and motivates the improved IDEAL design.

Block-matching Engines: Fig. 6 shows the structure of the BM engine (E_{BM}) with its three main components: 1) a buffer to hold the assigned reference patch (RPB) 2) an Euclidean distance engine (EDE), and 3) a priority queue (MQ) to save the coordinates of the 16 best matches found so far sorted according to the distance from the reference patch. Processing starts by reading a reference patch from PB and into RPB. The BM continues by: 1) reading from PB the patches of the corresponding search window one at time, 2) calculating the distance from the RPB patch using the EDE, and if the distance is below a preset threshold T_match, 3) inserting it into MQ. The EDE uses 16 subtractors, 16 multipliers, and a 16-input

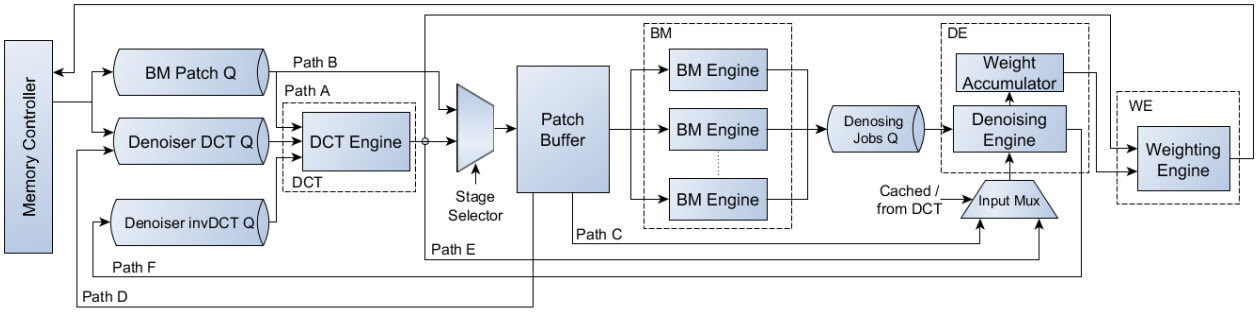


Figure 5: Block diagram of the basic accelerator ($IDEAL_B$).

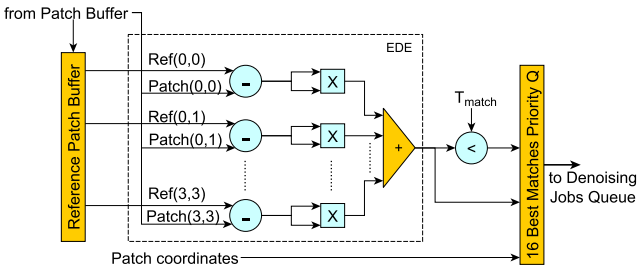


Figure 6: Block-matching (BM) engine.

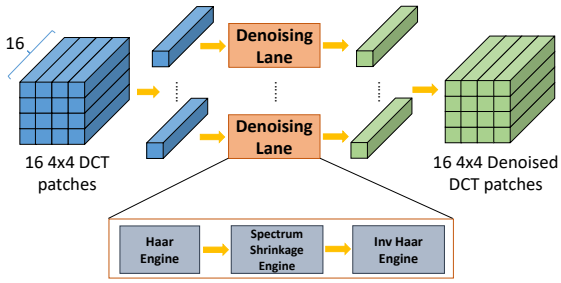


Figure 7: Denoising (DE) engine.

adder tree to calculate the 4×4 patch distance according to Eq. (2) in one cycle.

Denoising Engine: Fig. 7 shows the E_{DE} which processes the 16 best matches in the DCT domain. Those come from the PB directly for DE1 channel 1, from the PB through E_{DCT} for DE2 channel 1, or from off-chip through E_{DCT} otherwise. The 16-patch stack is split into stripes along the the z-dimension. The stripes are assigned each to one of the 16 *Denoising Lanes* (DEL). Each DEL comprises three pipelined stages: Haar Engine (E_H), Spectrum Shrinkage Engine (E_{SS}), and Inverse Haar Engine (E_{IH}). E_H implements the Haar transformation, a matrix-vector multiplication of a 16×16 matrix of constant coefficients and the input stripe. We exploit the constants to reduce resources. As a result, E_H comprises only 32 multipliers, 10 2-input adders, four 4-input adder trees, and four 8-input adder trees. E_{SS} takes the 16-element Haar-transformed stripe and either zeroes those elements that are below a preset threshold (for DE1) or performs element-wise multiplication by Wiener filter coefficients attenuating noisy elements (for DE2). Finally, E_{IH} also implements a matrix-vector multiplication of the transpose of the 16×16 coefficients matrix by the filtered stripe. The sparsity, power-of-2 and repetitions of the transposed matrix elements reduces the number of needed multipliers to 10 along with 16 5-input adder trees.

E_{BM} processes either 49×49 (BM1) or 39×39 (BM2) candidate patches per reference patch. Meanwhile, assuming a 3-channel image and the max number of best matches is 16, E_{DE} processes only $16 \times 3 = 48$ patches. Thus, a single E_{DE} can ideally serve up to $1,521/48 = 31$ E_{BM}s. However, as Section 6.6 explains, we use 16 E_{BM}s and one E_{DE}.

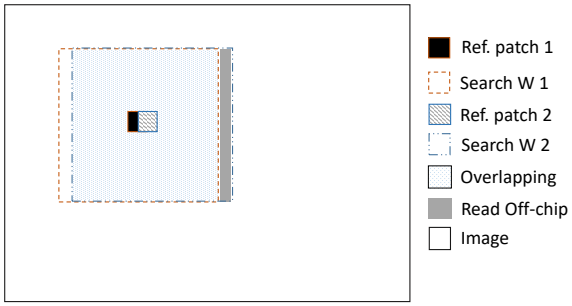


Figure 8: Overlapping search windows for consecutive reference patches.

4.1 Off-chip Bandwidth

Since the reference patches stride P_s is typically smaller than the patch dimension P_D, the search windows of consecutive reference patches are almost completely overlapping as Fig. 8 shows. For example, assuming $P_s = 1$ and $N_s = 49$, out of the 49×49 patches in the search window of reference patch 1, the next search window can reuse 48×49 patches. Buffering these reusable patches in the on-chip PB has the following benefits: 1) Reduces off-chip accesses; Only the gray area in Fig. 8 needs to be read off-chip for each subsequent reference patch. 2) Significantly reduces the pressure on E_{DCT} allowing it to be reused for the three job queues Q_{BMP}, Q_D and Q_{iD}. As a result, the number of pixels read off-chip for each subsequent reference patch is only $(N_s + P_D - 1) \times P_s$.

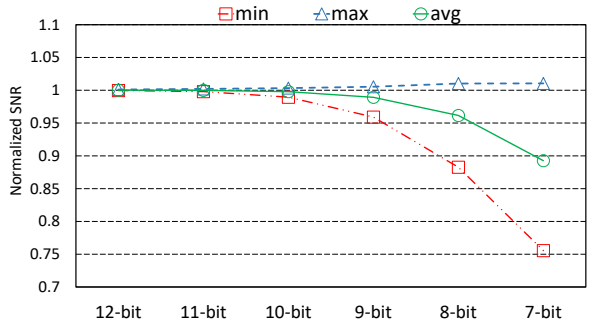


Figure 9: SNR for different fraction precisions normalized to floating-point implementation.

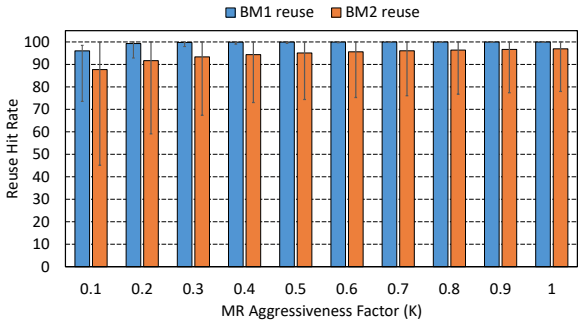


Figure 10: MR hit rate as a function of K.

4.2 Reduced Precision Arithmetic

The original BM3D implementation uses floating-point arithmetic. Reducing precision and using a fixed-point representation are well known techniques with many applications in signal and image processing [11, 41]. Fig. 9 shows how the output image quality changes when using fixed-point with 7 to 12 fractional bits. The figure shows the average, minimum, and maximum signal-to-noise ratio (SNR) normalized over that of the floating-point implementation. The minimum relative SNR with even 10 bits is at least 98.9%. IDEAL uses a 12-bit fractional part with the integer part customized along the pipeline stages to fit the dynamic range of values. Assuming the input image has an 8-bit channel depth, DCT, Haar transform and inverse Haar transform values use 11, 13 and 15 bits for the integer part respectively.

4.3 Patch Buffer Configuration

The PB must serve multiple E_{BM}s concurrently processing adjacent reference patches. The naïve PB implementation would use multiple ports consuming more area and power. As the search windows of adjacent reference patches are almost entirely overlapping, a single-port PB provides adequate performance. Specifically, the buffered patches are read one at a time and broadcast to the E_{BM}s which use or copy the one they need. At times an E_{BM} may need to stall waiting for the next necessary patch. This single-port PB degrades performance by only 12.5% on average compared to the multi-port PB. However, it significantly reduces area and power.

For an $M \times M$ E_{BM} organization, the collective search area comprises $(N_s + (M-1) * P_s)^2$ patches. For the 4×4 E_{BM}s, a 49×49 search window, and $P_s = 1$, a PB of at most 128KB is sufficient assuming 4×4 pixel patches and a 3-byte fixed-point representation of DCT values. The Wiener stage needs less buffering as $N_s = 39$ and patches are buffered in the color domain with 1-byte per pixel.

5 ACCELERATOR OPTIMIZATION

As Section 6.1 demonstrates, while IDEAL_B outperforms the CPU and the GPU by 363× and 18.9× respectively it meets UI application needs for up to 4MP images only. Accordingly, Section 5.1 motivates and presents *Matches Reuse* (MR), a hybrid software/hardware optimization that greatly reduces computation time. Since MR relies on the nature of typical images, Section 5.2 investigates not only its potential performance gains but also its effect on the output quality. Finally, Section 5.3 presents the MR optimized accelerator $IDEAL_{MR}$.

5.1 Matches Reuse Optimization

Since most of the time is spent in BM1 and BM2, MR targets these steps. In BM1 and BM2, a patch matches a reference patch if their l_2-Norm (distance) is below a preset threshold T_{match}. In typical images, adjacent reference patches are likely to be *almost similar* and thus to have almost the same list of 16 best matches. Exploiting this observation, MR significantly reduces the search effort by reusing the best matches of a reference patch for its subsequent reference patch if they are similar enough.

Since reusing the best matches for a subsequent dissimilar reference patch would deteriorate output quality, MR's similarity criterion should be stricter than BM3D's patch similarity criterion. Thus, MR amends the block-matching step as follows: calculate the l_2-Norm (distance) between the current reference patch P_c and the previous reference patch P_p. If the distance is less than a stricter threshold $K \times T_{match}$, where $0 < K < 1$, the 16 best matches for P_c are found by having the BM engine test: 1) The 16 best matches of P_p that also fall within P_c's search window, and 2) the rightmost column of $N_s \times P_s$ patches of P_c's search window, which constitutes the part of P_c's search window that does not overlap with P_p's search window. Thus, BM reduces the number of patches searched from $N_s \times N_s$ to $N_s \times P_s + 16$. This reduces computation by 37× for $N_s = 49$ and $P_s = 1$. If P_c does not strictly match P_p, MR performs the original search.

As K approaches 1, performance should improve at the expense of reduced quality as less similar reference patches reuse the best matches. The speedup depends also on the image content; rapid changes in colors limit the likelihood that successive reference patches are similar enough for best matches reuse. MR can be extended to reuse the matches of earlier reference patches as well. However, checking for reuse with just the immediately preceding patch proved sufficient.

MR can be used in all SBCF algorithms and since they only modify the filters in DE1 and DE2, the block-matching steps are expected to still dominate execution time.

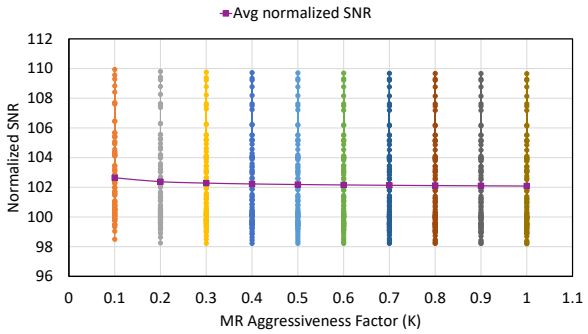


Figure 11: Per image normalized SNR as a function of K.

5.2 Matches Reuse: Potential and Quality

Fig. 10 shows the MR *hit rate*, that is the fraction of reference patches for which MR attempts reuse, as a function of the MR aggressiveness factor K. The figure reports the minimum, maximum, and average hit rates and shows that MR is highly effective at reducing computation even though it depends on image content. A hit rate of X% does not mean that patches are reused at the same rate but that MR restricts its search this often. For BM1, the average hit rate even with $K = 0.1$ is 96% saturating at 99.9% for $K > 0.5$. However, as the minimum hit rate illustrates, MR's effectiveness depends on image content: the minimum hit rate is 74% with $K = 0.1$ and never gets below 99.4% once $K > 0.5$. Such high hit rates should not be surprising as even abrupt changes in an image affect neighboring reference patches gradually because BM3D slides its reference window one pixel at a time. BM2 follows a similar trend as BM1 but the hit rate is generally lower and the sensitivity to image content is higher. This is expected as BM1 and BM2 operate respectively in the DCT and the color domains.

Fig. 11 reports how the output image quality, reported as per image SNR relative to the original BM3D, varies as a function of K. The curve highlights the average relative SNR. For $K = 0.1$, the average SNR is 2.64% higher than the original BM3D, and as K increases, the improvement drops to 2%. Image content also affects the image quality with MR. Images depicting mostly homogeneous areas tend to benefit more with relative SNR improving by up to 10%, while others with less homogeneous areas may see some deterioration which was up to 2% for all K configurations over the image set we studied.

The experiments show that MR may even improve quality over BM3D. This occurs because MR is more resistant to noise artifacts than BM3D, which the latter may consider to be features of the original image. This property of MR is easier to illustrate in the extreme case of a uniform color image. In an example run with such an image, MR improved quality by 49% over BM3D. BM3D tends to *over-fit* the matches to a given reference patch since, due to happenstance, it is often possible to find matches with a nearly identical noise pattern. Effectively, BM3D considers the specific noise pattern an embedded feature of these patches and fails to eliminate it. MR, by reusing the matches of the previous reference patch, is less prone to over-fitting leading to better diversity and more sparsity of the coefficients in the transform domain such that the noise can be more easily identified, isolated and eliminated.

5.3 Architecture Modifications

In IDEAL_{B}, all E_{BM}s operate in lock step across reference patches since the search effort is constant. With MR, each E_{BM} needs to advance independently as each is performing a different number of computations. Accordingly, IDEAL_{MR} comprises 16 processing lanes sharing the same controller. Fig. 12 shows one lane of IDEAL_{MR}. The modifications made are illustrated in the following subsections.

Per-BM Denoising Engine: Since MR significantly prunes the search done by an E_{BM}, the pressure now increases on E_{DE}. We found that using MR increases the average throughput of each E_{BM} to almost that of E_{DE}. Hence, per E_{BM}, IDEAL_{MR} dedicates one E_{DE} and three E_{DCT}s: 1) one for channel 1 DCT for BM1, 2) one performs DCT for channels 2 and 3 for DE1 or channels 1,2 and 3 for DE2, and 3) one for the inverse DCT of all channels after DE.

Per-BM Search Window Buffer: Since the E_{BM}s advance independently, their search windows are likely non-overlapping. Thus, Fig. 12 shows that IDEAL_{MR} uses smaller per-E_{BM} Search Window Buffers (*SWB*) instead of a shared PB. While PB held DCT transformed patches (BM1) or color domain patches (BM2), each SWB holds the search window for the corresponding E_{BM} in the color domain for both BM1 and BM2. Each SWB needs to hold $(N_s+P_D-1)^2$ pixels where P_D is the patch size. This comes at the expense of recalculating the DCT of the patches that are searched for the subsequent reference patch. However, in IDEAL_{MR} 1) there are dedicated E_{DCT}s per E_{BM} making recalculation possible, 2) the MR optimization significantly reduces the number of patches searched per reference patch and the energy overhead of recalculating the DCT for those is negligible compared to the MR energy savings, and 3) the aggregate capacity over SWBs is less than that of PB.

Scheduling: Work is divided among the E_{BM}s at image row granularity. An E_{BM} processes a whole image row and then proceeds with the next available image row if one remains to be processed. This assignment increases the chances of reducing computation through MR, as adjacent reference patches are processed by the same E_{BM}. Furthermore, this allows the buffered search window to be reused across successive reference patches and reduces off-chip bandwidth consumption.

Since off-chip memory accesses return a 64-byte block, IDEAL_{MR} uses SWBs that can each hold up to two search windows to efficiently utilize the off-chip bandwidth. Specifically, each SWB has $(N_s + P - 1)$ *entries*, as many as the rows of the search window. Each entry holds two 64B memory blocks. Using two blocks per SWB entry handles the case where the search window rows do span two memory blocks due to alignment. Moreover, it allows the SWB to effectively *prefetch* the next search windows along the same image row with one access. As a result, IDEAL_{MR}'s performance is within 9.5% of that possible with an ideal single-cycle latency off-chip memory.

Exploiting MR across rows could further reduce the processing time but would also increase the implementation complexity; matches would have to be shared across E_{BM}s. Thus, IDEAL_{MR} does not implement it.

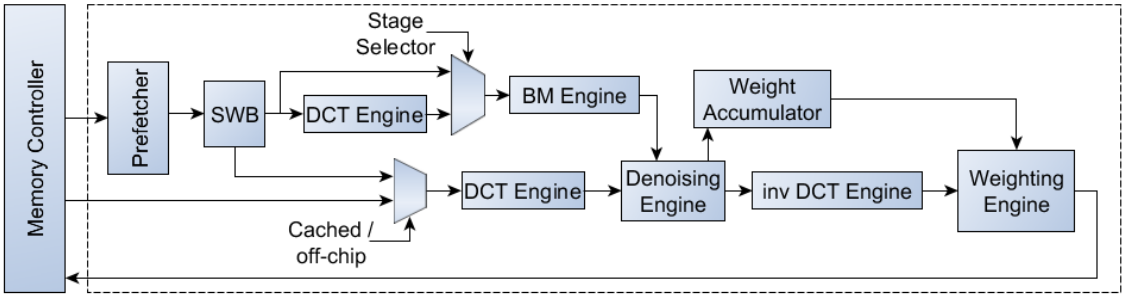


Figure 12: One lane of IDEAL$_{MR}$. The accelerator features 16 lanes sharing the same memory controller.

Table 2: Accelerator hardware parameters.

Parameter	IDEAL$_B$	IDEAL$_{MR}$
Technology	65nm	
Frequency	1 GHz	
BM Engines	16	
Denoising Engines	1 shared	16
DCT Engines	1 shared	16 × 3
On-chip Buffer	126.75 KB	16 × 6.5 KB
Fraction Precision	12-bit	
Memory Controller	2-channel, 32 in-flight requests	
Off-chip DRAM	4GB, DDR3-1333	

Table 3: CPU parameters.

Processor	Intel Xeon E5-2650 v2
Technology	22nm
Frequency	2.60 GHz
Cores	8 (×2 HW threads)
L1, L2, L3	32 KB D + 32 KB I, 256 KB, 20 MB
Memory	4-channel, 48 GB

Table 4: GPU parameters.

GPU	NVIDIA GeForce GTX 980
Technology	28nm
Frequency	1.126 GHz
CUDA Cores	2048
L1/texture, L2	24 KB, 2MB
Shared memory	96 KB
Memory	4 GB GDDR5, 224 GB/s

6 EVALUATION

This section compares and contrasts the performance, power, and where possible the area, of various implementations of BM3D-based denoisers: 1) highly-optimized software implementations of BM3D targeting high-end CPUs and GPUs, 2) two neural network (NN) denoisers accelerated on a high-end NN hardware accelerator, and 3) the basic IDEAL$_B$ and the optimized IDEAL$_{MR}$ accelerators. It also presents a sensitivity study for the IDEAL design choices.

6.1 Methodology

Accelerator Modeling: We developed a cycle-accurate simulator for IDEAL$_B$ and IDEAL$_{MR}$ which was used with the parameters shown in Table 2. The simulator integrates with DRAMSim2 [46] to model the off-chip accesses to main memory. We implemented the accelerators in VHDL and synthesized the designs through the Synopsys Design Compiler using a TSMC 65nm cell library. We use the McPAT [35] version of CACTI to model the area and power consumption of the on-chip buffers as an SRAM compiler is not available to us. The accelerator target frequency is set to 1GHz given CACTI's estimate for buffer speed. To show how IDEAL scales on a newer process technology, we also report the area and power consumption of IDEAL using an STM 28nm cell library. The input data set comprises 30 publicly available RAW format images [5] with resolutions varying from 8MP to 42MP. The images depict nature, street, and texture scenes. We report the results for two MR configurations with K = 0.25 and K = 0.5.

CPU Implementation: We implemented three versions of the BM3D algorithm in C++ targeting general-purpose CPUs: 1) single-thread, 2) multi-threaded to exploit all the hardware threads available on the processor, and 3) MR-optimized single-thread implementation with two configurations K = 0.25 and K = 0.5. The implementations were optimized to exploit the Intel AVX extended vector instruction set, were compiled with GCC 5.1.1 at the -O3 optimization level, and were run on a high-end Intel Xeon E5-2650 v2 detailed in Table 3. For energy and power measurements on these systems, we followed the methodology of Yazdani *et al.* [53]. Specifically, we measured the energy consumption of these runs using the the PAPI API [50] which utilizes the Intel RAPL library [2].

GPU Implementation: We implemented BM3D in CUDA and ran the experiments on an NVIDIA GTX 980 GPU with the specifications shown in Table 4. The application was compiled with the nvcc of the CUDA Toolkit v8.0 at the -O3 optimization level. Performance and power were measured using the NVIDIA Visual Profiler [4] and following the methodology of Yazdani *et al.* [53]. The measurements do not include the memory transfers between the CPU and the GPU (performance is shown to be unsatisfactory even without including this overhead).

Table 5: ML1 & ML2 neural network parameters.

	ML1	ML2
NN type	FCNN	CNN
Number of layers	5	15
Input patch/tile size	39 × 39	320 × 320
Output patch size	17 × 17	256 × 256
Layer dimensions	L1: 1522 × 3072 L2: 3073 × 3072 L3: 3073 × 2559 L4: 2560 × 2047 L5: 2048 × 289	each layer: 64 × 64 kernel: 3 × 3
Model size (# of weights)	27.8M	560K

Table 6: The implementations we compare along with the corresponding abbreviations.

SW/HW	Implementation	Abbreviation
SW	Single-thread CPU	CPU
	Multi-threaded CPU	Threads
	Single-thread CPU + MR, K=0.25	MR (0.25)
	Single-thread CPU + MR, K=0.5	MR (0.5)
	GPU	GPU
HW	Machine Learning 1	ML1
	Machine Learning 2	ML2
	$IDEAL_B$	IDEAL_B
	$IDEAL_{MR}$, K=0.25	IDEAL (0.25)
	$IDEAL_{MR}$, K=0.5	IDEAL (0.5)

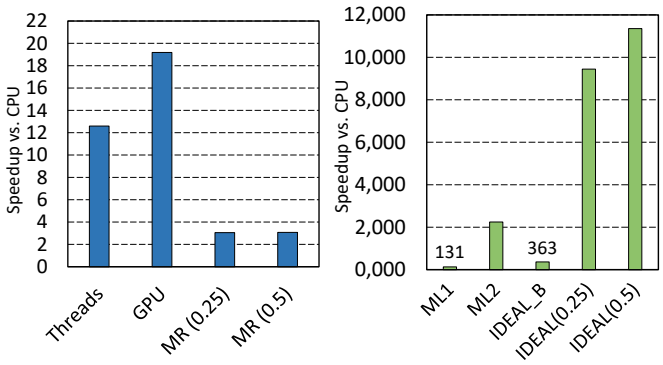
Machine Learning Implementations: Several recent works exploit machine learning (ML) for computational imaging applications [9, 10, 27, 30, 45, 55]. We chose two state-of-the-art ML-based denoisers shown to rival BM3D quality. The denoiser of Burger *et al.* (ML1) employs a 5-layer fully-connected NN (*FCNN*) with the dimensions shown in Table 5 [9]. The denoiser of Gharbi *et al.* (ML2) uses a 15-layer convolutional neural network (*CNN*) to jointly demosaic and denoise an input image [27]. The parameters and architecture of the CNN are shown in Table 5.

We measure the execution time on *DaDianNao*, a state-of-the-art deep neural network accelerator proposed by Chen *et al.* [14]. We developed a custom cycle-accurate simulator for this purpose. To model power consumption, we synthesize DaDianNao on the same 65nm technology as IDEAL and model the on-chip eDRAMs using Destiny [42]. While the ML1 model requires 56MB, we assume it fits on chip as even then the performance remains unsatisfactory. For ML2, a 1.125MB SRAM is sufficient for the weights and replaces the original 32MB eDRAM on-chip Synapse Buffer.

6.2 Execution Time Performance

All measuremnts are normalized to the baseline single-thread CPU implementation and they are averaged over all input images.

Software Implementations: Fig. 13a shows that the multi-threaded CPU and the GPU implementations improve performance by 12.6× and 19× respectively. The figure also shows that the single-thread CPU implementation with the MR optimization results in a 3× speedup for either K value. This is expected as: 1) the block-matching steps consume 67% of the processing time (Fig. 4), and 2) the MR optimization was found to reduce the search effort of the block-matching steps by 29× and 31× on average respectively for the two K values. Integrating MR into the GPU implementation would improve its performance by at most 6.4× given that currently block-matching accounts for 87% of the total execution time. Thus, performance would remain unsatisfactory.



(a) SW implementations. **(b) Accelerators.**

Figure 13: Speedup vs. single-thread CPU.

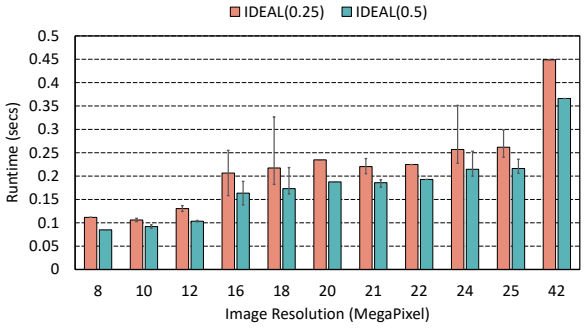


Figure 14: $IDEAL_{MR}$ runtime for different resolution images.

Hardware Implementations: Fig. 13b shows speedup with the hardware-accelerated implementations. Running ML1 and ML2 on DaDianNao is 131× and 2,243× faster respectively. $IDEAL_B$ is on average 363× faster than the baseline CPU and 18.9× faster than the GPU. Although $IDEAL_B$ does not outperform ML2, it consumes much less energy as Section 6.3 shows. $IDEAL_{MR}$ outperforms all other implementations and is 9,446× and 11,352× faster than the baseline CPU for $K = 0.25$ and $K = 0.5$ respectively. Since the accelerator pipelines the BM and the DE steps, the speedup over $IDEAL_B$ scales linearly with the reduction in the BM's search effort. $IDEAL_{MR}$ is 27× and 31× faster than $IDEAL_B$ for the two K configurations respectively.

Per Image Performance: Fig. 14 shows $IDEAL_{MR}$'s runtime for images of different resolution. The runtime depends on image content as this affects the probability of reusing the best matches. For the images studied, processing time remains within UI applications

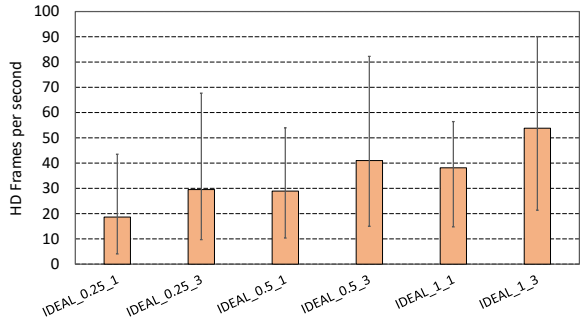


Figure 15: HD frames per second processed by IDEAL$_{MR}$ under different configurations.

Table 7: Power breakdown for all implementations in Watts.

	Core	LLC	DRAM	Total
CPU	25.9	11.9	4.7	42.5
Threads	96.8	24.2	9.1	130.1
GPU	-	-	-	144

	On-Chip		DRAM	Total
	Core	Buffers		
ML1	40.91		NC	NC
ML2	9.04	3.97	0.44	13.45
IDEAL$_B$	1.29	0.39	3.83	5.51
IDEAL$_{MR}$	9.2	2.84	6.16	18.2

limits: IDEAL$_{MR}$ can process a 42MP image in less than 0.5 seconds and the more common 16MP images in 0.13 to 0.18 seconds. Fig. 15 shows the average, minimum and maximum frames per second (FPS) performance of different IDEAL$_{MR}$ configurations for High Definition (HD) frames taken from a different dataset of 34 HD frames depicting nature, city, and texture scenes. We use the abbreviation IDEAL_x_y for IDEAL$_{MR}$ configured with $K = x$ and reference patch stride $P_s = y$. On average, all configurations achieve 30 FPS or higher except for IDEAL_0.25_1. IDEAL$_{MR}$ can reach 90 FPS for the relaxed configuration IDEAL_1_3 with which FPS does not drop below 22. Higher performance would be possible if the search window dimensions could be reduced.

6.3 Power

Table 7 shows the average static and dynamic power dissipation, for the studied implementations. The GPU power measurement tool does not provide a breakdown hence the table lists only the total power. The multi-threaded software implementation dissipates 3× the power compared to the baseline CPU as it uses all the 16 cores. The GPU with full utilization consumes 144W. IDEAL$_B$ is the lowest power-consuming solution at 1.68W on-chip and 5.5W total while IDEAL$_{MR}$ is more power efficient as it consumes 12.05W on-chip and 18.2W in total while being 31× faster than IDEAL$_B$.

ML1 consumes 41W on-chip power while being considerably slower than IDEAL$_{MR}$ and thus we did not measure its off-chip memory power consumption. ML2 consumes 13W on-chip, almost 1W more than the on-chip power of IDEAL$_{MR}$, but performs fewer off-chip accesses since it uses a 4MB on-chip activation memory.

Table 8: The effect of prefetching and on-chip buffering on IDEAL$_{MR}$ speedup over CPU.

Configuration	Pref+Buff	No Pref	None
IDEAL $_{0.25}$	9, 445×	7, 144×	278×
IDEAL $_{0.5}$	11, 352×	8, 176×	286×

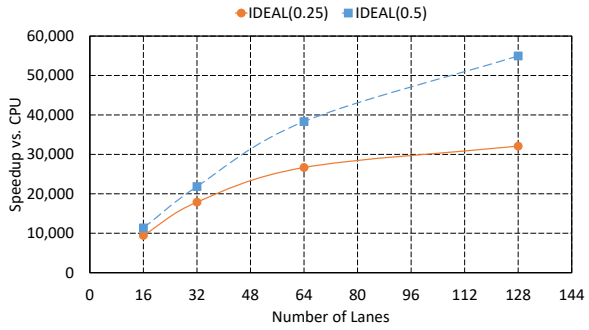


Figure 16: Performance sensitivity to the number of lanes of IDEAL$_{MR}$.

While its total power is lower than IDEAL$_{MR}$, so is its performance. Thus, IDEAL$_{MR}$ is 3.95× more energy efficient than ML2.

Given the speed of IDEAL$_{MR}$, a direct comparison shows that it is 35, 595× and 7, 064× more energy efficient than the CPU and the GPU implementations respectively. However, the CPU and GPU measurements are on actual systems and a direct comparison may not be appropriate. Nevertheless, given the three to four orders of magnitude, there should be no doubt that IDEAL$_{MR}$ is more energy efficient as well.

6.4 Area

IDEAL$_B$ with 16 BM engines, 1 DE engine, 1 DCT engine and a common 126.75KB PB occupies 5.5 mm^2. IDEAL$_{MR}$, with 16 BM engines, 16 DEs, 48 DCT engines and 16 per-BM 6.5KB SWBs requires 23.08mm^2. The DEs are the most expensive components totaling 79% and 62% of the area and power consumption of IDEAL$_{MR}$. The original DaDianNao with 32MB eDRAM running ML1 has an area of 80.4mm^2 while the customized version running ML2 needs 41mm^2.

6.5 Optimization Effect Breakdown

Besides MR, IDEAL$_{MR}$ incorporates prefetching and on-chip buffering. To quantify the effect of these optimizations, Table 8 reports IDEAL$_{MR}$'s performance relative to the baseline CPU when these optimizations are selectively disabled. Three configurations are shown: 1) Prefetching + Buffering, 2) No prefetching, and 3) No prefetching and buffering. Disabling the prefetcher reduces speedups to 7, 144× and 8, 176× respectively for IDEAL $_{0.25}$ and IDEAL $_{0.5}$. Eliminating the on-chip buffers further degrades speedups down to 278× and 286× respectively.

6.6 Sensitivity Study: Scalability

For IDEAL$_B$, the E_{DE} can ideally serve up to 31 E_{BM}s. However, we found that the utilization of each E_{BM} degrades below 90% for

Table 9: Area and power consumption vs. precision.

Precision	12-bit	11-bit	10-bit	9-bit	8-bit
Area (mm^2)	23.08	21.45	19.97	17.54	15.4
Power (W)	12.05	11.65	11.41	10.21	9.07

configurations with more than 16 E_{BM}s. This is due to the single-ported PB that reads and broadcasts one patch at a time to all the E_{BM}s. As the number of E_{BM}s increases, the non-overlapping area of the corresponding search windows increases causing more stalls. Thus, we do not show this study in the interest of space.

Given that most of the time in BM3D is taken by BM1 and BM2, the number of lanes $IDEAL_{MR}$ uses is the key design parameter that directly affects performance. Accordingly, Fig. 16 reports how performance varies relative to the baseline CPU while scaling the number of lanes in $IDEAL_{MR}$ from 16 up to 128. While performance scales linearly going from 16 to 32 lanes, going to 64 lanes or higher the performance improvements become increasingly sublinear and more so for $K = 0.25$. These diminishing performance returns are due to the limited off-chip bandwidth. The evaluated design uses a dual-channel DDR3-1333 memory controller that can deliver up to 21 GB/s. IDEAL $_{0.25}$ hits this bandwidth ceiling earlier at 64 lanes while IDEAL $_{0.5}$ hits it at the 128-lanes configuration. A higher K value results in higher and more regular reuse across lanes, which as a result tend to advance more synchronously. When the lanes stay close to one another, their memory requests often coalesce.

6.7 Sensitivity Study: Technology Node

On a newer STM 28nm cell library $IDEAL_B$ scales well requiring 1.44mm^2 and consuming 0.65W on-chip power. Similarly, $IDEAL_{MR}$ needs an area of 7.9 mm^2 and consumes 5.1W on-chip.

6.8 Sensitivity Study: Precision Tuning

A design parameter that greatly affects area and power is precision, accordingly Table 9 shows how area and power vary for precisions in the range of 12 down to 8 bits. Section 4.2 reported no visual artifacts even with 9 bits of precision. The results show to what extent designers can use precision as a design knob to meet the constraints of different applications.

7 AUGMENTING FUNCTIONALITY

This section presents an example where $IDEAL_{MR}$ was extended to support an additional CI application, sharpening. As expected, extending $IDEAL_{MR}$ to support sharpening required surgical additions only to the DE. By changing the DE, it should be possible to implement other BM3D variants that have demonstrated superior quality such as for example deblurring [20] and upsampling [22].

The modified $IDEAL_{MR}$ implements the technique of Dabov *et al.* [19] to jointly denoise and sharpen images. It uses the same BM3D pipeline with a single minor modification: After denoising the 3D transform-domain coefficients, sharpening is achieved by taking the α-root of their magnitude for some $\alpha > 1$. The modified accelerator incorporates α-rooting components in the DE pipeline after the inverse Haar engine (see Fig. 7). For the 65nm technology these modifications require extra 0.09 mm^2 of area and 0.12W of power. Processing throughput remains unaffected.

8 RELATED WORK

Previous work on accelerating BM3D includes algorithmic approximations as well as implementations targeting different computing platforms such as GPUs, FPGAs and custom hardware. Sarjanoja *et al.* presented a heterogeneous implementation of BM3D using OpenCL and CUDA [47]. On an NVIDIA GeForce GTX 650, their implementation was shown to be 7.5× faster than a CPU implementation. Honzátko's CUDA implementation running on an NVIDIA GeForce GTX 980 was 10× faster than on an Intel Core i7 processor [23]. Both aforementioned implementations restrict BM3D configuration parameters such as the search window size N_s and reference patch stride P_s. Tsai *et al.* accelerated block-matching using an approximated-nearest-neighbor (ANN) search in order to process a 512×512 image in 0.7 seconds on a high-end GPU [48]. Our exact CUDA implementation with the same algorithmic parameters results in a 19× speedup over our CPU implementation which would still be unsatisfactory if incorporated ANN.

Zhang *et al.* proposed a custom hardware accelerator for BM3D which can process 25 '720×576' frames per second of a BT656 PAL 0.4MP video [1, 54]. They restrict the BM3D parameters with $N_s = 15$ and $P_s = 4$ to reduce computations by two orders of magnitude. In our experiments, $IDEAL_{MR}$ achieves 52 FPS for 0.4MP frames even with the unmodified BM3D parameters.

Cardoso proposed an FPGA implementation of BM3D [6] whose pipeline is similar to $IDEAL_B$ having 16 BM modules followed by one DE but that makes several approximations: 1) using l_1-Norm instead of l_2-Norm for the block-matching distance, 2) not implementing Haar transform but just a single-level Haar decomposition and, 3) restricting the search parameters to $N_s = 39$ and $P_s = 4$. On the Xilinx 28nm ZYNQ-7000 ZC706 SoC, the design operates at 125 MHz, consumes 2.9W and processes an 8MP image in 1.5 seconds. Our $IDEAL_B$ would be 21× faster under the same parameters.

Clemons *et al.* proposed a patch memory system tailored to applications that process 2D and 3D data structures such as images [15]. The system exploits multi-dimensional locality and provides efficient caching, prefetching, and address calculation. Leveraging this memory system for $IDEAL_{MR}$ is left for future work.

9 CONCLUSIONS

We developed highly optimized implementations of BM3D for image frame denoising in both software and hardware and analyzed their performance, power, and area. We proposed $IDEAL_{MR}$, a BM3D accelerator that outperforms CPU and GPU implementations by 4 and 3 orders of magnitude while being faster than an accelerated ML alternative by 5.4×. Future work may focus on modifying our accelerators to provide support for additional filters and thus more CI applications or on improving performance and energy efficiency of ML accelerators to enable ML approximations of BM3D to be used for UI applications.

10 ACKNOWLEDGMENTS

This work was supported in part by an NSERC Engage Grant and an NSERC Discovery Grant.

REFERENCES

[1] 2010. BM3D assembly device designed on basis of ASIC. (July 28 2010). http://www.google.us/patents/CN101789043A?cl=en CN Patent App. CN 201,010,102,701.

[2] 2016. Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf. (September 2016).

[3] 2017. Bosch's Driver assistance systems - Predictive pedestrian protection. http://products.bosch-mobility-solutions.com/en/de/_technik/component/SF_PC_DA_Predictive-Pedestrian-Protection_SF_PC_Driver-Assistance-Systems_5251.html?compId=2880. (2017).

[4] 2017. NVIDIA Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler. (2017).

[5] 2017. Photography Blog. (2017). http://www.photographyblog.com

[6] Bernardo Manuel Aguiar Silva Teixeira Cardoso. 2015. *Algorithm and Hardware Design for Image Restoration.* Master's thesis. Faculty of Engineering, the University of Porto, Porto, Portugal. https://repositorio-aberto.up.pt/bitstream/10216/84329/2/35861.pdf

[7] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. 2005. A Non-Local Algorithm for Image Denoising. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 2 (CVPR '05)*. IEEE Computer Society, Washington, DC, USA, 60–65. https://doi.org/10.1109/CVPR.2005.38

[8] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. 2011. Non-Local Means Denoising. *Image Processing On Line* 1 (2011), 208–212. https://doi.org/10.5201/ipol.2011.bcm_nlm

[9] Harold C. Burger, Christian J. Schuler, and Stefan Harmeling. 2012. Image denoising: Can plain neural networks compete with BM3D?. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*. 2392–2399. https://doi.org/10.1109/CVPR.2012.6247952

[10] Harold C. Burger, Christian J. Schuler, and Stefan Harmeling. 2013. Learning how to combine internal and external denoising methods. In *Proceedings of the 35th German Conference on Pattern Recognition (GCPR 2013)*.

[11] Frank Cabello, Julio León, Yuzo Iano, and Rangel Arthur. 2015. Implementation of a fixed-point 2D Gaussian Filter for Image Processing based on FPGA. In *2015 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*. 28–33. https://doi.org/10.1109/SPA.2015.7365108

[12] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. 1991. The Information Visualizer, an Information Workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '91)*. ACM, New York, NY, USA, 181–186. https://doi.org/10.1145/108844.108874

[13] S. Grace Chang, Bin Yu, and Martin Vetterli. 2000. Adaptive Wavelet Thresholding for Image Denoising and Compression. *IEEE TRANSACTIONS ON IMAGE PROCESSING* 9, 9 (2000), 1532–1546.

[14] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 609–622. https://doi.org/10.1109/MICRO.2014.58

[15] Jason Clemons, Chih C. Cheng, Iuri Frosio, Daniel Johnson, and Stephen W. Keckler. 2016. A patch memory system for image processing and computer vision. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783754

[16] Kostadin Dabov, Alessandro Foi, and Karen Egiazarian. 2007. Video denoising by sparse 3D transform-domain collaborative filtering. In *2007 15th European Signal Processing Conference*. 145–149.

[17] Kostadin Dabov, Alessandro Foi, Vladimir Katkovnik, and Karen Egiazarian. 2006. Image denoising with block-matching and 3D filtering. In *Electronic Imaging 2006*. International Society for Optics and Photonics, 606414–606414.

[18] Kostadin Dabov, Alessandro Foi, Vladimir Katkovnik, and Karen Egiazarian. 2007. Image Denoising by Sparse 3-D Transform-Domain Collaborative Filtering. *IEEE Transactions on Image Processing* 16, 8 (Aug 2007), 2080–2095. https://doi.org/10.1109/TIP.2007.901238

[19] Kostadin Dabov, Alessandro Foi, Vladimir Katkovnik, and Karen Egiazarian. 2007. Joint image sharpening and denoising by 3D transform-domain collaborative filtering. In *Proc. 2007 Int. TICSP Workshop Spectral Meth. Multirate Signal Process., SMMSP*, Vol. 2007. Citeseer.

[20] Kostadin Dabov, Alessandro Foi, Vladimir Katkovnik, and Karen Egiazarian. 2008. Image restoration by sparse 3D transform-domain collaborative filtering. In *Electronic Imaging 2008*. International Society for Optics and Photonics, 681207–681207.

[21] Aram Danielyan, Alessandro Foi, Vladimir Katkovnik, and Karen Egiazarian. 2008. Image and video super-resolution via spatially adaptive blockmatching filtering. In *Proceedings of International Workshop on Local and non-Local Approximation in Image Processing (LNLA)*.

[22] Aram Danielyan, Alessandro Foi, Vladimir Katkovnik, and Karen Egiazarian. 2008. Image upsampling via spatially adaptive block-matching filtering. In *2008 16th European Signal Processing Conference*. 1–5.

[23] David Honzátko. 2015. *GPU Acceleration of Advanced Image Denoising*. Ph.D. Dissertation. Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic. https://is.cuni.cz/webapps/zzp/download/130165253/?lang=en

[24] Karen Egiazarian, Jaakko Astola, Mika Helsingius, and Pauli Kuosmanen. 1999. Adaptive denoising and lossy compression of images in transform domain. *Journal of Electronic Imaging* 8, 3 (1999), 233–245. https://doi.org/10.1117/1.482673

[25] Michael Elad and Michal Aharon. 2006. Image Denoising Via Sparse and Redundant Representations Over Learned Dictionaries. *IEEE Transactions on Image Processing* 15, 12 (Dec 2006), 3736–3745. https://doi.org/10.1109/TIP.2006.881969

[26] Alessandro Foi, Vladimir Katkovnik, and Karen Egiazarian. 2007. Pointwise Shape-Adaptive DCT for High-Quality Denoising and Deblocking of Grayscale and Color Images. *IEEE Transactions on Image Processing* 16, 5 (May 2007), 1395–1411. https://doi.org/10.1109/TIP.2007.891788

[27] Michaël Gharbi, Gaurav Chaurasia, Sylvain Paris, and Frédo Durand. 2016. Deep Joint Demosaicking and Denoising. *ACM Trans. Graph.* 35, 6, Article 191 (Nov. 2016), 12 pages. https://doi.org/10.1145/2980179.2982399

[28] Jose A. Guerrero-Colon and Javier Portilla. 2005. Two-level adaptive denoising using Gaussian scale mixtures in overcomplete oriented pyramids. In *IEEE International Conference on Image Processing 2005*, Vol. 1. I–105–8. https://doi.org/10.1109/ICIP.2005.1529698

[29] Felix Heide, Markus Steinberger, Yun-Ta Tsai, Mushfiqur Rouf, Dawid Pająk, Dikpal Reddy, Orazio Gallo, Jing Liu abd Wolfgang Heidrich, Karen Egiazarian, Jan Kautz, and Kari Pulli. 2014. FlexISP: A Flexible Camera Image Processing Framework. *ACM Transactions on Graphics (Proceedings SIGGRAPH Asia 2014)* 33, 6 (December 2014).

[30] Viren Jain and Sebastian Seung. 2009. Natural Image Denoising with Convolutional Networks. In *Advances in Neural Information Processing Systems 21*, D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou (Eds.). Curran Associates, Inc., 769–776. http://papers.nips.cc/paper/3506-natural-image-denoising-with-convolutional-networks.pdf

[31] Lynn Jenner. 2015. Hubble's High-Definition Panoramic View of the Andromeda Galaxy. https://www.nasa.gov/content/goddard/hubble-s-high-definition-panoramic-view-of-the-andromeda-galaxy. (Jan. 5 2015).

[32] Gerald C. Kane and Alexandra Pear. 2016. The Rise of Visual Content Online. http://sloanreview.mit.edu/article/the-rise-of-visual-content-online/. (Jan. 4 2016).

[33] Charles Kervrann and Jérôme Boulanger. 2006. Optimal Spatial Adaptation for Patch-Based Image Denoising. *IEEE Transactions on Image Processing* 15, 10 (Oct 2006), 2866–2878. https://doi.org/10.1109/TIP.2006.877529

[34] John E. Krist. 1992. Deconvolution of hubble space telescope images using simulated point spread functions. In *Astronomical Data Analysis Software and Systems I*, Vol. 25. 226.

[35] Sheng Li, Jung H. Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 469–480.

[36] Markku Mäkitalo and Alessandro Foi. 2011. Spatially adaptive alpha-rooting in BM3D sharpening. In *Image Processing: Algorithms and Systems IX, San Francisco, California, USA, January 24-25, 2011*. 787012. https://doi.org/10.1117/12.872606

[37] Robert B. Miller. 1968. Response Time in Man-computer Conversational Transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I (AFIPS '68 (Fall, part I))*. ACM, New York, NY, USA, 267–277. https://doi.org/10.1145/1476589.1476628

[38] Mihir Mody. 2016. ADAS Front Camera: Demystifying Resolution and Frame-Rate. http://www.eetimes.com/author.asp?section_id=36&doc_id=1329109. (March 7 2016).

[39] Junichi Nakamura. 2005. *Image Sensors and Signal Processing for Digital Still Cameras*. CRC Press, Inc., Boca Raton, FL, USA.

[40] Jakob Nielsen. 2009. Powers of 10: Time Scales in User Experience. https://www.nngroup.com/articles/powers-of-10-time-scales-in-ux/. (Oct. 5 2009).

[41] Wayne T. Padgett and David V. Anderson. 2009. *Fixed-Point Signal Processing*. Morgan & Claypool. https://books.google.ca/books?id=h590cd_BagMC

[42] Matt Poremba, Sparsh Mittal, Dong Li, Jeffrey S. Vetter, and Yuan Xie. 2015. DESTINY: A tool for modeling emerging 3D NVM and eDRAM caches. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1543–1546. https://doi.org/10.7873/DATE.2015.0733

[43] Javier Portilla, Vasily Strela, Martin J. Wainwright, and Eero P. Simoncelli. 2003. Image denoising using scale mixtures of Gaussians in the wavelet domain. *IEEE Transactions on Image Processing* 12, 11 (Nov 2003), 1338–1351. https://doi.org/10.1109/TIP.2003.818640

[44] Rajeev Ramanath, Wesley E. Snyder, Youngjun Yoo, and Mark S. Drew. 2005. Color image processing pipeline. *IEEE Signal Processing Magazine* 22, 1 (Jan 2005), 34–43. https://doi.org/10.1109/MSP.2005.1407713

[45] Marc'Aurelio Ranzato, Y-lan Boureau, Sumit Chopra, and Yann Lecun. 2007. A Unified Energy-Based Framework for Unsupervised Learning. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics (AISTATS-07)*, Marina Meila and Xiaotong Shen (Eds.), Vol. 2. Journal of Machine Learning Research - Proceedings Track, 371–379. http://jmlr.csail.mit.edu/proceedings/papers/v2/ranzato07a/ranzato07a.pdf

[46] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10, 1 (Jan 2011), 16–19.

[47] Sampsa Sarjanoja, Jani Boutellier, and Jari Hannuksela. 2015. BM3D image denoising using heterogeneous computing platforms. In *2015 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 1–8. https://doi.org/10.1109/DASIP.2015.7367257

[48] Yun-Ta Tsai, Markus Steinberger, Dawid Pająk, and Kari Pulli. 2014. Fast ANN for High-quality Collaborative Filtering. In *Proceedings of High Performance Graphics (HPG '14)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 61–70. http://dl.acm.org/citation.cfm?id=2980009.2980016

[49] Gerd Waloszek and Ulrich Kreichgauer. 2009. User-Centered Evaluation of the Responsiveness of Applications. In *Proceedings of the 12th IFIP TC 13 International Conference on Human-Computer Interaction: Part I (INTERACT '09)*. Springer-Verlag, Berlin, Heidelberg, 239–242. https://doi.org/10.1007/978-3-642-03655-2_29

[50] Vincent M. Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. 2012. Measuring Energy and Power with PAPI. In *2012 41st International Conference on Parallel Processing Workshops*. 262–268. https://doi.org/10.1109/ICPPW.2012.39

[51] Paul Worthington. 2014. One Trillion Photos in 2015. http://mylio.com/true-stories/tech-today/one-trillion-photos-in-2015-2. (Dec. 11 2014).

[52] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44. https://doi.org/10.1109/ISPASS.2014.6844459

[53] Reza Yazdani, Albert Segura, Jose-Maria Arnau, and Antonio Gonzalez. 2016. An ultra low-power hardware accelerator for automatic speech recognition. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783750

[54] Hao Zhang, Wenjiang Liu, Ruolin Wang, Tao Liu, and Mengtian Rong. 2016. Hardware architecture design of block-matching and 3D-filtering denoising algorithm. *Journal of Shanghai Jiaotong University (Science)* 21, 2 (2016), 173–183. https://doi.org/10.1007/s12204-016-1709-0

[55] S. Zhang and E. Salari. 2005. Image denoising using a neural network based non-linear filter in wavelet domain. In *Proceedings. (ICASSP '05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, Vol. 2. ii/989–ii/992 Vol. 2. https://doi.org/10.1109/ICASSP.2005.1415573