2D operators on topographic and non-topographic architectures - implementation, efficiency analysis, and architecture selection methodology

Ákos Zarándy, Csaba Rekeczky

Computer and Automation Research Institute, Hungarian Academy of Sciences,

zarandy@sztaki.hu

Eutecus Inc Berkeley, California

zarandy@eutecus.com, rcsaba@eutecus.com

SUMMARY

Topographic and non-topographic image processing architectures and chips, developed within the CNN community recently, are analysed and compared. It is achieved on a way that the 2D operators are collected to classes according to their implementation methods on the different architectures, and the main implementation parameters of the different operator classes are compared. Based on the results, an efficient architecture selection methodology is formalized.

1 INTRODUCTION

Cellular Neural/non-linear Networks (CNN) were invented in 1988 [1]. This new field attracted well beyond one hundred researchers in the next two decades, called nowadays the CNN community. They focused on three main areas: the theory, the implementation issues, and the application possibilities. In the implementation area, the first 10 years yielded more than a dozen CNN chips made by only a few designers. Some of them followed the original CNN architecture [6], others made slight modifications, such as the full signal range model [10] [11], or discrete time CNN (DTCNN) [7], or skipped the dynamics, and made dense threshold logic in the black-and-white domain [8] only. All of these chips had cellular architecture, and implemented the programmable A and/or the B template matrices of the CNN Universal Machine [3] [26]

In the second decade, this community slightly shifted the focus of chip implementation. Rather than implementing classic CNN chips with A and B template matrices, the new target became the efficient implementation of neighbourhoods processing. Some of these architectures were topographic with different pixel/processor ratios, others were non-topographic. Some implementations used analogue processors and memories, others digital ones. Certainly, the different architectures have different advantages and drawbacks. One of the goals is to compare these architectures and the actual chip implementations themselves. This attempt is not trivial, because their parameter gamut and operation modes are rather different. To solve this problem, we have categorized the most important 2D wave type operations and examined their implementation methods and efficiency on these architectures.

In this study, we have compared the following five architectures, of which the first one is used as the reference of comparison.

1. DSP-memory architecture (in particular DaVinci processors from TI [18])
2. Pipe-line architecture (CASTLE [24][23], Falcon [16], C-MVA [21])
3. Coarse-grain cellular parallel architecture (Xenon [25]);
4. Fine-grain fully parallel cellular architecture with discrete time processing (SCAMP [15], Q-Eye [19]);
5. Fine-grain fully parallel cellular architecture with continuous time processing (ACE-16k [9], ACLA [12][13]).

As we will see, one of the key questions here is speed versus the size of the processed image. In the classic DSP-memory architectures (1) the image size to be processed is practically unlimited, however, the frame rate falls drastically with increasing image size. In a typical image processing application (such as a video image analytics in security applications), a QCIF sized image (176×144) can be processed at video rate.

Based on the result of this analysis, we have calculated the major implementation parameters of the different operation classes for every architectures. These parameters are the maximal resolution, frame-rate, pixel clock, and computational demand, the minimal latency, and the flow-chart topology. Having these constraints, an efficient architecture can be selected to a given algorithm.

The paper starts with the brief description of the different architectures (Section 2), which is followed by the categorization of the 2D operators and their implementation methods on them (Section 3). Then the major parameters of the implementations are compared (Section 4). Finally, in Section 5 an efficient architecture selection method is introduced.

2 ARCHITECTURE DESCRIPTIONS

In this section, we describe the architectures examined using the basic spatial grayscale and binary functions (convolution, erosion) of non-propagating type.

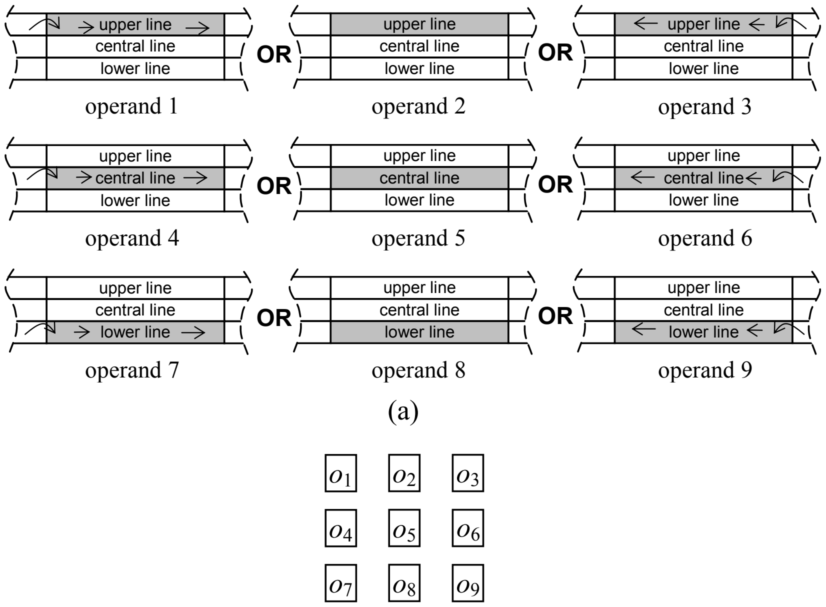
2.1 Classic DSP-memory architecture

Here we assume 32 bit DSP architecture with cache memory large enough to store the required number of images and the program internally. In this way, we have to practically estimate/measure the required DSP operations. Most of the modern DSPs have numerous MACs and ALUs. To avoid comparing these DSP architectures, which would lead too far from our original topic, we use the DaVinci video processing DSP by Texas Instrument, as a reference.

We use 3×3 convolution as a measure of grayscale performance. The data requirements of the calculation are 19 bytes (9 pixels, 9 kernel values, result), however, many of these data can be stored in registers, hence, only an average of a four-data accesses (3 inputs, because the 6 other ones have already been accessed for the previous pixel position, and one output) is needed for each convolution. From a computational point of view, it needs 9 multiple-add (MAC) operations. It is very typical that the 32 bit MACs in a DSP can be split into four 8 bit MACs, and other auxiliary ALUs help loading the data to the registers in time. Measurement shows that, for example, the Texas DaVinci family with the TMS320C64x core needs only about 1.5 clock cycles to complete a 3×3 convolution.

The operands of the binary operations are stored in 1 bit/pixel format, which means that each 32bit word represents a 32×1 segment of an image. Since the DSP's ALU is a 32 bit long unit, it can handle 32 binary pixels in a single clock cycle. As an example, we examine how a 3×3 square shaped erosion operation is executed. In this case erosion is a nine input OR operation where the inputs are the binary pixels values within the 3×3 neighborhood. Since the ALU of the DSP does not contain 9 input OR gate, it is executed sequentially on an entire 32×1 segment of the image. The algorithm is simple: the DSP has to prepare the 9 different operands, and apply bit-wise OR operations on them.

Figure 1 shows the generation method of the first three operands. In the figure a 32×3 segment of a binary image is shown (9 times), as it is represented in the DSP memory. Some fractions of horizontal neighboring segments are also shown. The first operand can be calculated by shifting the upper line with one bit position to the left and filling in the empty MSB with the LSB of the word from its right neighbor. The second operand is the un-shifted upper line. The position and the preparation of the remaining operands are also shown in Figure ***1***a.



(a)

(b)

$$e_1=o_1 \text{ OR } o_2 \text{ OR } o_3 \text{ OR } o_4 \text{ OR } o_5 \text{ OR } o_6 \text{ OR } o_7 \text{ OR } o_8 \text{ OR } o_9$$

(c)

Figure 1. Illustration of the binary erosion operation on a DSP. (a) shows the 9 pieces of 32×1 segments of the image (operands), as the DSP uses them. The operands are the shaded segments. The arrows indicate shifting of the segments. To make it clearer, consider a 3×3 neighborhood as it is shown in (b). For one pixel, the form of the erosion calculation is shown in (c). $o_{1,}$ $o_{2, \dots}$ o_9 are the operands. The DSP does the same, but on 32 pixels parallel.

This means that we have to apply 10 memory accesses, 6 shifts, 6 replacements, and 8 OR operations to execute a binary morphological operation for 32 pixels. Due to the multiple cores and the internal parallelism, the Texas DaVinci spends 0.5 clock cycles with the calculation of one pixel.

In the low power low cost embedded DSP technology the trend is to further increase the clock frequency, but most probably, not higher than 1 GHz, otherwise, the power budget cannot be kept. Moreover, the drawback of these DSPs is that their cache memory is too small, which cannot be increased significantly without significant cost increase. The only way to significantly increase the speed is to implement a larger number of processors, however, that requires a new way of algorithmic thinking, and software tools.

The DSP-memory architecture is the most versatile from the point of views of both in functionality and programmability. It is easy to program, and there is no limit on the size of the processed images, though it is important to mention that in case of an operation is executed on an image stored in the external memory, its execution time is increasing roughly with an order of magnitude. Though the DSP-memory architecture is considered to be very slow, as it is shown later, it outperforms even the processor arrays in some operations. In QVGA frame size, it can solve quite complex tasks, such as video analytics in security applications on video rate [20]. Its power consumption is in the 1-3W range. Relatively small systems can be built by using this architecture. The typical chip count is around 16 (DSP, memory, flash, clock, glue logic, sensor, 3 near sensor components, 3 communication components, 4 power components), while this can be reduced to the half in a very basic system configuration.

2.2 Pipe-line architectures

The basic idea of this pipe-line architecture is to process the images line-by-line, and to minimize both the internal memory capacity and the external IO requirements. Most of the early image processing operations are based on 3×3 neighborhood processing, hence 9 image data are needed to calculate each new pixel value. However, these 9 data would require very high data throughput from the device. As we will see, this requirement can be significantly reduced by applying a smart feeder arrangement.

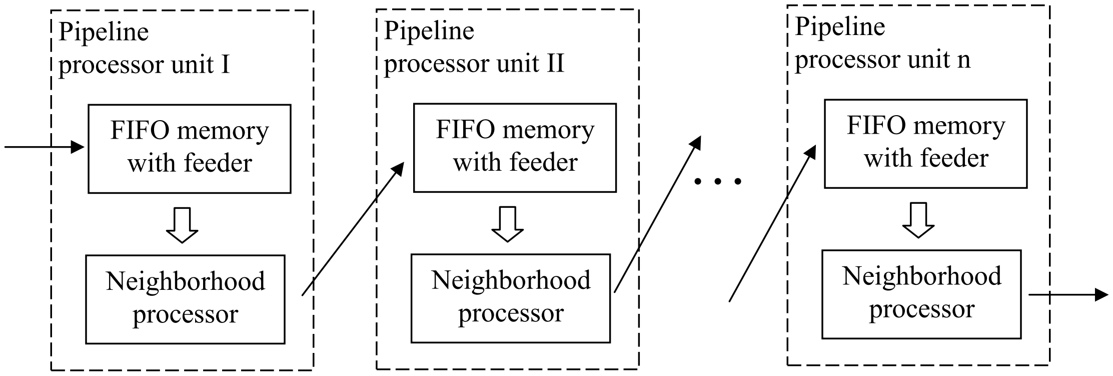
Figure 2 shows the pipe-line processor architecture. It is built up from a chain of processor units. Each unit contains two parts, the memory (feeder) and the neighborhood processor. Both the feeder and the neighborhood processor can be configured 8 or 1 bit/pixel wide, depending on whether the unit is used for grayscale or binary image processing. The feeder contains, typically, two consecutive whole rows and a row fraction of the image. Moreover, it optionally contains two more rows of the mask image, depending on the input requirements of the implemented neighborhood operator. In each pixel clock period, the feeder provides 9 pixel values for the neighborhood processor and the mask value optionally if the operation requires it. The neighborhood processor can perform convolution, rank order filtering, or other linear or nonlinear spatial filtering on the image segment in each pixel clock period. Some of these operators (e.g., *hole finder*, or a *CNN* emulation with A and B templates) require two input images. The second input image is stored in the mask. The outputs of the unit are the resulting and, optionally, the input and the mask images. Note that the unit receives and releases synchronized pixels flows sequentially. This enables to cascade multiple pieces of the described units. The cascaded units form a chain. In such a chain, only the first and the last units require external data communications, the rest of them receives data from the previous member of the chain and releases the output towards the next one.

An advantageous implementation of the row storage is the application of FIFO memories, where the first three positions are tapped to be able to provide input data for the neighborhood processor. The last positions of rows are connected to the first position of the next row (Figure 2b). In this way, pixels in the upper rows are automatically marching down to the lower rows.

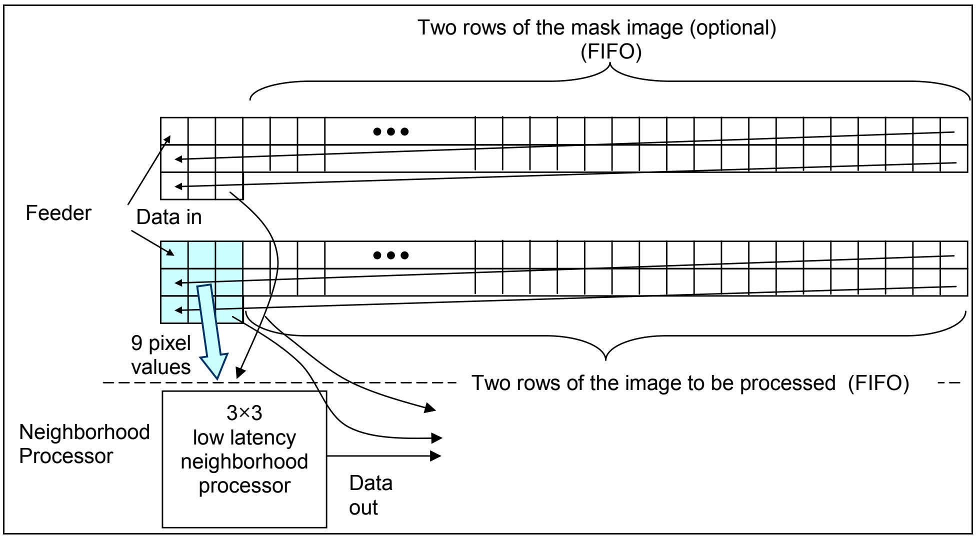
The neighborhood processor is of special purpose, which can implement one or a few different kinds of operators with various attributes and parameter. They can implement convolution, rank-order filters, grayscale or binary morphological operations, or any local image processing functions (e.g. Harris corner detection, Laplace operator, gradient calculation, etc,). In architectures CASTLE [24][23] and Falcon [16], e.g., the processors are dedicated to convolution processing where the template values are the attributes. The pixel clock is matched with that of the applied sensor. In case of a 1 megapixel frame at video rate (30 FPS), the pixel clock is about 30 MHz (depending on the readout protocol). This means that all parts of the unit should be able to operate at least at this clock frequency. In some cases the neighborhood processor operates on an integer multiplication of this frequency, because it might need multiple clock cycles to complete a complex calculation, such as a 3×3 convolution. Considering ASIC or FPGA implementations, clock frequency between 100-300 MHz is a feasible target for the neighborhood processors within tolerable power budget.

The multi-core pipe-line architecture is built up from a sequence of such processors. The processor arrangement follows the flow-chart of the algorithm. In case of multiple iterations of the same operation, we need to apply as many processor kernels, as many iterations we need. This easily ends up requiring a few dozens of kernels. Fortunately, these kernels, especially in the black-and-white domain, are relatively inexpensive, either on silicon, or in FPGA.

Depending on the application, the data-flow may contain either sequential segments or parallel branches. It is important to emphasize, however, that the frame scanning direction cannot be changed, unless the whole frame is buffered, which can be done in external memory only. This introduces a relatively long (dozens of millisecond) additional latency.



(a)


Two rows of the mask image (optional)
(FIFO)
Feeder
Data in
9 pixel values
Two rows of the image to be processed (FIFO)
Neighborhood Processor
3×3 low latency neighborhood processor
Data out

(b)

Figure 2. Pipe-line architecture. The processor chain (a), and one single pipe-line processor unit (b). The single arrows show the single pixel trains, while the block arrows shows the parallel pixel channels.

For capability analysis, here we use the Spartan 3ADSP FPGA (XC3SD3400A) from Xilinx as a reference, because this low-cost, medium performance FPGA was designed especially for embedded image processing. It is possible to implement roughly 120 grayscale processors within this chip, as long as the image row length is below 512, or 60 processors, when the row length is between 512 and 1024.

2.3 Coarse-grain cellular parallel architectures

The coarse-grain architecture is a locally interconnected 2D cellular processor arrangement, as opposed to the pipe-line one. A specific feature of the coarse-grain parallel architectures is that each processor cell is topographically assigned to a number of pixels (e.g., an 8×8 segment of the image), rather than to a single pixel only. Each cell contains a processor and some memory, which is large enough to store few bytes for each pixel of the allocated image segment. Exploiting the advantage of the topographic arrangement, the cells can be equipped with photo sensors enabling to implement a single chip sensor-processor device. However, to make this sensor sensitive enough, which is the key in high frame-rate applications, and to keep the pixel density of the array high, at the same time, certain vertical integration techniques are needed for photosensor integration.

In the coarse-grain architectures, each processor serves a larger number of pixels, hence we have to use more powerful processors, than in the one-pixel per processor architectures. Moreover, the processors have to switch between serving pixels frequently, hence more flexibility is needed that an analog processor can provide. Therefore, it is more advantageous to implement 8 bit digital processors, while the analog approach is more natural in the one pixel per processor (fine-grain) architectures. (See the next subsection.)

As an example for the coarse-grain architecture, we briefly describe the Xenon chip [25]. As can be seen in Figure 3, Xenon chip [25] is constructed of an 8×8, locally interconnected cell arrangement. Each cell contains a sub-array of 8×8 photosensors; an analog multiplexer; an 8 bit AD converter; an 8 bit processor with 512 bytes of memory; and a communication unit of local and global connections. The processor can handle images in 1, 8, and 16 bit/pixel representations, however, it is optimized for 1 and 8 bit/pixel operations. Each processor can execute addition, subtraction, multiplication, multiply-add operations, comparison, in a single clock cycle on 8 bit/pixel data. It can also perform 8 logic operations on 1 bit/pixel data in packed-operation mode in a single cycle. Therefore, in binary mode, one line of the 8×8 sub-array is processed jointly, similarly to the way we have seen in the DSP. However, the Xenon chip supports the data shifting and swapping from hardware, which means that the operation sequence, what we have seen in Figure 1 takes 9 clock cycles only. (The swapping and the accessing the memory of the neighbors do not need extra clock cycles.) Besides, the local processor core functions, Xenon can also perform a global OR function. The processors in the array are driven in a single instruction multiple data (SIMD) mode.

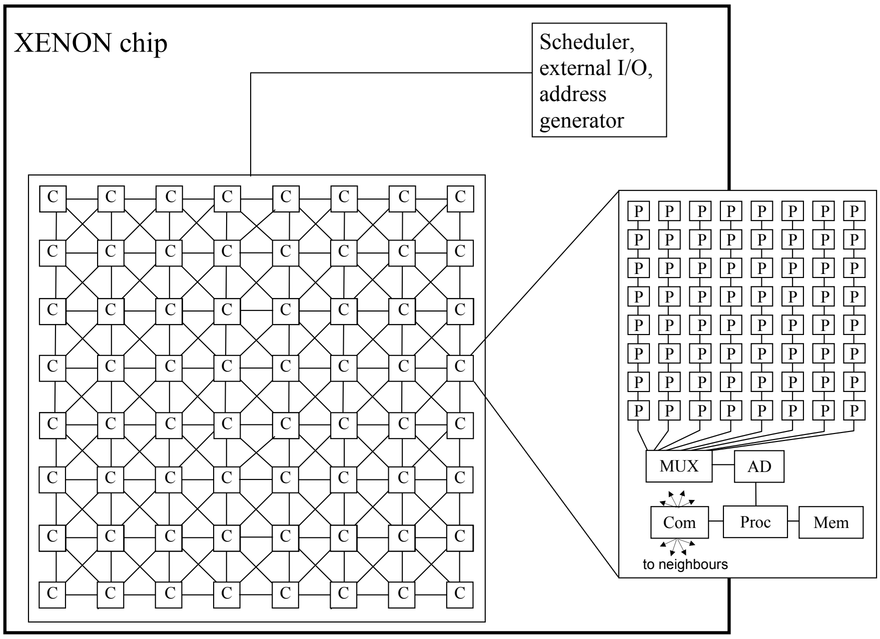


Figure 3. Xenon is a 64 core coarse-grain cellular parallel architecture (C stands for processor cores, while P represents pixels).

Xenon is implemented on a 5x5mm silicon die with 0.18 micron technology. The clock cycle can go up to 100MHz. The layout is synthesized, hence the resulting 75micron equivalent pitch is far from being optimal. It is estimated that through aggressive optimization it could be reduced to 40 micron (assuming a bump bonded sensor layer), which would almost double the resolution achievable on the same silicon area. The power consumption of the existing implementation is under 20mW.

2.4 Fine-grain fully parallel cellular architectures with discrete time processing

The fine-grain, fully parallel architectures are based on rectangular processor grid arrangements where the 2D data (images) are topographically assigned to the processors. The key feature here is that for the fine-grain arrangement there is a one-to-one correspondence between the pixels and the processors. This certainly means that at the same time the composing processors must be simpler and less powerful, than in the previous, coarse-grain case. Therefore, fully parallel architectures are typically implemented in analog domain, though bit-sliced digital approach is also feasible.

In the discussed cases, the discrete time processing type fully parallel architectures are equipped with a general purpose, analog processor, and an optical sensor in each cell. These sensor-processors can handle two types of data (image) representations: grayscale and binary. The instruction set of these processors include addition, subtraction, scaling (with a few discrete factors only), comparison, thresholding, and logic operations. Since it is a discrete time architecture, the processing is clocked. Each operation takes 1-4 clock cycles. The individual cells can be masked. Basic spatial operations, such as convolution, median filtering, or erosion, can be put together as sequences of these elementary processor operations. In this way the clock cycle counts of a convolution, a rank order filtering, or a morphologic filter are between 20 and 40 depending on the number of weighting coefficients.

It is important to note that in case of the discrete time architectures (both coarse- and fine-grain), the operation set is more elementary (lower level) than on the continuous time cores (see the next section). While in the continuous time case (CNN like processors) the elementary operations are templates (convolution, or feedback convolution) [2][3], in the discrete time case, the processing elements can be viewed as RISC (reduced instruction set) processor cores with addition, subtraction, scaling, shift, comparison, and logic operations. When a full convolution is to be executed, the continuous time architectures are more efficient. In the case of operations when both architectures apply a sequence of elementary instructions in an iterative manner (e.g., rank order filters), the RISC is the superior, because its elementary operators are more versatile more accurate, and faster.

The internal analog data representation has both architectural and functional advantages. From architectural point of view, the most important feature is that no AD converter is needed on the cell level, because the sensed optical image can be directly saved in the analog memories, leading to significant silicon space savings. Moreover, the analog memories require smaller silicon area than the equivalent digital counterparts. From the functional point of view, the topographic analog and logic data representations make the implementation of efficient diffusion, averaging, and global OR networks possible.

The drawback of the internal analog data representation and processing is the signal degradation during operation or over time. According to experience, accuracy degradation was more significant in the old ACE16k design [9] than in the recent Q-Eye [19] or SCAMP [15] ones. While in the former case 3-5 grayscale operations led to significant degradations, in the latter ones even 10-20 grayscale operations can conserve the original image features. This makes it possible to implement complex nonlinear image processing functions (e.g., rank order filter) on discrete time architectures, while it is practically impossible on the continuous ones (ACE16k).

The two representatives of discrete time solutions, SCAMP and Q-Eye, are slightly similar in design. The SCAMP chip was fabricated by using 0.35 micron technology. The cell array size is 128×128. The cell size is 50×50 micron, and the maximum power consumption is about 200mW at 1.25MHz clock rate. The array of Q-Eye chip has 144×176 cells. It was fabricated on 0.18 micron technology. The cell size is about 30×30 micron. Its speed and power consumption range is similar to that of the SCAMP chip. Both SCAMP and Q-Eye chips are equipped with single-step mean, diffusion, and global OR calculator circuits. Q-Eye chip also provides hardware support for single-step binary 3×3 morphologic operations.

2.5 Fine-grain fully parallel cellular architecture with continuous time processing

Fully parallel cellular continuous time architectures are based on arrays of spatially interconnected dynamic asynchronous processor cells. Naturally, these architectures exhibit fine-grain parallelism, to be able to perform continuous time spatial waves physically in the continuous value electronic domain. Since these are very carefully optimized, special purpose circuits, they are super-efficient for computations they were designed to perform. We have to emphasize, however, that they are not general purpose image processing devices. Here we mainly focus on two designs. Both of them can generate continuous time spatial-temporal propagating waves in a programmable way. While the output of the first one (ACE-16k [9]) can be in the grayscale domain, the output of the second one (ACLA [12][13]) is always in the binary domain.

The ACE-16k [9] is a classical CNN Universal Machine type architecture equipped with feedback and feed-forward template matrices [3], sigmoid type output characteristics, dynamically changing state, optical input, local (cell level) analog and logic memories, local logic, diffusion and averaging network. It can perform full-signal range type CNN operations [4]. Therefore, it can be used in retina simulations or other spatial-temporal dynamical system emulations, as well. Its typical feed-forward convolution execution time is in the 5-8 microsecond range, while the wave propagation speed from cell-to-cell is up to 1 microsecond. Though its internal memories, easily re-programmable convolution matrices, logic operations, and conditional execution options make it attractive to use as a general purpose high-performance sensor-processor chip for the first sight, its limited accuracy, large silicon area occupation (~80×80 micron/cell on 0.35 micron 1P5M STM technology), and high power consumption (4-5 Watts) prevent the immediate usage in various vision application areas.

The other architecture in this category is the Asynchronous Cellular Logic Array (ACLA) [12], [13]. This architecture is based on spatially interconnected logic gates with some cell level asynchronous controlling mechanisms, which allow ultra high-speed spatial binary wave propagation only. Typical binary functionalities implemented on this network are: *trigger wave, reconstruction, hole finder, shadow*, etc. Assuming more sophisticated control mechanism on the cell level, it can even perform *skeletonization* or *centroid* calculations. Their implementation is based on a few minimal size logic transistors, which makes them hyper-fast, extremely small, and power-efficient. They can reach 500 ps/cell wave propagation speed, with 0.2mW power consumption for a 128×128 sized array. Their very small area requirement (16×8 micron/cell on 0.35 micron 3M1P AMS technology) makes them a good choice to be implemented as a co-processor in any fine-grain array processor architecture.

3 IMPLEMENTATION AND EFFICIENCY ANALYSIS OF VARIOUS OPERATORS

Based on the implementation methods, in this section, we introduce a new 2D operator categorization. Then, the implementation methods on different architectures are described and analyzed from the efficiency aspect.

Here we examine only the 2D single-step neighborhood operators, and the 2D, neighborhood based wave type operators. The more complex, but still local operators (such as Canny edge detector) can be built up by using these primitives, while other operators (such as Hough or Fourier transform) require global processing, which is not supported by these architectures.

3.1 Categorization of 2D operators

Due to their different spatial-temporal dynamics, different 2D operators require different computational approaches. The categorization (Figure 4) was done according to their implementation methods on different architectures. It is important to emphasize that we categorize operators (functionalities) here, rather than wave types, because the wave types are not necessarily inherited by the operator itself, but rather by its implementation method on a particular architecture. As we will see, the same operator is implemented with different spatial wave dynamic patterns on different architectures. The most important 2D operators, including all the CNN operators [22]are considered here.

The first distinguishing feature is the location of active pixels [22]. If the active pixels are located along one or few one-dimensional stationary or propagating curves at a time, we call the operator front-active. If the active pixels are everywhere in the array, we call it area-active.

The common property of the front-active propagations is that the active pixels are located only at the propagating wave fronts [5]. This means that at the beginning of the wave dynamics (transient) some pixels become active, others remain passive. The initially active pixels may initialize wave fronts which start propagating. A propagating wave front can activate some further passive pixels. This is the mechanism how the wave proceeds. However, pixels apart from a waveform cannot become active [22]. This theoretically enables us to compute only the pixels which are along the front lines, and not waste efforts on the others. The question is which are the architectures that can take advantage of such a spatially selective computation.

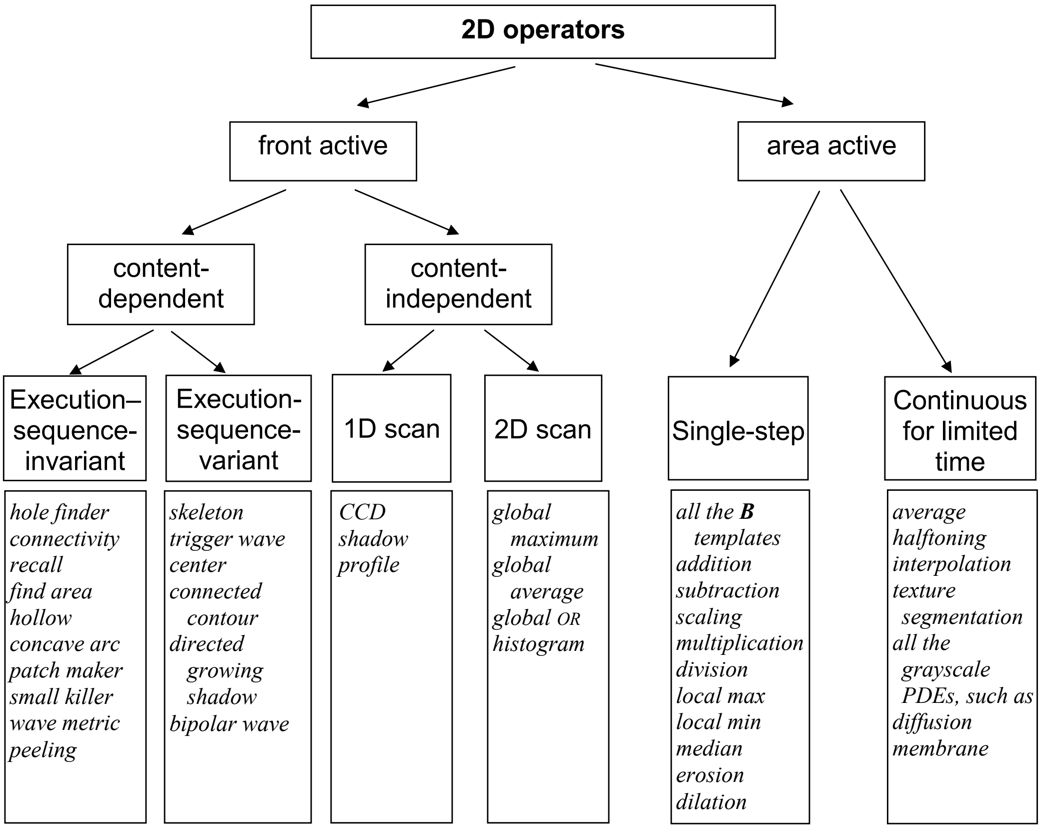


Figure 4. 2D local operator categorization

The front active operators such as *reconstruction*, *hole finder*, or *shadow* are typically binary waves. In CNN terms, they have binary inputs and outputs, positive self-feedback, and space invariant template values. Figure 4 contains three exemptions: *global max*, *global average*, and *global OR*. These functions are not wave type operators by nature; however, we will associate a wave with them which solves them efficiently.

The front active propagations can be content-dependent or content-independent. The content-dependent operator class contains most of the operators where the direction of the propagation depends on the local morphological properties of the objects (e.g., shape, number, distance, size, connectivity) in the image (e.g., *reconstruct*). An operator of this class can be further distinguished as execution-sequence-variant (*skeleton*, etc) or execution-sequence-invariant (*hole finder, recall, connectivity*, etc). In the first case the final result may depend on the spatial-temporal dynamics of the wave, while in the latter it does not. Since the content-dependent operator class contains the most interesting operators with the most exciting dynamics, they are further investigated in Section 3.1.1.

We call the operators content-independent when the direction of the propagation and the execution time do not depend on the shape of the objects (e.g., *shadow*). According to propagation, these operators can be either one- (e.g., *CCD*, *shadow, profile [27]*) or two-dimensional (*global maximum, global OR, global average, histogram*). Content-independent operators are also called single-scan, for their execution requires a single scanning of the entire image. Their common feature is that they reduce the dimension of the input 2D matrices to vectors (*CCD, shadow, profile, histogram*) or scalars (*global maximum, global average, global OR*). It is worth to mention that on the coarse- and fine-grain topographic array processors the *shadow, profile* and *CCD* are content-dependent operators, and the number of the iterations (or analog transient time) depends on the image content only. The operation is completed, when the output is ceased to change. Generally, however, , it is less efficient to include a test to detect a stabilized output, than to let the operator run in as many cycles as it would run in the worst case.

The area active operator category contains the operators where all the pixels are to be updated continuously (or in each iteration). A typical example is *heat diffusion*. Some of these operators can be solved in a single update of all the pixels (e.g., all the CNN **B** templates *[27]*), while others need a limited number of updates (*halftoning, constrained heat diffusion*, etc.).

The fine-grain architectures update every pixel location in fully parallel in each time instance. Therefore, the area active operators are naturally the best fit for these computing architectures.

3.1.1 Execution-sequence-variant versus execution-sequence-invariant operators. The crucial difference in fine-grain and pipe-line architectures is in their state overwriting methods. In the fine-grain architecture the new states of all the pixels are calculated in parallel, and then the previous one is overwritten again in parallel, before the next update cycle is commenced. In the pipe-line architecture, however, the new state is calculated pixel-wise, and it is selectable whether to overwrite a pixel state before the next pixel is calculated (pixel overwriting), or to wait until the new state value is calculated for all the pixels in the frame (frame overwriting). In this context, update means the calculation of the new state for an entire frame. Figure 5 and Figure 6 illustrate the difference between the two overwriting schemes. In case of an execution-sequence-variant operation, the result depends on the frame overwriting schemes.

Here the calculation is done pixel-wise, left to right and row-wise top to down. As we can see, overwriting each pixel before the next pixel's state is calculated (pixel overwriting) speeds up the propagation in the directions which corresponds to the direction the calculation proceeds.

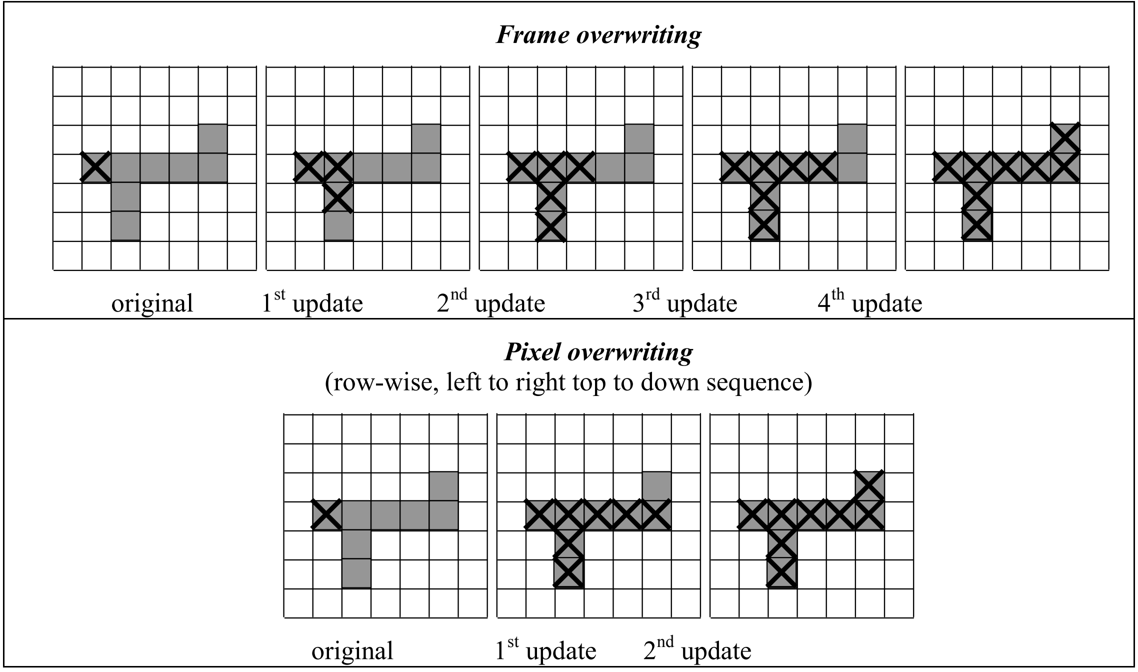


Figure 5. Execution-invariant sequence in different overwriting schemes. Given an image with grey objects against white background. The propagation rule is that the propagation starts from the marked pixel (denoted by X), and it can go on within the grey domain, proceeding one pixel in each update. In the figure, we can see the results of each update. Update means calculating the new states of all the pixels in the frame.

Based on the above, it is easy to draw the conclusion that the two updating schemes lead to two completely different propagation dynamics and final results in execution-variant cases. One is slower, but controlled, the other one is faster, but uncontrolled. The first can be used in cases when speed maximization is the only criterion, while the second is needed when the shape and the dynamics of the propagating wave front count. We called the former case execution-sequence-invariant operators, the latter one execution-sequence-variant operators (Figure 4).

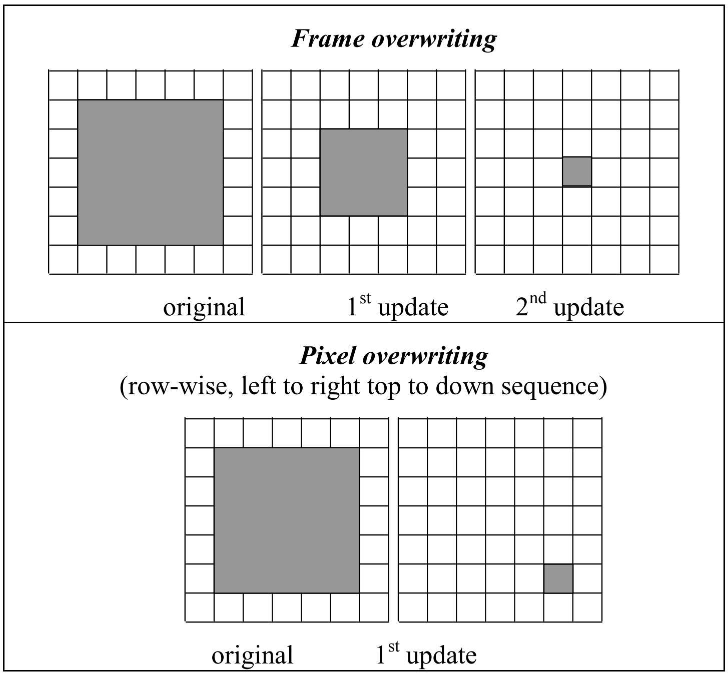


Figure 6. Execution-variant sequence in different overwriting schemes. Given an image with grey objects against white background. The propagation rule is that those pixels of the object, which has both object and background neighbor should became background. In this case, the subsequent peeling leads to find the centroid in the frame overwriting method, while it extracts one pixel of the object in the pixel overwriting mode.

In the fine-grain architecture we can use the frame overwriting scheme only. In the coarse-grain architecture both pixel overwriting and frame overwriting methods can be selected within the individual sub-arrays. In this architecture, we may determine even the calculation sequence, which enables speed-ups in different directions in different updates. Later, we will see an example to illustrate how the *hole finder* operation propagates in this architecture. In the pipe-line architecture, we may decide which one to use, however, we cannot change the direction of the propagation of the calculation, without paying a significant penalty for it in memory size and latency time.

3.2 Processor utilization efficiency of the various operation classes

In this subsection, we will analyze the implementation efficiency of various 2D operators from different aspects. We will study both the execution methods and the efficiency from the processor utilization aspect. Efficiency is a key question, because in many cases one or a few wave fronts sweep through the image, and one can find active pixels only in the wave fronts, which is less than one percent of the pixels, hence, there is nothing to calculate in the rest of image. We define a measure of efficiency of processor utilization with the following form:

$$\eta=O_r/O_t \qquad (1)$$

where:

O_r: the minimum number of required elementary steps to complete an operation, assuming that the inactive pixel locations are not updated

O_t: is the total number of elementary steps performed during the calculation by all the processors in the particular processor architecture.

The efficiency of processor utilization figure will be calculated in the following where it applies, because this is a good parameter (among others) to compare the different architectures.

3.2.1 Execution-sequence-invariant content-dependent front-active operators. A special feature of content-dependent operators is that the path and length of the path of the propagating wave front drastically depend on the image contents itself. For example, the range of the necessary frame overwritings with a hole finder operation varies from zero overwriting to n/2 in a fine-grain architecture, assuming n×n pixel array size. Hence, neither the propagation time, nor the efficiency can be calculated without knowing the actual image.

Since the gap between the worst and best case is extremely high, it is not meaningful to provide these limits. Rather, it makes more sense to provide approximations for certain image types. But before that, we examine how to implement these operators on the studied architectures. For this purpose, we will use the *hole finder* operator, as an example. Here we will clearly see how the wave propagation follows different paths, as a consequence of varying propagation speed corresponding to different directions. Since this is an execution-sequence-invariant operation, it is certain that wave fronts with different trajectories lead to the same good result.

The *hole finder* operation, that we will study here, is a "grass fire" operation, in which the fire starts from all the boundaries at the beginning of the calculation, and the boundaries of the objects behave like firewalls. In this way, at the end of the operation, only the holes inside objects remain unfilled.

The *hole finder* operation may propagate to any direction. On a **fine-grain architecture** the wave fronts propagate one pixel steps in each update. Since the wave fronts start from all the edges, they meet in the middle of the image in typically n/2 updates, unless there are large structured objects with long bays which may fold the grass fire into long paths. In case of a text for example, where there are relatively small non-overlapping objects (with diameter k) with large but not spiral like holes, the wave stops after *n/2+k* operations. In case of an arbitrary camera image with an outdoor scene, in most cases *3*n* updates are enough to complete the operation, because the image may easily contain large objects blocking the straight paths of the wave front.

On a **pipe-line architecture**, thanks to the pixel overwrite scheme, the first update fills up most of the background (Figure 7). Filling in the remaining background requires typically *k* updates, assuming the largest concavity size with *k* pixels. This means that on a pipe-line architecture, roughly *k+1* steps are enough, considering small, non-overlapping objects with size *k*.

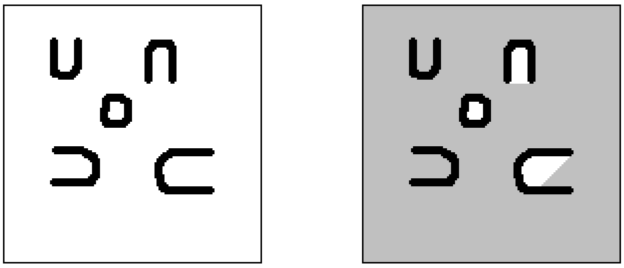


Figure 7. Hole finder operation calculated with a pipe-line architecture. (a): original image. (b): result of the first update. (The freshly filled up areas are indicated with grey, just to make it more comprehensible. However, they are black on the black-and-white image, same as the objects.)

In the **coarse-grain architecture** we can also apply the pixel overwriting scheme within the *N*×*N* sub-arrays (Figure 8). Therefore, within the sub-array, the wave front can propagate in the same way, as in the pipe-line architecture. However, it cannot propagate beyond the boundary of the sub-array, in a single update. In this way, the wave front can propagate N positions in the direction which correspond to the calculation directions, and one pixel in the other directions, in each update. In this way, in *n/N* updates, the wave-front can propagate n positions in the supported directions. However, the *k* sized concavities in other directions would require *k* more steps. To avoid these extra steps, without compromising the speed of the wave-front, we can switch between the top-down and the bottom-up calculation directions after each update. The resulting wave-front dynamics is shown in Figure 9. This means that for an image, containing only few, non-overlapping small objects with concavities, we need about *n/N+k* steps to complete the operation.

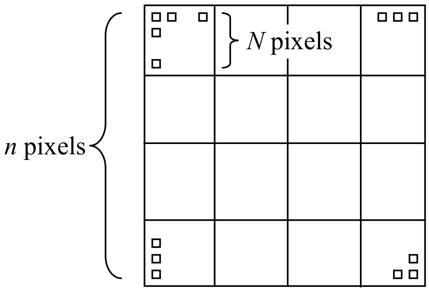


Figure 8. *Coarse-grain architecture with $n \times n$ pixels. Each cell is to process an $N \times N$ pixel sub-array.*

The **DSP-memory architecture** offers several choices depending on the internal structure of image. The simplest is to apply pixel overwriting scheme, and switch the direction of the calculation. In case of binary image representation, only the vertical directions (up or down) can be efficiently selected, due to the packed 32 pixel line segment storage and handling. In this way the clean vertical segments (columns of background with maximum one object) are filled up after the second update, and filling up the horizontal concavities would require k steps.

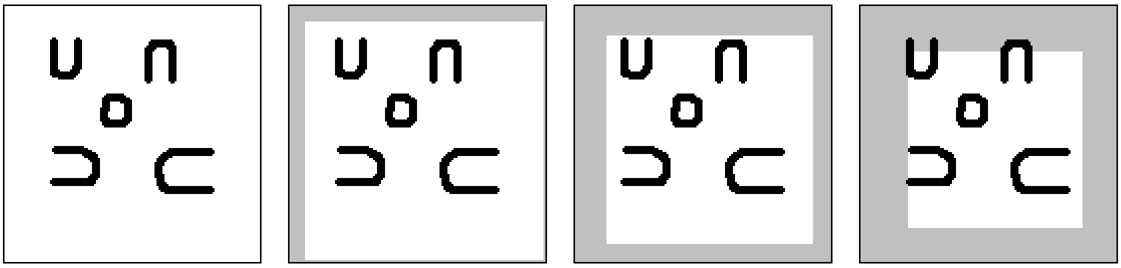


Figure 9. *Hole finder operation calculated in a coarse-grain architecture. The first picture shows the original image. The rest shows the sequence of updates, one after the other. The freshly filled-up areas are indicated with grey (instead of black) to make it easier to follow the dynamics of calculation.*

3.2.2 Execution-sequence-variant content-dependent front active operators The calculation method of the execution-sequence-variant content-dependent front active operators is very similar to that of their execution-sequence-invariant counterparts. The only difference is that in each of the architectures the frame overwriting scheme should be used. This does not make any difference in fine-grain architectures, however, it slows down all the other architectures significantly. In the DSP-memory architectures, it might even make sense to switch to one byte/pixel mode, and calculate updates in the wave fronts only.

3.2.3 1D content-independent front active operators (1D scan). In the 1D content-independent front active category, we use the vertical shadow (north to south) operation as an example. In this category, varying the orientation of propagation may cause drastic efficiency differences on the non-topographic architectures.

On a **fine-grain discrete time** architecture the operator is implemented in a way that in each time instance, each processor should check the value of its upper neighbor. If it is +1 (black), it should change its state to +1 (black), otherwise the state should not change. This can be implemented in one single step in a way, that each cell executes an *OR* operation with its upper neighbor, and overwrites its state with the result. This means that in each time instance the processor array executed n^2 operations, assuming $n \times n$ pixel array size.

In discrete time architectures, each time instance can be considered as a single iteration. In each iteration the shadow wave front moves by one pixel to the south, that is we need n steps for the wave front to propagate from the top row to the bottom (assuming boundary condition above the top row). In this way, the total number of operations, executed during the calculation is n^3. However, the strictly required number of operations is n^2, because it is enough to do these calculations at the wave front, only ones in each row, starting from the top row, and going down row by row, rolling over the results from the front line to the next one. In this way, the efficiency of the processor utilization in *vertical shadow* calculation in the case of fine-grain discrete time architectures is

$$\eta = 1/n \qquad (2)$$

Considering computational efficiency, the situation is the same in **fine-grain continuous architectures**. However, from the point of power efficiency the Asynchronous Cellular Logic Network [13] is very advantageous, because only the active cells in the wave front consume switching power. Moreover, the extraordinary propagation speed (500 ps/cell) compensates for the low processor utilization efficiency.

If we consider a **coarse-grain architecture** (Figure 8), the *vertical shadow* operation is executed in a way that each cell executes the above *OR* operation from its top row, and goes on from the top downwards in each column. This means that $N\times N$ operations are required for a cell to process its sub-array. It does not mean, however, that in the first $N\times N$ steps the whole array is processed correctly, because only the first cell row has all the information for locally finalizing the process. For the rest of the rows their upper boundary condition have not "arrived", hence at these locations correct operations cannot be performed. Thus, in the first $N\times N$ steps, the first N rows were completed only. However, the total number of operation executed by the array during this time is

$$O_{NxN}=N*N * n/N * n/N=n*n, \qquad (3)$$

because there are $n/N * n/N$ processors in the array, and each processor is running all the time. To process also the rest of the lines we need to perform

$$O_t=O_{NxN} * n/N=n^3/N. \qquad (4)$$

The resulting efficiency is:

$$\eta=N/n \qquad (5)$$

It is worth to stop at this result for a while. If we consider a fine-grain architecture (N=1), the result is the same as we obtained in (2). Its optimum is N=n (one processor per column) when the efficiency is 100%. It turns out that in case of *vertical shadow* processing, the efficiency increases by increasing the number of the processor columns, because in that case, one processor has to deal with less columns. However, the efficiency does not increase when the number of the processor rows is increased. (Indeed, one processor/column is the optimal, as it was shown.) Thought the unused processor cells can be switched off with minor extra effort to increase power efficiency, but it would certainly not increase processor utilization.

Pipe-line architecture as well as **DSP-memory architecture** can execute *vertical shadow* operation with 100% processor utilization, because there are no multiple processors in a column working parallel.

We have to note, however, that shadows to other three directions are not as simple as the one to downwards. In **DSP architectures**, *horizontal shadows* cause difficulties, because the operation is executed parallel on a 32×1 line segment, hence only one of the positions (where the actual wave front is located) performs effectual calculation. If we consider a left to right shadow, this means that once in each line (at left-most black pixel), the shadow propagation should be calculated precisely for each of the 32 positions. Once the "shadow head" (the 32 bit word, which contains the left-most black pixel) is found, and the shadow is calculated within this word, the task is easier, because all the rest of the words in the line should be filled with black pixels, independently of their original content. Thus the overall resulting cost of a *horizontal shadow* calculation on a **DSP-memory architecture** can be even *20* times higher than that of a *vertical shadow* for a 128×128 sized image. Similar situation might happen in **coarse-grain architectures**, if they handle $n\times 1$ binary segments.

While **pipe-line architectures** can execute the *left to right and top to bottom shadows* in a single update at each pixel location, the other directions would require *n* updates, unless the direction of the pixel flow is changed. The reason for such a high inefficiency is that in each update, the wave front can propagate only one step in the opposite direction.

3.2.4 2D content-independent front active operators (2D scan). The operators belonging to the 2D content-independent front active category require simple scanning of the frame. In *global max* operation for example, the actual maximum value should be passed from one pixel to another one. After we scanned all the pixels, the last pixel carries the global maximum pixel value.

In **fine-grain architectures** this can be done in two phases. First, in *n* comparison steps, each pixel takes over the value of its upper neighbor, if it is larger than its own value. After *n* steps, each pixel in the bottom row contains the largest value of its column. Then, in the second phase after the next *n* horizontal comparison steps, the global maximum appears at the end of the bottom row. Thus, to obtain the final result requires *2n* steps. However, as a fine-grain architecture executes $n\times n$ operations in each step, the total number of the executed operations are $2n^3$. However, the minimum number of requested operation to find the largest value is n^2 only. Therefore, the efficiency in this case is:

$$\eta=1/2n \qquad (6)$$

The most frequently used operation in this category is *global OR*. To speed up this operation in the fine-grain arrays, a *global OR* net is implemented usually [9][3]. This $n \times n$ input OR gate requires minimal silicon space, and enables us to calculate *global OR* in a single step (a few microseconds).

However, in that case, when a fine-grain architecture is equipped with *global OR*, the global maximum can be calculated as a sequence of iterated *threshold* and *global OR* operations with interval halving (successive approximation) method applied in parallel to the whole array. This means that a global threshold is applied first for the whole image at level ½, and if there are pixels, which are larger than this, we will do the next global thresholding at ¾, and so on. Assuming 8 bit accuracy, this means that in 8 iterations (16 operations), the global maximum can be found. The efficiency is much better in this case:

$$\eta=1/16$$

In **coarse-grain architectures**, each cell calculates the *global maximum* in its sub-array in $N \times N$ steps. Then n/N vertical steps come, and finally, n/N horizontal steps to find the largest values in the entire array. The total number of steps in this case is $N^2 + 2n/N$, and in each step, $(n/N)^2$ operations are executed. The efficiency is:

$$\eta= n^2 /(N^2 + 2n/N)*(n/N)^2=1/(1+2n/N^3) \qquad (7)$$

Since the sequence of the execution does not matter in this category, it can be solved with 100% efficiency in **pipe-line** and the **DSP-memory architectures**.

3.2.5 Area active operators. The area active operators require some computation in each pixel in each update; hence, all the architectures work with 100% efficiency. Since the computational load is very high here, it is the most advantageous for the many-core architectures, because the speed advantage of the many processor can be efficiently utilized.

3.3 Multi-scale processing

Generally, multi-scale processing technique is applied in those situations, when the calculation of an operator on a downscaled image leads to acceptable result from accuracy point of view. Since the calculation of the operator requires significantly smaller computational effort in a lower resolution, in many cases the downscaling, the upscaling (if needed), and the calculation on the downscaled domain requires less computational effort than the calculation of the operator in the original scale. Diffusion is a typical example for this.

Here we discuss how the approximation of the *diffusion* operator leads to a multi-scale representation, and analyze its implementation on the discussed architectures. However, with a similar approach, other binary or grayscale front- and area- active operators can be scaled down and executed, as well.

Two ways are used generally to compute the *diffusion* operator on topographic array computers. The first is the iterative way. The second way is to implement it on a hardwired resistive grid, as we have seen in analog fine-grain topographic architectures. Here we deal with the first option.

The problem with the iterative implementation of *diffusion* equation is that after a few iterations the differences of the neighboring pixels become very small, and the propagation slows down. Moreover, if there are some computational errors, due to the limited precision of the processors, calculation of the *diffusion* equation will be useless and irrelevant, after a while. To obtain accurate solution would require floating point number representation and a large number of iterations. However, one can approximate it by using multi-scale approach, as it is shown in Figure 10. As we can see, 10 iterations on a full scale image result in small blurring only, while the same 10 iterations on a downscaled image lead to large scale *diffusion*. The downscaling and the up scaling with linear interpolation need less computational effort, than a single iteration of the diffusion. Moreover, the calculation of an iteration on the downscaled image requires only $1/s^2$ (s is the downscaling factor) of computational power. Naturally, it should be kept in mind that this method can be used in that cases only when the accuracy of the approximated diffusion operator is good satisfactory in a particular application.

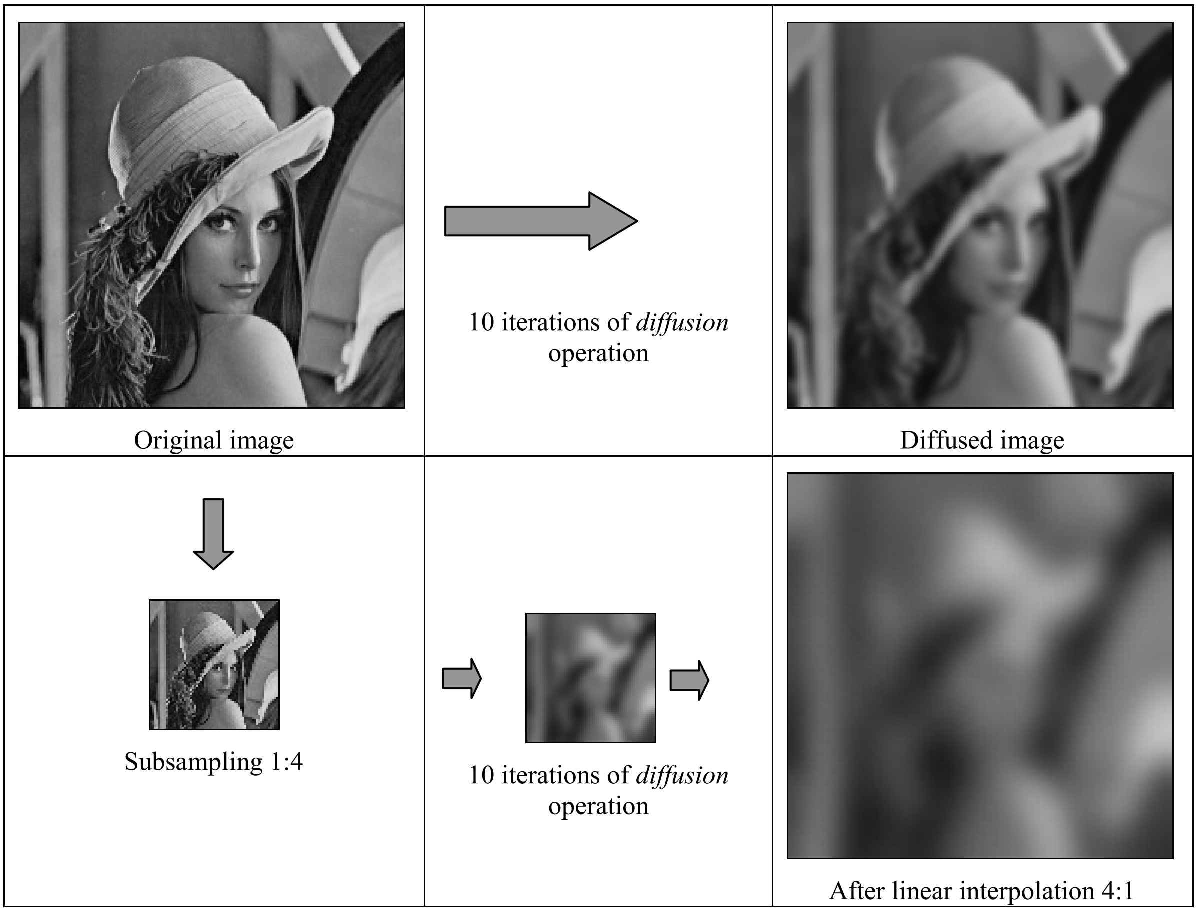

Original image
10 iterations of *diffusion* operation
Diffused image
Subsampling 1:4
10 iterations of *diffusion* operation
After linear interpolation 4:1

Figure 10. Iterative approximation of the diffusion operator combining different spatial resolutions.

The multi-scale iterative *diffusion* can be implemented on classic DSP-memory architectures, multi-core pipe-line architectures (Figure 11), and on coarse-grain architectures as well. In the discussed fine-grain architectures the multi-scale approach cannot be efficiently implemented, because in most cases the image should be read out, resampled, and written back, which takes long time.

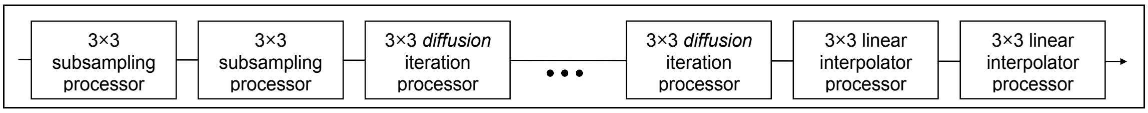

3×3 subsampling processor
3×3 subsampling processor
3×3 *diffusion* iteration processor
• • •
3×3 *diffusion* iteration processor
3×3 linear interpolator processor
3×3 linear interpolator processor

Figure 11. Implementation of multi-scale diffusion calculation approach on a pipe-line architecture. In this example, it starts with two subsampling steps. The pixel clock drops into 1/16th. Then the computationally hard diffusion calculations can be applied much easier since more time is available for each pixel. The processing is completed with the 2 interpolation steps.

4 COMPARISON OF THE ARCHITECTURES

As we have stated in the previous section, front active wave operators run well under 100% efficiency on topographic architectures, since only the wave fronts need calculation, and the processors of the array in non-wave front positions do dummy cycles only or may be switched off. On the other hand, the computational capability (GOPs) and the power efficiency (GOPs/W) of multi-core arrays are significantly higher than those of DSP-memory architectures. To be able to compare these different architectures, we analyze here their existing realizations. We show the efficiency figures of these architecture realizations in different categories. To make fair comparison with relevant industrial devices we have selected two market-leading, video processing units, a DaVinci video processing DSP from Texas Instruments (TMS320DM6443) [18], and a Spartan 3 DSP FPGA from Xilinx (XC3SD3400A) [28]. Both of these products' functionalities, capabilities and prices were optimized to efficiently perform embedded video analytics.

Table I summarizes the basic parameters of the different architecture realizations, and indicates the processing time of a 3×3 convolution, and a 3×3 erosion. To make the comparison easier, values are calculated for images of 128×128 resolution. For this purpose, we considered 128×128 Xenon and Q-Eye chips. Some of these data are from catalogues, other ones are from measurements, or estimation. As fine-grain architecture examples, we included both the SCAMP and Q-Eye architectures.

We can see from Table I, the DSP was implemented on 90nm, while the FPGA on 65 nm technologies. In contrast Xenon, Q-Eye, and SCAMP were implemented on more conservative technologies and their power budget is an order of magnitude smaller. When we compare the computational power figures, we also have to take these parameters into consideration.

Table I shows the speed advantages of the different architecture realizations, compared to DSP-memory setup both in 3×3 neighborhood arithmetic (8 bit/pixel) and morphologic (1 bit/pixel) cases. This indicates the speed advantage of the area active single step, and the front active content-dependent execution-sequence-variant operators. In Table II, we summarize the speed relations of the rest of the wave type operations. The table indicates the computed values, using the formulas that we have derived in the previous section. In some cases, however, the coarse- and especially the fine-grain arrays contain some special accelerator circuits, which takes the advantage of the topographic arrangement and the data representation (e.g., global OR network, mean network, diffusion network). These are marked by notes, and the real speed-up with the special hardware is shown in parenthesis.

Table I Computational parameters of the different architecture realizations for arithmetic (3×3 convolution) and logic (3×3 binary erosion) operations.

	DSP **(DaVinci^{+})**	**Pipe-line** **(FPGA^{++})**	**Coarse-grain** **(Xenon)**	**Fine-grain** **(SCAMP/Q-Eye)**
Silicon technology	90nm	65nm	180nm	350/180nm
Silicon area mm^2			100	100/50
Power consumption	1.25 W	2-3W	0.08 W	0.20 W
Arithmetic proc. clock speed	600 MHz	250 MHz	100 MHz	1,2 / 2.5 MHz
Number of arithmetic proc.	8	120	256	16384
Efficiency of arithmetic calc.	75% *	100%	80% ***	50% **
Arit. computational speed	3.6 GMAC	30 GMAC	20 GMAC	~20GOPS****
3×3 convolution time	42.3 µs*****	4.9 µs	12.1 µs	22 µs ****
Arithmetic speed-up	1	8.6	3.5	1.9
Morph. proc. clock speed	600 MHz	83 MHz	100 MHz	1,2 / 5 MHz
Number of morphologic proc.	64	864	2048	147456
Morphologic processor kernel type	2 × 32 bit	96 × 9 bit	256 × 8 bit	16384 × 9 bit
Efficiency of morphological calc.	28% *	100%	90% ***	100%
Morphologic computational power	10 GOPS	71 GOPS	184 GOPS	737 GOPS
3×3 morphologic operation time	13.6 µs*****	2.05 µs	1.1 µs	0.2 µs
Morphologic speed-up	1	6.6	12.4	68.0

$^{+}$ Texas Instrument DaVinci video processor (TMS320DM64x)

$^{++}$ Xilinx Spartan 3ADSP FPGA (XC3SD3400A)

* processors are faster than cache access

** data access from neighboring cell is an additional clock cycle

*** due to pipe-line stages in the processor kernel, (no effective calculation in each clock cycle)

**** no multiplication, scaling with few discrete values

***** these data-intensive operators slow down to 1/3rd or even 1/5th when the image does not fit to the internal memory (typically above 128×128 with a DaVinci, which has 64kByte internal memory)

Among the multi-core processor architectures, the pipe-line is the only one, that can handle both high-resolution and low resolution images too, due to the relatively small memory demand. While the coarse- and fine-grain architectures require the storage of 6-8 entire frames, the pipe-line architecture needs only few a lines for each processor. In case of a mega-pixel image, it can be less than one third of the frame. This means that as opposed to the coarse- and fine-grain architectures, the pipe-line architecture can **trade speed for resolution**. This is very important, because the main criticism of the topographic architectures is that they cannot handle large images, and many of the users do not need their 1000+ FPS. The price what the pipe-line architectures pay for this trade-off is their rigidity. Once the architecture is downloaded to an FPGA (or an ASIC is fabricated), it cannot be flexibly reprogrammed, only the computational parameters can be varied. It is very difficult to introduce conditional branching, unless all the passes of the branching are implemented on silicon (multi-thread pipeline), or significant delay or latency is introduced.

In our comparison tables, we have represented a typical FPGA as a vehicle to implement the pipe-line architectures. The only reason is that all the currently available pipe-line architectures are implemented in FPGAs is mainly attributed to much lower costs and quicker time-to-market development cycles. However, they could also be certainly implemented in ASIC, which would significantly reduce their power consumption, and decrease their large-volume prices making it possible to process even multi-mega pixel images at a video rate.

Table II Speed relations in the different function groups calculated for 128×128 sized images. The notes indicate the functionalities by which the topographic arrays are speeded up with special purpose devices.

	DSP (DaVinci[+])	**Pipe-line (FPGA[++])**	**Coarse-grain (Xenon)**	**Fine-grain discrete time (SCAMP/ Q-Eye)**	**Fine-grain continuous time (ACLA)**
1D content-independent front active operators					
processor util. efficiency	100%	100%	N/n: 6.25%	1/n: 0.8%	1/n: 0.8%
speed-up in advantageous direction (vertical)	1	6.6	0.77	0.53	188
speed-up in disadvantageous direction (horizontal)	1	1	2	10.6	3750
2D content-independent front active operators					
processor util. efficiency	100%	100%	*$1/(1+2n/N^3)$: 66%*	1/2n: 0.4%	-
speed-up (*global OR*)	1	6.6	8.2 (13*)	0.27 (20*)	n/a
speed-up (*global max*)	1	8.6	2.3	n/a	n/a
speed-up (*average*)	1	8.6	2.3	n/a (2.5)**	n/a
Execution-sequence-invariant content-dependent front active operators					
hole finder with k=10 sized small objects	4 updates	k+1 updates (11)	n/N+k (26)	n/2+k updates (74)	n/2+k updates (74)
speedup	1	2.4	1.9	3.7	1500
Area active					
processor util. efficiency	100%	100%	100%	100%	
speedup	1	8.6	3.5	1.9 (210***)	n/a
Multi-scale					
1:4 scaling	1	8.6	3.5	0.1	n/a

[+] Texas Instrument DaVinci video processor (TMS320DM64x)

[++] Xilinx Spartan 3ADSP FPGA (XC3SD3400A)

* Hard wired global OR device speeds up this function (<1 μs concerning the whole array)

** Hard wired mean calculator device makes this function available (~2 μs concerning the whole array)

*** Diffusion calculated on resistive network (<2 μs concerning the whole array)

Table III shows the computational power, the consumed power and the power efficiency of the selected architecture realizations. As we can see, the three topographic arrays have over hundred times power efficiency advantage compared to DSP-memory setups. This is due to their local data access, and relatively low clock frequency. In case of ASIC implementation, the power efficiency of the pipe-line architecture would also be increased with a similar factor.

Table III Computational power, and the consumed electronic power, and their proportion in different architecture realizations for convolution operations. As a comparison, the Cell Multiprocessor developed by IBM-Sony-Toshiba [17] is also given.

	GOPs	W	GOPs/W
DaVinci	3.6	1.25	2.88
Pipe-line (FPGA)	30	3	10
Xenon (64x64)	10	0.02	500
SCAMP (128x128)	20	0.2	100
Q-Eye	25	0.2	125
Cell multiprocessor	225	85	2.6

Figure 12 shows the relation between the frame-rate and the resolution in a video analysis task. Each of the processors had to calculate 20 *convolutions*, 2 *diffusions*, 3 *means*, 40 *morphologies* and 10 *global OR*s. Only the DSP-memory and pipe-line architectures support trading between resolution and frame-rate. The characteristics of these architecture realizations form lines. The chart shows the performance of the three discussed chips too. The chips are represented here with their physical sizes. Naturally, this chart belongs to this particular task with the given operations "basket". By using a different one, different chart would come out.

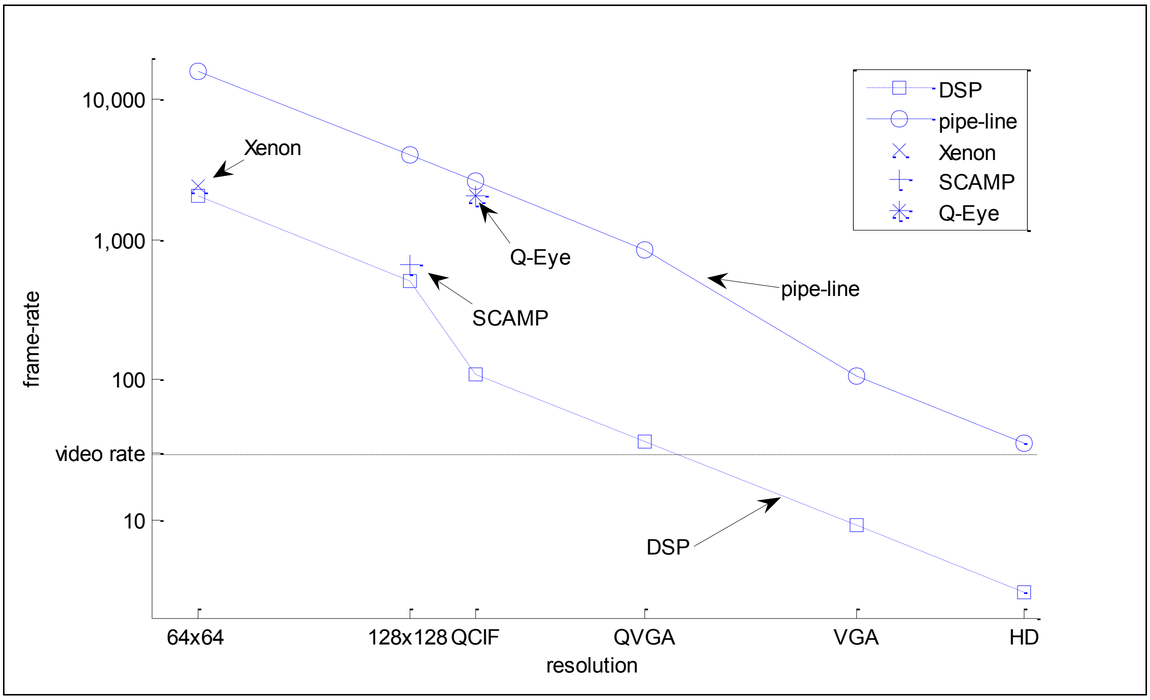


Figure 12. Frame-rate versus resolution in a typical image analysis task. Both of the axes are in logarithmic scale.

As it can be seen in Figure 12, both SCAMP and Xenon have the same speed as the DSP. In the case of Xenon, this is so, because its array size is 64×64 only. In the case of SCAMP, the processor was designed for very accurate low power calculation by using a conservative technology. In this particular task, the Q-Eye chip was almost as fast as the pipe-line architectures, thanks to its integrated diffusion circuitry and hardware support of binary morphology.

5 EFFICIENT ARCHITECTURE SELECTION

So far, we have studied how to implement the different wave type operators on different architectures, identified constrains and bottlenecks, and analyzed the efficiency of these implementations. After having these results in our hand, we can define rules for optimal image processing architecture selection for topographic problems.

Image processing devices are usually special purpose architectures, optimized for solving specific problems or a family of similar algorithms. Figure 13 shows a method of special purpose processor architecture selection. It always starts with the understanding of the problem in all aspect. Then, different algorithms suitable for solving the problem are derived. The algorithms are described with flowchart, with the list of the used operations, and with the specification of the most important parameters. In this way, a set of formal data describes the algorithms, which are as follows: ***resolution, frame-rate, pixel clock, latency, computational demand*** (type and number of operators), ***and flowchart***. Other application-specific (secondary) parameters are also given: maximal ***power consumption***, maximal ***volume***, ***cost*** etc. The algorithm derivation is a human activity supported by various simulators for evaluation and verification purposes.

The next step is the architecture selection. By using the previously compiled data, we can define a methodology for the architecture selection step. As we will see, based on the formal specifications, we can derive the possible architectures. There might not be any, there might be exactly one, or there might be several, according to the demands of the specification of the algorithm.

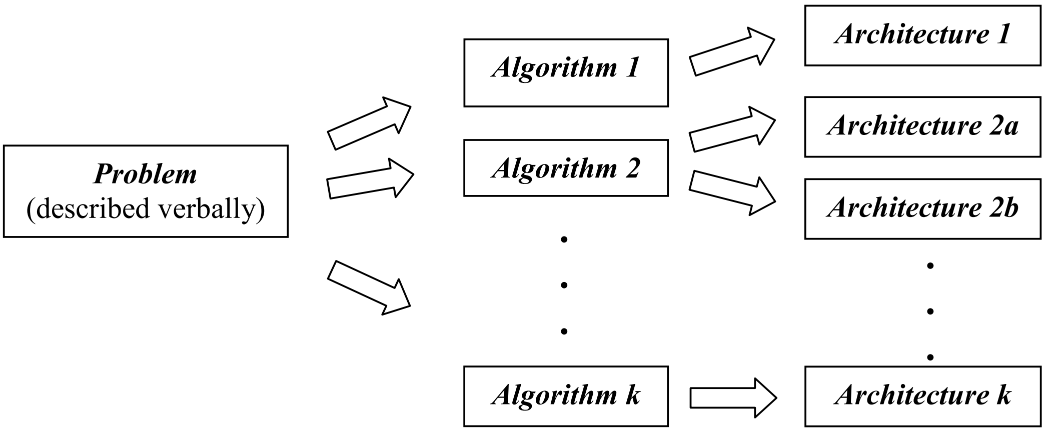


Figure 13. Methodology of special purpose processor architecture selection

The first step of the method is the comprehensive analysis of the parameter set. Fortunately, in many cases it immediately leads to a single possible architecture. If it does not lead to any architecture, in a second step, we have to seek for options, how to fulfill the original specification demands. If it leads to multiple architectures, a ranking is needed based on secondary parameters.

The three most important parameters are the ***frame-rate***, the ***resolution***, and their product, the minimal value of the ***pixel clock***.[1] In many cases, especially in challenging applications, these parameters determine the available solutions. Figure 14 shows frame-rate – resolution matrix. The matrix is divided into 16 segments, and each segment indicates the potential architectures that can operate in that particular parameter environment. The matrix shows the minimal pixel clock figures (red) in the grid points also.

In Figure 14, the pipe-line and the DSP can be positioned freely between frame-rate and resolution without constrains. Thus they appear everywhere, under a certain pixel clock rate. The digital coarse-grain sensor-processor arrays appear in the low resolution part (left column), while the analog (mixed-signal) fine-grain sensor-processor arrays appear in both the low and medium resolution columns.

The next important parameter is the ***latency***. Latency is critical when the vision device is in a control loop, because large delays might make the control loops instable. It is worth to distinguish three latency requirement regions:

- very low latency *(latency* <2ms; e.g. missile, UAV, high speed robot controlling);
- low latency (2ms < *latency* <50ms; e.g. robotic, automotive);
- high latency (50ms < *latency; e.g.* security, industrial quality check).

Latency has two components. The first is the readout time of the sensor, and the second is the completion of the processing on the entire frame. The readout time is negligible in the fine-grain mixed-signal architectures, since the analog sensor readout should be transferred to an analog memory through a fully parallel bus. The

[1] The minimal value of the pixel clock is equivalent to the product of the frame-rate and the number of pixels (resolution). If the image source is a sensor, the pixel clock of the processor is defined by the pixel clock of the sensor. Since there are short blank periods in the sensor readout protocol for synchronization purposes, the pixel clock is slightly higher then the minimal pixel clock even in those cases, when the integration is done parallel with the readout (CMOS or CCD rolling shutter mode). However, in low light applications, the integration time is much longer than the readout time. In these cases, the sensor pixel clock can be orders of magnitude higher than the minimal pixel clock.

readout time is also very small (~100μs) in the coarse-grain digital processor array, because there is an embedded AD converter array to do conversion in parallel. The DSPs and the pipe-line processor arrays use external image sensors, in which the readout time usually is in the millisecond range. Therefore, in case of very low latency requirements, the mixed-signal and the digital focal plane arrays can be used. (There are some ultra-high frame-rate sensors with high speed readout, which can be combined with pipe-line processors. However, these can be applied in very special applications only due to their high complexities and costs.)

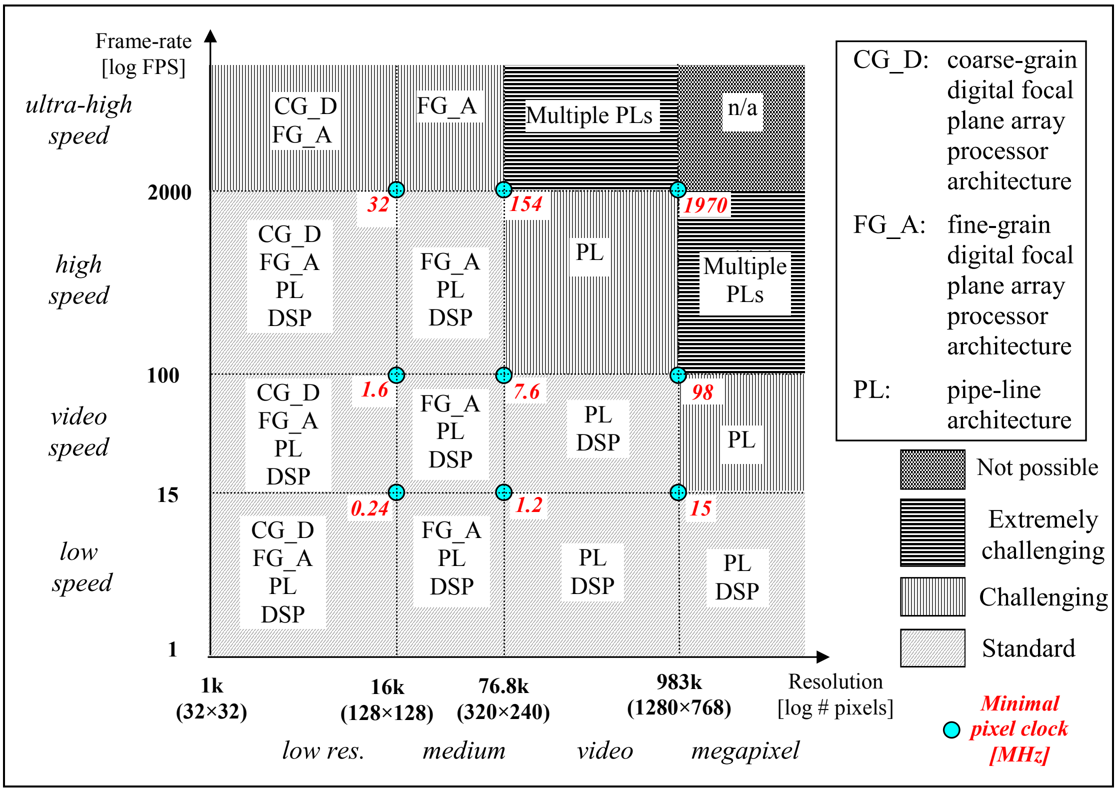


Figure 14. Feasible architectures in the frame-rate – resolution matrix

In the low latency category, those architectures can be used only, in which the sensor readout time plus the processing time is smaller than the latency requirements. In the high latency region, the latency does not mean any bottleneck.

The next descriptor of the algorithms is the ***computational demand***. It is a list of the applied operations. Using the execution time figures that we calculated for different operations on the examined architectures, we can simply calculate the total execution time. (In case of the pipe-line architecture the delay of the individual stages should be summed up.) The total processing time should satisfy the following two relations:

$$t_{total\_processing} < t_{latency} - t_{readout}$$

$$t_{total\_processing} < 1/frame\_rate$$

The last primary parameter is the ***program flow***. The array processors and the DSP are not sensitive for branches in program flow. However, the pipe-line architectures are challenged by the conditional program flow branches, because the branching should happen at the calculation of the first pixel of the frame, but at that time, the branching condition is not yet calculated. (The condition is calculated during the processing of the entire frame.) Since that, before the branching, the application of a frame-buffer is required, which generates significant hardware overhead and latency increase.

There are three secondary design parameters. The first is the ***power consumption***. Generally, the ASIC solutions need much less power than the FPGA or DSP solutions. The second is the physical size of the circuit. Smaller physical size can be achieved with sensor-processor arrays, because the combination of these two functionalities reduces the chip count. The third parameter is the cost. In case of low volume, the DSP is the cheapest, because the invested engineering cost is the smaller there. In case of medium volume, the FPGA is the most economical, while in case of high volume, the ASIC solutions are the cheapest.

6 CONCLUSIONS

We have categorized the 2D operators into 6 sets, based on their implementation methods on different image processing architectures. By using this categorization, the efficiency figures of 2D operators were calculated considering different architectures. This enabled us to compare the architectures, and formalize an efficient architecture selecting methodology for given algorithms. Moreover, we have measured, collected, or calculated some key parameters of existing implementations. Comparing the different architectures, we can draw the following conclusions:

- The computational speed on digital coarse-grain architectures is roughly the same as on fine-grain architectures (Figure 12). The accuracy of the digital one is better. However, the required silicon area is also larger (Table I).
- The analog/mixed signal fine-grain architecture can take advantage of utilizing various specific processing networks, such as the mean grid, diffusion grid, global OR grid, etc (Table II).
- In focal-plane sensor-processor application where the specification requires lower precision, the analog fine-grain implementations are more advantageous.
- In applications where high-precision calculation is required, the coarse-grain architecture is more advantageous.
- It is important to note that in the case of array processors, the speed up rate changes with the processor array size. In some cases the speed advantage is proportional to the number of the processors in the array (area active single step, and the front active content-dependent execution-sequence-variant operators), while in the rest of the cases, it is proportional with the number of the processors located in one row/column (Table II).
- As it is shown in Table III, the GOPs/W figure of the studied topographic many-core architectures are orders of magnitude better than the single or many core high-end processors used nowadays in PCs and servers,. This makes any of those much more suitable for embedded mobile applications, compared to a DSP or a RISC processor.

REFERENCES

[1] L.O. Chua and L. Yang, “Cellular Neural Networks: Theory and Applications”, IEEE Transactions on Circuits and Systems, vol. 35, no. 10, October 1988, pp. 1257-1290, 1988.

[2] L.O. Chua and T. Roska, “The CNN Paradigm”, IEEE Transactions on Circuits and Systems - I, vol. 40, no. 3, March 1993, pp. 147-156, 1993.

[3] T. Roska and L.O. Chua, “The CNN Universal Machine: An Analogic Array Computer”, IEEE Transactions on Circuits and Systems - II, vol. 40, March 1993, pp. 163-173. 1993.

[4] S. Espejo, R. Carmona, R. Domínguez-Castro and A. Rodríguez-Vázquez “A VLSI-Oriented Continuous-Time CNN Model”, International Journal of Circuit Theory and Applications, Vol. 24, pp. 341-356, May-June 1996.

[5] Cs. Rekeczky and L. O. Chua, “Computing with Front Propagation: Active Contour and Skeleton Models in Continuous-time CNN”, Journal of VLSI Signal Processing Systems, Vol. 23, No. 2/3, pp. 373-402, November-December 1999.

[6] J.M.Cruz, L.O.Chua, and T.Roska, “A Fast, Complex and Efficient Test Implementation of the CNN Universal Machine”, Proc. of the third IEEE Int. Workshop on Cellular Neural Networks and their Application (CNNA-94), pp. 61-66, Rome Dec. 1994.

[7] H.Harrer, J.A.Nossek, T.Roska, L.O.Chua, “A Current-mode DTCNN Universal Chip”, Proc. of IEEE Intl. Symposium on Circuits and Systems, pp135-138, 1994.

[8] A. Paasio, A. Dawindzuk, K. Halonen, V. Porra, “Minimum Size 0.5 Micron CMOS Programmable 48x48 CNN Test Chip” European Conference on Circuit Theory and Design, Budapest, pp. 154-15, 1997.

[9] Gustavo Liñan Cembrano, Ángel Rodríguez-Vázquez, Servando Espejo-Meana, Rafael Domínguez-Castro: ACE16k: A 128x128 Focal Plane Analog Processor with Digital I/O. Int. J. Neural Syst. 13(6): 427-434 (2003)

[10] S. Espejo, R. Carmona, R. Domingúez-Castro, and A. Rodrigúez-Vázquez, "CNN Universal Chip in CMOS Technology", Int. J. of Circuit Theory & Appl., Vol. 24, pp. 93-111, 1996

[11] S. Espejo, R. Carmona, R. Domínguez-Castro and A. Rodríguez-Vázquez “A VLSI-Oriented Continuous-Time CNN Model”, International Journal of Circuit Theory and Applications, Vol. 24, pp. 341-356, May-June 1996.

[12] P.Dudek "An asynchronous cellular logic network for trigger-wave image processing on fine-grain massively parallel arrays", IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing,. 53 (5): pp. 354-358, 2006.

[13] A. Lopich, P. Dudek, “Implementation of an Asynchronous Cellular Logic Network As a Co-Processor for a General-Purpose Massively Parallel Array”, ECCTD 2007, Seville, Spain.

[14] A. Lopich, P. Dudek., " Architecture of asynchronous cellular processor array for image skeletonization", Circuit Theory and Design, Volume: 3, On page(s): 81-84, 2005.

[15] P.Dudek and S.J.Carey, "A General-Purpose 128x128 SIMD Processor Array with Integrated Image Sensor", Electronics Letters, vol.42, no.12, pp.678-679, June 2006

[16] Z. Nagy, P. Szolgay "Configurable Multi-Layer CNN-UM Emulator on FPGA" IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, Vol. 50, pp. 774-778, 2003

[17] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy “Introduction to the Cell multiprocessor” IBM J. Res. & Dev. vol. 49 no. 4/5 July/September 2005

[18] www.ti.com

[19] 176x144 Q-Eye chip, www.anafocus.com

[20] Video security application: http://www.objectvideo.com/ efficient

[21] Cs. Rekeczky, J. Mallett, A. Zarandy, „Security Video Analitics on Xilinx Spartan -3A DSP”, Xcell Journal, Issue 66, fourth quarter 2008, pp: 28-32

[22] Á. Zarándy, "The Art of CNN Template Design", Int. J. Circuit Theory and Applications - Special Issue: Theory, Design and Applications of Cellular Neural Networks: Part II: Design and Applications, (CTA Special Issue - II), Vol.17, No.1, pp.5-24, 1999

[23] Á. Zarándy, P. Keresztes, T. Roska, and P. Szolgay, "CASTLE: An emulated digital architecture; design issues, new results", Proceedings of 5th IEEE International Conference on Electronics, Circuits and Systems, (ICECS'98), Vol. 1, pp. 199-202, Lisboa, 1998

[24] P. Keresztes, Á. Zarándy, T. Roska, P. Szolgay, T. Bezák, T. Hídvégi, P. Jónás, A. Katona, "An emulated digital CNN implementation", Journal of VLSI Signal Processing Special Issue: Spatiotemporal Signal Processing with Analogic CNN Visual Microprocessors, (JVSP Special Issue), Kluwer, 1999 November-December

[25] P. Földesy, Á. Zarándy, Cs. Rekeczky, and T. Roska „Configurable 3D integrated focal-plane sensor-processor array architecture”, Int. J. Circuit Theory and Applications (CTA), pp: 573-588, 2008

[26] L.O. Chua, T. Roska, T. Kozek, Á. Zarándy “CNN Universal Chips Crank up the Computing Power”, IEEE Circuits and Devices, July 1996, pp. 18-28, 1996.

[27] T. Roska, L. Kék, L. Nemes, Á. Zarándy, M. Brendel and P. Szolgay, "CNN Software Library (Templates and Algorithms) Version 7.2", (DNS-1-1998), Budapest, MTA SZTAKI, 1998, http://cnn-technology.itk.ppke.hu/Library_v2.1b.pdf

[28] http://www.xilinx.com/support/documentation/data_sheets/ds706.pdf