Imperial College London

MASTERS THESIS

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

Automatic Code Generation for Simultaneous Convolutional Kernels on Cellular Processor Arrays

Author:
Edward John Stow

Supervisor:
Prof. Paul Kelly

June 15, 2020

Submitted in partial fulfilment of the requirements for the MEng in Computing of Imperial College London

ABSTRACT

Recent developments in cellular processor array technology have produced new low power processor architectures for computer vision applications where one of the most fundamental computational operations is convolutional filters. In this thesis we develop a new framework, Cain, for compiling convolutional filters into code for CPAs with the aim to optimise code for simultaneous kernels. We present Cain as a tool that at its core is applicable to any SIMD CPA, and we propose an implementation for targeting the Scamp5 architecture.

We formalise the objectives of such a compiler using logic programming and go on to develop a novel graph search algorithm suited to the particular challenges faced in code generation and the application of heuristics. We evaluate Cain for Scamp5, to test the relationships between properties of filters and program lengths. We go on to test Cain against well known convolutional filters and to find it outperforms an alternative compiler, AUKE.

ACKNOWLEDGEMENTS

I'd like to thank Paul Kelly and Riku Murai for their countless insights and helpful discussions and express my appreciation to Thomas Debrunner for his work on AUKE that was influential in producing Cain. Finally I'd like to thank Jesus, my church family, Hannah, and George for their continued support and putting up with me ranting about kernels and algorithms.

Contents

Chapter 1

Introduction

Real-time applications that use computer vision are currently bound to traditional camera sensors that export each pixel at each frame to a host where it is processed, often using GPUs or FPGAs for the advantages of parallelism that they provide. This method requires high-performance buses between the sensors and hosts, especially where high frame-rates are required. For example: a 2MP camera, with 10-bit pixel depth, running at 2200fps, requires a bus capable of 45.6 Gbit/s to transfer the stream of data continuously, such as a PCI-e x8 Gen3 interface[26]. For many applications, a solution to this requirement for an expensive and power intensive bus is to move some of the processing into the sensor chip itself.

Moving some of the processing on to the sensor can be achieved with Focal-Plane Sensor-Processors (FPSPs). These specialist sensor-processor combinations, often synonymous with Cellular Processor Arrays (CPAs), come in various forms for specific applications, normally related to high frame rate or low latency situations [27]. The principal behind them is that small processors are embedded directly into the sensor. For our purpose we look only at a general-purpose fine-grain architecture, but one can imagine the plethora of alternatives that could be designed with various use cases.

One of the most widely used methods for analysing the image data is convolutional kernels. Many operations that can be applied to images for use in feature detection, or as part of Neural Networks, are distilled into convolutional kernels. Traditionally these convolutional kernels are applied by the host processor. More recently came a revolution in using GPUs and then FPGAs accelerate each application of a kernel in parallel. While this significantly increased throughput, these methods are still bound by the data making its way to and through the host. 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. As for FPSPs, the technology is comparatively new and since, by design, they offer novel ways to interact with the data, there has been less work done to find code generation systems to make efficient use of their architectural features when applying convolutional kernels.

One such system that does exist, however, is AUKE [8]. Its Reverse-Split algorithm is able to take a single convolutional kernel and generate code for the SCAMP-5 FPSP chip to apply said kernel efficiently using the analogue computation features of the architecture. The primary objective of this work is to produce an improved alternative to AUKE, with the ability to produce code for applying multiple convolutional kernels at a time. The aim is that being able to apply kernels simultaneously will allow for optimisation of common sub-expressions.

In this thesis I will show how I produced these contributions and justify these claims:

- The primary contribution is Cain, a framework for compiling simultaneous convolutional kernels for cellular processor arrays.

- I use logic programming to formalise the constraints and scope of code generation of convolutional filters on multiply-free SIMD cellular processor arrays.
- I present an implementation for targeting Scamp5 using Cain that is able to fully harness the instruction set available to generate more efficient code.
- I have developed a novel graph search algorithm that successfully uses an imperfect local heuristic to find better instruction sequences than competing algorithms.
- I have evaluated Cain and discussed the relationships between the size of kernels, number of kernels, and sparsity of kernels on program length.

We conclude this thesis with a discussion of future works including developing the scope of Cain to non-linear functions and machine-learning aided heuristics.

Chapter 2

Background

In this chapter we look at the technology that Cain is built for and the technology it succeeds. By looking at Scamp[5] and AUKE[8], we see how the problem space is laid out and how existing solutions look to solve it, as well as introduce the technical knowledge required to follow the design process of Cain presented in later chapters.

2.1 SCAMP

The SCAMP-5 architecture is a general-purpose fine-grain SIMD FPSP [5]. It has a 256 by 256 pixel array, and along with each pixel is a small processing element (PE). Each of these processors contains 14 binary registers, 7 analogue registers, an arithmetic logic unit, the ability to send and receive signals from its 4 cardinal neighbours, and all execute the same instruction at one time. In particular 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`.

For example; the macro instruction `add(A, B, C)` 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 clobbered, 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. For example: `mov2x(A, B, north, east)` which is comprised of `bus(XS, B); bus(A,XE)`, which first means that $NEWS_{\text{east}} := B_{\text{north,east}}$ and then $A := XE = NEWS_{\text{east}} = B_{\text{north,east}}$.

While interesting uses of the `bus` instructions exist, allowing adding and subtracting from neighbouring PEs, individual macro instructions are still highly restricted in comparison 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 [6]. The lack of a multiplication function puts the problem of generating convolutional filter code for Scamp into the broad class of theory about multiply-free convolutional filters.

The chip has been shown to be capable of operating at 100,000 fps, largely because it is not limited by the speed of an output bus to transfer all the pixel data[4]. 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 asynchronous event driven outputs. This effectively allows each individual pixel to come to a judgement of if it saw something useful, and fire an event if it did, which then sends the ordinal coordinates of that pixel back to the host. In this way no direct image data is transferred, but instead one can imagine an event being fired at a pixel when a particular feature is detected by the sensor.

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.

2.2 AUKE

Automatic Kernel Code Generation for Analogue SIMD (AUKE) is an algorithm for generating code given a single convolutional kernel created by T. Debrunner[8]. It can be characterised by 4 main steps: kernel approximation; the reverse split algorithm; graph relaxation; and finally register allocation. The first step is an approximation to ensure that every element is represented by an integer number of binary fractional parts (with a configurable minimum size) - this is because division by 2 is used to generate fractions of the initial value, and it helps to bound the size of the problem. A fundamental concept in AUKE is to consider any convolutional kernel as a set of coordinates for its constituent parts in their proportions, called a goal.

We can see in Figure 2.2.1 that the kernel is first split by the lowest common denominator, 1. For each 1 in the kernel, K, a coordinate or 'atom' is put into the goal, G_K; and given the relative coordinate of that value from the kernel's origin, here assumed to be the centre. If one of the elements in K were to be 0.5, 0.5 would be the lowest common denominator and so there would be an atom in G_K for each 0.5 that makes up the values in K. This representation of a kernel underpins the algorithm that then considers how a goal can be achieved by moving these atoms around the PEs, adding or subtracting as it goes.

$$K = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \Longrightarrow G_K = \begin{Bmatrix} (-1,1), & (-1,0), & (-1,0), & (-1,-1), \\ -(1,1), & -(1,0), & -(1,0), & -(1,-1) \end{Bmatrix}$$

Figure 2.2.1: Decomposition of the Vertical Sobel Filter into atoms as performe in AUKE[8]

Instructions are represented by transformations of goals. For example, if we have a goal, we can apply a move transformation that is AUKE's representation of doing two `movx` instructions, as seen in 2.2.2. This transformation shows that the U is in fact directly transformable from the L.

$$\begin{aligned} L &= \{(1,-1), -(0,1)\} \\ &\qquad\qquad \texttt{movx}(R_B, R_L, west) \\ B &= \{(0,-1), -(-1,1)\} \\ &\qquad\qquad \texttt{movx}(R_U, R_B, north) \\ U &= \{(0,0), -(-1,2)\} \end{aligned}$$

Figure 2.2.2: Example of a "1 ← 1 ↑" transformation in AUKE representing multiple Scamp5 Instructions, where R_G is the register that G is mapped too after allocation.

The code is generated via a recursive search algorithm that splits a goal into 3 sub-goals, U, L, and R. The algorithm finds U and L such that U is a single transformation away from L such that if the search can find solutions for L and R (two smaller problems) it can trivially create U and therefore the desired goal. If two goals are equal they are merged such that they aren't calculated twice. This process is carried out until only a single goal, called the initial-goal, is left. This algorithm is able to entirely search the problem space, given a couple of assumptions. Most notably, the assumption that every sub-goal generated is a subset of the final goal. This assumption means the algorithm does not search every possible set of execution steps, as that would lead to an intractable algorithm. That assumption does however mean certain optimisations that make use of subtracting two goals to form a higher level goal will never be found.

The algorithm can be made efficient and useful by intelligently selecting the order with which U, L, and R are selected at every recursive step. By selecting pairs of U and L that are likely to lead to efficient code, the algorithm can quickly find a way to the initial-goal and from then on return up the recursive search early if a previous avenue of the depth first search had already reached the initial-goal at a lower cost.

The Graph Relaxation step aims to mitigate missing optimal solutions because of the assumption that sub goals are always subsets of the final goal by using a 'retiming' algorithm used in integrated circuit design. A graph represents the goal states and the edges dictate the transformations between them. This step allows optimisations to be made that break the

assumption while retaining the semantics of the graph by exchanging multiple expensive shift operations that are redundant in the slightly wider context of the surrounding nodes with less expensive shifts.

The final step is to perform register allocation on the graph to be able to generate usable code. A maximum bound of registers is already accounted for since spilling is not an option for the SCAMP-5 architecture, meaning the Reverse-Split algorithm stops early if too many sub-goals are guaranteed to need to be stored at a time. This step needs to decide the physical locations of goals, and for this task; variable liveness is considered for each node of the graph representation, and a graph colouring algorithm is used to find a solution.

These are the important understandings required to build and evaluate Cain, but there is a wider context of ideas and works about Cellular processor array design and convolutions that are relevant to understanding the choices made when designing and creating Cain. These ideas are focus of the next chapter.

Chapter 3

Related Works

This section explores various ideas and concepts that form the scope of study that Cain is a part of. These works demonstrate the breadth of ideas researched and considered in order to produce Cain, and the motivations that make efficient vision based convolutional filtering a useful tool.

3.1 Digital CNNs on SCAMP5

An alternative approach to using Scamp5's analogue capabilities researched to enable a similar intended behaviour, specifically for convolutional neural networks, is to do the processing in the digital registers of the SCAMP-5 chip[2]. The principle behind this method is to exchange sensor resolution for bit depth, because each PE by itself has far too few digital (1-bit) registers. Their approach split the sensor into 4 by 4 groups of PEs giving them a bit depth of 16 for their computation. They were able to implement ternary weight kernels and max pooling, and use the analogue registers to store their images that weren't currently used for operations. While the computation methods used are different, the communication process to send values to their required PE cluster while computing the desired result is the same as when using single PEs to calculate filters. This means that AUKE's fundamental algorithm would still be relevant for finding optimal ways to compute the desired kernel. And by extension a method of optimising multiple kernels would be similarly useful.

3.2 General Purpose Massively Parallel Computers

The concept of having a large array of small processing elements is not new. Developed in theory in the 1960s and first operational in 1975, ILLIAC IV is often considered to be the first massively parallel computer[16]. Before ILLIAC IV, machines were being designed and made that already took advantage of the SIMD paradigms that ILLIAC IV builds on. It was designed to have 4 quadrants, each made up of a Control Unit and an array 64 processing elements, which each also have their own memory unit. In reality only one quadrant was ever built[3].

Each PE in ILLIAC IV had 4 arithmetic registers and an instruction set able to operate on up to 64bit operands, and each PE's memory unit could store 2048 words of 64bit each. This makes each ILLIAC IV PE far more capable than a Scamp5 PE in terms of data handling and computation. They are able to do full 32bit and 64 bit floating point arithmetic. In the Scamp5 architecture PEs communicate via a shared register accessible by 4 neighbouring PEs in a 2 dimensional grid. In ILLIAC IV, PEs can be thought of as in a ring such that with one instruction a value in a register can be routed around the ring; each of the 64 values

being shifted N steps. This is not a constant time operation, instead jumps of -8, -1,1, and 8 are direct and other shifts are made of partial-shifts. This in effect gives ILLIAC IV a 1 dimensional array of PEs when we consider how kernel convolutions might be processed.

In 1973, "DAP - A Distributed Array Processor"[22] was published. It describes the use of an array of digital PEs each using very few registers (all single-bit registers) and an adder, with the appropriate control circuitry. This architecture is very similar, in respect to potential algorithms, to the SCAMP5 architecture. The main exception appears to be that in this design each PE has a memory unit with 4K-bits of storage, that can be accessed by neighbouring PEs too.

3.3 FINITE IMPULSE RESPONSE

Finite Impulse Response (FIR) filters are a staple of electrical engineering that share many fundamental concepts with convolutional filters. An FIR filter in a basic form is like a 1-dimensional convolution filter over a time series of values. This similarity suggests that the long standing work in optimising FIR filters may be transferable to convolutional filters. FIR filters are often implemented in the simplest electronic logic possible, therefore many are multiply-free.

One optimisation when looking at multiply-free FIR filters is to change the representation of coefficients to the canonic signed digit (CSD) representation [24]. This method aims to reduce the number of non zero bits in a binary form of a number by allowing the use of -1. This representation is very effective in multiply-free systems because subtracting numbers as opposed to only adding them carries no added complexity so reducing the number of non-zero terms in an integer coefficient reduces the total number of operations that are needed to perform a multiply. By reducing the total number of operations, filters can be created with few components and less logical depth. Table 3.3.1 shows how the number 189 uses 6 non-zero digits in its binary form whereas only 4 in its CSD representation.

Table 3.3.1: 189 represented in Binary, CSD and MSD

Binary:	0	0	1	0	1	1	1	1	0	1
CSD :	0	1	0	-1	0	0	0	-1	0	1
MSD:	0	1	0	-1	0	0	0	-1	0	1
	0	0	1	1	0	0	0	-1	0	1
	0	1	0	-1	0	0	0	0	1	1
	0	0	1	1	0	0	0	0	1	1

Other optimisations presented in [24] include common sub-expression elimination. Their conclusion is that by finding occurrences of a few well known common sub-expressions within the CSD representation of an FIR filter they are able to reduce the logical operators required in their implementations more than other contemporary optimisation methods.

Minimal signed digit (MSD) representations build on CSD[17] by relaxing the property of

CSD that consecutive digits are never both non-zero. In doing this MSD does not uniquely represent every number but instead provides a set of options to represent each number. Table 3.3.1 shows the 4 representations of 189 in MSD. The average number of options grows exponentially with the number of digits in the CSD form. Using MSD means that where multiple coefficients are needed, the set of possible representations for each coefficient can be searched to find the combination of MSD representations with the most common sub-expressions that will yield more optimised circuits with fewer adding units.

3.4 MNIST ON SCAMP5

Mnist is a dataset of hand drawn digits[18] and is a well known benchmark for image recognition machine learning algorithms. Mnist has been used as a demonstration of CNNs on Scamp5 by Matthew Wong's AnalogNet [25], showing that with careful network design, FPSPs can be used to run the first layer of a CNN, with the subsequent layers being performed on the host processor. The method involved training the network to account for noise introduced by Scamp5's analogue computation, using AUKE to generate Scamp5 code, which was then used to generate the noise model in training the subsequent layers.

This work was pushed forward by the development of quantisation, allowing for the digital registers in Scamp5 Processing elements to be used to expand storage and allow for 2 layers of the convolution to happen on the FPSP chip, though it was found that noise became too large of a problem in the analogue arithmetic process and the performance of the CNNs was not improved. This work did, however, bring about better register management and alternative ways transfer the data off the PEs and to the host processor efficiently, and so AnalogNet2[14] was created to make these optimisations.

3.5 SUPER-OPTIMISATION

Super-optimisation, coined in 1987, is the process of taking the specification for some function and exhaustively searching through possible sequences of instructions to find novel and optimal programs than are not traditionally found by compilers [20]. This primitive part of the process is effectively what can be done via logic programming as we see in section 4. To verify a possible program a super-optimiser expresses its input as Boolean-logic operations and does the same for the candidate instruction sequence. This ensures the program is correct, but it takes an intractable amount of compute time and memory to test.

What makes super-optimisation tractable, however, is an extra probabilistic test used to filter the search: the generated candidates are run on sample inputs and those that produce incorrect results are quickly culled. As it turns out, testing programs is far cheaper than proving them, and this made super-optimisation feasible for longer instruction sequences. In 1987, they required that the programs can be run on the target hardware as emulation of another systems would slow down the process significantly. Super-optimiser also uses pruning to remove instruction sequences that contain sub-sequences that are known to have shorter

implementations, or be useless, such as moving a value between two registers, back and forth.

Further work on super-optimisation is implemented in GSO[13], the GNU Superoptimiser, taking the exhaustive search concept and providing a mechanism to make it portable, allowing the definition of non-native instruction sets. Other improvements include rejecting the use of instructions that use previously un-set registers, or the carry-bit, and insisting on a strict order of register use to minimise redundant code sequences.

Super-Optimisation has been used to generate optimal code for small common functions that can then be applied in larger programs - however the cost of optimising programs whole programs is still infeasible.

3.6 LINNEA

Linnea[1] is a code generator for complex linear algebra computations, and in many ways shares the same problems as Cain. Linnea takes an expression such as

$$b := (X^T X)^{-1} X^T y$$

and treats it as a node in a graph. Then it generates edges from that node using pattern matching to find parts that can be rewritten or computed, to make new nodes and so on until every part of the expression has a computation planned. Much like in Cain, at each node there are numerous ways to proceed and so the graph becomes quite extensive, and is generated as the search finds new nodes. Linnea also has a concept of partially visited nodes; to implement this they use a '`next_successor`(n)' function which returns the next child node of n each time it is called on n, until no more child nodes exist. This allows Linnea to implement a graph traversal algorithm we have called Heir-Ordered-Search, which is discussed in section 5.3.5, and implemented in Cain as well. As Linnea generates and searches its graph of algebraic rewrites it can create redundant nodes where paths converge to the same semantic result, though with different costs to compute. To optimise the search, nodes are compared via a normal form that allows nodes to be check for equality. This process is not guaranteed to be solvable, but when it is, it allows for the size of the graph to reduced, nodes to be merged, and for the smaller of the costs used in the future.

3.7 SEARCH ALGORITHMS

In Section 5.3 we see that finding efficient code to generate presents itself as a graph search problem. There are many graph search algorithms for efficiently finding a minimum path between two nodes under different conditions, the most famous perhaps being Dijkstra's algorithm[9]. For many applications, where globally comparable heuristics are available, Hart, Nilsson & Raphael's A* algorithm[15] is the best solution, providing the optimal path in the fewest number of nodes that need visiting. Algorithms such as Greedy-best-first-search

are able to more quickly find the optimal path in many cases, but have a higher worst case bound.

These search algorithms consider the nodes in a graph as part of one of three sets, the visited-set, the frontier-set, and the unseen-set. If a path from the start node to a node n is known, then n is in the visited-set. If a node n has an edge connecting it to a node m in the visited-set, but n isn't in the visited-set itself, n is in the frontier-set. Finally all other nodes are in the unseen-set. In general, search algorithms differ by how they choose which frontier node to visit.

Algorithms such as Beam-search do not guarantee an optimal path, but by putting a cap on the number of nodes in the frontier set, keeping the ones based on a heuristic, they are able to reduce the memory overhead of searching the graph, which could otherwise make the graph traversal intractable. Like the other heuristics seen so far, this algorithm requires that we can sort the frontier set from most promising to least promising: a property that we show is not so trivial in section 6.4.

In this chapter we have looked at various technologies as paradigms related to computing convolutional filters, and for performing optimal code generation in different contexts. These make up the wider search for ways to produce efficient code for cellular processor arrays in Cain. To better understand the problem at hand we next look to define it more formally.

Chapter 4

Problem Definition & Answer Set Programming

In this chapter we look at how the problem of code generation can be more formally reasoned, using Answer Set Programming. Though this formalisation we will see the basic requirements and constraints that bound the creation of a code generator like AUKE, or Cain, and how we can use Answer Set Programming to both define the problem and produce a naive solution.

4.1 Logic Programming

Answer Set Programming is an area of declarative logic programming used to solve various problems often relating to abductive reasoning or that are NP-hard[19]. ASP solvers take a set of logical constructs and constraints, called a knowledge base, that describe a problem as well as a goal; then they search for a set of possible truths, to find a set of truths that are sufficient to prove the goal for the described problem. Effectively they search for an explanation that is coherent with the knowledge described, that implicates the goal. Logically this does not mean the resultant solution is definitely true, but solvers can go on to find all possible solutions, so if the problem is fully defined, the actually true solution will be one of the solutions found.

Answer Set Programming comes from declarative logic programming, whereby logic prepositions are declared and a solver derives their logical consequence. ASP differs from other logical programming semantics in its handling of negation[12]. In fact there are two types of negation; "negation as failure" and "strong negation". The two can be informally described as: being satisfied if something cannot be proved; and only being satisfied if it has been disproved, respectively. Here, we only need to consider how negation as failure works, which is denoted by "not" rather than $\neg$ as strong negation would be.

In ASP, a program Π is a set of statements that take the form:

$$A \leftarrow B_1, ..., B_m, notC_1, ..., notC_n \tag{4.1.1}$$

where B_1 to B_m, and C_1 to C_n are all logical atoms and together make the body of the statement that implies the head atom, A. The statement defines that if all of B_1 to B_m are true and C_1 to C_n are not true then A is true. This means that if the body is empty then A is always true.

For a program Π, M is a set of atoms in Π. A program that contains no negated atoms in any of the bodies of its statements has one and only one stable model Π^s, simply achieved

by repeatedly testing each statement against an initially empty set M and adding to M the head of the statement, A, if the body of the statement holds under M. Once no statement in Π can be applied, $M = \Pi^s$, the stable model of Π.

In ASP, for every possible M, we produce Π_M by:

- Removing from Π all statements that have a $notC$ in their body where $C \in M$ (as their statement's head cannot hold if all of M holds).
- Then removing all other negative atoms in the statements of Π.

Since Π_M now has no negated atoms it has one stable model. If M is equal to Π^s_M, the stable model of Π_M, then M is also called a stable model of Π, also known as an Answer Set.

Programs are more commonly written in first order logic, such that they can use functions and numbers to make the code more readable and in many cases tractable to write. This however means every statement needs to be grounded before it can be properly processed, a step that is often done before solving, or done dynamically in the solver[11]. Using first order logic allows statements such as $p(X) \leftarrow q(X)$ to be written, from which it is inferred that $p()$ can take at least any argument that $q()$ can. A grounding engine will take this statement along with $q(1)$ and produce the ground atoms $\{q(1), p(1)\}$ which form the program $p(1) \leftarrow q(1).\ q(1)$. There are also several short hand ways of expressing things such as some minimum and maximum number of atoms from a set being true at the same time.

4.2 A More Formal Definition of The Problem

When we apply this paradigm to the problem of finding a set of instructions that produce the correct convolutional filter, we first must define in logical terms what the state of the processor looks like over time, and what effect an instruction at a time-step has on the next state. We also define things such as every time-step must have one and only one instruction; and what the initial processor state is. Finally we logically constrain the instructions such that by the last instruction the state of the processor is such that a registers contains the result of performing the intended convolution.

4.2.1 Given Truths

The code begins a list of truths that assert the initial state of the processor and search paramters, in codeGen_in.lp:

```
1 reg(a).
2 reg(b).
3 %reg(c).
4 %reg(d).
5 %reg(e).
```

These lines define `a` and `b` as registers (`reg/1`), while lines 3 to 5 are commented out for this example, and so those registers do not exist as far as the program is concerned.

```
6 init_reg(a).
7 init_count(1).
```

Line 6 defines that `a` is not only a register but is also the initial register that the input value will start in. Line 7 defines how many *units*[*] start in register `a`. Lines 8 to 16 are a comment block for a different example kernel.

```
17 % 2x2 box
18 goal_in(0,0,1,a).
19 goal_in(0,1,1,a).
20 goal_in(1,0,1,a).
21 goal_in(1,1,1,a).
```

These lines define the kernels we wish to produce; `goal_in(AX, AY, C, R)` means that register `R` must contain `C` *units* from the relative PE coordinates (`AX`, `AY`). This defines the filter we want to generate instructions for as the 2by2 Box filter: $\begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$.

```
23 moves(7).
```

Finally, the last line to configure our search is to put a maximum bound on the size of the problem by asserting that there are only 7 time-steps in which to compute the kernel.

4.2.2 PROBLEM ENCODING

The rules that govern how the instructions are used and affect the state, and what constitutes a valid solution, are contained in codeGen_enc.lp. The separation of these files is only so that it is clear what predicates are configurable for the search and which are functional for the search to work.

```
1 dir(left).
2 dir(up).
3 dir(down).
4 dir(right).
```

Here we define the four cardinal directions available to the Move instruction, This could be extended if the target hardware was able to make a Move instruction with another direction.

```
6 goal_reg(R) :- goal_in(_,_,_,R).
7 init_tile(X,Y) :- goal_in(X,Y,_,_).
8 in(AX,AY,C,R,AX,AY,0) :- init_tile(AX,AY),
9                          init_reg(R),
10                         init_count(C).
```

[*] '*units*' can be read as 'atoms' from the approximation of the convolutional kernel, but the theory is not restricted to the atom approximation, so 'unit' refers to some part of a coefficant in the filter

Line 6 asserts that a register R is a final register (goal_reg/1) if any goal_in/4 predicates with R are true, which means goal_reg(a) will hold in our example. Similarly, line 7 ensures that init_tile(X,Y) holds for all the PE coordinates of the non-zero coefficients in the target filter. Line 8 defines using the in/7 predicate that every PE, for which a non-zero coefficient from the target filter corresponds, contains C *units* in register R that originate from its own coordinates, at the initial time-step 0. The predicate in(AX,AY,C,R,X,Y,T) is true if C *units* originating from the PE at (AX, AY) are in register R of the PE at coordinates (X, Y), at time-step T.

```
12 1 {
13  move(R,RR,D,T) : reg(R), reg(RR), dir(D);
14  add(R,RA,RB,T) : reg(R), reg(RA), reg(RB);
15  sub(R,RA,RB,T) : reg(R), reg(RA), reg(RB);
16  null(T)
17  } 1 :- moves(M), T = 1..M.
```

These lines define what instructions are available to the processor and that for every time-step from 1 to M (7, as set in line 7 of codeGen_in.lp) inclusive, there must be one and only one instruction; move/4, add/4, sub/4, or null/1. Each of these instructions also have their own requirements, such as their operands being registers. This is where more requirements such as Scamp5's add instruction not allowing both inputs to be the same register could be imposed. This means that for a time-step T any of the instructions are permissible with any valid register operands, or direction for Move, regardless of if the instruction does anything at all, or makes progress towards achieving the goal. The set of realised instructions, at their respective time-steps, are what make up $M \in \Pi$ and so can limit the scope of what the solver searches for and checks, allowing the the derived predicates, like in/7, to be produced, added to M, and then checked again for stability.

```
19 in(AX,AY,C,R,X,Y,T) :- move(R,RR,left ,T),
20                        in(AX,AY,C,RR,X+1,Y,T-1).
21 in(AX,AY,C,R,X,Y,T) :- move(R,RR,up   ,T),
22                        in(AX,AY,C,RR,X,Y-1,T-1).
23 in(AX,AY,C,R,X,Y,T) :- move(R,RR,right,T),
24                        in(AX,AY,C,RR,X-1,Y,T-1).
25 in(AX,AY,C,R,X,Y,T) :- move(R,RR,down ,T),
26                        in(AX,AY,C,RR,X,Y+1,T-1).
27
28 in(AX,AY,C,R,X,Y,T) :- add(R,RA,RB,T),
29                        in(AX,AY,CA,RA,X,Y,T-1),
30                        in(AX,AY,CB,RB,X,Y,T-1),
31                        C=CA+CB.
32 in(AX,AY,C,R,X,Y,T) :- add(R,RA,RB,T),
33                        in(AX,AY,C,RA,X,Y,T-1),
34                        not in(AX,AY,_,RB,X,Y,T-1).
35 in(AX,AY,C,R,X,Y,T) :- add(R,RA,RB,T),
36                        in(AX,AY,C,RB,X,Y,T-1),
```

```
37                          not in(AX,AY,_,RA,X,Y,T-1).
38
39 in(AX,AY,C,R,X,Y,T) :- sub(R,RA,RB,T),
40                          in(AX,AY,CA,RA,X,Y,T-1),
41                          in(AX,AY,CB,RB,X,Y,T-1),
42                          C=CA-CB.
43 in(AX,AY,C,R,X,Y,T) :- sub(R,RA,RB,T),
44                          in(AX,AY,C,RA,X,Y,T-1),
45                          not in(AX,AY,_,RB,X,Y,T-1).
46 in(AX,AY,C,R,X,Y,T) :- sub(R,RA,RB,T),
47                          in(AX,AY,-C,RB,X,Y,T-1),
48                          not in(AX,AY,_,RA,X,Y,T-1).
```

These lines define how the state, through time as recorded by the `in/7` predicate, is affected by instructions at every time-step `T`. The instruction `move/4` has four cases, one for each direction. `add/4` and `sub/4` each have three cases, one each to account for both operands existing, and two each to handle the cases where one of the operands has not been defined so far. This is needed since we only define the initial state of a PE as some number of origin *units* in one register, for each PE that appears in the target filter. This means that instead of a register `R` containing 0 *units* from (`AX`, `AY`) in the PE at (`X`, `Y`), the predicate `in(AX,AY,0,R,X,Y,T)` simply doesn't hold at all, and it doesn't need to because the absence of it can be used to infer that `C` is 0. There are no rules for `null/1` since it makes no changes to the state.

```
51 reset(R,T) :- move(R,_,_,T).
52 reset(R,T) :- add(R,_,_,T).
53 reset(R,T) :- sub(R,_,_,T).
54 in(AX,AY,C,R,X,Y,T+1) :- in(AX,AY,C,R,X,Y,T),
55                            not reset(R,T+1),
56                            not moves(T).
```

Since the state is recorded with a predicate as a function on time-steps `T`, we must define the behaviour of all registers that are not overwritten by the instruction at `T`. The `reset/2` predicates holds if a register at a time-step is overwritten. Then the rule starting on line 54 ensures that the state is the same as the previous time-step everywhere that isn't reset. This also asserts that the next state only remains the same inside the maximum time-steps set; this is to stop registers holding values beyond the end of the program since that would slow down the search significantly and is useless for finding solutions.

```
58 :- in(AX,AY,CA,R,X,Y,T),
59    in(AX,AY,CB,R,X,Y,T),
60    CA!=CB.
```

These lines simply assert a logical constraint on the processor that it is impossible for there to be two different counts of *units* from the same origin in the same register in a PE at a single time-step. This is done by having an empty head of the statement which means that if the body holds, then the model M is not stable.

```
62 :- goal_in(AX,AY,C,R), moves(M),
63    not in(AX,AY,C,R,0,0,M).
64 :- in(AX,AY,C,R,0,0,M), goal_reg(R), moves(M), C!=0,
65    not goal_in(AX,AY,C,R).
```

The two rules here are what constrain the sets of possible instructions to ones that actually produce the desired result. Firstly the problem is constrained such that for every `goal_in/4` predicate, the model M is not stable if it has not been shown that that goal has been fulfilled by the last time-step, `M`. Then we constrain the state such that if a goal register (`goal_reg/1`) contains a particular non-zero count of a *unit* by time-step `M` that is not defined as a goal, then the model is not stable. These rules assert that output registers must contain the *units* specified by the `goal_in/4` predicate and nothing else.

```
67 #maximize{1,T:null(T)}.
68
69 #show move/4.
70 #show add/4.
71 #show sub/4.
72 #show null/1.
```

These last lines are for the solver, and so do not make up the logic of the program, but rather what solutions to present. Line 67 tells the solver to maximise the number of `null/1` atoms in the answer set. More null operations for a given maximum number of steps defined by `moves/1` mean that we minimise the number of other instructions. Lines 69 to 72 simply tell the solver to only output the atoms in the answer set of these particular types - this is because the answer set must include every `in/7` atom as part of the stable model, and every other atom that is produced in satisfying the program, but we only care about the instruction atoms that tell us how to produce the target filter.

4.2.3 2x2 Box filter Example

To test the ASP program, we now look at a 2x2 Box filter to see what solutions are found. Listing 4.1 shows the direct output of running the solver. Iclingo finds 3 solutions before determining that it has found an optimum; the first taking 6 non-null instructions, then 5, and finally 4.

```
$ iclingo 0 codeGen_in.lp codeGen_enc.lp
> clingo version 5.2.2
> Reading from codeGen_in.lp ...
> Solving...
> Answer: 1
> move(b,a,up,1) move(a,b,left,3) move(a,b,down,4) move(b,a,
   left,5) add(b,a,b,2) add(a,b,a,6) null(7)
> Optimization: -1
> Answer: 2
```

```
> move(a,a,down,1) move(b,a,left,2) move(a,b,up,5) add(b,a,b
   ,3) add(a,b,a,6) null(4) null(7)
> Optimization: -2
> Answer: 3
> move(b,a,down,1) move(b,a,left,3) add(a,b,a,2) add(a,a,b,4)
    null(5) null(6) null(7)
> Optimization: -3
> OPTIMUM FOUND
>
> Models       : 3
>   Optimum    : yes
> Optimization : -3
> Calls        : 1
> Time         : 189.203s (Solving: 176.13s 1st Model: 3.92s
   Unsat: 116.16s)
> CPU Time     : 189.195s
```

Listing 4.1: Terminal output of solving for a 2x2 Box filter with ASP using iclingo

The optimal solution can be interpreted as follows in 4.2.1. The last number in each instruction tells us the order of instructions, and the move instruction `move(b, a, down,1)` means that the value in register `a` is moved downwards (or south) into register `b`, meaning it can be represented by the assignment $b := a_{up}$.

		$b := a_{\text{up}}$	$a := a + b$	$b := a_{\text{right}}$	$a := a + b$
Register a:	$\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix}$	$\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix}$	$\begin{bmatrix}0&1&0\\0&1&0\\0&0&0\end{bmatrix}$	$\begin{bmatrix}0&1&0\\0&1&0\\0&0&0\end{bmatrix}$	$\begin{bmatrix}0&1&1\\0&1&1\\0&0&0\end{bmatrix}$
Register b:	$\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}$	$\begin{bmatrix}0&1&0\\0&0&0\\0&0&0\end{bmatrix}$	$\begin{bmatrix}0&1&0\\0&0&0\\0&0&0\end{bmatrix}$	$\begin{bmatrix}0&0&1\\0&0&1\\0&0&0\end{bmatrix}$	$\begin{bmatrix}0&0&1\\0&0&1\\0&0&0\end{bmatrix}$

(4.2.1)

This example shows the validity of the program, and the method of using ASP to find provably optimal code. It also demonstrates the failings of ASP as a general solution to the problem, given being provably optimal is not as important as being efficient in finding good solutions. The example was run on an Intel Core i7-7700HQ CPU @ 2.80GHz based system with 16GB of RAM, running Ubuntu 18 and iclingo 5.2.2. It took 189 seconds to complete the search to exhaustively show that there are no better solutions than one that takes 4 instructions, with the solution with 4 instructions being found after 75 seconds.

We can see that this program describes precisely the requirements of any tool to produce correct convolutional filter code for a SIMD cellular processor array. It is trivial to see how more instructions, with varying behaviours, can be added without affecting the principles of

correctness. The paradigm shown here can be extended to allow multiplication and division, though real or rational counts of *units* is not trivially available in common ASP solvers.

This program, however, is not a tractable solution for real use in code generation because it is very inefficient. The solver that tries to produce the answer set has no heuristic knowledge of how to proceed or pick instructions and cannot find valid solutions efficiently. Also grounding the predicates into all possible atoms presents huge memory overhead problems which - while can be improved with incremental grounding[11] - still do not allow the solver to complete within a tractable time for anything but the most trivial example shown. One important feature of this design in using ASP is that the solver can prove exhaustively that the solution it finds is optimal since it make no heuristic assumptions about the solution and will search all possible valid instructions no matter how unhelpful.

In this chapter we have looked at how Answer Set Programming can be used to more formally define the problem we aim to solve with Cain. The formalisation is based on modelling a convolutional filter as coefficients of inputs at relative coordinates which is a direct view of what convolutional kernels are, instead of the atom approximation used by AUKE[8] and Cain. This model clearly relates the two views via the representation of state in terms of a 'count' of *units* originally from a PE, now in a register in a PE. Though Answer Set Programming itself is not a tractable solution to the problem, it provides the framework with which to view code generation and the principals behind SIMD cellular processor arrays.

Chapter 5

Search Mechanism

Cain performs a reverse search that takes a kernel in the form of atoms[8] and applies the inverse of instructions to achieve an initial goal. In this chapter we look at how this works, and the intuition behind doing the search in reverse. We also see why the differences between Cain and AUKE allow Cain to handle more general instructions as a default case.

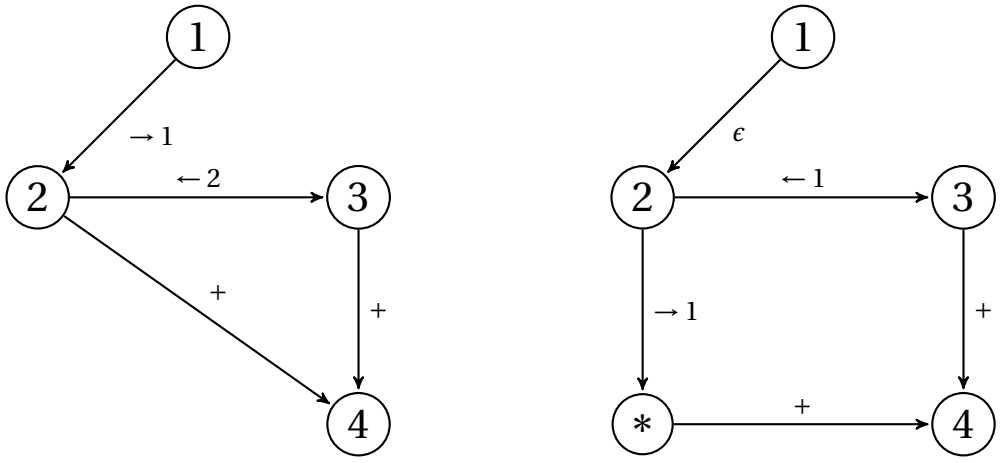
5.1 Atoms, Goals, and Goal-Bags

In Cain, filters comprise one or more kernels, each represented as a goal in a goal-bag that represents the goals that need to be processed. Atoms are a single unit of a coefficient of a kernel, just like in AUKE, except that atoms don't need to have a unique identification, and in the Cain implementation are 3-dimensional[†]. Goals are bags (also called multisets) of atoms, much like how in AUKE goals are sets of atoms that are unique. Finally, goal-bags are bags of goals. A single kernel can be represented by a goal, but to achieve multiple kernels, and to split a goal into sub-components, we use a goal-bag to group together all the goals that are currently relevant. We will use braces for goal-bags for readability, but they are multisets or 'bags', not sets. The initial goal is simply a binary number of positive atoms at the origin which represents the input to the filter.

$$\begin{array}{lrcl} \text{Atom:} & a,b,c & := & (x,y,z,\pm) \\ \text{goal:} & G,U,L,R & := & [a,a,b,c,...] \\ \text{goal-bag:} & C,B,FG & := & \{G,U,U,L,...\} \\ \text{initial goals:} & IC & := & \{[(0,0,0,+)_0,...,(0,0,0,+)_n] \mid \text{for } n \in [0,2^k)\} \end{array} \tag{5.1.1}$$

Generally when discussing graphs we use paths to mean the current plan of instructions that traverse through the graph, and when paths are defined by their nodes, this is a simplification as the paths actually contain the instructions required and what all the operands are etc.

[†]The third dimension is used to allow for multi-channel kernels in Scamp5 since the architecture is of planer PEs, not volumetric. It is also trivial to allow for N-dimensional atoms at the expense of computational efficiency at compile time.



(a) Model pre-relaxation (b) Model after relaxation

Figure 5.2.1: AUKE transformation graph relaxation stage. With $T = \leftarrow 1$, applied to node 2.

5.2 SEARCHING ALGORITHM

5.2.1 AUKE

In AUKE at each step of the search algorithm we start with the current goals that need to be solved, essentially a goal-bag, C. If $|C| = 1$ then a direct solution is looked for that transforms the initial goal into $G \in C$. Otherwise, a pair generation function generates a list of (U, L) such that U is a single transformation from L, and either: $G_0 = U + L + R_0$, $G_0 \in C$; or $G_0 = U + R_0$, $G_1 = L + R_1$, with $G_0, G_1 \in C$. The algorithm can then produce a new goal-bag B such that $\forall g \in C.g \neq G_n \implies g \in B$ and $L, R_n \in B$. Then the transformations that would turn B into C are appended to the plan.

If we take an example: $C = \{[(-1,0,0,+)],[1,0,0,+]\}$, then U and L must either be $\emptyset$ or $[(-1,0,0,+)]$ and $[(1,0,0,+)]$. If they are $\emptyset$ then $R = G, G \in C$ and so $B = C$ and no progress is made. The L that would be added to B can be ignored because an empty goal is implicitly solved by doing nothing. In the alternative case $G_0 = [(-1,0,0,+)] + R_0$ and $G_1 = [(1,0,0,+)] + R_1$ with $R_0 = R_1 = \emptyset$ and so $B = [(-1,0,0,+)]$ or $B = [(1,0,0,+)]$, depending on which way round U and L are, which makes no difference here as the problem is symmetric. It is then trivial to achieve an initial goal of $[(0,0,0,+)]$ by a simple translation. The problem here is that this way of searching first decides that we must move one of the two atoms over by two PEs, and then back one to the middle, taking a total of 3 instructions in the instruction set AUKE targets.

However, we can plainly see that a simpler solution is to make two move instructions, one for each goal from the initial goal. AUKE's way to deal with this is by graph relaxation; it generates the dependency graph of the computation and then re-weights it by the actual cost of each transformation. AUKE is then able to change existing nodes in the graph by applying an invertible operation T to all incoming edges, and then T^{-1} to all outgoing edges, adding nodes if required as we see in Figure 5.2.1. The aim is to have T and T^{-1} cancel out

operations that already exist in the edges to reduce the overall cost. The issue that arises with these steps is that it can make generalising instructions more complex and potentially limit the instructions available to invertible transformations.

5.2.2 REVERSE SEARCH

In Cain the search also starts with the final goal (though there can be multiple) but instead of insisting that $L \subset G$ where $G \in C$, we enforce that $U \in C$. This effectively just means that at every step of the search algorithm we take one goal from the current goal-bag and find a transformation that creates it from a list of goals, 'lowers' $Ls = [L_0, ..., L_n]$. Then B, the new goal-bag to solve, is $C \cup Ls$. Here we must ensure that the union operator on bags respects that duplicates in C or LS can be paired off as shown in Algorithm 1. This algorithm is the basis on which the the search is based, but it relies on two parts, firstly a function that takes the goal-bag of current goals and produces some iterator or list of pairs (U, Ls) such that a known transformation exists that takes Ls and produces U (Section 6.4); and secondly a driver function that determines how the graph of all possible goal-bags will be traversed (5.3).

```
Data: current goal-bag to solve, C, and goal-pair iterator pairs
Result: new goal-bags to solve, B
if pairs is null then
    pairs = generatePairs(C);
end
(Upper, Lowers), newPairs = pairs;
B = C.copy();
B.remove(Upper);
N = [];
forall Lower in Lowers do
    B.remove(Lower);
    N.add(Lower);
end
forall Lower in N do
    B.add(Lower);
end
yield B, newPairs;
```

Algorithm 1: Core premise of the reverse search algorithm used in Cain

We use idea of separating the graph traversal and pair generation because the order in which instructions are processed has a vital impact on performance. For any goal, G, consisting of n uniquely positioned atoms, there are at least $2^{n-1} - 1$ ways to split G into two non-empty goals that could be added back together to make G. More precisely, if we define $count_a^G$ as the number of a atoms in G, then the total number of ways to split that goal into

two non-empty goals becomes:

$$|splits(G)| = \frac{1}{2}\left(\left(\prod_{a\in \text{supp}\, G} 1 + count_a^G\right) + \left(\prod_{a\in \text{supp}\, G} (1 + count_a^G) \mod 2\right)\right) - 1 \tag{5.2.1}$$

$$|splits(G)| = \left\lceil \frac{1}{2}\left(\prod_{a\in \text{supp}\, G} 1 + count_a^G\right) - 1 \right\rceil \tag{5.2.2}$$

The intuition behind this function is that for each atom $a \in \text{supp}\, G$ there are $1 + count_a^G$ ways to divide the $count_a^G$ atoms between two goals*. This decision is made for every unique atom to produce the product, but this product includes all possible sub-goals $G' \subset G$ whereas we are only interested in pairs of sub-goals that can be added together, meaning half of that product represents duplicate answers. The exception to halving is when G can be perfectly divided into equal sub-goals; this case happens when $count_a^G$ is even for all $a \in \text{supp}\, G$ and so is accounted for before the division by two in Equation 5.2.1 by the second product. Finally we subtract the case where one of the sub-goals is an empty goal since these do not define a meaningful addition. In Equation 5.2.1 we use the the ceiling function to shortcut the symmetric intuition. This result demonstrates the number of different `add` instructions possible to chose from for each goal in a current goal-bag to solve, and shows that that the number of options grows quickly with the size of kernel, and so does the number of possible goal-bags to solve in the solution space. Further to this, using the subtract instruction potentially allows for an infinite solution space as a goal can be achieved in an infinite number of ways. At this point we are forced to make assumptions about the *generatePairs* function and concede that it produces a finite number of pairs, that we must assume contain the helpful pairs to consider.

5.3 GRAPH TRAVERSAL

Algorithm 1 requires a driver function to determine how to search the total solution space. Before this though, we need to define the search space so we can compare the available algorithms that might help. To define the search space we say that every goal-bag is a node in a graph, where every edge is an reversed instruction that transforms the goal-bag into a new node - as such these edges are directed. Figure 5.3.1 shows a simplified trivial example of said graph. The important differences here are largely to do with the scale of the graph. Here there are only up to a few edges emanating from each node, where as we have shown that the are normally very many possible ways to achieve a goal and so there should be many instructions for each node. This graph can be considered infinite since for example, we could continually use the `movx(A,A,north)` instruction forever. Even if we did make assumptions to constrain the graph similar to AUKE, the scale would still be intractable. Dead-ends can be found when an instruction would be invalid, for example if the goal-bag grew such that it no longer is containable inside the PE's registers. Numerous cycles also exist in the real graph though we show only one here; $9 \rightarrow 18 \rightarrow 10 \rightarrow 9$. While cycles are an important consideration, they can be accounted for relatively simply by tracking nodes that have already been

*The 'supp' or Support of a multiset or bag is the set of the unique elements in the multiset

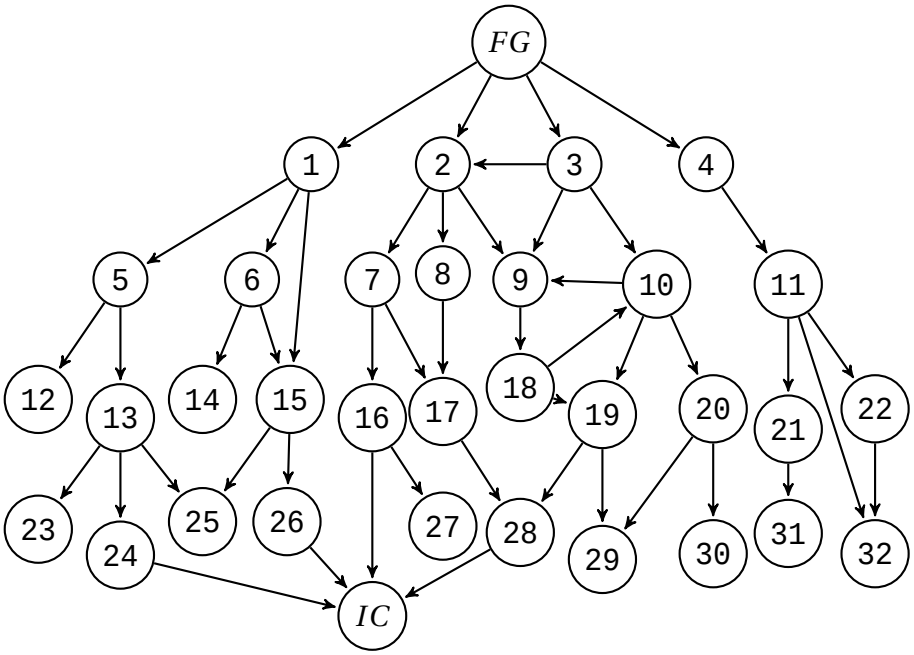


Figure 5.3.1: Graph representation of the search space.

seen, and combined with heuristics that favour reaching IC searching though cycles should be minimal. In theory there is a path from every node to every other, and many instructions are invertable so cycles exist all over the graph.

5.3.1 BREADTH-FIRST-SEARCH

When we look at appropriate graph traversal algorithms we first consider that the problem we have is to find the shortest path from FG to IC. In Cain we could consider each edge to have the same cost, and so using an optimal algorithm like Dijkstra can be achieved simply by performing breadth-first-search. While theoretically this solution is able to guarantee we find IC in the shortest number of instructions, it incurs an enormous memory overhead. Before a node is first visited we have no information about its children, and generating that information is done effectively via the *generatePairs* function. This means that each node we visit has a significant computational cost, and then the children goal-bags need to be stored too, along with the information about what instruction is being used. Combined with the scale of the graph, that each node usually has in the hundreds of edges emanating from it, breadth-first-search is not an effective graph traversal strategy. Cain can still be configured to use it, but for instruction-sets and architectures like Scamp5 it is unhelpful. If we look at Figure 5.3.1 we can see how BFS will perform, and that we visit 30 nodes in order to reach IC in a minimal number of instructions.

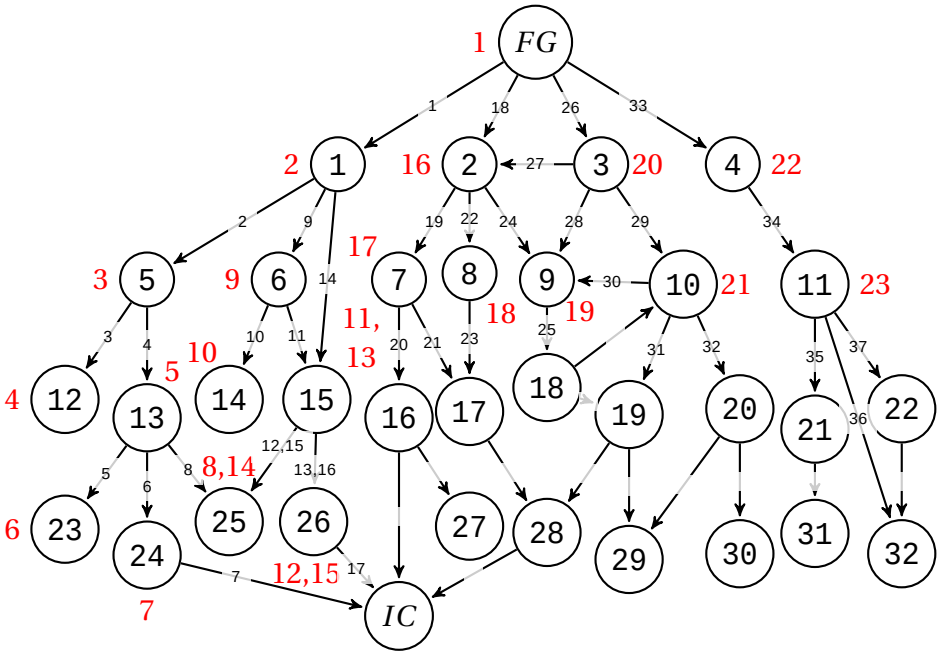


Figure 5.3.2: Graph representation of the search space using DFS. Edges are numbered in the order they are considered, and nodes are numbered, in red, in the order that their children need to be created.

5.3.2 DEPTH-FIRST-SEARCH

The obvious alternative to breadth-first-search is depth-first-search; this is what AUKE uses and it has many merits, though for it to be at all effective we must introduce some idea of priority, or ranking of potential instructions. If DFS is used without heuristics it is doomed to perform worse than BFS since it will most likely choose an unhelpful instruction. With an effective heuristic however, DFS is capable of searching the area of the graph that is most likely to quickly find IC. The issue then becomes whether a reliable heuristic can be found (Section 6.4). In the example graph we will assume that the *generatePairs* function returns pairs from most likely to least likely and they are drawn from left to right. We know that the minimal path we are looking for has 4 instructions, so we can see that at the initial node, FG, the heuristic is accurate - in fact both nodes 1 and 2 lie on paths from FG to IC with lengths 4, and nodes 3 and 4 don't. The heuristics perform less well at node 1, but given a node might have more than 100 potential instructions, being 3rd isn't all that awful. This example is also contrived to make a point about how ineffective heuristics can be mitigated.

While DFS is able to find a solution, it is not guaranteed to be optimal. We see in the example graph, Figure 5.3.2, that the first solution found will be $FG \rightarrow 1 \rightarrow 5 \rightarrow 13 \rightarrow 24 \rightarrow IC$ and is found after only 7 node visits. We must extend DFS to continue searching for better paths in order to improve the code generated. After the first solution the search will go on to reach $FG \rightarrow 1 \rightarrow 6 \rightarrow 15 \rightarrow 26$; at this point we could search further to find IC but since we already have a solution of length 5, we know that no solution found will be better than 5 instructions so we limit the search depth. Next we go back to node 1 and we see that we

could go to 15, but have already been here. If we simply ignored this node because it might risk getting trapped in an infinite cycle we would miss an optimal solution. In Cain we store a mapping from goal-bag to a cost function, the search depth for example, when we visit a node, and use this to check if we have seen a node before. If this visit took fewer instructions to reach, we update the map and continue searching. An optimal solution, $FG \rightarrow 1 \rightarrow 15 \rightarrow 26 \rightarrow IC$, is found after 15 node visits. At this point we have not visited enough nodes to know that this is optimal, so the search continues, now with the lower maximum depth of 3. After 23 node visits* the extended, depth limited, DFS search has shown that the solution with 4 instructions is optimal. We see that on 3 occasions a node is visited twice, this is because faster paths to the nodes were found. We re-process the node because the alternative is to store the whole graph as we produce it, which would make the memory overhead too onerous.

5.3.3 A* SEARCH

The A* Search Algorithm [15] takes an heuristic lead approach based on the cost required for the optimal path from FG to IC that passes through any goal-bag n. This cost is denoted as $f(n) = g(n) + h(n)$ where $g(n)$ is the cost from FG to n and $h(n)$ the cost from n to IC. A* guarantees to find the optimal path if we can estimate $g(n)$ and $h(n)$ denoted as $\hat{g}(n)$ and $\hat{h}(n)$ respectively.

$\hat{g}(n)$ can be evaluated simply as the minimum cost found so far to each n as the search progresses through the nodes in the graph. $\hat{h}(n)$ on the other hand is more difficult: to find the optimal path the A* algorithm requires the guarantee that $\hat{h}n \leq h(n)$. This is a trivial problem for physical systems where for example $\hat{h}(n)$ can be defined as the euclidean distance from n to the target node. Such an estimate is not simple to produce for goal-bags, however. Though it would be possible to use $\hat{h}(n) = 0$, this would lead to effectively just recreating the breadth-first-search with no heuristics. We can relax the constraint that $\hat{h}$ must not be an overestimate at the cost of not finding the optimal solution first - this can be allowable for this search problem because achieving provably optimal results is expected to be intractable and code generation is always a compromise between compile time and code efficacy. In Section 6.4 we see that we currently have no effective way to produce any useful heuristic for comparing any two goal-bags, so while Cain supports using the A* algorithm it is unused.

5.3.4 STOW-OPTIMISED-TRAVERSAL

We now propose a novel graph traversal algorithm that aims to solve the problem of having poor quality heuristics on large graphs, that can only be defined as an ordering of child nodes. Consider for an node n there are a set of 'successor' nodes, Γ_n, that are the nodes for which edges from n connect. We assume that the only heuristic available is that for a node n we can sort Γ_n from best to worst. This heuristic is good enough to perform the Depth-first-search with heuristics presented in Section 5.3.2 but overcomes the primary problem that in

*counting the number of times a node's children are generated

a large graph where we can not expect to search much of it, poor choice in heuristic can get the search stuck in a large but fruitless part of the graph for long periods of time.

To implement Stow-Optimised-Traversal (SOT) we need to define the concept of a partially seen node. When a node is visited, an iterator of its children nodes, ordered by an heuristic, is created. In Cain this is the iterator defined by the Reverse Search Algorithm 1 as it yields. If we want to visit a child node we must take from the iterator a node giving us a new child node and the parent node is partially visited. For example if we have node 1 in Figure 5.3.1 we write it as an iterator of node 1 denoted as 1_5, if we want to visit a child of 1_5 then we receive node 5 from the iterator and 1_5 becomes 1_6. We consider every node as a partially seen node with its first child to be produced when visited. In abstract we use n_m to mean the partially seen node n that will next return its mth successor.

There are three main components to the traversal:

- The double ended queue, D, provides the order to search nodes.
- The cost cache, C, stops cycles in the graph while allowing us to search nodes we have already seen if it could provide a better solution.
- Storing the $minCost$ allows us to limit the search if there is no way to reach the target before any known solution, allowing for effective branch pruning.

Algorithm 2 shows a pseudo-code implementation of SOT, based on the notion of storing iterators of child nodes. We assume that the iterator is lazy, such that when created it stores what information it needs, but the computational expense of generating the children nodes to iterate through, and applying any heuristic is put off until at least the first invocation of the iterator.

The $cost$ function can implement any metric of total cost as long as it is monotonically increasing as the path is appended to. This is required so that we can be sure that if the cost of a path $S \rightarrow^* n$ is higher than the lowest cost found so far for a path $S \rightarrow^* T$ there is no path $S \rightarrow^* n \rightarrow^* T$ with a lower cost than the lowest cost found so far. ϵ is the minimum increase in cost for adding a new node to the path, or the minimum extra cost to reach the target. This extra value acts partially like $\hat{h}$ in A* but we use it only to cut branches as early as possible, without needing to generate children. In Cain we set the cost function to be simply the length of the path for Scamp5 under the assumption that all instructions take the same time. We set ϵ to 1 since if the current node is not target node, then the target node is at least one edge, and so the cost of one instruction, away. In Cain the $iterator$ function is the reverse search algorithm, returning a goal-bag iterator, along with information about what instruction was chosen and it's register requirements.

Figure 5.3.3 shows how SOT traverses through our example graph. One of the problems in DFS was that it would not search node 2 for a significantly long time, if the node 1 had been a red herring DFS will waste lots of time searching the whole sub-graph connected to 1 before looking to other children of FG. This means the search spends more time traversing at deeper depths, and so effectively relies on heuristics at shallow depths being reliable. We see that SOT changes this by having the traversal dive deep, but 'fail-fast' when a dead end is

Result: List of nodes making the path from S to T
minCost = ∞
$C = \{\}$
$D = []$
$D \leftarrow_{\text{front}} (\text{iterator}(S), [S])$
while D *is not empty* **do**
  $n_m, P = \leftarrow_{\text{front}} D$
  if $m > 1$ *or* $C_n > cost(P)$ **then**
    $C_n = \text{cost}(P)$
    if $n = T$ **then**
      minCost = cost(P)
      yield P
    else if $cost(P) + \epsilon < minCost$ *and* n_m **then**
      $m, n_{m+1} = n_m$
      $D \leftarrow_{\text{front}} (\text{iterator}(m), P + [m])$
      $D \leftarrow_{\text{back}} (n_{m+1}, P)$
    end
  end
end

Algorithm 2: The Stow-Optimised-Traversal algorithm. $b, a_{b+1}, = a_b$ takes b from the iterator a_b, where a_{b+1} is the rest of the iterator. $\leftarrow$ is used to mean adding or polling the double ended queue. C is a mapping from each node to the shortest path found so far.

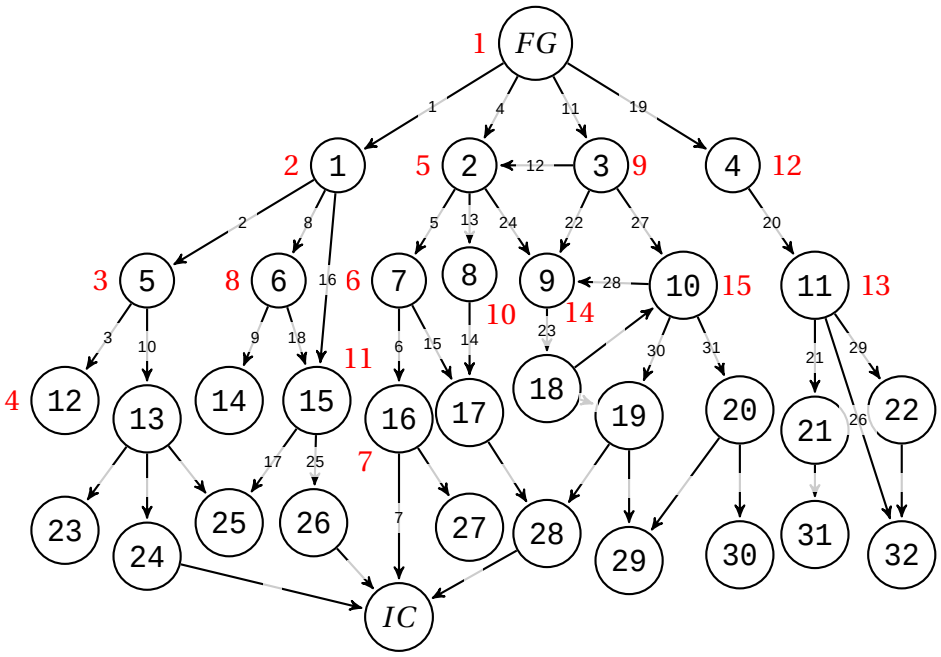


Figure 5.3.3: Graph representation of the search space using SOT. Edges are numbered in the order they are considered, and nodes are numbered, in red, in the order that their children need to be created.

found, like at node 12. At this point SOT goes back to *FG* and goes down node 2 effectively saving the computation of completely searching areas of the graph that are not connected to *IC*. In doing this, SOT does rely on heuristics deeper in the graph to be reliable since lots of good progress can be put on hold with one dead-end. When we look at the order in which the heuristic is trusted, we see that the heuristic at *FG* is trusted first, then down in a greedy traversal to node 12, but then we assume that *FG* was wrong and look at node 2 instead. Once we have done a search of the first children in succession of node 2, SOT goes to node 6, in effect saying that we trust the heuristic at *FG* that suggested node 1 over node 3, but it was the heuristic at node 1 that was wrong.

5.3.5 HEIR-ORDERED-SEARCH

A similar but different search approach, used in Linnea[1], is to search the first child of every node we current know of, then the second, then third, and so on. This aims to solve the same problem as SOT but in a slightly different way. Using a priority queue, in Heir-Ordered-Search, there is no guaranteed order of which nodes should be picked to provide the next child if they have all produce the same number of children so far. If we assume preference is given to the partially seen nodes that were put in the priority queue first, then we see Heir-Ordered-Search would follow the same traversal as SOT for the first 10 edges in the example graph. This is assuming the same branch pruning and cost cache features are used. Cain also supports Heir-Ordered-Search as an option.

We have seen that there are many graph traversal algorithms available, each with their positives and negatives. The challenge when picking a suitable traversal algorithm is primarily about what extra information is available by way of heuristics and the quality of that information. This as well as the memory constraints of large graphs makes algorithms like Breadth-First-Search and A* Search unviable for the graphs Cain generates when producing Scamp5 code. Depth-First-Search, Heir-Ordered-Search, and Stow-Optimised-Traversal all take advantage of the heuristics available, with their own ways to mitigate poor quality heuristics, and prioritise different parts of the graph.

5.4 REGISTER ALLOCATION

So far our search as not considered register allocation, and so the search has effectively assumed that register allocation will always work, and there will be enough registers. This means many of the nodes we visit may actually be invalid, and others definitely lead to invalid solutions, even if they themselves are valid. There are two parts to register allocation that affect the search mechanisms, firstly we have branch pruning, and secondly we have the register allocation algorithm itself. In Cain every atom's 3rd dimension is used to allow for multi-channel kernels, using the input registers effectively as a virtual 3rd dimension.

Interestingly this feature required no changing of the instruction definitions for Scamp5.

Cain ensures the instructions implemented for any architecture are able to assert their various register allocation requirements, such as on Scamp5, as seen in Table 6.1.1. This informs the reverse search algorithm (Algorithm 1) to firstly allow multiple lower goals that have the same atoms to be produced only once (by only adding one of the goals to the new goal-bag), but also to ensure pruning takes into account times when upper goals cannot be written to the same register as their lower goal operands.

5.4.1 PRUNING

Every node in the graph that makes up our search space is defined by a goal-bag. A goal is a collection of atoms that is transformed and combined by instructions and so goals represent the contents of registers. Therefore a goal-bag is the representation of many registers, but without a limit, or persistent ordering. It is simple to see that as we traverse the graph and reach a node, we can judge whether its goal-bag will fit into the available registers of the Processing Elements. We only consider architectures without memory in their processing elements, so spilling is not possible. If a goal-bag contains too many goals we simply stop searching further. This means the *generatePairs* function from Algorithm 1 doesn't need to be dependent on the number of registers available, though for heuristics it may take this into consideration. The other side of this is that it is quite possible for a valid reordering of the instructions to be allocatable and so this means our search is now not only defining what operations need to be done, but what order they should be done too.

This pruning however is not optimal - it can be improved by also looking at how much further we have before we reach our current cost limit, and how quickly the size of the goal-bag can be shrunk to one, as is necessary for it to be IC. For example, if we assume:

- Every instruction costs 1 PE clock cycle.
- Applying any instruction forwards on a goal-bag of size n will produce a goal-bag of maximum size $n+1$.
- We have found a solution with a cost of 5 already.

Then if we visit a node with 3 registers in use, and a cost so far of 3, then we know there are no instructions that can be applied to the IC goal-bag that will allow it to use 3 registers within 1 instruction, and if it were possible in 2 instructions then we wouldn't have reduced the cost, so we can prune this node early.

What must also be accounted for is that instructions may have restrictions on what registers or combinations of registers may be used. Many instructions in Scamp5 have such restrictions, such as in the $\texttt{add}(a,b,c)$ instruction, b must not be the same register as c, and in $\texttt{sub}(a,b,c)$, a must not equal c. This means in the reverse search algorithm, when we apply a pair we must look at the actual instruction being used, and determine if the Lowers can be simplified where the goals are the same or if they must be separate, and if the instruction's result must be in a different register to its operands then there is an extra register live at that time, and so that node may need to be pruned.

5.4.2 LINEAR SCAN

In the generation of code we never need to consider data dependent control flow. This is simply because the idea of running these filters on a SIMD cellular processor array is that all the Processing Elements do the same thing, and that the filters are a function linear combination of the inputs. This has the nice effect that all the code produced can be considered as a single 'basic block' for register allocation, and since the order of instructions is set, and we know that the live variables always fit into the registers, we can use the Linear Scan[21] algorithm to assign goals to registers. There are only a couple of important caveats to address; the input and output registers must be correct. This is solved in Cain by doing the assignment in reverse so the final output registers are guaranteed to be free, and then tracking when the initial-goal register is needed and ensuring it will be free by then. This process can however fail in some cases, for example:

- There is an Identity-Kernel as an output that is not at the input location. The reverse search algorithm would assume that the input and Identity-Kernel can be put in the same register but if this not the case, register allocation will fail.
- A kernel is produced into the input register while the initial-goal is still live. This would require some way to move the input instruction or output kernel into a different register which is not considered a part of the reverse search.

These edge cases mean that even if the search algorithm finds a plan from FG to IC it may not in fact produce usable code at the cost it proclaims. To solve this register allocation is performed before a plan is accepted.

The search algorithm used allows for a series of unary-instructions to be applied if the node being searched has the same number of goals as the initial-goal bag, but this is trivially implemented and doesn't affect the theory of the search mechanism, other that we do not bother to check the cost of these instructions, allowing Cain to find plans that are longer than previous plans by a few instructions. In this chapter we saw how the problem is defined as a graph traversal, and how goals are used, like in AUKE, to allow the definition of instructions to take goal inputs and return outputs, the Lowers and Uppers respectively. Now we have the background understanding to look at how heuristics, specifically for Scamp5, are used to inform the graph traversal.

Chapter 6

Scamp5 Pair Generation on Cain

In this chapter we further discuss the instructions available in the Scamp5 architecture, and specifically how these instructions inform the design of the *generatePairs* function required for the reverse search algorithm (Algorithm 1). Cain has been designed to make the *generatePairs* functionality a modular component allowing for different architectures to be targeted for code generation without a complete rewrite of the reverse search implementation.

6.1 Instructions

One of the aims of Cain was to develop a more general approach to defining and enabling the instruction used. In AUKE, elemental transformations are the basis for conversion to Scamp5 or other architectures used in simulation. This, however, limits the useful application of AUKE to CPAs with the same inherent limits such as having no multiply instruction. Cain lifts these limits by allowing for the arbitrary design of instructions, at the cost of implementing the *generatePairs* function for each architecture. This cost can however be mitigated with good implementation design and code-reuse. Cain is still generally limited to SIMD PEs, but as long as some instruction or sequence of instructions produces a goal output from some number of goal inputs and has a consistent effect on PEs, it could be implemented in Cain. In theory an instruction in Cain could be implemented that does multiplication by an integer. Cain, as it is currently implemented is still limited to performing linear combinations of its inputs, though this idea is expanded on in section 9.2.6.

Cain implements all of the arithmetic and neighbour macro analogue instructions available in the Scamp5 API, except `abs`. This is omitted because `abs` cannot be represented in a linear kernel. A full list of the supported instructions is found in table 6.1.1.

Table 6.1.1: Scamp5 Macro Analogue Instructions, in Cain and AUKE. $a, b, c...$ are registers, while *dir* can be any of `north`, `east`, `south`, or `west`. Where the operands do not correspond with the Uppers, registers must be clobbered; we only shows the obvious intended inputs and outputs in the examples. Atoms are coloured for clarity; purple Atoms are a combination of red and blue.

Name	Function	Atom Example		AUKE	Cain
		Uppers	Lowers		
`mov`(a,b)	$a := b$	$\lfloor 1 \rfloor$	$\lfloor 1 \rfloor$	No*	Yes
`res`(a)	$a := 0$	$\lfloor 0 \rfloor$		No	Yes

Name	Function	Atom Example		AUKE	Cain
		Uppers	Lowers		
res(a,b)	$a := 0,\ b := 0,\quad a \neq b$	$\begin{bmatrix}0\end{bmatrix}, \begin{bmatrix}0\end{bmatrix}$		No	Yes
add(a,b,c)	$a := b + c,\quad b \neq c$	$\begin{bmatrix}1 & 2\end{bmatrix}$	$\begin{bmatrix}1 & 1\end{bmatrix}, \begin{bmatrix}0 & 1\end{bmatrix}$	Yes	Yes
add(a,b,c,d)	$a := b + c + d,$ $b \neq c,\quad b \neq d,\quad c \neq d$	$\begin{bmatrix}1 & 0\\ 2 & 1\end{bmatrix}$	$\begin{bmatrix}1 & 0\\ 1 & 0\end{bmatrix}, \begin{bmatrix}0 & 0\\ 1 & 0\end{bmatrix}, \begin{bmatrix}0 & 0\\ 0 & 1\end{bmatrix}$	No	Yes
sub(a,b,c)	$a := b - c,\quad a \neq c$	$\begin{bmatrix}1 & -1\end{bmatrix}$	$\begin{bmatrix}1 & 0\end{bmatrix}, \begin{bmatrix}0 & 1\end{bmatrix}$	Yes	Yes
neg(a,b)	$a := -b,\quad a \neq b$	$\begin{bmatrix}1 & -1\\ 0 & 2\end{bmatrix}$	$\begin{bmatrix}-1 & 1\\ 0 & -2\end{bmatrix}$	Yes	Yes
abs(a,b)	$a := \lvert b\rvert,\quad a \neq b$	$\left\lvert\begin{bmatrix}1 & 0\\ 0 & -2\end{bmatrix}\right\rvert$	$\begin{bmatrix}1 & 0\\ 0 & -2\end{bmatrix}$	No	No
divq(a,b)	$a := \frac{1}{2}b + error,$ $a \neq b$	$\begin{bmatrix}1 & 0\\ 2 & 1\end{bmatrix}$	$\begin{bmatrix}2 & 0\\ 4 & 2\end{bmatrix}$	Yes	Yes
div(a,b,c)	$a := \frac{1}{2}c$ $b := \frac{-1}{2}c + error$ $c := c + error$ $a \neq b,\quad a \neq c,\quad b \neq c$	$\begin{bmatrix}1 & 0\\ 2 & 1\end{bmatrix}$	$\begin{bmatrix}2 & 0\\ 4 & 2\end{bmatrix}$	Yes†	Yes
div(a,b,c,d)	$a := \frac{1}{2}d$ $b := \frac{-1}{2}d + error$ $c := d + error$ $a \neq b,\quad a \neq c,\quad a \neq d$ $b \neq c,\quad b \neq d$	$\begin{bmatrix}1 & 0\\ 2 & 1\end{bmatrix}$	$\begin{bmatrix}2 & 0\\ 4 & 2\end{bmatrix}$	No	Yes
diva(a,b,c)	$a := \frac{1}{2}a$ $b := \frac{-1}{2}a + error$ $c := a + error$ $a \neq b,\quad a \neq c,\quad b \neq c$	$\begin{bmatrix}1 & 0\\ 2 & 1\end{bmatrix}$	$\begin{bmatrix}2 & 0\\ 4 & 2\end{bmatrix}$	Yes†	Yes
movx(a,b,dir)	$a := b_{dir}$	$\begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}$	$\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix}$	Yes	Yes
mov2x$(a,b,dir1,dir2)$	$a := b_{dir1,dir2}$	$\begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}$	$\begin{bmatrix}0 & 0\\ 1 & 0\end{bmatrix}$	No	Yes

Name	Function	Atom Example		AUKE	Cain
		Uppers	Lowers		
`addx`(a,b,c,dir)	$a \coloneqq b_{dir} + c_{dir}$, $b \neq c$	$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$	No	Yes
`add2x`$(a,b,c,dir1,dir2)$	$a \coloneqq b_{dir1,dir2} + c_{dir1,dir2}$, $b \neq c$	$\begin{bmatrix} 0 & 2 \\ 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$	No	Yes
`subx`(a,b,c,dir)	$a \coloneqq b_{dir} - c$ $a \neq c$	$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix}$	No	Yes
`sub2x`$(a,b,c,dir1,dir2)$	$a \coloneqq b_{dir1,dir2} - c$ $a \neq c$	$\begin{bmatrix} 0 & -2 \\ 2 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 2 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 2 \\ 0 & 0 \end{bmatrix}$	No	Yes

As we can see, Cain is able to generate code using far more of the instructions available than AUKE. Making full use of instructions is principally what gives Cain advantages over AUKE in generating the shortest code.

6.2 EXHAUSTIVE PAIR-GENERATION

The first of two ways that Cain generates pairs, we call the Exhaustive approach. In this approach a *pairList* is created to store every pair, each representing an *Uppers* list, a *Lowers* list, and the relevant instruction. For every goal G in the current goal-bag C we consider all the Unary-instructions, meaning `mov`, `neg` and the division instructions, as well as `movx`, and `mov2x` for all the direction arguments they can take. All these instructions create one output, and so we apply them in reverse such that the *upper* is G and the *lower* is whatever is required for the instruction to produce G. These are pairs, and are all added to the *pairList*.

Then we consider instructions with multiple inputs - it would be intractable to allow complete freedom here since for an `add`(a,b,c) instruction the ways to produce any G are infinite, with use of a negative input. Instead of allowing all possible instructions we generate splits of the goal G, which still produces in the order of 2^n results as we have shown in equation 5.2.2.

We also consider the potential for the minimally signed digit version of counts of atoms to be utilised[17]. If G contains 7, 14, or 15 of any atom, then we compute a goal G' with 8, or 16 respectively, and the required negative G^- that would allow G to produced. These are added to the list of splits looked at. We check for 7, 14, and 15 manually rather than a more complete algorithm because the splitting process is to slow on large goals anyway, so larger counts of an atom won't be seen.

*AUKE does not properly account for Scamp5 instructions that have restrictions on registers appearing multiple times, so there is no need for a simple register move instruction.

†AUKE can only achieve this by reserving a register for division, allowing for `divq` to be 'upgraded', a process that is effectively manual.

For each split we can generate an ($add(a,b,c)$ instruction pair, and two $\texttt{sub}(a,b,c)$ instruction pairs. The assumption we make is that it is adequate to constrain the *Lowers* to containing sub-goals of G and their negations. All the respective 'x' versions of the $\texttt{add}(a,b,c)$ and $\texttt{sub}(a,b,c)$ instructions are also generated, widening this assumption slightly to be that *Lowers* may contain translated sub-goals. For the $\texttt{add}(a,b,c,d)$ instruction, each of the splits is split again, producing all the ways to split G into 3 parts. The pairs created in this process are all added to the *pairList*.

Finally this list is sorted via a cost function to ensure the instruction with the most promise in reaching the initial-goal are put first. This list is returned as an iterator from the *generatePairs(C)* function. While not truly exhaustive because of the restrictions placed by taking sub-goals of G, this method is a compromise between full and complete searching of the solution space, and tractable computing power, that leans towards the former and we have found to be successful for Scamp5 and the filters tested.

6.3 Atom Distance-Based Pair-Generation

The Atom Distance-Based approach is more involved, and has tighter assumptions on allowable *Lowers*. In this method, like before, we create a *pairList* to store all the instructions we consider. We look at every $(A,B) \in C^2$; and create a *distance-map* where the key is the distance between two atoms as well as if negation occurred, and the value is a Goal, initially empty. Then for each atom $a \in \text{supp}A$ and atom $b \in \text{supp}B$ we add the goal in the *distance-map*, where the key is the distance between a and b, the minimum of $count_a^A$ and $count_b^B$ worth of b atoms. If A is the same goal as B (not simply that they contain the same atoms, but are actually equivalent) then we add half the number of atoms to the goals in the *distance-map*. This gives us the property that the *distance-map* contains sub-goals of B that can be produced by sub-goals of A, followed by a possible translation and negation.

For every key-value entry in *distance-map*, (d,T) we can add Pairs to the *pairList*. If T is a translation of A then the appropriate `neg`, `movx` and `mov2x` instructions are added to *pairList* that produce A given some *lower* that is closer to being T. Otherwise T is a sub-goal of B and T_{-d} is a sub-goal of A. In this case we split A by taking away all the atoms in T_{-d} and produce the `Add` and `Sub` instructions and their 'x' varieties where A is the *upper* and T_{-d} (or a translation there of) is one of the lowers, and the other *lower* is the trivially calculated. The $\texttt{add}(a,b,c,d)$ instruction is not used here.

If A is the same goal as B this process is amended, the empty-translation is removed; and `mov`, `movx`, and `mov2x` are not considered. Instead we have that T is a sub-goal of A, and T_{-d} is a sub-goal of A, and so we can split A by removing the atoms of T_{-d} like before, and also split A into three parts, removing T and T_{-d} to produce an $\texttt{add}(a,b,c,d)$ instruction. Again, the 'x' variants are added too, except instead of following d, the translations move the operands closer to the centre of the kernel.

Once all the pairs have been added, and every $(A,B) \in C^2$ checked, the *pairList* is sorted by the heuristic. The Atom Distance-Based method for generating Pairs makes far more assumptions than the exhaustive search, but this allows it to run much faster for large goals

in C. The assumptions include that the best translation to apply is either in the direction of a common pattern found in another goal, which seems perhaps sensible, or towards the centre of the kernel, which can easily be a mistake if a split that is produced has a common pattern with an existing goal that is not nearer the centre of the kernel. This is, however, the compromise made to support large kernels that otherwise would take too long to compute using the exhaustive method.

6.4 HEURISTICS

The heuristics used in the search for efficient code are vitally important. We have seen that the search space quickly becomes enormous with many possible sequences of instructions that can be traversed. The primary component that allows Cain to function more effectively than the Answer Set Programming implementation of the same problem is the heuristics that guide the search towards the solution rather than an uninformed guess at the best instructions to search.

6.4.1 GLOBALLY COMPARABLE HEURISTICS

Many graph traversal algorithms, as we have seen in section 5.3, require as heuristic $\hat{h}(n)$ that can estimate the cost of reaching the end of the path from any node. To be optimal the heuristic $\hat{h}(n)$ should always be more than a the minimum bound, and less than or equal to $h(n)$ the cost of true optimal path from n to the end node. Deriving such an heuristic for goal-bags proves non-trivial; unlike with physical distance where an euclidean distance can is a simple to calculate minimum bound that serves well as $\hat{h}(n)$, with goal-bags the simple heuristic is a maximum bound, that predicts the cost of moving atoms one by one to the centre of the kernel and adding them up. This heuristic can still be used for $\hat{h}(n)$ at the cost of no longer being guaranteed to find the optimal path.

To explore this idea we configured Cain to use an A* search with the following heuristic:

$$\begin{aligned} \hat{f}(n) &= g(n) + \hat{h}(n) \\ g(n) &= \mathit{depth} \\ \hat{h}(n) &= \sum_{G \in C} \left(-1 + \sum_{a \in G} \left(\mathit{Manhattan\ distance\ of\ a} + 1 \right) \right) \end{aligned} \tag{6.4.1}$$

The premise is that every atom needs to be moved to the correct location, and added to the goal, with the -1 account for the first atom in a goal not needing to be added to anything. This heuristic is simplistic and doesn't account for the features in Scamp5 such as being able to move to PEs in one instruction, or adding 3 registers up together, but fixing those issues will not help the heuristic to quantify the value of common sub-goals and exploit them. This method also has the problem of increased memory usage to store the large set of frontier nodes. Although nodes to not need to be stored with their partially seen state, the number of nodes quickly increase the memory footprint of the search. Other algorithms such as Beam

search mitigate this by restricting the size of the frontier set, but then choosing which nodes to eliminate can become a slightly different, yet just as hard problem.

One of the difficult compromises to make when designing an heuristic is to consider how we might like a complex goal to be split into multiple parts. We see that no matter how a goal is split, the $\hat{h}(n)$ we use in 6.4.1, will produce the same cost, but we know that some splits are far more efficient, such as in the Sobel-Filter:

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 1 \end{bmatrix} \tag{6.4.2}$$

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & -1 \end{bmatrix} \tag{6.4.3}$$

Clearly 6.4.2 is a far more effective way to achieve the Sobel Filter, but $\hat{h}(n)$ has no means to reason about it. In both 6.4.2 and 6.4.3 the heuristic applied the right hand side would produce 18. Furthermore we can see that applying the heuristic to the original filter gives us 19, showing that this heuristic values splitting goals up into sub-goals. This becomes a problem, however, later in a search when we may have the following options:

$$\left\{ \begin{bmatrix} 0 & 0 & 0 \\ 4 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \right\} \leftarrow \left\{ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \right\} \tag{6.4.4}$$

$$\left\{ \begin{bmatrix} 0 & 0 & 0 \\ 4 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \right\} \leftarrow \left\{ \begin{bmatrix} 0 & 0 & 0 \\ 4 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \right\} \tag{6.4.5}$$

If we assume that the initial-goal has 4 centred atoms, then 6.4.5 is a better instructions to use, since it is clearly just one move away from the initial-goal. But, under $\hat{h}(n)$ 6.4.4 gives us a cost of 6, whereas 6.4.5 gives us 7. This shows that since $\hat{h}(n)$ favours separate goals over larger goals it tends to produce many one atom goals rather than larger goals that are closer to the initial-goal.

We have found that producing a reliable and effective heuristic for $h(n)$ is not feasible, and requires substantive methods to find common sub-expressions that are computationally expensive and when simply drive the search to find programs that mimic the behaviour of the heuristic more closely than the actual problem, or instruction set.

6.4.2 Local Heuristics

Instead of developing a globally applicable heuristic we can develop an heuristic that only applies to child nodes of a parent. This is to say that we can order the children nodes of a

parent node, but make no assumptions as to how this child node compares to its cousin. This fundamentally changes the question from 'How quickly can this goal-bag be solved?' to 'How much better is this goal-bag than its siblings?'. This approach allows the heuristic design to not worry about absolute values and being generally correct, and instead focus on what is good and bad, locally in the graph, about this particular child.

For Scamp5 in Cain we implement a cost function to determine the ordering of child nodes, it takes as input a the new goal-bag C of the child node. Though this C is actually an approximation that doesn't account for operand register interference checks such as defined in Table 6.1.1.

$$cost(C) = distances(C) + repeatedGoals(C) + divisions(C) \tag{6.4.6}$$

$$distances(C) = \sum_{G \in C} (d(G,C) + |G|) \tag{6.4.7}$$

$$d(G,C) = \begin{cases} \sum_{a \in G} \big(manhattan(a) + p(a)\big) & \nexists B \in C.B \neq G \wedge G \subset B \\ \frac{1}{2} \sum_{a \in G} \big(manhattan(a) + p(a)\big) & \text{otherwise.} \end{cases} \tag{6.4.8}$$

$$p(a) = \begin{cases} 1 & \text{a is negative} \\ 0 & \text{otherwise.} \end{cases} \tag{6.4.9}$$

$$repeatedGoals(C) = \sum_{G \in patterns(C)} \begin{cases} |G|^2 & \exists a, b \in G.a \neq b \\ 0 & \text{otherwise.} \end{cases} \tag{6.4.10}$$

$$patterns(C) = \begin{array}{c} \text{the subset of supp. } C \text{ with uniqueness defined as:} \\ unique(A,B) \coloneqq \nexists T \in Translations.A_T = B \end{array} \tag{6.4.11}$$

$$divisions(C) = \frac{\max(\{count_a^G | G \in IC, a \in G\})}{\min(\{count_a^G | G \in C, a \in G\})} + \frac{\max(\{count_a^G | G \in C a \in G\})}{\min(\{count_a^G | G \in IC, a \in G\})} \tag{6.4.12}$$

Equation 6.4.6 splits the cost function into 3 main parts. Firstly we count up all atoms and also their distances and add an extra cost for negatives. This serves the primary deterrent for the problem getting larger by penalising more atoms, though we include a relief for when a goal G is a sub-goal of another goal in C such that we don't over penalise moves that allow us to take advantage of G to produce other goals.

The *repeatedGoals* function targets patterns in the goals of C, heavily penalising every repeated pattern for its size, but avoiding patterns that contain all the same atom as not to discourage instructions that use use larger parts of the initial-goals. This ability to reward common goals heavily incentives the search to use common sub-goals.

The final section is concerned with the scale of the goals in C, It takes the ratios of numbers of atoms in the initial-goals and the min and max count of atoms in the goals in C to penalise the use of goals with larger numbers of the same atom as in IC and also for having goals with smaller counts. This helps to stop the search always splitting down to singleton goals where it isn't needed.

This cost function is a creation of trial, error, and intuition and is balanced carefully for the Scamp5 instruction-set. Allowing it to exist only as a comparative means of heuristic between sibling nodes allowed us to remove the idea that it should represent the cost to

reach *IC* and instead allowed us to tailor it to ensure that the next move is the best it can be.

This has shown us how performing the search in reverse is vital, since producing a heuristic to achieve a arbitrary final-goals would be far more complex, and most likely nothing like as effective, whereas making the assumption that all atoms eventually need to end up on the middle allows for far simpler reasoning about any heuristic.

6.5 KERNEL SIZE AND CONSEQUENCES

This this section we look at the example of a Sobel-filter[23], with both vertical and horizontal kernels. We see how Cain is able to out perform AUKE in terms of shortened code, and the costs that Cain unwittingly accepts in order to achieve this. The *Final-Goals* as presented to Cain to solve is:

$$\begin{aligned} FG &= \left\{ \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \right\} \\ &= \left\{ \begin{bmatrix} (-1,1,0,+), & (1,1,0,-), \\ (-1,0,0,+), & (1,0,0,-), \\ (-1,0,0,+), & (1,0,0,-), \\ (-1,-1,0,+), & (1,-1,0,-) \end{bmatrix}, \begin{bmatrix} (-1,1,0,+), & (-1,-1,0,-), \\ (0,1,0,+), & (0,-1,0,-), \\ (0,1,0,+), & (0,-1,0,-), \\ (1,1,0,+), & (1,-1,0,-) \end{bmatrix} \right\} \end{aligned} \tag{6.5.1}$$

Given that the initial-goal is a single centred positive atom Cain produces a plan able to compute this filter in 9 Scamp5 instructions. The plan does not have register allocation, so constraints on registers are not followed where it is immaterial to the plan, and numbers are used instead, that refer to the written order of the goals in the current goal-bag as any step. The first steps makes full use of Scamp5's somewhat unintuitive implementation of `sub2x` producing the following series of goal-bags:

$$\begin{aligned} FG = &\left\{ \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \right\} \\ &\uparrow \quad \texttt{sub2x}(0,0,\texttt{west},\texttt{west},0) \\ &\left\{ \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \right\} \\ &\uparrow \quad \texttt{sub2x}(1,1,\texttt{south},\texttt{south},1) \\ &\left\{ \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \right\} = C_2 \end{aligned} \tag{6.5.2}$$

At this stage, we can already see a compromise that has been made to the validity of the filter however. When running a convolutional filter it is expected that the size of the output

is smaller than the input since the edge pixels will be undefined if they are a function of neighbouring pixels that don't exist. So for an $n \times n$ filter a band of edge pixels $(n-1)/2$ are expected to be invalid. In this program so far, however, the (sub2x) instructions as assumed that a PE exists 2 steps over, instead of just 1. This means that the band of invalid edge pixels on the western and southern boarders of the array is thicker. This can have serious consequences in applications of the produced code since, if unaccounted for, garbage values can be returned to the host. This sort of behaviour occurs because Cain assumes an infinite sensor size and make no restrictions on movements. We can try to quantify this behaviour by tracking the translations of atoms, In this case it is trivial since in a single instruction atoms are moved twice in a single direction, so to achieve the FG we have to assume that a PE exists that is 2 steps to the west and also a PE 2 steps to the south. When we validate the code, if we track for each register all the positions that were used in making the value it holds, such that when an operation occurs we include the position of the operand registers and all their tracked positions we can build a map of the computational shape of the generated code. We can represent this as kernels where values only exist for positions that are required for the computation, we call this the effective shape of the kernel given the implementation:

$$\left\{ \begin{bmatrix} . & . & . & . & . \\ . & 1 & . & -1 & . \\ 0 & 2 & 0 & -2 & . \\ . & 1 & . & 1 & . \\ . & . & . & . & . \end{bmatrix}, \begin{bmatrix} . & . & . & . & . \\ . & 1 & 2 & 1 & . \\ . & . & 0 & . & . \\ . & -1 & -2 & -1 & . \\ . & . & 0 & . & . \end{bmatrix} \right\} \qquad \begin{aligned} d &= 1 \\ d_e &= 2 \end{aligned} \tag{6.5.3}$$

The generated code for the Sobel-filter continues (in reverse) with a strategy to produce are current goal-bag C_2 from IC:

$$\begin{array}{c} C_2 = \left\{ \begin{bmatrix} 0&0&1 \\ 0&0&2 \\ 0&0&1 \end{bmatrix}, \begin{bmatrix} -1&-2&-1 \\ 0&0&0 \\ 0&0&0 \end{bmatrix} \right\} \\ \uparrow \\ \left\{ \begin{bmatrix} 0&0&1 \\ 0&0&2 \\ 0&0&1 \end{bmatrix}, \begin{bmatrix} 0&0&0 \\ 0&-1&-1 \\ 0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0 \\ -1&-1&0 \\ 0&0&0 \end{bmatrix} \right\} \\ \uparrow \\ \left\{ \begin{bmatrix} 0&1&0 \\ 0&1&0 \\ 0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0 \\ 0&-1&-1 \\ 0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0 \\ -1&-1&0 \\ 0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0 \\ 0&1&0 \\ 0&1&0 \end{bmatrix} \right\} \\ \uparrow \\ \left\{ \begin{bmatrix} 0&1&0 \\ 0&1&0 \\ 0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0 \\ -1&-1&0 \\ 0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0 \\ 0&1&0 \\ 0&1&0 \end{bmatrix} \right\} \\ \uparrow \\ \left\{ \begin{bmatrix} 0&1&0 \\ 0&1&0 \\ 0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0 \\ 0&-1&0 \\ 0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0 \\ 0&1&0 \\ 0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0 \\ 0&1&0 \\ 0&1&0 \end{bmatrix} \right\} \\ \uparrow \\ \left\{ \begin{bmatrix} 0&0&0 \\ 0&-1&0 \\ 0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0 \\ 0&1&0 \\ 0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0 \\ 0&1&0 \\ 0&1&0 \end{bmatrix} \right\} \\ \uparrow \\ \left\{ \begin{bmatrix} 0&0&0 \\ 0&-1&0 \\ 0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0 \\ 0&1&0 \\ 0&0&0 \end{bmatrix} \right\} \\ \uparrow \\ \left\{ \begin{bmatrix} 0&0&0 \\ 0&1&0 \\ 0&0&0 \end{bmatrix} \right\} = IC \end{array} \qquad \begin{array}{l} \texttt{addx}(1,1,2,\texttt{north}) \\ \\ \texttt{addx}(0,0,3,\texttt{east}) \\ \\ \texttt{movx}(1,1,\texttt{east}) \\ \\ \texttt{subx}(1,1,\texttt{west},2) \\ \\ \texttt{subx}(0,1,\texttt{north},0) \\ \\ \texttt{subx}(2,1,\texttt{south},0) \\ \\ \texttt{neg}(0,0) \end{array} \tag{6.5.4}$$

This program has 9 instructions in total, making it only 2 instructions longer than the code AUKE generates for just one of the kernels. Instruction length, however, isn't the only metric that is important on Scamp5. Circuit-depth is the measure of how many operations an input value is used for though-out the program. Analogue operations induce noise into the values every time a value is written to, so while the number of instructions is an important indicator, we can achieve higher accuracy from Scamp5 with a more shallow circuit-depth. This program has circuit depths of 4 and 5 respectively. In general we can this of the circuit depth-problem as the difference between adding to an accumulator, or adding in a binary tree. While the number of add operations remains the same, using a binary tree reduces the circuit-depth from n to $\log(n)$, resulting in far less noise in the output.

For another example we can produce code for an approximated 5×5 Gaussian-filter. In this Example we see the code produced, in 19 instructions and a circuit-depth of 14, has the following effective shape:

$$\frac{1}{64}\begin{bmatrix} . & . & 0 & . & 0 & . & . \\ . & 0 & 1 & 2 & 1 & 0 & . \\ . & 1 & 4 & 6 & 4 & 1 & . \\ . & 2 & 6 & 10 & 6 & 2 & . \\ . & 1 & 4 & 6 & 4 & 1 & . \\ . & . & 1 & 2 & 1 & . & . \\ . & . & . & . & . & . & . \end{bmatrix} \qquad \begin{aligned} d &= 2 \\ d_e &= 3 \end{aligned} \tag{6.5.5}$$

Given d is equal to the maximum number of PE's away from the centre of the kernel in a single axis that an atom is; $2d+1$ is the width of the kernel. We hypothesise that given an optimal implementation of a kernel, the 'effective d', which we call d_e, is less than or equal to $2d$. For any kernel we observe that d_e is larger than d when a common pattern of atoms is produced that appears in the kernel twice, more than d PE's away from each other in any one axis. This means the maximum bound for this translation of the pattern is $2d$, since the subgoals must both exist within $2d+1$ PEs. If the translation is at most $2d$ in any one direction, then no PE is needed further away than this for an optimal implementation. Any implementation where $d_e > 2d$ cannot be better than one where $d_e = 2d$ since the translation cannot need to be any further than $2d$.

As we have seen, heuristics are vital part of producing efficient code. Without informed guidance the search algorithm stands little chance to find any solution, let alone an efficient one. Due to the size of the graph and the limits on both memory usage and reasonable compute time available for code generation, Cain is only a functioning solution to the problem of code generation if the heuristics used are effective. In AUKE, for a single Sobel kernel, the first step has 112 options to search through using AUKE's exhaustive search without max-set optimisation[7], whereas using Cain, and with the exhaustive pair-generation there are 4031 instructions that could be considered. Using the Atom Distance-Based method in Cain yields

52 potential instructions, which is small. This was the intention of developing the method, to be applicable where the exhaustive method is ineffective, defined dynamically by a threshold on the maximum number of atoms within a single goal of the current goal-bag.

Chapter 7

Implementation of Cain

In this chapter we look at the implementation of the algorithms shown in Cain. Cain is written in Java to take advantage of Object-Oriented paradigms that allow Cain to be configurable and adaptable to other Cellular-Processing-Array architectures. Using Java also allowed us to easily develop multi-threading functionality compared to something like Python. Java was chosen over C++ simply because of prior experience with the existing standard libraries, development tools and the generally more forgiving error messages, though this is all personal taste and has no real barring on the production of Cain.

7.1 Object Definitions

Cain defines an `Atom` as 3 integers and a boolean, representing the (x, y, z) coordinate as we have seen, and if the atom is positive, or negative. Cain also defines a comparator on `Atom`s such that `Goals`, which each are simply a wrapper for an `ArrayList<Atom>` can store atoms in a consistent order. This allows for efficient lookup within goals and sameness checking. One of the notable uses of making a `Goal` a wrapper for an `ArrayList<Atom>` is to define equality. In Cain, Goals have two types of equality that we call sameness and equivalence. Sameness simply implies that the atoms contained are the same, whereas equivalence means they refer to the same goal directly, these are analogous, and implemented with Java's `.equals()` method and `==` operator respectively.

Goal-bags are defined as a class that extends an `ArrayList<Goal>`, allowing for a few modifications and the imposing of a two stage ordering scheme. The first ordering scheme is to order by goal size, the second is to order by goal-size then the ordering of the atoms. The first ordering scheme is most commonly used and makes removal of goals from the goal-bag by sameness or equivalence more efficient. The second is used in the `GoalCache` class that uses a `ConcurrentHashMap<GoalBag, Double>`. Having the full order on the goals contained in the `ArrayList` in the goal-bag allows for the equality check done by the hash-map to check the sameness of goal-bags efficiently. Since the primary use of `GoalBags` is seen in the plan finding stage, before register allocation, the order of goals can be arbitrary but a consistent order allows for more efficient algorithms, and extending existing classes allowed development to focus on algorithmic design rather than designing data structures.

A `GoalPair` is another class consisting of a list of upper goals, a list of lower goals, and the `Transformation` object that takes the lowers to produce the uppers. A `Plan` object is the way Cain stores the steps taken to go from FG to C, and implements a linked list, where each `Plan` refers to the previous plan, and stores the current goal-bag and the last `GoalPair` used. This allows for a new `Plan` to be efficiently created without copying out lists of instructions from the plan it continues from.

`WorkState`s contain a plan, a depth, a goal-bag, C, and a goal-pair iterator. This allows the reverse search algorithm (1) to take a `WorkState` and produce two `WorkState`s, one with a child node, and one that continues to search the parent node.

`Atom`s, `Goal`s, `GoalBag`s, `GoalPair`s, `Plan`s, and `WorkState`s are all immutable or effectively immutable, allowing for multiple worker thread to run the Reverse Search algorithm concurrently, assuming the graph traversal algorithm has a way to share work.

7.2 Strategy Patterns

Cain makes significant use of strategy patterns[10], this allows things like the graph traversal algorithm, the pair generation function and the plan cost function to be configured simply and easily, and swapped at run-time.

Cain uses a relatively simple interface to define a traversal algorithm:

```
1  package uk.co.edstow.cain.traversal;
2
3  import uk.co.edstow.cain.structures.WorkState;
4
5  public interface TraversalSystem {
6
7      void add(WorkState child, WorkState next);
8      void add(WorkState child);
9      WorkState poll();
10     WorkState steal(TraversalSystem system) throws InterruptedException;
11
12 }
```

The Reverse Search algorithm starts by polling the traversal system for a `WorkState`. For every work state it gets one of two things happen: either the `WorkState`'s goal-pair iterator is finished; or two new `WorkState`s are put back into the `TraversalSystem`, the new child node, and the next step of partly seen parent node. The child node may be `null` if it known that the child is not valid, for example, because of register usage. The `add(WorkState child)` function is used to prime the system with the *FG* node, and if `poll()` returns `null`, meaning the system has no more `WorkState`s, then the steal function can be used, passing another traversal system (of compatible type), to attempt to find work from another worker thread's traversal system.

This interface allows Cain to have many implementations to choose between for different circumstances. Different graph traversal strategies may respond well to different heuristics, or different sized filters, so being able to swap between them makes Cain more versatile. This design philosophy is used throughout Cain, for things such as defining the cost function for a `Plan` or when to use the exhaustive or atom distance based pair generation systems. Or even which architecture to target since the whole pair generation system is accessed via the `PairGenFactory` which is used to produce a `PairGen` object that returns the goal-pairs one by one. So by implementing a new `PairGenFactory` and appropriate `Transformation` classes one can change the instruction-set Cain targets.

7.3 MULTI-THREADING

One of the reasons for choosing Java over Python was to make experimenting with multi-threading easier. Cain supports having multiple worker threads search the graph together, using a 'work stealing' method such that if a worker thread is ever out of work it will go through the other workers in order, attempting to steal a `WorkState` from them. Each worker thread has its own `TraversalSystem` and the first of n worker threads is seeded with the FG node and then all workers are started. We observe that after an initial steal, it is unusual for workers to run out of work locally, and so they work independently except that the `GoalCache` and minimum cost of plans found so far are shared and so each worker indirectly communicates their work with the others to reduce re-searching space that has been searched before.

Another useful feature of Java is the Streams API which is heavily used in the implementation of Exhaustive Pair-Generation. We produce a parallel stream of goals in the current goal-bag C, and treat of these as an upper for which we generate all the goal-pairs. `flatMap` is used to take the stream of uppers and produce a stream of goal-pairs, with each upper being used to created many goal-pairs. This allows Cain to make use of multi-threaded systems even when using only one graph traversal worker thread.

7.4 SCAMP5 VERIFICATION EMULATOR

Cain also includes a Scamp5 Emulator, It is limited to only the analogue functions of the Scamp5 device but simulates the functionality of hardware at the logical level of registers; this is to say that it implements an 'negate-then-add' function that is the building block of the `bus` functions that make up the macro analogue instructions in the Scamp5 API. This level of emulation allows use to easily detect register allocation issues since a `bus` instruction may have no repeated registers. And also implement a basic, but effective, noise model based on every operation, and the magnitude of the values that take part in the operation. To also act as a verification system the emulator stores the state of each register as a map of origin locations to values, thus allowing us to see for any given register where it's own value originally came from. The noise accumulated for each register is stored separately from the map so it can be added as wanted in reading the 'perfect' or 'noisy' value of any register. The emulator also allows us to input an image and apply the generated code to it and save the result.

In this chapter we have seen how Cain takes the algorithmic designs so far and how we have implemented them. Cain is now an open-source tool that anyone can run themselves and through use of the strategy patterns throughout its implementation it can be readily modified to suit various meet the architectural requirements of CPA other than that of Scamp5.

Chapter 8

Performance Evaluation

In this chapter we will look at how well Cain is able to produce implementations of filters for the Scamp5 architecture. We will investigate what effects the performance of Cain; in terms of the number of instruction (program length), the circuit depth. By looking at various filters we will build a picture of the expected performance of Cain in a variety of situations.

8.1 Test System and Methods

8.1.1 Test Computer

All performance evaluation is conducted on an Intel Core i7-7700HQ CPU @ 2.80GHz based system with 16GB of RAM, running Ubuntu 18; as well as Java 1.8 (Oracle) and Python 3.6 to run Cain and AUKE respectively.

8.1.2 Notes on Practical Issues

We found that during evaluation when processing large kernels with widths above 5, Cain could reach points where the JVM garbage collector was using significant CPU time and the JVM would reach maximum memory usage. At these times we endeavoured to increase the JVM memory limit within reason for the physical hardware. Other issues such as thermal throttling have largely been overlooked and assumed to have impacted the tests minimally. Cain was run continuously during various experiments where CPU temperatures quickly stabilised and so we assume that every filter was given a fair run regardless of the order of processing and the small differences in temperature.

8.2 Cain Performance

8.2.1 Nodes Explored

Hypothesis 1. *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 algorithm that is independent of physical compute performance. With this in mind we test Hypothesis 1 practically by constructing 100 samples of

randomly generated single kernel filters as in Equation 8.2.1. Running Cain as per the configuration in Table 8.2.1 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 Scamp5 heuristic disabled and replaced with a random sort. This allows us to compare Cains heuristics against an unaided benchmark.

$$\left\{ \frac{1}{2} \begin{bmatrix} u_1 & u_2 & u_3 \\ u_4 & u_5 & u_6 \\ u_7 & u_8 & u_9 \end{bmatrix} \right\} \tag{8.2.1}$$

Given $u_1..u_9$ are integers sampled uniformly from the range [0..8]

We found that Cain was unable to find any plan for any of the 100 sample filters without it's heuristics, principally demonstrating that effective heuristics are required in Cain for any tractable 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 Figure 8.2.1. 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, and try instructions less likely to be optimal later. This model fit the data well as we see short plans found quickly, and while improvements can be made, it is clear that they are found less often as the search continues.

Maximum Nodes to Explore:	20000
Maximum Search Time:	60s
Traversal Algorithm:	Stow-Optimised-Traversal
Cost Function:	Plan Length then Circuit Depth
Worker Threads:	1
Forced Depth Reduction:	1
Forced Cost Reduction:	0
Scamp5 Instructions:	All Available
Exhaustive Search Threshold:	10
Available Registers:	6

Table 8.2.1: Cain configuration used for testing Hypothesis 1

8.2.2 KERNEL SIZE

Hypothesis 2. *For a random single kernel filter of size $n \times n$, the length of the program found to produce the filter will be proportional to n^2.*

Hypothesis 2 follows the principal that the cost in generating code for filters is directly related to the number of positions in the kernels that have a non-zero value. Each one of them requiring its own amount of work. There are two issues that the arise; firstly common sub-expressions, and secondly the distance to the centre of the the atoms in the kernels. With

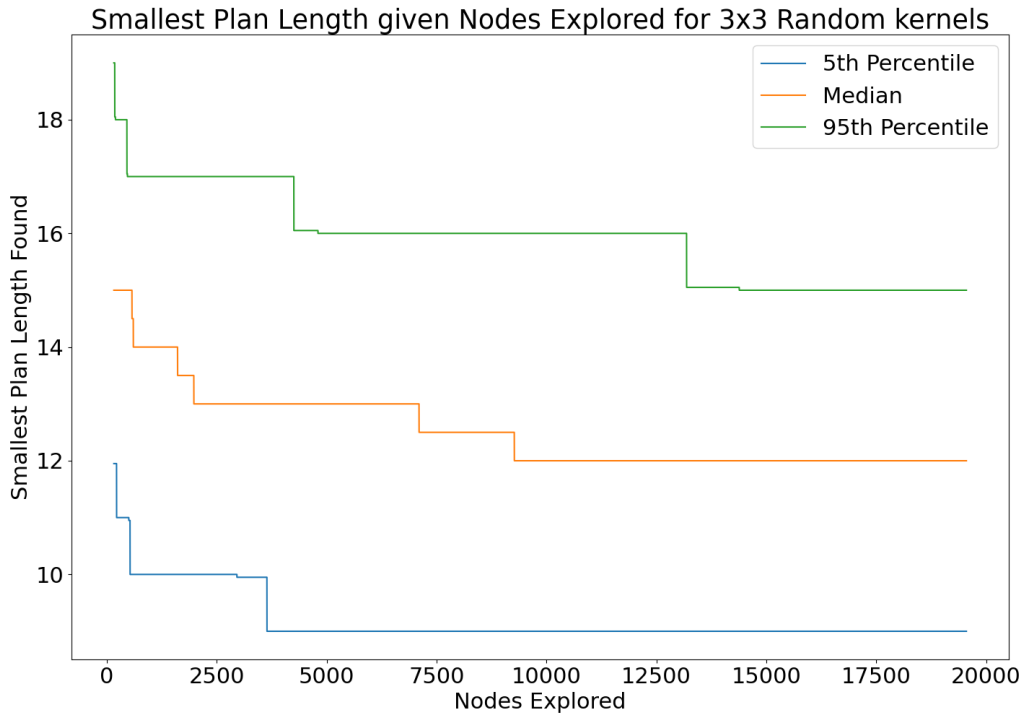


Figure 8.2.1: Graph showing the median number of instructions in the best plans found before n nodes have been explored by Cain. With 100 samples of randomly generating singular 3×3 kernel filters.

a larger kernel we should expect to find useful common sub expressions more often, and so be able to reduce the length of the program significantly, though with randomly generated of the style seen in Equation 8.2.1 with may be less common than hoped. Secondly, as the kernels get larger more work will need to be done to move the atoms, and the translations between common sub expressions will grow, as well as the translations to or from the centre. This should increase the length of program.

To test this hypothesis we construct 50 sample filters for each kernel width from 1 to 6, using the same method as in Equation 8.2.1 except changing the kernel size. We run each one on Cain with the configuration shown in Table 8.2.1 except we removed the 'Maximum Nodes to Explore' limit and increased the 'Maximum Search time' to 120 seconds. This is because we expected the larger kernels to take longer to reach a point of diminishing returns and wanted to test the 'best case' performance of Cain on the kernels within reason.

We note, in Table 8.2.2 that as the kernel size grew it became increasingly more common that the best plan was found later in the search, this is expected since with a larger kernel there are far more opportunities for the heuristic to be wrong, and the plans are in general longer. What we also see however is that maximum nodes explored before finding what becomes the shortest plan does not follow the time taken very well. We surmise that this is because the time taken to explore nodes in the larger kernels is significantly higher due to

Kernel Width	Best Plan Found After 60s	Latest Best Plan Found	Best Plan Found after 40000 Nodes	Maximum Nodes Before Best Plan Found
1	0	63ms	0	60
2	0	13932ms	0	15263
3	2	83480ms	4	59265
4	2	70491ms	15	69055
5	28	117868ms	23	90307
6	32	119613ms	24	86993

Table 8.2.2: The number of samples for which the shortest plan was found after 60s and after expanding 40000 nodes, and the maximums. 50 randomly generated sample filters were produced for each kernel width, with the search limited to 120 seconds per filter.

the number of options available when choosing how to decompose it.

We see in Figure 8.2.2 that the length of plans increases slightly more than the square of the kernel width. Though between kernel widths 2 and 4 the relationships appears proportional, the smaller and larger kernels show that the relationship does not hold. The covariance coefficient of k and $\sqrt{l}$ is 0.96. Our data only covers kernel widths for 1 to 6 so it is difficult to have certainty, but it is reasonable not to expect such a simple relationship once the size of kernel causes us to reach other limitations such as the number of registers. We conclude that the relationship between kernel width k and the shortest plan Cain can find under the conditions l is $l > k^2$ by a small margin of error.

8.2.3 SIMULTANEOUS KERNEL FILTERS

One of the significant features we claim we have achieved in Cain is to efficiently generate code for filters with multiple kernels, and do this simultaneously such that shared common sub-expressions can be reused. We propose Hypothesis 3 on the basis that using shared common sub-goals in multiple kernels should offer performance improvements. Due to the way Cain searches for plans, we know that given enough time it will find a solution that simply computes the individual kernels separately, assuming no lower cost alternative is found first. For this reason we aim to demonstrate that even in less time than would be required to find that simple solution, Cain will find solutions that are more effective.

Hypothesis 3. *As the number of kernels in a filter increases, the length of the implementation will grow slower than linearly as common-sub-expressions are found.*

To test this, we again generate kernels using the using the method in Equation 8.2.1. For kernel counts from 1 to 3 we generated 50 filters each and test them all using the configuration in Table 8.2.1 except again we remove the maximum nodes explored constraint. We plot the results in Figure 8.2.3 and see that the results appear very close to linear, suggesting that common sub expressions are not effectively being taken advantage of.

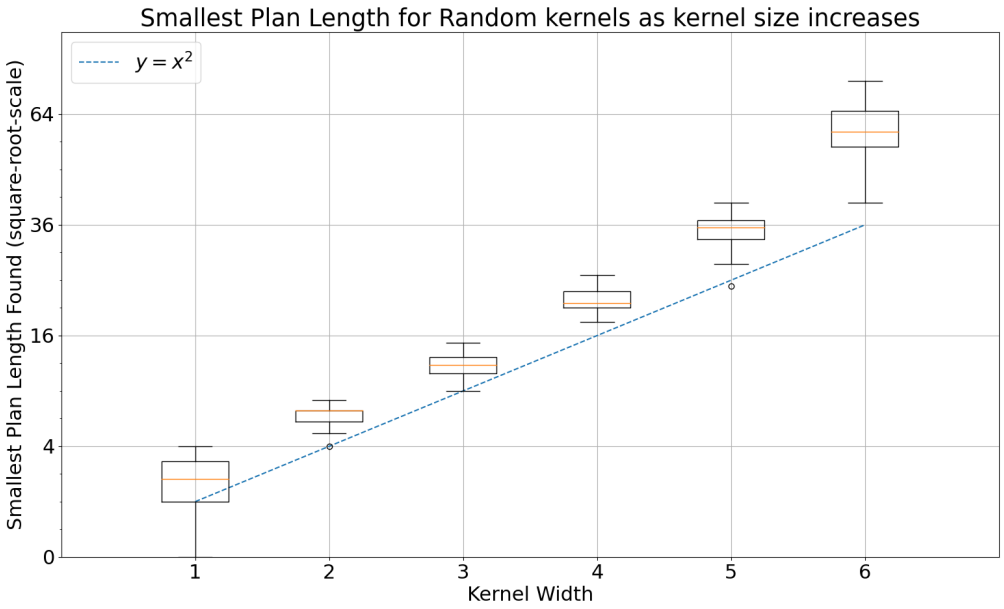


Figure 8.2.2: Graph showing the number of instructions in the shortest plans found plotted over kernel width. 50 randomly generated single kernel filters were produced and tested per kernel width.

Next we look at more realistic scenario, we produce 10 filters each with 2 kernels, randomly generated as before. Then we use Cain to generate code for the first kernel, making sure to preserve the input register, and then the second, ensuring we preserve the register for the first result. We can then compare this to the program Cain generates when given both kernels at the same time to compile. We repeat this for 3 kernel and 4 kernel filters. Cain is given 60 seconds to search for each individual kernel, and 60 seconds times the number of kernels for each simultaneous kernel filter. Figure 8.2.4 shows the result, and from this we conclude that Cain produces shorter code very inconsistently such that there appears to be little benefit.

We propose two main explanations for why Cain is unable to reduce the program length. Firstly, the heuristics used to determine the best instructions to search penalise large goals greatly, such that Cain has an incentive to quickly split large goals up. This is an effective strategy when there are enough registers, such that the other factors in the heuristic are able to make a meaningful impact on the decisions made. The heuristics were primarily designed by trial and error on various well known filters, and this neglected more than 2 simultaneous kernels where we see these problems begin to occur. Using these dense random kernels also means that while there are many common patterns, they do not each contribute as much to the final goals as in many of the analytically designed kernels. The second reason is simply that the scale of the problem is expanded hugely by looking at multiple kernels at once. The search graph is made much larger and will contain many more redundant paths to nodes already seen. This inevitably reduces the rate of useful work that is achieved.

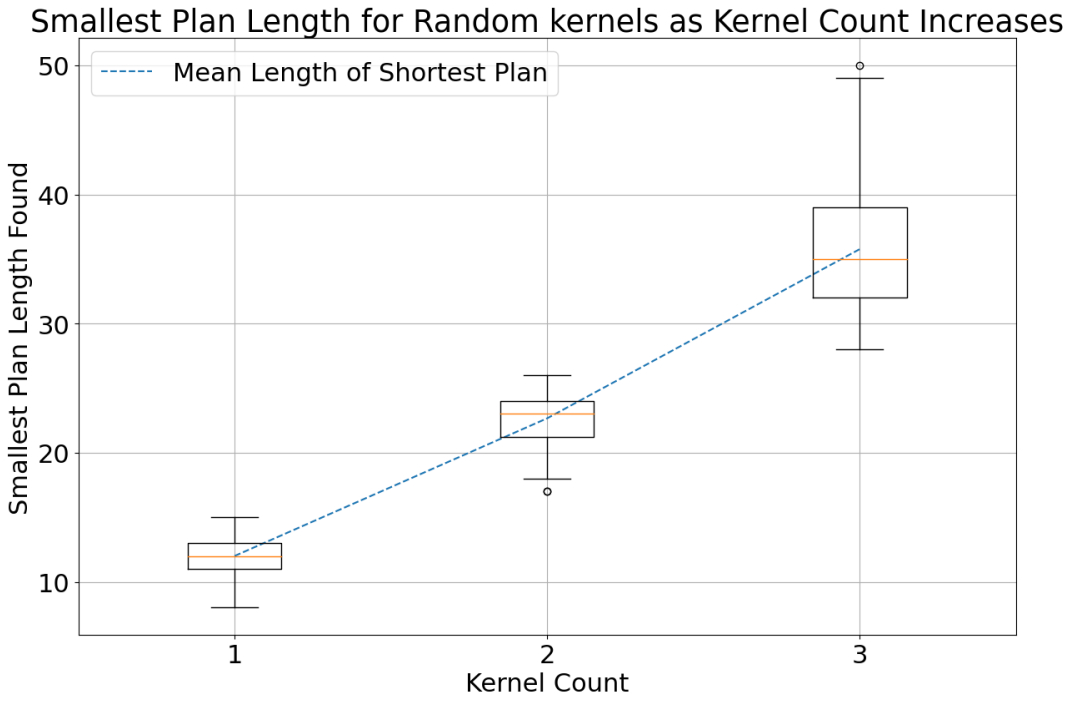


Figure 8.2.3: Graph showing the number of instructions in the shortest programs found by Cain for filters with 1, 2, and 3 random 3×3 kernels. 50 samples were produced for each kernel count.

To test this explanation we diverge from what is possible on Scamp5 hardware and allow Cain to use 24 registers instead of just 6. This should eliminate problems with register usage, allowing the heuristics to split the problem up lots before running out of space. We then re-run the experiment, testing both 6 and 24 registers.

Figure 8.2.5 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 also see that there is little improvement to the individually compiled kernels when more registers are available for 2 and 3 kernel filters. With 4 kernel filters, 2 of the individually compiled kernels have only 3 registers available. Here we see that this has impacted the length of the implementation significantly since when there are 24 total registers those 2 kernels will have 21 registers, and we see a significant reduction in the sum of program lengths for these kernels.

8.2.4 SPARSITY

Hypothesis 4. *The lengths of programs decrease proportionally with the sparsity of the kernel.*

We define sparsity as the proportion of zeros in a kernel. As sparsity increases a kernel has fewer non-zero values, and so fewer atoms. We now look at determining what relationship

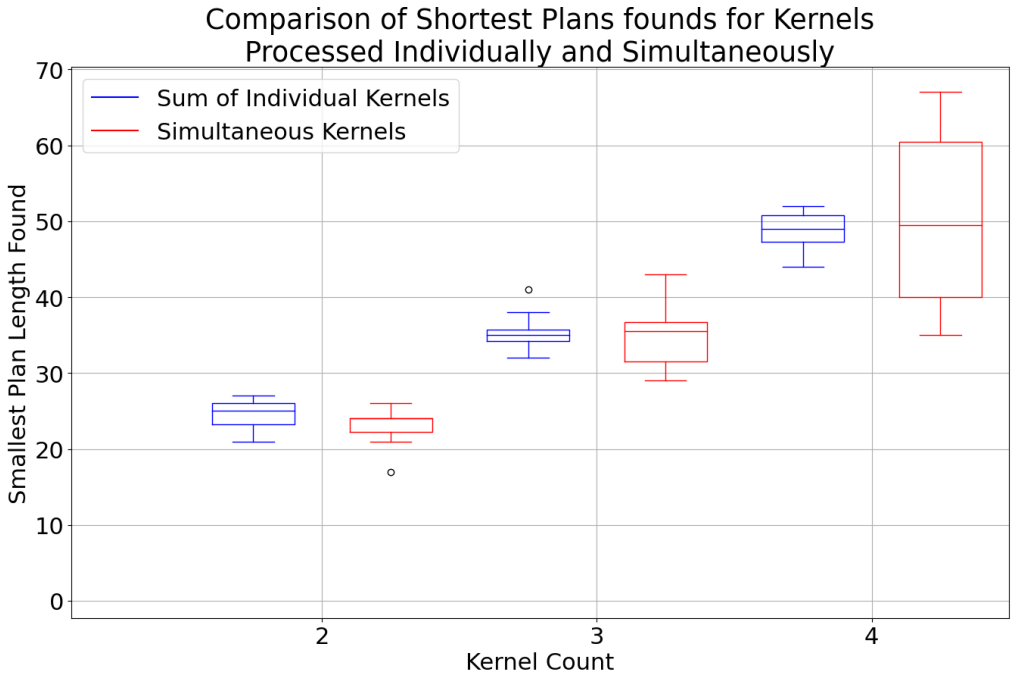


Figure 8.2.4: Graph comparing the sum of shortest code lengths found for kernels compiled individually, against the same kernels compiled simultaneously as one filter. For each kernel count 10 filters are produced, consisting of their respective number of randomly generated 3×3 kernels.

this takes with regards to program length. We expect that the length of the program will be proportional to the number of non-zero coefficients since this appears to be the primary factor we have thus far observed. To test Hypothesis 4 we construct 25 sample kernels for every *sparsity* from 0 to 1 in steps of 0.1 as shown in Equation 8.2.2.

$$\left\{ \frac{1}{2} \begin{bmatrix} c_1 & c_2 & c_3 & c_4 & c_5 \\ c_6 & c_7 & c_8 & c_9 & c_{10} \\ c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\ c_{16} & c_{17} & c_{18} & c_{19} & c_{20} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} \end{bmatrix} \right\} \tag{8.2.2}$$

Given c_n is 0 with probability *sparsity* and is otherwise sampled uniformly from the range of integers [0..8]

We also included 25 samples of 'inflated filters' which have the form shown in Equation 8.2.3. This gives each 'inflated filter' a sparsity of $\frac{16}{25}$. These methods of generating kernels means the true sparsity can could be anywhere from 0 to 1. While this does affect the sparsity of the kernels, it allows for simpler comparison with results for other experiments and will still sarcastically represent the relationship we aim to show, or otherwise force us to reject it.

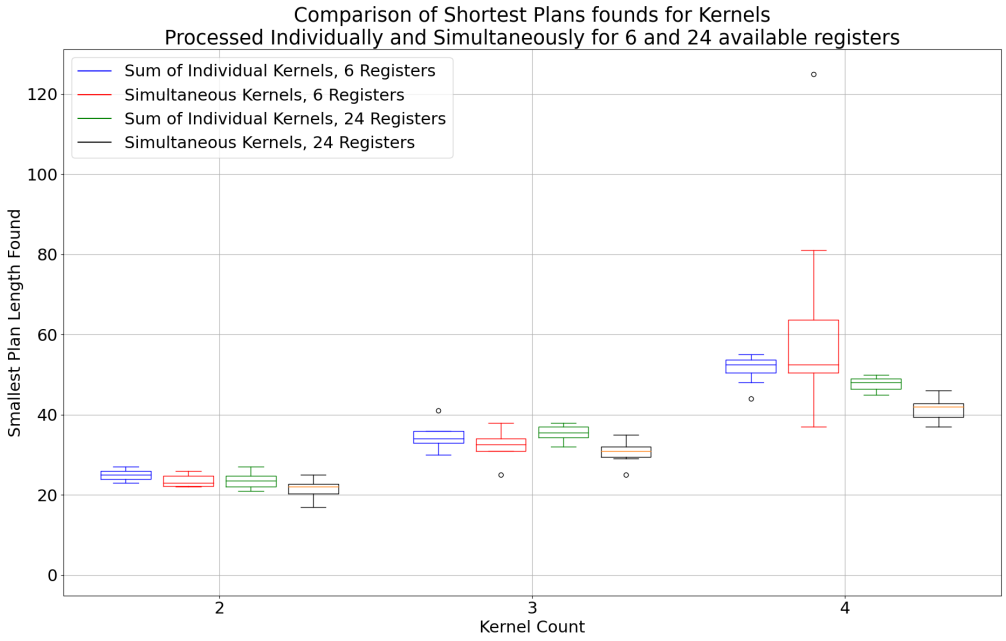


Figure 8.2.5: Graph comparing the sum of shortest code lengths found for kernels compiled individually, against the same kernels compiled simultaneously as one filter; when using 6 registers and when using 24. For each kernel count: 10 filters are produced, consisting of their respective number of randomly generated 3×3 kernels.

$$\left\{ \frac{1}{2} \begin{bmatrix} c_1 & 0 & c_2 & 0 & c_3 \\ 0 & 0 & 0 & 0 & 0 \\ c_4 & 0 & c_5 & 0 & c_6 \\ 0 & 0 & 0 & 0 & 0 \\ c_7 & 0 & c_8 & 0 & c_9 \end{bmatrix} \right\} \tag{8.2.3}$$

Given c_n is sampled uniformly from the range of integers [0..8]

Each filter is compiled and the shortest plan aggregated and plotted in Figure 8.2.6. We see that our hypothesis is broadly correct with a co-variance coefficient between stated sparsity and program length of -0.94. Interestingly we see that the structured sparsity of the 'inflated filters' reduce both the spread of results and the median, though not by much.

8.2.5 Noise, Circuit Depth and the Pareto Frontier

Program length is not the only measure of success for the implementation of a kernel. Given we are generating code for an analogue processor we must consider noise and how this effects the result of any computation. To do this we should first define a model for our noise, and while this has been researched for Scamp5 before [5], they produce a model for copying

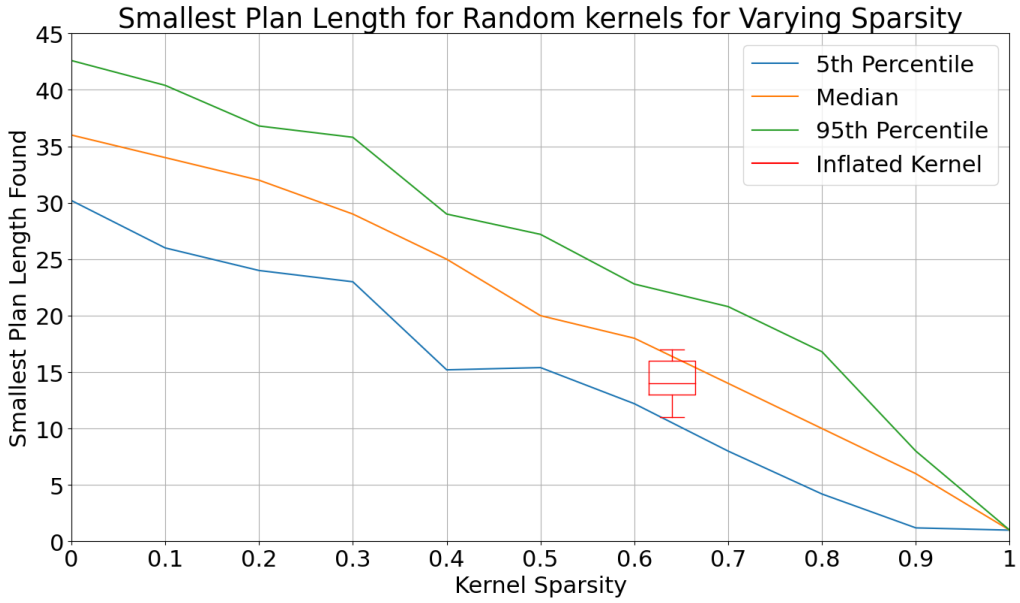


Figure 8.2.6: Graph plotting shortest plan found over kernel sparsity, with inflated kernels for comparison. 25 randomly generated 5×5 kernels were produced for each sparsity tested. We also tested 25 randomly generated inflated kernels, to compare the structured sparsity with random sparsity.

from register B to A (8.2.4) which is dependant on an ϵ and δ that represent the random error at that time and a fixed noise pattern of the PE array. We simplify this model to serve our purpose by separating the noise and the idealised values stored in a register. Our model, as applied to every Bus instruction, is given in 8.2.5.

$$A_{i,j} \Longleftarrow B_{i,j} + k_1 B_{i,j} + k_2 + \epsilon_{i,j}(t) + \delta_{i,j} \tag{8.2.4}$$

$$\begin{aligned} A &\Longleftarrow -B \\ A_e &\Longleftarrow B_e + k_1(B_e + |B|) + k_2 \end{aligned} \tag{8.2.5}$$

This noise model allows us to compare programs in more abstract terms - without concerning where on the device's PE array a calculation happens, and in a repeatable way. It allows us to consider A_e to be the total sum of noise and error, rather than a precise value to add to the final idealised result. For our tests $k_1 = 0.005$ and $k_2 = 0$. k_2 is zero because it 'tends to be unimportant'[5]. These values need not directly relate to what hardware produce a real world output and instead need only to produce a comparable indication of noise intensity. The resultant numbers therefore cannot be directly applied to Scamp5, but they should enable us to test Hypothesis 5.

Hypothesis 5. *Circuit Depth is a better indicator of noise than program Length.*

Hypothesis 5 is based on the idea that noise compounds together as successive operations are performed on accumulating data. We can see clearly that when adding several

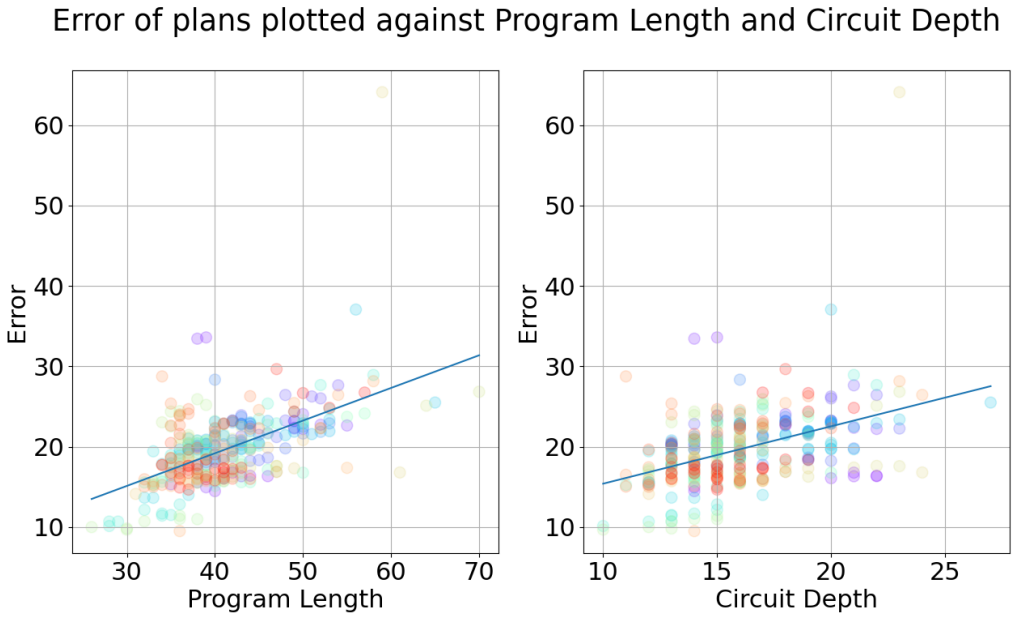


Figure 8.2.7: Graphs of noise as a result of running implementations of random 5×5 kernels on our Scamp5 Emulator, plotted against the Program Length and Circuit Depth, with best fit lines. 322 samples were produced by generating 50 kernels and collecting every plan Cain produces as it searches, and using our Scamp5-Emulator to predict an error value.

values together the compound noise is greater if one register is used to accumulate the sum, making the depth $n-1$ for n values to be added, than if the values were summed in a binary tree fashion, making the depth $\log_2(n)$.

To test this hypothesis we construct 50 random filters as before (8.2.1), with kernel widths of 5. We then use Cain to produce plans under the configuration seen in Table 8.2.1 except without the limit on nodes to explore, and with a shorter time limit of 15 seconds per filter. Every plan produced (not just the shortest) is emulated in our Scamp5-Emulator (Section 7.4) using the noise model in Equation 8.2.5. We then extract just the noise value A_e of the output register A to produce out data points.

In Figure 8.2.7 we can see that both program length and circuit depth broadly correlate with error, under our noise model. The co-variance coefficient of length with error is 0.55 while for circuit depth with error it is 0.44. This shows that in general, under our noise model, length is in fact a better indicator of error.

When we consider this problem at the filter by filter level, we find that the first plan found that has minimal error of all that were found also has minimal length 23 times out of 50 samples, whereas that plan had minimal circuit depth 16 times. Conversely we have that the earliest plan of minimal length also has minimal noise 18 times, but the earliest found plan of minimal circuit depth was only also a plan with minimal noise 16 times. This all shows that we do not have a simple and effective indication of noise that can be applied during the search process.

If our main goals in generating code for Scamp5 are to reduce noise and reduce program

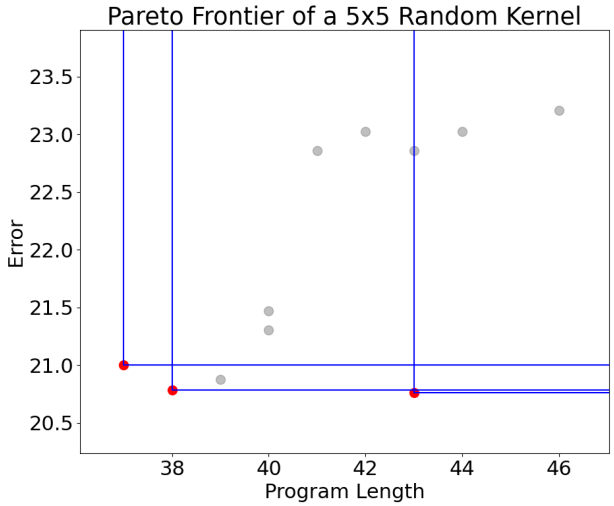


Figure 8.2.8: The Pareto Frontier of plans generated for a random 5×5 filter, minimising program length and error under the noise model in Equation 8.2.5

length (to allow for higher frame rates) we can construct a Pareto Frontier graph that shows which of the generated plans fulfil these objectives. The principal is that the frontier consists of those plans that are not beaten in both program length and error. Figure 8.2.8 shows such a graph for one of the filters in this experiment, showing that there exists a trade-off between program length and noise and one of 3 different implementations of the filter could be used to suit the needs of the user.

8.3 Direct Performance Comparison

To test the performance of Cain, the only comparable framework is AUKE, so both code generators were given: 7 well known kernels; 2 filters used for handwriting recognition[14]; and 8 randomly generate kernels. AUKE was tested using the default parameters found in its demo script, apart from that the number of registers was increased from 3 to 6. This was done to ensure that the more complex filters would be possible while remaining inside the physical capabilities of the SCAMP5 system. When testing we found that for many filters AUKE performed better with fewer registers, so we tested all kernels with 6 and 3 registers available and if the program produced is shorted given 3 registers we show the result in angle brackets. 9 tests are conducted where multiple kernels are to be produced simultaneously, AUKE was not designed to do this so this procedure was followed to stay consistent:

- All filters are compiled independently on AUKE given 6 and 3 registers
- The filters are ordered by highest cost to lowest
- The filters are compiled in order with the appropriate number of registers available given some would be used to store the initial value and outputs of other kernels.

This method is a fair way to assess the abilities of AUKE for multiple simultaneous kernels in a systematic way, though it is clear that with a little human intervention results could be improved by, for example, sharing division operations. While equally and more effective ways to decide which kernel to have AUKE compile first may exist, that is not the focus of the performance evaluation and this method is sufficient to grasp the differences in the produced code. This method also means, however, that AUKE is given more time in total than Cain to compile simultaneous kernel filters. This comparison is not designed to demonstrate if Cain or AUKE finds implementations faster but rather, within what should be ample time, which generates shorter programs. Debrunner claims in his performance evaluation of AUKE[7] that with the heuristics enabled minimal code is produced in the order of milliseconds so 60 seconds is plenty of time to converge to solution that an effective representation of AUKE's best effort. We have seen that Cain also exhibits a point of diminishing returns and we expect 60 seconds to be a reasonable time with which to limit Cain and achieve results that adequately represent Cains performance in a real use case setting.

The 7 well known kernels, used to make 9 filters, all have significant symmetries and so it should be possible to find repeated sub-expressions within the filters that allow for substantial performance improvement over a naive algorithm. They are comprised of 3 categories of filter: Sobel edge detection filters, Box filters, and approximated Gaussian blur filters. 3 of the 9 filters are made up of 2 kernels, to test how the common sub-expressions in each category allows for efficient computation of the whole.

The Analog2Net and MaxPooled filters are extracted directly from work done to show how convolutional neural networks might be used on Scamp5 architecture[14]. For their tests AUKE was used to generate the code, which was extended with the use of digital registers to store intermediary values allowing for more registers to be available. We include these as tangible examples of convolutional filters that were designed and trained specifically for Scamp5.

The 8 randomly generated kernels are split into 2 groups of 6 filters, all these filters use 3 by 3 kernels with coefficients made of integer multiples of eighths. The first group is generated similarly to as we have seen in Equation 8.2.1 but dividing by 8 instead of 2. The second group differs by picking a number in the range $[1,8]$ with 50% probability and 0 otherwise, giving us sparse kernels. For each group 4 randomised kernels are created, then the remaining two filters are produced by combining the kernels in pairs. This allows us to exemplify how much Cain is able exploit any potential sub-expressions in the filters. The random kernels are designed to simulate an expected performance for any other small convolutional neural network that might be used on Scamp5.

In Table 8.3.1 we see the 5 run configurations of the Cain algorithm as well as the AUKE run configuration. This allows us to compare what features of Cain affect the performance of results compared to the AUKE baseline.

Table 8.3.1: Kernels Tested in AUKE and Cain

Name	Approximated Filter	AUKE	Cain					
		60	60	60	60	60	60	Seconds
		1	4	4	4	4	4	Threads
		DFS	SOT	SOT	DFS	HOS	SOT	Traversal
		6*	6	6	6	6	6	Registers
		Basic	All	Basic	All	All	All	Instructions
		-	10	10	10	10	0	Threshold
Vertical Sobel	$\left\{\begin{bmatrix}1&0&-1\\2&0&-2\\1&0&-1\end{bmatrix}\right\}$	7	5	7	5	5	5	
Vertical and Horizontal Sobel	$\left\{\begin{bmatrix}1&0&-1\\2&0&-2\\1&0&-1\end{bmatrix},\begin{bmatrix}1&2&1\\0&0&0\\-1&-2&-1\end{bmatrix}\right\}$	(7 + 7)	9	13	10	9	9	
2x2 Box	$\left\{\begin{bmatrix}0&1&1\\0&1&1\\0&0&0\end{bmatrix}\right\}$	4	3	4	3	3	4	
3x3 Box	$\left\{\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}\right\}$	8	6	9	6	6	6	
5x5 Box	$\left\{\begin{bmatrix}1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\end{bmatrix}\right\}$	15⟨14⟩	8	15	8	8	8	
5x5 and 3x3 Box	$\left\{\begin{bmatrix}0&0&0&0&0\\0&1&1&1&0\\0&1&1&1&0\\0&1&1&1&0\\0&0&0&0&0\end{bmatrix},\begin{bmatrix}1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\end{bmatrix}\right\}$	(8 + ⟨14⟩)	13	24	15	15	11	
3x3 Gauss	$\left\{\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}\right\}$	12	10	12	10	10	10	
5x5 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⟨21⟩	19	25	23	20	19	
5x5 and 3x3 Gauss	$\left\{\frac{1}{64}\begin{bmatrix}0&1&2&1&0\\1&4&6&4&1\\2&6&10&6&2\\1&4&6&4&1\\0&1&2&1&0\end{bmatrix},\frac{1}{64}\begin{bmatrix}0&0&0&0&0\\0&4&8&4&0\\0&8&16&8&0\\0&4&8&4&0\\0&0&0&0&0\end{bmatrix}\right\}$	(⟨21⟩ + 12)	26	36	41	26	26	
AnalogNet2	$\left\{\frac{1}{4}\begin{bmatrix}0&0&0\\-3&1&0\\-3&0&2\end{bmatrix},\frac{1}{4}\begin{bmatrix}-4&-1&1\\-1&2&0\\1&1&0\end{bmatrix},\frac{1}{4}\begin{bmatrix}-1&2&0\\-1&1&-3\\0&-3&0\end{bmatrix}\right\}$	$(12^\dagger + 21^\dagger + 16^\dagger)$	21	30	27	21	20	
MaxPooled	$\left\{\frac{1}{4}\begin{bmatrix}1&1&0\\0&0&0\\1&0&0\end{bmatrix},\frac{1}{4}\begin{bmatrix}1&0&-2\\0&-2&0\\-1&-2&1\end{bmatrix},\frac{1}{4}\begin{bmatrix}0&1&1\\-2&-2&-1\\0&0&0\end{bmatrix},\frac{1}{4}\begin{bmatrix}0&-1&-1\\0&0&0\\1&0&1\end{bmatrix}\right\}$	$(8 + 18^\dagger + 10^\dagger + 13^\dagger)$	19	27	22	19	18	
Random0 3x3 [0-8] 0%	$\left\{\frac{1}{8}\begin{bmatrix}1&8&1\\8&7&8\\7&2&1\end{bmatrix}\right\}$	$30^\dagger\langle 24^\dagger\rangle$	15	21	16	15	13	
Random1 3x3 [0-8] 0%	$\left\{\frac{1}{8}\begin{bmatrix}2&2&5\\4&6&5\\2&5&0\end{bmatrix}\right\}$	35⟨20⟩	11	15	14	11	11	
Random2 3x3 [0-8] 0%	$\left\{\frac{1}{8}\begin{bmatrix}7&0&3\\3&8&2\\0&4&8\end{bmatrix}\right\}$	$37\langle 33^\dagger\rangle$	12	19	17	13	12	

Random3 3x3 [0-8] 0%	$\left\{\frac{1}{8}\begin{bmatrix}6&1&4\\0&7&6\\7&7&3\end{bmatrix}\right\}$	$34^{\dagger}$	13	22	16	16	13	
Random0&1 3x3 [0-8] 0%	$\left\{\frac{1}{8}\begin{bmatrix}1&8&1\\8&7&8\\7&2&1\end{bmatrix},\frac{1}{8}\begin{bmatrix}2&2&5\\4&6&5\\2&5&0\end{bmatrix}\right\}$	$(\langle 24^{\dagger}\rangle + \langle 30\rangle)$	24	29	38	23	21	
Random2&3 3x3 [0-8] 0%	$\left\{\frac{1}{8}\begin{bmatrix}7&0&3\\3&8&2\\0&4&8\end{bmatrix},\frac{1}{8}\begin{bmatrix}6&1&4\\0&7&6\\7&7&3\end{bmatrix}\right\}$	$(\langle 33^{\dagger}\rangle + 34^{\dagger})$	21	39	30	23	21	
Random0 3x3 [1-8] 50%	$\left\{\frac{1}{8}\begin{bmatrix}3&0&0\\0&4&0\\0&0&6\end{bmatrix}\right\}$	14	7	10	7	7	7	
Random1 3x3 [1-8] 50%	$\left\{\frac{1}{8}\begin{bmatrix}0&0&8\\0&4&3\\0&0&4\end{bmatrix}\right\}$	$11^{\dagger}$	7	11	7	8	7	
Random2 3x3 [1-8] 50%	$\left\{\frac{1}{8}\begin{bmatrix}0&6&8\\0&7&0\\0&0&8\end{bmatrix}\right\}$	$23^{\dagger}\langle 16^{\dagger}\rangle$	9	12	9	9	9	
Random3 3x3 [1-8] 50%	$\left\{\frac{1}{8}\begin{bmatrix}5&0&0\\0&2&0\\7&0&6\end{bmatrix}\right\}$	$20^{\dagger}$	9	15	9	9	9	
Random0&1 3x3 [1-8] 50%	$\left\{\frac{1}{8}\begin{bmatrix}3&0&0\\0&4&0\\0&0&6\end{bmatrix},\frac{1}{8}\begin{bmatrix}0&0&8\\0&4&3\\0&0&4\end{bmatrix}\right\}$	$(15 + 13^{\dagger})$	11	18	12	11	11	
Random2&3 3x3 [1-8] 50%	$\left\{\frac{1}{8}\begin{bmatrix}0&6&8\\0&7&0\\0&0&8\end{bmatrix},\frac{1}{8}\begin{bmatrix}5&0&0\\0&2&0\\7&0&6\end{bmatrix}\right\}$	$(\langle 16^{\dagger}\rangle + 22^{\dagger})$	15	23	17	15	14	

We can see that Cain performs best in these examples using the Stow-Optimised-Traversal algorithm (Algorithm 2) with the Heir-Ordered-Search beating it only once (Random0&1 3x3 [0-8] 0%), and Depth-First-Search performing the worst. The threshold value in Cain refers to the maximum atom count of goal for which Cain used its exhaustive pair generation function, using the Atom Distance-Based method if any of the goals in the current goal-bag are too large. We see that with the threshold at 0, so Cain will only use the Atom Distance-Based method, results are frequently better than when it is set to 10. We believe this is because this method by design produces far fewer pairs, far faster. This allows Cain to cover a broader search graph though it notably misses opportunities for optimisation seen for the 2x2 box filter since it handles the use of the `add`(a,b,c,d) instruction differently.

When we constrict Cain to using the same instructions as AUKE as available, we see that AUKE produces shorter code for 5 filters. All of these filters were based off either Box filters or Gaussian filters. 3 of these time were as a result of using less registers in AUKE which was not tested in Cain. We propose that AUKE was optimised more for these well known kernels than random kernels and Cain's heuristics are designed with the full instruction set in mind, so are not necessarily as well suited to having these fewer options available.

*Some kernels were also run with 3 available registers which reduced the program length, we report these values in angle brackets.

†The code produced uses instructions with invalid register usage (see Table 6.1.1). The program would have to be extended to allow them to work, assuming registers are available.

We have see how Cain performs under varying conditions, and sought to provide stochastic guidance to estimate the length of programs in realistic conditions. This knowledge allows us to not only gain insight into how Cain might be improved but also how the Scamp5 architecture might be developed to improve performance.

Chapter 9

Conclusions And Further Work

In this thesis we have seen the challenges faced when producing code for Cellular Processor Arrays and how in Cain we have tried to address these challenges to generate efficient and effective code. We have looked at how other comparable systems are designed from those with similar objectives such as AUKE[8], and others with different aims but common hurdles such as Linnea[1].

9.1 Conclusions

We have presented Cain and described how it functions, combining the core principles of atom approximation and heuristic based graph searching. We have demonstrated that by allowing for more general operations, and searching a more exhaustive graph of possible states, Cain is able to find more efficient implementations of kernels for the Scamp5 architecture.

We have looked at various graph traversal algorithms and presented a new, novel algorithm that is uniquely qualified for the graph search problem faced in Cain.

Cellular processor arrays lend themselves very naturally to convolutional filters, especially in situations where large quantities of data need to be processed in very short windows of time, where the data transfer rate into the processor array can be entirely parallel but the data transfer out of the array is serialised. High speed image recognition is a perfect example of this[14], and especially so in the context of low power devices used in mobiles, robots and drones. With the aim of Cain being to shorten the length of programs that implement convolutional filters and reduce the error in the case of analogue processors, Cain is able to help reduce the power usage of the vision systems on mobile devices, allowing for battery savings and/or longer up time for robots and UAVs.

Our overarching hypothesis, the reason to compile kernels simultaneously, is that performance improvements should be possible if we can exploit common sub expressions in convolutional filter kernels on CPAs. We have shown that this is true but, for Scamp5 at least, we have seen that the number of available registers is often a limiting factor for even small kernels.

9.2 Further Work

This thesis has produced a framework, Cain, that allows for significant further expansion. The research of cellular processor arrays is not new but recent times have brought about

significant process for low power devices such as Scamp5. There is much more to look into that expands the scope of search-ability and how we use heuristics in searching.

9.2.1 Machine Learning Driven Compiler Optimisation

Currently Cain uses a hand crafted local heuristic, that we have shown is effective but far from optimal, and often inefficient. Machine learning has been applied to many areas of compiler optimisation, but one area that could be especially promising is to apply it to the heuristics problem in Cain. One of the main features of the heuristic we have presented and used is to find patterns of atoms that are repeated at various translations. This could be an interesting place to use convolutional neural networks to find common patterns and then attention mechanisms to generate an estimated value of said repeated patterns. This could prove to allow for global heuristics in Cain to be effective, and so allow for far more effective searching.

9.2.2 Computed Noise as a Cost Function

In this thesis, Cain was developed and tested separately from any Scamp5 hardware, with very limited testing of code once Cain had already been mostly completed. This meant that developing the noise model could not be a focus of the project and so it is an afterthought enabled by emulating the generated code after searching has completed. If a model for noise could be developed that could be deduced in reverse, and effectively model the noise and error patterns of Scamp5 hardware, then we could reject the notion of length and circuit depth as a proxy for noise and error. The Scamp5 analogue hardware has inherent bias in its calculations which, if addressed by a more adept error model, could be installed as part of the cost function used in Cain and replace the assumption of reduced length and depth being always beneficial.

9.2.3 Noise and Error Modelling in Scamp5

We have developed a Scamp5 Emulator that is capable of implementing the noise model we put forward in Equation 8.2.5. But we believe that this model is unsatisfactory for real use as a means of quantifying expected noise and error in Scamp5 analogue programs. Developing this model, in a way that is repeatable and based on recorded data, would allow for a more in depth understanding of how to reduce error in programs generated. The model presented by Carey *et al.*[5] approaches the model from an electrical engineering perspective, adding random noise at every register transfer. If this model could be developed to produce an expected error away from the idealised value over many instructions, accounting for register decay and other such physical phenomena, then it could allow for far superior testing of programs without needing to perform practical tests on the hardware.

9.2.4 ADDRESSABLE MEMORY

Current Cain stores the entire state of a PE as a goal-bag. This definition is restrictive and doesn't account for the available CPA architectures that have addressable memory and complex PEs. Implementing better register allocation would allow for register spilling. This could also be extended to allow for 3 dimensional filters to be computed on 2 dimensional CPAs where there are clearly too few registers to hold all the required values.

Scamp5 has digital registers that have been almost entirely overlooked in this thesis as they do not so readily allow us to perform the arithmetic operations we would like since they are binary. Methods that have been researched already include use of them as storage for values [14], and use of them for ternary weight kernels[2]. We propose that Cain could be extend to integrate the use of binary registers as extra memory potentially allowing for more efficient use of existing hardware.

9.2.5 LOWER LEVEL INSTRUCTION

Cain is thus far limited to those Analogue Macro instructions provided by the current Scamp5 API. While these allow for interesting features that combine translation with addition and subtraction, if the definition of the processor state was extended to include the *News* register specifically then the use of the binary instructions directly on Scamp5 may allow for more efficient code to be generated. For example:

$$\begin{array}{l}\texttt{mov2x(A,C,east,north)}\\ \texttt{mov2x(B,C,east,south)}\end{array} \Longrightarrow \begin{array}{l}\texttt{bus(XW,C)}\\ \texttt{bus(A,XN)}\\ \texttt{bus(XW,C)}\\ \texttt{bus(B,XS)}\end{array} = \begin{array}{l}\texttt{bus(XW,C)}\\ \texttt{bus(A,XN)}\\ \texttt{bus(B,XS)}\end{array}$$

In this trivial example we see that one bus operation is repeated where it doesn't need to be, so by searching for program code at `bus` operations level, each of which correspond to a single binary instruction, we may be able to further reduce the cost of kernel implementations. This simple case could be solved by a simple optimiser after the current compilation, but we suspect more effective optimisations may be possible if these lower level operations could be searched directly.

This research area can quickly become very focused on the Scamp5 architecture, but benefits to other lower power CPA architectures may be possible if we are able to incorporate an understanding of special register functions into the search, in a way this is more widely applicable to other architectures.

9.2.6 NON-LINEAR OPERATORS

Cain specifically compiles filters made of linear combinations of inputs. This property allows Cain to manipulate atoms, combining them and dividing goals to suit its needs and find valid convolutional kernel implementations. This does however limit the use of Cain to such

linear functions; meaning for example that layers of a convolutional neural network must be computed individually to allow for a non-linear activation function between the applications of the linear kernels.

In Cain there is not an inherent reason that atoms should relate to units of coefficients of inputs values at a particular coordinate. This definition only serves to provide a simple platform for the definition of instructions and to provide a reasonable limit on the ways to break down a goal. Another definition of atoms could easily have each *atom* hold a coefficient of its own rather than be defined as a single unit. In this new system a goal should not contain multiple *atoms* from the same coordinates, but instead contain one *atom* with a value equal to the count of identical atoms that would be found in the current system. This would allow for the *generatePairs* function to make a decision between accuracy and cost as the search progresses, perhaps allowing for more accuracy for parts of the kernel that are considered important, and less accuracy for other parts. Since atoms only need to be understood by the *generatePairs* function that is implemented, we see that in reality the definition of an atom can be as malleable as required to perform the intended functions. At this level we could break down even the notion of position that atoms have; we can choose instead to define a special type of atom, *sAtom*. This *sAtom* could be given the feature, by means of the pair generation function used, that if conditions are right in a goal, then it can be produced by some non-linear function applied to some other goals that become the lower goals of the pair. Two *sAtoms* may be able to be combined or divided to find more efficient deconstructions.

$$
\begin{aligned}
&\text{Given:}\\
&\qquad \alpha = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}\\
&\qquad \beta = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}\\
&\text{U:}[sAtom_{\sin(\alpha)}] \leftarrow_{\sin} \text{L:} \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}\\
&\text{U:}[sAtom_{\cos(\alpha)\cos(\beta)}] \leftarrow_{\times} \text{Ls:}[sAtom_{\cos(\alpha)}], [sAtom_{\cos(\beta)}]\\
&\qquad G = \left[a \forall\, a \in \alpha, sAtom_{\cos(\alpha)\cos(\beta)}, sAtom_{\sin(\alpha)\sin(\beta)} \right]\\
&\qquad \{G\} \Leftarrow_{+} \left\{ \alpha, [sAtom_{\cos(\alpha)\cos(\beta)}, sAtom_{\sin(\alpha)\sin(\beta)}] \right\} = C\\
&\text{Then:}\\
&\qquad C \Leftarrow_{+} \left\{ \alpha, [sAtom_{\cos(\alpha)\cos(\beta)}], [sAtom_{\sin(\alpha)\sin(\beta)}] \right\}\\
&\qquad\quad \Leftarrow_{\times} \left\{ \alpha, [sAtom_{\cos(\alpha)}], [sAtom_{\cos(\beta)}], [sAtom_{\sin(\alpha)\sin(\beta)}] \right\}\\
&\qquad\quad \Leftarrow_{\times} \left\{ \alpha, [sAtom_{\cos(\alpha)}], [sAtom_{\cos(\beta)}], [sAtom_{\sin(\alpha)}], [sAtom_{\sin(\beta)}] \right\}\\
&\qquad\quad \Leftarrow^{*}_{\sin} \left\{ \alpha, [sAtom_{\cos(\alpha)}], [sAtom_{\cos(\beta)}], \beta \right\}\\
&\qquad\quad \Leftarrow^{*}_{\cos} \{\alpha, \beta\} = \left\{ \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \right\}\\
&\qquad \textbf{or:}\\
&\qquad C \Leftarrow_{id} \left\{ \alpha, [sAtom_{\cos(\alpha-\beta)}] \right\}\\
&\qquad\quad \Leftarrow_{\cos} \{\alpha, [\alpha - \beta]\} = \left\{ \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 2 \end{bmatrix} \right\}
\end{aligned}
$$

Here we see how *sAtoms* can have transformations (←) applied to fulfil an upper goal containing the *sAtom* given some lower goals made of regular atoms. Then we see how we can apply transformations to goal-bags (⇐) to progress with the search. Since, in the way Cain is designed, the atom level manipulation is already tied to the definition of the instruction set, we can make this adaptation with minimal implementation changes to the rest of Cain, and without changes to the principals of the graph traversal or reverse search algorithms. However, this does raise concerns for the heuristics used, since such changes would heavily impact how we go about predicting which of the now even more potential child nodes is most likely to lead to an optimal solution.

9.2.7 HIGH SPEED OBJECT RECOGNITION

One of the perceived benefits of focal plane sensor processors like Scamp5 is to perform low power high speed image recognition services. Cain aims to facilitate this research by allowing kernels to be processed in fewer instructions, so less power per frame and potentially faster given lighting conditions. Cain also has the ability to take multi channel inputs for its kernels, which could be used to take the result of a previous frame as input to the next. These features may open up new possibilities for motion detection and object detection at a scale that can be deployed locally on UAVs or other applications where the power budget is minimal.

Bibliography

[1] BARTHELS, H. ; PSARRAS, C. ; BIENTINESI, P. : *Linnea: Automatic Generation of Efficient Linear Algebra Programs*. https://arxiv.org/pdf/1912.12924.pdf. Version: 2019

[2] BOSE, L. ; CHEN, J. ; CAREY, S. ; DUDEK, P. ; MAYOL-CUEVAS, W. : A Camera That CNN: Towards Embedded Neural Networks on Pixel Processor Arrays. (2019), 11. – IEEE/CVF International Conference on Computer Vision (ICCV) 2019 ; Conference date: 27-10-2019 Through 02-11-2019

[3] BOUKNIGHT, W. ; DENENBERG, S. ; MCINTYRE, D. ; RANDALL, J. ; SAMEH, A. ; SLOTNICK, D. : The Illiac IV system. In: *Proceedings of the IEEE* 60 (1972), 05, S. 369 – 388. http://dx.doi.org/10.1109/PROC.1972.8647. – DOI 10.1109/PROC.1972.8647

[4] CAREY, S. J. ; BARR, D. R. W. ; WANG, B. ; LOPICH, A. ; DUDEK, P. : Locating high speed multiple objects using a SCAMP-5 vision-chip. In: *2012 13th International Workshop on Cellular Nanoscale Networks and their Applications*, 2012. – ISSN 2165–0160, S. 1–2

[5] CAREY, S. ; LOPICH, A. ; BARR, D. ; WANG, B. ; DUDEK, P. : A 100,000 fps vision sensor with embedded 535GOPS/W 256 × 256 SIMD processor array. (2013), 01, S. C182–C183. ISBN 978–1–4673–5531–5

[6] CHEN, J. : *scamp5 kernel api macro analog.hpp File Reference*. https://personalpages.manchester.ac.uk/staff/jianing.chen/scamp5d_lib_doc_html/scamp5__kernel__api__macro__analog_8hpp.html. Version: Jan 2020

[7] DEBRUNNER, T. : *Automatic Code Generation and Pose Estimation on Cellular Processor Arrays (CPAs)*, Imperial College London, Diplomarbeit, 2019. https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1617-pg-projects/DebrunnerT-Automatic-Code-Generation-and-Pose-Estimation-on-Cellular-Processor-Arrays.pdf

[8] DEBRUNNER, T. ; SAEEDI, S. ; KELLY, P. H. J.: AUKE: Automatic Kernel Code Generation for an Analogue SIMD Focal-Plane Sensor-Processor Array. In: *ACM Trans. Archit. Code Optim.* 15 (2019), Jan., Nr. 4. http://dx.doi.org/10.1145/3291055. – DOI 10.1145/3291055. – ISSN 1544–3566

[9] DIJKSTRA, E. W.: A note on two problems in connexion with graphs. In: *Numerische mathematik* 1 (1959), Nr. 1, S. 269–271

[10] GAMMA, E. : *Design patterns : elements of reusable object-oriented software*. Reading, Mass. : Addison-Wesley, 1995

[11] GEBSER, M. ; KAMINSKI, R. ; KAUFMANN, B. ; OSTROWSKI, M. ; SCHAUB, T. ; THIELE, S. : Engineering an Incremental ASP Solver. In: *ICLP* Bd. 5366, Springer, 2008 (Lecture Notes in Computer Science), S. 190–205

[12] GELFOND, M. ; LIFSCHITZ, V. : The Stable Model Semantics for Logic Programming. In: KOWALSKI, R. (Hrsg.) ; BOWEN (Hrsg.) ; KENNETH (Hrsg.): *Proceedings of International Logic Programming Conference and Symposium*, MIT Press, 1070-1080

[13] GRANLUND, T. ; KENNER, R. : Eliminating Branches Using a Superoptimizer and the GNU C Compiler. In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. Association for Computing Machinery (PLDI '92). – ISBN 0897914759, 341-352

[14] GUILLARD, B. : *Optimising convolutional neural networks for super fast inference on focal-plane sensor-processor arrays*, Imperial College London, Diplomarbeit, 2019. https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1819-pg-projects/Optimising-convolutional-neural-networks-for-super-fast-inference-on-focal-plane-sensor-processor-arrays.pdf

[15] HART, P. E. ; NILSSON, N. J. ; RAPHAEL, B. : A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In: *IEEE Transactions on Systems Science and Cybernetics* 4 (1968), Nr. 2, S. 100–107

[16] HORD, R. : *The Illiac IV: The First Supercomputer*. Springer Berlin Heidelberg https://books.google.ca/books?id=mG2rCAAAQBAJ. – ISBN 9783662103456

[17] IN-CHEOL PARK ; HYEONG-JU KANG: Digital filter synthesis based on an algorithm to generate all minimal signed digit representations. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21 (2002), Dec, Nr. 12, S. 1525–1529. http://dx.doi.org/10.1109/TCAD.2002.804374. – DOI 10.1109/TCAD.2002.804374. – ISSN 1937–4151

[18] LECUN, Y. ; CORTES, C. ; BURGES, C. : MNIST handwritten digit database. In: *ATT Labs [Online]*. 2 (2010). http://yann.lecun.com/exdb/mnist

[19] LIFSCHITZ, V. : What is Answer Set Programming? In: *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI Press, 2008 (AAAI'08). – ISBN 9781577353683, S. 1594–1597

[20] MASSALIN, H. : Superoptimizer: A Look at the Smallest Program. In: *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*. IEEE Computer Society Press (ASPLOS II). – ISBN 0818608056, 122âĂŞ126

[21] POLETTO, M. ; SARKAR, V. : Linear Scan Register Allocation. In: *ACM Trans. Program. Lang. Syst.* 21 (1999), Sept., Nr. 5, 895âĂŞ913. http://dx.doi.org/10.1145/330249.330250. – DOI 10.1145/330249.330250. – ISSN 0164–0925

[22] REDDAWAY, S. F.: a distributed array processor. In: *ISCA '73: Proceedings of the 1st annual symposium on Computer architecture*. New York, NY, USA : ACM Press, 1973, S. 61–65

[23] SOBEL, I. : An Isotropic 3x3 Image Gradient Operator. In: *Presentation at Stanford A.I. Project 1968* (2014), 02

[24] VINOD, A. ; LAI, E. ; MASKELL, D. L. ; MEHER, P. : An improved common subexpression elimination method for reducing logic operators in FIR filter implementations without increasing logic depth. In: *Integration* 43 (2010), Nr. 1, 124 - 135. http://dx.doi.org/https://doi.org/10.1016/j.vlsi.2009.07.001. – DOI https://doi.org/10.1016/j.vlsi.2009.07.001. – ISSN 0167–9260

[25] WONG, M. : *Analog Vision - Neural Network Inference Acceleration using Analog SIMD Computation in the Focal Plane*, Imperial College London, Diplomarbeit, 2018. https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1718-pg-projects/WongM-Analog-Vision.pdf

[26] XIMEA: *xiB - PCI Express Cameras with high speed and resolution.* https://www.ximea.com/pci-express-camera/pci-express-camera

[27] ZARÁNDY, Á. : *Focal-Plane Sensor-Processor Chips.* Springer New York (SpringerLink : Bücher). https://books.google.co.uk/books?id=AKwyCZvsXWMC. – ISBN 9781441964755