AUKE: Automatic Kernel Code Generation for an Analogue SIMD Focal-Plane Sensor-Processor Array

THOMAS DEBRUNNER, Imperial College London, UK
SAJAD SAEEDI, Imperial College London, UK
PAUL H J KELLY, Imperial College London, UK

Focal-plane Sensor-Processor Arrays (FPSPs) are new imaging devices with parallel Single Instruction Multiple Data (SIMD) computational capabilities built into every pixel. Compared to traditional imaging devices, FPSPs allow for massive pixel-parallel execution of image processing algorithms. This enables the application of certain algorithms at extreme frame rates ($> 10,000$ frames per second). By performing some early-stage processing in-situ, systems incorporating FPSPs can consume less power compared to conventional approaches using standard digital cameras. In this paper we explore code generation for an FPSP whose 256×256 processors operate on analogue signal data, leading to further opportunities for power reduction — and additional code synthesis challenges.

While rudimentary image processing algorithms have been demonstrated on FPSPs before, progress with higher level computer vision algorithms has been sparse due to the unique architecture and limits of the devices. This paper presents a code generator for convolution filters for the SCAMP-5 FPSP, with applications in many high level tasks such as convolutional neural networks, pose estimation etc. The SCAMP-5 FPSP has no effective multiply operator. Convolutions have to be implemented through sequences of more primitive operations such as additions, subtractions and multiplications/divisions by two. We present a code generation algorithm to optimise convolutions by identifying common factors in the different weights and by determining an optimised pattern of pixel-to-pixel data movements to exploit them. We present evaluation in terms of both speed and energy consumption for a suite of well-known convolution filters. Furthermore, an application of the method is shown by the implementation of a Viola-Jones face detection algorithm.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**;

Additional Key Words and Phrases: Focal-plane Sensor-Processor Arrays, Kernel Filtering, Automatic Code Generation

ACM Reference Format:
Thomas Debrunner, Sajad Saeedi, and Paul H J Kelly. 2018. AUKE: Automatic Kernel Code Generation for an Analogue SIMD Focal-Plane Sensor-Processor Array. *ACM Trans. Arch. Code Optim.* 1, 1, Article 1 (December 2018), 25 pages. https://doi.org/10.1145/3291055

1 INTRODUCTION

Conventional image processing systems consist of multiple distinct hardware components. The image gets captured by a digital camera before being transferred over a bus to the system's main memory. After that, image processing algorithms are applied on a CPU, GPU, or FPGA. While

Authors' addresses: Thomas Debrunner, Imperial College London, Kensington, LONDON, SW7 2AZ, UK, thomas.debrunner16@imperial.ac.uk; Sajad Saeedi, Imperial College London, Kensington, LONDON, SW7 2AZ, UK, s.saeedi@imperial.ac.uk; Paul H J Kelly, Imperial College London, Kensington, LONDON, SW7 2AZ, UK, p.kelly@imperial.ac.uk.

image capture and processing systems are advancing, this paradigm inherently forces a compromise between frame rate and power consumption. Overcoming this limitation has the potential to open up new applications and capabilities in always-on mobile devices, wearables, autonomous navigation etc.

Contemporary hardware bus and memory systems are easily capable of performing frame-by-frame data transfer at common frame rates, but, for example, achieving 100,000fps on a 256×256 sensor with 8 bits of resolution per pixel would require a bus capable of a data rate of 6.5 GB/s [7]. Simple image processing algorithms like filtering can generally be done in real-time. However, more complex algorithms like object detection, convolutional neural networks, or simultaneous localization and mapping (SLAM) not always scale to real-time performance or require complex, energy intensive hardware. Additionally, the analogue readout in most CMOS image sensors account for 50%-70% of the sensor energy consumption, which is due to prioritizing of having high-fidelity measurements and also the readout being performed on all pixels [28]. These energy overheads pose difficulties to implement such systems in embedded, energy-constraint environments. Focal-plane processor devices [50], such as ACE400 [14], ACE16K [30], MIPA4K, [37], and the various iterations of the SCAMP (SIMD current-mode analog matrix processor) chip ([18], [16], [15], [8]) try to address this problem by shifting image processing from a dedicated processing unit onto the image sensor's focal plane, by adding processing elements to each pixel. Unlike traditional computers, most of the chips mentioned above, including the SCAMP-5 [8] which is the focus of the paper, store the captured and intermediate images as analogue instead of digital values. Arithmetic operations are carried out directly on physical currents. The reasoning behind this is an effort to increase speed while keeping area and power dissipation at a minimum [17]. Analogue technology has also been used recently by IBM to accelerate training of deep neural networks [2]. At the very end of the processing spectrum, it has been shown that the inference in neural networks can also be accelerated significantly to the speed light, by utilizing an all-optical diffractive neural networks , where the network is 3D-printed [29]. Also, angle sensitive pixels are used in ASP Vision [10] to perform the first layer of convolution by hardware, when the light is captured. Similarly in ISAAC [42] and DaDianNao [12], specialized hardware is designed to accelerated inference.

Analogue and digital FPSPs can be included in real-world computer vision pipelines that require high frame rates. They can be used in conjunction with other technologies such as GPUs/CPUs/FPGAs to perform some parts of the processing pipeline. In general, there is a cost associated with the data transfers to and from the FPSP device. In the case of in-sensor devices, such as the SCAMP, the device acts as a camera itself. Data processing is immediately carried out on the image sensor before the processed image, or a lower dimensional representation, is transferred to the host. In this case, the cost associated with data transfer is at most as high as with traditional cameras. For instance, on the studied device, there is an analogue global summation functionality that can sum up tiles of 32×32 pixels in a single machine cycle. After that, only the result has to be digitised and sent back to the host, reducing data transfers. Operations like this occur frequently in computer vision pipelines. Another example is the average pooling in neural networks, where only the results need to be sent to a CPU or GPU.

With the advantages such as high effective frame rate and low power consumption, the major disadvantage in devices such as SCAMP-5 is difficulty in coding, especially for high level computer vision algorithms. For example, only seven registers are available for each pixel and there is no hardware support for multiplication.

We introduce a novel approach that allows code generation for convolution and filtering applications on FPSPs, as an exploration to map the capabilities of pixel parallel SIMD as a strategy for local-plane sensor processing, and to guide research and development of future FPSP device architectures. Although results are reported for the SCAMP chip, the method is applicable for any pixel-parallel SIMD processing device with local connectivity. A use case of the proposed code generation method has been demonstrated by using our method to implement a Viola-Jones face recognition program [45]. Analogue FSPSs are inherently limited by the number of the operations they apply to data, since every operation introduces noise to the values. This holds true for all analogue devices, meaning that even future versions of SCAMP will need a way to generate programs as short as possible in order to minimise loss. While digital FPSPs do not suffer from the accuracy problem, they can still benefit from shorter programs. Furthermore, our method allows the approximation of multiplications on a device without a multiplication unit. It is expected that the method can help to reduce the number of expensive multiplications in favour of cheaper arithmetic operations on devices that do support multiplication. Moreover, this method can also be applied to filtering in VLSI circuits and in super-low-power digital microcontrollers, where similar issues are present [1].

A closely related work to this paper is the RedEye project [28], an analogue ConvNet image sensor architecture. Both our work and RedEye take advantage of specialised hardware to perform early vision processing for applications that rely on convolution. The objectives in both works are to perform high-speed convolution at low power for applications that need always-on vision capability. RedEye has specifically been designed for operations needed in deep convolutional neural networks, while our work presents a code generation method for performing convolutions on general purpose pixel-parallel processors without specific considerations for neural networks.

1.1 Background

FPSP devices have been used for several image processing applications. But these applications are limited due to coding challenges [18]. Martel et al. developed real-time high dynamic range (HDR) imaging [33], which can be used for robust navigation of autonomous cars while driving under extremely variable lighting conditions. Bose et al. propose a four degrees-of-freedom visual odometry algorithm [5], which can be used in high-speed drones. Martel et al. also present a method to jointly estimate visual quantities such as optical flow and image gradient on the chip [32]. However, current applications are restricted due to the limited hardware resources and parallel coding challenges.

From computational perspective, in computer vision, stencil and convolution reduction are the main component of many pipelines. Thus optimization of memory efficiency of convolutions [11] or their algorithmic complexity, as done in [23] using the Winograd transform [49], can improve the overall efficiency of the pipelines. Several works on domain-specific languages (DSLs) have been introduced to optimize the image processing pipelines on modern systems such as GPUs. There is a large and successful body of work on tiling optimisations for locality and parallelism in MIMD implementations of convolutional filters and finite-difference stencils; notable examples include Halide [38], Polymage [35], OPS [40] and ExaStencils [25]. For instance, Halide presents a new language and compiler which allows developers to write fast image processing pipelines [38]. Halide separates the definition of the algorithm in the pipeline from the concerns of the optimization, such as SIMD vectorization, tiling, or parallelization. PolyMage is another DSL where complex fusion, tiling, and storage optimization is performed automatically [35]. These works are primarily focused on loop fusion and tiling - that is, finding a schedule for execution of local convolution operators that makes effective use of caches and multicore processors. Our work addresses different concerns than Halide or PolyMage, because of the differences in the hardware. Unlike GPUs, FPSPs have limited number of registers, support limited mathematical operations, and also compute in analogue which introduces computational noise. While for Halide or PolyMage various mathematical operations are easily accessible on GPUs, we have to apply approximation techniques on every operand to perform the desired mathematical operations. Our paper presents a different approach, targeting single convolutional operators, on SIMD hardware with no cache or scratchpad memory. It might be possible to combine these approaches, if we were to consider rather different processor designs, but that lies far beyond the scope of the current work. There has been interesting related work on retiming evaluation of convolution filters to reduce register pressure [39, 43], and, more generally, exploiting distributivity to maximise code motion opportunities [31].

A core concept of our method is the idea to approximate multiplications using a Canonical Signed Digit (CSD) [3] representation. In binary representation the probability of a bit being zero is 50%. For CSD representation however, the probability of a bit being zero tends to almost 66%, as shown in [41]. Many contributions have been made in minimizing add/subtract networks for constant multiplication, such as [21] and [36] which introduce subexpression sharing to CSD networks to minimize hardware resources. Graph construction based methods for the construction of the multipliers used in digital filters have been shown in [6] and [13]. These methods explore the search space by trying different combinations of previously-computed sub-results, either exhaustively or with the help of heuristics. A generalization of these graph based methods has been presented in [46]. Similarly, a method was presented in [4] to find the optimal sequences of basic operations used in matrix multiplication in Toom-Cook methods. The sequence is determined via an optimised research on graphs. Significant position encoding (SPE) is another technique that uses a ternary number system with -1, 0, and 1 for efficient implementation of forward inference in neural networks [44]. SPE uses fewer bits for data representation, and still it is able to achieve high accuracy of classification, thanks to inherent robustness of neural networks to noisy signal and weight values. The algorithm presented in this paper tackles a similar problem. In contrast to solutions aimed at integrated circuit design, where multiplications and divisions by factor 2 are free, they do incur a cost on the focal-plane processor. This renders previous approaches unsuitable for the problem at hand.

1.2 Contributions

In the context of this background and related work, we summarize the main contributions offered in this paper:

- We propose a novel method to automatically generate code for kernel filtering. Our approach is, to the best of authors' knowledge, the first to introduce automatic optimized kernel code generation for FPSPs. Our algorithm takes a filtering kernel of given size and values as input, and generates the SCAMP operations, such as addition, division by two and shifting in four directions, to apply the kernel to the incoming images, while optimizing to minimize the number of operations. This is done by avoiding redundant computations, commonly found in kernel convolutions, especially when implemented multiplier-free using the CSD technique.
- We present a novel algorithm, which we call, reverse splitting, based on a representation of the state of the search for an optimum kernel evaluation strategy as a multiset of partial value representatives. This is an advance on prior work (notably [13]), in the possibility to also assess the transformation costs associated with shifting values both locally, as well as in scale.
- We present experimental evaluation of the effectiveness of the resulting FPSP code on a suite of standard kernel filter examples, compared with the best available OpenCV implementations for CPUs and GPUs. Our proposed solution achieves estimated execution times of 1.4-12.5μs per 256×256 frame, with estimated energy consumption of 2-15μJ — in summary, up to 200 times less energy per frame than a CPU solution using the state-of-the-art OpenCV library, while achieving up to ten times the frame rate.
- We demonstrate the approach with the Viola-Jones face detection algorithm (whose box kernels are designed for fast sequential execution using integral images). Although CPU

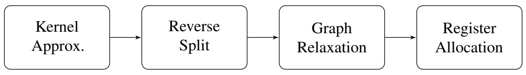


Fig. 1. The code generation is composed of four steps. In 'Kernel Approximation', the kernel values are approximated to meet the FPSP hardware limitations. In 'Reverse Split', a computational graph is obtained that represents a plan to compute the kernel. The graph reuses intermediate results. In 'Graph Relaxation', the graph weights are rebalanced to reduce the number of total instructions. Finally, in 'Register Allocation', using a graph coloring algorithm, the physical registers are allocated to the graph.

implementations are faster, our FPSP implementation requires almost an order of magnitude less energy per frame.

An implementation of the algorithm in Python has been open-sourced[1]. A video description of the paper is available online[2].

The rest of the paper is organized as follows. Sections 2 and 3 present the proposed method for filter kernel code generation. Section 4 demonstrates experimental results, and 5 summarizes the paper.

2 PROBLEM FORMULATION AND NOTATIONS

The automatic filter code generation is performed in four steps. Fig. 1 shows the outline of the proposed method. In a first step, the coefficients of the input filter get approximated to fit to the hardware capabilities of the device. This step is outlined in **Section 2.1**. The filter then gets transformed into a multiset notation to be decomposed by the reverse split algorithm. The reverse split algorithm, described in **Section 3.1**, tries to recursively decompose the filters multiset into additions of subsets. The algorithm continues, until it reaches the multiset notation of the goal represented by the initial sum, the state of the image before any filter application. This way, by trying to reach the initial sum from the final sum, it obtains a plan on how to compose the final sum from the initial sum. In a third step, it reduces the amount of necessary instructions by exploiting equivalence transforms on the computational graph. This operation is described in **Section 3.2**. In a last step, a graph coloring algorithm computes a valid register allocation for the computation graph. This is described in **Section 3.3**. The physical register allocation yields a program that can be executed on the FPSP without any further modifications.

2.1 Approximation Formulation

This section presents numerical value approximation and the filter kernel approximation. These approximations are necessary to map the problem to the device capabilities.

2.1.1 *Value Approximation.*

The studied device does not support multiplications by arbitrary constants. Multiplications by integers can be simulated by adding values to themselves. Divisions by integers can be performed by evenly splitting current into multiple registers at the same time [17]. This division technique requires at least $n + 1$ available registers to perform a division by n. The chip only has 7 analogue registers which we can utilize most effectively by using them to store intermediate results. To save registers for this purpose and for reasons of simplicity, we restrict any division to be a division by two. This is similar to the problem faced in integrated circuit design, where in the absence of multipliers and adders only divisions and multiplications by two are possible.

Kernel convolution is a weighted sum of the pixel values in the vicinity of a centre pixel. In order to perform this task on a chip with the stated limitations in place, every coefficient in the filter kernel is represented in truncated, fixed point Canonical Signed Digit (CSD) form [41]. Let $\alpha \in \mathbb{R}$ be an

[1]https://github.com/najiji/auto_code_cpa
[2]https://youtu.be/NILZILnEevA

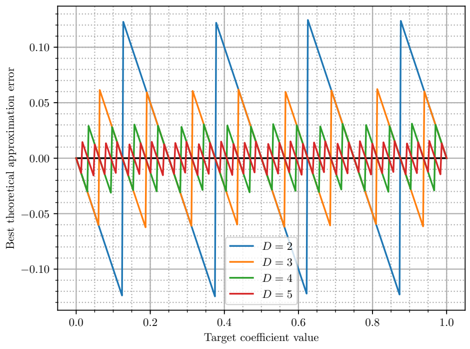


Fig. 2. Theoretical absolute error introduced by the approximation of filter values (Eq. (1)) for different approximation depths. The induced errors get smaller with greater approximation depth.

arbitrary value in a convolution kernel and I be the intensity value of a pixel. The product $\alpha \cdot I$ is approximately written in CSD form as follows:

$$\alpha \cdot I \approx \sum_{d=-\lceil \log_2(|\alpha|) \rceil}^{D} a_d \cdot \frac{1}{2^d} \cdot I, \qquad a_d \in \{-1, 0, 1\} \tag{1}$$

The coefficients, a_d, define if we subtract, ignore or add a certain scaling. D is the depth of the approximation. The lower summation bound $\lceil \log_2(|\alpha|) \rceil$ guarantees that there are big enough coefficients for the sum to converge.

Example 1. *The product* $0.6 \cdot I$ *can be approximated as follows (*$D = 3$*):*

$$0.6 \cdot I \approx \sum_{d=-\lceil -0.73 \rceil}^{3} a_d \cdot \frac{1}{2^d} = (a_0 2^0 + a_1 2^{-1} + a_2 2^{-2} + a_3 2^{-3})I \tag{2}$$

$$= (0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3})I = 0.625I \tag{3}$$

Based on Eq. (1), an optimal set of a_d coefficients given a desired α and depth is computed by an algorithm that iterates over the sequence $\{s_i\} = \{2^{\lceil \log_2(|\alpha|) \rceil}, \cdots 2, 1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \cdots \frac{1}{2^D}\}$, deciding in every iteration whether it is beneficial to use the coefficient positively, negatively or not at all. This process introduces an approximation error to the computation result. Intuitively, approximating to greater depths yields better results. The theoretically induced absolute error on the coefficient for various approximation depths is illustrated in Fig. 2. The effects of the approximation to the resulting image after filter application is shown in Fig. 6.

2.1.2 Filter Representation. In this section, we reuse the approximation Eq. (1), to represent a convolutional filter as counts of small, equal scalings of the pixel values. This description as accumulates of fractions of same size, allows our algorithm to find reuse opportunities between neighbouring cells.

We apply a convolutional kernel $K \in \mathbb{R}^{m \times n}$ to a large enough image at pixel (i, j). $I_{i,j}$ is the original intensity of this pixel. We define R to be the set of coordinates u, v in the neighbourhood of this pixel, extending to the edge of the filter. An example is shown in Fig. 3. Specifically that is for n odd: $u \in \{-\frac{n-1}{2} \dots \frac{n-1}{2}\}$, for m odd: $v \in \{-\frac{m-1}{2} \dots \frac{m-1}{2}\}$. For n even: $u \in \{-\frac{n-2}{2} \dots \frac{n}{2}\}$, for m even: $v \in \{-\frac{m-2}{2} \dots \frac{m}{2}\}$.

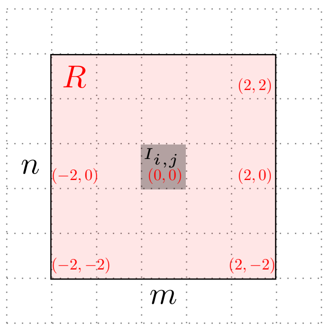


Fig. 3. The definition of the R set. The R set contains the coordinates of all the pixels of a filter kernel, relative to its centre. The coordinates are indicated in red. They are independent of where in the image the filter gets applied to. The shown filter is a 5×5 filter, a general filter can have any rectangular shape.

Applying the kernel K on the image I at position i, j is then expressed as:

$$I'_{i,j} = \sum_{u,v \in R} K_{u,v} \cdot I_{i+u,j+v}. \tag{4}$$

The kernel coefficients $K_{u,v}$ are replaced by the approximation formulation Eq. (1). Then Eq. (4) becomes: (D being the approximation depth)

$$I'_{i,j} = \sum_{u,v \in R} \left(\sum_{d=-\lceil \log_2(|K_{u,v}|) \rceil}^{D} a_{d,u,v} \cdot 2^{-d} \cdot I_{i+u,j+v} \right) \tag{5}$$

With $2^{-d} = 2^{-d+D} \cdot 2^{-D}$ we can define the function $N(u, v)$ as:

$$I'_{i,j} = \sum_{u,v \in R} \left(\underbrace{\sum_{d=-\lceil \log_2(|K_{u,v}|) \rceil}^{D} a_{d,u,v} \cdot 2^{-d+D}}_{:=N(u,v)} \right) \cdot 2^{-D} \cdot I_{i+u,j+v} \tag{6}$$

$$= \sum_{u,v \in R} N(u, v) \cdot 2^{-D} I_{i+u,j+v} \tag{7}$$

$2^{-D} I$ is the smallest scaling of an input value used in the approximation. $N(u, v) \in \mathbb{Z}$ then gives the count of these "smallest scalings" for every coordinate $u, v \in R$. An approximated filter can therefore be fully described by the function $N(u, v)$ together with the approximation depth D.

EXAMPLE 2. *Consider the following* 1×3 *filter*

$$K = \begin{bmatrix} 0.128 & 0.49 & 0.13 \end{bmatrix} \approx \frac{1}{8} \begin{bmatrix} 1 & 4 & 1 \end{bmatrix}, \tag{8}$$

where the kernel has been approximated by depth $D = 3$. For this kernel, $N(u, v)$s are:

$$N(-1,0) = \sum_{d=0}^{3} a_{d,-1,0} \cdot 2^{-d+3} = (0)2^3 + (0)2^2 + (0)2^1 + (1)2^0 = 1 \tag{9}$$

$$N(0,0) = \sum_{d=-2}^{3} a_{d,0,0} \cdot 2^{-d+3} = 4 \tag{10}$$

$$N(1,0) = \sum_{d=-2}^{3} a_{d,1,0} \cdot 2^{-d+3} = 1 \tag{11}$$

$$\tag{12}$$

2.2 Notation, Definitions

This section introduces notation and definitions used throughout the paper. Filters and states are represented as multisets. A multiset behaves like a normal set with the alteration that every element can occur multiple times, both positively as well as negatively. A multiset is denoted as $S = \langle A, m_A \rangle$, where A is a classical set of all the items that may occur in S. m_A is a function $A \longrightarrow \mathbb{Z}$ that assigns the multiplicity to every item in A. This representation allows a straight forward mapping of hardware instructions to operations on multisets.

2.2.1 Filters as multisets.

We represent a filter in terms of a multiset of so called **Partial Value Representatives (PVRs)**. We call this multiset then a **Sum of Partial Value Representatives (SOPVR)**. A PVR represents a "smallest scaling" (2^{-D}) of the pixel value at a certain coordinate location. Coordinates are according to the definition of R (See Fig. 3).

EXAMPLE 3. *The following filter K and the (multi)set (SOPVR) FS represent the same filter. D is chosen as 3 in this example, this defines the counts of PVRs in FS. Correspondences between K and FS have been highlighted in color. The number of PVRs per coordinate corresponds to the unary representation of the integer in the filter.*

$$K = \frac{1}{8}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \qquad FS = \left\{ \begin{array}{llll} (-1,1), & (0,1), & (0,1), & (1,1) \\ (-1,0), & (-1,0), & (0,0), & (0,0) \\ (0,0), & (0,0), & (1,0), & (1,0) \\ (-1,-1), & (0,-1), & (0,-1), & (1,-1) \end{array} \right\} \tag{13}$$

DEFINITION 1. *Let a **Partial Value Representative (PVR)**, denoted as (a_u, a_v), be a representation of the pixel value at coordinates a_u, a_v scaled by 2^{-D} (the smallest scaling). a_u and a_v are coordinates relative to the center of the filter (See Fig. 3). A PVR corresponds to the factor $2^{-D} I_{i+u,j+v}$ in Eq. 6.*

DEFINITION 2. *Let a **Sum of Partial Value Representatives (SOPVR)** be a multiset $S = \langle R, m_s \rangle$. R is the set of all possible coordinates (u, v) (See Fig. 3). m_s is the multiplicity function assigning an integer count to every item in R. A PVR $x = (u, v)$ is said to be included iff $|m_s(x)| > 0$. A SOPVR can be represented as a filter kernel and vice versa. To obtain a SOPVR from standard filter notation, one has to take $N(u, v)$ (as defined in Eq. 6) as the multiplicity function. The resulting set has exactly $N(u, v)$ PVRs to all coordinates $(u, v) \in R$. The number of PVRs at every coordinate is determined by the unary representation of the integer.*

DEFINITION 3. *The **final sum (FS)** SOPVR is the target filter kernel approximation in SOPVR notation.*

$$FS = \langle R, m_{FS} \rangle \qquad m_{FS}(u, v) = N(u, v) \quad \forall (u, v) \in R \tag{14}$$

The multiplicity function m_{FS} is given by the N function defined in Eq. (6). The final sum represents the value we desire to be present in a register after a filter application.

DEFINITION 4. *The **initial sum (IS)** SOPVR is the identity filter in SOPVR notation.*

$$IS = \langle R, m_{IS} \rangle \qquad m_{IS}(u,v) = \begin{cases} 2^D & \text{if } (u,v) = (0,0) \\ 0 & \text{otherwise} \end{cases} \qquad \forall (u,v) \in R \tag{15}$$

The initial sum represents the value present in a register of the device prior to any operations. The set contains exactly 2^D PVRs at (0,0).

EXAMPLE 4. *Consider the following 1×3 filter*

$$K = \begin{bmatrix} \frac{1}{8} & \frac{4}{8} & \frac{1}{8} \end{bmatrix} = \frac{1}{8} \begin{bmatrix} 1 & 4 & 1 \end{bmatrix} \tag{16}$$

With depth $D = 3$, in our notation, every PVR (u,v) corresponds to $\frac{1}{8}$ of the original pixel value at (u,v), relative to the center of the filter. The final sum of this kernel is

$$FS = \left\{ \begin{array}{ccc} (-1,0), & (0,0), & (0,0), \\ (1,0), & (0,0), & (0,0) \end{array} \right\} \tag{17}$$

With the initial sum being (representing the scaled identity kernel: $\frac{1}{8}\begin{bmatrix} 0 & 8 & 0 \end{bmatrix}$)

$$IS = \left\{ \begin{array}{cccc} (0,0), & (0,0), & (0,0), & (0,0) \\ (0,0), & (0,0), & (0,0), & (0,0) \end{array} \right\} \tag{18}$$

While the SOPVRs represent filter kernels, they also represent single values present in FPSP registers. For example, prior to performing any operations, the value present in the source register corresponds to the sum of all the PVRs in IS. After running the filter program, the value in the target register should equal the sum of all the PVRs in FS. The advantage of the SOPVR notation is that instructions on FPSP can be easily mapped to operations on SOPVRs.

2.2.2 Communication operations. FPSP devices allow communication of every pixel with neighbouring pixels. At the same time, every pixel performs the same stream of operations on their local data.

EXAMPLE 5. *Assume a pixel contains a sum in a register that can be expressed as the SOPVR $\{(0,0),(0,1)\}$. This SOPVR represents the sum of the local input value and an equal contribution of the input value from the pixel above. This contribution has been obtained via a local communication and an addition in the past. Since all pixels have performed the same operations on their data in the past, the pixel to the right must have the SOPVR $\{(1,0),(1,1)\}$ in the same register (relative to the local coordinate system). We can copy this SOPVR from the pixel to the right using local communications. The existence of $\{(1,0),(1,1)\}$ in the pixel to the right requires the existence of SOPVR $\{(0,0),(0,1)\}$ in the local pixel. Because of this we can formulate communication as a purely local operation of **transforming** $\{(0,0),(0,1)\} \rightarrow \{(1,0),(1,1)\}$. Transformation by any integer distance both horizontally and vertically are possible by communicating multiple times, handing the values over.*

Let F and G be two SOPVRs. $\text{supp}(F)$ is the *support* of F (that is, the set of elements that exist in F), $\text{card}(F)$ is the (absolute) *cardinality* of F (the sum of positive and negative multiplicities of all the elements of F).

DEFINITION 5. *G and F can be **transformed by shift** into each other iff there is an offset mapping from every PVR in F to a PVR in G. Formally, $\exists m,n \in \mathbb{Z}$ such that $\forall x \in \text{supp}(F), \quad \exists y \in \text{supp}(G)$ with $x_u + m = y_u, x_v + n = y_v$ and $m_F(x) = m_G(y)$ and vice versa.*

Example 6. *SOPVR $G = \{(0,0),(0,1)\}$ and $F = \{(5,-2),(5,-1)\}$ can be shifted to each other with $m = 5, n = -2$. SOPVRs $G = \{(0,0),(1,0)\}$ and $F = \{(0,-1),(1,-2)\}$ cannot be shifted into each other, as there is no m, n to shift G to F*

2.2.3 Negation Operations. A local inversion of a value on the FPSP results in a sign change for all the multiplicities of PVRs for the value's SOPVR.

Definition 6. *F and G can be* ***transformed by negation*** *into each other iff $m_F(x) = -m_G(x) \forall x \in R$*

Example 7. *The negation of $G = \{(1,-2),(0,1)\}$ is $F = \{-(1,-2),-(0,1)\}$*

2.2.4 Scaling Operations. A PVR represents a 2^{-D} contribution of the input value at offset (u,v). A division by 2 of a value on FPSP therefore results in halving the number of PVRs of each location in the SOPVR. A division can only be performed if every PVR in the SOPVR appears at least twice. Similarly, doubling a value results in doubling the number of PVRs for all coordinates.

Definition 7. *F can be* ***transformed by scale*** *into G iff $G \subset F$ and $m_F(x) = 2 \cdot m_G(x) \forall x \in R$ and vice versa.*

Example 8. *SOPVR $G = \left\{ \begin{array}{cc} (0,0), & (0,0) \\ -(0,1), & -(0,1) \end{array} \right\}$ can be transformed into $F = \{(0,0),-(0,1)\}$.*

2.2.5 Arithmetic Operations. The FPSP has multiple registers. The contents of each register can be expressed in terms of a SOPVR each. Addition and subtraction are pure pixel-local operations that yield new SOPVRs each. The resulting SOPVRs have the added / subtracted multiplicities for all PVRs.

Definition 8. *SOPVR $\langle R, m_A \rangle = F + G$ with $m_A(x) = m_F(x) + m_G(x) \forall x \in R$ is the* ***addition*** *of F and G. SOPVR $\langle R, m_S \rangle = F - G$ with $m_S(x) = m_F(x) - m_G(x) \forall x \in R$ is the* ***subtraction*** *of F and G.*

2.3 Graph Representation

It is useful to represent a sequence of operations as a computation graph. Every node of the graph represents a SOPVR. multiple adjacent transformations (shift, scale, negation) are condensed in a single edge. The symbols in **Table 1** are used to represent transformations and additions/subtractions in a graph.

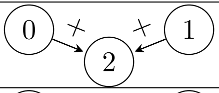
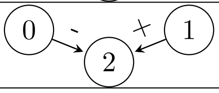
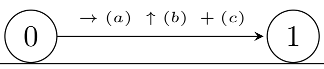
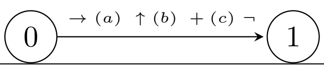
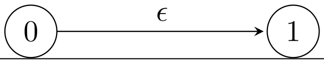
Operation	Representation	FPSP instructions
Addition		add
Subtraction		sub
Shift right by a, up by b, scale by c		north, east, south, west, div, add
Shift, scale and negate		north, east, south, west, div, neg
Empty shift (Copy)		copy, nop

Table 1. Graph representation styles. Each node number represents a SOPVR, and the edges show the transformations or add/subtraction operations between them.

3 CODE GENERATION

Any filter can be approximated to the approximation depth D. The initial sum SOPVR (IS), and the final sum SOPVR (FS) are uniquely given by the filter and D. IS is the source register state prior to performing any operations. FS is the target register state after kernel execution. A solution to the filter problem is given by a series of transformations and additions / subtractions that generate FS from IS while only holding as many intermediate SOPVRs at the same time as there are physical registers. This section presents an algorithm that achieves this. As shown in Fig. 1, the algorithm is composed of three modules:

- **Reverse Splitting**: generates transformations needed to produce FS from IS,
- **Graph Relaxation**: exploits the redundant transformations, using a computational graph, and
- **Register Allocation**: dedicates the resources to execute the algorithm.

3.1 Reverse Splitting

The Reverse Splitting algorithm generates a graph showing how to reach FS from IS by means of transformations and additions / subtractions while reusing intermediate results. FS is a heterogeneous SOPVR with PVRs from various coordinates while IS solely consists of PVRs at $(0,0)$. SOPVRs often share similar structure which allows them to be transformed into each other. A smart algorithm makes use of this fact in order to reuse sub results.

Example 9. *FS for a 3×3 Box filter with all ones, consists of 9 PVRs, one for every coordinate location in the kernel.* $(D = 0)$

$$K = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad FS = \left\{ \begin{array}{lll} (-1,-1), & (0,-1), & (1,-1), \\ (-1,0), & (0,0), & (1,0), \\ (-1,1), & (0,1), & (1,1) \end{array} \right\} \tag{19}$$

While the initial set IS just consists of a single PVR: $IS = \{\ (0,0)\ \}$. *A valid program is a sequence of transformations and additions that generates FS from IS. Suppose we have a program that would generate the following SOPVR (S) as a sub result, which corresponds to the middle row of the filter*

$$S = \left\{\ (-1,0), \quad (0,0), \quad (1,0)\ \right\} \subset FS \tag{20}$$

The PVRs that remain (the rows above and below) can then be grouped into the following two SOPVRs

$$FS - S = \underbrace{\left\{ \begin{array}{l} (-1,-1), \\ (0,-1), \\ (1,-1) \end{array} \right\}}_{R_0} + \underbrace{\left\{ \begin{array}{l} (-1,1), \\ (0,1), \\ (1,1) \end{array} \right\}}_{R_1} \tag{21}$$

The reason this particular decomposition is profitable is that SOPVRs R_0 and R_1 can be computed in a single shift instruction from S. A smart algorithm should compute the remaining parts R_0 and R_1 by reusing S. Note that in this case, the re-usability stems from the separability of the kernel.

3.1.1 *Search Space.*

The trivial case is when IS is directly transformable into FS. In that case, there is a known shortest program given by the transformation. In the general case however, IS and FS are not transformable into each other. In that case, FS is an addition of two sub-SOPVRs. Each new SOPVR is again either directly transformable from IS, from another given SOPVR, or is the sum of two new sub-SOPVRs. Splitting the multi-sets further, one eventually reaches a point where the only remaining SOPVRs is directly transformable from IS. The complete graph of all split possibilities contains every possibility of reaching FS by means of transformations and additions of sub-SOPVRs of FS, starting from IS.

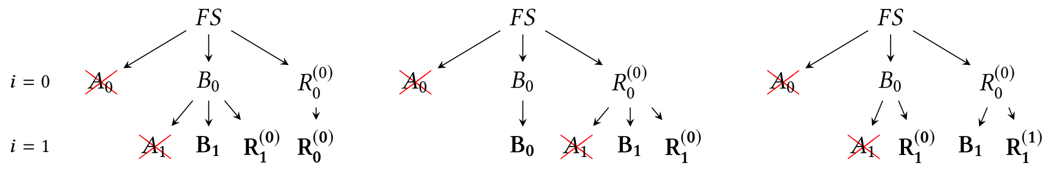


Fig. 4. The three possibilities to select SOPVRs to split of the reverse splitting algorithm at $i = 1$. The A SOPVRs can be generated from the B SOPVRs; therefore, it is sufficient to keep the B and R SOPVRs. The sets in bold are the ones to consider for further splitting.

3.1.2 Bounding the search space. The search space we have described does not include SOPVRs that are not direct sub-multisets of FS. It is assumed that it is never beneficial to generate an SOPVR that is not part of FS. Therefore, every SOPVR generated by the algorithm is a sub-multiset of FS. This assumption has the consequence that the algorithm finds graphs under the assumption that all shifts and scales incur the same cost. In fact some shifts and scales can be moved, and perhaps cancelled, through graph relaxation, as described in Section 3.2. Using this technique, previously excluded solutions are re-discovered. We have good reasons, and experimental results, to show that this is an effective heuristic. We do not currently have a proof of its optimality.

3.1.3 Algorithmic discovery. We propose an algorithm that is capable of exploring this search space efficiently, while providing results that reuse previous sub results. The algorithm starts with the FS SOPVR and splits it into three parts: $FS = A_0 + B_0 + R_0^{(0)}$, where we call A_i the **A-SOPVR**, B_i the **B-SOPVR** and $R_i^{(j)}$ the **Remainder-SOPVR** obtained at step i, for the $j - th$ SOPVR. The key idea of the algorithm is that we require A_i and B_i to be transformable into each other. Unless the SOPVR is directly transformable from IS, it is always possible to find such A and B, since every single PVR is transformable into every other single PVR. $R_i^{(j)}$ always contains the remainder that is not covered by A_i and B_i. The R SOPVR can also be empty, if a perfect split into A and B was made.

Since A_0 and B_0 are required to be transformable into each other, a program that can generate A_0 from B_0 is trivially given by the transformation sequence. Given this, it is sufficient to continue the search only for solutions for B_0 and $R_0^{(0)}$, as it is known how to generate A_0 then. This reduces the problem to finding a solution for B_0 and $R_0^{(0)}$. The algorithm executes recursively, eliminating A SOPVRs, until it reaches a single SOPVR that can be directly transformed from IS. In every step, the plan is extended with a record of the operations one needs to perform in order to arrive back at FG from the current step. For $i = 0$, the record contains the order to transform B_0 into A_0 and then add $A_0 + B_0 + R_0^{(0)} = FS$. The next step of the algorithm is then performed on B_0 and $R_0^{(0)}$.

For $i = 1$, we have two SOPVRs to split (B_0 and $R_0^{(0)}$). Since there is now more than one SOPVR to split, the algorithm has to evaluate all possible split pairs along the following choices of SOPVRs to split: (see Table 2 and Fig. 4):

Choice	Split	Retain for next step
B_0	$B_0 = A_1 + B_1 + R_1^{(0)}$	$\{R_0^{(0)}, B_1, R_1^{(0)}\}$
$R_0^{(0)}$	$R_0^{(0)} = A_1 + B_1 + R_1^{(0)}$	$\{B_0, B_1, R_1^{(0)}\}$
B_0 and $R_0^{(0)}$	$L_0 = A_1 + R_1^{(0)}$, $R_0^{(0)} = B_1 + R_1^{(1)}$	$\{R_1^{(0)}, B_1, R_1^{(1)}\}$

Table 2. The three possibilities to select SOPVRs to split at $i = 1$.

The number of possible choices to choose from depends on the number of available SOPVRs. Assume that at step i, there are n SOPVRs. Analogous to the case $i = 1$, we can either split any of the n SOPVRs individually or, we can choose any two of the SOPVRs to split together. This yields $n + \binom{n}{2}$ possibilities to choose from. The R SOPVRs can be empty; therefore, any split will yield either one more, one less, or the same number of SOPVRs for the next step. Note that for any step, the labelling (B, R) from the previous step is irrelevant. Every step treats the incoming SOPVRs equivalently.

The algorithm terminates, when it reaches a single SOPVR, that can be directly transformed from from IS. Once terminated, the obtained plan holds the sequence of instructions of splitting the final sum into sub-sums until only the initial value remains. To perform a filter application, one has to follow the plan in reverse order, with the splits becoming additions. To do so, one has to generate the A SOPVRs from the B SOPVRs, and add them together until FS is reached. Every SOPVR present at a certain time in the plan has to be held in a physical register. Since there is no possibility for register spilling, we have to ignore all branches of the search tree that yield more SOPVRs than available registers. This ensures, that any generated plan fits the hardware limits. The potentially quite large number of possibilities $n + \binom{n}{2}$ to choose SOPVRs to split is therefore not a problem in practice, as the studied FPSP device has only 7 registers.

Algorithm 1 is a pseudo code recursive implementation of the proposed method. It works exhaustively by evaluating all the possible splits in every step. At every step, the algorithm adds a step to the plan with the SOPVRs to have in this step, as well as the A, B split-pair required to reach it. Whenever the algorithm finds a solution, it adds it to the total list of solutions. In every step, the splitting will either yield one less, the same amount of, or one more SOPVR. All branches that require more SOPVRs than available registers are not explored, as such plans would not be feasible on hardware.

An improvement, not shown in Algorithm 1, is to cut branches as soon as they reach a worse cost as the best plan found so far. The instruction count is used as this metric. The function generatePairs generates an exhaustive list of all valid A and B SOPVRs that can be applied to the current SOPVRs. Another improvement not shown is to perform static heuristics on the split pairs, and explore the more promising pairs first. This is considered in the next section.

3.1.4 Non-Exhaustive search. While for small filters exhaustive search is possible, the number of possible splits gets very large for bigger problems. The result of the generatePairs function, as well as the number of available SOPVRs to split at each node, determines the branching factor of the search tree. To perform heuristic search, the generatePairs function is modified to not return an exhaustive list of possible pairs, but batches of likely good split pairs first. By finding good solutions early on, the algorithm can then cut off branches which exhibit a higher cost early on. This leads the algorithm to probe the likely more interesting subtree of every node first. A series of heuristics are applied to generate the best pairs first. The following ordering metric is applied:

$$b(A, B) = \frac{\text{card}(A)}{|d(A, B)|}, \tag{22}$$

$d(A, B)$ is the length of the transformation between A and B and card(A) the cardinality (the number of PVRs) of A. In every step of the plan, the algorithm eliminates an A SOPVR, leaving the PVRs in B and R. A short plan is desirable, therefore, the ordering Eq. (22) favours large A SOPVRs which eliminate more PVRs in the step. Upon execution of the program, A has to be created from B. This operation takes as many instructions as the transformation distance between the two SOPVRs. For a shorter program, it is desirable to have shorter transformation distances, which is why there is a trade-off between the size of A and the transformation distance. As another static evaluation,

ALGORITHM 1: Reverse split algorithm. The $<<$ operation signifies an "append" operation. generatePairs generates all the split-pairs that can be applied to the current SOPVRs. isTransformable(a, b) returns true, iff a and b are transformable into each other. At every recursion, as well as in the base case, a step of the form (SOPVRs, ($B \mapsto A$)) is appended to the current plan. A and B are the split-pairs used in the step. $nRegisters$ is the number of registers allowed to use for the computation. One would start searching for plans for a final sum FS by invoking ReverseSplit(FS, []). The output is then a list of possible plans to generate FS from IS.

```
Function ReverseSplit (SOPVRs, currentPlan) :
    if |SOPVRs| == 1 and isTransformable(SOPVRs[0], IS) then          // Base case
        plans << currentPlan + (SOPVRs, (IS ↦ SOPVRs[0]));
        return;
    pairs ← generatePairs(SOPVRs);                    // Generate all possible split pairs
    for (A, B) ∈ pairs do                             // Branch for every pair
        newSOPVRs ← [];                               // Array for the SOPVRs for the next step
        for S ∈ SOPVRs do
            if (A + B) ⊆ S then
                R ← S − (A + B);                      // Single SOPVR was split, create R
            else if A ⊆ S then
                R ← S − A;                            // This SOPVR contains A, create R
            else if B ⊆ S then
                R ← S − B;                            // This SOPVR contains B, create R
            else
                R ← S;                                // This SOPVR is not involved in split. Retain
            if |R| > 0 then
                newSOPVRs << R;                       // Only retain R if not empty
        newSOPVRs << B;                               // A is discarded, we have to retain B
        if |newSOPVRs| ≤ nRegisters then              // Recursive search if enough registers
            ReverseSplit(newSOPVRs, currentPlan + (SOPVRs, (B ↦ A)))
End Function
```

maximum size pairs are preferred. In a maximum size pair, all the available PVRs that match the pair's transformation vector are included. Also, splits that exhibit spatial proximity and aligned PVRs are preferred as they were found to tend to be easier to split in the next steps. This is especially the case when PVRs are aligned in columns and rows.

3.2 Graph Relaxation

As noted in Section 3.1.2, the reverse splitting algorithm finds plans to assemble SOPVRs under the constraint that it never uses anything that is not a sub-multiset of FS. This greatly reduces the search tree and makes the algorithm feasible in practice. There are also strong arguments for this constraint. Consider a SOPVR T that is not a sub-multiset of the FS ($T \not\subseteq FS$).

(1) If there is no SOPVR that can be transformed from T and is a subset of FG ($\nexists A \subseteq FS$ with T & A transformable), neither T nor any of its transformations would ever form part of the solution.
(2) If there is an SOPVR A that can be transformed from T and is a subset of FS ($\exists A \subseteq FS$, T & A transformable), one could just have generated A in the first place. This does not reduce generality as every SOPVR that is transformable from T is also transformable from A.

For case (1) it is clear such a step would never be beneficial for the final program. Case (2) however, only holds if we assume the cost of all transformations to be the same. There might be a third SOPVR $B \subseteq FS$ that is also transformable from T (and therefore also A). While it is true that one can generate A first, and then transform A into B, it might be cheaper to transform into $T \not\subseteq FS$ first and then generate A and B from T. In that case, there is a valid reason for having

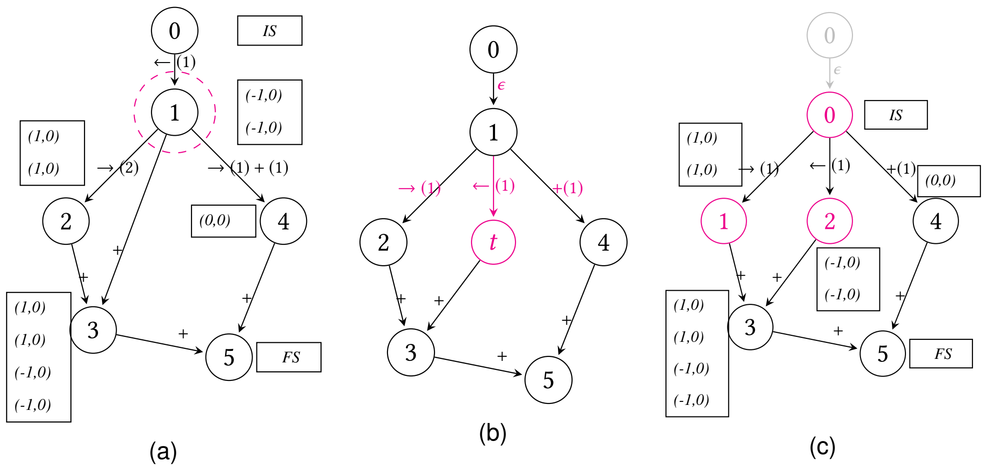


Fig. 5. Original and relaxed graph for **Example 10**. Node 0 is IS, and node 5 is FS. Next to each node, the SOPVR is shown. Node 1 in (a) can be relaxed. (b) shows the new edge weights and the new node that was introduced to make all edges of node 1 to be of transformation type. (c) shows the final step of assigning new numbers and removing the original start node.

the intermediate result T despite not being part of the final sum. Therefore, one can see that the previously chosen constraint to only consider direct sub-multisets of FS is too strong. In fact not only sub-multisets of FS, but also all SOPVRs that can be transformed into sub-SOPVRs of FS are potentially beneficial. However, replacing a SOPVR with any of its transformations in a plan *only* changes the transformation distances. The order of addition/subtraction remains the same as obtained by the reverse splitting algorithm. Therefore, for every plan that includes such a beneficial SOPVR, there exists a similar plan with the same addition order that fulfils the constraint and gets discovered by the reverse splitting algorithm.

EXAMPLE 10. *A very simple example is the following one dimensional filter K*

$$K = \begin{bmatrix} 1 & \frac{1}{2} & 1 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 2 & 1 & 2 \end{bmatrix} \tag{23}$$

with the final sum

$$FS = \left\{ \begin{matrix} (0,0) & (-1,0) & (-1,0) \\ (1,0) & (1,0) \end{matrix} \right\} \tag{24}$$

and initial sum

$$IS = \left\{ \ (0,0) \quad (0,0) \ \right\} \tag{25}$$

Fig. 5(a) shows a graphical representation of the plan obtained by the reverse splitting algorithm. One can see that there might be an opportunity for relaxation at node 1, as both outgoing transformation edges shift the value in opposite direction to the incoming transformation edge. Since node 1 also has an addition edge, a new node t is introduced with an ϵ edge between node 1 and node t. Note that this does not alter the program. By utilizing the method stated later in this paper, one can then write down the computational cost of the horizontal shift dimension as $C(\{1\}) = \sum_{k=0}^{n} |w(i_k)| + \sum_{l=0}^{m} |w(o_l)| = 1+2+0+1 = 4$. *Introducing a retiming of λ yields* $C(\{1\}) = \sum_{k=0}^{n} |w(i_k) - \lambda| + \sum_{l=0}^{m} |-w(o_l) - \lambda| = |1-\lambda| + |2-\lambda| + |0-\lambda| + |1-\lambda|$. *This expression is minimised for* $\lambda = 1$, *by the argument stated in this paper. Fig. 5(b) shows the application of the retiming to the computational graph. In a last step, the now empty edges are removed and a sequence number is given to the t node. This is decided by static analysis in order to minimise the register requirement. This is shown in Fig. 5(c). The retimed* *version of the computational graph requires two instructions less than the original version. Note that this increases the liveness of node* 3 *from* 2 *to* 3*, increasing the demand for registers.*

Albeit simplified, an approach from integrated circuit design, called retiming [24], is used. The method states, that if drawing a circle (as shown in Fig. 5-a by the dashed circle), that contains at least one node in the computation graph, but neither the input or output node, it is allowed to change the graph the following way: If we add a constant to a transformation direction on all edges that go into the circle, we have to subtract the same constant from transformation of all edges that point out of the circle. If following this rule, the original function of the program is preserved.

A problem arises by the fact that our graphs do not only contain transformation edges (i.e shift, scale and negation), but also addition edges on which relaxation cannot be applied. To address this problem, a new node is created at the origin of the add instructions, with an empty transformation edge from the node to be relaxed to the newly generated node, as shown by node t in Fig. 5-b. The edge between the origin node and node t is initially an empty transformation, but after the retiming, the edge is assigned a weight.

We consider a situation, with a node a having n input edges i_k, $k \in \{0...n\}$ and m output edges o_l, $l \in \{0...m\}$. Let $w(e)$ be the weight of edge e in a particular dimension (horizontal shift, vertical shift, scale). As relaxation is done on each dimension individually we omit the notion of dimension. The total computational cost $C(a)$ of node a in a dimension is given as the sum of the absolute value of the weights of all incoming and outgoing edges.

$$C(a) = \sum_{k=0}^{n} |w(i_k)| + \sum_{l=0}^{m} |w(o_l)| \tag{26}$$

If we relax the weights by an integer $\lambda \in \mathbb{Z}$ following the retiming theory, the total computational cost of the node in the dimension is given by

$$C(a) = \sum_{k=0}^{n} |w(i_k) - \lambda| + \sum_{l=0}^{m} |w(o_l) + \lambda| \tag{27}$$

This follows from the fact that we subtract the relaxation from all incoming edges and add the relaxation to all outgoing edges to maintain the functionality. The goal is to minimise the computational cost C given λ:

$$\min_{\lambda} C(a) = \min_{\lambda} \sum_{k=0}^{n} |w(i_k) - \lambda| + \sum_{l=0}^{m} |-w(o_l) - \lambda| \tag{28}$$

This is a geometric median problem for which the solution is the median of the weights [20]:

$$\lambda = \text{median}(\{w(i_0), .., w(i_n), -w(o_0), .., -w(o_m)\}), \tag{29}$$

with λ known, it is straight forward to apply the relaxation to the graph. Note that in many cases, the relaxation increases the amount of required physical register required in an instant of the program. If it would exceed the number of registers physically available, the relaxation is not performed.

3.3 Register Allocation

To perform the computational graph on the FPSP hardware, the nodes of the computational graph have to be mapped onto the physical registers. FPSP cells cannot spill values into the main memory; therefore, the program must not exceed the physical register count. The reverse splitting and the graph relaxation steps take this into account and only produce graphs that are guaranteed to comply with this limit. However, the actual allocation of register number to nodes still has to be decided.

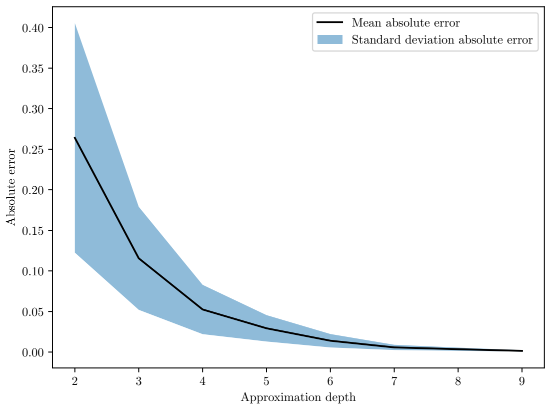


Fig. 6. Absolute pixel intensity value errors for pixels in range $[0 - 1]$ sampled on a set of 1000 random images with a set of 100 random kernels. The filter kernels have been approximated to increasing depths.

To allocate the registers to the computational graph G, first a liveness analysis is performed for every node of the graph. The liveness of a node a is the set of all other nodes that have to be present at the time node a is computed. Nodes are computed in the order of their numbers. From this information, a bidirectional dependency graph DG is generated, with vertices $\mathrm{V}(DG)$ that are the same as in the computational graph G, and edges $\mathrm{E}(DG)$ that stem from the liveness analysis; i.e. $\mathrm{E}(DG) = \{(a, b) : \forall a \in \mathrm{V}(DG), b \in \text{liveness}(a)\}$. If $a, b \in \mathrm{V}(DG)$ and $(a, b) \in \mathrm{E}(DG)$, then a and b are live at the same time and cannot be allocated to the same physical register. The program is inherently in single static assignment form (SSA), meaning that every variable only gets assigned exactly once and is assigned before use. This property would allow for a polynomial time register allocation algorithm such as [19]. However, the time spent in the code generation pipeline is dominated by the reverse split search. Register allocation on the comparatively short programs takes an insignificant amount of time. Due to this, we implemented a simple backtracking graph colouring algorithm which proved to be fast enough for all practical use cases. The algorithm assigns the k available physical registers to DG, by finding a graph colouring of DG using k colours.

4 EXPERIMENTS

This section presents experimental results. First the effect of the kernel approximation is analyzed. Then several metrics are shown that demonstrates the performance of the search algorithm and its pair generation function. At the end, the implementation and evaluation of a simple but powerful face detector, similar to the one described in [45], is presented. An in-depth discussion of the face detection algorithm is provided by [47].

4.1 Effects of Approximation

While the theoretical limits of the approximation of coefficients are shown in Fig. 2, this does not give an insight into the actual effects of the approximation on the image convolution result. To show the effect of the approximation, a set of 100 random 3×3 convolution kernels was generated. Each coefficients of each kernel is independent and identically distributed from a uniform distribution between $[0, 1]$. All coefficients in the kernels were approximated with the method from Eq. (7). A test set of 1000 random images from the *Caltech101* ([26]) was chosen. Each filter was applied to the images, once in original kernel (perfect) configuration and once in the approximated configuration. An error metric was calculated as the absolute differences in pixel intensities after the application of the kernels.

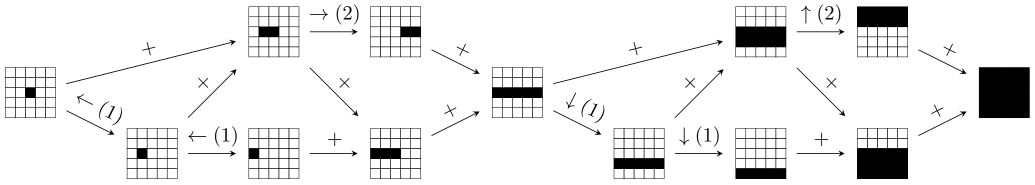


Fig. 7. Graphical representation of the computation graph of a 5×5 Box filter. From an initial value at position $(0, 0)$ (IS), the program gradually generates the missing parts in order to efficiently assemble the final sum. Summing up all the shifts and additions, this program requires 14 instructions.

Fig. 6 shows the real-world absolute pixel intensity errors that result from approximating the kernels to various depths using Eq. (7). One can see, that every approximation step yields around half the error rate of the previous approximation. This is expected, since we are approximating at powers of two. Another fact to note is that when approximating to a depth of 8, and assuming an 8-bit input image, the kernels are approximated to the same depth as the input image.

4.2 Performance Evaluation

In this section, several experiments are explained, designed to demonstrate the performance of the proposed method. The efficiency of the heuristic search method is compared with exhaustive search. Then the program length is analyzed and comparisons with CPU/GPU implementations are shown.

4.2.1 Full Heuristic vs. Exhaustive Search. Fig. 8 compares the fully exhaustive search with all heuristics disabled to the heuristic search on the Sobel filter. There is an ordering given to the split pairs based on the implementation of the algorithm. Albeit uninformed, it makes the algorithms performance deterministically implementation dependent. In order to remove this effect, the ordering of the pairs was randomized in the exhaustive runs, making them stochastic. To every stochastic run of the algorithm, we report mean as well as standard deviation of the results. One can see from the figure that the heuristic search manages to find the optimum of 8 instructions almost immediately while the exhaustive search takes (in general) a much longer time to reliably find it. The figure also shows best and worst case runs from the sampling of 256 performed exhaustive runs. In the best case, the exhaustive runs finds the optimal solution almost immediately as well. This however is just a matter of chance, whereas the heuristic search yields the immediate result in every run.

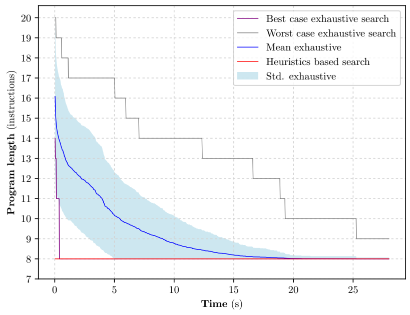


Fig. 8. Full exhaustive search vs. heuristic search on Sobel 3×3 filter. Sampled over 256 independent runs.

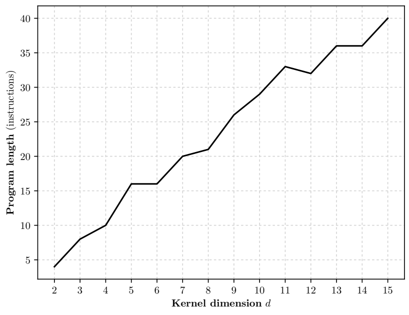


Fig. 9. Program length for Box filters of size $d \times d$. Note that the program length increases linearly.

To further elaborate on the optimization, we discuss the Box filters, kernels with 1 on all elements. These filters do not need value approximation and expose exactly one PVR per coordinate in their final sum SOPVR. This makes them especially well suited for optimization as there is a large potential of reusing previously computed values. For example, for the 5×5 Box filter, the algorithm produces the code shown in Listing 1. As we only have one PVR per coordinate, we can represent individual SOPVRs graphically. Fig. 7 shows the computational graph for the 5×5 Box filter, annotated with a graphical representation of the SOPVRs.

Listing 1. Generated code for a 5×5 Box filter. Functions `north`, `east`, `south`, and `west` are used for shifting the image on the device, see Table 1. For a graphical representation, see Fig. 7.

```
B = east(A)
A = add(A, B)
B = east(B)
B = add(B, A)
A = west(A)
A = west(A)
A = add(B, A)
B = north(A)
A = add(A, B)
B = north(B)
B = add(A, B)
A = south(A)
A = south(A)
A = add(B, A)
```

4.2.2 Program Length. Another interesting property of the FPSP is apparent, when comparing program lengths. Fig. 9 depicts the program length vs. the dimensionality of the filter, for Box filters. The graph shows that the program length for Box filters of dimension d increases linearly with d and not, as one would expect, quadratically. This stems from the fact that the algorithm implicitly separates the filter into two linear sums: $K = [1, \ldots, 1]^T \times [1, \ldots, 1]$.

4.2.3 Comparisons with CPU and GPU implementations. To compare the execution times of kernels on the FPSP hardware in comparison with standard CPU and GPU implementations, various test runs have been performed. A selection of eight well-known filter kernels was executed on a 256×256 $8bit$ grayscale image. The image resolution and color was chosen to achieve comparable results to the FPSP implementation, as the SCAMP chip features the same resolution. The chosen algorithms for CPU and GPU are the default algorithms shipped with *OpenCV 3.3.0*. The CPU version of OpenCV was compiled with both *Threading Building Blocks (TBB)* as well as *Integrated Performance Primitives (IPP)* enabled. The GPU algorithms were compiled for *CUDA V8.0.61* to run on *NVidia CUDA* equipped GPUs. To measure the time and power, the same algorithm was applied $10,000$ times to the same image, and the average results are reported. The reported time is therefore the average runtime of these filter applications. Only computation time was measured. Transfers between hard drive and system memory as well as from system memory to GPU memory were performed outside the timing loop. A number of different *Intel* CPUs from different generations were tested as well as three *NVidia* graphics cards. The statistics reported for the SCAMP FPSP chip are based on the reported 10 MHz instruction rate [9], as well as the length of the programs generated by the algorithm. The values for these filters can be perfectly represented so there is no loss in quality introduced by the approximation.

Fig. 10 shows the execution times of a single application of various filter kernels on different hardware components. The parallel nature of the SCAMP FPSP allows it to perform all of the tested filter kernels in a fraction of the time needed by the other devices. Fig. 11 shows the average continuous power consumption of the devices, at computing the same filter kernels. Most CPUs consume about $20W$ to $30W$, with the *i7-3720QM* using slightly less. This is most probably due to it being a mobile CPU while the others are desktop class CPUs. The generated programs do not incur any loss in quality. The figure demonstrates that compared with many other processing systems, FPSP consumes at least 20 times less power. The *TITAN X* GPU consumes a considerable amount more power than the CPUs, with values ranging up to more than $150W$. The SCAMP chip consumes only $1.23W$ under full load, which is a lot less than any of the other tested devices. The benefit in

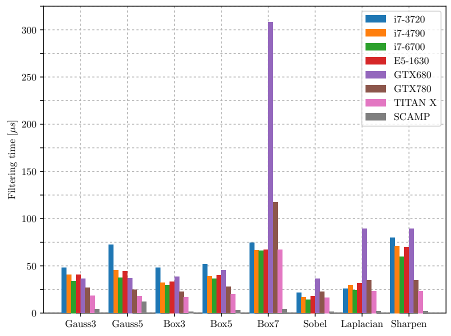


Fig. 10. Runtime of various well-known filter kernels on different types of CPU and GPU hardware (averaged samples from $10,000$ runs), as well as the estimated runtime for the generated algorithm for the FPSP. For the CPU and GPU values, the algorithms are the default algorithms shipped with *OpenCV 3.3.0*. Due to the analogue operation of SCAMP the resulting images do incur more noise. However, the performance gain mainly comes from the pixel-parallel architecture rather than the analogue operation.

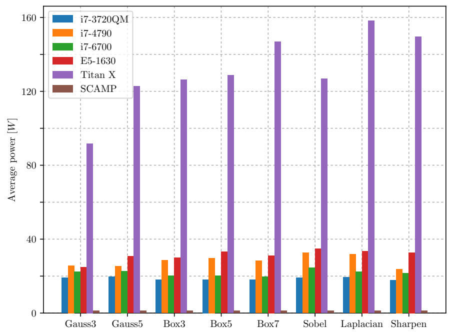


Fig. 11. Focal-plane Sensor-Processor Arrays (FPSPs) are very power efficient. This graph shows average power consumption of FPSP versus several other hardware components, performing filtering with various kernels, averaged over 10,000 applications of the filter kernels. The FPSP programs are generated by our method. Due to the analogue operation of SCAMP the resulting images do incur more noise. However, the performance gain mainly comes from the pixel-parallel architecture rather than the analogue operation.

Units: mJ	i7-3720	i7-4790	i7-6700	E5-1630	Titan X	SCAMP
Gauss3	0.941	0.786	0.617	1.021	1.542	**0.005**
Gauss5	1.411	1.143	0.878	1.397	2.120	**0.015**
Box3	0.868	0.932	0.602	1.000	2.186	**0.002**
Box5	0.926	1.167	0.737	1.336	2.568	**0.004**
Box7	1.297	1.922	1.296	2.094	10.820	**0.005**
Sobel	0.406	0.556	0.360	0.594	2.178	**0.002**
Laplacian	0.497	0.939	0.555	1.041	4.092	**0.003**
Sharpen	1.418	1.677	1.295	2.281	3.768	**0.002**

Table 3. Energy spent per filter application for various well-known filters on various hardware components.

energy ranges from a 180 times improvement (*Sobel, i7-6700*) all the way to a 2100 times less energy per frame (*Box7, TITAN X*).

Table 3 report the energy the devices spend per *single* filter application. This value was computed by dividing the continuous power consumption by the frame rate. Still, the *TITAN X* GPU has the highest power consumption for all filters. However, the difference to the CPUs is considerably smaller than it was for the continuous power consumption. This follows naturally from the fact, that the GPU manages to perform more filter applications in the same time. The results look different for the SCAMP FPSP. Since it has not only the smallest computation time per filter, but also the lowest continuous power consumption, the energy spent per frame is much smaller than for any device.

Fig. 15 shows that an FPSP with specification such as the SCAMP would be capable of performing the full face detection algorithm at competitive speeds. However, due to fact that the SCAMP chip is implemented as an analogue device, every operation performed introduces an irreversible error on the data.

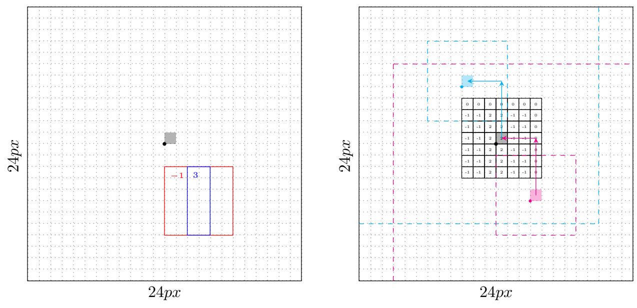


Fig. 12. An example of a feature consisting of two sums. The large 24×24 square is the detection frame of the processing element at coordinates $(12, 12)$ in the middle, shaded in gray. This processing element has to obtain the values of the weighted sums of the pixels in the red and blue rectangles. To facilitate the problem, the algorithm translates the feature to a filter kernel, aligned at the center of the detection frame. After computing the kernel and thresholding, the (binary) result gets shifted to the top-left to be evaluated by the processing element of which the features center coincides with the center of our detection frame. At the same time, we get the result from the processing element on the bottom-right.

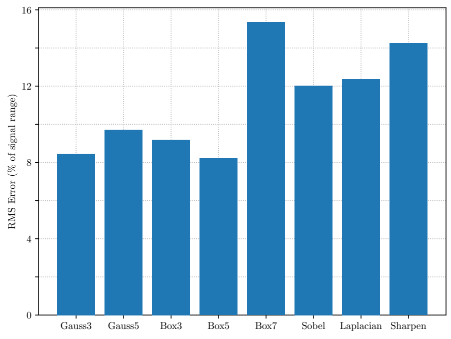


Fig. 13. The analogue nature of analogue FPSP adds errors to the output. Induced root mean square pixel intensity differences between noisy and perfect image measured on a SCAMP 5 device. The errors are averaged over 100 randomly chosen real images. Averaging filters such as Box and Gauss suffer less from the accuracy loss than differential filters such as Sobel or Laplacian.

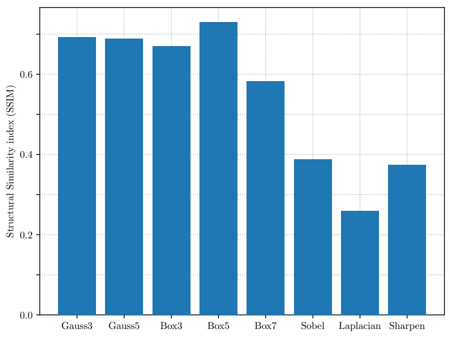


Fig. 14. This graph shows the Structural Similarity Index (SSIM) [48] of a noise free filter application on CPU and a noisy filter application on SCAMP. The SSIM assesses image quality based on the loss of structural information in the image. The averaging filters suffer less from the accuracy loss as they also average out induced noise.

4.3 Accuracy of Analogue Computations

Focal-Plane Sensor-Processing Arrays can be both implemented using analogue as well as digital logic. This paper focuses on the analogue SCAMP-5 device. This section elaborates the effects of analogue computation on the accuracy of the results. All experiments have been carried out on a SCAMP-5 hardware device. The ground truth values were computed with a realistic simulator for the SCAMP chip [3] with the noise model deactivated. Both the simulator and the hardware ran the same code produced by our method. All instructions of the class the generated filters use are implemented as a double register-transfer on the hardware [16]. The noise model differentiates between signal dependent noise and signal independent noise [8]. While signal dependent noise can be mitigated by operating on smaller signal amplitudes, signal independent noise can be mitigated by operating on larger signal amplitudes. This trade-off could lead to further optimisations which this paper does not take into account.

We apply eight filters including Gauss3, Gauss5, Box3, Box5, Box7, Sobel, Laplacian, and Sharpen to 100 random images from the *Caltech 101* dataset [26]. Fig. 13 shows the RMS pixel intensity errors compared to a noise-free computation averaged over 100 sample images. Fig. 14 shows the Structural Similarity Index (SSIM) [48] between a noise free and a noise modelled filter

[3]https://github.com/najiji/cpa-sim

application. SSIM is a metric to assess perceived image quality by measuring structural information loss instead of absolute errors. Even for short and simple filters, the analogue nature of the chip adds substantial errors. While the effect is less strong for averaging filters such as gaussian and box filters, it is stronger for differential filters such as Laplacian and Sobel. The speed gains of the device are achieved thanks to single-pixel parallelism, whereas the power savings are mostly the result of the slow clock frequency. Massive parallelism, as the SCAMP device, allows for strong speed and energy gains at the expense of silicon area per pixel. Limited silicon area makes the implementation of accurate analogue or fully functional digital circuitry more challenging. A future device could be placed more favourably in the parallelism-silicon device space. It is expected that future digital devices would suffer less from the accuracy problem, while still be benefiting from our method.

4.4 Face Detection

In this section, a face detection experiment on FPSP is demonstrated. A face detector implementation, also available in *OpenCV*, based on the pre-trained values of the *FrontalFace* Haar cascade contributed by [27], is used to compare the results.

In this face detector, every processing element is responsible for the detection of objects in its 24×24 neighborhood. To do so, the processing element has to compute the features, which are essentially thresholded differences of partial sums, of the pixels in its neighborhood (Haar features). A straight forward approach would be to just to represent the features as a large 24×24 filter mask. However, looking at the provided features by *OpenCV*, there are a lot of small features at considerable offset from the center of the detection frame. Fig. 12 shows an example of a feature with two sums to be evaluated. Since shifting is a very natural and cheap operation on the FPSP, and every pixel performs the same computation simultaneously, it does not matter where exactly inside the detection frame a processing element computes the kernel, as the result of the computation can easily get shifted to the right place. This allows us to relocate the computation of the sums for the feature to a location that is most suitable for the specific processing element. This is, naturally, the position where the center of the feature coincides with the processing elements location. The very sparse 24×24 pixel kernel of the straight forward approach therefore gets replaced by a smaller, dense kernel of the same size as the area covered by the features sums. An additional benefit comes from the fact, that in later stages a lot of the Haar features are equivalent in shape, but evaluated at different locations. Since the FPSP computes the results in every location at the same time, we can reuse the computations of other cells to evaluate multiple, similar Haar features at the same time. It turns out that of the 2913 features in the pretrained *OpenCV* cascade, only 1656 are distinct in shape. This allows us to reduce the amount of sums to be computed by 43%. A face detector using the full 25 OpenCV stages was implemented. Due to simulator code size limitations, testing of more than 7 stages could not be performed. Experiments on the *MIT CBCL* [34] dataset showed a difference in classification rate, recall and precision of 2%, 1% and 2% compared to OpenCV. This comes with the benefit of a 90% reduction in energy consumption. CPUs and GPUs benefit from the *integral image* representation [45] for efficient summation. The FPSP in contrast sums up the pixel values in a naive but massively parallelized way. This is why the FPSP only exhibits a boost in energy consumption while having a slightly longer runtime. Nevertheless, it is impressive that due to the massive parallelism, this much more naive algorithm shows runtimes in the same order of magnitude as CPU, even though it runs a more computation intensive algorithm at a clock frequency that is two orders of magnitude lower.

5 CONCLUSIONS

This paper presents a formalism to write convolutional kernels as multisets of approximation factors on FPSP. Furthermore, a formalism is presented to encode the chips hardware functionalities into

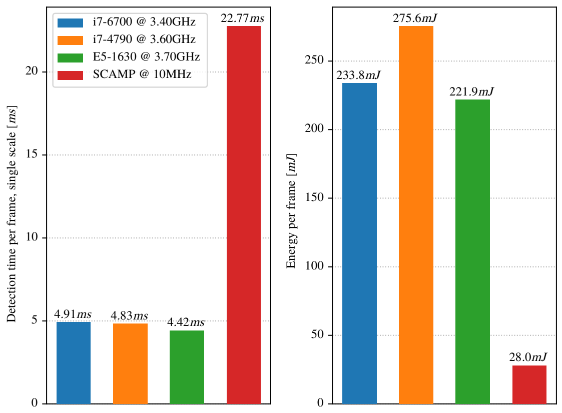


Fig. 15. Speed and energy consumption of CPUs (measured) and FPSP (estimated) at performing the full 25 stage *OpenCV* face detector. Note that detection was only performed at a single scale on a 256×256 image, hence the good CPU performance. Although having a longer processing time, the FPSP can yield significant energy savings. Note also that these values assume a relatively slow FPSP running at 10 MHz.

operations on multisets. An algorithm, called the *reverse splitting* algorithm, which uses this formalism to find plan for building up the convolutional kernel. The algorithm makes use of heuristics, which in every step provide it with likely good choices for creating a good program. Subsequent software is presented that can optimize the program by performing local optimizations on the output of the *reverse splitting* algorithm. This is done by changing edge values in the computation graph by a method known as *retiming*, commonly applied in integrated circuit design. Known algorithms such as graph coloring for register allocation have been used in subsequent steps to create a real, runnable FPSP program from the intermediate representations as well as validating the result against the input. The system has proved to be very reliable in generating code for arbitrary filter kernels. Especially the heuristics has proven to be essential in speeding up the algorithm to acceptable levels. It has been shown, that the code generated by the algorithm is in most cases equivalent, or outperforms code written by human experts for the same convolutional filter. Experimental results on a set of well-known kernels demonstrates that FPSP can perform the filtering task consuming up to 200 times less energy per frame than a CPU, while running at up to 10 times the frame rate of a CPU. Experiments using a face detector showed similar performance to a CPU implementation, running slightly slower than the CPU implementation on a 10 *MHz* FPSP. A future FPSP is expected to complete the task faster than CPU. It was shown that current FPSP would run the face detector using roughly 10% of the power a CPU requires.

Convolutional Neural Networks such as *ImageNet* ([22]) contain large amounts of convolutional filters in their first layers. Arbitrary kernel code generation as presented in this work would allow the computation of these layers completely pixel parallel in hardware, significantly speeding up recognition performance. Arbitrary filter code generation is a basic building block for many more sophisticated high-level Computer Vision applications such as edge detection, image segmentation or object recognition. In the future, we would like to develop a compiler that can directly compile efficient FPSP code from a high-level language such as C. Also imaginable is a novel programming language better aimed at the pixel-parallel execution capabilities of FPSP hardware.

ACKNOWLEDGMENTS

This research is supported by Engineering and Physical Sciences Research Council [grant number EP/K008730/1]. We thank Piotr Dudek and his colleagues at Manchester University for kindly providing access to the SCAMP device and simulator, and Fabio Luporini for helpful comments on the manuscript.

REFERENCES

[1] 2016. Design of hardware efficient FIR filter: A review of the state-of-the-art approaches. *Engineering Science and Technology, an International Journal* 19, 1 (2016), 212 – 226.

[2] Stefano Ambrogio, Pritish Narayanan, Hsinyu Tsai, Robert M. Shelby, Irem Boybat, Carmelo di Nolfo, Severin Sidler, Massimo Giordano, Martina Bodini, Nathan C. P. Farinha, Benjamin Killeen, Christina Cheng, Yassine Jaoudi, and Geoffrey Burr. 2018. Equivalent-accuracy accelerated neural-network training using analogue memory. 558 (2018), 60–76.

[3] Algirdas Avizienis. 1961. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on electronic computers* 3 (1961), 389–400.

[4] Marco Bodrato and Alberto Zanoni. 2007. Integer and Polynomial Multiplication: Towards Optimal Toom-Cook Matrices. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC)*. 17–24.

[5] L. Bose, J. Chen, S. J. Carey, P. Dudek, and W. Mayol-Cuevas. 2017. Visual Odometry for Pixel Processor Arrays. In *2017 IEEE International Conference on Computer Vision (ICCV)*. 4614–4622.

[6] DR Bull and DH Horrocks. 1991. Primitive operator digital filters. *IEE Proceedings G (Circuits, Devices and Systems)* 138, 3 (1991), 401–412.

[7] Stephen J Carey, David RW Barr, Bin Wang, Alexey Lopich, and Piotr Dudek. 2012. Locating high speed multiple objects using a SCAMP-5 Vision-Chip. In *Cellular Nanoscale Networks and Their Applications (CNNA), 2012 13th International Workshop on*. IEEE, 1–2.

[8] Stephen J Carey, Alexey Lopich, David RW Barr, Bin Wang, and Piotr Dudek. 2013. A 100,000 fps vision sensor with embedded 535GOPS/W 256×256 SIMD processor array. In *VLSI Circuits (VLSIC), 2013 Symposium on*. IEEE, C182–C183.

[9] Stephen J Carey, Alexey Lopich, and Piotr Dudek. 2011. A processor element for a mixed signal cellular processor array vision chip. In *Circuits and Systems (ISCAS), 2011 IEEE International Symposium on*. IEEE, 1564–1567.

[10] H. G. Chen, S. Jayasuriya, J. Yang, J. Stephen, S. Sivaramakrishnan, A. Veeraraghavan, and A. Molnar. 2016. ASP Vision: Optically Computing the First Layer of Convolutional Neural Networks Using Angle Sensitive Pixels. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 903–912.

[11] Xiaoming Chen, Jianxu Chen, Danny Z. Chen, and Xiaobo Sharon Hu. 2017. Optimizing Memory Efficiency for Convolution Kernels on Kepler GPUs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[12] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Annual IEEE/ACM International Symposium on Microarchitecture*. 609–622.

[13] Andrew G Dempster and Malcolm D Macleod. 1995. Use of minimum-adder multiplier blocks in FIR digital filters. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 42, 9 (1995), 569–577.

[14] Rafael Dominguez-Castro, Servando Espejo, Angel Rodriguez-Vazquez, Ricardo A Carmona, Peter Foldesy, Ákos Zarándy, Péter Szolgay, Tamás Szirányi, and Tamás Roska. 1997. A 0.8-/spl mu/m CMOS two-dimensional programmable mixed-signal focal-plane array processor with on-chip binary imaging and instructions storage. *IEEE Journal of Solid-State Circuits* 32, 7 (1997), 1013–1026.

[15] Piotr Dudek. 2003. A flexible global readout architecture for an analogue SIMD vision chip. In *Circuits and Systems, 2003. ISCAS'03. Proceedings of the 2003 International Symposium on*, Vol. 3. IEEE, III–III.

[16] Piotr Dudek. 2005. Implementation of SIMD vision chip with 128×128 array of analogue processing elements. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*. IEEE, 5806–5809.

[17] Piotr Dudek and Peter J Hicks. 2000. A CMOS general-purpose sampled-data analog processing element. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 47, 5 (2000), 467–473.

[18] Piotr Dudek and Peter J Hicks. 2005. A general-purpose processor-per-pixel analog SIMD vision chip. *IEEE Transactions on Circuits and Systems I: Regular Papers* 52, 1 (2005), 13–20.

[19] Sebastian Hack and Gerhard Goos. 2006. Optimal register allocation for SSA-form programs in polynomial time. *Inform. Process. Lett.* 98, 4 (2006), 150–155.

[20] J. B. S. Haldane. 1948. Note on the median of a multivariate distribution. *Biometrika* 35, 3-4 (1948), 414–417.

[21] Richard I Hartley. 1996. Subexpression sharing in filters using canonic signed digit multipliers. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 43, 10 (1996), 677–688.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[23] Andrew Lavin and Scott Gray. 2016. Fast Algorithms for Convolutional Neural Networks. In *Computer Vision and Pattern Recognition (CVPR)*. 4013–4021.

[24] Charles E Leiserson, Flavio M Rose, and James B Saxe. 1983. Optimizing synchronous circuitry by retiming. In *Proceedings of the 3rd Caltech Conference on VLSI*. 87.

[25] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Grösslinger, Frank Hannig, Harald Köstler, Ulrich Rüde, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, Sebastian Kuckuk, Hannah Rittich, and Christian Schmitt. 2014.

ExaStencils: Advanced Stencil-Code Engineering. In *Euro-Par Workshops*.

[26] Fei-Fei Li, Marco Andreetto, and Marc 'Aurelio Ranzato. 2003. Caltech101 Image Dataset. (2003). http://www.vision.caltech.edu/Image_Datasets/Caltech101/

[27] Rainer Lienhart. 2013. Haarcascade Frontalface Default. https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml. (2013).

[28] R. LiKamWa, Y. Hou, Y. Gao, M. Polansky, and L. Zhong. 2016. RedEye: Analog ConvNet Image Sensor Architecture for Continuous Mobile Vision. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 255–266.

[29] Xing Lin, Yair Rivenson, Nezih T. Yardimci, Muhammed Veli, Yi Luo, Mona Jarrahi, and Aydogan Ozcan. 2018. All-optical machine learning using diffractive deep neural networks. *Science* (2018).

[30] G Linan, S Espejo, R Dominguez-Castro, and A Rodriguez-Vazquez. 2002. Architectural and basic circuit considerations for a flexible 128×128 mixed-signal SIMD vision chip. *Analog Integrated Circuits and Signal Processing* 33, 2 (2002), 179–190.

[31] Fabio Luporini, David A. Ham, and Paul H. J. Kelly. 2017. An algorithm for the optimization of finite element integration loops. *ACM Trans. Math. Softw.* 44 (2017), 3:1–3:26.

[32] J. N. P. Martel, M. Chau, P. Dudek, and M. Cook. 2015. Toward joint approximate inference of visual quantities on cellular processor arrays. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2061–2064.

[33] J. N. P. Martel, L. K. Mueller, S. J. Carey, and P. Dudek. 2016. A Real-time High Dynamic Range Vision System with Tone Mapping for Automotive Applications. In *CNNA 2016; 15th International Workshop on Cellular Nanoscale Networks and their Applications*. 1–2.

[34] MIT. 2013. CBCL Face Database #1. http://www.ai.mit.edu/projects/cbcl. (2013).

[35] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. *SIGARCH Comput. Archit. News* 43, 1 (2015), 429–443.

[36] Robert Pasko, Patrick Schaumont, Veerle Derudder, Serge Vernalde, and Daniela Durackova. 1999. A new algorithm for elimination of common subexpressions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18, 1 (1999), 58–68.

[37] Jonne Poikonen, Mika Laiho, and Ari Paasio. 2009. MIPA4k: A 64×64 cell mixed-mode image processor array. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*. IEEE, 1927–1930.

[38] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. 519–530.

[39] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2018. Register optimizations for stencils on GPUs. In *PPOPP*.

[40] I. Z. Reguly, Gihan R. Mudalige, and Michael B. Giles. 2018. Loop Tiling in Large-Scale Stencil Codes at Run-Time with OPS. *IEEE Transactions on Parallel and Distributed Systems* 29 (2018), 873–886.

[41] George W Reitwiesner. 1960. Binary arithmetic. In *Advances in computers*. Vol. 1. Elsevier, 231–308.

[42] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 14–26.

[43] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2014. A framework for enhancing data reuse via associative reordering. In *PLDI*.

[44] Hong-Phuc Trinh, Marc Duranton, and Michel Paindavoine. 2015. Efficient Data Encoding for Convolutional Neural Network Application. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 4 (2015), 49:1–49:21.

[45] Paul Viola and Michael Jones. 2001. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, Vol. 1. IEEE.

[46] Yevgen Voronenko and Markus Püschel. 2007. Multiplierless multiple constant multiplication. *ACM Transactions on Algorithms (TALG)* 3, 2 (2007), 11.

[47] Yi-Qing Wang. 2014. An analysis of the Viola-Jones face detection algorithm. *Image Processing On Line* 4 (2014), 128–148.

[48] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.

[49] Shmuel Winograd. 1980. *Arithmetic complexity of computations*. Society for Industrial and Applied Mathematics Philadelphia.

[50] Ákos Zarándy. 2014. *Focal-Plane Sensor-Processor Chips*. Springer Publishing Company, Incorporated.

Received May 2018