Autonomous Robots (2022) 46:893–910
https://doi.org/10.1007/s10514-022-10053-w

Compiling CNNs with Cain: focal-plane processing for robot navigation

Edward Stow[1] · Abrar Ahsan[2] · Yingying Li[2] · Ali Babaei[2] · Riku Murai[1] · Sajad Saeedi[2] · Paul H. J. Kelly[1]

Received: 1 July 2021 / Accepted: 28 June 2022 / Published online: 10 September 2022

Abstract
Focal-plane Sensor-processors (FPSPs) are a camera technology that enables low power, high frame rate computation in the image sensor itself, making them suitable for edge computation. To fit into the sensor array, FPSPs are highly resource-constrained, with limited instruction set and few registers - which makes developing complex algorithms difficult. In this work, we present Cain, a compiler for convolutional filters that targets SCAMP-5, a general-purpose FPSP. Cain generates code to evaluate multiple convolutional kernels at the same time. It generates code that avoids the need for hardware multipliers, while orchestrating the exploitation of common sub-terms—leading to a large reduction in instruction count compared to both straightforward and prior optimized approaches. We demonstrate the capability enabled by Cain on SCAMP-5 with robotic navigation for near-sensor high-speed and low-power computation, by using Cain to implement a neural network on the focal plane.

Keywords Convolution · SIMD · Image sensor · Analogue computing · Edge inference

1 Introduction

Real-time robotic and computer vision applications are currently bound to traditional camera sensors that transfer each pixel at each frame to a host where it is processed. This requires high-performance buses between the sensors and hosts, especially where high frame-rates are required. Some autonomous driving scenarios may require new information for every 1 cm travelled to be vigilant of unexpected scenarios, so at 80 km/hr a frame rate of 2222 Hz would be required. A 2 mega-pixel camera, with 10-bit pixel depth, running at such a frame rate, requires a bus capable of 45.6 Gbit/s—which is currently only possible with devices such as a PCI-e x8 Gen3 interface (XIMEA, 2021). For many applications, streaming data at such volumes is too demanding—in energy, bandwidth and latency—hence requiring an alternative solution.

In high speed robotics, low latency predictions become a requirement to ensure proper navigation and obstacle avoidance. The latency of a standard camera becomes a bottleneck in these cases and is an important obstacle in achieving low latency predictions. One way to avoid this bottleneck is to do some pre-processing of image data in the sensor, to reduce the amount of data that is to be transferred downstream. Codesign of hardware and software for computer vision applications is an emerging research field to address the limitations of conventional systems (Saeedi et al., 2018). Focal-plane Sensor-processors (FPSPs) are a promising avenue for reducing the data transfer between the camera and the processing unit. FPSPs, often synonymous with Cellular Processor Arrays (CPAs) and Pixel Processor Arrays (PPAs), perform processing on the sensor chip itself and are often designed for tasks which require high frame rates or low latency (Zarándy, 2011). The principle behind them is

✉ Edward Stow
edward.stow16@imperial.ac.uk

Abrar Ahsan
abrar.ahsan@ryerson.ca

Yingying Li
yingying.li@ryerson.ca

Ali Babaei
ali.babaei@ryerson.ca

Riku Murai
riku.murai15@imperial.ac.uk

Sajad Saeedi
s.saeedi@ryerson.ca

Paul H. J. Kelly
p.kelly@imperial.ac.uk

[1] Department of Computing, Imperial College London, London, UK

[2] Department of Mechanical and Industrial Engineering, Toronto Metropolitan University, Toronto, Canada

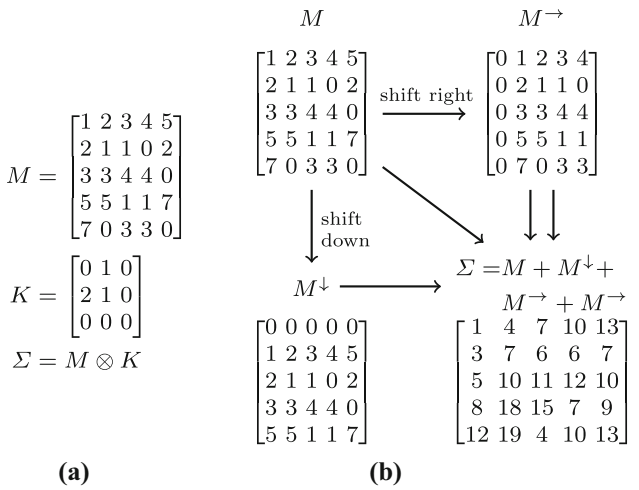


Fig. 1 Performing kernel convolution using shift and addition operations. **a** For matrix M and kernel K, the classic definition of convolution is shown by Σ. **b** The convolution described in (**a**) can be performed by shifting M in two directions and adding the results. The shift and addition operations are derived specifically for the kernel K

that a small processor is embedded directly with each pixel of the sensor. While FPSPs come in various forms for specific applications, in this paper we explore a general-purpose fine-grain architecture SCAMP-5 (Carey et al., 2012), but one can imagine alternatives that could be designed for various use cases.

One of the most widely used methods for image analysis is convolution kernels. From edge detection using Sobel filters to document recognition using Convolutional Neural Networks (CNNs) (LeCun et al., 1998), convolutional kernels are the foundation for many complex computer vision applications. Traditionally, application of the convolutional kernels to the image data occurs on a CPU, but more recently GPUs and FPGAs are used to accelerate the computations in parallel (Abadi et al., 2016; Chen et al., 2016). The convolution operation is the sum of products of the elements of an input matrix and a kernel matrix. Figure 1a shows an example, where matrix M is convolved with kernel K resulting in matrix Σ. Alternatively, the convolution operation can be done differently as shown in Fig. 1b. The kernel K describes that each generated pixel i is the sum of four elements: pixel i itself, the adjacent pixel above i and 2 of the pixel on i's left. Therefore, to do the convolution for all pixels, matrix M is shifted to the right and also down (to 'move' the adjacent pixels to i onto i itself), then the produced matrices and original matrix are summed with the right-shifted matrix added twice, as shown in Fig. 1b. This indicates that the convolution operation can be done by shifting the images in proper directions and adding up the proper factors of the pixel values.

Several systems have been designed to optimise the processing of convolutional kernels on GPUs and FPGAs, leading to a vast array of techniques to reduce the number of operational cycles needed to apply kernels to input data. While this significantly increased throughput, these methods are still bounded in latency as the image must make its way from the camera through to the host system. As for FPSPs, the ability to process the data on the focal plane enables the kernels to be applied to the image data at very low latency. Furthermore, the unique ability to select the data which is transferred from the device to the host reduces the data volume, which allows for high frame rates. However, the technology is comparatively new. By design, they offer novel ways to interact with the data, and while work has been done to provide a Domain-Specific-Language and associated tools to program such hardware (Martel, 2019), there has been less work done so far to produce code generation systems to make efficient use of their architectural features when applying convolutional kernels in particular. One such system that does exist for FPSPs, however, is AUKE (Debrunner et al., 2019b).

In this work, we present an improved alternative to AUKE, with the ability to produce code for applying multiple convolutional kernels at a time. The problem is presented as a dynamic graph search problem in which we must efficiently generate and traverse possible processor states to find a path that describes the relevant convolutional computation. By incorporating instruction selection and instruction scheduling into the core of search process, we enable the use of more novel features of FPSP architectures than AUKE is able to use. By optimising the code for multiple kernels simultaneously, common sub-expressions between kernels can be exploited and produced only once rather than for each kernel. This reduces the computational expense of applying the kernels, enabling applications to run at a faster frame rate. We conduct robotics experiments demonstrating that a network architecture trained and implemented using Cain can successfully achieve competitive performance on FPSP hardware.

The primary objective of this work is to push the boundary of code generation for FPSP devices through simultaneous kernel optimisation. We offer the following contributions:

- Cain[1]: A code generation algorithm which effectively makes use of common sub-expressions across filters consisting of multiple convolutional kernels. Our graph search strategy—which enables Cain to efficiently search large graphs—combines instruction scheduling, instruction selection and register-allocation constraints into the core of the search to make better use of specific hardware capabilities in SIMD processors.
- We show how this search can be tractable for problems of interest through a problem formulation based on AUKE's multi-set–of–Atoms problem representation, combined

[1] Available at https://github.com/ed741/cain.

with a ranking heuristic and a hybrid graph-generator–graph-search exploration strategy.
- We show how this approach allows flexible exploitation of hardware capabilities (such as three-operand adds and multi-step shifts), and generates very efficient use of additions to avoid multiplies.
- Evaluation of the effectiveness of Cain on the SCAMP-5 Focal-plane Sensor-processor. We compare against AUKE and test the effectiveness of simultaneous kernel optimisation. We also explore how our simultaneous kernel optimisation extends to future devices with more registers per pixel.
- We present a practical demonstration and comparative evaluation of robotic collision avoidance using our AnalogNavNet model running on the SCAMP-5 FPSP, contrasting it with alternative processing architectures.

The remainder of the paper is organised as follows. Section 2 describes the SCAMP-5 and its instruction sets, Sect. 3 briefly describes related works such as AUKE and robotic navigation methods similar to AnalogNavNet, Sect. 4 explains our proposed code generation algorithm Cain, and in Sect. 5, detailed comparison is made between Cain and AUKE, together with an evaluation of the effectiveness of simultaneous kernel optimisation. In Sect. 6, we present our experimental work, using Cain to implement our robot navigation model, AnalogNavNet. Finally, Sect. 7 concludes our work, with a discussion about potential future research.[2]

2 BACKGROUND: SCAMP-5 Focal-plane sensor-processor

In this section, we discuss the capabilities of the next generation camera technology SCAMP-5, and give an overview of the functionality used by Cain.

SCAMP-5 has been demonstrated in many different computer vision applications, ranging from Visual Odometry systems (Murai et al., 2020; Bose et al., 2017; Debrunner et al., 2019a), an end-to-end neural sensor which performs learnt pixel exposures (Martel et al., 2020), to Convolutional Neural Networks (Wong et al., 2020; Bose et al., 2019; Liu et al., 2020). Its distinctive ability to perform computation on the focal-plane reduces power consumption and data transfers, making the device promising for edge computation.

The SCAMP-5 architecture is a general-purpose fine-grain SIMD FPSP (Carey et al., 2013). It has a 256×256 pixel array, and along with each pixel is a small Processing Element (PE). All 65,536 processors execute the same instruction at one time. In addition to 14 binary registers, each PE has analogue registers A through to F as well as a *NEWS* register. Each PE can also address an *XN*, *XE*, *XS*, and *XW* register that is actually that PE's respective neighbours' *NEWS* registers. Each PE uses an analogue bus to link its available analogue registers, and because values are stored as charge; analogue arithmetic is done directly on the bus that connects the registers rather than on a separate arithmetic unit.

Instructions in the architecture control how register values are let into and out of the bus with the caveat that values are inverted due to the nature of the analogue electronics. Each macro instruction like `add`, `sub`, and `mov` are made of multiple bus instructions that create the desired behaviour, where the `bus`$n(w_1, ..w_n, r_0..r_k)$ instruction has the general rule that the values of registers $r_0..r_k$ are summed up, negated, and divided equally between the n receiving-registers $w_1..w_n$. Since a bus operation directly controls which registers are opened to the PE's common analogue bus, a register may only appear once in each `bus` instruction. Each bus instruction also incurs significant noise and error factors, especially for `bus2` and `bus3` (Chen, 2020).

Macro instruction arguments are written as if they are assignment statements. For example; the macro instruction `add(A, B, C)` means $A := B + C$ and is made up of two bus instructions: `bus(NEWS, B, C)` meaning the *NEWS* register now contains the value of $-(B + C)$; and then `bus(A, NEWS)` so that register A contains $B + C$. We can see here that the `add` instruction has additional constraints, such that the two operands cannot be the same register, and that the *NEWS* register is overwritten, and left containing $-(B + C)$ as a side effect. When using macro instructions, we restrict the registers to A to F, and allow the macros themselves to make use of the *NEWS* and neighbouring *NEWS* registers for us by means of a direction value. We use subscripts to denote the registers of neighbouring PEs. For example: `mov2x(A, B, north, east)` computes $A := B_{\text{north,east}}$ in two `bus` instructions: `bus(XS, B); bus(A,XE)`. The first means that $XS_{\text{north,east}} := B_{\text{north,east}}$ which is equivalent to $NEWS_{\text{east}} := B_{\text{north,east}}$ and then the second instruction means $A := XE = NEWS_{\text{east}} \implies A = B_{\text{north,east}}$.

While interesting uses of the `bus` instructions exist, allowing adding and subtracting from neighbouring PEs, individual macro instructions are still highly restricted in comparison to most modern instruction sets. Only primitive analogue operations are available to each PE such as: Move, Add, Subtract, Divide by two, and to acquire the value from the sensor (Chen, 2020). The lack of a multiplication instruction means the problem of generating convolutional filter code for SCAMP-5 builds on the theory of multiplier-free FIR filters (Chandra and Chattopadhyay, 2016).

The chip has been shown to be capable of operating at 100,000 FPS, largely because it is not limited by the speed of

[2] This paper is partly based on our earlier workshop paper (Stow et al., 2022), extended with the collision avoidance application example.

an output bus to transfer all the pixel data (Carey et al., 2012). Instead of only offering an analogue or digitally encoded output of all pixels at a time, like traditional camera sensors, the SCAMP-5 architecture allows binary outputs per pixel, and even event driven outputs. This allows each PE to come to a judgement on its input pixel data and fire its own event that sends the coordinates of the PE to the host; allowing information transfer without divulging the actual image.

The architecture uses an off-chip controller to manage the fetch-decode-execute cycle, with every pixel's processor receiving the same instruction, making it a single-instruction-multiple-data (SIMD) design. This has benefits in terms of simplicity and efficiency as none of the Processing Elements need to be able to fetch instructions for themselves. There is also provision for masking pixels such that only selected PEs execute instructions.

One important consideration to be made when using and designing algorithms related to the SCAMP-5 chip is noise introduced by the nature of the analogue computation. Every use of the 7 analogue registers introduces noise to the values stored. This makes finding optimal code to perform the convolutions ever more vital for accurate results.

3 Related work

In this section we will look briefly at alternative systems to perform convolutional kernels on SCAMP-5 as well as relevant robotic navigation systems.

3.1 Convolutions on SCAMP-5

Given an $N \times N$ convolutional kernel, AUKE's reverse-split algorithm generates code for SCAMP-5 which applies the kernel efficiently to the captured image on the focal-plane using analogue computation. AUKE is, however, limited to compiling just a single convolutional kernel at a time using a reduced instruction set that omits the more powerful instructions available in SCAMP-5. AUKE's reverse split algorithm produces a data-dependency graph of 'elemental' operations which broadly captures common sub-expressions but is restricted to intermediate results whose values are a subset of the original kernel's values. This is then further optimised with a graph re-timing algorithm that aims to reduce the computation by relaxing that previous constraint. Instructions can then be selected and scheduled, and registers allocated, from this data-flow graph.

A method for computing binary weighted convolutional kernels on the SCAMP-5 FPSP is demonstrated in Liu et al. (2020). Their approach stores the kernel coefficients, -1 or 1, in the digital registers of the PEs. They repeat a shift-accumulate procedure predicated on the weights to the image. The method works for convolutions with a stride equal to the size of the kernel; denser strides are performed by shifting the kernel weights and repeating the shift-accumulate algorithm. This method allows different kernels to be run in different parts of the sensor but is limited to binary weights as memory is limited and always has worst case performance characteristics regardless of sparsity or potential common sub-expressions.

3.2 Robotic navigation

There are various low-power robot navigation methods and implementations for more traditional processor, as well as SCAMP-5. Many primarily target drones and others are designed for ground vehicles in preset courses.

(Giusti et al., 2016) implemented a network for drone navigation in a forest trail using camera input. The network has a total of 4 convolution layers and a Maxpooling layer between each convolution layer, followed by a fully-connected layer of 200 neurons and a final layer of 3 neurons for navigation prediction.

(Loquercio et al., 2018) implemented DroNet, a network that allows a UAV to successfully fly at high altitudes and in indoor environments. The network consists of a ResNet-8 with 3 residual block, followed by dropout and ReLU. This is then split into 2 separate fully-connected layers with 1 neuron each, one for steering and one for collision probability.

(Kim and Chen, 2015) implemented a network for indoor drone navigation. It has a total of 5 convolution layers with pooling and ReLU after each convolution, followed 2 fully-connected layers back-to-back of 4096 neurons each, followed by an output layer of 6 neurons.

While these networks are effective in their own fields, transferring them to SCAMP-5 is not practical as the networks are too large, both increasing the noise accumulated, and requiring too much memory to store the activations of the neurons. Nevertheless, these works are insightful benchmarks for small and efficient vision based robot navigation.

Other works which utilise SCAMP-5 for robotics navigation do exist. (Greatwood et al., 2019) performs drone racing with SCAMP-5, using the FPSP to efficiently detect the gates. This means the only data transferred off the sensor is the gate's size and location. The on-sensor processing and the minimal data transfer enables the gate detection to operate at 500 FPS. SCAMP-5 has also been employed as a visual sensor for agile robot navigation that allows ground vehicles to drive around a pre-set course of gates (Liu et al., 2021a). The gates are labelled with predetermined patterns that enable the SCAMP-5 to generate control signals for the ground vehicle. The proposed method achieved 200 FPS in an indoor setting and 2000 FPS with outdoor lighting conditions. These systems depend on clear visual cues, and their algorithms are tailored to detect particular patterns and fiducial markers.

(Chen et al., 2020) use features computed from conventional vision algorithms such as motion parallax, and static and dynamic corners, to feed a recurrent neural network (RNN) that outputs proximity distance to any nearby obstacles thus allowing for obstacle avoidance navigation. They were able to achieve about 250FPS for the full system in an indoor setting.

A CNN based method is proposed in (Liu et al., 2021b) for mobile robot localisation and tracking. A binary weighted CNN is directly implemented on the SCAMP-5 vision chip to extract features that determine a rover's position out of 64 positions in the simulated environment.

4 Cain

Cain is a framework for compiling convolutional filters, designed to search through a configurable Focal-plane Sensor-processor Array (FPSP) instruction set to find efficient code. A fundamental concept Cain uses is to only consider a single arbitrary PE in the FPSP, and perform everything relative to it. This works for SIMD architecture like SCAMP-5 because every PE will be executing the same steps synchronously in parallel. The assumption we make when producing code is that the neighbours of our arbitrary PE will exist and so will have done the same work but at a relative offset in the input image. The aim is to search through the graph of possible Processing Element states in such a way that common sub-expressions in the given kernels are exploited and used to reduce the cost of any path from initial to final PE states. To do this Cain searches backwards, starting with a set of final kernels, these are the convolutional filter, and applying instructions in reverse to simplify the kernels until only the identity kernel[3] is left. Figure 2a shows a high level overview of this process. Searching backwards is a design choice that makes the search more effective because it means the aim at each step is to make what needs to be solved simpler than before. This means heuristics can be produced to always direct the search towards the identity kernel rather than a system of heuristics trying to accurately predict the path towards an arbitrary set of final kernels. We present this as a dynamic graph search problem because the size of the graph is intractable. Given the AnalogNet2 filter in Eq. (1), Cain identifies 37163 potential child nodes in the first step alone. This can be reduced to 239 if we are willing to accept a less than exhaustive search of the solution space. This restriction is applied when the computational cost of computing the full set of child nodes is too high, which is often the case early in the search process.

[3] Single-entry matrix. Not to be confused with identity matrix.

4.1 Definitions

This section provides an overview of notation and definition used in this paper. Cain is designed such that different definitions could be used without changing the fundamental search algorithm but the definitions we use here to explain Cain for SCAMP-5 are based largely on AUKE's, which provide an elegant way to conceptualise the convolutional kernels without multiplication.

Example 1 We will look at a simple example of how a convolutional kernel is represented in Cain. Here we use AnalogNet2 (Wong et al., 2020; Guillard, 2019) which is a CNN designed for SCAMP-5.

$$AnalogNet2 = \left\{ \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ -3 & 1 & 0 \\ -3 & 0 & 2 \end{bmatrix}, \frac{1}{4}\begin{bmatrix} -4 & -1 & -1 \\ -1 & 2 & 0 \\ 1 & 1 & 0 \end{bmatrix}, \frac{1}{4}\begin{bmatrix} -1 & 2 & 0 \\ -1 & 1 & -3 \\ 0 & -3 & 0 \end{bmatrix} \right\} \quad (1)$$

Since SCAMP-5 does not have multiplication we must approximate the kernel and because it does have division-by-two instructions the natural approximation to make is to find the nearest integer multiple of $\frac{1}{2^d}$ for each coefficient in the kernel, given some number of divisions d. In our example we have already extracted the common denominator such that $d = 2$ and this perfectly represents the kernel. The larger d is, the larger the search space and complexity of the problem, so d can be limited to allow an acceptable amount of approximation error such that the resulting program is shorter and computational expense of compiling it is reduced.

Definition 1 Let an Atom, denoted as $(x, y, z, sign)$, be a representation of $\frac{1}{2^d}$ of a pixel value at coordinate x, y, on the zth channel. x, y are coordinates relative to the arbitrary PE and so also the centre of the kernel, and z refers to an image input channel. The sign is used to negate the value if necessary.

Definition 2 Let a Goal, denoted as $\{atom_1, atom_2, ...\}$, be a multi-set of Atoms. The Goal represents an arbitrary kernel, however, scaled by 2^d. The aggregate of the values represented by each of the Atoms yields the same result as applying the scaled kernel.

Representing a convolutional kernel as a Goal is a convenient way to support multiply-free instruction set, such as SCAMP-5. One can simply view this as unrolling the multiply instruction into additions. Using Goals simply re-frames the problem by scaling everything by 2^d, and approximating coefficients to the nearest number of Atoms.

Definition 3 Let a Goal-Bag, denoted as $\{goal1, goal2, ...\}$, be a multi-set of Goals. The Goal-Bag is used to capture the

state of our arbitrary PE. This includes defining the Final-Goals, the set of convolution kernels we wish to compute; and the Initial-Goals, the set of Goals which the computation will start from.

Using these definitions of Goals and Atoms we see that the first kernel from Example 1 can be represented by G

$$K = \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ -3 & 1 & 0 \\ -3 & 0 & 2 \end{bmatrix}, \tag{2}$$

$$G = \left\{ \begin{matrix} (-1,0,0,-), & (-1,0,0,-), & (-1,0,0,-), \\ (0,0,0,+), & (-1,-1,0,-), & (-1,-1,0,-), \\ (-1,-1,0,-), & (1,-1,0,+), & (1,-1,0,+) \end{matrix} \right\} \tag{3}$$

As our Goal notation is verbose, we provide a compact version that disambiguates Goals from kernels

$$G = \left\langle \begin{matrix} 0 & 0 & 0 \\ -3 & 1 & 0 \\ -3 & 0 & 2 \end{matrix} \right\rangle \implies \frac{1}{2^2}\begin{bmatrix} 0 & 0 & 0 \\ -3 & 1 & 0 \\ -3 & 0 & 2 \end{bmatrix} \star \text{Image Input}$$

where the $\star$ operator applies the left-hand convolutional kernel to the right-hand array (4)

By repeating this for process the rest of the convolutional kernels in the AnalogNet2 filter, the Final-Goals Goal-Bag $\boldsymbol{FG}$ is produced:

$$\boldsymbol{FG} = \left\{ \left\langle \begin{matrix} 0 & 0 & 0 \\ -3 & 1 & 0 \\ -3 & 0 & 2 \end{matrix} \right\rangle, \left\langle \begin{matrix} -4 & -1 & -1 \\ -1 & 2 & 0 \\ 1 & 1 & 0 \end{matrix} \right\rangle, \left\langle \begin{matrix} -1 & 2 & 0 \\ -1 & 1 & -3 \\ 0 & -3 & 0 \end{matrix} \right\rangle \right\} \tag{5}$$

Since, in our example, $d = 2$; the Goal representation of the identity kernel (G_{ID}) that makes up the Initial-Goals, is based on the approximation of the Final-Goals:

$$K_{ID} = \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 0 \end{bmatrix} \implies G_{ID} = \left\langle \begin{matrix} 0 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 0 \end{matrix} \right\rangle \tag{6}$$

Moving a value around the processor array is expressed by translating every Atom of a Goal. Addition and subtraction can be expressed by combining two Goals into one, making sure to cancel out positive and negative Atoms with the same coordinates. Since Cain searches backwards, we apply these operations in reverse. For 2-operand addition this means we take a Goal, G, that we wish to generate code for, then produce 2 new Goals that when added together produce G. Defining Goals as multi-sets of Atoms makes this process intuitive as we can simply split the Atoms between two Goals in every possible permutation (or fewer if we are willing to assume some are non-optimal, or willing to miss potentially better code for the sake of more efficient code generation). This definition also restricts the reverse search process since when splitting a Goal we cannot split an Atom. It follows that one way to naively search backwards to find a solution that computes G is to split G between the 4 coordinates populated with Atoms such that they can be added together (a Goal for each colour in 2). Then for each of these 4 Goals we can translate them such that all the atoms are in the centre of the kernel. For example we read the value of the red Atoms in G from the west thus translating the Atoms eastwards. We see that the red and green parts of G are now the same and so only need calculating once and this can be done by negating that Goal then splitting the 3 Atoms into Goals containing 1 and 2 Atoms each. Finally we can use the divide instruction which, in reverse, will double the number of Atoms from 1 to 2 and finally to 4, which gives us the identity Goal G_{ID}. The resulting code is then simply these steps reversed to produce a usable program.

4.2 Search strategy

Cain's reverse search algorithm works iteratively taking the state of an arbitrary PE, defined as a Goal-Bag:

$$\boldsymbol{F} := \{G_1, G_2, G_2, G_3 ...\} \tag{7}$$

This is a node in our search graph and represents the state we aim to achieve by executing the instructions that form a path from the initial-Goals to this node. In the search graph, nodes are generated dynamically as the graph is explored. Figure 2b shows a simplified view of how a graph might look as it is generated and searched. We simplify the exploration such that in each iteration of the search algorithm we produce a Goal-Bag Pair of an ***Uppers*** Goal-Bag and a ***Lowers*** Goal-Bag as well as an instruction, with the following constraints:

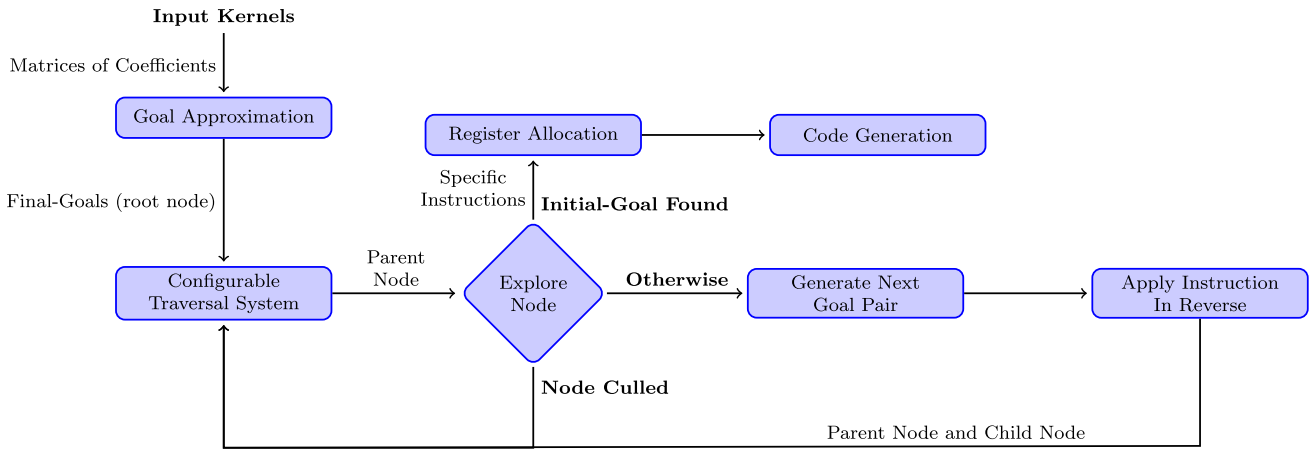
$$(\boldsymbol{U}, \boldsymbol{L}), inst = nextPair(\boldsymbol{F})$$
$$\text{where } \boldsymbol{U} \subseteq \boldsymbol{F},\ \boldsymbol{U} = inst(\boldsymbol{L}) \tag{8}$$

The new child node, $\boldsymbol{C}$, is then produced by applying the instruction in reverse using the following rule, with the instruction becoming an edge in the graph:

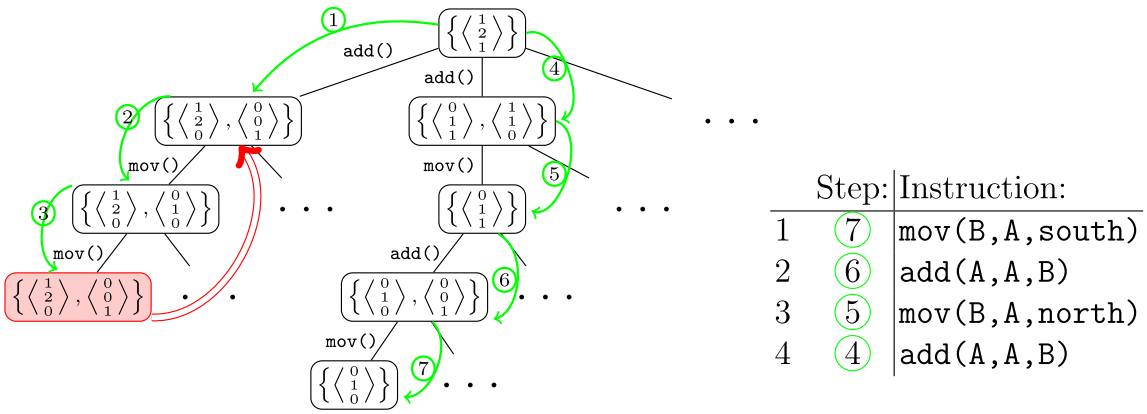
$$\boldsymbol{C} = (\boldsymbol{F} \backslash \boldsymbol{U}) \cup \boldsymbol{L} \tag{9}$$

Following our AnalogNet2 example from Eq. (5), the first iteration of the search algorithm will start with $\boldsymbol{FG}$ and the Pair of Goal-Bags Cain produces is as follows:

$$\boldsymbol{U} = \left\{ \left\langle \begin{matrix} -1 & 2 & 0 \\ -1 & 1 & -3 \\ 0 & -3 & 0 \end{matrix} \right\rangle \right\}, \tag{10}$$



(a) Cain System Overview.



(b) Example Cain graph.

Fig. 2 **a** shows an overview of the Cain system. **b** is a graph showing how Cain might search a simplified 1-dimensional problem using CGDS. Numbered steps show the order that the paths are explored with child nodes generated the first time a search step starts at a parent node. Nodes are checked for being the Initial-Goal when pointed too. The red node, and edge, correspond to a dead-end where a duplicate node has been found at a higher cost than previously seen and so the node is not traversed further. We see a path to the Initial-Goal is found after 7 steps, and the code produced by this path is presented on the right. The `mov()` instruction in step 5 exploits a common sub-expression such that the two Goals in its output Goal-Bag are produced together, thus shortening the code

$$\boldsymbol{L} = \left\{ \left\langle \begin{matrix} -1 & 2 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{matrix} \right\rangle, \left\langle \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & -3 \\ 0 & 0 & 0 \end{matrix} \right\rangle, \left\langle \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & -3 & 0 \end{matrix} \right\rangle \right\} \tag{11}$$

$$inst = \boldsymbol{U} \leftarrow add(L_1, L_2, L_3) \tag{12}$$

$$\boldsymbol{C} = \left\{ \begin{matrix} \left\langle \begin{matrix} 0 & 0 & 0 \\ -3 & 1 & 0 \\ -3 & 0 & 2 \end{matrix} \right\rangle, \left\langle \begin{matrix} -4 & -1 & -1 \\ -1 & 2 & 0 \\ 1 & 1 & 0 \end{matrix} \right\rangle, \\ \left\langle \begin{matrix} -1 & 2 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{matrix} \right\rangle, \left\langle \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & -3 \\ 0 & 0 & 0 \end{matrix} \right\rangle, \left\langle \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & -3 & 0 \end{matrix} \right\rangle \end{matrix} \right\} \tag{13}$$

The multi-set semantics here mean that if the Goals in $\boldsymbol{L}$ are all already part of $\boldsymbol{F}$ then the number of Goals to solve is reduced, and so by applying more pairs $(\boldsymbol{U}, \boldsymbol{L})$ we traverse the graph of Goal-Bags, until we reach the initial-state, where the only Goal in the Goal-Bag is the identity Goal. In our example (Eq. 10) we see that the sub-expression of 3 negative Atoms is reused in C_4 and C_5 since applying a `mov2x` next could eliminate C_5 from $\boldsymbol{C}$. There is also further potential to reuse this by how we split C_1. Once the initial Goal-Bag is found the path from the initial Goal-Bag back to the Final-Goals becomes the list of instructions that form our generated program.

After this point Cain continues searching for shorter paths, and can cull any nodes with longer paths. During the search the same Goal-Bags may be reproduced in different ways, we cull the current node any time a Goal-Bag is produced

Algorithm 1: Child Generator Deque Search

```
Input: s
deque ← [(s, null)]
while deque ≠ [] do
    n, g ← deque[0]
    deque ← deque[1..]
    if g = null then
        do node computation on n
        g ← childGenerator(n)
    end
    c ← g.yield()
    if c ≠ null then
        deque ← [(c, null)] + deque + [(n, g)]
    end
end
```

that has already been seen at a lower or equal cost, or if the Goal-Bag has more Goals than available registers.

The second part of the search strategy defines the search order. Each invocation of the reverse search algorithm produces one new node, $\boldsymbol{C}$, and the input node is incremented to know how many of its children have been produced so far. Cain uses this simple definition to allow several graph traversal algorithms to be implemented. Using Depth-First-Search (DFS), Cain can simply maintain a stack of the nodes. On each cycle the top node is popped off the stack and given to the reverse search algorithm. Then the incremented parent node is put back on the stack, followed by the new child node.

While DFS performs well in AUKE, it struggles in Cain because the number of child nodes at every level is far greater, since each edge is only one instruction and there are multiple kernels to consider. This means the size of the graph we would like to search is much larger and we are unable to search even a small fraction of it. To overcome this we use a graph-traversal algorithm that, for our purposes, we call Child Generator Deque Search (CGDS). The aim of this algorithm is to ensure that the search does not end up 'trapped' in one small part of the graph, but can effectively search traverse many children of many of the nodes that are found where DFS will search all of the children of nodes at the extent of the paths it searches before searching the second children of nodes earlier in the graph. Algorithm 1 shows a pseudo-code implementation of CGDS. In each cycle the front of the queue is polled, if the node has not been seen before, Cain checks to see if it can be directly transformed from the initial-state Goal-Bag, this is the 'node computation'. The node is then passed to the reverse search algorithm to attempt to produce the next new child node and to increment parent node—this is implicit in calling '*yield()*' on g. The child node, if it exists, is put on the front of the queue and the incremented parent node is put on the back. We do not claim that CGDS is novel, but we have found it superior to obvious alternatives, and the strategy used in (Barthels et al., 2019); for details see (Stow, 2020).

4.3 Cost function

In the reverse search algorithm we see that the pairs of ***Uppers*** and ***Lowers*** are produced one at a time. While this simplification allows us to produce more generic graph traversal implementations; what allows Cain to efficiently find solutions, are the heuristics that allow us to order the pairs that are produced for a node from the most promising to the least. This type of heuristic provides the order of siblings to search so we call it a 'local heuristic'. It doesn't compare nodes in different parts of the graph, which we would call a 'global heuristic'. We found that we were unable to find effective global heuristics because traversal algorithms that take advantage of such heuristics end up producing huge frontier sets of nodes making the memory requirements too large. The use of local heuristics drives the SCAMP-5 code generation in Cain instead, though support for best-first-search with global heuristics is available in Cain. The local heuristics used for SCAMP-5 are based on generating every child node of the parent and then ordering them based on a cost function. There are 3 main components considered for the cost: Atom distance, repeated Goals, and divisions. A simplified formula is shown in Eq. (14).

$$cost(\boldsymbol{C}) = \begin{array}{l} total(\boldsymbol{C}) + dists(\boldsymbol{C}) \\ +reps(\boldsymbol{C}) + divs(\boldsymbol{C}) \end{array} \tag{14}$$

$$total(\boldsymbol{C}) = \sum_{G \in \boldsymbol{C}} |G| \tag{15}$$

$$dists(\boldsymbol{C}) = \sum_{G \in \boldsymbol{C}} \sum_{a \in G} ((|a.x| + |a.y| + n(a))\, s(G, \boldsymbol{C})) \tag{16}$$

$$n(a) = \begin{Bmatrix} 1 \text{ if } a \text{ is a negative Atom} \\ 0 \qquad \text{otherwise.} \end{Bmatrix} \tag{17}$$

$$s(G, \boldsymbol{C}) = \begin{Bmatrix} \frac{1}{2} \text{ if} \nexists B \in \boldsymbol{C}.G \subset B \\ 1 \qquad \text{otherwise.} \end{Bmatrix} \tag{18}$$

$$reps(\boldsymbol{C}) = \sum_{\substack{G \in \boldsymbol{C}.G \text{ is unique} \\ \text{wrt any translations}}} \begin{Bmatrix} |G|^2 \ \exists a, b \in G.a \neq b \\ 0 \qquad \text{otherwise.} \end{Bmatrix} \tag{19}$$

$$divs(\boldsymbol{C}) = \frac{2^d}{\min(multiplicity(a) \forall a \in G. \forall G \in \boldsymbol{C})} \tag{20}$$

The Atom distance part counts up how many Atoms every Goal in $\boldsymbol{C}$ has, and how far from the centre they are, with some relief if the Goal is a sub-Goal of another Goal in $\boldsymbol{C}$. The repeated Goals portion of the cost penalises $\boldsymbol{C}$ by the square of number of Atoms in each Goal, unless that Goal is equal to a translation of another Goal in $\boldsymbol{C}$. The divisions component penalises $\boldsymbol{C}$ for the number of division operations that would be required to produce the Goals from the identity-kernel Goal, G_{ID}.

The heuristics presented here are designed based on the use of the analogue functions in SCAMP-5, where the com-

plexity and throughput of additions and data movement are similar. Since the definitions of *inst* in Eq. (8) is so general as to accommodate many potential operations and modes of processing (digital and analogue), Cain can be easily extended to use a different set of instructions where analogue noise, for example, is not an issue. Such a change might then benefit from a revised heuristic cost function.

5 Evaluation

All performance evaluation in this section is conducted on an Intel Core i7-7700HQ CPU (4 cores, 8 threads) with a base frequency of 2.80GHz. The computer has 16GB of RAM, runs Ubuntu 18; as well as Java 1.8 (Oracle) and Python 3.6 to run Cain and AUKE respectively. The implementation of AUKE used, as developed by Debrunner, can be found on Github.[4] Cain source code can be found at https://github.com/ed741/cain.

5.1 Performance evaluation against AUKE

Comparison of our work Cain against AUKE is performed by comparing resulting code generated from the respective compilers, given the same input filters. Both compilers are given 60 seconds to find a solution using all 6 registers. Note as Cain supports multi-threading, it spawns 4 worker threads to perform the search.

As shown in Table 1, Cain significantly outperforms AUKE. Cain supports a wider set of instructions in contrast of AUKE, enabling generation of more efficient code. Not only this, the search strategy used by Cain is better than AUKE's, as shown in 5×5 Gaussian Kernel, were using the same set of instructions (Basic), code generated by Cain is half in length when compared to output of AUKE's. Although, in further testing, AUKE is able to produce less inefficient code for this kernel given fewer registers. When given multiple kernels, Cain is able to perform simultaneous kernel optimisation. For example when combining 3×3 and 5×5 Gaussian, unlike AUKE, Cain is implemented to utilise the common sub-expressions between the kernels, thus, generating shorter code than naively concatenating the code for each of the Gaussian kernels. Neither Cain or AUKE perform a compete exhaustive search.

The AnalogNet2 filter is the kernels used in AnalogNet2 (Wong et al., 2020; Guillard, 2019), which is a CNN for SCAMP-5, capable of MNIST digit recognition. Cain requires only 21 instructions whereas AUKE produces kernel code which has in total 49 instructions. Reduced code not only improves the execution time, but also reduces the noise build up, which is significant problem as discussed in (Wong et al., 2020). If the aim of finding sub-expressions is to eliminate redoing work, then the number of add and subtract operands is a proxy for how effective the search for sub-expressions is, regardless of how translations are handled. Table 2 shows that AUKE's code has 40 add or subtract operands whereas Cain's code has only 27. We have compared the runtime of AnalogNet2's convolution kernels, generated by AUKE and Cain on the physical SCAMP-5. Note, as AUKE produces code which performs invalid register manipulation, the fixed code as used in (Guillard, 2019), which executes on the device is 81 instructions long. The execution time of the code produced by AUKE and Cain for the convolution kernels were 35μs and 9μs respectively, showing almost 4 times speedup.

5.2 Effectiveness of the search strategy

If Cain has an effective heuristic we will quickly see a point of diminishing returns in code length, as Cain continues to search new nodes and takes more time. We can track the number of nodes that are explored before finding any plan in Cain, and so use this as a measure of the search strategy and heuristics that is more independent of physical compute performance. With this in mind we test the effectiveness of our heuristic by constructing 100 samples of randomly generated single kernel filters as in Eq. (21). Running Cain as per the following configuration—Maximum Nodes to Explore: 20,000, Maximum Search Time: 60s, Worker Threads: 1—allows us to collect as many plans as can be found in the given time limit. We then ran Cain again, but with Cain's SCAMP-5 heuristic disabled and replaced with a random sort. This allows us to compare Cains heuristics against an unaided benchmark.

$$\frac{1}{8}\begin{bmatrix} u_1 & u_2 & u_3 \\ u_4 & u_5 & u_6 \\ u_7 & u_8 & u_9 \end{bmatrix}$$
$$\text{Given } u_1..u_9 \text{ are integers sampled uniformly from the range } [0..8] \tag{21}$$

We found that Cain was unable to find any plan for any of the 100 sample filters without its heuristics, principally demonstrating that effective heuristics are required in Cain for any tangible progress to be made. We plot the lengths of the best plans found against the number of nodes expanded before the plan is found in Fig. 3. We can see that improvements are fewer and further between after the first 2500 nodes are explored. After this we see that we can expect at most a reduction equal to the reduction seen at 2500 for the rest of the nodes explored. This clearly demonstrates a point of diminishing returns for these filters. If the heuristic is effective we expect it to direct the search towards short plans first,

[4] https://github.com/najiji/auto_code_cpa/tree/75c017e5ad28c0f3f040fb9f84d7f8727d035baa.

Table 1 Kernels tested in AUKE and Cain

Name	Approximated filter	AUKE Basic	Cain All	Cain Basic
3 × 3 Gauss	$\left\{ \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right\}$	12	**10**	12
5 × 5 Gauss	$\left\{ \frac{1}{64}\begin{bmatrix} 0 & 1 & 2 & 1 & 0 \\ 1 & 4 & 6 & 4 & 1 \\ 2 & 6 & 10 & 6 & 2 \\ 1 & 4 & 6 & 4 & 1 \\ 0 & 1 & 2 & 1 & 0 \end{bmatrix} \right\}$	50	**19**	25
5 × 5 and 3 × 3 Gauss	$\left\{ \frac{1}{64}\begin{bmatrix} 0 & 1 & 2 & 1 & 0 \\ 1 & 4 & 6 & 4 & 1 \\ 2 & 6 & 10 & 6 & 2 \\ 1 & 4 & 6 & 4 & 1 \\ 0 & 1 & 2 & 1 & 0 \end{bmatrix}, \frac{1}{64}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 8 & 4 & 0 \\ 0 & 8 & 16 & 8 & 0 \\ 0 & 4 & 8 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \right\}$	(50 + 12)	**26**	39
AnalogNet2	$\left\{ \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ 3 & 1 & 0 \\ 3 & 0 & 2 \end{bmatrix}, \frac{1}{4}\begin{bmatrix} 4 & 1 & 1 \\ 1 & 2 & 0 \\ 1 & 1 & 0 \end{bmatrix}, \frac{1}{4}\begin{bmatrix} 1 & 2 & 0 \\ 1 & 1 & 3 \\ 0 & 3 & 0 \end{bmatrix} \right\}$	(13 + 21 + 15)	**21**	30

Values on the righthand side of the table refer to the number of SCAMP-5 macro instructions in the programs generated by AUKE and Cain for each filter. AUKE can only use the 'basic' macro instructions, so Cain is run twice; to compare its effectiveness under the same restrictions as AUKE. Since AUKE does not offer a way to compile multiple kernels at once, values for each kernel are given separately

Table 2 Comparison of Code for the AnalogNet2 filter generated by AUKE and Cain

AUKE			Cain
Kernel 2	Kernel 3	Kernel 1	
`1 mov(B,A);`	`22 mov(C,A);`	`38 divq(A,A);`	`1 diva(A,D,E);`
`2 divq(B,B);`	`23 divq(C,C);`	`39 divq(A,A);`	`2 div(D,E,C,A);`
`3 divq(B,B);`	`24 divq(C,C);`	`40 movx(D,A,west);`	`3 movx(E,D,west);`
`4 movx(C,B,north);`	`25 movx(D,C,south);`	`41 neg(D,D);`	`4 movx(C,E,north);`
`5 neg(C,C);`	`26 neg(D,D);`	`42 movx(E,D,south);`	`5 neg(F,E);`
`6 neg(D,C);`	`27 movx(E,C,east);`	`43 add(D,D,E);`	`6 subx(B,F,east,A);`
`7 movx(E,D,west);`	`28 sub(D,D,E);`	`44 add(E,A,D);`	`7 addx(E,E,D,south);`
`8 neg(E,E);`	`29 movx(E,C,north);`	`45 movx(A,A,south);`	`8 add2x(D,F,D,north,north);`
`9 add(F,B,E);`	`30 add(E,E,D);`	`46 movx(A,A,east);`	`9 sub2x(F,D,south,south,C);`
`10 movx(B,D,east);`	`31 add(D,D,D);`	`47 add(A,D,A);`	`10 add2x(D,C,D,east,south);`
`11 add(B,B,E);`	`32 add(D,E,D);`	`48 add(A,A,A);`	`11 add(E,E,D);`
`12 movx(D,E,south);`	`33 movx(E,C,west);`	`49 add(A,E,A);`	`12 movx(D,A,north);`
`13 movx(D,D,south);`	`34 sub(C,C,E);`		`13 add2x(A,C,A,east,east);`
`14 sub(B,B,D);`	`35 add(D,D,C);`		`14 movx(C,B,east);`
`15 add(B,B,F);`	`36 movx(C,C,north);`		`15 add(D,F,D);`
`16 add(B,C,B);`	`37 add(C,D,C);`		`16 add2x(F,F,E,east,south);`
`17 movx(C,C,west);`			`17 movx(E,B,south);`
`18 add(B,B,C);`			`18 addx(A,B,A,south);`
`19 movx(C,F,south);`			`19 addx(A,B,A,west);`
`20 add(B,C,B);`			`20 add2x(B,F,B,north,west);`
`21 add(B,B,F);`			`21 add(C,D,C,E);`

The Input Register is 'A' and the output registers for the 3 kernels are 'A','B','C' respectively. For AUKE, kernel 2 is run first since testing showed it was longest so this gives AUKE more registers to use

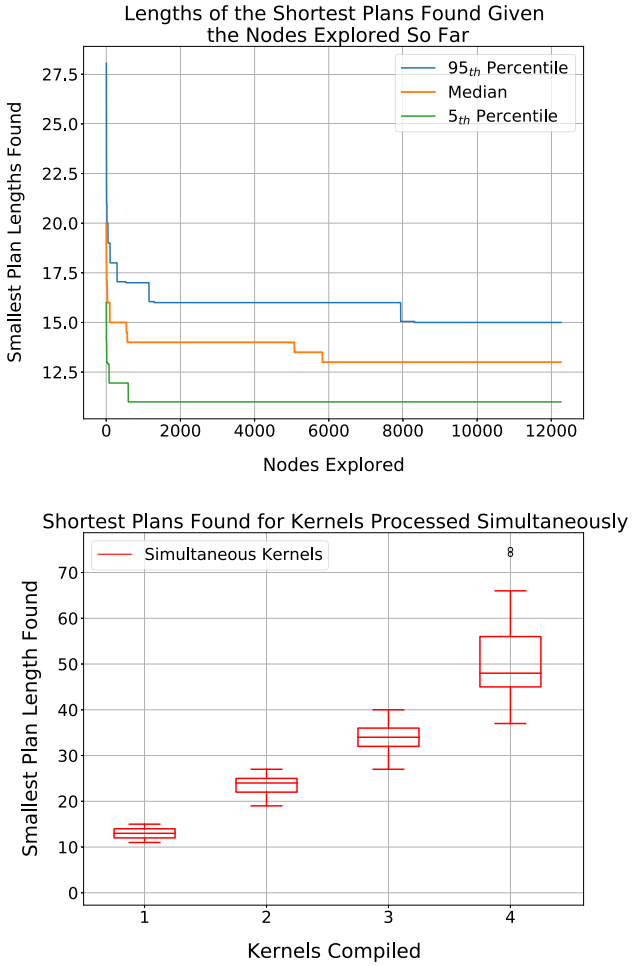


Fig. 3 Left: Graph showing the median number of instructions in the best plans found before n nodes have been explored by Cain. With 100 samples of randomly generated singular 3×3 kernel filters. Right: Graph showing the number of instructions in the shortest programs found by Cain for filters with 1, 2, 3, and 4 random 3×3 kernels. 25 samples were produced for each kernel count

and try instructions less likely to be optimal later. This model fits the data well as we see short plans are found quickly, and while improvements can be made, it is clear that they are found less often as the search continues.

A perfect heuristic would be able to direct the search straight to the globally optimal solution, so clearly our heuristic is imperfect. There are many situations when compiling large and sparse kernels where our heuristic incurs excessive computational overhead that produces a poor balance between high-quality navigation through the search space and simply searching more nodes to find a more optimal solution. This could be addressed with further analysis of potential heuristic functions or even a machine learning derived heuristic.

5.3 Effectiveness of the simultaneous Kernel optimisation

One of the significant features of Cain is to efficiently generate code for filters with multiple kernels, and do this simultaneously such that shared common sub-expressions can be reused. As it is possible for Cain to perform exhaustive searches for plans, given sufficient time, it will find a solution that simply computes the individual kernels independently, or find a solution with lower cost—utilising the common sub-expressions.

First, we wish to test whether the length of generated code is sub-linear to the number of input kernels. To test this, we again generate kernels using the using the method in Eq. (21). For kernel counts from 1 to 4 we generated 25 filters each and test them all using the same configuration as before except that we remove the maximum nodes explored constraint, and allow 4 worker threads. We plot the results in Fig. 3 and see that the results appear worse than linear, suggesting that common sub-expressions are not effectively being taken advantage of.

We hypothesise that the limited number of registers in the SCAMP-5 architecture is the major limiting factor in producing efficient code. To test this we increase the number of available registers to 18. For filters with 1 kernel up to 10 kernels we generate 10 samples each. Every kernel in the 100 filters is produced as in Eq. (21). For each sample, Cain compiles the kernels individually, given the appropriate number of registers such that other kernels in the filter would not be overwritten. Then we compile the kernels simultaneously using Cain. All compilations are given 60s to run, with 4 worker threads.

Figure 4 shows the results of this test. We see clearly that when register limitations are not a restricting factor Cain is able to consistently improve the performance of filter implementations by compiling them simultaneously. We see that improvements grow with more kernels, and it appears that the length of code generated for simultaneously compiled kernels increases sub-linearly. This supports the idea that with more kernels, ever more common-sub expressions can be exploited.

Since we are working with analogue computation in our evaluation, there is a limit to the length of code that could be run before the accumulated noise of each instruction compounds to make the results unreliable. This problem can be partially mitigated by reducing code length or logical depth as Cain does but still presents a challenge to complex and large kernels. Since Cain can be programmed to use an instruction set based on digital computation that does not suffer these problems, we can use these findings to inform the design of future CPA architectures that use digital and analogue computation.

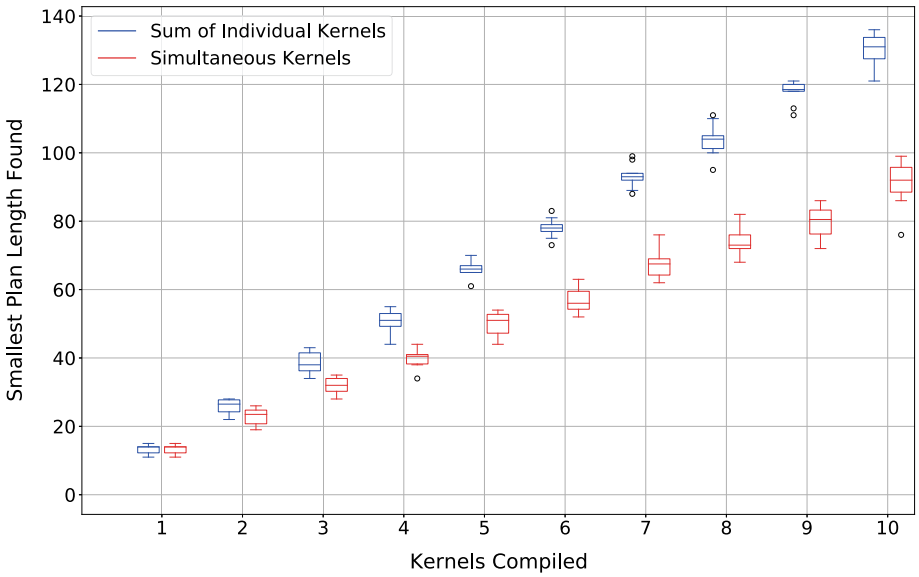


Fig. 4 Graph comparing the sum of the shortest SCAMP-5 code lengths found for kernels compiled individually, against the same kernels compiled simultaneously as one filter. For each filter a total of 18 registers were made available (more than in SCAMP-5) to reduce register availability as a limiting factor. In total 100 filters are produced, 10 for each number of kernels per filter. Each kernel is a randomly generated 3×3 kernel with coefficients uniformly selected in eighths from 0 to 1 (inclusive)

6 AnalogNavNet

To demonstrate the use-cases for Cain and Focal-Plane Sensor-Processors like SCAMP-5, we present AnalogNavNet, a convolutional neural-network based model for collision avoidance and robot navigation. AnalogNavNet has been physically implemented for a corridor and a race-track environment.[5] We evaluate the robotic navigation using a Jetson Nano, comparing the accuracy, inference time, and power consumption of this architecture as implemented on the FPSP (SCAMP-5), CPU (Quadcore ARM A57), GPU (128-core NVIDIA Maxwell), and a Visual Processing Unit (VPU, Intel Myriad Neural Compute Stick 2).

Our proposed method uses a CNN, similar to (Giusti et al., 2016) and (Kim and Chen, 2015), to learn directly from image features to navigate through an indoor corridor and race-track environment. The objective is similar to that of (Chen et al., 2020) in which they implement an RNN that learns from camera images along with range measurements from proximity sensors.

6.1 Network architecture

As shown in Fig. 5a, AnalogNavNet is split into two halves, with the convolutions, ReLU activations, and the pooling happening on the pixel-processor, and the fully-connected layer and Soft-max activations happening on the onboard micro-controller.

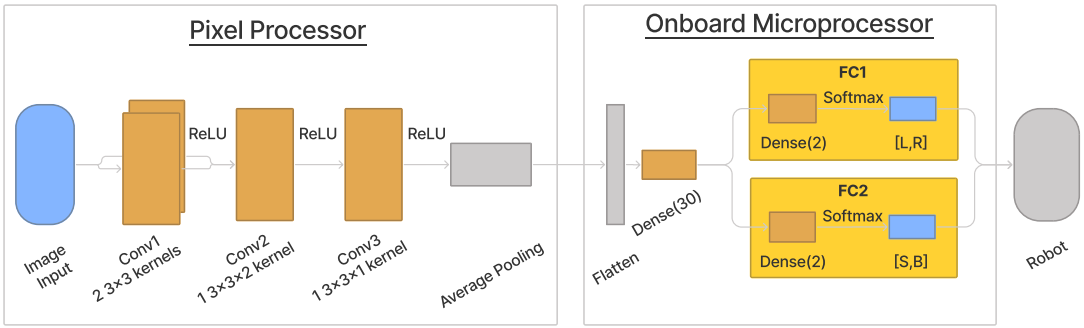
The Convolutional side of AnalogNavNet has a total of 4 kernels only (2 from first convolution layer, 1 from second convolution layer, and 1 from third convolution layer). Each convolutional kernel is of size 3×3 each, with the second layer convolutional kernel having a 2-channel input, with ReLU activations between each layer. Since we are targeting the SCAMP-5 FPSP, the initial image is of size 256×256 but the final feature map is average-pooled with strides of 32×32 to generate a final feature map of size 8×8. The feature map is passed onto the fully connected layer on the onboard micro-controller.

The pooling layer is aggregated into a single vector as part of being fed out of the FPSP, making use of standard pixel averaging functionality within the sensor. As the SCAMP-5 onboard micro-controller is very rudimentary and has a low clock frequency, the processing speed bottleneck happens here. Large fully-connected layers, while providing high accuracy, cannot be implemented on the SCAMP-5 micro-controller while retaining high frame rates. AnalogNavNet therefore uses a fully-connected layer with just 30 neurons. From here the network is split into two branches, each branch taking the 30 neuron outputs and using a simple dense layer with 2 neurons followed by a soft-max to aid the conversion from network prediction result to robot control instructions. These outputs encode turning left or right and forwards or stop on the two branches respectively.

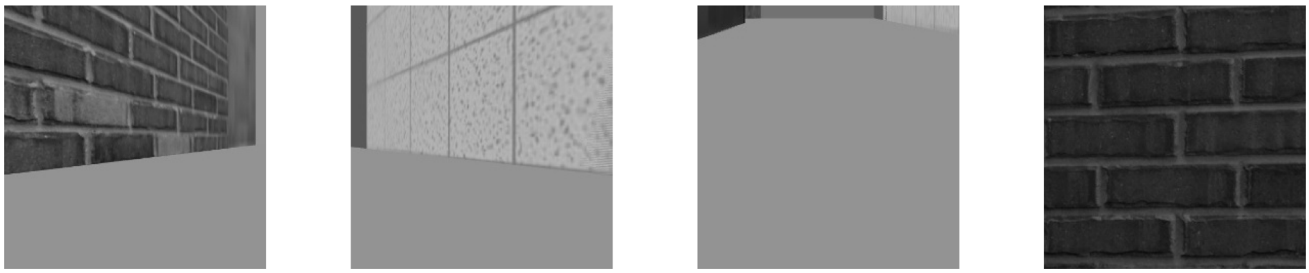
6.2 Network training

The network is initially trained using a labelled dataset we developed by simulating corridor environments in Robot Operating System (ROS) and Gazebo. Images are collected at various points within the simulated corridor maze, and the corridor walls vary in texture to allow the network to learn different features. Each captured image is divided and scaled into 4 256×256 sample images which are then labelled with one or more appropriate actions that the robot should take when they observe this part of the scene: go left, right, for-

[5] Video of robot can be found at cain.edstow.co.uk.



(a) AnalogNavNet



(b) Left **(c)** Right **(d)** Straight **(e)** Brake

Fig. 5 **a** AnalogNavNet: Convolutions are executed on the focal-plane, and the fully-connected layers on the onboard micro-controller. **b–e** Labelled images from the simulation dataset rendered using Gazebo in a custom corridor environment. Left label translates to a right turn, Right label to a left turn, Straight label translates to move forward, and Brake is to stop the robot

Table 3 Convolutional kernels of AnalogNavNet, produced by training the network, along with the length of code to compute them for SCAMP-5 as produced by Cain

Name	Kernel	Code Length
Conv1	$\frac{1}{8}\begin{bmatrix}4&3&4\\1&6&4\\1&4&4\end{bmatrix}, \frac{1}{8}\begin{bmatrix}0&3&4\\2&1&0\\7&0&2\end{bmatrix}$	24
Conv2 channel 1	$\frac{1}{8}\begin{bmatrix}1&1&0\\2&1&1\\1&0&1\end{bmatrix}$	10
Conv2 channel 2	$\frac{1}{8}\begin{bmatrix}1&2&2\\0&3&0\\0&3&3\end{bmatrix}$	9
Conv3	$\frac{1}{8}\begin{bmatrix}1&3&2\\4&4&3\\4&1&3\end{bmatrix}$	14

wards, and stop (see Fig. 5b–e). The 4 labels, split into the 2 branches of the network, are trained using binary cross-entropy to predict the appropriate actions. For validation data the textures in the simulation are changed and another set of samples produced for a total of $\sim$ 61,000 simulated training samples and $\sim$ 35,000 simulated validation samples.

During this initial training the fully-connected layer has 200 neurons, once trained and validated using the simulated environment the convolutional weights are frozen and the fully-connected layer is reset and reduced to final 30 neurons. The model is then retrained and achieves a validation accuracy of 96.36% for the Left or Right branch, and 79.6% for moving Forward or Stop branch. The network is then tested in a simulated environment using Cain to produce SCAMP-5 instructions for the convolutional kernels. A SCAMP-5 simulator running in ROS is used to ensure the behaviour is correct and the quantisation errors introduced by the multiply-free computation do not severely impact the model's performance.

With the first two convolution layer still frozen the network is further trained on a smaller 'real-world' dataset with 19,733 training samples and 9297 validation samples. Freezing the weights ensures that they do not converge into smaller values that are likely to exacerbate quantisation errors. The real-world data is made up of images of corridors all from the same building with the same pre-processing and labelling as before. The model achieves a validation accuracy of 93.8% for the Left or Right branch, and 78.9% for moving Forward or Stop branch.

Table 3 provides the approximated kernels compiled by Cain, along with the code lengths obtained as a reference for Cain's performance in a real-world example.

A second dataset was also collected for the race-track environment. The training process followed the same structure as

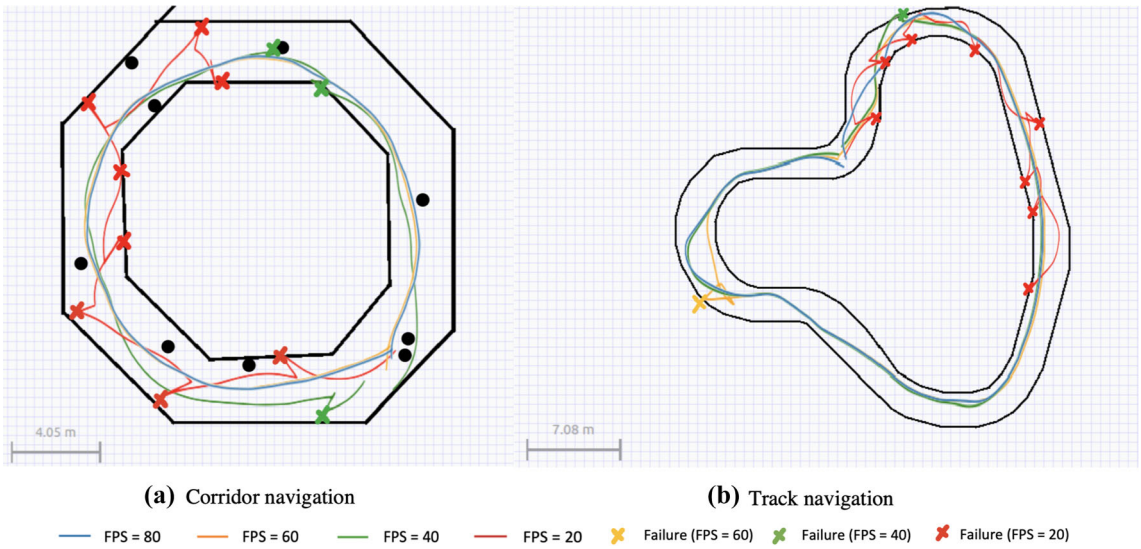


Fig. 6 **a** Trajectory in a corridor environment. **b** Trajectory in a track environment

the previous, with the only difference being is that the camera view had to be lowered in order to capture the track edges.

6.3 Network performance at different FPS

Two simulated experiments were performed to evaluate the performance of the network and robot at different FPS. For two environments, an octagonal corridor and a track, a network is trained as described in Sect. 6.2 up to the point preceding real-world samples. For each environment a TurtleBot3 Waffle Pi is implemented with the full-floating point precision network to autonomously navigate through the corridor at speed of 0.8m/s, and the track at 0.7m/s. The robot is simulated to process the environment at 80, 60, 40, and 20 FPS while the trajectories are recorded (see Fig. 6a and b). Trajectories of various FPS are plotted in solid lines in different colours and each cross represents a crash occurred during navigation. When the robot hits obstacles due to a lack of timely updates to the speed controller, the robot is manually driven away from the obstacles, moved back to a position before crashing happened, and set to continue navigating. This process was repeated for these failures until the robot completed one full lap or 8 crashes. With increasing FPS, the trajectory of the robot becomes smoother, resulting in fewer crashes during the navigation along the corridor and track.

6.4 Robot evaluation

In order to evaluate the robotic navigation, the Jetson Nano was mounted to a differential drive chassis (Fig. 7) and run in a straight corridor of approximately 25 m to test if it collides with the walls. Since the two motors are not identical,

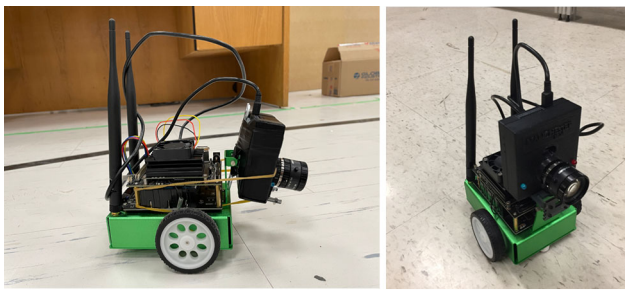


Fig. 7 **a** SCAMP-5 mounted for track experiment. **b** SCAMP-5 mounted for corridor experiment

there is always a slight left or right deviation, which the PID controller corrects using the outputs of the network. If the robot collides with the wall or starts travelling in the opposite direction, the run is considered a failure. The experiment was run total of 20 times for each hardware option, Table 4 shows the percentage of successful runs. For fair comparisons, the specification of the lens on the SCAMP-5, and a webcam used for the CPU, GPU, VPU are kept similar. The angle of view of the lens on SCAMP-5 is 56.3×43.7 degrees; whereas, the field of view of the lens on the webcam is 54×41 degrees. Since the quantised weights are only a restriction for the SCAMP-5, the other processors use the full-floating point precision that the network was trained for.

On comparing the results of the same network running on a CPU, GPU, VPU, and the SCAMP-5 we find that the CPU performed poorly as the lower frame rate prevented early enough updates of the PID values, resulting in corrections happening too late. The GPU and VPU have very similar

Table 4 Robotic navigation success rate on different devices

Hardware	Navigation success	Kernel resolution
SCAMP-5	85%	3 Binary Places
VPU	80%	Full-Floating Point
GPU	85%	Full-Floating Point
CPU	60%	Full-Floating Point

The navigation tests were run on environments different from the training dataset corridors

Table 5 Per-frame computation time in milli-seconds, for AnalogNavNet architecture

Hardware	Inference time [ms]	FPS
SCAMP-5	11.76	85
VPU	22.72	31
GPU	26.26	28
CPU	49.84	17

For inference time, image data retrieval time is excluded for CPU, GPU, and VPU, while it effectively is zero for SCAMP-5

results, although the VPU results were subject to a systemic disadvantage caused by the physical mounting of the device.

6.5 Inference Time

For each of the SCAMP-5, VPU, CPU, and GPU, inference time was calculated by measuring the average over 5000 frames of per-frame computation time incurred on each system using their respective system utilities. Table 5 shows the recorded per-frame computation time excluding the cost of capturing and retrieving the image data from the camera for computation on the CPU, GPU and VPU. To achieve this, we subtracted 10ms from their inference times, corresponding to the average time it takes to capture a frame from the external camera and transfer it to the Jetson Nano. Despite this, the SCAMP-5 is $\sim 2\times$ faster than the VPU and GPU. These figures include the total time for inference; both convolutional part as well as the fully connected part, and the transfer of the features from the focal-place to the micro-controller in the case of SCAMP-5.

In practice, this data shows that not only are FPSPs effective at reducing the data retrieval bottleneck, but they enable lower latency computation regardless of data transfer rates. The FPSP does not suffer from this bottleneck as the image is processed directly in place at the pixel level. This shows the advantage of realising most of the computation directly on the focal-plane in an analogue manner; the image retrieval latency is simply reduced to zero.

6.6 Energy consumption

Per-frame energy consumption was determined by measuring the power drawn by each device divided by the recorded FPS. The power drawn by the GPU and CPU were measured using JetsonStats, and a USB power meter was used for the VPU and SCAMP-5. The SCAMP-5 power measurement includes the image sensor and focal-plane processing, while the CPU, GPU and VPU inference results do not account for the energy consumption required for image capture and transfer. Table 6 shows the power and energy draw of the system. In order to calculate the total energy used during inference, For the VPU and SCAMP-5, the Jetson Nano's power during inference, approximately 2.97 Watts, and the inference power by their respective devices, is divided by the recorded FPS. For the CPU and GPU, Jetson Nano's power during the inference is used, and is divided by the FPS. Both VPU and SCAMP-5 uses far less power and energy during inference than the GPU and CPU. As SCAMP-5 is a prototype device, there is no idle-mode for the device, so it is running as soon as the device is plugged in, but as the inference time of the SCAMP-5 is much faster than the VPU, the total energy per frame is better than a low-power VPU. Comparing SCAMP-5 to the rest of devices, the inference energy per frame consumption is approximately 42% of VPU, 15% of GPU and 8% of CPU. Although the inference power of VPU is slightly lower than SCAMP-5 by 0.69 W, SCAMP-5 has a large advantage on FPS that means the extra power cost is insignificant in terms of general performance.

6.7 Discussion

AnalogNavNet successfully proves the viability of Cain for compiling convolutional kernels used in basic navigation and obstacle avoidance. The models produced perform in the real world, but there are significant limitations present. While the SCAMP-5 is able to run the network at a much faster FPS than CPU, GPU and VPU, the network had to be modified in order to be accommodated, leading to loss in precision. Significantly deeper networks could not be implemented via this method due to the noise introduced by each operation. If more registers were available, larger networks could fit inside the system, but an increase in registers would also lead to an increase in energy cost, which is a trade-off with limited utility given the noise constraints. A faster onboard microprocessor would allow for faster dense layer operations, a proposition that would be easily possible in a commercial setting. One of the limiting factors in the real-world experiments is the physical limitation of the robot itself; the chassis is front-heavy and higher speeds lead to a constant wobble which adversely affects navigation and is not accounted for in the training process.

Table 6 Power consumption at idle and during inference for each of the systems tested, and energy per-frame based on the inference power and recorded FPS

Hardware	Idle power[W]	Inference power[W]	Inference energy per frame[mJ]
Nano+SCAMP-5	3.92	5.01	58.94
Nano+VPU	3.00	4.32	139.4
Nano using GPU	2.31	11.37	406.07
Nano using CPU	2.31	12.16	715.29

SCAMP-5 has no idle state as it is a prototype device so its idle-power is equivalent to sending `nop` instructions. The Nano power does not include power drawn by the camera

7 Conclusion

We have presented Cain, a compiler which produces SCAMP-5 instructions from a set of convolutional kernels. Although the effectiveness of simultaneous kernel optimisation is limited on the current iteration of the SCAMP-5, we demonstrate, that with the increased number of registers, the length of the output of Cain is sub-linear to the number of kernels given. We have conducted extensive comparison against AUKE, and we demonstrate that the code generated by Cain is more efficient, and exhibits almost 4x speed up when the generated kernel is executed on the SCAMP-5 device.

We have presented an end-to-end working example of robotic navigation using SCAMP-5 based on our AnalogNavNet model. We have evaluated the performance and energy efficiency of AnalogNavNet running on 4 types of processor and found that SCAMP-5 is significantly faster and uses less energy per-frame than the alternatives. We have presented compelling evidence that FPSPs are a promising technology for edge computation, and by providing easy to use, yet efficient code generation toolkit, we hope to accelerate the relevant research in this field.

Acknowledgements We would like to thank Piotr Dudek, Stephen J. Carey, and Jianing Chen at the University of Manchester for kindly providing access to SCAMP-5, and their support in our work. This work was partially supported by the Engineering and Physical Sciences Research Council (EPSRC), Grant reference EP/P010040/1

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., & Kudlur, M. (2016). Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation OSDI 16)*, (pp. 265–283).

Barthels, H., Psarras, C., & Bientinesi, P. (2019). Linnea: Automatic generation of efficient linear algebra programs. arXiv:1912.12924.

Bose, L., Chen, J., Carey, S. J., Dudek, P., & Mayol-Cuevas, W. (2017). Visual odometry for pixel processor arrays. In *2017 IEEE international conference on computer vision (ICCV),* (pp. 4614–4622).

Bose, L., Chen, J., Carey, S. J., Dudek, P., & Mayol-Cuevas, W. (2019). A camera that CNNs: Towards embedded neural networks on pixel processor arrays. In *Proceedings of the IEEE international conference on computer vision (ICCV),* (pp. 1335–1344).

Carey, S. J., Barr, D. R. W., Wang, B., Lopich, A., & Dudek, P. (2012). Locating high speed multiple objects using a scamp-5 vision-chip. In *2012 13th international workshop on cellular nanoscale networks and their applications,* (pp. 1–2).

Carey, S. J., Lopich, A., Barr, D. R. W., Wang, B., & Dudek, P. (2013). A 100,000 fps vision sensor with embedded 535GOPS/W 256 × 256 SIMD processor array. In *2013 symposium on VLSI circuits,* (pp. C182–C1830).

Chandra, A., & Chattopadhyay, S. (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), 212–226.

Chen, J. (2020). scamp5 kernel api macro analog.hpp file reference. https://scamp.gitlab.io/scamp5d_doc/.

Chen, J., Liu, Y., Carey, S. J., & Dudek, P. (2020). Proximity estimation using vision features computed on sensor. In *2020 IEEE international conference on robotics and automation (ICRA),* (pp. 2689–2695).

Chen, Y. H., Krishna, T., Emer, J. S., & Sze, V. (2016). Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-state Circuits, 52*(1), 127–138.

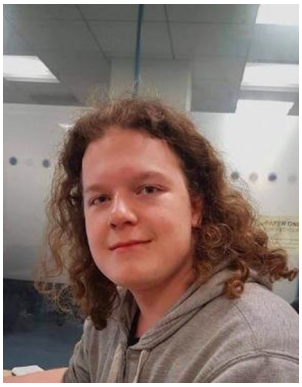
Debrunner, T., Saeedi, S., Bose, L., Davison, A. J., & Kelly, P. H. J. (2019a). Camera Tracking on Focal-Plane Sensor-Processor Arrays.

Debrunner, T., Saeedi, S., & Kelly, P. H. J. (2019b). AUKE: Automatic kernel code generation for an analogue SIMD focal-plane sensor-processor array. *ACM Transactions on Architecture and Code Optimization 15*(4).

Giusti, A., Guzzi, J., Ciresan, D. C., He, F. L., Rodriguez, J. P., Fontana, F., Faessler, M., Forster, C., Schmidhuber, J., Caro, G. D., Scaramuzza, D., & Gambardella, L. M. (2016). A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots. *IEEE Robotics and Automation Letters, 1*(2), 661–667.

Greatwood, C., Bose, L., Richardson, T., Mayol-Cuevas, W., Clarke, R., Chen, J., & Carey, S. (2019). Towards drone racing with a pixel processor array. In *11th international micro air vehicle competition and conference (IMAV),* (pp. 76–82).

Guillard, B. (2019). Cnns-on-fpsps. https://github.com/brouwa/CNNs-on-FPSPs/tree/c6b5c51839e9e3c453681e5b0a3e3ef541ba3cce.
Kim, D. K., & Chen, T. (2015). Deep neural network for real-time autonomous indoor navigation. arXiv:1511.04668.
LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86*(11), 2278–2324.
Liu, Y., Bose, L., Chen, J., Carey, S., Dudek, P., & Mayol-Cuevas, W. (2020). High-speed light-weight cnn inference via strided convolutions on a pixel processor array. In *British machine vision virtual conference 2020*.
Liu, Y., Bose, L., Greatwood, C., Chen, J., Fan, R., Richardson, T., Carey, S. J., Dudek, P., & Mayol-Cuevas, W. (2021). Agile reactive navigation for a non-holonomic mobile robot using a pixel processor array. *IET Image Processing, 15*(9), 1883–1892.
Liu, Y., Chen, J., Bose, L., Dudek, P., & Mayol-Cuevas, W. (2021b). Bringing a robot simulator to the scamp vision system. arXiv:2105.10479.
Loquercio, A., Maqueda, A. I., Del-Blanco, C. R., & Scaramuzza, D. (2018). DroNet: Learning to Fly by Driving. *IEEE Robotics and Automation Letters, 3*(2), 1088–1095.
Martel, J. (2019). Unconventional processing with unconventional visual sensing. PhD thesis, Institut National des Sciences Appliquées de Lyon.
Martel, J. N. P., Müller, L. K., Carey, S. J., Dudek, P., & Wetzstein, G. (2020). Neural sensors: Learning pixel exposures for HDR imaging and video compressive sensing with programmable sensors. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 42*(7), 1642–1653.
Murai, R., Saeedi, S., & Kelly, P. H. J. (2020). BIT-VO: Visual Odometry at 300 FPS using Binary Features from the Focal Plane. In *IEEE/RSJ international conference on intelligent robots and systems (IROS)*.
Saeedi, S., Bodin, B., Wagstaff, H., et al. (2018). Navigating the landscape for real-time localization and mapping for robotics and virtual and augmented reality. *Proceedings of the IEEE, 106*(11), 2020–2039.
Stow, E. (2020). Automatic code generation for simultaneous convolutional kernels on cellular processor arrays. Master's thesis, Imperial College London.
Stow, E., Murai, R., Saeedi, S., & Kelly, P. H. J. (2022). Cain: Automatic code generation for simultaneous convolutional kernels on focal-plane sensor-processors. In B. Chapman & J. Moreira (Eds.), *Languages and compilers for parallel computing* (pp. 181–197). Cham: Springer International Publishing.
Wong, M. Z., Guillard, B., Murai, R., Saeedi, S., & Kelly, P. H. J. (2020). AnalogNet: convolutional neural network inference on analog focal plane sensor processors. arXiv preprint arXiv:2006.01765.
XIMEA. (2021). xiB - PCI Express Cameras with high speed and resolution. https://www.ximea.com/pci-express-camera/pci-express-camera.
Zarándy, Á. (2011). *Focal-Plane Sensor-Processor Chips*. SpringerLink: Bücher, Springer, New York.



E. Stow is a PhD Student in the Software Performance Optimisation Group at Imperial College London, having completed a M.Eng degree in Computing at Imperial College in 2020.

A. Ahsan is a student at the Department of Electrical and Computer Engineering, Toronto Metropolitan University. He is a member of Robotics and Computer Vision Lab at Toronto Metropolitan University. His research interests include robotics, computer vision, and unconventional vision sensors.

Y. Li is a student at the Department of Mechanical and Industrial Engineering, Toronto Metropolitan University. He is a member of Robotics and Computer Vision Lab at Toronto Metropolitan University. His research interests include mechatronics, computer vision, and vision sensors.

A. Babaei is a master's student in the Department of Mechanical and Industrial Engineering at Toronto Metropolitan University. His research interests include computer vision and robotics, particularly perception in vision-aided navigation. His current research focuses on human-computer interaction and robotic navigation using FPSP.

R. Murai received M.Eng in Computing in 2019 from the Imperial College London. He is currently a PhD student in the Department of Computing at Imperial College London. His research interests include robotics and computer vision. In particular, the use of novel hardware and distributed computations.

S. Saeedi is an Assistant Professor at Toronto Metropolitan University. He received his PhD in Electrical and Computer Engineering from the University of New Brunswick, Fredericton Canada. He is currently working on semantic perception, bringing deep learning advances to robotic systems. His research interests span over simultaneous localization and mapping (SLAM), focal-plane sensor-processor arrays (FPSP), collaborative robotic systems, ground/aerial/marine robotics, and artificial intelligence and its applications in computer vision, robotics, and control systems.

P. H. J. Kelly has been on the faculty at Imperial College London since 1989, has a BSc in Computer Science from UCL (1983) and has a PhD in Computer Science from the University of London (1987). He leads Imperial's Software Performance Optimisation research group, working on domain-specific compiler technology.