Computational complexity of an optical model of computation

Damien Woods

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

A thesis submitted for the degree of Doctor of Philosophy

Department of Computer Science
National University Of Ireland, Maynooth

Supervisor: J. Paul Gibson
Research Adviser: Thomas J. Naughton
External Examiner: Cristopher Moore
Internal Examiner: Barak Pearlmutter
Department Head: Ronan Reilly

August 2005

Abstract

We investigate the computational complexity of an optically inspired model of computation. The model is called the continuous space machine and operates in discrete timesteps over a number of two-dimensional complex-valued images of constant size and arbitrary spatial resolution.

We define a number of optically inspired complexity measures and data representations for the model. We show the growth of each complexity measure under each of the model's operations.

We characterise the power of an important discrete restriction of the model. Parallel time on this variant of the model is shown to correspond, within a polynomial, to sequential space on Turing machines, thus verifying the parallel computation thesis. We also give a characterisation of the class NC. As a result the model has computational power equivalent to that of many well-known parallel models. These characterisations give a method to translate parallel algorithms to optical algorithms and facilitate the application of the complexity theory toolbox to optical computers.

Finally we show that another variation on the model is very powerful; illustrating the power of permitting nonuniformity through arbitrary real inputs.

Acknowledgements

Many thanks are due to my supervisor Paul Gibson for his encouragement, time and belief in me. His efforts in the writing-up stage of the project are especially appreciated. I am very grateful to Tom Naughton for many things, in particular his illuminating answers to my many optics questions. Also appreciated are his good books, beers, cinema trips and friendship. Paul and Tom have taught me a thing or two about never not writing incorrect wronglish.

I thank my examiners Cris Moore and Barak Pearlmutter for stimulating conversations about my work, their thoughts and time are very much appreciated. I also thank Maurice Margenstern and Fred Lunnon for useful discussions.

The staff and postgrads of the Computer Science Department are acknowledged for making my place of work something more than just a place of work. My breadth of knowledge and quality of life were improved by interesting discussions, debates, violent clashes and laughs on a variety of topics with friends and colleagues. In particular I mention Aidan Delaney, Peter Gaughran, Taina Lehtimäki, Niall Murphy, Damian Murphy and Turlough Neary. Kai Wulff gets a special mention, we went through this process together, he is a good friend.

Xavier Jolly, Alan McDonnell and Ciarán Byrne; thanks for all those years of great music.

I'm fortunate to have great parents, I wouldn't be here without them, in fact I wouldn't be there nor anywhere. I don't thank my siblings for stealing my food but I do thank them for being a good laugh.

During my studies I was funded by a number of entities. They, and their euros and pounds, are gratefully acknowledged: The Irish Research Council for Science Engineering and Technology, NUI Maynooth and Enterprise Ireland.

I am grateful to Tony Seda of UCC for allowing and encouraging me to spend ample time preparing for my viva examination. Thanks to Carlos Martin-Vide for permitting my attendance lectures of the 1$^{\text{st}}$ International PhD School in Formal Languages and Applications, in Tarragona, Spain (2001). During those two months I gained knowledge from the lecturers, made some friends and enjoyed the sunshine.

Most of all I thank my closest friend Beverley Henley. Her love, encouragement, intelligence and humour are always appreciated. Thanks are due for the copious amounts of tea, her drug of choice. While writing our theses together she tried her very best to get me addicted, but alas she has failed.

Now to thank myself, with a really good beer ... ahh, forget it, I've found the Gold Blend.

To my family:
Beverley, Rosealeen, Owenie, Adele,
Gene, Niamh, Owen and Arlene.
Youse get one chapter each,
please fight over who gets what.

Contents

List of Figures

1

Introduction

The study of computational models that are inspired by nature has witnessed huge research growth in recent times. quantum, biomolecular, membrane, and dynamical systems computing are but a few such fruitful fields. When viewed in a certain light, nature seems to provide many kinds of prefabricated and highly optimised computers. Usually the difficult part is finding the light.

In this work we study a computational model that is inspired by classical Fourier optics. Our work is motivated by theoretical questions: What can we compute with optics? What can we not compute? How does optics relate to other notions of computation?

The model we study is called the Continuous Space Machine (CSM). The model is originally by Naughton and the earliest version was published in 2000 [Nau00b]. The CSM was developed for the analysis of (analog) Fourier optical computing architectures and algorithms, specifically pattern recognition and matrix algebra processors [Goo96]. The model uses images, arranged in a grid structure, for data storage. The program too resides in images. The CSM has the ability to perform Fourier transformation, resizing, addition, multiplication, and other operations on images, all in unit time.

On the one hand we show that the most general form of the model is not suited to analysis using standard techniques from complexity theory. On the other hand, one of our main results is that an important restriction of the model verifies the parallel compution thesis, and as such is intimately related to a wide variety of sequential and parallel computational models.

1.1 Natural computation

In the last decade or so there has been increasing interest in computation inspired by nature. This has spawned a wide variety of computational models, in the hope of one day building novel computers. Additionally there has been a considerable effort to try to understand natural phenomena through the lens of computational complexity theory. For the theorist, a new computational model gives a fresh language in which to speak about something that is already understood. Hopefully, the language leads to novel intuitions on something we are not so sure about. We present a selection of models and recent results.

In 1987, Head [Hea87] introduced the idea of a splicing system to model DNA from a formal language point of view. On the other hand Adleman [Adl94] programed DNA, using a straightforward molecular biology procedure, to solve a small instance of the Hamiltonian path problem. This, along with Lipton's theoretical work [Lip95], generated much excitement and led to a proliferation of biomolecular techniques being applied to a range of algorithmic problems [Yok02].

In the early 1980s, Feynman [Fey82] wondered how one would simulate a quantum mechanical system using a classical computer (and subsequently proposed the concept of quantum computation). Deutsch [Deu85] described a universal quantum computer. Shor's polynomial time quantum factoring algorithm [Sho94] and Grover's square root time quantum search algorithm [Gro96] illustrate that Feynman's problem might indeed be a difficult one to solve efficiently. At the present time quantum computation[1] is an exciting and fruitful research area for physicists and computer scientists alike [NC00].

Membrane computing is a rapidly growing field, despite the lack of current implementations for membrane computers [Pău00, Pău02]. Membrane computers model cellular processes using a tree-like membrane structure where processing on multisets is carried out. Recently Sosík [Sos03] has shown that a certain type of the membrane system is at least as powerful as parallel computation thesis models. The model has "active membranes and 2-division". The result was shown by giving a linear time algorithm for the PSPACE-complete problem *quantified Boolean formula*. This result was subsequently improved upon [AMVP03], and a similar result using membrane creation has also been given [GNPJRC05]. It is an open problem whether there are certain types of membrane systems that either characterise PSPACE (and verifies the parallel computation thesis) or perhaps $\mathsf{NP} \cup \text{co-}\mathsf{NP}$.

An interesting area of research is that of investigating the computational complexity of dynamical systems, for example by showing that they simulate Turing machines and other computational models [Moo90, SS91, Moo91, Moo98, KM99]. Moore and others have looked at the complexity of certain physical processes in terms of how computationally complex they are to predict [MN97, Moo97, MN99, MM00].

Inspired by classical recursion function theory, Moore initiated [Moo96] the study of real recursive functions. Real recursive functions are defined

[1]Incidentally, the quantum computing literature is rather unique in that one can find hilarious comments of the type: "As any actual computer must, first and foremost, be a physical device, the correct theory of computation ought to be a branch of physics rather than a branch of mathematics" [WC98].

using differentiation and can be viewed as defining programs that run in continuous time. This and other analog systems were further developed, and related to, standard notions of computational complexity by Campagnolo and others, for example see [Cam01, CMC02, Cam02, Cam04, dSG02, dSG04, MC04, BH04a, BH04b].

1.2 Optical computation

Besides isolated studies [Cau90], computational complexity of optical computers has received relatively little attention in comparison to the resources devoted to the designs, implementations and algorithms for physical optical computers (for example see [Goo96, YJY01, Lee95, AS92, LP92, McA91], and the references therein). Some authors have complained about the lack of suitable models [LP92, Fei88]. Many other areas of natural computing, such as those discussed above, have not suffered from this problem. Even so, we discuss some optical computation research that is close to the goals of the theoretical computer scientist.

Reif and Tyagi [RT97] study two optically inspired models. The first model is a 3D VLSI model augmented with a 2D discrete Fourier transform (DFT) primitive and parallel optical interconnections. The second model is a DFT circuit with operations (multiplication, addition, comparison of two inputs, DFT) that compute over an ordered ring. Time complexity is defined for both models in the obvious way. For the first model, volume complexity is defined as the volume of the smallest convex box enclosing an instance of the model. For the DFT circuit, size is defined as the number of edges plus gates. Constant time, polynomial size/volume, algorithms for a number of problems are reported including 1D DFT, matrix multiplication, sorting and string matching [RT97].

Feitelson [Fei88] gives a call to theoretical computer scientists to apply their knowledge and techniques to optical computing. He then goes on to describe a slight generalisation on the concurrent read, concurrent write parallel random access machine (CRCW PRAM), by augmenting it with two optically inspired operations. The first is the ability to write the same piece of data to many global memory locations at once. Secondly, if many values are concurrently written to a single memory location then a summation of those values is computed in a single timestep. He mentions that this model can simulate the CRCW PRAM with only a constant time overhead, whereas the converse constant time simulation cannot be proved since the best (bounded fan-in) CRCW PRAM algorithms can not perform the summation in constant time. Essentially Feitelson is using ‘unbounded fan-in with summation’ and ‘unbounded fan-out’. His architecture mixes a well known discrete model with some optical capabilities and his analysis does not go much further than what we have already mentioned.

The model we study is inspired by analog Fourier optical computing architectures, specifically pattern recognition and matrix algebra processors [Goo96, NJK$^+$99]. For example, these architectures have the ability to do unit time Fourier transformation using coherent (laser) light and lenses. Motivations for the original model definition can be found in [Nau00b]. Subsequent to the original CSM definition Naughton showed [Nau00a] that the model can simulate Turing machines, thus giving a lower bound on its computational power. Later we showed [NW01] that the model could simulate Type-2 Turing machines [Wei00]. We also gave an ω-language that is Type-2 (and Turing machine) undecidable, but is CSM decidable. In this thesis we continue the study and clarification of the power of this model.

1.3 Parallel complexity theory

There has been much work towards developing algorithms and techniques for speeding up interesting problems by using parallel architectures [Rei93, Qui94, Akl97, GGKK03]. To date, the theoretical work on optical computation has been in this vein, as can be seen from the references in the previous section. An alternative approach is to ask the following question: How does a given optical model relate to standard sequential and parallel models? Establishing a relationship with computational complexity theory, by describing both upper and lower bounds on the model, gives immediate access to a large collection of useful proof techniques and algorithms.

1.3.1 The parallel computation thesis

The parallel computation thesis [Gol77] states that parallel time corresponds to sequential space for reasonable parallel and sequential models. This broad statement needs to be qualified somewhat. Given a parallel model M we say that M verifies the parallel computation thesis if

$$M\text{-TIME}(T^{O(1)}(n)) = \text{SPACE}(T^{O(1)}(n))\,.$$

The notation is essentially saying that the class of problems solvable in parallel time for M corresponds, within a polynomial, to the class of problems solvable in sequential space on a Turing machine. Of course the thesis can never be proved, it relates the intuitive notion of parallelism to the mathematical notion of a Turing machine. When results of this type were first shown researchers were suitably impressed; their parallel models truly had great power. For example if M verifies the thesis then M decides PSPACE (including NP) languages in polynomial time. However there is another side to this coin. It is straightforward to verify that given our current best algorithms, M will use at least a superpolynomial amount of some other resource (like space or number of processors) to decide a PSPACE-complete or NP-complete language. Since the composition of polynomials is itself a polynomial, it follows that if we restrict the parallel computer to use at most polynomial time and polynomial other resources, then it can at most solve problems in P.

Nevertheless, asking if M verifies the thesis is an important question. Certain problems (e.g. the class NC) are efficiently parallelisable, in that they can be solved exponentially faster on parallel computation thesis models than on sequential models. If M verifies the thesis then we know it will be useful to apply M to these problems. We also know that if M verifies the thesis then there are (P-complete) problems for which it is widely believed that we will not find exponential speed up using M.

The parallel computation thesis was initially discussed by Chandra and Stockmeyer [CS76] and Goldschlager [Gol77]; Goldschlager was the first to study the thesis in some detail. The first models that satisfied the above equality, and thus motivated the stating of the thesis, were the vector machine model of Pratt, Rabin and Stockmeyer [PRS74] and the multiplication random access machine of Hartmanis and Simon [HS74]. A host of other models have been shown to verify the thesis, we mention a few by name, details can be found in the references: uniform circuits [Bor77], PRAMs [FW78], conglomerates [Gol78], k-PRAMs [SS79], alternating Turing machines [CKS81], SIMDAGs [Gol82], array processing machines [vLW87], associative storage modification machines [TvEB93]. Of particular interest in our work are vector machines and uniform circuits; these are introduced in later chapters.

Of course, not all parallel models verify the parallel computation thesis. If P $\neq$ PSPACE then the parallel Turing machines of Widermann [Wie84, vEB90] are too weak to verify the thesis, yet for some problems they are more powerful than standard sequential models. In addition, van Emde Boas [vEB90] and Parberry [Par87] survey a number of parallel models that seem to be strictly more powerful than parallel computation thesis models. As mentioned above, it is currently unknown whether certain types of membrane computers verify the thesis [Sos03, AMVP03]. Dymond and Cook [DC89], and Parberry [Par87] both suggest variations on the parallel computation thesis. Dymond and Cook put forward the thesis that the resources time and hardware on parallel machines are simultaneously polynomially related to sequential Turing machine tape head reversals and space requirements respectively. The thesis of Parberry states that in a parallel machine with unbounded fan-in communication, time and word size complexity are simultaneously equivalent to alternations and time on an alternating Turing machine, respectively, and within a constant and a polynomial, respectively.

The interested reader will find discussions on the parallel computation thesis in a number of complexity theory books and publications, for example: [vEB90, KR90, GHR95, BDG88b].

1.4 Thesis overview

In Chapter 2 we define our model at its most general. We then define a total of seven complexity measures, each corresponding to some real-world resource.

A natural question to ask from these definitions is: How does each operaton affect resource growth over time? In Chapter 3 we answer this question for each pair of operations and complexity measures. Some results seem quite intuitive, while others do not. As is to be expected, under certain operations many of the measures do not grow at all. Others grow at rates comparable to massively parallel models. By allowing operations like Fourier transform we are mixing the continuous and discrete worlds, hence some measures grow to infinity in one timestep. The chapter closes with the definition of a restriction on the model, called the $\mathcal{C}_2$-CSM.

In preparation for later results, Chapter 4 introduces some image data structures and a high level programming language for the model. This new programming language is intimately related with the model definition and has no more or less power. Its main advantage is ease of use; it shortens and simplifies CSM programs.

One of our main results is showing that the $\mathcal{C}_2$-CSM verifies the parallel computation thesis. That is, within a polynomial $\mathcal{C}_2$-CSM parallel time corresponds to sequential space. The two inclusions necessary to show this are given in separate chapters. Chapter 5 gives a $\mathcal{C}_2$-CSM simulation of the vector machine model. This simulation is useful from the point of view of translating vector machine algorithms into optical ones.

Chapter 6 gives the converse inclusion by simulation of the $\mathcal{C}_2$-CSM with circuits. We use logspace uniform, bounded fan-in circuits which are known to verify the parallel computation thesis. Both simulations imply some immediate corollaries on the class of $\mathcal{C}_2$-CSMs. For example, we show that a subclass of $\mathcal{C}_2$-CSMs accept exactly the class NC.

In Chapter 7 we recall a more general variation of the model, essentially a $\mathcal{C}_2$-CSM with arbitrary real inputs. The chapter shows the hideous power of allowing such oracle-type inputs, as we prove that this model can decide any language over a finite alphabet; thus providing further evidence of the relevance of our more restricted version of the model.

We close with some remarks and possible directions for future work. Much of the work presented here has appeared in published form [WN05, WG05a, WG05b, Woo, NW01].

2

Model definition

2.1 Introduction

In this chapter we introduce the CSM. The model uses complex-valued images, arranged in a grid structure, for data storage. The program also resides in images. The CSM has the ability to perform Fourier transformation, complex conjugation, multiplication, addition, thresholding and resizing of images. It has some simple control flow operations and is deterministic. To analyse such a model we define complexity measures that are inspired by optics. For example, spatial resolution corresponds to the intuitive notion of number of pixels. We give a total of seven measures.

We begin by defining images and some of the optically inspired functions that are assumed as basic CSM operations. Earlier versions of the definitions in this chapter were created in collaboration with T. Naughton [WN05].

2.2 CSM

Definition 2.2.1 (Complex-valued image) *A complex-valued image (or simply, an image) is a function* $f : [0,1) \times [0,1) \to \mathbb{C}$*, where* $[0,1)$ *is the half-open real unit interval.*

We let $\mathcal{I}$ denote the set of all complex-valued images.

2.2.1 Optical functions

We define six functions that are implemented in six of the CSM's ten operations. Let each $f \in \mathcal{I}$ be parameterised by orthogonal dimensions x and y; we often indicate this by writing f as $f(x,y)$. The function $h : \mathcal{I} \to \mathcal{I}$ gives the one-dimensional (1D) Fourier transformation (in the x-direction) of its 2D argument image f. The function h is defined as

$$h\left(f(x,y)\right) = h'\left(F\left(\alpha,y\right)\right)\ , \tag{2.2.1}$$

where $F(\alpha, y)$ is the Fourier transform (FT) in the x-direction of $f(x, y)$, defined as [Van92, Goo96]

$$F(\alpha, y) = \int_{-\infty}^{\infty} f(x, y) \exp\left[\mathrm{i}2\pi\alpha x\right] \mathrm{d}x\,,$$

where $\mathrm{i} = \sqrt{-1}$, and where $h'(F(\alpha, y)) = F(\theta\alpha, y)$. Here, h' uses the value θ to linearly rescale its argument F so that F is defined over $[0, 1) \times [0, 1)$. The function $v : \mathcal{I} \to \mathcal{I}$ gives the 1D Fourier transformation (in the y-direction) of its 2D argument image f, and is defined as

$$v\left(f(x, y)\right) = v'\left(F(x, \beta)\right)\,, \tag{2.2.2}$$

where $F(x, \beta)$ is the FT in the y-direction of $f(x, y)$, defined as [Van92, Goo96]

$$F\left(x, \beta\right) = \int_{-\infty}^{\infty} f(x, y) \exp\left[\mathrm{i}2\pi\beta y\right] \mathrm{d}y\,,$$

and where $v'(F(x, \beta)) = F(x, \theta\beta)$.

The function $* : \mathcal{I} \to \mathcal{I}$ gives the complex conjugate of its argument image,

$$*(f(x, y)) = f^*(x, y)\,, \tag{2.2.3}$$

where f^* denotes the complex conjugate of f. The complex conjugate of a scalar $z = a + \mathrm{i}b$ is defined as $z^* = a - \mathrm{i}b$.

The function $\cdot : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$ gives the pointwise complex product of its two argument images,

$$\cdot\,(f(x, y), g(x, y)) = f(x, y)g(x, y)\,, \tag{2.2.4}$$

The function $+ : \mathcal{I} \times \mathcal{I} \to \mathcal{I}$ gives the pointwise complex sum of its two argument images,

$$+\,(f(x, y), g(x, y)) = f(x, y) + g(x, y)\,. \tag{2.2.5}$$

The function $\rho : \mathcal{I} \times \mathcal{I} \times \mathcal{I} \to \mathcal{I}$ performs amplitude thresholding on its first image argument using its other two real valued ($z_\mathrm{l}, z_\mathrm{u} : [0, 1) \times [0, 1) \to \mathbb{R}$) image arguments as lower and upper amplitude thresholds, respectively,

$$\rho\left(f(x, y), z_\mathrm{l}(x, y), z_\mathrm{u}(x, y)\right) = \begin{cases} z_\mathrm{l}(x, y), & \text{if } |f(x, y)| < z_\mathrm{l}(x, y) \\ |f(x, y)|\,, & \text{if } z_\mathrm{l}(x, y) \leqslant |f(x, y)| \leqslant z_\mathrm{u}(x, y) \\ z_\mathrm{u}(x, y), & \text{if } |f(x, y)| > z_\mathrm{u}(x, y)\,. \end{cases} \tag{2.2.6}$$

The amplitude of an arbitrary $z \in \mathbb{C}$ is denoted $|z|$ and is defined as $|z| = \sqrt{z(z^*)}$, where we take the positive root.

2.2.2 CSM

Next we will define the CSM. To prepare, we let $\mathbb{N}$ be the set of nonnegative integers and $\mathbb{N}^+ = \{1, 2, 3, \ldots\}$. For a given CSM M we let $\mathcal{N}$ be a countable set of images that encode M's addresses. Additionally, for a given M there is a function $\mathfrak{E} : \mathbb{N} \to \mathcal{N}$ such that $\mathfrak{E}$ is Turing machine decidable, under some *reasonable* representation of images as words. We call $\mathfrak{E}$ the address encoding function of M. An address is simply an element of $\mathbb{N} \times \mathbb{N}$.

Definition 2.2.2 (CSM) *A continuous space machine or CSM is a quintuple $M = (\mathfrak{E}, L, I, P, O)$, where*

$\mathfrak{E} : \mathbb{N} \to \mathcal{N}$ *is the address encoding function*

$L = ((s_\xi, s_\eta), (a_\xi, a_\eta), (b_\xi, b_\eta))$ *are the addresses: sta, a, and b,*

$I = \left((\iota_{1_\xi}, \iota_{1_\eta}), \ldots, (\iota_{k_\xi}, \iota_{k_\eta})\right)$ *are the addresses of the k input images,*

$P = \left\{(\zeta_1, p_{1_\xi}, p_{1_\eta}), \ldots, (\zeta_r, p_{r_\xi}, p_{r_\eta})\right\}$ *are the r programming symbols and their addresses where $\zeta_j \in (\{h, v, *, \cdot, +, \rho, st, ld, br, hlt\} \cup \mathcal{N}) \subset \mathcal{I}$,*

$O = \left((o_{1_\xi}, o_{1_\eta}), \ldots, (o_{l_\xi}, o_{l_\eta})\right)$ *are the addresses of the l output images.*

Each address is an element from $\{0, 1, \ldots, \Xi - 1\} \times \{0, 1, \ldots, \mathcal{Y} - 1\}$ where $\Xi, \mathcal{Y} \in \mathbb{N}^+$. Addresses a and b are distinct.

Addresses whose contents are not specified by P in a CSM definition are assumed to contain the constant image $f(x, y) = 0$.

We interpret this definition to say that M is (initially) defined on a grid of images bounded by the constants Ξ and $\mathcal{Y}$, in the horizontal and vertical directions respectively. We have specified the location of programming symbols on the grid. We have also specified where inputs would be placed in an instance of M and where we would expect to find outputs. The address *sta* is the program start location. Addresses a and b are special well-known images that are used by many of the model's operations as well-known addresses.

Some remarks are warrented on $\mathfrak{E}$, the address encoding function. Since $\mathfrak{E}$ is computable in the sense described above, the decoding function $\mathfrak{E}^{-1} : \mathcal{N} \to \mathbb{N}$ is also computable. Notice that we do not force any particular $\mathfrak{E}$ on the programmer; we do not want to stop her from inventing novel addressing schemes.

When stating that $\mathfrak{E}$ was computable we added the caveat *reasonable* for the representation of $\mathcal{N}$ as words. When writing CSMs we do not want the mapping (from $\mathcal{N}$ to words) to hide complicated behaviour that might undermine any claims about efficiency of our algorithms. However there is no standard way to represent elements of $\mathcal{N}$ as words. Anyone writing a CSM should offer a convincing argument that their mapping from $\mathcal{N}$ to words is reasonable. Other authors have also raised this representation issue [Moo98, vEB90, WW86] for different models, but with similar motivations.

Finally, for an address encoding to be judged reasonable, it must be the case that $\mathfrak{E}$ and $\mathfrak{E}^{-1}$ are reasonable in the sense that they do not hide complicated behaviour. For example a CSM with an *intractable* $\mathfrak{E}$ could be programed to use $\mathfrak{E}$ as an oracle for 'efficiently' solving some intractable problem. This is quite unreasonable. In a subsequent chapter we place a tighter restriction on the computability of $\mathfrak{E}$.

2.2.3 CSM computation

We adopt a few notational conveniences for this section. Let $c \in \mathbb{N} \times \mathbb{N}$ be an address, the *image* at address c is denoted $\widehat{c}$. Sometimes the image at address c represents a natural number, in such cases that number is denoted $\mathfrak{E}^{-1}(\widehat{c})$. Each of e, u and w is a sequence, where each element of the sequence is from $\mathcal{I} \times \mathbb{N} \times \mathbb{N}$ (an image followed by an address).

Definition 2.2.3 (CSM configuration) *A configuration of a CSM M is a pair $\langle c, e \rangle$, where $c \in \{0, \ldots, \Xi - 1\} \times \{0, \ldots, \mho - 1\}$ is an address called the control. Also, $e = ((i_{0\,0}, 0, 0), \ldots, (i_{\gamma\,\delta}, \gamma, \delta), \ldots)$ is a tuple with finitely many elements that contains M's images and each of their addresses, with $i_{\gamma\,\delta} \in \mathcal{I}$ denoting image at address (γ, δ). The elements of tuple e are ordered first by each δ then by each γ.*

An *initial configuration* of M is a configuration $C_{\text{sta}} = \langle c_{\text{sta}}, e_{\text{sta}} \rangle$, where $c_{\text{sta}} = (s_\xi, s_\eta)$ is the program start address *sta*, and e_{sta} contains all elements of P and the k input images $(\varphi_1, \iota_{1_\xi}, \iota_{1_\eta}), \ldots, (\varphi_k, \iota_{k_\xi}, \iota_{k_\eta})$ as given by I. A *final configuration* of M is a configuration where the control is pointing at the *hlt* instruction. In other words, a configuration of the form $C_{\text{hlt}} = \langle (\gamma, \delta), (u, (hlt, \gamma, \delta), w) \rangle$, where u and w are given above. Notice that $\widehat{(\gamma, \delta)} = hlt$.

In Definition 2.2.4 we will define how a configuration may be altered by each CSM operation, in other words a computation step. Some of the operations can address (access) arbitrary images in the grid, using address parameters. We introduce some notation to help us describe this. The function $\phi((\gamma, \delta)) = (\gamma + 1, \delta)$ advances the control horizontally forward by one grid element while $\phi^{(k)}(c)$ is shorthand for function composition, e.g. $\phi^{(2)}(c) = \phi(\phi(c))$. At a given configuration $\langle c, e \rangle$ we let $\xi_1 = \mathfrak{E}^{-1}(\widehat{\phi(c)}), \xi_2 = \mathfrak{E}^{-1}(\widehat{\phi^{(2)}(c)}), \eta_1 = \mathfrak{E}^{-1}(\widehat{\phi^{(3)}(c)}), \eta_2 = \mathfrak{E}^{-1}(\widehat{\phi^{(4)}(c)})$. Hence each of ξ_1, ξ_2, η_1 and η_2 is the natural number represented by the 1st, 2nd, 3rd and 4th image after the control position, respectively. We let the *scaling relationships* for *st* and *ld* be $x' = (x + \gamma - \xi_1)/(\xi_2 - \xi_1 + 1)$ and $y' = (y + \delta - \eta_1)/(\eta_2 - \eta_1 + 1)$. Recall from the CSM definition that (a_ξ, a_η) is the well-known address a. We write $a(x, y)$ to mean the image stored in address a. Definition 2.2.4 is followed by an explanation and summarised in Figure 2.2.1. Additionally the operations *st* and *ld* are illustrated in Figure 2.2.2.

Definition 2.2.4 ($\vdash_M$) *Let $\vdash_M$ be a binary relation on configurations of CSM M. The relation $\vdash_M$ contains exactly the following ten elements.*

$$\langle c, (u, (i_{a_\xi a_\eta}, a_\xi, a_\eta), w)\rangle \vdash_M \langle \phi(c), (u, (h(i_{a_\xi a_\eta}), a_\xi, a_\eta), w)\rangle, \text{ if } \widehat{c} = h \tag{i}$$

$$\langle c, (u, (i_{a_\xi a_\eta}, a_\xi, a_\eta), w)\rangle \vdash_M \langle \phi(c), (u, (v(i_{a_\xi a_\eta}), a_\xi, a_\eta), w)\rangle, \text{ if } \widehat{c} = v \tag{ii}$$

$$\langle c, (u, (i_{a_\xi a_\eta}, a_\xi, a_\eta), w)\rangle \vdash_M \langle \phi(c), (u, (*(i_{a_\xi a_\eta}), a_\xi, a_\eta), w)\rangle, \text{ if } \widehat{c} = * \tag{iii}$$

$$\langle c, (u, (i_{a_\xi a_\eta}, a_\xi, a_\eta), w)\rangle \vdash_M \langle \phi(c), (u, (\cdot(i_{a_\xi a_\eta}, i_{b_\xi b_\eta}), a_\xi, a_\eta), w)\rangle, \text{ if } \widehat{c} = \cdot \tag{iv}$$

$$\langle c, (u, (i_{a_\xi a_\eta}, a_\xi, a_\eta), w)\rangle \vdash_M \langle \phi(c), (u, (+(i_{a_\xi a_\eta}, i_{b_\xi b_\eta}), a_\xi, a_\eta), w)\rangle, \text{ if } \widehat{c} = + \tag{v}$$

$$\langle c, (u, (i_{a_\xi a_\eta}, a_\xi, a_\eta), w)\rangle \vdash_M \langle \phi(c), (u, (\rho(i_{a_\xi a_\eta}, \widehat{\phi(c)}, \widehat{\phi^{(2)}(c)}), a_\xi, a_\eta), w)\rangle, \text{if } \widehat{c} = \rho \tag{vi}$$

$$\langle c, (u_{\gamma\delta}, (i_{\gamma\delta}(x,y), \gamma, \delta), w_{\gamma\delta})\rangle \vdash_M \langle \phi^{(5)}(c), (u_{\gamma\delta}, (a\,(x',y')\,, \gamma, \delta), w_{\gamma\delta})\rangle,$$
$$\forall \gamma, \delta \textit{ s.t. } \xi_1 \leqslant \gamma \leqslant \xi_2, \eta_1 \leqslant \delta \leqslant \eta_2, \forall (x,y) \in [0,1) \times [0,1), \text{ if } \widehat{c} = st \tag{vii}$$

$$\langle c, (u, (a\,(x',y')\,, a_\xi, a_\eta), w)\rangle \vdash_M \langle \phi^{(5)}(c), (u, (i_{\gamma\delta}(x,y), a_\xi, a_\eta), w)\rangle,$$
$$\forall \gamma, \delta \textit{ s.t. } \xi_1 \leqslant \gamma \leqslant \xi_2, \eta_1 \leqslant \delta \leqslant \eta_2, \forall (x,y) \in [0,1) \times [0,1), \text{ if } \widehat{c} = ld \tag{viii}$$

$$\langle c, (u)\rangle \vdash_M \langle \mathfrak{E}^{-1}(\widehat{\phi(c)}), \mathfrak{E}^{-1}(\widehat{\phi^{(2)}(c)}), (u)\rangle, \text{ if } \widehat{c} = br \tag{vii}$$

$$\langle c, (u)\rangle \vdash_M \langle c, (u)\rangle, \text{ if } \widehat{c} = hlt \tag{vii}$$

The first six elements of $\vdash_M$ define the CSM's implementation of the functions defined in Eqs. (2.2.1) through (2.2.6). Notice that in each case the image at the well-known address a is overwritten by the result of applying one of h, v, $*$, $\cdot$, $+$ or ρ to its argument (or arguments). The value of the control c is then simply incremented to the next address, using the function ϕ defined earlier. The seventh element of $\vdash_M$ defines how the store operation copies the image at well-known address a to a 'rectangle' of images specified by the st parameters $\xi_1, \xi_2, \eta_1, \eta_2$. The eighth element of $\vdash_M$ defines how the load operation copies a rectangle of images specified by the ld parameters $\xi_1, \xi_2, \eta_1, \eta_2$ to the image at well-known address a. Operations st and ld are illustrated in Figure 2.2.2. Elements nine and ten of $\vdash_M$ define the control flow operations branch and halt, respectively. When the image at the address specified by the control c is br, the value of c is updated to the address encoded by the two br parameters. Finally, the hlt operation always maps a final configuration to itself.

Let $\vdash_M^*$ denote the transitive closure of $\vdash_M$. A halting computation by M is a finite sequence of configurations beginning in an initial configuration and ending in a final configuration: $C_{\text{sta}} \vdash_M^* C_{\text{hlt}}$.

For convenience, we use an informal 'grid' notation when specifying programs for the CSM. In our grid notation the first and second elements of an address tuple refer to the horizontal and vertical axes of the grid, respectively, and image $(0,0)$ is at the bottom left-hand corner of the grid. The images in a grid must have the same orientation as the grid. Hence in a given image f, the first and second elements of a coordinate tuple refer to the horizontal and vertical axes of f, respectively, and the coordinate $(0,0)$ is located at the bottom left-hand corner of f. Figure 2.2.1 informally explains the elements of $\vdash_M$, as they appear in this grid notation.

$\boxed{\mathrm{h}}$	: perform a horizontal 1D Fourier transform on the 2D image in a. Store result in a.
$\boxed{\mathrm{v}}$	: perform a vertical 1D Fourier transform on the 2D image in a. Store result in a.
$\boxed{*}$	: replace image in a with its complex conjugate.
$\boxed{\cdot}$	: multiply (point by point) the two images in a and b. Store result in a.
$\boxed{+}$	: perform a complex addition of a and b. Store result in a.
$\boxed{\rho}\boxed{z_\mathrm{l}}\boxed{z_\mathrm{u}}$	: $z_\mathrm{l}, z_\mathrm{u} \in \mathcal{I}$; filter the image in a by amplitude using z_l and z_u as lower and upper amplitude threshold images, respectively.
$\boxed{\mathrm{st}}\boxed{\xi_1}\boxed{\xi_2}\boxed{\eta_1}\boxed{\eta_2}$	: $\xi_1, \xi_2, \eta_1, \eta_2 \in \mathbb{N}$; copy the image in a into the rectangle of images whose bottom left-hand corner address is (ξ_1, η_1) and whose top right-hand corner address is (ξ_2, η_2).
$\boxed{\mathrm{ld}}\boxed{\xi_1}\boxed{\xi_2}\boxed{\eta_1}\boxed{\eta_2}$	: $\xi_1, \xi_2, \eta_1, \eta_2 \in \mathbb{N}$; copy into a the rectangle of images whose bottom left-hand corner address is (ξ_1, η_1) and whose top right-hand corner address is (ξ_2, η_2).
$\boxed{\mathrm{br}}\boxed{\xi}\boxed{\eta}$	: $\xi, \eta \in \mathbb{N}$; unconditionally branch to the image at address (ξ, η).
$\boxed{\mathrm{hlt}}$	: halt.

Figure 2.2.1: The set of CSM operations, given in our informal grid notation. For formal definitions see Definition 2.2.4.

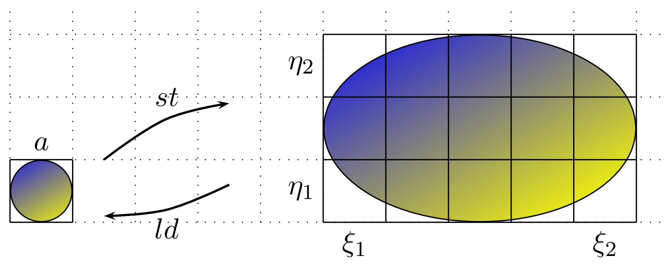

Figure 2.2.2: Operations *st* and *ld*.

The CSM has the ability to overwrite its own program. When analysing programs this is an undesirable feature and we do not use program overwriting in this work. In particular if any code is written outside the initial grid it will never be executed since Definition 2.2.3 ensures that in a valid configuration the control always remains within the initial grid.

	Symbol	Name	Description
1.	T	TIME	Number of timesteps
2.	G	GRID	Number of grid images
3.	R_{S}	SPATIALRES	Spatial resolution
4.	R_{A}	AMPLRES	Amplitude resolution
5.	R_{P}	PHASERES	Phase resolution
6.	R_{D}	DYRANGE	Dynamic range
7.	ν	FREQ	Frequency of illumination

Table 2.3.1: Summary of complexity measures for characterising CSMs.

2.3 Complexity of computations

In this section we define computational complexity measures for the CSM. We want our measures to be straightforward to analyse, while at the same time be meaningful by reflecting the reality of optical computing. A total of seven complexity measures are given and are summarised in Table 2.3.1.

All finite resource bounding functions f are from $\mathbb{N}$ into $\mathbb{N}$. We take the notation $O(f(n))$ to have the usual meaning as the set of all functions g such that for some nonnegative $r \in \mathbb{R}$ and $n_0 \in \mathbb{N}$: $g(n) \leqslant rf(n)$ for all $n \geqslant n_0$. We wish to avoid some of the ambiguities introduced by the the O notation [Knu97]. In an equality with O terms on one side of the '=' and non-O terms on the other, we write the O terms on the right. $\Omega(f(n))$ is the set of all functions g such that $g(n) \geqslant rf(n)$ for all $n \geqslant n_0$. Also $g(n)$ is $\Theta(f(n))$ if and only if $g(n)$ is $O(f(n))$ and $g(n)$ is $\Omega(f(n))$. All finite resource bounds that we use have the usual properties of being nondecreasing, and time (in the case of CSM TIME) or space (in the case of CSM 'space-like' measures) constructible on a Turing machine [BDG88a]. Such constructibility assumes a reasonable representation of images as finite length words, with the resource bounding function being defined on the length of such words. All logarithms are to the base 2. We refer the reader to the literature for the definitions of the various complexity classes used in our work (for example Johnson [Joh90] provides a good survey). However, we mention that NSPACE and DSPACE respectively denote nondeterministic and deterministic Turing machine space complexity classes.

Definition 2.3.1 *The* TIME *complexity of a CSM M is the number of configurations in the computation sequence of M, beginning with the initial configuration and ending with the first final configuration.*

The definition of TIME assigns unit cost to each instruction.

Definition 2.3.2 *The* GRID *complexity of a CSM M is the minimum number of images, that are arranged in a rectangular grid, for M to compute correctly on all inputs.*

From the CSM definition GRID is at least $\Xi\mathcal{Y}$. In previous work on the model [Nau00b, Nau00a, NW01, WN05] the number of grid images remained constant throughout a computation. Here we have altered the model (by introducing the address encoding function) so that GRID may grow over time.

Next we define SPATIALRES complexity. Let a *pixel* λ be a constant function defined on a real-valued rectangle with rational endpoints, as follows. $\lambda : [0, W) \times [0, H) \to z$ where $z \in \mathbb{C}$; $W, H \in \mathbb{Q}$; $0 < W, H \leqslant 1$; and $[0, W), [0, H) \subset \mathbb{R}$. Let a *raster image* be an image composed entirely of nonoverlapping pixels, each of the pixels are of equal height H, equal width W, identical orientation, and arranged into Φ columns and Ψ rows such that $\Phi W = 1 = \Psi H$. Formally, given $\Phi\Psi$ pixels $\{\lambda_{1,1}, \ldots, \lambda_{\Phi,\Psi}\}$, a raster image f' consisting of exactly these pixels is defined as

$$f'(x, y) = \lambda_{\gamma,\delta}, \text{ where } \gamma \text{ satisfies } \frac{\gamma - 1}{\Phi} \leqslant x < \frac{\gamma}{\Phi}, \quad \text{and } \delta \text{ satisfies } \frac{\delta - 1}{\Psi} \leqslant y < \frac{\delta}{\Psi}.$$

Let the *spatial resolution* of a raster image be $\Phi\Psi$: The number of pixels in that image. Let the process of *rasterising* an image be the function $S : \mathcal{I} \times (\mathbb{N} \times \mathbb{N}) \to \mathcal{I}$, defined as $S(f(x, y), (\Phi, \Psi)) = f'(x, y)$, where $f'(x, y)$ is a raster image, with $\Phi\Psi$ pixels arranged in Φ columns and Ψ rows, that somehow approximates $f(x, y)$.

If we choose a reasonable and realistic rasterisation then the details of S are not important; it suffices to say that (Φ, Ψ) can be regarded as defining a sampling grid with uniform sampling both horizontally and vertically, although the sampling rates in both directions can differ. For example, in this work S maps the value at the centre of each pixel-sized part of an image to a pixel of that value. In general, increasing the spatial resolution of the sampling (increasing Φ and/or Ψ) results in a closer approximation to $f(x, y)$.

Definition 2.3.3 *The* SPATIALRES *complexity of a CSM M is the minimum $\Phi\Psi$ such that if each image $f(x, y)$ in the computation of M is replaced with $S(f(x, y), (\Phi, \Psi))$ then M computes correctly on all inputs.*

If no such $\Phi\Psi$ exists then M has infinite SPATIALRES complexity. It can be seen that if the result of M's computation is determined solely by features within its images that are located at rational (respectively, irrational) coordinates then M would require finite (respectively, infinite) SPATIALRES. In optical image processing terms, and given the fixed size of our images, SPATIALRES corresponds to the space-bandwidth product of a detector or SLM (spatial light modulator – a device for encoding a function in an optical wavefront).

The AMPLRES complexity of a CSM M is the minimum amplitude resolution necessary for M to compute correctly on all inputs. This is formalised as follows. Consider the function $A : \mathcal{I} \times \{1, 2, 3, \ldots\} \rightarrow \mathcal{I}$ defined as

$$A(f(x,y),\mu) = \left\lfloor |f(x,y)|\mu + \frac{1}{2} \right\rfloor \frac{1}{\mu} \exp(\mathrm{i} \times \arg(f(x,y))) \,, \tag{2.3.1}$$

where $|\cdot|$ gives the amplitudes of its image argument, $\arg(\cdot)$ gives the phase angles (in the range $(0, 2\pi]$) of its image argument, and the floor operation is defined as operating separately on each value in its image argument. The value μ is the cardinality of the set of discrete nonzero amplitude values that each complex value in $A(f,\mu)$ can take, per half-open unit interval of amplitude. (Zero will always be a possible amplitude value irrespective of the value of μ.) To aid in the understanding of Eq. (2.3.1), note that

$$f(x,y) = |f(x,y)| \exp(\mathrm{i} \times \arg(f(x,y))) \,.$$

Definition 2.3.4 *The* AMPLRES *complexity of a CSM M is the minimum μ such that if each image $f(x,y)$ in the computation of M is replaced by $A(f(x,y),\mu)$ then M computes correctly on all inputs.*

If no such μ exists then M has infinite AMPLRES complexity. It can be seen that if the result of M's computation is determined solely by amplitude values within its images that are rational (respectively, irrational), then M would require finite (respectively, infinite) AMPLRES. For example, CSM instances that make use of only unary, binary and integer images (see Section 4.2) have constant AMPLRES of 1. Instances that use real number and real matrix images (see Section 4.2) have infinite AMPLRES complexity. In optical image processing terms AMPLRES corresponds to the amplitude quantisation of a signal.

The PHASERES complexity of a CSM M is the minimum phase resolution necessary for M to compute correctly on all inputs. Consider the function $P : \mathcal{I} \times \{1, 2, 3, \ldots\} \rightarrow \mathcal{I}$ defined as

$$P(f(x,y),\mu) = |f(x,y)| \exp\left(\mathrm{i} \left\lfloor \arg(f(x,y)) \frac{\mu}{2\pi} + \frac{1}{2} \right\rfloor \frac{2\pi}{\mu} \right) \,. \tag{2.3.2}$$

The value μ is the cardinality of the set of discrete phase values that each complex value in $P(f,\mu)$ can take.

Definition 2.3.5 *The* PHASERES *complexity of a CSM M is the minimum μ such that if each image $f(x,y)$ in the computation of M is replaced by $P(f(x,y),\mu)$ then M computes correctly on all inputs.*

If no such μ exists then M has infinite phaseRes complexity. It can be seen that if the result of M's computation is determined solely by phase values within its images that are rational (respectively, irrational) modulo 2π, or by a finite (respectively, infinite) set of rational phase values modulo 2π, then M would require finite (respectively, infinite) phaseRes. In optical image processing terms phaseRes corresponds to the phase quantisation [GS70] of a signal.

Definition 2.3.6 *The* dyRange *complexity of a CSM M is the ceiling of the maximum of all the amplitude values stored in all of M's images during M's computation.*

In optical processing terms dyRange corresponds to the dynamic range of a signal.

The seventh of our complexity measures is freq. The freq complexity of a CSM M is the minimum optical frequency necessary for M to compute correctly. The concept of minimum optical frequency is now explained. In optical implementations of the h and v operations (such as those suggested in [WN05]), one of the factors that determine the dimensions of the Fourier spectrum of $f \in \mathcal{I}$ is the frequency of the coherent illumination employed. Increasing the frequency of the illumination results in a smaller Fourier spectrum (components are spatially closer to the zero frequency point). In our definitions of h and v, we employ the constant θ to rescale the Fourier spectrum of f such that it fits into the dimensions of an image: $[0,1) \times [0,1)$. In general, however, a Fourier spectrum of an image will be infinite in extent. Therefore, according to the relationship between optical frequency and Fourier spectrum dimensions [Goo96, Van92], such a constant θ only exists when the wavelength of the illumination is zero, corresponding to illumination with infinite frequency. With a finite optical frequency, the h and v operations will remove all Fourier components with a spatial frequency higher than the cut-off imposed by θ. This is called low-pass filtering in signal processing terminology, and is equivalent to a blurring of the original signal. Given particular rasterisation and quantisation functions for the images in M, and a particular θ, the blurring effect might not influence the computation.

Definition 2.3.7 *The* freq *complexity of a CSM M to be the minimum optical frequency such that M computes correctly on all inputs.*

If approximations of a FT are sufficient for M, or if M does not execute h or v, then M requires finite freq. If the original (unbounded) definitions of h and v must hold then M requires infinite freq. Note also that using the traditional optical methods (such as those outlined in [WN05]), any lower bound on SPATIALRES complexity will impose a lower bound on FREQ complexity. In the context of traditional optical methods, this imposition is referred to as the diffraction limit. (The optical wavelength should be a constant times smaller than the smallest spatial feature that needs to be resolvable in an image.) In order not to rule out the applicability of novel sub-wavelength resolution techniques that go beyond the diffraction limit for our CSM algorithms we give each FREQ complexity as an upper bound.

Often we wish to make analogies between space on some well-known model and 'space-like' resources on the CSM. For this purpose we define the following convenient term.

Definition 2.3.8 *The* SPACE *complexity of a CSM M is the product of all of M's complexity measures except* TIME.

In [WN05] the interested reader will find a discussion on defining energy of computations in terms of the above mentioned complexity measures.

For each measure we have defined the complexity of a computation (sequence of configurations). We extend this definition to the complexity of a (possibly non-final) configuration in the obvious way. Additionally, we sometimes talk about the complexity of an image, this is simply the complexity of the configuration that the image is in.

2.4 Discussion

From our CSM definition of a CSM computation, operations compute on images in a parallel way. Images may represent a massive amount of data. Hence in Flynn's taxonomy [Fly72, MS98] CSMs are classed as single instruction, multiple data (SIMD) machines.

In earlier versions of the model [Nau00b, Nau00a, NW01, WN05] the number of grid images remained constant throughout the computation, that is GRID was always constant. The address encoding function $\mathfrak{E}$ was introduced to enable us to treat GRID as a useful complexity resource.

For the interested reader [WN05] contains an example address encoding with its optical implementation. Even though GRID is always constant in [WN05], we can still apply the same ideas for optical addressing, since for any given CSM instance the number of images is bounded by GRID. Also in [WN05] (and the references therein) there are some suggested implementations for the model's optical operations.

We have defined a number of complexity measures that reflect resource usage in optics. In the next chapter we analyse these measures with respect to the model's operations.

3

Complexity of CSM operations and the $\mathcal{C}_2$-CSM

3.1 Introduction

In the previous chapter we defined our model and a number of complexity measures. Here, we wish to understand how the various complexity measures grow, with respect to the CSM's operations over TIME. We tackle this question for each operation and complexity measure. Some of results seem quite intuitive, while others do not. As is to be expected, under certain operations many of the measures do not grow at all. Others grow at rates comparable to massively parallel models. By allowing operations like Fourier transformation we are mixing the continuous and discrete worlds, hence some measures grow to infinity in one timestep. We begin with some motivations.

Given any model of computation, execution of one of the model's operations may or may not increase the value of some complexity measure at a given time in the computation. For example, a move of a Turing machine tape head sometimes increases space by 1 and always increases time by 1. For sequential models it is usually quite obvious how execution of a single operation will effect the complexity of the computation. In parallel models execution of a single operation can lead to large growth. For example a multiplication or shift operation in a unit cost parallel model (such as the unrestricted vector machines of Pratt and Stockmeyer [PS76]) can double the length of a binary string in one step. When binary strings are interpreted as numbers, such multiplications and shifts generate large values very quickly. Characterising the growth in complexity is useful for proving upper bounds on computational power and setting reasonable restrictions on the model [BDG88b].

In Section 3.2 of this chapter we present general worst case complexity growth for the CSM. All restrictions of the CSM model will have complexity growth less or equal to this; a useful fact when proving upper bounds on computational power. It turns out that in the general case the CSM is extremely powerful, in more ways than one. This leads to the idea of finding meaningful restrictions of the CSM. In Section 3.3 we define an important restriction, called the $\mathcal{C}_2$-CSM.

3.2 Worst case CSM complexity growth

Our complexity growth analysis is worst case, hence we assume that at each computation step we want to preserve all information in each image. These results give some intuition into which operations may be useful for (say) TIME efficient solutions to various problems. For example a multiplicative increase in some resource in a single step is powerful in terms of TIME efficiency.

It turns out that some resources do not increase under certain operations, while others increase by a reasonable amount. Others are unbounded and increase to ∞. This gives strong motivation for restrictions on the model and raises some interesting questions.

When we defined complexity measures in Section 2.3 we used phrases along the lines of "... if each image in a CSM computation is replaced by a discretised image ...". In order to simplify our reasoning in this chapter we assume that the images actually have been replaced. This does not affect the outcomes of valid computations.

Table 3.2.1 lists the effect of each CSM operation on the complexity of computations. Specifically, for each operation and each complexity measure the table defines the value of the complexity measure after execution of the operation (at TIME $T+1$). The complexity of a configuration at TIME $T+1$ is at least the value it was at TIME T, since complexity functions are always nondecreasing. At a given timestep T each image in a CSM configuration has the same complexity. Our definition of TIME assigns unit time cost for each operation, hence we do not have a TIME column. Most entries are immediate from the definitions of the operation and complexity measure. For entries that are not immediate, a theorem is referenced directly below the entry in brackets.

	GRID	SPATIALRES	AMPLRES	DYRANGE	PHASERES	FREQ
h	G_T	∞ [3.2.1]	∞ [3.2.1]	∞ [3.2.2]	∞ [3.2.1]	∞ [3.2.1]
v	G_T	∞ [3.2.1]	∞ [3.2.1]	∞ [3.2.2]	∞ [3.2.1]	∞ [3.2.1]
$*$	G_T	R_{S_T}	R_{A_T}	R_{D_T}	R_{P_T} [3.2.4]	ν_T
$\cdot$	G_T	R_{S_T}	$(R_{A_T})^2$ [3.2.5]	$(R_{D_T})^2$ [3.2.6]	R_{P_T} [3.2.7]	ν_T
$+$	G_T	R_{S_T}	∞ [3.2.8]	$2R_{D_T}$ [3.2.9]	∞ [3.2.10]	ν_T
ρ	unbounded [3.2.11]	R_{S_T}	R_{A_T}	R_{D_T}	R_{P_T}	ν_T
st	unbounded [3.2.11]	R_{S_T}	R_{A_T}	R_{D_T}	R_{P_T}	ν_T
ld	unbounded [3.2.11]	unbounded [3.2.12]	R_{A_T}	R_{D_T}	R_{P_T}	unbounded [3.2.12]
br	G_T [3.2.13]	R_{S_T}	R_{A_T}	R_{D_T}	R_{P_T}	ν_T
hlt	G_T	R_{S_T}	R_{A_T}	R_{D_T}	R_{P_T}	ν_T

Table 3.2.1: Upper bounds on CSM resource usage after a single timestep. For each operation and complexity measure pair, the table entry defines the worst case upper bound on CSM resource usage at TIME $T+1$. This value is given in terms of the resources used at TIME T: GRID $= G_T$, SPATIALRES $= R_{S_T}$, AMPLRES $= R_{A_T}$, DYRANGE $= R_{D_T}$, PHASERES $= R_{P_T}$ and FREQ $= \nu_T$. Where necessary, proofs are referenced in brackets.

Theorem 3.2.1 *(h/v & SPATIALRES, AMPLRES, PHASERES and FREQ) Let $\langle c, e\rangle_T$ be a configuration of CSM M at* TIME *T where either $\widehat{c} = h$ or $\widehat{c} = v$. Also, let R_{S_T}, R_{A_T}, R_{P_T} and ν_T be the* SPATIALRES, AMPLRES, PHASERES *and* FREQ *respectively of $\langle c, e\rangle_T$. Then*

$$R_{\mathrm{S}_{T+1}} = R_{\mathrm{A}_{T+1}} = R_{\mathrm{P}_{T+1}} = \nu_{T+1} = \infty\,.$$

Proof. For each complexity measure the statement is trivially true in the case that the measure is infinite at TIME T. We give a proof for the other case where each measure is finite at TIME T.

The statement is proved for the measure in question if there is no finite minimum value for that measure at TIME $T+1$. We use any (rectangular) step image, such as

$$a(x,y) = \begin{cases} \frac{1}{R_{\mathrm{A}_T}}, & \text{if } \frac{1}{2} - \frac{1}{R_{\mathrm{S}_{xT}}} \leqslant x < \frac{1}{2} \text{ and } \frac{1}{2} - \frac{1}{R_{\mathrm{S}_{yT}}} \leqslant y < \frac{1}{2} \\ 0, & \text{otherwise.} \end{cases}$$

The values $R_{\mathrm{S}_{xT}}$ and $R_{\mathrm{S}_{yT}}$ are the spatial resolutions in the horizontal and vertical directions respectively. The image $a(x,y)$ is representable with finite SPATIALRES, AMPLRES, PHASERES and FREQ. However its (horizontal or vertical) Fourier spectrum is a sinc function containing an infinite number of spatially separated components and is therefore not representable by finite SPATIALRES nor FREQ. The amplitudes of the peaks in this Fourier spectrum monotonically decrease in value, never reaching zero, and therefore are not representable by finite AMPLRES.

Goodman and Silvestri [GS70] discuss a method of phase quantisation that is equivalent to PHASERES. They prove that phase quantisation in the Fourier domain will, in general, cause degradation in the resulting inverse FT. In particular they show that the step function above can not be perfectly reconstructed from its phase discretised FT, and thus we need infinite PHASERES to represent the FT of such a function. The reader is encouraged to consult [GS70] for details. □

All theorems in this section are a worst case analysis. If one believes that our operations and complexity measures are meaningful, then the previous theorem essentially tells us that applying standard complexity theory to analyse continuous FTs is pointless. Obviously the result is of little relevance to CSMs that do not use the FT, or only use approximations of the FT. Typically in optical setups it will be the case that at some point in the computation discretisations are introduced.

Theorem 3.2.2 *(h/v & DYRANGE) Let $\langle c, e\rangle_T$ be a configuration of CSM M at* TIME *T where either $\widehat{c} = h$ or $\widehat{c} = v$. Also, let R_{D_T} be the* DYRANGE *of $\langle c, e\rangle_T$. Then $R_{\mathrm{D}_{T+1}} = \infty$*

Proof. Take the constant image $a = 1$. The horizontal FT $h(a)$ has value 0 everywhere except at $x = 0$ where it is a δ function, for all y. Hence there is no finite minimum DYRANGE that bounds the value at $h(a(0, y))$.

A similar argument holds for v, the only difference being we get the δ function at $v(a(x, 0))$. □

Again, the statement of Theorem 3.2.2 is a worst case scenario. It is worthwhile noting the following. In one dimension, Rayleigh's (also called Parseval's) theorem [Bra78] states

$$\int_{-\infty}^{\infty} |a(x)|^2 \,\mathrm{d}x = \int_{-\infty}^{\infty} |\mathcal{F}(u)|^2 \,\mathrm{d}u \,. \tag{3.2.1}$$

Where $\mathcal{F}(u)$ is the FT of $a(x)$. For images, this is usually interpreted to mean that the amount of energy in an image is equal to the amount of energy in its FT.

From the definition of h (see Eq. (2.2.1)) each horizontal line in $h(a)$ is the 1D FT of the corresponding horizontal line in the input image a. Then, according to Eq. (3.2.1), for a given y-value both horizontal lines have the same total energy. Hence the integral (sum) of the energy of all lines in $h(a)$ will equal that in a. From a complexity analysis of a given CSM instance, lets suppose we know that $h(a)$ has finite SPATIALRES. Then we know that we cannot get a true δ function after a FT. Hence in such cases Parseval's theorem will allow us to specify a finite upper bound on the DYRANGE of $h(a)$ in terms of the complexity of a.

Theorem 3.2.4 below shows that executing a single $*$ operation does not change the PHASERES of a CSM computation. To show this we will use the following lemma that recalls the phase discretisation function P.

Lemma 3.2.3 *Let $z \in \mathbb{C}$, $\mu \in \{1, 2, 3, \ldots\}$ and (from Eq. (2.3.2)) let*

$$P(z, \mu) = |z| \exp\left(\mathrm{i} \left\lfloor \arg(z)\frac{\mu}{2\pi} + \frac{1}{2} \right\rfloor \frac{2\pi}{\mu}\right) \,. \tag{3.2.2}$$

Then

$$P(z, \mu) \in \left\{ z' \;\middle|\; z' = |z| \exp\left(\mathrm{i}\mu' \frac{2\pi}{\mu}\right), \mu' \in \{1, 2, \ldots, \mu\} \right\} \,.$$

Proof. Let $j = \left\lfloor \arg(z)\frac{\mu}{2\pi} + \frac{1}{2} \right\rfloor$, hence $j \in \mathbb{Z}$. Let $\arg(z)$ have range $0 < \arg(z) \leqslant 2\pi$. By substituting for $\arg(z)$ in j it is clear that $j \in \mu' \cup \{0\}$. Since we are working in radians $j = 0$ gives the same value in P as $j = \mu$ in Eq. (3.2.2), hence $j \in \mu'$ and the proof is complete. □

The essence of the following theorem is that multiplying the phase angle by -1 (that is, complex conjugation) does not increase PHASERES.

Theorem 3.2.4 *($*$ & PHASERES) Let $\langle c, e\rangle_T$ be a configuration of CSM M at* TIME *T where $\widehat{c} = *$. Also, let R_{P_T} be the* PHASERES *of $\langle c, e\rangle_T$. Then $R_{\mathrm{P}_{T+1}} = R_{\mathrm{P}_T}$.*

Proof. The statement is trivially true in the case that PHASERES is infinite. We give a proof for the other case where PHASERES is finite.

From the definition of the $*$ operation (Definition 2.2.4), the only image affected by $*$ is a. We will show that in the worst case the set of phase values in range(a) at TIME $T + 1$ is a subset of the set of phase values in range(a) at TIME T. Let $z \in \text{range}(a(x, y))$, we write $z = |z| \exp(\mathrm{i} \arg(z))$ and by definition, $*(z) = |z| \exp(\mathrm{i}(-\arg(z)))$.

Let $R_{\mathrm{P}_T} = \mu$, then at time T for each $z \in \text{range}(a)$

$$P(z, \mu) = |z| \exp\left(\mathrm{i} \left\lfloor \arg(z)\frac{\mu}{2\pi} + \frac{1}{2}\right\rfloor \frac{2\pi}{\mu}\right)$$

where P was defined in Eq. (2.3.2). By Lemma 3.2.3

$$P(z, \mu) \in \left\{ z' \;\middle|\; z' = |z| \exp\left(\mathrm{i}\mu'\frac{2\pi}{\mu}\right), \mu' \in \{1, 2, \ldots, \mu\} \right\}. \tag{3.2.3}$$

By definition, after the $*$ operation

$$*(P(z, \mu)) = |z| \exp\left(\mathrm{i}\left(-\left\lfloor \arg(z)\frac{\mu}{2\pi} + \frac{1}{2}\right\rfloor \frac{2\pi}{\mu}\right)\right).$$

We then substitute into Eq. (3.2.3) to get

$$*(P(z, \mu)) \in \left\{ z'' \;\middle|\; z'' = |z| \exp\left(\mathrm{i}\left(-\mu'\frac{2\pi}{\mu}\right)\right), \mu' \in \{1, 2, \ldots, \mu\} \right\}. \tag{3.2.4}$$

Our notation is in radians hence $n\mu'\frac{2\pi}{\mu}$, for all $n \in \mathbb{Z}$, is an element of our set of μ angles, in this case $n = -1$. Hence Eqs. (3.2.3) and (3.2.4) are equivalent. □

Theorem 3.2.5 *($\cdot$ & AMPLRES) Let $\langle c, e\rangle_T$ be a configuration of CSM M at* TIME *T where $\widehat{c} = \cdot$, and R_{A_T} be the* AMPLRES *of $\langle c, e\rangle_T$. Then $R_{\mathrm{A}_{T+1}} = (R_{\mathrm{A}_T})^2$.*

Proof. The statement is trivially true in the case that AMPLRES is infinite. We give a proof for the other case where AMPLRES is finite.

By definition, the operation $\cdot$ acts on images a and b and the result is placed in image a. All images other than a are unaffected so we ignore them. For any $x, y \in [0, 1)$, let $z_a = \text{range}(a(x, y))$, $z_b = \text{range}(b(x, y))$. At TIME $T + 1$ and coordinates (x, y), and from the definition of $\cdot$, recall that image a is replaced with the new image

$$a'(x, y) = z_a z_b = |z_a|\,|z_b| \exp(\mathrm{i}(\arg a(x, y) + \arg b(x, y)))$$

Let $R_{\mathrm{A}_T} = \mu$ in Eq. (2.3.1), the values in a and b at TIME T are of the form

$$A(z,\mu) = \left\lfloor |z|\,\mu + \frac{1}{2} \right\rfloor \frac{1}{\mu} \exp(\mathrm{i} \times \arg(z))\,.$$

Since we are interested only in AMPLRES we will ignore the phase term.

$$|A(z,\mu)| = \left\lfloor |z|\,\mu + \frac{1}{2} \right\rfloor \frac{1}{\mu}$$

at TIME T. Then, at TIME $T+1$

$$\begin{aligned} |A(z_a,\mu)|\,|A(z_b,\mu)| &= \left(\left\lfloor |z_a|\,\mu + \frac{1}{2} \right\rfloor \frac{1}{\mu} \right) \left(\left\lfloor |z_b|\,\mu + \frac{1}{2} \right\rfloor \frac{1}{\mu} \right) \\ &= \left\lfloor |z_a|\,\mu + \frac{1}{2} \right\rfloor \left\lfloor |z_b|\,\mu + \frac{1}{2} \right\rfloor \frac{1}{\mu^2}\,. \end{aligned}$$

We are proving the theorem for the case that AMPLRES is finite, hence we know that at TIME T, $|z_a|$ and $|z_b|$ are rationals, moreover they are of the form $|z_a| = \frac{n}{\mu}$ and $|z_b| = \frac{m}{\mu}$ for some $n, m \in \mathbb{N}$. By substitution we simplify the above expression to get

$$|A(z_a,\mu)|\,|A(z_b,\mu)| = \frac{n\,m}{\mu^2}$$

In the worst case we require AMPLRES $\mu^2 = (R_{\mathrm{A}_T})^2$ to represent the values in image a at TIME $T+1$. □

Theorem 3.2.6 *($\cdot$ & DYRANGE) Let $\langle c, e\rangle_T$ be a configuration of CSM M at TIME T where $\widehat{c} = \cdot$, and let R_{D_T} be the DYRANGE of $\langle c, e\rangle_T$. Then $R_{\mathrm{D}_{T+1}} = (R_{\mathrm{D}_T})^2$.*

Proof. The statement is trivially true in the case that DYRANGE is infinite. We give a proof for the other case where DYRANGE is finite.

By definition $\cdot$ acts on images a and b and the result affects only a. For any $x, y \in [0, 1)$ let $z_a \in \mathrm{range}(a(x,y))$, $z_b \in \mathrm{range}(b(x,y))$. At TIME $T+1$ and coordinates (x,y), and from the definition of $\cdot$, recall that image a is replaced with the new image

$$a'(x,y) = z_a z_b = |z_a|\,|z_b| \exp(\mathrm{i}(\arg a(x,y) + \arg b(x,y)))$$

Since DYRANGE is defined in terms of amplitude we will ignore the phase term.

$$|a'(x,y)| = |z_a z_b| = |z_a|\,|z_b|$$

If $|z_a| = |z_b| = R_{\mathrm{D}_T}$ then at TIME $T+1$ we get the worst case of

$$R_{\mathrm{D}_{T+1}} = |a(x,y)| = (R_{\mathrm{D}_T})^2$$

It is easy to see that for all other values of $|z_a|$ and $|z_b|$, $R_{\mathrm{D}_{T+1}} < (R_{\mathrm{D}_T})^2$.$\square$

Unlike the case for AMPLRES, PHASERES is not affected by image multiplication as the next theorem shows.

Theorem 3.2.7 *($\cdot$ & PHASERES) Let $\langle c, e\rangle_T$ be a configuration of CSM M at* TIME *T where $\widehat{c} = \cdot$, and let R_{P_T} be the* PHASERES *of $\langle c, e\rangle_T$. Then $R_{\mathrm{P}_{T+1}} = R_{\mathrm{P}_T}$.*

Proof. The statement is trivially true in the case that PHASERES is infinite. We prove the statement for the other case where PHASERES is finite.

By definition $\cdot$ acts on images a and b and the result affects only a. We will show that the set of possible phase values in range(a) at time $T+1$ is a subset of the set of possible phase values in range(a) $\cup$ range(b) at time T. For some (x, y) let $z_a = a(x, y)$ and $z_b = b(x, y)$. By definition

$$\cdot(z_a, z_b) = |z_a|\,|z_b| \exp\left(\mathrm{i}(\arg z_a + \arg z_b)\right) .$$

Since we are proving the theorem for the case that PHASERES is finite, we know (from Eq. (2.3.2)) that at TIME T, $\arg z_a = \frac{n}{R_{\mathrm{P}_T}} 2\pi$ and $\arg z_b = \frac{m}{R_{\mathrm{P}_T}} 2\pi$, for some $n, m \in \mathbb{N}$. This gives

$$\cdot(z_a, z_b) = |z_a|\,|z_b| \exp\left(\mathrm{i}\left(\frac{n+m}{R_{\mathrm{P}_T}}\right) 2\pi\right)$$

which is a member of the set of *possible* phase values in range(a) $\cup$ range(b) at TIME T. $\square$

Theorem 3.2.8 *($+$ & AMPLRES) Let $\langle c, e\rangle_T$ be a configuration of CSM M at* TIME *T where $\widehat{c} = +$. Also, let R_{A_T} be the* AMPLRES *of $\langle c, e\rangle_T$. Then $R_{\mathrm{A}_{T+1}} = \infty$.*

Proof. Suppose the AMPLRES and PHASERES at time T are $R_{\mathrm{A}_T} = 1$ and $R_{\mathrm{P}_T} = 4$ respectively. Also let a be the constant image $a(x, y) = \mathrm{i} = e^{\mathrm{i}\frac{1}{2}\pi}$ and b be the constant image $b(x, y) = 1 = e^0$. After the $+$ operation, at time $T+1$, image a has value $a(x, y) = 1 + \mathrm{i} = \sqrt{2}e^{\mathrm{i}\frac{1}{4}\pi}$. The CSM requires ∞ AMPLRES to represent the amplitude value $\sqrt{2}$. $\square$

If we restrict PHASERES to be 1 or 2 then we don't meet the worst case scenario described in the previous theorem, we introduce this restriction in Section 3.3.

Theorem 3.2.9 *($+$ & DYRANGE) Let $\langle c, e\rangle_T$ be a configuration of CSM M at* TIME *T where $\widehat{c} = +$. Also, let R_{D_T} be the* DYRANGE *of $\langle c, e\rangle_T$. Then $R_{\mathrm{D}_{T+1}} = 2R_{\mathrm{D}_T}$*

Proof. The operation $+$ has no effect on SPATIALRES hence without loss of generality we can assume that a and b are everywhere constant images, $a(x,y) = z_a = r_a e^{\mathrm{i}\Theta_a 2\pi}$ and $b(x,y) = z_b = r_b e^{\mathrm{i}\Theta_b 2\pi}$.

Suppose $z_a = z_b$ and $|z_a| = R_{\mathrm{D}_T}$. In this case $z_a + z_b = 2z_a = 2r_a e^{\mathrm{i}\Theta_a 2\pi}$ and hence $R_{\mathrm{D}_{T+1}} = 2R_{\mathrm{D}_T}$. In fact this is the worst case since adding any pair of complex values that lie on the origin-centred disk of radius R_{D_T} gives a new complex value on the origin-centred disk of radius $2R_{\mathrm{D}_T}$. □

Theorem 3.2.10 *(+ & PHASERES) Let $\langle c, e\rangle_T$ be a configuration of CSM M at TIME T where $\widehat{c} = +$. Also, let R_{P_T} be the PHASERES of $\langle c, e\rangle_T$. Then $R_{\mathrm{P}_{T+1}} = \infty$.*

Proof. The statement is trivially true in the case that PHASERES is infinite. We prove the statement for the other case where PHASERES is finite.

The operation $+$ has no effect on SPATIALRES hence without loss of generality we can assume that a and b are everywhere constant images.

Let $a(x,y) = 2$ and $b(x,y) = \mathrm{i}$. After the $+$ operation, at time $T+1$, image a has value

$$\sqrt{5}\exp\left(\mathrm{i}\tan^{-1}\left(\frac{1}{2}\right)\right) .$$

Niven [Niv56, Corollary 3.12] shows that

$$\frac{\tan^{-1}\left(\frac{1}{2}\right)}{\pi}$$

is irrational. Given this, we require infinite PHASERES for addition. □

Theorem 3.2.11 *(st/ld/ρ & GRID) Let $\langle c, e\rangle_T$ be a configuration of CSM M at TIME T where either $\widehat{c} = st$, $\widehat{c} = ld$ or $\widehat{c} = \rho$. Also, let G_T be the GRID of $\langle c, e\rangle_T$. Then there is no upper bound on the value of G_{T+1}.*

Proof. From Definition 2.2.2 the address decoding function $\mathfrak{E}^{-1} : \mathcal{N} \to \mathbb{N}$ is Turing machine decidable. This is the only restriction on $\mathfrak{E}^{-1}$. For an arbitrary M there is no upper bound on the rate of growth of its $\mathfrak{E}^{-1}$ and so there is no bound on the value of the natural number that an address parameter of *st* maps to. Hence after a *st* operation we cannot bound GRID in terms of G_T, or any of the other complexity measures. The same argument holds for *ld* and ρ. □

The previous theorem highlights the caveat of *reasonableness* in the definition of $\mathfrak{E}$. When we defined $\mathfrak{E}$ we did not wish to restrict the CSM programmer from coming up with a novel $\mathfrak{E}$ suited to her needs. However, for reasonable addressing functions we would expect the growth rate of $\mathfrak{E}^{-1}$, with respect to the ordering on $\mathcal{N}$, to be reasonable. For example, in Section 3.3 we restrict $\mathfrak{E}$ to being logspace Turing machine computable, which is an agreed notion of reasonableness in parallel complexity theory. As one can imagine, a complicated $\mathfrak{E}$ will leave lots of headaches for the optical engineer who has to implement it. Not only that, we would also have an incomplete analysis of the complexity of the CSM in question (unless of course we work the growth rate of $\mathfrak{E}$ into our complexity analysis). The same remark applies to the next theorem.

Theorem 3.2.12 *(ld & SPATIALRES/FREQ) Let $\langle c, e\rangle_T$ be a configuration of CSM M at* TIME *T where $\widehat{c} = ld$. Also, let R_{S_T} and ν_T be the* SPATIALRES *and* FREQ *respectively of $\langle c, e\rangle_T$. Then there is no upper bound on the value of $R_{\mathrm{S}_{T+1}}$ nor ν_{T+1}.*

Proof. The SPATIALRES of an image is the product of its horizontal SPATIALRES and its vertical SPATIALRES: $R_{\mathrm{S}_T} = R_{\mathrm{S}_{x_T}} R_{\mathrm{S}_{y_T}}$. After a *ld* operation, with the *ld* parameters $\xi_1, \xi_2, \eta_1, \eta_2$, image a has SPATIALRES

$$\begin{aligned} R_{\mathrm{S}_{T+1}} =& (\xi_2 - \xi_1 + 1) R_{\mathrm{S}_{x_T}} (\eta_2 - \eta_1 + 1) R_{\mathrm{S}_{y_T}} \\ =& R_{\mathrm{S}_T} (\xi_2 - \xi_1 + 1)(\eta_2 - \eta_1 + 1) \end{aligned} \tag{3.2.5}$$

From Theorem 3.2.11 there is no upper bound on the growth of $\mathfrak{E}^{-1}$. Hence there is no upper bound on the natural number *ld* parameters above. After a *st* operation there is no upper bound on SPATIALRES in terms of R_{S_T}, or any of the other complexity measures. Analogously, for FREQ the upper bound is in terms of $\mathfrak{E}^{-1}$ rather than any of the complexity measures. □

If we have agreed upon a reasonable (bound on) $\mathfrak{E}$, then Eq. (3.2.5) enables us to specify an upper bound on SPATIALRES and FREQ at TIME $T + 1$.

Even though *br* has address parameters, the previous arguments do not apply.

Theorem 3.2.13 *(br & GRID) Let $\langle c, e\rangle_T$ be a configuration of CSM M at* TIME *T where $\widehat{c} = br$. Also, let G_T be the* GRID *of $\langle c, e\rangle_T$. Then $G_{T+1} = G_T$.*

Proof. From the definition of a CSM configuration (Definition 2.2.3) the control must always be inside the initial (TIME 1) grid. Hence branching outside the current grid will always result in an undefined computation. Since in a valid computation *br* address parameters must always be inside the current grid, the *br* operation never increases the GRID complexity. □

3.3 $\mathcal{C}_2$-CSM

Motivated by the results in Table 3.2.1, we define a restricted class of the CSM model, denoted $\mathcal{C}_2$-CSM.

Definition 3.3.1 ($\mathcal{C}_2$-CSM) *A $\mathcal{C}_2$-CSM is a CSM whose computation* TIME *is defined for $t \in \{1, 2, \ldots, T(n)\}$ and has the following restrictions:*

- *For all* TIME *t both* AMPLRES *and* PHASERES *have constant value of 2.*
- *For all* TIME *t each of* GRID*,* SPATIALRES *and* DYRANGE *is $O(2^t)$, and* SPACE *is redefined to be the product of all complexity measures except* TIME *and* FREQ*.*
- *Operations h and v compute the discrete FT (DFT) in the horizontal and vertical directions respectively.*
- *Given some* reasonable *binary word representation of the set of addresses $\mathcal{N}$, the address encoding function $\mathfrak{E} : \mathbb{N} \to \mathcal{N}$ is decidable by a logspace Turing machine.*

We give some remarks on this definition. We have replaced the FT with the DFT [Bra78, Wea89]. FREQ is now solely dependent on SPATIALRES (rescaling the Fourier spectrum by changing FREQ is no longer necessary); hence FREQ is not an interesting complexity measure for $\mathcal{C}_2$-CSMs. Because of this we do not analyse $\mathcal{C}_2$-CSMs in terms of FREQ complexity.

Given that in a $\mathcal{C}_2$-CSM AMPLRES and PHASERES are both 2, SPACE is now

$$2 \cdot 2 \cdot G_t \cdot R_{\mathrm{S}_t} \cdot R_{\mathrm{D}_t} = O(2^{3t})\,.$$

where G_t, R_{S_t} and R_{D_t} are GRID, SPATIALRES and DYRANGE at TIME t. The SPACE restriction is not unique to our model, other instances of such restrictions can be found in the literature [Par87, Gol82].

Due to the AMPLRES and PHASERES restrictions, the inputs and outputs to the DFT must be from the set $\{0, \pm\frac{1}{2}, \pm 1, \pm\frac{3}{2}, \ldots\}$. This set is not closed under the DFT (e.g. due to the DFT normalisation factor). The programmer should always ensure that the DFT only operates on values that lead to output values in the above set. This remark does not affect the results in this work.

In Section 2.2.2 we stated that address encoding and decoding functions should be Turing machine computable. Here we strengthen this condition, we force $\mathfrak{E}$ to be more reasonable. At first glance sequential logspace computability may seem like a strong restriction, but in fact it is quite weak. From an optical implementation point of view it should be the case that $\mathfrak{E}$ is as simple as possible, otherwise we cannot assume fast addressing. Other (sequential and parallel) models usually have a very restricted 'addressing function': in most cases it is simply the identity function on $\mathbb{N}$ (or maps $\mathbb{N}$ to its standard binary word encoding). In this context an addressing function corresponds to the addresses in a RAM, or the connection pattern [Gol78] for a network of processors. Without an explicit or implicit restriction on the computational complexity of $\mathfrak{E}$, finding non-trivial upper bounds on the power of $\mathcal{C}_2$-CSMs is impossible. For example $\mathfrak{E}$ could encode an arbitrarily complex halting Turing machine.

As a (possibly) weaker restriction we could, for example, give a specific $\mathfrak{C}$. However, this prohibits the programmer from developing her own novel, and reasonable, schemes.

In Chapters 5 and 6 we will prove that the $\mathcal{C}_2$-CSM verifies the parallel computation thesis.[1]

3.4 Discussion

We have analysed the growth of the CSM complexity measures with respect to the CSM operations over TIME. Table 3.2.1 shows that many variations on the CSM can not be analysed if we restrict ourselves to the standard tools from complexity theory. The table gives a starting point for developing restrictions of the model. In particular it motivated the definition of the $\mathcal{C}_2$-CSMs, a restricted class of CSMs.

It should be noted that the results in this chapter are independent of any particular data representations or program restrictions. If we restrict ourselves to certain (continuous or discrete) data representations then clearly we change the properties of CSM computations and we can reduce the upper bounds on complexity growth. Another way to restrict the model would be to place restrictions on the syntactic structure of programs, as was done with the vector machine model [PS76].

Table 3.2.1 describes growth in complexity if inputs are finite. The irrational values that give rise to the infinities in Table 3.2.1 are computable reals in the sense of computable analysis [Wei00]. It would be interesting to analyse this aspect of the model by making use of results from (say) the framework of real recursive function theory [Moo96, Cam01, dSG02, BH04a] or other approaches to analog or real computation [BCSS97, Wei00], from the point of view of parallel complexity theory.

The results we prove in this chapter are not only interesting from a computational complexity viewpoint, but from the physical viewpoint also. For example Goodman [GS70] studies PHASERES in the same way we do, and is mostly motivated by the practical concerns of reconstructing digital holograms.

The $\mathcal{C}_2$-CSM is more realistic than the CSM in terms of optical implementation; many current optical information processing devices are pixellated (liquid-crystal display SLMs and digital cameras) and operate over a finite set of grey levels. Positive and negative rationals are routinely represented in optical architectures [NJK$^+$99, Van92, AS92, McA91, Fei88]. Clearly, the SPACE limitation also decreases the difficulty of implementation.

[1]Our notation "$\mathcal{C}_2$-CSM" is inspired by the fact that models that verify the parallel computation thesis are said to be members of the second machine class [vEB90], denoted $\mathcal{C}_2$ (for example in [Sos03]).

4

2D data and programs

4.1 Introduction

In this chapter we define some useful data representations for the CSM. We give representations for words, numbers and matrices as images. Using these representations we define what it means to decide a language by CSM. In Section 4.4 we introduce a programming language for the CSM. This language is an abstraction of the low-level definition given in Chapter 2. Our high-level language facilitates the writing of CSM programs that are easier to read and write, without sacrificing ease of complexity analysis.

4.2 Representing data as images

There are many ways to represent elements of finite, countable and uncountable sets as images. We define six useful representations, these are illustrated in Figure 4.2.1. The following definition gives the image representation of the symbol 1 as an image having value 1 at its centre and value 0 everywhere else. The symbol 0 is represented by the image that has value 0 everywhere.

Definition 4.2.1 (Binary symbol image) *The symbol $\psi \in \{0,1\}$ has the binary symbol image representation f_ψ,*

$$f_\psi(x,y) = \begin{cases} 1, & \text{if } x, y = 0.5, \psi = 1 \\ 0, & \text{otherwise}. \end{cases}$$

We extend this representation scheme to binary words using 'list' and 'stack' images.

Definition 4.2.2 (Binary list image) *The word $w = w_1 w_2 \cdots w_k \in \{0,1\}^+$ has the binary list image representation f_w,*

$$f_w(x,y) = \begin{cases} 1, & \text{if } x = \frac{2i-1}{2k}, y = 0.5, w_i = 1 \\ 0, & \text{otherwise}, \end{cases}$$

where $w_i \in \{0,1\}, 1 \leqslant i \leqslant k$. Image f_w is said to have length k and the pair (f_w, k) uniquely represents w.

The binary list image representation requires $\Theta(k)$ SPATIALRES and FREQ. Only 1 AMPLRES, PHASERES and DYRANGE are needed. Often we wish to access a particular subword in a list image. To do this we simply store (or rescale) the single list image across many images and we can then address the images we want. In general we require linear GRID for such rescaling.

Definition 4.2.3 (Binary stack image) *The word $w = w_1 w_2 \cdots w_k \in \{0,1\}^+$ has the binary stack image representation f_w,*

$$f_w(x,y) = \begin{cases} 1, & \textit{if } x = 1 - \frac{3}{2^{k-i+2}}, y = 0.5, w_i = 1 \\ 0, & \textit{otherwise}\,, \end{cases}$$

where $w_i \in \{0,1\}, 1 \leqslant i \leqslant k$. Image f_w is said to have length k and the pair (f_w, k) uniquely represents w.

The binary stack image representation requires $\Theta(2^k)$ SPATIALRES and FREQ. Only 1 AMPLRES, PHASERES and DYRANGE are needed. To access the i^{th} bit in a stack image we 'pop' the stack i times. Popping involves spreading the stack over two horizontally adjacent images, the leftmost image now contains only the topmost stack element. Despite the fact that stacks require exponential SPATIALRES they are nevertheless useful when we wish to use at most constant GRID.

If the alphabet is $\{1\}$ (instead of $\{0,1\}$) we replace the word "binary" with the word "unary" in Definitions 4.2.1, 4.2.2 and 4.2.3. In Definitions 4.2.2 and 4.2.3 each unary/binary symbol in w is represented by a corresponding value of 0 or 1 in f_w. Notice that in the unary/binary stack image, w's leftmost symbol w_1, is represented by the rightmost value in the sequence of values representing w in f_w, this means that w_k is represented by the topmost stack element.

We represent a single natural, integer, real or complex value r by an image with a single peak of value r.

Definition 4.2.4 (Number image) *The number r has the number image representation f_r,*

$$f_r(x,y) = \begin{cases} r, & \textit{if } x, y = 0.5 \\ 0, & \textit{otherwise}\,. \end{cases}$$

The complexity of this representation varies with the type of number. All number images use 1 SPATIALRES, constant FREQ and $|r|$ DYRANGE. Natural number images use 1 AMPLRES and 1 PHASERES. Integer images use 1 AMPLRES and 2 PHASERES. Real number images use ∞ AMPLRES and 1 PHASERES. Finally, complex number images use ∞ AMPLRES and ∞ PHASERES.

To represent a $R \times C$ matrix of real values we define RC peaks that represent the matrix values and use both dimensions of a stack-like or list-like image.

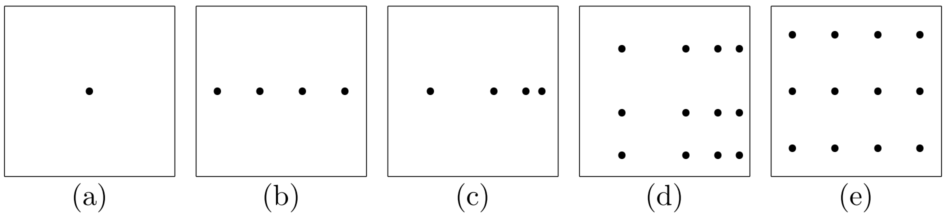
Figure 4.2.1: Representing data by spatially separated peaks. The (possibly) nonzero peaks are coloured black and the white areas denote value zero. (a) Binary symbol image and number image, (b) list image, (c) stack image, (d) stack-matrix image, (e) list-matrix image.

Definition 4.2.5 ($R \times C$ stack-matrix image) *The $R \times C$ matrix A, with real components $a_{ij}, 1 \leqslant i \leqslant R, 1 \leqslant j \leqslant C$, has the $R \times C$ stack-matrix image representation f_A,*

$$f_A(x,y) = \begin{cases} a_{ij}, & \text{if } x = 1 - \frac{1+2k}{2^{j+k}}, y = \frac{1+2l}{2^{i+l}} \\ 0, & \text{otherwise}, \end{cases}$$

where

$$k = \begin{cases} 1, & \text{if } j < C \\ 0, & \text{if } j = C \end{cases}, \qquad l = \begin{cases} 1, & \text{if } i < R \\ 0, & \text{if } i = R \end{cases}.$$

This stack-matrix image representation is illustrated in Figure 4.2.1(d) for $R = 3$ and $C = 4$. This representation requires $O(2^{R+C})$ spatialRes and freq. Since we are representing real numbers the representation uses ∞ amplRes. Only 1 amplRes, phaseRes and dyRange are needed. The stack-matrix image representation can be manipulated using image rescaling not only in the horizontal direction (as with the stack above), but also in the vertical direction. We also define a list-matrix image that is more spatialRes efficient.

Definition 4.2.6 ($R \times C$ list-matrix image) *The $R \times C$ matrix A, with real components $a_{ij}, 1 \leqslant i \leqslant R, 1 \leqslant j \leqslant C$, has the $R \times C$ list-matrix image representation f_A,*

$$f_A(x,y) = \begin{cases} a_{ij}, & \text{if } x = \frac{2i-1}{2C}, y = \frac{2j-1}{2R} \\ 0, & \text{otherwise}, \end{cases}$$

The list-matrix image representation is illustrated in Figure 4.2.1(e), once again $R = 3$ and $C = 4$. The complexity of the list-matrix representation is the same as the stack-matrix representation except it requires only $O(RC)$ spatialRes and freq.

4.3 Language deciding by CSM

Definition 4.3.1 (Language deciding; stack images) *CSM M_L decides $L \subseteq \{0,1\}^*$ if M_L has initial configuration $\langle c_{sta}, e_{sta}\rangle$ and final configuration $\langle c_{hlt}, e_{hlt}\rangle$, and the following hold:*

- *sequence e_{sta} contains the two input elements $(f_w, \iota_{1_\xi}, \iota_{1_\eta})$ and $(f_{1^{|w|}}, \iota_{2_\xi}, \iota_{2_\eta})$*
- *e_{hlt} contains the output element $(f_1, o_{1_\xi}, o_{1_\eta})$ if $w \in L$*
- *e_{hlt} contains the output element $(f_0, o_{1_\xi}, o_{1_\eta})$ if $w \notin L$*
- *$\langle c_{sta}, e_{sta}\rangle \vdash_M^* \langle c_{hlt}, e_{hlt}\rangle$, for all $w \in \{0,1\}^*$,*

where f_w is the binary stack image representation of $w \in \{0,1\}^*$, $f_{1^{|w|}}$ is the unary stack image representation of the unary word $1^{|w|}$. Images f_0 and f_1 are the binary symbol image representations of the symbols 0 and 1, respectively.

In this definition addresses $(\iota_{1_\xi}, \iota_{1_\eta})$, $(\iota_{2_\xi}, \iota_{2_\eta}) \in I$ and address $(o_{1_\xi}, o_{1_\eta}) \in O$, where I and O are as given in Definition 2.2.2. We use the stack image representation of words. The unary input word $1^{|w|}$ is sufficient for M_L to determine the length of input word w. (For example the binary stack image representations of the words 00 and 000 are identical.) This definition has the drawback of using stacks which have exponential SPATIALRES in input word length. The next definition is identical in all respects except that it uses natural number images instead of stacks, thus swapping exponential SPATIALRES for linear DYRANGE.

Definition 4.3.2 (Language deciding; list & natural number images) *CSM M_L decides $L \subseteq \{0,1\}^*$ if M_L has initial configuration $\langle c_{sta}, e_{sta}\rangle$ and final configuration $\langle c_{hlt}, e_{hlt}\rangle$, and the following hold:*

- *sequence e_{sta} contains the two input elements $(f_w, \iota_{1_\xi}, \iota_{1_\eta})$ and $(f_n, \iota_{2_\xi}, \iota_{2_\eta})$*
- *e_{hlt} contains the output element $(f_1, o_{1_\xi}, o_{1_\eta})$ if $w \in L$*
- *e_{hlt} contains the output element $(f_0, o_{1_\xi}, o_{1_\eta})$ if $w \notin L$*
- *$\langle c_{sta}, e_{sta}\rangle \vdash_M^* \langle c_{hlt}, e_{hlt}\rangle$, for all $w \in \{0,1\}^*$,*

where f_w is the binary list image representation of $w \in \{0,1\}^*$, f_n is the natural number image representation of $n = |w|$. Images f_0 and f_1 are the binary symbol image representations of the symbols 0 and 1, respectively.

4.4 CSM programming language

Reading and writing CSM programs using the low-level syntax given in Chapter 2 can be cumbersome and time consuming. In this section we define a higher-level language to simplify reasoning about the model. The syntax we use removes much of the technical detail found in low-level CSM programs, thus shortening and simplifying them.

Why do we have two levels of abstraction? The initial definition of the model has the advantage of being low-level enough to be somewhat closer to the reality of optical implementations than the high-level programming language we will give here. The higher-level programming language has the obvious advantage of being more 'friendly'.[1] This language abstracts away from some details of the lower-level definition, but as we will show does not hide any more than a small constant factor overhead in resource usage, so we can still provide meaningful complexity analysis. Before continuing, if the reader wants to get a feel for the language she may glance ahead to Section 4.5 to see some example code.

4.4.1 Programming language syntax

We define the programming language syntax with the context free grammer in Figure 4.4.1. In this grammer ' | ' denotes 'or', ':' denotes 'rewrites as' and 'ϵ' is the empty word. Additionally, nonterminals have an uppercase initial letter and are written in `Typewriter font`, all other words and characters are terminals.

As is defined in Figure 4.4.1, a program is a collection of user defined functions. User defined functions consist of a function header and a list of statements. A function header consists of a function name and a comma separated list of parameters, each parameter being an image address. A single semicolon appears instead of one of the commas, for readability purposes only: As a convention we place read-only image addresses on the left of the semicolon and read/write image addresses on the right. A statement is either a basic operation, an **if** or **while** control flow operation, or a call to a user defined function.

Statements refer to single images either explicitly by using a CSM address (e.g. [2,2,3,3]) or implicitly by using a variable name. When referring to images explicitly it is possible that `Img` defines a rectangle of images (e.g. [2,6,13,18]). When calculating SPATIALRES of programs we need to be aware that the underlying CSM architecture would rescale any such rectangle of images to a single image before performing an operation on them. We will return to this point later.

[1] These friends are merely symbolic.

$$\begin{array}{rcl}
\texttt{Program} & : & \texttt{Program UserDefinedFn} \mid \texttt{UserDefinedFn} \\
\texttt{UserDefinedFn} & : & \texttt{Name(ImgList; ImgList) StatementList}\ \textbf{end} \\
\texttt{StatementList} & : & \texttt{StatementList Statement} \mid \epsilon \\
\texttt{Statement} & : & \texttt{BasicOp} \mid \texttt{ControlFlow} \mid \texttt{FnCall} \\
\texttt{BasicOp} & : & \mathrm{h}(\texttt{Img};\texttt{Img}) \mid \mathrm{v}(\texttt{Img};\texttt{Img}) \mid *(\texttt{Img};\texttt{Img}) \\
 & & \mid \rho\,(\texttt{Img},\texttt{Img},\texttt{Img};\texttt{Img}) \mid \cdot(\texttt{Img},\texttt{Img};\texttt{Img}) \\
 & & \mid +(\texttt{Img},\texttt{Img};\texttt{Img}) \mid \texttt{Img} \leftarrow \texttt{Img} \\
\texttt{ControlFlow} & : & \texttt{IfElse} \mid \texttt{WhileLoop} \\
\texttt{FnCall} & : & \texttt{Name(ImgList;ImgList)} \\
\texttt{IfElse} & : & \textbf{if}\ \texttt{Condition}\ \textbf{then}\ \texttt{StatementList} \\
 & & \textbf{else}\ \texttt{StatementList}\ \textbf{end if} \\
\texttt{WhileLoop} & : & \textbf{while}\ \texttt{Condition}\ \textbf{do}\ \texttt{StatementList}\ \textbf{end while} \\
\texttt{Condition} & : & (\texttt{Img} == \texttt{Img}) \\
\texttt{Name} & : & \texttt{NameName} \mid \mathrm{a} \mid \mathrm{b} \mid \cdots \mid \mathrm{z} \mid \mathrm{A} \mid \mathrm{B} \mid \cdots \mid \mathrm{Z} \mid 0 \mid 1 \mid \cdots \mid 9 \\
\texttt{ImgList} & : & \texttt{Img},\texttt{ImgList} \mid \texttt{Img} \mid \epsilon \\
\texttt{Img} & : & [\,\texttt{D},\texttt{D},\texttt{D},\texttt{D}\,] \mid \texttt{Name} \\
\texttt{D} & : & \texttt{D}\,\texttt{D} \mid 0 \mid 1 \mid \cdots \mid 9
\end{array}$$

Figure 4.4.1: CSM programming language syntax.

4.4.2 Programming language operational semantics

The semantics of the basic elements of our programming language are defined by the the code fragments in Figure 4.4.2. Each basic element of the language is defined in terms of low-level CSM code and is explained further below.

A 'system stack' is used to store temporary information during the execution of CSM programs, the need for such a stack becomes apparent when we define control flow structures and functions below. There are a number of ways to implement the system stack using images. These implementations will vary in the type and amount of resources they require. In order to facilitate a straightforward complexity analysis of stacks, in this section we assume that stack *elements* have constant complexity. In general this will not be the case. However for a given system stack structure only the stack elements themselves will vary in complexity. Since each element will be an explicit image in our program a complexity analysis of a program will take into account the complexity of underlying system stack elements.

A possible stack implementation would be to use a single image; to 'push' image a onto the stack we place the stack image and image a side by side and rescale to one image. Then to 'pop' image a from the stack we simply rescale the stack image over two juxtaposed images. This implementation has the advantage of using only constant GRID, DYRANGE, AMPLRES, PHASERES and FREQ for push, pop and stack storage of n stack elements, but has the disadvantage that $O(2^n)$ SPATIALRES is required. Another possible implementation uses a single image to store each stack element. For this to work for arbitrary length stacks we could reserve an entire row of the CSM grid for the stack. The leftmost image on the row would be the bottom stack element and a counter image would keep track of the address of the current top element. If addresses are implemented as natural number images then this implementation uses linear GRID and DYRANGE, and constant SPATIALRES, AMPLRES, PHASERES and FREQ for push, pop and stack storage of n stack elements. Both of the above implementations require $O(n)$ TIME.

The programmer may be able to imagine alternative system stack implementations. It should be clear that picking a particular implementation forces some constraints (e.g. resource usage) on the programmer. Hence we do not decide on any specific implementation here, instead the programmer should always define the system stack structure she wishes to use.

All programs have constant depth control flow structures and a constant number of nested function calls, with respect to the input. Stack length is dependent only on the depth of these constructs. Hence in our work the choice of stack (from the above two) is irrelevant with respect to asymptotic complexity analysis.

Basic operations

Figure 4.4.2 defines a new primitive for each of operations (i)–(vi) in Definition 2.2.4. Also, the two operations *st* and *ld* have been combined into one operation called copy and denoted '$\leftarrow$' (we often refer to this as 'rescale'). The *hlt* and *br* operations are replaced with alternative control flow mechanisms. The *hlt* operation is simulated by the end of a program (e.g. an 'end of file' marker). The obvious way to simulate *br* is with a goto command, instead we use (friendlier) **if** and **while** control flow constructs, described below.

Each of the seven code fragments make use of addresses a and b. Immediately before execution of a statement it could well be the case that these addresses hold some important data. Hence at the beginning of the code fragments the contents of a (and in some cases b) are pushed to the system stack. At the end of each code fragment the system stack is popped to a, and in some cases b, thus restoring the original contents to a or b. Defining semantics in this way allows us to write algorithms that do not explicitly refer to the addresses a and b, and so are less cumbersome.

Note that the operations move their parameter images to image a before operating. A parameter may define an entire rectangle of images, so there will be some rescaling (shrinking) involved. The programmer/analyst should be aware of this fact when calculating SPATIALRES.

h (img_1 ; img_2)

push	a	↲
ld	img_1	↲
h	↲	
st	img_2	↲
pop	a	↲

v (img_1 ; img_2)

push	a	↲
ld	img_1	↲
v	↲	
st	img_2	↲
pop	a	↲

$*$ (img_1 ; img_2)

push	a	↲
ld	img_1	↲
$*$	↲	
st	img_2	↲
pop	a	↲

$\cdot$ (img_1 , img_2 ; img_3)

push	a	↲
push	b	↲
ld	img_2	↲
st	b	↲
ld	img_1	↲
$\cdot$	↲	
st	img_3	↲
pop	b	↲
pop	a	↲

$+$(img_1 , img_2 ; img_3)

push	a	↲
push	b	↲
ld	img_2	↲
st	b	↲
ld	img_1	↲
$+$	↲	
st	img_3	↲
pop	b	↲
pop	a	↲

ρ (img_1 , img_2 , img_3 ; img_4)

push	a	↲	
ld	img_1	↲	
ρ	img_2	img_3	↲
st	img_4	↲	
pop	a	↲	

$[\xi_1', \xi_2', \eta_1', \eta_2'] \leftarrow [\xi_1, \xi_2, \eta_1, \eta_2]$

push	a	↲			
ld	ξ_1	ξ_2	η_1	η_2	↲
st	ξ_1'	ξ_2'	η_1'	η_2'	↲
pop	a	↲			

Figure 4.4.2: Semantics of the CSM programming language, in terms of low-level CSM code. The symbol ‘ ↲ ’ is shorthand for ‘branch to the leftmost address of the row immediately below’.

Control flow

Figure 4.4.3 defines the basic **if** and **while** structures that we use for control flow. The code fragments use program self-modification and direct addressing to simulate indirect addressing. This technique was given by Rojas [Roj96] and was first applied to the CSM by Naughton [Nau00b]. These implementations of **if** and **while** work only if the condition is testing equality of an image with either of the binary symbol images f_0 or f_1 (Definition 4.2.1). However it is quite straightforward to write programs that test more complicated conditions by using a combination of image thresholding and this Boolean test.[2]

[2]Note that equality testing on arbitrary real or complex numbers will never work. This undesirable situation is prevented by the fact that the set of possible addresses $\mathcal{N}$ is fully ordered and countable, while the set of images $\mathcal{I}$ is partially ordered and uncountable.

```
**if** (img$_1$ == $f_1$) **then**
  SX
**else**
  SY
**end if**
SZ
```

push	a	↲			
ld	0	2	0	1	↲
push	a	↲			
ld	"*br* SX"	↲			
st	0	2	1	1	↲
ld	"*br* SY"	↲			
st	0	2	0	0	↲
br	0	img$_1$	↲		

(a) Low level code to implement **if**. The following instructions are placed immediately before the code fragments SX and SY: $\boxed{\text{pop}}\boxed{a}\boxed{\text{st}}\boxed{0}\boxed{2}\boxed{0}\boxed{1}\boxed{\text{pop}}\boxed{\text{a}}$, to restore the contents of rows 0 and 1. A "*br* SZ" instruction is appended to the end of both SX and SY. For conditions of the form (img$_1$ == f_0) the instructions $\boxed{\text{ld}}\boxed{\text{“}\textit{br}\text{ SX”}}$ and $\boxed{\text{ld}}\boxed{\text{“}\textit{br}\text{ SY”}}$ are simply swapped.

```
**while** (img$_1$ == $f_1$) **do**
  SX
**end while**
SY
```

push	a	↲			
ld	0	2	0	1	↲
push	a	↲			
ld	"*br* SX"	↲			
st	0	2	1	1	↲
ld	"*br* SY"	↲			
st	0	2	0	0	↲
br	0	img$_1$			

(b) Low-level CSM code to implement **while**. The following instructions are placed immediately before the code fragments SX and SY: $\boxed{\text{pop}}\boxed{a}\boxed{\text{st}}\boxed{0}\boxed{2}\boxed{0}\boxed{1}\boxed{\text{pop}}\boxed{\text{a}}$, to restore the contents of rows 0 and 1, A $\boxed{\text{br}}\boxed{0}\boxed{\text{img}_1}$ instruction is appended to the end of SX, to enforce looping. For conditions of the form (img$_1$ == f_0) the instructions $\boxed{\text{ld}}\boxed{\text{“}\textit{br}\text{ SX”}}$ and $\boxed{\text{ld}}\boxed{\text{“}\textit{br}\text{ SY”}}$ are simply swapped.

Figure 4.4.3: Program control flow in the form of '**if**' and '**while**'. SX and SY represent sequences of program statements. "*br* SX" statements are used for indirect addressing.

The low-level CSM code for the **if** statement and **while** loop makes use of image a, and rows 0 & 1. These may be accessed by any code in the body of an **if** statement or **while** loop, hence we must push image a and the relevant values from rows 0 and 1 on the system stack immediately before the code body is executed. This data is then restored (with two 'pop' instructions) immediately before the end of execution of the **if** or **while** blocks. For any given program with maximum logical depth of d, a stack of size $O(d)$ is sufficient for control flow. Programs are of constant length (with respect to the input) and so have constant logical depth; hence stack size is always constant.[3]

User defined functions

All image identifiers are global: Any image on the grid can be accessed from any function or control structure, there is no notion of identifier scoping. For convenience we talk about 'function inputs' and 'function outputs' below, even though there is no such formal notion at the implementation level.

We use the same syntax for both function headers and function calls. The function inputs and outputs are CSM addresses. As a convenience the input addresses are specified by a comma (',') separated list of image addresses before a single semicolon (';'), the output addresses are listed immediately after the semicolon.

In the text of programs there are exactly three places that new image identifiers are defined. These definitions occur where new identifiers are mentioned in (i) basic operations, (ii) function headers and (iii) function calls. Each unique identifier in a program gets rewritten to low-level code as a unique grid address.

The body of a function is translated to low-level code, this code has a beginning and an end address. When a function is called, control flow switches to its beginning address. When the function terminates, control flow switches back to the origin of the call.

Identifiers in a function header and function call may differ. To cater for this, some additional 'book-keeping' code is written with the function's low-level code. When a function is called this book-keeping code (i) overwrites the image identifiers that appear in the function header with those that are in the call and (ii) overwrites the relevant images in the function body with those in the call. After the the function code has executed the original identifiers are written back to the function code. The system stack is used for this book-keeping. Since programs are of constant length this book-keeping requires constant TIME and constant stack length. For the first function that is called in a program this book-keeping is not necessary and does not occur.

[3]Recursion is not permitted in our high-level language. Also, unlike low-level programs, high-level programs cannot modify themselves.

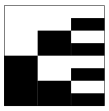
Figure 4.5.1: The output image g for code in Example 4.5.2, where $l = 3$. Black areas denote 1 and white areas denote 0.

Comments are preceded by '//'. At the beginning of a function two special comments let the reader know the names of any image constants and image variables used in the program (other than those mentioned in the function header).

4.5 Examples

Example 4.5.1 We give a short program that tests $i > j$. Here i and j are integer number images, as defined in the beginning of this chapter. The constants f_{-1}, f_0 and f_1 in the program are also integer images. The output, bool, is f_1 if $i > j$ and f_0 otherwise. Assuming integer image inputs, the program has $\max(|i|, |j|)$ DYRANGE and constant TIME, GRID, SPATIALRES, AMPLRES, PHASERES and FREQ.

```
is_greater_than ( $i$, $j$; bool)
  // constants: $f_{-1}$, $f_0$, $f_1$.
  // variables: $-j$, difference.
  $\cdot$( $j$ , $f_{-1}$ ; $-j$ )
  $+$( $i$ , $-j$ ; difference )
  $\rho$( difference , $f_0$ , $f_1$ ; bool )
end // is_greater_than
```

Example 4.5.2 The following is an example that generates an image representing all 2^l binary words of length l in $O(l)$ TIME. It makes use of the function given in the previous example. l is assumed to be a natural number image representing some value strictly greater than 0. The algorithm requires $O(2^l)$ SPATIALRES to represent all the words in a single image. $O(l)$ GRID and DYRANGE is required since part of the algorithm involves rescaling list images to their full length. The words are represented as vertically juxtaposed binary list images, all rescaled into a single image, as shown in Figure 4.5.1.

```
generate_binary_words ($l$, $\overline{0}$, $\overline{1}$; $g$, $|g|$)

 // constants: $f_0$, $f_1$, $f_{-1}$.
 // variables: flag, top_0_bottom_1, $l-1$, $-|g|$, difference.

     // Generate the image top_0_bottom_1,
     // where the top half is 0 the bottom half is 1
 [ 0, 0, 2, 2 ] $\leftarrow$ $\overline{0}$
 [ 0, 0, 3, 3 ] $\leftarrow$ $\overline{1}$
 top_0_bottom_1 $\leftarrow$ [ 0, 0, 2, 3 ]

     // Assign values for the base case
 $g$ $\leftarrow$ top_0_bottom_1
 $|g|$ $\leftarrow$ $f_1$

     // If $l > 1$ then enter loop
 is_greater_than ( $l$, 1; flag)
 while (flag == $f_1$) do

       // Vertically juxtapose 2 copies of $g$ and rescale into 1 image
   [ 1, $|g|$, 2, 2 ] $\leftarrow$ $g$
   [ 1, $|g|$, 3, 3 ] $\leftarrow$ $g$
   [ 1, $|g|$, 2, 2 ] $\leftarrow$ [ 1, $|g|$, 2, 3 ]

       // Place top_0_bottom_1 to the left of the
       // vertically juxtaposed copies of $g$
   [ 0, 0, 2, 2 ] $\leftarrow$ top_0_bottom_1

       // Make a new image $g$ that represents all words of length $|g| + 1$
   $g$ $\leftarrow$ [ 0, $|g|$, 2, 2 ]

       // Increment $|g|$
   +( $|g|$ , $f_1$ ; $|g|$ )

       // If $l > |g|$ then enter loop again
   is_greater_than ( $l$, $|g|$; flag)
 end while
end // generate_binary_words
```

The example gives an indication of the type of speed-up (over sequential models) achievable with the CSM.

4.6 Discussion

There are many ways to represent data as images. The data representations we give are not meant to be exhaustive, we choose to present the representations that are used in subsequent chapters. Often it is the case that a new CSM algorithm may depend on a novel data representation technique. Given this, one should bear in mind that data representations should be in some sense reasonable. If a representation function (e.g. mapping words to images) is too powerful we are lessening what we say about the complexity of our algorithm. Moore [Moo98] has an interesting discussion on this topic. He notes that most authors use the best-known representation function from discrete to continuous, mapping the symbol sequence of a word to the digit sequence of a number (e.g $w_0 w_1 w_2 \ldots$ is represented by $0.w_0 w_1 w_2 \ldots$). Such representations (e.g. those that appear in [SS94]) can be generated by iterated affine maps. Moore [Moo98] presents results on real-time language recognition by dynamical systems. This leads to his thesis that "*reasonable [representations] consist of reasonable maps, iterated in real time as the symbols of the word are input one by one*".

For the case of our model we choose to represent natural, integer, real or complex numbers in a very direct way. The representation function simply maps a value r to an image whose centre has value r. If we ignore all of the image except the centre point [i.e. ignore the constant (zero) part of the image] then this representation is the identity function and thus, we argue, a reasonable representation. Suppose the number we wish to represent has fractional parts. In order to represent the value as a number image our complexity measures will incur a cost that is at worst exponential in the length of the fractional part. For example the real number representation of the decimal number 0.01 requires AMPLRES of 100. We return to this point in Section 7.4.

The list (and list-matrix) image representation function is affine, however it is not a (sequentially) iterated map as in [Moo98]. We feel that this is still a reasonable representation function. The CSM is a parallel model, and as such needs to be able to access n input symbols in less than n TIME, otherwise efficient (sublinear TIME) parallel computation is impossible. Intuitively the list and list-matrix representations are quite simple, we are in effect taking a 'picture' of the word or matrix.

The stack and stack-matrix image representations are not affine. However, if we use stack images in a nontrivial way, then we immediately incur a cost of exponential SPATIALRES in the represented word length. The same argument holds for stack-matrix images, so we are getting no free lunches in either case.

There were two main goals in mind when designing the programming language, (i) algorithms should be easy to understand and (ii) complexity analysis should be straightforward. It would have been possible to give our programming language a more concise, modern syntax. However, we feel that this would have sacrificed the second goal somewhat. The complexity resources we defined for the CSM are rather unusual compared to standard parallel resources such as word size and number of processors. An overly concise language may have made analysis of programs difficult for someone not familiar with the CSM architecture. We have not given a complete formal specification of the language, however it is described in enough detail for the purposes of our work.

5

Lower bounds on $\mathcal{C}_2$-CSM power

5.1 Introduction

The main result of this chapter is a lower bound on the computational power of the $\mathcal{C}_2$-CSM. We show that TIME on the $\mathcal{C}_2$-CSM is at least as powerful as space on Turing machines. More precisely

$$\text{NSPACE}(S(n)) \subseteq \mathcal{C}_2\text{-CSM–TIME}(O(S(n) + \log n)^4)$$

For example polynomial time $\mathcal{C}_2$-CSMs accept the PSPACE languages. This inclusion is one of two that are necessary to show that the $\mathcal{C}_2$-CSM verifies the parallel computation thesis. We show the other inclusion in Chapter 6.

The result is proved by giving a quadratic TIME $\mathcal{C}_2$-CSM simulation of the vector machine model of Pratt and Stockmeyer. The simulation is carried out with a cubic SPACE overhead in vector machine space.

We begin by introducing vector machines, and the variant that we simulate called index-vector machines. We then proceed to describe our representation of vectors as images and the grid layout of the simulation program. Then, in a sequence of theorems we simulate each vector machine operation, all the while being mindful of resource overheads. We finish with a discussion.

5.2 Vector machines

The vector machine model was originally proposed by Pratt, Rabin and Stockmeyer [PRS74]. In this work we mostly use the conventions of a later paper by Pratt and Stockmeyer [PS76] and of [BDG88b].

A vector V is a binary sequence (from $\{0, 1\}$) that is infinite to the left only. Vectors are *ultimately constant*; after some finite number of bits every bit to the left is either always 0 or always 1. Vectors are written from right to left, Pratt and Stockmeyer adopt this convention because they want vector machines to simulate operations on integers [PS76]. The length of vector V, denoted $|V|$, is defined to be the length of the non-constant part of V.

Often we interpret a vector as a number. Here we adopt the convention from [BDG88b] and say that an ultimately 0 sequence represents a positive number and an ultimately 1 sequence represents a negative number. Also the non-constant part represents a positive binary integer in the usual way, with the rightmost vector bit representing the least significant numerical bit. The negative integer $-n$ is represented by the complement of the vector representing $n \geqslant 0$. For example 5 is represented as $\ldots 000101$ and -5 as $\ldots 111010$. In this scheme 0 has two representations: $\ldots 000$ and $\ldots 111$. Since positive (respectively negative) binary numbers will always have a leading 1 (respectively 0) the most significant bit of a number and the beginning of the ultimately constant part are both well defined. This is like the 1's complement binary representation of integers [Sav76] except that in 1's complement notation there is a single single leading sign bit ψ as opposed to an infinite repeating sequence of the sign bit ψ

A vector machine (program) is a sequence of instructions, where each instruction is of the form given in the following definition.

Definition 5.2.1 (Balcázar *et al* [BDG88b]) *Vector machine instructions and their meanings. In a program, instructions are labelled to facilitate the* goto *instruction.*

Vector instruction	*Meaning*
$V_i := x$	*Load the positive constant binary number x into vector V_i.*
$V_k := \neg V_i$	*Bitwise parallel negation of vector V_i.*
$V_k := V_i \wedge V_j$	*Bitwise parallel 'and' of two vectors.*
$V_k := V_i \uparrow V_j$	*If V_j is ultimately 0 (resp. 1) then shift V_i to the left (resp. right) by the distance given by the binary number v_j and store the result in V_k. If $V_j = 0$ then V_i is copied to V_k.*
$V_k := V_i \downarrow V_j$	*If V_j is ultimately 0 (resp. 1) then shift V_i to the right (resp. left) by the distance given by the binary number v_j and store the result in V_k. If $V_j = 0$ then V_i is copied to V_k.*
goto m *if* $V_i = 0$	*If $V_i = 0$ then branch to the instruction labelled m.*
goto m *if* $V_i \neq 0$	*If $V_i \neq 0$ then branch to the instruction labelled m.*

Configurations, computations, accepting computations and computation time are all defined in the obvious way. Computation space is the maximum over all configurations, of the sum of the lengths of the vectors in each configuration. A language accepting vector machine on input w has an input vector of the form ...000w where $w \in 1\{0,1\}^*$. Vector machines may be deterministic or nondeterministic. In Flynn's taxonomy [Qui94] vector machines are classed as single instruction, multiple data (SIMD) machines.

Example 5.2.1 A loop that executes t times, with vector V_2 as the loop counter [PS76].

```
    $V_1 := \ldots 0001$
    $V_2 := \ldots 00010^{n-1}$
$i$     $V_1 := V_1 \uparrow V_1$
        $V_2 := V_2 \downarrow 1$
    goto $i$ if $V_2 \neq 0$
```

Initially vector V_1 represents the number 1, after t iterations it represents an exponential tower of 2s, of height larger than t. Shifts ($\uparrow$ and $\downarrow$), and parallel bitwise operations, have impressive power. Vector machines generate, and operate on, huge objects very quickly.

5.2.1 Index-vector machines

Pratt and Stockmeyer [PS76] characterised the power of a restricted version of the model. This restricted version is called an index-vector machine and the class of such machines is denoted $\mathcal{V}_I$.

Definition 5.2.2 (Pratt and Stockmeyer [PS76]) *A vector machine is of class $\mathcal{V}_I$ (equivalently, an index-vector machine) if its registers are partitioned into two disjoint sets, one set called index registers and the other called vector registers, such that (i) each Boolean operation in the program involves either only index registers or only vector registers; and (ii) each shift instruction is of the form*

$$V_1 := V_2 \uparrow I, \quad V_1 := V_2 \downarrow I, \quad I := J \uparrow 1, \quad I := J \downarrow 1$$

where V_1 and V_2 are vector registers, and I and J are index registers. For language recognition the input register is placed in a vector register.

It is straightforward to prove the following lemma by induction on t.

Lemma 5.2.3 (Pratt and Stockmeyer [PS76]) *Given index-vector machine $M \in \mathcal{V}_I$ with n as the maximum input length, there is a constant c such that vector length in index (respectively vector) registers is bounded above by $c + t$ (respectively $2^{c+t} + n$) after t timesteps.*

Pratt and Stockmeyer [PS76] were unable to characterise the power of vector machines with unrestricted shifts, however they conjectured that for language recognition they have no more power than index-vector machines. Interestingly, Simon later proved[1] this conjecture [Sim77]; unrestricted vector shifts give no extra power up to a polynomial in time.

[1]This was shown by giving a polynomial space Turing machine that decides equivalence of straight-line RAM programs that use special instruction sets and data representations.

Pratt and Stockmeyer's main result is a characterisation of the power of index-vector machines. This characterisation is given by the following two inclusions, proved for time bounded index-vector machines and space bounded Turing machines.

Theorem 5.2.4 ([PS76])

$$\mathrm{NSPACE}(S(n)) \subseteq \mathcal{V}_I\text{-TIME}(O(S(n)+\log n)^2)$$
$$\mathcal{V}_I\text{-TIME}(T(n)) \subseteq \mathrm{DSPACE}(O(T(n)(T(n)+\log n)))$$

In other words, index-vector machines verify the parallel computation thesis. Moreover it was shown that modulo a polynomial, deterministic and nondeterministic vector machines have equal power [PRS74]. It follows that deterministic and nondeterministic polynomial time are equal on such machines.

We will prove by simulation that $\mathcal{C}_2$-CSMs are at least as powerful as index-vector machines (up to a polynomial in time). More precisely

$$\mathcal{V}_I\text{-TIME}(T(n)) \subseteq \mathcal{C}_2\text{-CSM–TIME}(O(T^2(n)))\,. \tag{5.2.1}$$

The simulation simultaneously has a polynomial overhead in space.

To prove this we simulate each index-vector machine instruction in $O(\log|V_{\max}|)$ TIME where $|V_{\max}| \in \mathbb{N}$ is the maximum length of (the non-constant part of) any of the unrestricted vectors mentioned in the instruction. Additionally we simulate the index-vector shifts in linear TIME. From Lemma 5.2.3 this TIME bound ensures that our overall simulation executes in quadratic TIME, which is sufficient for the inclusion given by Eq. (5.2.1). The SPACE bound on the simulation is $O(|V_{\max}|^3)$.

5.3 Representation

We first give the representation of vectors by images and then the grid layout of the simulating $\mathcal{C}_2$-CSM . Then we give a series of theorems, each proof contains a $\mathcal{C}_2$-CSM program that simulates a vector machine instruction. We state resource usage for each program, ignoring AMPLRES and PHASERES as these both have constant value of 2. Also we ignore FREQ, (recall from the $\mathcal{C}_2$-CSM definition the we do not analyse $\mathcal{C}_2$-CSMs in terms of FREQ). Finally Corollary 5.4.12 gives the overall resource usage for our simulation of an index-vector machine.

5.3.1 Image representation of vectors

Throughout the remainder of the current chapter we use the following notation. If we have a vector V_i then $v_i \in \{0,1\}^*$ is the non 'ultimately constant' part of V_i. If the 'ultimately constant' part of V_i is 0^ω (respectively 1^ω) then $\mathrm{sign}(v_i) = 0$ (respectively $\mathrm{sign}(v_i) = 1$).

The vector V_i is represented by three images

$$V_i \overset{rep}{\Longrightarrow} (\,\overline{v_i}, \overline{|v_i|}, \overline{\mathrm{sign}(v_i)}\,)$$

The image $\overline{v_i}$ is the binary list image representation of v_i. Image $\overline{|v_i|}$ is the natural number image represention of $|v_i|$ (the length of v_i). Accessing these images will incur SPATIALRES and DYRANGE costs that are linear in $|v_i|$. Image $\overline{\mathrm{sign}(v_i)}$ is the binary symbol image representing 0 if vector V_i is ultimately 0 and 1 if V_i is ultimately 1. We use the same representation scheme for vector program constants.

The simulation uses natural number images as addresses, which are clearly reasonable in the sense of the $\mathcal{C}_2$-CSM definition. Hence addressing incurs a (linear) DYRANGE cost.

5.3.2 Grid layout

Figure 5.3.1 illustrates the grid layout for our index-vector machine simulation. Rows 0 and 1 are used for rough work (e.g. simulating shifts) and for evaluating conditions in **if** statements and **while** loops (as described in Section 4.4.2). Row 2 is reserved for the program stack that is used by whatever compiler converts the CSM programming language to low level CSM code.[2] Row 3 is used for constant images and variable images. The following constant images are given as input to the vector machine simulation program: f_{-1}, f_0, $f_{\frac{1}{2}}$, f_1 and f_2. In fact f_{-1} and $f_{\frac{1}{2}}$ are the only constants that are necessary as we can easily generate the others from these. A number of variable (temporary) images are used within each routine. For example the algorithm in Theorem 5.4.3 uses the variables: list_neg_ones, $\overline{-v_i}$ and list_ones. Row 4 is used for vector images. Vector images $\overline{v_i}$, $\overline{|v_i|}$ and $\overline{\mathrm{sign}(v_i)}$ are located at addresses $(3i, 4)$, $(3i+1, 4)$ and $(3i+2, 4)$ respectively. Rows 5 to G_y are for the vector machine simulation program. G_y is a constant related to the size of our simulation program. So the vertical GRID complexity of our simulation is constant. On the other hand the horizontal GRID complexity is linear in vector machine space, specifically G_x is equal to a small constant times the length of the longest vector in the simulated vector machine's computation.

With the exception of rows 0 and 1 we do not explicitly refer to any numerical address values in the simulation (all variables are given names from the outset). Therefore the above layout is one of a number of example layouts that would work for the simulation.

5.4 $\mathcal{C}_2$-CSM simulation of vector machines

We begin by giving a straightforward simulation of vector assignment.

[2]Two possible stack structures were discussed in Section 4.4.2, either of these may be used and will not effect the asymptotic resource usages given in this chapter.

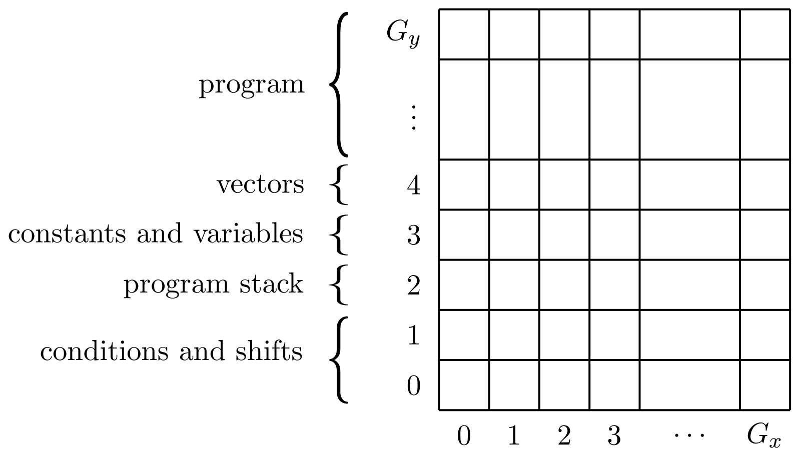

Figure 5.3.1: Grid layout for index-vector machine simulation.

Theorem 5.4.1 ($V_i := x$) *The vector machine assignment instruction $V_i := x$ is simulated by a* $\mathcal{C}_2$-CSM *in* $O(1)$ TIME*,* $O(1)$ GRID*,* $O(|x|)$ DYRANGE*,* $O(\max(|x|, |v_i|))$ SPATIALRES*.*

Proof. The straight-line CSM program below simulates $V_i := x$. In this program the images representing x are simply copied to the images representing V_i.

```
assignment ( $\overline{x}$, $\overline{|x|}$, $\overline{\text{sign}(x)}$; $\overline{v_i}$, $\overline{|v_i|}$, $\overline{\text{sign}(v_i)}$ )
  $\overline{v_i} \leftarrow \overline{x}$
  $\overline{|v_i|} \leftarrow \overline{|x|}$
  $\overline{\text{sign}(v_i)} \leftarrow \overline{\text{sign}(x)}$
end // assignment
```

We require $O(\max(|x|, |v_i|))$ SPATIALRES to represent x and v_i as binary list images. DYRANGE of $O(|x|)$ is needed to represent $|x|$ as a natural number image. No address goes beyond the initial grid limits hence we require only constant GRID. □

Before moving on to simulate another vector machine operation, we give a program that will be frequently used. The program quickly generates a list image, where each list element is identical. We state the lemma for the specific case that each list element is a binary symbol image. By simply changing the value of one input, the algorithm generalises to arbitrary repeated lists (with a suitable change in resource use, dependent only on the complexity of the new input image element).

Lemma 5.4.2 (generate_list$(f_\psi, l; g)$) *A list image g that contains l list elements, each of which is a copy of input binary image f_ψ, is generated in* $O(\log l)$ TIME, $O(l)$ GRID, SPATIALRES *and* DYRANGE.

Proof. The algorithm (below) horizontally juxtaposes two copies of f_ψ and rescales them to a single image. This juxtaposing and rescaling is repeated on the new image; the process is iterated a total of $\lceil \log l \rceil$ times to give a list of length $2^{\lceil \log l \rceil}$, giving the stated TIME bound. Finally, l images are selected (in constant TIME) from this list and rescaled to give the output. $O(l)$ SPATIALRES is necessary to store the list in a single image. $O(l)$ GRID is used to stretch the list out to its full length in the two final program instructions. Recall that we are using natural number images for addresses, hence $O(l)$ DYRANGE is used when we stretch the list across $2^{\lceil \log l \rceil}$ images.

```
generate_list ($f_\psi$, $l$; $g$)

  // constants: $f_{-1}$, $f_1$, $f_2$.
  // variables: $l_c$, flag, $-l_c$, difference.

  $g \leftarrow f_\psi$
  $l_c \leftarrow f_1$
  $\rho(\, l\, , 0\, , 1\, ; \text{flag}\,)$
  while ( flag == $f_1$ ) do

       // horizontally juxtapose two copies of $g$ and rescale
    [ 0, 0, 0, 0 ] $\leftarrow g$
    [ 1, 1, 0, 0 ] $\leftarrow g$
    $g \leftarrow$ [ 0, 1, 0, 0 ]
    $\cdot(\, l_c\, , f_2\, ; l_c\,)$
    $\cdot(\, l_c\, , f_{-1}\, ; -l_c\,)$
    $+(\, l\, , -l_c\, ; \text{difference}\,)$
    $\rho(\, \text{difference}\, , 0\, , 1\, ; \text{flag}\,)$
  end while

     // Select $l$ images and rescale to one image
  [ 1, $l_c$, 0, 0 ] $\leftarrow g$
  $g \leftarrow$ [ 1, $l$, 0, 0 ]
end // generate_list
```
□

Theorem 5.4.3 ($V_k := \neg V_i$) *The vector machine negation instruction $V_k := \neg V_i$ is simulated by a $\mathcal{C}_2$-CSM in* $O(\log |v_i|)$ TIME, $O(|v_i|)$ GRID *and* DYRANGE, *and* $O(\max(|v_k|, |v_i|))$ SPATIALRES.

Proof. The CSM program below simulates $V_k := \neg V_i$. To generate $\neg V_i$ the program generates a list of '-1's of length $|v_i|$. This list image is then multiplied by $\overline{v_i}$; changing each 1 in $\overline{v_i}$ to -1 and leaving each 0 unchanged. Then we add 1 to the each element in the resulting list. A simple **if** statement negates $\overline{\text{sign}(v_i)}$.

```
$\neg$ ( $\overline{v_i}$, $\overline{|v_i|}$, $\overline{\text{sign}(v_i)}$; $\overline{v_k}$, $\overline{|v_k|}$, $\overline{\text{sign}(v_k)}$ )

  // constants: $f_{-1}$, $f_0$, $f_1$.
  // variables: list_neg_ones, $\overline{-v_i}$, list_ones.

    // generate list of $-1$s
  generate_list ($f_{-1}$, $\overline{|v_i|}$; list_neg_ones)

    // change each 1 in $\overline{v_i}$ to $-1$
  $\cdot$ ( $\overline{v_i}$ , list_neg_ones ; $\overline{-v_i}$ )

    // generate list of 1s
  generate_list ($f_1$, $\overline{|v_i|}$; list_ones)

    // change $-1$s to 0s and 0s to 1s in $\overline{v_i}$, place result in $\overline{v_k}$
  $+$( $\overline{-v_i}$ , list_ones ; $\overline{v_k}$ )
  if ( $\overline{\text{sign}(v_i)} == f_1$ ) then
    $\overline{\text{sign}(v_k)} \leftarrow f_0$
  else
    $\overline{\text{sign}(v_k)} \leftarrow f_1$
  end if
end // $\neg$
```

Each call to the function generate_list($\cdot$) requires $O(\log |v_i|)$ TIME, otherwise TIME is constant. The remaining resource usages are necessary for accessing vectors and rescaling them to their full length. □

The proof of the following straightforward lemma gives a program that decides which of two vectors is the longer in constant TIME. It also shows that we can decide the max or min of two integer images in constant TIME. The result will be used in Theorem 5.4.5 in the $\mathcal{C}_2$-CSM simulation of the vector instruction $\wedge$.

Lemma 5.4.4 **(**max($\cdot$) **and** min($\cdot$)**)** *The* max *(or* min*) length of the vectors V_i and V_j is decided in* $O(1)$ TIME*,* $O(1)$ GRID*,* $O(\max(|v_i|, |v_j|))$ SPATIALRES*,* $O(\max(|v_i|, |v_j|))$ DYRANGE*.*

Proof. The max($\cdot$) algorithm thresholds the value $|v_j| - |v_i|$ to the range $[0, 1]$.

```
max ($\overline{v_i}$, $\overline{|v_i|}$, $\overline{\mathrm{sign}(v_i)}$, $\overline{v_j}$, $\overline{|v_j|}$, $\overline{\mathrm{sign}(v_j)}$; longest, |longest|, sign(longest))

  // constants: $\underline{f_{-1}}$, $f_0$, $f_1$.
  // variables: $\overline{-|v_i|}$, difference, flag.

  $\cdot$( $\overline{|v_i|}$, $\underline{f_{-1}}$ ; $\overline{-|v_i|}$ )
  +( $\overline{|v_j|}$, $\overline{-|v_i|}$ ; difference )
  $\rho$( difference, $f_0$, $f_1$ ; flag )
  if ( flag == $f_0$ ) then
      // $\overline{|v_i|}$ is max
    longest ← $\overline{v_i}$
    |longest| ← $\overline{|v_i|}$
    sign(longest) ← $\overline{\mathrm{sign}(v_i)}$
  else
      // $\overline{|v_j|}$ is max
    longest ← $\overline{v_j}$
    |longest| ← $\overline{|v_j|}$
    sign(longest) ← $\overline{\mathrm{sign}(v_j)}$
  end if
end // max
```

To decide the min length of two vector images, we use the above algorithm except that the code in the **if** block is exchanged with the code in the **else** block. The function header for min(·) is formatted as follows:

min($\overline{v_i}$, $\overline{|v_i|}$, $\overline{\mathrm{sign}(v_i)}$, $\overline{v_j}$, $\overline{|v_j|}$, $\overline{\mathrm{sign}(v_j)}$; shortest, |shortest|, sign(shortest)).
□

Theorem 5.4.5 ($V_k := V_i \wedge V_j$) *The vector machine instruction $V_k := V_i \wedge V_j$ is simulated by a $\mathcal{C}_2$-CSM in $O(\log \max(|v_i|, |v_j|))$ TIME, $O(\max(|v_i|, |v_j|, |v_k|))$ SPATIALRES, and $O(\max(|v_i|, |v_j|))$ GRID and DYRANGE.*

Proof. We use multiplication of vector images to simulate $V_i \wedge V_j$ in a parallel fashion. However this does not work directly if $|v_i| \neq |v_j|$, in such cases we pad the shorter of the vector images so that both are then of the same length. To find the longer and shorter of the two vectors we make use of the max(·) and min(·) routines given in Lemma 5.4.4. The following program simulates $\wedge$.

```
$\wedge$ ($\overline{v_i}$, $\overline{|v_i|}$, $\overline{\text{sign}(v_i)}$, $\overline{v_j}$, $\overline{|v_j|}$, $\overline{\text{sign}(v_j)}$; $\overline{v_k}$, $\overline{|v_k|}$, $\overline{\text{sign}(v_k)}$ )

  // constants: $f_{-1}$, $f_0$, $f_1$.
  // variables: longest, |longest|, sign(longest), shortest, |shortest|,
  sign(shortest), difference, difference+1, pad, padded_shortest.

     // Find which vector is the longest and which is the shortest
  max($\overline{v_i}$, $\overline{|v_i|}$, $\overline{\text{sign}(v_i)}$, $\overline{v_j}$, $\overline{|v_j|}$, $\overline{\text{sign}(v_j)}$; longest, |longest|, sign(longest))
  min($\overline{v_i}$, $\overline{|v_i|}$, $\overline{\text{sign}(v_i)}$, $\overline{v_j}$, $\overline{|v_j|}$, $\overline{\text{sign}(v_j)}$; shortest, |shortest|, sign(shortest))
  · ( $f_{-1}$ , |shortest| ; −|shortest| )
  +( longest , −|shortest| ; difference )
  +( difference , $f_1$ ; difference+1 )

     // Pad the shortest vector image with 1s or 0s
     // so that both vector images are of the same length
  if ( sign(longest) == $f_1$ ) then
    generate_list ($f_1$, difference; pad)
    [ 1, difference, 1, 1 ] ← pad
    [ difference+1, |longest|, 1, 1 ] ← shortest
    padded_shortest ← [ 1, |longest|, 1, 1 ]
  else
    [ 1, |longest|, 1, 1 ] ← $f_0$
    [ difference+1, |longest|, 1, 1 ] ← shortest
    padded_shortest ← [ 1, |longest|, 1, 1 ]
  end if

     // Now that the vector images are of the same length
     // a single multiplication step simulates $v_i \wedge v_j$
  · ( longest , padded_shortest ; $\overline{v_k}$ )
  · ( sign(longest) , sign(shortest) ; $\overline{\text{sign}(v_k)}$ )
  $\overline{|v_k|}$ ← |longest|
end // $\wedge$
```

The algorithm requires $O(\log \max(|v_i|, |v_j|))$ TIME for the generate_list($\cdot$) call (the worst case is when exactly one of the vectors is of length 0). The rest of the algorithm runs in $O(1)$ TIME, including determining which vector is longer, padding of the shorter vector and parallel multiplication of vectors. The remaining resource usages on vector images in the theorem statement are for accessing and storing to a single image, and stretching to full length. $\square$

The proofs of the next two lemmas give algorithms to simulate vector left shift and right shift. In simple terms, the main idea is to copy large numbers of images to simulate shifting.

Lemma 5.4.6 (left_shift(·)) *A left shift of distance $n \geqslant 0$ on a vector V_i, to create vector V_k, is simulated in $O(1)$* TIME, *$O(|v_i+n|)$* GRID *and* DYRANGE, *and $O(\max(|v_i+n|, |v_k|))$* SPATIALRES.

Proof. The algorithm assumes that n is given as a natural number image. After the shift (in accordance with the definition of vector shift), 0s are to be placed in the rightmost positions. We simulate the shift by stretching $\overline{v_i}$ out to its full length, placing n zero images to the right of the stretched $\overline{v_i}$, and then selecting all of $\overline{v_i}$ along with the n zeros and rescaling back to one image.

```
left_shift ($\overline{n}$, $\overline{v_i}$, $\overline{|v_i|}$, $\overline{\text{sign}(v_i)}$; $\overline{v_k}$, $\overline{|v_k|}$, $\overline{\text{sign}(v_k)}$ )

  // constants: $\overline{f_0}$, $\overline{f_1}$.
  // variables: $\overline{|v_i|+1}$, $\overline{|v_i|+n}$, flag.

  [ 1, $\overline{|v_i|}$, 0, 0 ]  ←  $\overline{v_i}$
  +( $\overline{n}$ , $\overline{|v_i|}$ ; $\overline{|v_i|+n}$ )
  $\rho$( $\overline{n}$ , $f_0$ , $f_1$ ; flag )
  if  ( flag == $f_1$ )  then

      // $n > 0$, so zeros are placed in the new rightmost positions
    +( $\overline{|v_i|}$ , $f_1$ ; $\overline{|v_i|+1}$ )
    [ $\overline{|v_i|+1}$, $\overline{|v_i|+n}$, 0, 0 ]  ←  $f_0$
  end if
  $\overline{v_k}$  ←  [ 1, $\overline{|v_i|+n}$, 0, 0 ]
  $\overline{|v_k|}$  ←  $\overline{|v_i|+n}$
  $\overline{\text{sign}(v_k)}$  ←  $\overline{\text{sign}(v_i)}$
end // left_shift
```

□

Lemma 5.4.7 (right_shift(·)) *A right shift of distance $n \geqslant 0$ on a vector V_i, to create vector V_k, is simulated in $O(1)$* TIME, *$O(|v_i|)$* GRID, *$O(\max(|v_i|, |v_k|))$* SPATIALRES *and* DYRANGE.

Proof. The algorithm below assumes that n is given as a natural number image whose value is $\leqslant |v_i|$. We simulate the right shift by stretching $\overline{v_i}$ out to its full length, selecting the leftmost $|v_i| - n$ images and rescaling back to one image. If $n \geqslant |v_i|$ then the output of the algorithm is the representation of the zero vector.

```
right_shift ($\overline{n}$, $\overline{v_i}$, $\overline{|v_i|}$, $\overline{\text{sign}(v_i)}$; $\overline{v_k}$, $\overline{|v_k|}$, $\overline{\text{sign}(v_k)}$ )

  // constants: $f_{-1}$, $f_0$, $f_1$.
  // variables: $\overline{-n}$, $\overline{|v_i|-n}$, flag.

  $\cdot$( $\overline{n}$, $f_{-1}$ ; $\overline{-n}$ )
  +( $\overline{|v_i|}$, $\overline{-n}$ ; $\overline{|v_i|-n}$ )
  $\rho$( $\overline{|v_i|-n}$, $f_0$, $f_1$ ; flag )
  if ( flag == $f_1$ ) then

      // In this case $n < |v_i|$, so simulate shift
    [ 1, $\overline{|v_i|}$, 0, 0 ] $\leftarrow$ $\overline{v_i}$
    $\overline{v_k}$ $\leftarrow$ [ 1, $\overline{|v_i|-n}$, 0, 0 ]
    $\overline{|v_k|}$ $\leftarrow$ $\overline{|v_i|-n}$
    $\overline{\text{sign}(v_k)}$ $\leftarrow$ $\overline{\text{sign}(v_i)}$
  else

      // In this case $n = |v_i|$, output the zero vector
    $\overline{v_k}$ $\leftarrow$ $f_0$
    $\overline{|v_k|}$ $\leftarrow$ $f_0$
    $\overline{\text{sign}(v_k)}$ $\leftarrow$ $\overline{\text{sign}(v_i)}$
  end if
end // right_shift
```

Unlike the case of right_shift($\cdot$), n is not present in the expression for DYRANGE since it represents a value $\leqslant |v_i|$. The other resource usages are clear from the algorithm. □

We make use of the previous two results in the following theorem.

Theorem 5.4.8 **($V_k := V_i \uparrow V_j$)** *The vector machine instruction $V_k := V_i \uparrow V_j$ is simulated by a $\mathcal{C}_2$-CSM in $O(|v_j|)$* TIME*, $O(|v_i| + 2^{|v_j|})$* GRID *and* DYRANGE*, and $O(\max(|v_k|, |v_i| + 2^{|v_j|}))$* SPATIALRES*.*

Proof. The shift instruction ($\uparrow$) shifts V_i by the binary number stored in V_j. We simulate the shift by stretching V_i out to its full length; then selecting either part of V_i, or V_i and some extra zero images; and finally rescaling back to one image. The simulator's addresses are represented by natural number images whereas vectors are represented by binary list images. Hence we convert the binary number defined by ($\overline{v_j}$, $\overline{|v_j|}$, $\overline{\text{sign}(v_j)}$) to a natural number image called shift_distance. The algorithm below generates shift_distance from the binary index-vector image $\overline{v_j}$ and then performs the shift. The **while** loop executes $|v_j|$ times to generate shift_distance. Inside this loop there are three **if** statements. For the first (highest level) **if** statement: if V_j is positive execution always enters the **if then** part, if V_j is negative execution always enters the **else** part. If V_j is positive (respectively negative) the second (respectively third) **if** is executed only in the case that the current bit of V_j is 1 (respectively 0). Essentially these two lower level **if** statements are adding 2^l to shift_distance, at the l^{th} bit of V_j. A final **if** statement decides which direction to shift, passing shift_distance and the relevant vectors to one of the functions left_shift(·) or right_shift(·) which were given above.

```
↑ ($\overline{v_i}$, $\overline{|v_i|}$, $\overline{\text{sign}(v_i)}$, $\overline{v_j}$, $\overline{|v_j|}$, $\overline{\text{sign}(v_j)}$; $\overline{v_k}$, $\overline{|v_k|}$, $\overline{\text{sign}(v_k)}$ )

  // constants: $f_{-1}$, $f_0$, $f_1$, $f_2$.
  // variables: shift_distance, current_bit, current_power_2, flag.

  shift_distance ← $f_0$
  current_bit ← $\overline{|v_j|}$
  current_power_2 ← $f_1$
  [ 1, $\overline{|v_j|}$, 0, 0 ] ← $\overline{v_j}$
  $\rho$ ( $\overline{|v_j|}$ , $f_0$ , $f_1$ ; flag )

    // $O(|v_j|)$ TIME while loop to calculate shift_distance
  while ( flag == $f_1$ ) do
    if ( $\overline{\text{sign}(v_j)}$ == $f_0$ ) then

        // $V_j$ is ultimately 0 (positive)
      if ( [ current_bit, current_bit, 0, 0 ] == $f_1$ ) then
        +( shift_distance , current_power_2 ; shift_distance )
      end if
    else
        // $V_j$ is ultimately 1 (negative)
      if ( [ current_bit, current_bit, 0, 0 ] == $f_0$ ) then
        +( shift_distance , current_power_2 ; shift_distance )
      end if
    end if
    · ( current_power_2 , $f_2$ ; current_power_2 )
    +( current_bit , $f_{-1}$ ; current_bit )
    $\rho$ ( current_bit , $f_0$ , $f_1$ ; flag )
  end while

    // Decide which direction to shift, then shift
  if ( $\overline{\text{sign}(v_j)}$ == $f_0$ ) then
    left_shift (shift_distance, $\overline{v_i}$, $\overline{|v_i|}$, $\overline{\text{sign}(v_i)}$; $\overline{v_k}$, $\overline{|v_k|}$, $\overline{\text{sign}(v_k)}$ )
  else
    right_shift (shift_distance, $\overline{v_i}$, $\overline{|v_i|}$, $\overline{\text{sign}(v_i)}$; $\overline{v_k}$, $\overline{|v_k|}$, $\overline{\text{sign}(v_k)}$ )
  end if
end // ↑
```

The **while** loop efficiently generates a value of $O(2^{|v_j|})$ in $O(|v_j|)$ TIME. At different stages of the algorithm each of $\overline{v_i}$ and $\overline{v_j}$ are rescaled to their full length, across $|v_i|$ and $|v_j|$ images respectively. We get the value $O(|v_i|+2^{|v_j|})$ for GRID since in the worst case V_i is left shifted by the value $2^{|v_j|}$, and (when stretched) the resulting vector lies across $O(|v_i| + 2^{|v_j|})$ images. This upper bound also covers the right shift case (when V_j is negative). Analogously we get the same value for SPATIALRES and DYRANGE (except $|v_k|$ is also in the SPATIALRES expression as it could contain some values before the program executes). □

In the previous simulation V_j is either an index-vector or else represents 1, and by Lemma 5.2.3 index-vectors grow at most linearly over time. Hence this $O(|v_j|)$ TIME simulation will be sufficient for the resource usage given in our overall simulation (Corollary 5.4.12). The same remark holds for the following theorem.

Theorem 5.4.9 **($V_k := V_i \downarrow V_j$)** *The vector machine instruction $V_k := V_i \downarrow V_j$ is simulated by a $\mathcal{C}_2$-CSM in $O(|v_j|)$ TIME, $O(|v_i| + 2^{|v_j|})$ GRID and DYRANGE, and $O(\max(|v_k|, |v_i| + 2^{|v_j|}))$ SPATIALRES.*

Proof. We use the algorithm from the previous theorem (5.4.8), except we exchange the calls to the right_shift(·) and left_shift(·) programs. □

The proof of the following lemma gives a log TIME algorithm to decide if a list or vector image represents a word that consists only of zeros. It is possible to give a constant TIME algorithm that makes use of Fourier transformation (to 'sum' the entire list in constant TIME). However we choose not to use this FT algorithm here (we discuss this point in Section 5.5).

Lemma 5.4.10 *A $\mathcal{C}_2$-CSM that does not use Fourier transformation decides whether or not a list (equivalently vector) image $\overline{v_i}$ represents the word $0^{|v_i|}$ in $O(\log |v_i|)$ TIME, $O(|v_i|)$ GRID, SPATIALRES and DYRANGE.*

Proof. The algorithm splits the binary list image $\overline{v_i}$ into two, adds both halves (in a one step parallel point by point fashion), and repeats until the list is of length 1. If the result is the zero image then $\overline{v_i}$ represents a list of zeros, otherwise $\overline{v_i}$ represents a list with at least one 1.

```
is_vector_of_zeros ( $\overline{v_i}$, $\overline{|v_i|}$; $v_i$_is_zeros )

  // constants: $f_{-1}$, $f_0$, $f_{\frac{1}{2}}$, $f_1$, $f_2$.
  // variables: $n$, flag, $-n$, difference, padded_$\overline{|v_i|}$, $\frac{n}{2}$, $\frac{n}{2}$+1, sum.

    // Generate a list of length $2^n$, where $2^{n-1} < |v_i| \leqslant 2^n$, such that
    // the word $v_i$ is represented by the first $|v_i|$ positions and any
    // remaining positions each represent $f_0$. Call this list padded_$\overline{|v_i|}$.
  $n \leftarrow f_1$
  flag $\leftarrow f_1$
  while ( flag == $f_1$ ) do
    $\cdot$( $n$ , $f_2$ ; $n$ )
    +( $n$ , $f_{-1}$ ; $-n$ )
    +( $\overline{|v_i|}$ , $-n$ ; difference )
    $\rho$( difference , $f_0$ , $f_1$ ; flag )
  end while
  [ 1, $n$, 0, 0 ] $\leftarrow$ $f_0$
  [ 1, $\overline{|v_i|}$, 0, 0 ] $\leftarrow$ $\overline{v_i}$
  padded_$\overline{|v_i|}$ $\leftarrow$ [ 1, $n$, 0, 0 ]

    // Split padded_$\overline{|v_i|}$ into two halves. Add both halves.
    // Repeatedly split and add until list image is of length 1.
  flag $\leftarrow f_1$
  while ( flag == $f_1$ ) do
    $\cdot$( $n$ , $f_{\frac{1}{2}}$ ; $\frac{n}{2}$ )
    +( $\frac{n}{2}$ , 1 ; $\frac{n}{2}$+1 )
    +( [ 1, $\frac{n}{2}$, 0, 0 ] , [ $\frac{n}{2}$+1, $n$, 0, 0 ] ; sum )
    [ 1, $\frac{n}{2}$, 0, 0 ] $\leftarrow$ sum
    $n \leftarrow \frac{n}{2}$
    $\rho$( $n$ , $f_1$ , $f_2$ ; flag )
    +( flag , $f_{-1}$ ; flag )
  end while

    // If sum is the zero image then
    // the original list contained only 0 elements.
  $\rho$( sum , $f_0$ , $f_1$ ; flag )
  if ( flag == 0 ) then
    $v_i$_is_zeros $\leftarrow$ $f_1$
  else
    $v_i$_is_zeros $\leftarrow$ $f_0$
  end if
end // is_vector_of_zeros
```

□

Theorem 5.4.11 (goto m if $V_i = 0$) *The vector machine instruction* goto m if $V_i = 0$ *(or* goto m if $V_i \neq 0$*) is simulated by a* $\mathcal{C}_2$-CSM *in* $O(\log |v_i|)$ TIME*,* $O(|v_i|)$ GRID*,* SPATIALRES*, and* DYRANGE*.*

Proof. Due to the vector machine number representation we have chosen to use, there are exactly two representations for 0 in vectors; the constant sequences $\ldots 000$ and $\ldots 111$. Using our $\mathcal{C}_2$-CSM representation of vectors, if $\overline{|v_i|} = 0$ then the vector V_i is constant, and hence represents 0. We can test $\overline{|v_i|} = 0$ in constant time with an **if** statement.

However, using our image representation of vectors it may be the case that $\overline{|v_i|} = n > 0$ and yet V_i represents 0. In this case $\overline{v_i}$ represents a list of 0s (respectively 1s) and $\overline{\text{sign}(v_i)}$ represents 0 (respectively 1). Linearly searching through $\overline{v_i}$ will require exponential TIME (worst case) and as such is too slow. Instead we use the log TIME technique given by the previous program. In the case that V_i is ultimately 1 we make use of the $\neg(\cdot)$ program defined in Theorem 5.4.3.

```
vector_equals_zero ($\overline{v_i}$, $\overline{|v_i|}$, $\overline{\text{sign}(v_i)}$; $V_i$_equals_zero )

  // constants: $f_{-1}$, $f_0$, $f_{\frac{1}{2}}$, $f_1$, $f_2$.
  // variables: flag, $n$, $-n$, difference, padded_$\overline{|v_i|}$, $\frac{n}{2}$, $\frac{n}{2}$+1, sum.

  $\rho$( $\overline{|v_i|}$ , $f_0$ , $f_1$ ; flag )
  if ( flag == $f_0$ ) then

      // $V_i$ is of length 0, hence $V_i$ represents 0 and we are finished.
    $V_i$_equals_zero $\leftarrow$ $f_1$
  else
      // $V_i$ is of length greater than 0,
      // hence $V_i$ may or may not represent 0.
    if ( $\overline{\text{sign}(v_i)}$ == $f_0$ ) then
      is_vector_of_zeros ($\overline{v_i}$, $\overline{|v_i|}$; $V_i$_equals_zero )
    else
        // $V_i$ is ultimately 1, negate $V_i$ and test if it contains only zeros
      $\neg$ ( $\overline{v_i}$, $\overline{|v_i|}$, $\overline{\text{sign}(v_i)}$; $\neg\,\overline{v_i}$, $\neg\,\overline{|v_i|}$, $\neg\,\overline{\text{sign}(v_i)}$ )
      is_vector_of_zeros ($\neg\,\overline{v_i}$, $\neg\,\overline{|v_i|}$; $V_i$_equals_zero )
    end if
  end if
end // vector_equals_zero
```

For the goto part of the instruction we merely note that in the $\mathcal{C}_2$-CSM gotos are simulated by **if**s and **while**s. In fact, low-level $\mathcal{C}_2$-CSM code simulation of gotos is directly implemented by the *br* operation, in this case m would be replaced by the appropriate grid address.

Clearly the related instruction 'goto m if $V_i \neq 0$' is simulated with the same resource usage. □

At this point we have given a simulation for each index-vector machine operation. To summarise, in shift operations TIME is $O(|V_j|)$ where V_j is the index-vector defining the shift distance. For each other operation TIME is $O(\log |V_{\max}|)$ where $V_{\max}$ is the maximum length vector accessed in the simulation of the operation. The resources GRID, DYRANGE and SPATIALRES are linear in the longest vector accessed.

Given a vector machine M we can design a $\mathcal{C}_2$-CSM M' that simulates M. In particular, if vector machine M recognises a language L then we can easily modify our simulation of vector machines so that M' decides L. Combining this with Lemma 5.2.3 brings us to the following result.

Theorem 5.4.12 *Let M be an index-vector machine that decides $L \in \{0,1\}^*$ in time $T(n)$ for input length n. Then L is decided by a $\mathcal{C}_2$-CSM M' in $O(T^2(n))$ TIME, $O(2^{T(n)})$ GRID, SPATIALRES and DYRANGE.*

Proof. By Lemma 5.2.3 M's index-vectors have length $O(T(n))$, while unrestricted vectors have length $O(2^{T(n)})$. From the simulation theorems in this chapter, any operation with unrestricted vector inputs and outputs only, is simulated in log TIME in the length of the vectors. The remaining operations, right and left shift, are simulated in TIME that is linear in the length of their index-vector input.

From these bounds it is easy to work out that M decides L in $O(T^2(n))$ TIME and that each of GRID, SPATIALRES and DYRANGE is $O(2^{T(n)})$. □

From the previous theorem, M' uses $O(2^{3T(n)})$ SPACE to decide L, which corresponds to a cubic simulation in the space of the index-vector machine M.

Corollary 5.4.13 $\mathcal{V}_I$-TIME$(T(n)) \subseteq \mathcal{C}_2$-CSM-TIME$(O(T^2(n)))$

By Theorem 5.2.4 we get a relationship between space bounded Turing machines and time bounded $\mathcal{C}_2$-CSMs. This gives the main result of this chapter.

Corollary 5.4.14 NSPACE$(S(n)) \subseteq \mathcal{C}_2$-CSM-TIME$(O(S(n) + \log n)^4)$

5.5 Discussion

Our main result was proved by simulation of vector machines, such simulations are relatively rare in the literature [vEB90]. In addition to fulfilling our needs of giving a lower bound on $\mathcal{C}_2$-CSM power, the results in this chapter are useful to the practitioner since we have given a method to directly translate vector machine algorithms to optical algorithms.

The quadratic time bound for index-vector simulation is reasonably tight. In a certain sense this is not surprising, since both are SIMD models. It is probable that the power in the inclusion in Corollary 5.4.14 could be reduced from 4 to 2 by a direct Turing machine simulation.

The simulation uses the reasonable (we argue) natural number representation of images. Using this addressing scheme we incur a DYRANGE cost. For our simulation this cost is a constant times the longest vector. Since the binary vector values are represented by images with constant DYRANGE, it would be interesting if another addressing scheme could be employed that works (say) on binary values (e.g. binary list image addresses). We believe that DYRANGE could be reduced without a significant increase in the other complexity measures. Simultaneously, a constant GRID simulation might be possible, the main problem is to simulate vector shifts while using only a constant number of images. By using these trade-offs we conjecture that the SPACE could be reduced to linear in the space of the simulated index-vector machine, with only a polynomial increase in TIME.

It is interesting to note that we did not make use of Fourier transformation in the vector machine simulation. Optical computers are sometimes celebrated for having a constant time FT operation. Our results from this chapter prove that the $\mathcal{C}_2$-CSM has remarkable power without explicitly using Fourier transformation. However in a more fine grained analysis, say using the $\mathcal{C}_2$-CSM to design algorithms for NC and AC problems, some advantages of Fourier transformation might be observed.

We mentioned earlier that Simon's [Sim77] construction shows that unrestricted vector machines have no more power, up to a polynomial in time, than index-vector machines. If we remove the SPACE upper bound condition from $\mathcal{C}_2$-CSMs then loads and stores, or multiplication and cut-off division, of images are in a certain sense similar to unrestricted vector shift operations. (Both generate huge objects in small time.) We conjecture that Simon's result could be applied to enlarge the class of CSMs that have equivalent power to the $\mathcal{C}_2$-CSM , up to at least $\mathcal{C}_2$-CSMs without the SPACE restriction.

6

Upper bounds on $\mathcal{C}_2$-CSM power

6.1 Introduction

In this chapter we prove an upper bound on the computational power of the $\mathcal{C}_2$-CSM. In particular we show that $\mathcal{C}_2$-CSMs computing in TIME $T(n)$ accept at most the languages accepted by Turing machines in $O(T^2(n))$ space:

$$\mathcal{C}_2\text{-CSM-TIME}(T(n)) \subseteq \text{DSPACE}(O(T^2(n))) \,. \tag{6.1.1}$$

Combining this with the main result from the previous chapter gives the following relationship between TIME on $\mathcal{C}_2$-CSMs and sequential space

$$\text{NSPACE}(S^{O(1)}(n)) = \mathcal{C}_2\text{-CSM-TIME}(S^{O(1)}(n)) \,,$$

and thus the $\mathcal{C}_2$-CSM verifies the parallel computation thesis. We also obtain a characterisation of NC in terms of the $\mathcal{C}_2$-CSM.

The inclusion in Eq. (6.1.1) is proved by giving a logspace uniform family of circuits that simulate $\mathcal{C}_2$-CSMs. The circuits have size that is polynomial in $\mathcal{C}_2$-CSM SPACE and have depth that is quadratic in $\mathcal{C}_2$-CSM TIME. To prepare for such a simulation we give a representation of images as binary words. We then proceed to give a circuit that simulates each $\mathcal{C}_2$-CSM operation. Some final circuits simulate $\mathcal{C}_2$-CSM control flow and configuration change. We begin with a brief overview of circuits and uniformity.

6.2 Uniform circuits

There are many variations on the circuit model of computation. In this work we are using logspace uniform circuits over the complete basis $\wedge$, $\vee$ and $\neg$. We briefly describe the meaning of these terms, for further details the reader is referred to the literature, for example [Sav98, BDG88a, BDG88b].

A circuit is a finite directed acyclic graph. Each node computes a function and is one of three types; gates, inputs or constants. A gate is one of $\wedge$, $\vee$ or $\neg$, each having respective in-degree (or fan-in) of 2, 2 and 1. An input is an element of $\{x_1, \ldots, x_n\}$ and has fan-in of 0. Finally constants are one of 0 or 1 and have fan-in 0. Each node has out-degree (or fan-out) of 1. A subset of the nodes are called the outputs.

In a computation each input node is assigned a value from $\{0, 1\}$. The computation then proceeds in the obvious way through the graph. If there is only one output node then the circuit is a tree and computes a Boolean-valued function.

A circuit family is a set of circuits $\mathcal{C} = \{c_n : n \in \mathbb{N}\}$ where each c_n is a Boolean circuit with exactly n input nodes. A language $L \subseteq \Sigma^*$ is decided by the circuit family $\mathcal{C}_L$ if the characteristic function of the language $L \cap \{0, 1\}^n$ is computed by c_n, for each $n \in \mathbb{N}$.

We analyse circuits in terms of the complexity measures size and depth. The circuit family $\mathcal{C}$ has size complexity $\text{size}(n)$ if and only if for all n, $\text{size}(n)$ is an upper bound on the number of gates in c_n. $\mathcal{C}$ has depth complexity $\text{depth}(n)$ if and only if for all n, $\text{depth}(n)$ is an upper bound on the length of every path in c_n.

Given a circuit c_n, the *gate-wise encoding* of c_n is a string of 4-tuples, where each tuple encodes a single gate and is of the form

$$(g, b, g_l, g_r) \in \big(\{0, 1\}^+,\ \{\wedge, \vee, \neg\},\ \{0, 1\}^+ \cup \varnothing,\ \{0, 1\}^+ \cup \varnothing\big)$$

The tuple encodes a gate by specifying a gate label g, the operation b that the gate computes and the two inputs, g_l and g_r, to the gate. For $\neg$ gates exactly one of g_l or g_r is the special null symbol $\varnothing$.

Without further qualifying this definition we have *nonuniform* circuits. Nonuniform circuits are a powerful model of computation; for each $L \subseteq \{0, 1\}^*$ there is a circuit family $\mathcal{C}_L$ of exponential size that decides the membership problem for L. To see this, notice that our circuits are over a complete basis and so compute any Boolean function. Moreover, for each n the characteristic function of $L \cap \{0, 1\}^n$ is a Boolean function, hence there exists a c_n that computes it. In this chapter we consider only logspace uniform circuits.

Definition 6.2.1 (logspace uniformity) *A circuit family $\mathcal{C}$ is logspace uniform if there exists a transducer Turing machine such that for all n the gate-wise encoding of $\mathcal{C}$ is computable using $\log(\text{size}(c_n))$ workspace.*

When analysing transducer Turing machines we measure the space used by the work tapes only. In many places we make use of the fact that the composition of two logspace reductions is a logspace reduction [Sav98, BDG88a]. We use only one notion of uniformity in this chapter, so we frequently write “uniform” instead of “logspace uniform”.

Let U-SIZE,DEPTH$(\text{size}(n), \text{depth}(n))$ be the class of languages recognised by logspace uniform bounded fan-in circuits of size and depth $\text{size}(n)$ and $\text{depth}(n)$, respectively. The following result is well-known [Bor77, KR90].

Theorem 6.2.2

$$\text{NSPACE}(S^{O(1)}(n)) = \text{U-SIZE, DEPTH}(2^{n^{O(1)}}, S^{O(1)}(n))$$

Circuit depth is a measure of parallel time, hence logspace uniform circuits verify the parallel computation thesis. More precisely [KR90]

$$\text{NSPACE}(S(n)) \subseteq \text{U-SIZE}, \text{DEPTH}(O(2^{S(n)}), O(S^2(n)))$$

and [BDG88b, KR90]

$$\text{U-SIZE}, \text{DEPTH}(2^{S(n)}, S(n)) \subseteq \text{DSPACE}(O(S(n)))\,. \tag{6.2.1}$$

6.2.1 Subcircuits

Previously in Section 6.2 we gave the gate-wise encoding of a circuit as a string of 4-tuples: one tuple for each gate. This encoding is rather cumbersome for our purposes. Below we define the *subcircuit-wise encoding* of a circuit c as a string of tuples. Each tuple encodes a subcircuit of c (a subcircuit is itself a circuit). The tuple also encodes any connections (graph edges) between the subcircuits.

Definition 6.2.3 (subcircuit-wise encoding of a circuit) *Given a circuit c the subcircuit-wise encoding of c is a string of 3-tuples. Each tuple encodes a single subcircuit of c and is of the form*

$$(c_{\text{sub}}, f, (c_{\text{sub}_{\text{in}}}))$$

where $c_{\text{sub}} \in \{0,1\}^+$ is the subcircuit label; f is the function (on binary words) that the subcircuit computes; and $(c_{\text{sub}_{\text{in}}})$ is the sequence of inputs to the subcircuit.

Given a well defined subcircuit-wise encoding where all subcircuits have a well defined gate-wise encoding, then c is well defined. Moreover c_n is logspace uniform if for all n the subcircuit-wise encoding of c_n is logspace computable and for each of c_n's subcircuits the gate-wise encoding is logspace computable.

6.3 Representation

Suppose we are simulating a $\mathcal{C}_2$-CSM M that has TIME, GRID, SPATIALRES, and DYRANGE of $T(n)$, $G(n)$, $R_{\text{S}}(n)$, and $R_{\text{D}}(n)$ respectively. (Recall that in a $\mathcal{C}_2$-CSM both AMPLRES and PHASERES have constant value 2).

From the definition of the $\mathcal{C}_2$-CSM we have an upper bound on the SPACE $S(n)$ of M in terms of TIME $T(n)$

$$\begin{aligned} S(n) &= G(n) R_{\text{S}}(n) R_{\text{D}}(n) 4 \\ &\leqslant c_G 2^{T(n)} c_{R_{\text{S}}} 2^{T(n)} c_{R_{\text{D}}} 2^{T(n)} \end{aligned}$$

where constants c_G, c_{R_S} and c_{R_D} depend on the program. We *redefine* the TIME of M to be

$$T'(n) = \left\lceil \log(c2^{T(n)}) \right\rceil$$

for $c = \max(c_G, c_{R_\mathrm{S}}, c_{R_\mathrm{D}})$. Essentially this lengthens the computation of M by an amount proportional to the constants in the above expression. This trick appears in [BDG88b]. It saves us the bother of carrying around extra constants in the our simulations and uniformity proofs. In the sequel we write $T'(n)$ as $T(n)$. Now that $T(n)$ has been redefined to be $T'(n)$, SPACE is bounded above by

$$2^{T(n)} 2^{T(n)} 2^{T(n)} ,$$

specifically each of GRID, SPATIALRES and DYRANGE is bounded above by $2^{T(n)}$.

We now make another adjustment to the complexity of M. First, let

$$\begin{aligned} G(n) &= G_x(n) G_y(n) , \\ R_\mathrm{S}(n) &= R_{\mathrm{S}_x}(n) R_{\mathrm{S}_y}(n) , \end{aligned}$$

where $G_x(n)$ and $G_y(n)$ are the number of images in the horizontal and vertical directions respectively, and where $R_{\mathrm{S}_x}(n)$ and $R_{\mathrm{S}_y}(n)$ are the number of pixels in the horizontal and vertical directions respectively. We define

$$G_x(n) = G_y(n) = R_{\mathrm{S}_x}(n) = R_{\mathrm{S}_y}(n) = 2^{T(n)} .$$

So now GRID and SPATIALRES have both been increased while DYRANGE has not changed

$$\begin{aligned} G(n) = R_\mathrm{S}(n) &= 2^{2T(n)} , \\ R_\mathrm{D}(n) &= 2^{T(n)} . \end{aligned} \tag{6.3.1}$$

We get our final upper bound on M's SPACE

$$S(n) \leqslant 2^{2T(n)} 2^{2T(n)} 2^{T(n)} = 2^{5T(n)} .$$

Notice that these adjustments do not affect M's computation, we have simply defined M to be more TIME and SPACE inefficient.

6.3.1 Word representation of $\mathcal{C}_2$-CSM grid

In the circuit simulation M's grid is represented by a single binary word $\mathcal{G}$. As M's configurations vary over TIME, the contents of $\mathcal{G}$ vary, however $\mathcal{G}$ never changes in length: from the beginning of the simulation we define $\mathcal{G}$ to represent the most 'complex' grid for all TIME. This point will be further clarified after we define the structure of $\mathcal{G}$ with respect to M.

The word $\mathcal{G}$ is composed of $G(n)$ *image subwords* of equal length, if M has GRID $G(n)$ then

$$\mathcal{G} = \mathcal{G}_0\mathcal{G}_1 \cdots \mathcal{G}_{G(n)-1} .$$

For each image i in M there is an image subword $\mathcal{G}_i$ and vice versa. To represent M's 2D array of images as a word, we order the images first horizontally and then vertically; beginning with the lower leftmost image, proceeding left to right and then bottom to top. Then the order of the image subwords in $\mathcal{G}$ is given by

$$\begin{aligned}\mathcal{G} = {} & \mathcal{G}_0\mathcal{G}_1 \cdots \mathcal{G}_{G_x(n)-1} \\ & \mathcal{G}_{G_x(n)}\mathcal{G}_{G_x(n)+1} \cdots \mathcal{G}_{2G_x(n)-1} \\ & \mathcal{G}_{2G_x(n)}\mathcal{G}_{2G_x(n)+1} \cdots \mathcal{G}_{3G_x(n)-1} \\ & \vdots \\ & \mathcal{G}_{(G_y(n)-1)G_x(n)}\mathcal{G}_{(G_y(n)-1)G_x(n)+1} \cdots \mathcal{G}_{G_y(n)G_x(n)-1} .\end{aligned}$$

For the purpose of readability the single word $\mathcal{G}$ is split over multiple lines.

Next we show how each pixel in image i is represented in the image subword $\mathcal{G}_i$. This representation scheme is analogous to the previous representation of images as subwords. The image subword $\mathcal{G}_i$ is composed of $R_{\text{S}}(n)$ *pixel subwords* of equal length.

$$\mathcal{G}_i = \mathcal{G}_i[0]\mathcal{G}_i[1] \cdots \mathcal{G}_i[R_{\text{S}}(n) - 1]$$

For each pixel j in image i there is a pixel subword $\mathcal{G}_i[j]$ and vice versa. Analogous to the ordering on images, we order the pixels first by the horizontal direction and then by the vertical direction, beginning with the lower leftmost pixel. Then the order of the pixel subwords in $\mathcal{G}_i$ is given by

$$\begin{aligned}\mathcal{G}_i = {} & \mathcal{G}_i[0]\mathcal{G}_i[1] \cdots \mathcal{G}_i[R_{\text{S}_x}(n) - 1] \\ & \mathcal{G}_i[R_{\text{S}_x}(n)]\mathcal{G}_i[R_{\text{S}_x}(n) + 1] \cdots \mathcal{G}_i[2R_{\text{S}_x}(n) - 1] \\ & \vdots \\ & \mathcal{G}_i[(R_{\text{S}_y} - 1)R_{\text{S}_x}(n)]\mathcal{G}_i[(R_{\text{S}_y} - 1)R_{\text{S}_x}(n) + 1] \cdots \mathcal{G}_i[R_{\text{S}_y}(n)R_{\text{S}_x}(n) - 1] .\end{aligned}$$

The single word $\mathcal{G}_i$ is split over multiple lines for readability.

From Definition 3.3.1 each $\mathcal{C}_2$-CSM has constant AMPLRES of 2 and constant PHASERES of 2. Given DYRANGE of $R_{\text{D}}(n)$ it follows directly that the value (or range) of each pixel in a $\mathcal{C}_2$-CSM configuration is from the set $\{0, \pm\frac{1}{2}, \pm 1, \pm\frac{3}{2}, \ldots, \pm R_{\text{D}}(n)\}$. To represent this set as a set of binary words we use the 2's complement binary representation of integers [Sav76], with a slight modification: the binary sequence is shifted by one place to take care of the halves. For example Table 6.3.1 gives the binary representations for a $\mathcal{C}_2$-CSM with DYRANGE $R_{\text{D}}(n) = 2\frac{1}{2}$.

pixel value	2's complement	pixel value	2's complement
0	0000		
$\frac{1}{2}$	0001	$-\frac{1}{2}$	1111
1	0010	-1	1110
$1\frac{1}{2}$	0011	$-1\frac{1}{2}$	1101
2	0100	-2	1100
$2\frac{1}{2}$	0101	$-2\frac{1}{2}$	1011

Table 6.3.1: The 2's complement binary representations for pixel values in a $\mathcal{C}_2$-CSM with constant DYRANGE of $R_{\mathrm{D}}(n) = 2\frac{1}{2}$.

At this point we have defined the entire structure of the word $\mathcal{G}$ representing M's grid. The length of $\mathcal{G}$ is $|\mathcal{G}| = R_{\mathrm{S}}(n)G(n)\lceil\log(4R_{\mathrm{D}}(n)+1)\rceil$. Hence if M's SPACE complexity is $O(S(n))$ then $|\mathcal{G}|$ is also $O(S(n))$. Substituting for Eq. (6.3.1) allows us to define $|\mathcal{G}|$ in terms of TIME

$$|\mathcal{G}| = 2^{2T(n)}2^{2T(n)}\left\lceil\log(4\cdot 2^{T(n)}+1)\right\rceil .$$

Specifically the length of each pixel subword is

$$|\mathcal{G}_i[j]| = \left\lceil\log(4\cdot 2^{T(n)}+1)\right\rceil$$

and the length of each image subword is

$$|\mathcal{G}_i| = 2^{2T(n)}\left\lceil\log(4\cdot 2^{T(n)}+1)\right\rceil \tag{6.3.2}$$

These expressions will be useful for giving bounds on circuit complexity in terms of TIME.

To simulate the $T(n)$ computation steps of the $\mathcal{C}_2$-CSM M we will design a uniform circuit c_M that is of size $S^{O(1)}(n)$ and depth $O(T^2(n))$.

6.3.2 Word representation of $\mathcal{C}_2$-CSM configuration

A $\mathcal{C}_2$-CSM configuration $\langle c, e\rangle$ (see Definition 2.2.3) is represented as a binary word, which we write[1] as $(\mathrm{ctrl}, \mathcal{G})$ where ctrl is a binary word of length $2T(n)$ and $\mathcal{G}$ is as given above. Usually we interpret the 'instruction pointer' ctrl as a number that indexes the location of the next $\mathcal{C}_2$-CSM operation.

6.3.3 Word representation of $\mathcal{C}_2$-CSM constants and addresses

In order to simulate the $\mathcal{C}_2$-CSM operations *ld*, *st* and *br*, the circuit simulating a $\mathcal{C}_2$-CSM must somehow simulate or compute the address encoding

[1]The configuration $(\mathrm{ctrl}, \mathcal{G})$ would be written as "ctrl$\mathcal{G}$" in an actual circuit simulation.

function $\mathfrak{C} : \mathbb{N} \rightarrow \mathcal{N}$. In the $\mathcal{C}_2$-CSM definition we said that $\mathfrak{C}$ is decidable by a logspace Turing machine. Hence $\mathfrak{C}$, and its inverse, are computable by a logspace transducer Turing machine.

The circuit has access to the first $2^{T(n)} = G_x(n) = G_y(n)$ elements of $\mathfrak{C}$, given explicitly as a word. Specifically, for the range $\mathfrak{C}$ we write

$$\mathfrak{C}_{2^{T(n)}} = \mathrm{addr}_0 \mathrm{addr}_1 \cdots \mathrm{addr}_{2^{T(n)}-1} \tag{6.3.3}$$

where $\mathfrak{C}_{2^{T(n)}}(i) = \mathrm{addr}_i$ is called *address image word i*. Image words and address image words have the same length. The domain of $\mathfrak{C}$ is a list of (represented) numbers written as

$$\mathfrak{C}^{-1}_{2^{T(n)}} = 0^{T(n)}, 0^{T(n)-1}1, 0^{T(n)-2}10, \cdots, 1^{T(n)} \tag{6.3.4}$$

where the commas are for human-readability purposes only.

6.4 Circuit simulation of $\mathcal{C}_2$-CSM

We simulate a language deciding $\mathcal{C}_2$-CSM M, where M's input word w is of length n. M is simulated by the circuit c_M in the following way. At the first step of the simulation the circuit c_M is presented with the input word

$$(\mathrm{ctrl}_{\mathrm{sta}}, \mathcal{G}_{\mathrm{sta}})$$

that represents M's initial configuration (including M's input). The circuit c_M has $T(n)$ layers numbered 0 (the input layer) to $T(n) - 1$ (the output layer), each layer is composed of a number of subcircuits.

We give a separate simulation for each of the CSM's operations (the operations were given in Definition 2.2.4 and Figure 2.2.1). Each $\mathcal{C}_2$-CSM operation *op* is simulated by a circuit c_{op}, where the circuit encoding function $1^n \rightarrow \overline{c}_{op}$ is computable by a transducer Turing machine in space $\log \mathrm{size}(\overline{c}_{op})$.

6.4.1 Circuits computing $+$, $\cdot$, $*$, ρ, h and v

We begin by simulating the $+$ operation. Addition of two nonnegative integers written in binary is performed by an unbounded fan-in circuit of size $O(m^2)$ and constant depth and so is an AC^0 problem (the algorithm is called the carry look-ahead and is well known, for example see [KR90] or [Vol99]). Hence this problem is also in NC^1. Krapchenko [Kra70], and Ladner and Fischer [LF80] give tighter NC^1 adders that have depth $O(\log m)$ and lower the size bound to $O(m)$ [Weg87].

Theorem 6.4.1 (circuit simulation of $+$) *The $\mathcal{C}_2$-CSM operation $+$ is simulated by a logspace uniform circuit $c_{a:=a+b}$ of size $O(2^{2T(n)}T(n))$ and depth $O(\log T(n))$.*

Proof. Circuit: To add two pixel words we use Ladner and Fischer's NC^1 addition algorithm mentioned above. Extending this algorithm to work for the 2's complement binary representation is straightforward: the addition circuit is augmented by having four separate cases depending on whether both, or only one of, the inputs are positive or negative (for example see [Sav76]). Circuit size is thus increased by a multiplicative constant of 4 and a small additive constant, while circuit depth is increased only by a small additive constant.

Recall that the $\mathcal{C}_2$-CSM operation $+$ adds images a and b in a parallel point by point fashion and places the result in a. The circuit $c_{a:=a+b}$ has one adder subcircuit for each pixel j in a. Under the representation scheme described above, pixel word $\mathcal{G}_a[j]$ is added to pixel word $\mathcal{G}_b[j]$, resulting in a new word $\mathcal{G}_{a'}[j]$. The circuit $c_{a:=a+b}$ consists of $2^{2T(n)}$ adders and has the subcircuit-wise encoding

$$\overline{c}_{a:=a+b} = \left\{ \, (\, \mathcal{G}_{a'}[j],\ \text{adder}_j,\ (\mathcal{G}_a[j],\ \mathcal{G}_b[j])\,)\ \middle|\ 0 \leqslant j \leqslant 2^{2T(n)} - 1 \right\} .$$

The outputs are ordered by j. Each adder subcircuit has $O(\log p)$ depth and size $O(p)$ where $p = \lceil \log(4 \cdot 2^{T(n)} + 1) \rceil$ is pixel word length. Since there are $2^{2T(n)}$ adders (and each has size $O(T(n))$) the circuit has size $O(2^{2T(n)}T(n))$. The circuit has depth

$$O(\log p) = O\left(\log \left\lceil \log(4 \cdot 2^{T(n)} + 1) \right\rceil \right) = O(\log T(n))$$

Uniformity: We show that a logspace transducer can compute $1^n \to \overline{c}_{a:=a+b}$. Recall that we count only the workspace for transducers. At any step in the computation the transducer will have at most the encoding of the current gate and a constant number of counters on its worktapes. Since the circuit has $O(2^{2T(n)}T(n))$ gates, each of the gate labels in $\overline{c}_{a:=a+b}$ has length $O(T(n))$. Hence each gate label is computable in space $\log |\overline{c}_{a:=a+b}| = O(T(n))$. There is a counter for the number of the current gate and another for the number of the current adder subcircuit. Each adder subcircuit is logspace uniform and hence a constant number of counters is sufficient to construct each one. Again each of these counters has length $O(T(n))$. All gates and counters are computable in space $\log |\overline{c}_{a:=a+b}|$. □

As with addition there are NC^1 circuits for multiplication of binary numbers [KR90]. The best NC circuit is Schönage and Strassen's [SS71] NC^1 circuit, which uses the DFT. It has size $O(m \log m \log\log m)$ and depth $O(\log m)$. Unlike addition, Furst, Saxe and Sipser [FSS84] showed that multiplication is not in AC^0, by showing that parity is not in AC^0 [BS90]. In the following theorem we make use of the above NC^1 multiplication circuit. We omit the proof as it is almost the same as the previous one: the only difference is that we use a polynomial sized NC^1 multiplication circuit as opposed to the linear sized NC^1 adder we used previously.

Theorem 6.4.2 (circuit simulation of $\cdot$) *The $\mathcal{C}_2$-CSM operation $\cdot$ is simulated by a logspace uniform circuit $c_{a:=a\cdot b}$ of size $O(2^{2T(n)}T^2(n))$ and depth $O(\log T(n))$.*

Each pixel in a $\mathcal{C}_2$-CSM is integer valued, hence the $*$ (complex conjugation) operation is the identity function. So we have the following easy theorem.

Theorem 6.4.3 (circuit simulation of $*$) *The $\mathcal{C}_2$-CSM operation $*$ is simulated by a logspace uniform circuit $c_{a:=a^*}$ of size $O(2^{2T(n)})$ and constant depth.*

Proof. The circuit is the identity circuit $(\neg\neg a)$ which is logspace uniform.□

Theorem 6.4.4 (circuit simulation of ρ) *The $\mathcal{C}_2$-CSM operation ρ is simulated by a logspace uniform circuit $c_{a:=\rho(a,z_{\mathrm{l}},z_{\mathrm{u}})}$ of size $O(2^{2T(n)}T^2(n))$ and depth $O(\log T(n))$.*

Proof. Circuit: First we build a circuit $c_{u>v}$ that tests if the integer represented by one pixel word is greater than another. It is straightforward to give constant size and depth circuits that tell if two bits $b_1, b' \in \{0,1\}$ are equal and if one is greater than the other: The circuits $c_{b\equiv b'}$ and $c_{b>b'}$ respectively compute the $\equiv$ and $>$ expressions

$$\begin{aligned} b \equiv b' &= (b \wedge b') \vee (\neg b \wedge \neg b')\,, \\ b > b' &= b \wedge \neg b'\,. \end{aligned}$$

Using these we define the following Boolean expression on pixel words u and v.

$$u > v = \bigvee_{m=0}^{|u-1|} \left((u_m > v_m) \wedge \left(\bigwedge_{k=0}^{m-1} (u_k \equiv v_k) \right) \right) .$$

We build $c_{u>v}$ as follows. For each m the circuit computing $\bigwedge_{k=0}^{m-1} (u_k \equiv v_k)$ is realised as a $O(m)$ size, $O(\log m)$ depth tree. There are $|u|$ such trees, the root of each will have a $\wedge$ test with the constant depth circuit for $u_m > v_m$. Then we take the OR of these ANDs using a $\log |u|$ depth OR tree. Therefore the circuit $c_{u>v}$ has $O(|u|^2)$ size and $O(\log |u|)$ depth.

Using a similar construction we build the circuit $c_{u<v}$, this has the same complexity as $c_{u>v}$. Moreover these circuits would be extended to work on the 2's complement representation with only a multiplicative constant increase in size and additive constant increase in depth. We combine these circuits to create the pixel thresholding circuit $c_{v:=\rho(v,l,u)}$ that evaluates to l if $c_{v<l} \equiv 1$, to u if $c_{v>u} \equiv 1$ and to v otherwise.

Recall that the operation $\rho(a, z_{\mathrm{l}}, z_{\mathrm{u}})$ thresholds image a in a parallel point by point fashion and places the result in a. The circuit $c_{a:=\rho(a,z_{\mathrm{l}},z_{\mathrm{u}})}$

has one pixel thresholding subcircuit for each pixel j in a. Pixel word $\mathcal{G}_a[j]$ is thresholded below by pixel word $\mathcal{G}_{z_\mathrm{l}}[j]$ and above by pixel word $\mathcal{G}_{z_\mathrm{u}}[j]$ resulting in a new word $\mathcal{G}_{a'}[j]$. The circuit $c_{a:=\rho(a,z_\mathrm{l},z_\mathrm{u})}$ consists of $2^{2T(n)}$ parallel pixel thresholding subcircuits and is encoded as

$$\overline{c}_{a:=\rho(a,z_\mathrm{l},z_\mathrm{u})} = \left\{ (\,\mathcal{G}_{a'}[j],\ \mathrm{thresh}_j,\ (\mathcal{G}_a[j],\ \mathcal{G}_{z_\mathrm{l}}[j],\ \mathcal{G}_{z_\mathrm{u}}[j]))\ \middle|\ 0 \leqslant j \leqslant 2^{2T(n)} - 1 \right\}$$

ordered by j. Each thresh_j subcircuit has $O(\log p)$ depth and size $O(p^2)$ where $p = \lceil \log(4 \cdot 2^{T(n)} + 1) \rceil$ is pixel word length. Since there are $2^{2T(n)}$ pixel thresholding subcircuits (each has size $O(T^2(n))$) the circuit has size $O(2^{2T(n)}T^2(n))$. The entire circuit has depth

$$O(\log p) = O\left(\log \left\lceil \log(4 \cdot 2^{T(n)} + 1) \right\rceil \right) = O(\log T(n))$$

Uniformity: By stepping through the construction and applying the arguments given in Theorem 6.4.1 we see that the length of each gate label is bounded by $O(T(n))$ and a constant number of variables is sufficient to construct the circuit encoding. □

Next we give the simulations of the $\mathcal{C}_2$-CSM operations h and v. Recall that in the definition of the $\mathcal{C}_2$-CSM, operations h and v compute the DFT in the horizontal and vertical directions respectively. Fast Fourier transform (FFT) circuits are used to simulate h and v. (The FFT algorithm has a long history, but its entrance into computer science is usually cited to [CT65].) With input length m the FFT circuit is uniform and has size bounded above by $2m \log m$ and depth bounded above by $2 \log m$ (see [Sav98] for details).

Theorem 6.4.5 (circuit simulation of h) *The $\mathcal{C}_2$-CSM horizontal DFT operation h is simulated by a uniform circuit $c_{a:=h(a)}$ of size $O(2^{2T(n)}T(n))$ and depth $O(T(n))$.*

Proof. For each row of pixel words in image word a, the circuit $c_{a:=h(a)}$ has a single FFT subcircuit. The output is an image word a' such that each row in a' is the DFT of the same row in a. It is straightforward to verify the size, depth and uniformity conditions. □

The circuit $c_{a:=v(a)}$ that simulates $\mathcal{C}_2$-CSM vertical DFT operation v is constructed in a similar manner to $c_{a:=h(a)}$, except we replace the word “row” with “column”. We omit the proof of the following theorem.

Theorem 6.4.6 (circuit simulation of v) *The $\mathcal{C}_2$-CSM horizontal DFT operation v is simulated by a uniform circuit $c_{a:=v(a)}$ of size $O(2^{2T(n)}T(n))$ and depth $O(T(n))$.*

6.4.2 Circuits computing *ld* and *st*

From Eq. (6.3.4) each address image i is encoded by a length $T(n)$ binary number. In the following lemma we show how to encode i into an image word.

Lemma 6.4.7 ($c_{\mathfrak{E}(i)}$) *A binary number $i \in \{0,1\}^{T(n)}$ is encoded to an address image word by a uniform circuit $c_{\mathfrak{E}(i)}$ of size $O(2^{3T(n)}T^2(n))$ and depth $O(T(n))$.*

Proof. Circuit $c_{\mathfrak{E}(i)}$ takes i and $\mathfrak{E}_{2^{T(n)}}$ as input. Circuit $c_{\mathfrak{E}(i)}$ uses $2^{T(n)}$ parallel equality testing circuits to test i for equality with each number $j \in \{0,\dots,2^{T(n)}-1\}$. This initial part of $c_{\mathfrak{E}(i)}$ has size $O(2^{T(n)}T(n))$ and depth $O(\log T(n))$.

Recall the representation of image words $\mathrm{addr_j}$ in Eq. (6.3.3). For each j, the output of equality test j is ANDed with each symbol of $\mathrm{addr_j}$ (using $|\mathrm{addr_j}|$ AND gates for each j). Note that we are using fan-out > 1 here, we decrease the fan-out to 1 by repeating the input $2^{T(n)}T(n)$ times. At this stage of the computation we have a word that is composed of $2^{T(n)}$ subwords in the following way

$$0^{|\mathrm{addr}_0|}0^{|\mathrm{addr}_1|}\dots0^{|\mathrm{addr}_{i-1}|}\,\mathrm{addr}_i\,0^{|\mathrm{addr}_{i+1}|}\dots0^{|\mathrm{addr}_{2^{T(n)}-1}|}\,.$$

To 'extract' the address image word addr_i we simply OR the k^{th} symbol from each subword (in parallel) using k OR trees.

A constant number counters, each of length $O(T(n))$, is sufficient for a transducer TM to compute $1^n \to \overline{c}_{\mathfrak{E}(i)}$, hence the circuit is uniform. The circuit (using only fan-out 1 gates) has size $O(2^{3T(n)}T^2(n))$ and depth $O(T(n))$. □

We can efficiently test equality of images as the next lemma shows.

Lemma 6.4.8 ($c_{i\equiv i'}$) *Equality testing of two image words i and i' is computable by a uniform circuit $c_{i\equiv i'}$ of size $O(2^{2T(n)}T(n))$ and depth $O(T(n))$.*

Proof. For each bit b in i and i' we use a $i_b \equiv i'_b$ circuit (see Theorem 6.4.4). From Eq. (6.3.2) there are $|i| = 2^{2T(n)}\left\lceil\log(4\cdot2^{T(n)}+1)\right\rceil$ such b and hence the same number of circuits. The outputs of these circuits are combined in an OR tree. The circuit has size $O(2^{2T(n)}T(n))$ and depth $O(T(n))$.

Clearly, an OR tree on the output of bit equality tests is uniform. □

The technique used in in Lemma 6.4.7 could be called 'selecting' a single address word from a sequence of address words. This technique is used, with only slight modifications, to construct a circuit that 'selects' an image word from the grid word $\mathcal{G}$. We can construct a uniform circuit $c_{\mathrm{selectImage}(i)}$

that inputs $\mathcal{G}$ and a binary number i, and outputs (or selects) the i^{th} image word $\mathcal{G}_i$. To test images for equality $c_{\text{selectImage}(i)}$ uses the circuit from Lemma 6.4.8. $c_{\text{selectImage}(i)}$ has size $O(2^{4T(n)}T(n))$ and depth $O(T(n))$.

In the same way we can construct a circuit $c_{\text{selectPixel}(j)}$ that 'selects' a pixel word from the grid word $\mathcal{G}$. This uniform circuit takes a binary number j and the grid word $\mathcal{G}$ as input. It outputs the j^{th} pixel word in the grid word $\mathcal{G}$. $c_{\text{selectPixel}(j)}$ has size $O(2^{4T(n)}T(n))$ and depth $O(\log T(n))$. This circuit will be used in our *ld* simulation.

Equality testing of image words will be a useful tool in our simulation of the *ld* addressing mechanism. Specifically, it is used in the next circuit that computes the function $\mathfrak{E}^{-1}_{2^{T(n)}}(\text{addr}_i)$ for some input address image word addr_i [see (6.3.4)].

Lemma 6.4.9 **($c_{\mathfrak{E}^{-1}(\text{addr}_i)}$)** *An address image word* addr_i *is decoded to a binary number i by a uniform circuit* $c_{\mathfrak{E}^{-1}(\text{addr}_i)}$ *of size* $O(2^{2T(n)}T(n))$ *and depth* $O(T(n))$.

Proof. (Sketch). The circuit $c_{\mathfrak{E}^{-1}(\text{addr}_i)}$ is constructed in manner very similar to $c_{\mathfrak{E}(i')}$ in Lemma 6.4.7, The overall layout of the circuit is the same, only now we are computing the inverse function so we swap binary numbers for image address words. Additionally we use the circuit from Lemma 6.4.8 to do the initial image equality test. □

A naïve simulation of *ld* might simply copy the rectangle of image words to be loaded into the image word a. This works fine when the rectangle consists of exactly one image: The rectangle word and the image word a are of the same length. However the rectangle to be loaded may contain up to $R_{\text{S}}(n)G(n) = 2^{4T(n)}$ pixel words. Image word a contains exactly $2^{2T(n)}$ pixel words. Copying all $2^{4T(n)}$ pixel words to $\mathcal{G}_a$ will cause the length of $\mathcal{G}$ to increase, causing $\mathcal{G}$ to lose its structure. However, it turns out that there is a simple method to sidestep this problem.

Lemma 6.4.10 *In the computation of* $\mathcal{C}_2$*-CSM* M*, any rectangle of images defined by the ld or st parameters contains at most* $2^{2T(n)}$ *distinct image values.*

Proof. For *ld* the proof is by contradiction. From Eq. (6.3.1) M has SPATIALRES $2^{2T(n)}$. Let the rectangle being loaded be defined by the grid coordinates (ξ_1, η_1), (ξ_2, η_2). Let image f be the union of these $(\xi_2 - \xi_1 + 1)(\eta_2 - \eta_1 + 1)$ images. Suppose there strictly greater than $2^{2T(n)}$ values in the range of f. After the *ld* operation, image a has SPATIALRES strictly greater than $2^{2T(n)}$, which is a contradiction.

For *st* we simply observe that a contains at most $2^{2T(n)}$ pixels, hence the stored rectangle contains the same number of distinct image values. □

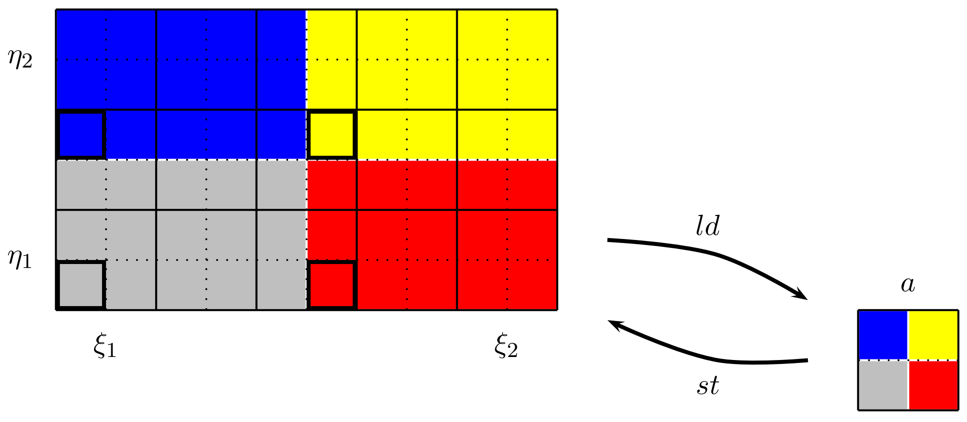

Figure 6.4.1: Illustration of Lemma 6.4.10. The SPATIALRES in this example is $R_{\rm S} = R_{{\rm S}_x} \cdot R_{{\rm S}_y} = 2 \cdot 2 = 4$. The pixel with the lowest index in each *distinct region* of the ld rectangle is highlighted.

So we have only to select $R_{\rm S}(n) = 2^{2T(n)}$ representative pixel words out of the total $2^{2T(n)}(\xi_2 - \xi_1 + 1)(\eta_2 - \eta_1 + 1)$. We choose the pixel with the lowest index in each (possibly) distinct region. This is illustrated in Figure 6.4.1. For each pixel word i in image word a, the pixel j to be loaded is defined as

$$j = \text{col}(i) + \text{row}(i) \cdot R_{{\rm S}_x} \cdot G_x \tag{6.4.1}$$

such that

$$\begin{aligned}\text{col}(i) &= (i \mod R_{{\rm S}_x})(\xi_2 - \xi_1 + 1) + (R_{{\rm S}_x}\xi_1)\\ \text{row}(i) &= \left\lfloor \frac{i}{R_{{\rm S}_x}} \right\rfloor (\eta_2 - \eta_1 + 1) + (R_{{\rm S}_y}\eta_1)\end{aligned}$$

and as usual $R_{{\rm S}_y} = R_{{\rm S}_y} = G_x = 2^{T(n)}$.

Lemma 6.4.11 **($c_{\text{pixelPosition}}$)** *The circuit $c_{\text{pixelPosition}}$ that evaluates Eq.* (6.4.1) *for a single i has size $O(2^{2T(n)}T(n))$ and depth $O(T(n))$, and is uniform. The circuit $c_{\text{pixelPosition}}$ takes the following binary number inputs: i, ξ_1, η_1, ξ_2, η_2. The output is a binary number j defined by Eq.* (6.4.1)*.*

Proof. We have already mentioned that $+$, $-$ and $\cdot$ have NC^1 circuits.

$2^{T(n)}$ is a constant for $c_{\text{pixelPosition}}$ so it is not difficult to write a uniform circuit for $i \bmod 2^{T(n)}$ (the $T(n)$ least significant bits of i are left untouched and the rest are set to 0 by a linear size, constant depth circuit).

The value i is bounded above by $2^{2T(n)}$ hence $\lfloor \frac{i}{2^{T(n)}} \rfloor \leqslant 2^{T(n)}$. So we can multiply $2^{T(n)}$ by each of $\{0, \dots, 2^{T(n)}\}$ (in parallel), then extract the smallest result that is greater than or equal to i. The circuit that computes $\left\lfloor \frac{i}{2^{T(n)}} \right\rfloor$ is constructed in a uniform way using multipliers; $\geqslant$ circuits; and AND trees. The size and depth are bounded above by $O(2^{2T(n)}T(n))$ and $O(T(n))$ respectively.

Given that we can compute each operation of Eq. (6.4.1) with a uniform circuit we can make a circuit for the entire expression. We have explicit bounds on the complexity of all operations so its easy to verify that $c_{\text{pixelPosition}}$ has size $O(2^{2T(n)}T(n))$ and depth $O(T(n))$. □

We have gathered enough tricks to give the simulation of ld.

Theorem 6.4.12 (circuit simulation of ld by $c_{a:=[(\xi'_1,\eta'_1),(\xi'_2,\eta'_2)]}$) *The $\mathcal{C}_2$-CSM operation $ld\,(\xi'_1,\ \eta'_1,\ \xi'_2,\ \eta'_1)$ is simulated by the uniform circuit $c_{a:=[(\xi'_1,\eta'_1),(\xi'_2,\eta'_2)]}$ of size $O(2^{6T(n)}T(n))$ and depth $O(T(n))$. This circuit takes as input the image words ξ'_1, η_1, ξ'_2 and η'_2, and outputs an image word a.*

Proof. The address image words ξ'_1, η_1, ξ'_2, η'_2 are decoded into four binary number words by four circuits, each computes one of

$$\begin{aligned} \mathfrak{E}^{-1}_{2^{T(n)}}(\xi'_1) &= \xi_1\,, & \mathfrak{E}^{-1}_{2^{T(n)}}(\eta'_1) &= \eta_1\,, \\ \mathfrak{E}^{-1}_{2^{T(n)}}(\xi'_2) &= \xi_2\,, & \mathfrak{E}^{-1}_{2^{T(n)}}(\eta'_2) &= \eta_2\,, \end{aligned}$$

as given by Lemma 6.4.9. In the next step we want to select $2^{2T(n)}$ pixels (as illustrated in Figure 6.4.1) from the grid word $\mathcal{G}$. For each pixel i in image word a we have a subcircuit. Subcircuit i gives i as input to $c_{\text{pixelPosition}}$ (Lemma 6.4.11) which outputs j (the index of the pixel we want to 'load'). Note that the circuit $c_{\text{pixelPosition}}$ also takes the values ξ_1, η_1, ξ_2, η_2 as input. The binary number j is passed to $c_{\text{selectPixel}}$ (given in the text). The output of subcircuit i represents the i^{th} pixel in image a after a ld operation.

The proof mentions only three subcircuits, each is of which is uniform. Furthermore these subcircuits are reused and connected in a uniform way. The size and depth conditions are straightforward to verify since we have shown the size and depth of all subcircuits. □

Each of the operations simulated so far affects only the contents of image a. Next we simulate st, this differs in the fact that a *rectangle* of images defined by the coordinates (ξ_1, η_1) and (ξ_2, η_2) is affected.

Before giving the simulation of st we define [in a manner similar to Eq. (6.4.1) used for ld] the set of pixels that are stored to. Let i be the index of a pixel word in image word a. From a single i, the index j of each pixel word that will be stored to, satisfies

$$j = (\text{col}(i) + u) + (\text{row}(i) + v) \cdot R_{\text{S}_x} \cdot G_x, \qquad \begin{array}{r} \text{for } 0 \leqslant u \leqslant \xi_2 - \xi_1, \\ \text{and } 0 \leqslant v \leqslant \eta_2 - \eta_1. \end{array} \tag{6.4.2}$$

Functions col(i) and row(i) were given in Eq. (6.4.1) and $R_{\mathrm{S}_x} = G_x = 2^{T(n)}$. For example, in Figure 6.4.1; if $i = 0$ (the lowest and leftmost pixel in a) then the grey rectangle[2] of pixels satisfy Eq. (6.4.2).

In the next lemma we give a circuit that computes a mask representing the (rectangle of) pixel words j that a given pixel word i in a is stored to.

Lemma 6.4.13 ($c_{st\mathrm{Pixel}(i)}$) *The circuit $c_{st\mathrm{Pixel}(i)}$ has size $O(2^{6T(n)}T^5(n))$ and depth $O(T(n))$, and is uniform. The circuit $c_{st\mathrm{Pixel}(i)}$ takes the following binary number inputs: i, ξ_1, η_1, ξ_2, η_2, and outputs a binary word m (called a mask) whose j'^{th} symbol is 1 if and only if j' satisfies Eq.* (6.4.2) *for these inputs.*

Proof. The circuit $c_{st\mathrm{Pixel}(i)}$ evaluates Eq. (6.4.2) for a single $i \in \{0, \dots, 2^{2T(n)}\}$. It consists of $2^{4T(n)}$ subcircuits (one for each pixel word in $\mathcal{G}$). For each $j' \in \{0, \dots, 2^{4T(n)} - 1\}$ there is a subcircuit that evaluates Eq. (6.4.2) for all $u', v' \in \{0, \dots, 2^{T(n)} - 1\}$ that satisfy the conditions on u and v respectively. If and only if the result of this evaluation equals j' then a 1 is output (j' satisfies Eq. (6.4.2) for the given i).

The results from each $j' \in \{0, \dots, 2^{4T(n)} - 1\}$ are output as a single word m, $|m| = 2^{4T(n)}$. For the given i the mask m has the property that symbol $m'_j = 1$ if and only if j' satisfies Eq. (6.4.2).

To evaluate Eq. (6.4.2) we make use of the circuit given in Lemma 6.4.11. By stepping through the construction of $c_{st\mathrm{Pixel}(i)}$ we get size $O(2^{6T(n)}T^5(n))$ and depth $O(T(n))$. The uniformity follows from the fact that all subcircuits used have been shown to be uniform and they are connected in a uniform way. □

Theorem 6.4.14 (circuit simulation of st by $c_{[(\xi'_1,\eta'_1),(\xi'_2,\eta'_2)]:=a}$)
The $\mathcal{C}_2$-CSM operation $st\,(\xi'_1,\ \eta'_1,\ \xi'_2,\ \eta'_1)$ is simulated by a logspace uniform circuit $c_{[(\xi'_1,\eta'_1),(\xi'_2,\eta'_2)]:=a}$ of size $O(2^{12T(n)}T^6(n))$ and depth $O(T(n))$. This circuit takes as input the image words ξ'_1, η_1, ξ'_2 and η'_2. It outputs a word of length $|\mathcal{G}|$ that contains the rectangle (defined by $(\xi'_1, \eta'_1), (\xi'_2, \eta'_2)$) of image words to be stored and zeros at all other positions.

Proof. The address image words ξ'_1, η'_1, ξ'_2, η'_2 are decoded into binary numbers (as in Theorem 6.4.12) to be used in $c_{st\mathrm{Pixel}(i)}$. The circuit $c_{[(\xi'_1,\eta'_1),(\xi'_2,\eta'_2)]:=a}$ uses i subcircuits as follows. For each pixel word i in image word a: Subcircuit i ANDs the j^{th} symbol in pixel word i with each of the $2^{4T(n)}$ outputs of $c_{st\mathrm{Pixel}(i)}$. At this stage of the computation we have i grid words; the i^{th} grid word is 0 everywhere except for the 'rectangular' part of the grid that pixel i is to be stored to. These i grid words are ORed using an OR tree, giving the final output grid word as defined in the theorem statement.

[2]The grey rectangle consists of the lowest and leftmost 15 pixels of equal value.

This final output word is a called a mask and contains the 'stored rectangle' and all other pixel words contain only zeros.

By stepping through the construction we get size $O(2^{12T(n)}T^6(n))$ and depth $O(T(n))$. The high order in the size analysis is due the reuse of many subcircuits (in parallel) that compute over grid masks. The uniformity follows from the fact that all subcircuits used have been shown to be uniform and are connected in a uniform way. □

6.4.3 Control flow

$\mathcal{C}_2$-CSM control flow is straightforward to simulate. Recall from Section 6.3.2 that the binary word ctrl represents the $\mathcal{C}_2$-CSM control (or instruction pointer). Simulating *br* involves finding the new value for ctrl from the *br* parameters.

Theorem 6.4.15 (circuit simulation of *br* by $c_{br\,(\xi',\eta')}$) *The $\mathcal{C}_2$-CSM branch operation $br\,(\xi',\eta')$ is simulated by a logspace uniform circuit $c_{br\,(\xi',\eta')}$ of size $O(2^{2T(n)}T(n))$ and depth $O(T(n))$.*

Proof. The circuit $c_{br\,(\xi',\eta')}$ decodes its address image word parameters into the binary numbers ξ_1 and η_2 by using two (parallel) instances of the circuit $c_{\mathfrak{E}^{-1}(\cdot)}$ given in Lemma 6.4.9. These values are translated into an image word index i by evaluating the expression $i = \xi + \eta \cdot G_x$. The index i points to the image word encoding the next operation to be executed. □

Simulating sequential program control flow simply involves an update to ctrl.

Theorem 6.4.16 ($c_{\text{controlFlow}}$) *The circuit $c_{\text{controlFlow}}$ that simulates $\mathcal{C}_2$-CSM control flow has size $O(2^{4T(n)}T(n))$ and depth $T(n)$. The circuit $c_{\text{controlFlow}}$ has inputs $\mathcal{G}$ and a binary number* ctrl*, and outputs a new value for* ctrl.

Proof. Using $c_{\text{selectImage(ctrl)}}$ the circuit selects the image word $\mathcal{G}_{\text{ctrl}}$ with index ctrl in $\mathcal{G}$. It then updates ctrl by a value that is dependant on $\mathcal{G}_{\text{ctrl}}$. Specifically, if $\mathcal{G}_{\text{ctrl}}$ represents

- one of $\{h, v, *, \cdot, +\}$, then increment ctrl by 1;
- ρ, then increment ctrl by 3;
- one of $\{ld, st\}$, then increment ctrl by 5;
- *br*, then execute circuit $c_{br\,(\xi'_1,\eta'_2)}$ from Lemma 6.4.15, its output overwrites ctrl;
- *hlt*, then do not change the value of ctrl.

□

Theorem 6.4.17 (circuit simulation of $C_{(i)} \vdash_M C_{(i+1)}$ by c_{step}) *Let M be a $\mathcal{C}_2$-CSM. The uniform circuit c_{step} simulates $C_{(i)} \vdash_M C_{(i+1)}$ and is of size $O(2^{12T(n)}T^6(n))$ and depth $O(T(n))$.*

Proof. The representation of configurations as words was given in Section 6.3.2. The circuit c_{step} computes the mapping

$$(\text{ctrl}_{(i)}, \mathcal{G}_{(i)}) \to (\text{ctrl}_{(i+1)}, \mathcal{G}_{(i+1)}) .$$

c_{step} uses $c_{\text{controlFlow}}$, from Theorem 6.4.16, to compute

$$(\text{ctrl}_{(i)}, \mathcal{G}_{(i)}) \to \text{ctrl}_{(i+1)} .$$

Another subcircuit, $c_{\text{operation}}$, computes

$$(\text{ctrl}_{(i)}, \mathcal{G}_{(i)}) \to \mathcal{G}_{(i+1)}$$

as follows.

$c_{\text{operation}}$ selects the image word '*op*' pointed to by $\text{ctrl}_{(i)}$ and tests which element of $\{h, v, *, \cdot, +, \rho, ld, st, br, hlt\}$ *op* represents. This circuit simulates *op* in two steps:

In the first step the relevant circuit

$$c_{op} \in \{c_{a:=h(a)},\ c_{a:=v(a)},\ c_{a:=a^*},\ c_{a:=a\cdot b},\ c_{a:=a+b},\ c_{a:=\rho(a,z_1,z_u)},\ c_{a:=[(\xi_1',\eta_1'),(\xi_2',\eta_2')]},\ c_{[(\xi_1',\eta_1'),(\xi_2',\eta_2')]:=a},\ c_{br\,(\xi_1',\eta_2')},\ c_{hlt}\} ,$$

is executed with the appropriate parameters. All of these circuits were defined earlier in this chapter except for c_{hlt}, here we define *hlt* to compute the empty word.

In the second step the output of c_{op} and is written to the new grid word $\mathcal{G}_{(i+1)}$. This is done by $c_{\text{operation}}$ in one of three ways ((i)–(iii) below) that depends on *op*.

(i) If $op \in \{h, v, *, \cdot, +, \rho, ld\}$ then *op* alters image a only, so we want to overwrite image word a and leave the rest of $\mathcal{G}$ alone. The circuit $c_{\text{operation}}$ contains a pixel mask m_a that has one bit b for every pixel word in $\mathcal{G}$. If the b^{th} pixel word in $\mathcal{G}$ is in image word a, then the b^{th} bit in m_a is 1. Otherwise bit b is 0.

The b^{th} symbol in $\neg m_a$ is ANDed with each symbol in the b^{th} pixel word in $\mathcal{G}_{(i)}$. We call this process "masking". Simultaneously, the output from c_{op} is replicated to form a word of length $|\mathcal{G}|$ which is then masked with m_a. These two masked outputs are ORed to give $\mathcal{G}_{(i+1)}$.

(ii) If $op = st$ then the part of the grid to be updated is dependent on the *st* parameters. The circuit $c_{[(\xi_1',\eta_1'),(\xi_2',\eta_2')]:=a}$ (that simulates *st* and was given in Theorem 6.4.14) outputs a mask m_{st} of length $|\mathcal{G}|$ that contains the 'stored rectangle' and is 0 everywhere else. We now make another mask by executing $2^{2T(n)}$ parallel instances of $c_{st\text{Pixel}(i)}$, each instance gets a different

$i \in \{0, \ldots, 2^{2T(n)}\}$ as input. ($c_{st\text{Pixel}(i)}$ was given in Lemma 6.4.13). The outputs from these $2^{2T(n)}$ computations are ANDed to give a mask whose b^{th} bit is 1 if and only if pixel b is in the rectangle to be stored by st. We use the masks in a similar fashion to (i) above to get $\mathcal{G}_{(i+1)}$.

(iii) The last case is for $op \in \{br, hlt\}$. In this case we simply copy $\mathcal{G}_{(i)}$ to $\mathcal{G}_{(i+1)}$.

We have proved that c_{step} simulates the configuration update $C_{(i)} \vdash_M C_{(i+1)}$.

The size of $O(2^{12T(n)}T^6(n))$ and depth of $O(T(n))$ is derived by stepping through the construction. The size complexity of the st operation dominates all others.

Uniformity: The circuit c_{step} contains a large number of subcircuits. Each subcircuit is logspace computable and these are composed in a logspace uniform way. □

Next we give the resource use for our circuit simulation of a $\mathcal{C}_2$-CSM.

Theorem 6.4.18 (circuit simulation of M by c_M) *Let M be a $\mathcal{C}_2$-CSM that computes for* TIME *$T(n)$. The uniform circuit c_M simulates M and is of size $O(2^{12T(n)}T^7(n))$ and depth $O(T^2(n))$.*

Proof. c_M is the composition of $T(n)$ instances of c_{step} from the previous theorem. The circuit is given the initial configuration of M

$$(\text{ctrl}_{\text{sta}}, \mathcal{G}_{\text{sta}})$$

as input. After $O(T^2(n))$ parallel timesteps c_M outputs the word representation of the final configuration of M. □

The size bound in the previous theorem seems quite high, however one should keep in mind that M has SPACE of $S(n) \leqslant O(2^{3T(n)})$. (This was the original SPACE bound on M before we redefined SPACE to suit our simulations).

Suppose M is a language deciding $\mathcal{C}_2$-CSM. In this case c_M is augmented so that it ORs the contents of the output image word, thus computing a $\{0, 1\}$-valued function. The resulting circuit has only a constant factor overhead in the size and depth of c_M. From this we state the following.

Corollary 6.4.19

$$\mathcal{C}_2\text{-CSM-TIME}(T(n)) \subseteq \text{U-SIZE, DEPTH}(O(2^{12T(n)}T^7(n)), O(T^2(n)))\,.$$

From the inclusion given by Eq. (6.2.1):

Corollary 6.4.20 $\mathcal{C}_2\text{-CSM-TIME}(T(n)) \subseteq \text{DSPACE}(O(T^2(n)))\,.$

Combining the above result with Theorem 5.4.14 from the previous chapter gives a relationship between nondeterministic sequential space, $\mathcal{C}_2$-CSM TIME and deterministic space.

Corollary 6.4.21

$$\begin{aligned} \text{NSPACE}(S(n)) &\subseteq \mathcal{C}_2\text{-CSM-TIME}(O(S(n)+\log n)^4) \\ &\subseteq \text{DSPACE}(O(S(n)+\log n)^8)) \end{aligned}$$

Our simulations could possibly be improved to reduce the degree of the above polynomials, maybe even to a quadratic. Any improvement beyond that would be difficult, since it would imply an improvement to the quadratic bound in Savitch's theorem.

To summarise, the $\mathcal{C}_2$-CSM verifies the parallel computation thesis:

Corollary 6.4.22 $\text{NSPACE}(S^{O(1)}(n)) = \mathcal{C}_2\text{-CSM-TIME}(S^{O(1)}(n))$

This establishes a link between space bounded sequential computation and TIME bounded $\mathcal{C}_2$-CSM computation. For example:

Corollary 6.4.23 $\mathcal{C}_2\text{-CSM-TIME}(n^{O(1)}) = \mathsf{PSPACE}$

From the proof methods used in Chapter 5 and this chapter we can strengthen this result by restricting the $\mathcal{C}_2$-CSM.

Corollary 6.4.24 *The* $\mathcal{C}_2$-CSM *without the operations h and v verifies the parallel computation thesis.*

Proof. Operations h and v were not used in the $\mathcal{C}_2$-CSM simulation of index-vector machines. □

Let a 1D-$\mathcal{C}_2$-CSM be a $\mathcal{C}_2$-CSM where GRID and SPATIALRES are *both* constant in either the vertical or the horizontal direction.

Corollary 6.4.25 *The 1D-$\mathcal{C}_2$-*CSM *verifies the parallel computation thesis.*

Proof. The index-vector machine simulation used only constant GRID and SPATIALRES in the vertical direction. So we easily have the statement for the case where GRID and SPATIALRES are constant in the vertical direction. Moreover by rotating the grid layout and all images by $90°$ we obtain a simulation where GRID and SPATIALRES are constant in the horizontal direction. □

The parallel computation thesis relates parallel time to sequential space, however in our simulations we explicitly gave *all* resource bounds. As a final result we show that the class of $\mathcal{C}_2$-CSMs that simultaneously use polynomial SPACE and polylogarithmic TIME decide exactly the languages in NC. Let

$$\mathcal{C}_2\text{-CSM-SPACE, TIME}(S(n), T(n))$$

be the class of languages decided by $\mathcal{C}_2$-CSMs that use SPACE $S(n)$ and TIME $T(n)$, respectively. Analogously let

$$\mathcal{V}_I\text{-SPACE, TIME}(S(n), T(n))$$

be the class of languages decided by index-vector machines that use space $S(n)$ and time $T(n)$ respectively. Recall [KR90]

$$\mathcal{V}_I\text{-SPACE, TIME}(n^{O(1)}, \log^{O(1)} n) = \mathsf{NC}\,,$$

and for uniform circuits

$$\text{U-SIZE, DEPTH}(n^{O(1)}, \log^{O(1)} n) = \mathsf{NC}$$

From the resource overheads in our simulations we get

$$\begin{aligned} &\mathcal{V}_I\text{-SPACE, TIME}(O(2^{T(n)}), T(n)) \\ &\subseteq \mathcal{C}_2\text{-CSM-SPACE, TIME}(2^{O(T(n))}, T^{O(1)}(n)) \\ &\subseteq \text{U-SIZE, DEPTH}(2^{T^{O(1)}(n)}, T^{O(1)}(n))\,. \end{aligned}$$

For the case where $T(n) = \log^{O(1)} n$

Corollary 6.4.26

$$\begin{aligned} \mathsf{NC} \subseteq &\mathcal{V}_I\text{-SPACE, TIME}(n^{O(1)}, \log^{O(1)} n) \\ &\subseteq \mathcal{C}_2\text{-CSM-SPACE, TIME}(n^{O(1)}, \log^{O(1)} n) \\ &\subseteq \text{U-SIZE, DEPTH}(n^{O(1)}, \log^{O(1)} n) \\ &\subseteq \mathsf{NC}\,. \end{aligned}$$

6.5 Discussion

We have shown an upper bound on $\mathcal{C}_2$-CSM power in terms of uniform circuits. Combining this with the lower bound from the previous chapter; the $\mathcal{C}_2$-CSM verifies the parallel computation thesis. In our proofs we gave explicit bounds on all resources. This enabled us to show that $\mathcal{C}_2$-CSMs with polynomial SPACE and polylogarithmic TIME decide exactly the languages in NC.

As noted in the text, our simulations could probably be improved to get a tighter relationship between $\mathcal{C}_2$-CSM TIME and sequential space. For example, the size bounds on the circuit simulation of *ld* and *st* could be improved. If so, this would enable us define a tighter bound on simultaneous resource usage between the $\mathcal{C}_2$-CSM and (say) uniform circuits, in the hope of exactly characterising NC^k by varying a parameter k to the model. We leave this as future work.

Our simulations have enabled us to place a number of restrictions on the class of $\mathcal{C}_2$-CSMs. Fourier transformation gives no more power up to a polynomial. This is not so suprising since the DFT is efficiently computed on parallel machines. Also, two dimensions give no more power up to a polynomial, since we may keep GRID and SPATIALRES in one of the dimensions as constants. Of course, in a more fine grained analysis we might see the advantages of these aspects of the model.

On a different note, our results show that the kind of optics modelled by the $\mathcal{C}_2$-CSM is simulated in reasonable time on any of the parallel architectures that verify the parallel computation thesis.

7

Nonuniformity and the CSM

7.1 Introduction

Here we show that CSM with arbitrary real inputs decides the membership problem for any language $L \subseteq \Sigma^*$. We prove this result by showing the CSM with real inputs simulates the analog recurrent network (ARNN) model. Our simulation includes an efficient matrix multiplication algorithm, this efficiency is not dependent on the use of real inputs.

The chapter is an improved version of what appears in [WN05]. Firstly, the simulations in [WN05] required exponential spatialRes, here spatialRes is quadratic. Secondly the results in [WN05] were given in low-level CSM code, here we use the programming language introduced in Chapter 4. Thus our proofs are more concise and straightforward. Finally, in the reference mentioned we gave a simulation of the general ARNN model, followed by a simulation of a language deciding ARNN. For brevity we give only the latter simulation here.

7.2 ARNNs and nonuniform circuits

ARNNs [SS94, Sie99] are finite size feedback first-order neural networks with real weights. The state of each neuron at time $t+1$ is given by an update equation of the form

$$x_i(t+1) = \sigma\left(\sum_{j=1}^{N} a_{ij}x_j(t) + \sum_{j=1}^{M} b_{ij}u_j(t) + c_i\right) \quad , \quad i = 1, \ldots, N \qquad (7.2.1)$$

where N is the number of neurons, M is the number of inputs, $x_j(t) \in \mathbb{R}$ are the states of the neurons at time t, $u_j(t) \in \Sigma^+$ are the inputs at time t, and $a_{ij}, b_{ij}, c_i \in \mathbb{R}$ are the weights. An ARNN update equation is a function of discrete time $t = 1, 2, 3, \ldots$. The network's weights, states, and inputs are often written in matrix notation as A, B and c, $x(t)$, and $u(t)$, respectively. The function σ is defined as

$$\sigma(x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } 0 \leqslant x \leqslant 1 \\ 1, & \text{if } x > 1 \ . \end{cases}$$

A subset P of the N neurons, $P = \{x_{k_1}, \ldots, x_{k_p}\}$, $P \subseteq \{x_1, \ldots, x_N\}$, are called the p output neurons. The output from an ARNN computation is defined as the states $\{x_{k_1}(t), \ldots, x_{k_p}(t)\}$ of these p neurons over time $t = 1, 2, 3, \ldots$.

7.2.1 Formal net deciding language membership

ARNN input/output (I/O) mappings can be defined in many ways [SS94]. We present a CSM that simulates a specific type of ARNN called a formal net. Formal nets are ARNNs that decide language membership problems and have the following I/O encodings. A formal net has two binary input lines, called the input data line (D) and the input validation line (V), respectively. If D is *active* at a given time t then $D(t) \in \Sigma$, otherwise $D(t) = 0$. $V(t) = 1$ when D is active, and $V(t) = 0$ thereafter (when D is deactivated it never again becomes active). An input to a formal net at time t has the form $u(t) = (D(t), V(t)) \in \Sigma^2$. The input word $w = w_1 \ldots w_k \in \Sigma^+$ where $w_i \in \Sigma, 1 \leqslant i \leqslant k$, is represented by $u_w(t) = (D_w(t), V_w(t)), t \in \mathbb{N}$, where

$$D_w(t) = \begin{cases} w_t & \text{if } t = 1, \ldots, k \\ 0 & \text{otherwise} \end{cases}, \quad V_w(t) = \begin{cases} 1 & \text{if } t = 1, \ldots, k \\ 0 & \text{otherwise} \end{cases}.$$

A formal net has two output neurons $O_d, O_v \in \{x_1, \ldots, x_N\}$, called the output data line and output validation line, respectively. Given a formal net $\mathcal{F}$ with an input word w and initial state $x_i(1) = 0$, $1 \leqslant i \leqslant N$, w is *classified* in time τ if the output sequences have the form

$$O_d = 0^{\tau-1}\psi_w 0^{\boldsymbol{\omega}}, \ \ O_v = 0^{\tau-1}10^{\boldsymbol{\omega}}, \tag{7.2.2}$$

where $\psi_w \in \Sigma$ and $\boldsymbol{\omega} = |\mathbb{N}|$. If $\psi_w = 1$ then w is accepted, if $\psi_w = 0$ then w is rejected. We now give a definition of deciding language membership by ARNN (from [SS94]).

Definition 7.2.1 (Formal net deciding language membership) *The membership problem for the language $L \subseteq \Sigma^+$ is decided in time $\boldsymbol{T}$ by the formal net $\mathcal{F}$ provided that each word $w \in \Sigma^+$ is classified in time $\tau \leqslant \boldsymbol{T}(|w|)$ and $\psi_w = 1$ if $w \in L$ and $\psi_w = 0$ if $w \notin L$.*

Siegelmann and Sontag [SS94] prove that for each language $L \subseteq \Sigma^+$ there exists a formal net $\mathcal{F}_L$ to decide the membership problem for L. Recall (from Section 6.2) the definition of a nonuniform circuit family $\mathcal{C}_L$ that decides a language L, and the fact that nonuniform circuits of exponential size decide all $L \in \{0,1\}^*$. The formal net $\mathcal{F}_L$ contains a real weight that encodes $\mathcal{C}_L$. Let $\text{size}_{\mathcal{C}_L} : \mathbb{N} \to \mathbb{N}$ be the size of $\mathcal{C}_L$. For a given input word $w \in \Sigma^+$, $\mathcal{F}_L$ retrieves the encoding of circuit $c_{|w|}$ from its real weight and simulates this encoded circuit on input w to decide membership in L, in time

$\boldsymbol{T}(|w|) = O(|w|[\text{size}_{\mathcal{C}_L}(|w|)]^2)$. Essentially the real weight is being used as an oracle. Given polynomial time, formal nets decide the nonuniform language class P/poly. Given exponential time, formal nets decide the membership problem for all languages.

7.3 CSM simulation of formal nets

We first describe a representation of ARNN matrices as images. Then we give a simulation of the ARNN update equation for a single ARNN timestep t, followed by some remarks on the efficiency of our algorithm.

7.3.1 Image representation of ARNNs

As usual we let $\overline{\kappa}$ be the image representation of κ. The following representations are used in Theorem 7.3.2. Recall our representation of matrices by list-matrix images from Definition 4.2.6 and illustrated in Figure 4.2.1(e).

The ARNN weight matrices A, B, and c are represented by $N \times N$, $N \times M$, and $N \times 1$ list-matrix images $\overline{A}$, $\overline{B}$, and $\overline{c}$, respectively. The state vector x is represented by a $1 \times N$ list-matrix image $\overline{x}$. The dimensions of the above matrices, N and M, are represented by the natural number images $\overline{N}$ and $\overline{M}$. These images define the ARNN.

For an ARNN timestep t, the ARNN input vector $u(t)$ is represented by a $1 \times M$ list-matrix image $\overline{u}$.

We also make use of the constant images $f(x, y) = 0$ and $f(x, y) = 1$, denoted $\overline{0}$ and $\overline{1}$, respectively. Natural number images are used as addresses.

7.3.2 ARNN simulation overview

From the neuron state update equation Eq. (7.2.1), each $x_j(t)$ is a component of the state vector $x(t)$. From $x(t)$ we define the $N \times N$ matrix $X(t)$ where each row of $X(t)$ is the vector $x(t)$. Therefore $X(t)$ has components $x_{ij}(t)$, and for each $j \in \{1, \ldots N\}$ it is the case that $x_{ij} = x_{i'j}$, $\forall i, i' \in \{1, \ldots N\}$. From $u(t)$ we define the $N \times M$ matrix $U(t)$ where each row of $U(t)$ is the vector $u(t)$. Therefore $U(t)$ has components $u_{ij}(t)$, and for each $j \in \{1, \ldots M\}$ it is the case that $u_{ij} = u_{i'j}$, $\forall i, i' \in \{1, \ldots N\}$. Using $X(t)$ and $U(t)$ we rewrite Eq. (7.2.1) as

$$x_i(t+1) = \sigma\left(\sum_{j=1}^{N} a_{ij} x_{ij}(t) + \sum_{j=1}^{M} b_{ij} u_{ij}(t) + c_i\right) \quad , \quad i = 1, \ldots, N \ . \quad (7.3.1)$$

In the simulation we generate $N \times N$ and $N \times M$ list-matrix images $\overline{X}$ and $\overline{U}$ representing $X(t)$ and $U(t)$, respectively. We then simulate, in parallel, the affine combination in Eq. (7.3.1) using the CSM's $+$ and $\cdot$ operators.

We use the CSM's amplitude filtering operation ρ to simulate the ARNN σ function.

Lemma 7.3.1 *The CSM operation ρ simulates $\sigma(x)$ in constant* TIME.

Proof. From the definition of ρ in Eq. (2.2.6), we set $z_{\mathrm{l}}(x,y) = 0$ (denoted $\overline{0}$) and $z_{\mathrm{u}}(x,y) = 1$ (denoted $\overline{1}$) to give

$$\rho(f(x,y), \overline{0}, \overline{1}) = \begin{cases} 0, & \text{if } |f(x,y)| < 0 \\ |f(x,y)|, & \text{if } 0 \leqslant |f(x,y)| \leqslant 1 \\ 1, & \text{if } |f(x,y)| > 1 \ . \end{cases}$$

Using our representation of ARNN state values by images, $\rho(\overline{x}, \overline{0}, \overline{1})$ simulates $\sigma(x)$. Also, ρ is a CSM operation hence simulating $\sigma(x)$ requires constant TIME. □

Using the representations given above we state the following.

Theorem 7.3.2 *For a single timestep t the formal net update, given by Eq.* (7.2.1), *is simulated by a CSM M in $O(N)$* TIME, *$O(N^2)$* GRID, *$O(N)$* DYRANGE, *$O(N^2)$* SPATIALRES *and* FREQ, *∞* AMPLRES *and constant* PHASERES.

Proof. The CSM has multiplication, addition and thresholding operations, hence it is clear that the M can compute Eq. (7.2.1). We must show that the update is computed within the resource bounds given. The following CSM program computes the update. Essentially, the program transforms Eq. (7.2.1) to Eq. (7.3.1), and parallelises the relevant multiplications and additions. Finally the CSM operation ρ is used to simulate the ARNN thresholding function (see Lemma 7.3.1).

The program calls two functions. The function create_matrix($\overline{x}$, $\overline{N}$; $\overline{X}$) creates a list-matrix image $\overline{X}$ where each row is a copy of the vector image $\overline{x}$. To do this, N copies of image $\overline{x}$ are placed in N vertically juxtaposed images. Then all N images are rescaled to one image in a single step. The function requires $O(N)$ TIME and $O(N)$ GRID.

The function sum_columns($\overline{AX}$, $\overline{N}$; $\Sigma\overline{AX}$) rescales the list-matrix image $\overline{AX}$ over N horizontally juxtaposed images, then sums all of these images. The rescaling is done in one step, the summation in $O(N)$ TIME and $O(N)$ GRID.

```
update ($\overline{A}$, $\overline{B}$, $\overline{c}$, $\overline{x}$, $\overline{u}$, $\overline{N}$, $M$; $\overline{x}$)

  // constants: $\overline{0}$, $\overline{1}$.
  // variables: tmp, affine-comb.

     // Matrix vector multiplication
  create_matrix($\overline{x}$, $\overline{N}$; $\overline{X}$)
  $\cdot$ ( $\overline{A}$ , $\overline{X}$ ; $\overline{AX}$ )
  sum_columns($\overline{AX}$, $\overline{N}$; $\Sigma\overline{AX}$)

     // same procedure for $\Sigma\overline{BU}$
  create_matrix($\overline{u}$, $\overline{N}$; $\overline{U}$)
  $\cdot$ ( $\overline{B}$ , $\overline{U}$ ; $\overline{BU}$ )
  sum_columns($\overline{BU}$, $M$; $\Sigma\overline{BU}$)

     // Sum vectors and threshold
  +( $\Sigma\overline{AX}$ , $\Sigma\overline{BU}$ ; tmp )
  +( $\overline{c}$ , tmp ; affine-comb )
  $\rho$ ( affine-comb , $\overline{0}$ , $\overline{1}$ ; $\overline{x}$ )
**end** // update
```

In a formal net, $M = 2$ hence we ignore M in our asymptotic complexity analysis. The function update($\cdot$) requires $O(N)$ TIME. We need $O(N^2)$ GRID since create_matrix($\cdot$) and sum_columns($\cdot$) rescale in the vertical and horizontal directions respectively. We are using natural number images as addresses, hence we need $O(N)$ DYRANGE to rescale the list-matrix images in the horizontal and vertical directions. $O(N^2)$ SPATIALRES and FREQ is needed to represent the $N \times N$ list-matrix images. PHASERES has a constant value of 2 since we are using real values only. Finally, ∞ AMPLRES is required to represent real-valued ARNN weight matrices. □

By making a few extra assumptions the above algorithm could be improved to $O(\log N)$ TIME. Firstly we can change the representation. Our list-matrix image is basically a spatially separated set of delta functions (when analysing our algorithms we pixelate these delta functions). Now, lets assume we use a representation of matrices where each matrix value is represented by a constant valued pixel. Also, we'll make the assumption that all matrices have dimensions that are powers of two ($N = 2^k$ for some constant k). Given this new setup it becomes possible to compute the update in $O(\log N)$ TIME. The function sum_columns($\overline{AX}$, N; $\Sigma\overline{AX}$) can be computed in $O(\log N)$ TIME, by placing the left half of $\overline{AX}$ in image a, the other half in image b, add a and b, halve again, and repeat.

Even better, the function create_matrix($\overline{x}$, N; $\overline{X}$) can be computed in constant TIME. Using the new pixel-like representation, $\overline{x}$ and $\overline{X}$ are the same image (remember that $\overline{X}$ contains N vertically juxtaposed copies of

$\overline{x}$), so create_matrix($\cdot$) is the identity function on $\overline{x}$.

With the extra assumptions in place we are matching the well-known $O(\log N)$ TIME parallel (e.g. PRAM) matrix multiplication algorithms. (Although, from the results in previous chapters we can guess that this would be possible.) Since CSM SPACE, is the product of all non-TIME measures, we see no improvement over standard parallel algorithms that use $O(n^3)$ space.

Could further improvements be made? For example Reif [RT97] uses the DFT for constant TIME matrix multiplication over an ordered ring. Here, we are computing over the reals so we could consider using the FT. However, our simulation will require infinite FREQ (at least if its going to be similar to what we have described above). With finite FREQ, every Fourier transformation will degrade our representation, and thus we will not have the infinite precision we need to simulate an ARNN.

Corollary 7.3.3 *A CSM D with real inputs decides any $L \in \{0,1\}^*$ in $O(\boldsymbol{T}N)$* TIME, *$O(|w| + N^2)$* GRID, *$O(|w| + N)$* DYRANGE, *$O(|w| + N^2)$* SPATIALRES *and* FREQ, *∞* AMPLRES *and constant* PHASERES, *where $\boldsymbol{T}$ is the time of the formal net that decides L.*

Proof. D is a language deciding CSM that takes input as list & natural number images (see Definition 4.3.2). The input word w is represented as a list image f_w. D rescales f_w over $|w|$ horizontally juxtaposed images. Then D repeatedly calls update($\cdot$) (from Theorem 7.3.2) and feeds one bit of w at a time as input through the input image $\overline{u}$. After each call to update($\cdot$), D examines the appropriate position of $\overline{x}$ to see if the simulated output validation line from Eq. (7.2.2) has value 1. When this happens the value of simulated output data line (i.e. the appropriate value from image $\overline{x}$) is copied to D's output image and the computation halts.

The resource usage follows from Theorem 7.3.2. □

Siegelmann and Sontag [SS94] show that when deciding a language from the nonuniform class P/poly, in the worst case the time function $\boldsymbol{T}$ is polynomial in input word length. When deciding an arbitrary language, in the worst case $\boldsymbol{T}$ is exponential in input word length. These results carry over to the CSM, through the above simulation, when we permit real inputs.

Linear precision (in time $\mathbf{T}$) suffices for the ARNN model [SS94]. Using our simulation the associated cost would be exponential AMPLRES complexity in $\mathbf{T}$.

7.4 Discussion

The CSM we used in this chapter is essentially a $\mathcal{C}_2$-CSM that has real inputs. Natural number images are used as addresses. We improved the SPACE bound in [WN05] from exponential to polynomial, by using list and list-matrix images rather than stack and stack-matrix images. We gave a linear TIME matrix multiplication algorithm and discussed an improvement to log TIME by further changing our representation.

In software engineering terms we introduced a data refinement and optimisation. In principle such transformations could be built into a compiler for our language, and automatically applied to many of our other algorithms.

At the present time there is a debate over the applicability of results such as those in this chapter. For example two opposing views can be found in a single volume [Dav04, Sta04]. Many pages of text have been published on this subject and we have nothing essentially new to add. From the physical viewpoint (using current and any currently predicted future technology) we can not purposely set up an optical experiment that encodes an arbitrary language. On the other hand lets suppose that an optical set-up happens to be deciding an arbitrary language (or some omnipotent deity has set it up to do so). We can read the result, but in general we do not know the problem that has been solved.

By allowing arbitrary real inputs, we are essentially getting access to a Turing-uncomputable oracle for free. Allowing real inputs in this way does not help us understand the complexity of the problems we are solving.

On another note, it is interesting that Siegelmann's approach differs to ours in the following sense. Siegelmann uses the complexity measure 'number of bits' to assert that linear precision (in time) suffices [Sie99, SS94]. In our set-up we are representing real values by amplitude. Using our AMPLRES resource measure we incur an exponential cost in 'number of bits' of precision. We use AMPLRES as a complexity measure since it corresponds to a real world resource in optical information processing. By representing a number using the magnitude of some physical quantity we are essentially using a unary encoding. In this light our (more expensive) resource seems more sensible than counting the number of bits.

8

Conclusion

8.1 Discussion

We have explored the computational power of a model of computation inspired by optical information processing. We have established links between the model and computational complexity theory.

A large number of complexity measures were defined for the model, these seem natural with respect to resource usage in Fourier optical computing. We investigated the growth of these resources under each of the model's operations. Some resources grow in a way similar to well-known parallel models of computation. By allowing operations like Fourier transformation we are mixing the continuous and discrete worlds, hence some measures grow to infinity in one timestep. This motivates the definition of a variant of our model, the $\mathcal{C}_2$-CSM. The $\mathcal{C}_2$-CSM is in many ways close to physical optical computers. Also, as we have shown, it is more suited to analysis using standard methods from complexity theory.

We gave a convenient programming language and a number of data representations for the CSM. Our language simplifies programming while facilitating a straightforward complexity analysis of algorithms. We have argued that our data representations are natural and reasonable.

One of our main results is that the $\mathcal{C}_2$-CSM verifies the parallel computation thesis. This gives a characterisation of the $\mathcal{C}_2$-CSM in terms of sequential and parallel complexity theory. Thus the $\mathcal{C}_2$-CSM, and the optics it models, exhibit a type of parallelism frequently found in models of computation. We have shown that $\mathcal{C}_2$-CSMs operating in polylogarithmic TIME and polynomial SPACE accept exactly the class NC.

Our results were given by explicit simulations. Hence, within the constraints of our model, we have given a direct method for scientists to translate any reasonable and efficient parallel algorithm into an efficient optical one. Due to the unusual nature of the model our simulations have a unique flavour.

Finally we have shown the hideous power of allowing arbitrary real inputs to the CSM.

8.2 Further work

The framework we have developed suggests a number of avenues for further research. We outline some possible ideas. Some of these points were already discussed in previous chapters.

1. How far can we loosen the definition of the $\mathcal{C}_2$-CSM, so that it still verifies the parallel computation thesis? For example, some models have restrictions similar to our SPACE restriction [Par87, Gol82], while others do not [Sim77, TLR92]. Simon [Sim77] has shown that unrestricted shifts give no extra power to vector machines, up to a polynomial in time. This result was subsequently applied [TLR92] to prove a similar result for PRAMs with shift instructions. Can we apply this work to the $\mathcal{C}_2$-CSM or does our instruction set cause problems? Our conjecture is that the SPACE restriction can be removed from the $\mathcal{C}_2$-CSM definition.

2. On the other hand, what further restrictions can we put on the $\mathcal{C}_2$-CSM such that it still satisfies the parallel computation thesis? For example, we know that we can remove either of the FT operations h or v. Also we have shown that the addressing function $\mathfrak{E}$ can be restricted so that it maps to natural number images only. Can we further restrict the set of operations, or at least restrict how they interact?

3. What kind of resource trade-offs can we find for the $\mathcal{C}_2$-CSM? For example, can the AMPLRES condition be swapped for an analogous DYRANGE condition in the $\mathcal{C}_2$-CSM definition, without changing the power of the model? Even stronger, we conjecture that this can be done in such a way as to reduce GRID to a constant, with only a polynomial increase in TIME. Thus swapping constant AMPLRES for constant DYRANGE and GRID. Since we have defined a large number of complexity measures, it is probable that there are many other interesting trade-offs.

4. The size bound on the circuit simulation in Chapter 6 could probably be lowered. Moreover, it would be interesting to see how tight we can make the simulations in both of Chapters 5 and 6, for example by getting as close as possible to the quadratic bound implied by Savitch's theorem.

5. We have characterised the classes NC and AC in terms of the $\mathcal{C}_2$-CSM. However, can we give an exact characterisation of NC^k or AC^k by varying some parameter k to the model? Can such a characterisation be found for some of our resources and not for others, or are they interchangeable?

6. It would be interesting to explore the issue of analog computation with the CSM, for the case that the inputs are finite rationals or integers. What would be the most natural way to define analog complexity classes for the CSM? How would analog classes for the CSM correspond with results for other analog models and with discrete parallel models? By exploring this avenue of research can we put forward a parallel computation thesis for analog models?

Notation

CSM	continuous space machine
DFT	discrete Fourier transform
FT	Fourier transform
$\mathcal{I}$	set of all images $f : [0,1) \times [0,1) \to \mathbb{C}$
$\overline{\kappa}$	representation of κ (e.g. representation of κ as an image)
$q \overset{rep}{\Longrightarrow} p$	q is represented as p
f_n	n-valued constant image; $f_n(x,y) = i,\ \ n \in \mathbb{C},\ x,y \in [0,1)$
f_w	stack or list image representing $w \in \{0,1\}^*;\ \ x,y \in [0,1)$
$\mathfrak{E} : \mathbb{N} \to \mathcal{N}$	Address encoding function
$\mathbb{N}$	non-negative integers
$\mathbb{N}^+$	integers strictly greater than 0
$\mathbb{Q}$	integers
$\mathbb{R}$	reals
$\mathbb{C}$	complex numbers
ω	cardinality of $\mathbb{N}$
$\{0,1\}^*$	set of all words of length $\geqslant 0$ over alphabet $\{0,1\}$
$\{0,1\}^+$	set of all words of length > 0 over alphabet $\{0,1\}$
$\lvert w \rvert$	length of the word $w \in \{0,1\}^*$
$\lvert z \rvert$	amplitude of $z \in \mathbb{C}$
i	$\sqrt{-1}$
$f^k(n)$	polynomial of f, e.g. $f^2(n) = (f(n))^2$
$f^{(k)}(n)$	function composition, e.g. $f^{(2)}(n) = f(f(n))$
h	CSM horizontal Fourier transform operation
v	CSM vertical Fourier transform operation
$+$	CSM point by point image addition operation
$\cdot$	CSM point by point image multiplication operation
ρ	CSM image threshold operation
st	CSM store image(s) from address a operation
ld	CSM load image(s) to address a operation
br	CSM branch operation
hlt	CSM halt operation
$T(n)$	CSM TIME complexity
$G(n)$	CSM GRID complexity
$R_{\mathrm{A}}(n)$	CSM AMPLRES complexity
$R_{\mathrm{S}}(n)$	CSM SPATIALRES complexity
$R_{\mathrm{P}}(n)$	CSM PHASERES complexity
$R_{\mathrm{D}}(n)$	CSM DYRANGE complexity
$\nu(n)$	CSM FREQ complexity
$S(n)$	CSM SPACE complexity

Bibliography

[Adl94] Lenonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, November 1994.

[Akl97] Selim G. Akl. *Parallel computation: models and methods.* Prentice Hall, 1997.

[AMVP03] Artiom Alhazov, Carlos Martn-Vide, and Linqiang Pan. Solving a PSPACE-complete problem by recognizing P systems with restricted active membranes. *Fundamenta Informaticae*, 58(2):67–77, November 2003.

[AS92] Henri H. Arsenault and Yunlong Sheng. *An introduction to optics in computers*, volume TT 8 of *Tutorial texts in optical engineering.* SPIE, 1992.

[BCSS97] Lenore Blum, Felipe Cucker, Michael Shub, and Steve Smale. *Complexity and real computation.* Springer, New York, 1997.

[BDG88a] José Luis Balcázar, Josep Díaz, and Joaquim Gabarró. *Structural complexity I*, volume 11 of *EATCS Monographs on Theoretical Computer Science.* Springer, Berlin, 1988.

[BDG88b] José Luis Balcázar, Josep Díaz, and Joaquim Gabarró. *Structural complexity II*, volume 22 of *EATCS Monographs on Theoretical Computer Science.* Springer, Berlin, 1988.

[BH04a] Olivier Bournez and Emmanuel Hainry. An analog characterisation of elementarily computable functions over the real numbers. In *International Colloquium on Automata Languages and Programming*, volume 3142 of *Lecture Notes in Computer Science*, pages 269–280. Springer, 2004.

[BH04b] Olivier Bournez and Emmanuel Hainry. Real recursive functions and real extensions of recursive functions. In Maurice Margenstern, editor, *Machines, Computations and Universality (MCU 2004)*, volume 3354 of *Lecture Notes in Computer*

Science, pages 116–127, Saint-Petersburg, Russia, September 2004. Springer.

[Bor77] Allan Borodin. On relating time and space to size and depth. *SIAM Journal on Computing*, 6(4):733–744, December 1977.

[Bra78] Ronald N. Bracewell. *The Fourier transform and its applications.* Electrical and electronic engineering series. McGraw-Hill, second edition, 1978.

[BS90] Ravi B. Boppana and Michael Sipser. *The complexity of finite functions.* Volume A of van Leeuwen [vL90], 1990.

[Cam01] Manuel Lameiras Campagnolo. *Computational Complexity of Real Valued Recursive Functions and Analog Circuits.* PhD thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, 2001.

[Cam02] Manuel Lameiras Campagnolo. The complexity of real recursive functions. In C. S. Calude, M. J. Dinneen, and F. Peper, editors, *Unconventional Models of Computation (UMC'02)*, volume 2509 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2002.

[Cam04] Manuel Lameiras Campagnolo. Continuous-time computation with restricted integration capabilities. *Theoretical Computer Science*, 317(1–3):147–165, 2004.

[Cau90] H. John Caulfield. Space-time complexity in optical computing. In B. Javidi, editor, *Optical information-processing systems and architectures II*, volume 1347, pages 566–572. SPIE, July 1990.

[CKS81] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, January 1981.

[CMC02] Manuel Lameiras Campagnolo, Cristopher Moore, and J.F. Costa. An analog characterization of the Grzegorczyk hierarchy. *Journal of Complexity*, 18(4):977–1000, 2002.

[CS76] Ashok K. Chandra and Larry J. Stockmeyer. Alternation. In *17th annual symposium on Foundations of Computer Science*, pages 98–108, Houston, Texas, October 1976. IEEE. Preliminary Version.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[Dav04] Martin Davis. The myth of hypercomputation. In Teuscher [Teu04].

[DC89] Patrick W. Dymond and Stephen A. Cook. Complexity theory of parallel time and hardware. *Information and Computation*, 80(3):205–226, March 1989.

[Deu85] David Deutsch. The Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London A*, 400:97–117, 1985.

[dSG02] Daniel da Silva Graça. The general purpose analog computer and recursive functions over the reals. Master's thesis, IST, Universidade Técnica de Lisboa, 2002.

[dSG04] Daniel da Silva Graça. Some recent developments on Shannon's general purpose analog computer. *Mathematical Logic Quarterly*, 50(4-5):473–485, 2004.

[Fei88] Dror G. Feitelson. *Optical Computing: A survey for computer scientists.* MIT Press, 1988.

[Fey82] Richard E. Feynman. Simulating Physics with Computers. *International Journal of Theoretical Physics*, 21(6/7):467–488, 1982.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, 1972.

[FSS84] Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, circuits and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proc. 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[GGKK03] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to parallel computing.* Addison Wesley, 2003.

[GHR95] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to parallel computation: P-completeness theory.* Oxford university Press, Oxford, 1995.

[GNPJRC05] Miguel A. Gutiérrez-Naranjo, Mario J. Pérez-Jiménez, and Francisco J. Romero-Campero. A linear solution for QSAT with membrane creation. In *Pre-Proceedings of the Sixth International Workshop on Membrane Computing (WMC6)*, pages 395–409. Vienna University of Technology, July 2005.

[Gol77] Leslie M. Goldschlager. *Synchronous parallel computation.* PhD thesis, University of Toronto, Computer Science Department, December 1977.

[Gol78] Leslie M. Goldschlager. A unified approach to models of synchronous parallel machines. In *Proc. 10th Annual ACM Symposium on Theory of Computing*, pages 89–94, 1978.

[Gol82] Leslie M. Goldschlager. A universal interconnection pattern for parallel computers. *Journal of the ACM*, 29(4):1073–1086, October 1982.

[Goo96] Joseph W. Goodman. *Introduction to Fourier optics.* McGraw-Hill, New York, second edition, 1996.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proc. 28th Annual ACM Symposium on Theory of Computing*, pages 212–219, May 1996.

[GS70] Joseph W. Goodman and A. M. Silvestri. Some effects of Fourier domain phase quantization. *IBM Journal of research and development*, 14:478–484, September 1970.

[Hea87] Tom Head. Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, 49(6):737–759, 1987.

[HS74] Juris Hartmanis and Janos Simon. On the power of multiplication in random access machines. In *Proceedings of the 15th annual symposium on switching and automata theory*, pages 13–23, The University of New Orleans, October 1974. IEEE.

[Joh90] David S. Johnson. *A catalog of complexity classes.* Volume A of van Leeuwen [vL90], 1990.

[KM99] Pascal Koiran and Cristopher Moore. Closed-form analytic maps in one and two dimensions can simulate Turing machines. *Theoretical Computer Science*, 210:217–223, 1999.

[Knu97] Donald E. Knuth. *The art of computer programming, Volume 1: Fundamental algorithms.* Addison Wesley, 1997.

[KR90] Richard. M. Karp and Vijaya Ramachandran. *Parallel algorithms for shared memory machines.* Volume A of van Leeuwen [vL90], 1990.

[Kra70] V.M. Krapchenko. Asymptotic estimation of addition time of a parallel adder. *Syst. Theory Res.*, 19:105–222, 1970.

[Lee95] John N. Lee, editor. *Design issues in optical processing.* Cambridge studies in modern optics. Cambridge University Press, 1995.

[LF80] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[Lip95] Richard J. Lipton. Using DNA to solve NP-complete problems. *Science*, 268:542–545, April 1995.

[LP92] Ahmed Louri and Arthur Post. Complexity analysis of optical-computing paradigms. *Applied optics*, 31(26):5568–5583, September 1992.

[MC04] Jerzy Mycka and José Félix Costa. Real recursive functions and their hierarchy. *Journal of Complexity*, 2004. In press.

[McA91] Alastair D. McAulay. *Optical computer architectures.* Wiley, 1991.

[MM00] Cristopher Moore and Jonathan Machta. Internal diffusion-limited aggregation: Parallel algorithms and complexity. *Journal of Statistical Physics*, 99:629–660, 2000.

[MN97] Cristopher Moore and Mats G. Nordahl. Predicting lattice gases is p-complete. Santa Fe Institute Working Paper 97-04-034, 1997.

[MN99] Cristopher Moore and Martin Nilsson. The computational complexity of sandpiles. *Journal of Statistical Physics*, 96:205–224, 1999.

[Moo90] Cristopher Moore. Undecidability and unpredictability in dynamical systems. *Physical Review Letters*, 64(20):2354–2357, May 1990.

[Moo91] Cristopher Moore. Generalized shifts: undecidability and unpredictability in dynamical systems. *Nonlinearity*, 4:199–230, 1991.

[Moo96] Cristopher Moore. Recursion theory on the reals and continuous-time computation. *Theoretical Computer Science*, 162(1):23–44, August 1996.

[Moo97] Cristopher Moore. Majority-vote cellular automata, Ising dynamics, and P-completeness. *Journal of Statistical Physics*, 88:795–805, 1997.

[Moo98] Cristopher Moore. Dynamical recognizers: Real-time language recognition by analog computers. *Theoretical Computer Science*, 201:99–136, May 1998.

[MS98] Russ Miller and Quentin F. Stout. Algorithmic techniques for networks of processors. In Mikhail J. Atallah, editor, *Algorithms and theory of computation handbook.* CRC, 1998.

[Nau00a] Thomas J. Naughton. Continuous-space model of computation is Turing universal. In Sunny Bains and Leo J. Irakliotis, editors, *Critical Technologies for the Future of Computing*, Proceedings of SPIE vol. 4109, pages 121–128, San Diego, California, August 2000.

[Nau00b] Thomas J. Naughton. A model of computation for Fourier optical processors. In Roger A. Lessard and Tigran Galstian, editors, *Optics in Computing 2000*, Proc. SPIE vol. 4089, pages 24–34, Quebec, Canada, June 2000.

[NC00] Michael A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information.* Cambridge University Press, 2000.

[Niv56] Ivan Niven. *Irrational numbers*, volume 11 of *The Carus Mathematical Monographs.* The Mathematical Association of America, Wiley, 1956.

[NJK$^+$99] Thomas Naughton, Zohreh Javadpour, John Keating, Miloš Klíma, and Jiří Rott. General-purpose acousto-optic connectionist processor. *Optical Engineering*, 38(7):1170–1177, July 1999.

[NW01] Thomas J. Naughton and Damien Woods. On the computational power of a continuous-space optical model of computation. In Maurice Margenstern and Yurii Rogozhin, editors, *Machines, Computations and Universality: Third International Conference*, volume 2055 of *Lecture Notes in Computer Science*, pages 288–299, Chişinău, Moldova, May 2001. Springer.

[Par87] Ian Parberry. *Parallel complexity theory.* Wiley, 1987.

[Pău00] Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, August 2000.

[Pău02] Gheorge Păun. *Membrane computing: an introduction.* Springer, 2002.

[PRS74] Vaughan R. Pratt, Michael O. Rabin, and Larry J. Stockmeyer. A characterisation of the power of vector machines. In *Proc. 6th annual ACM symposium on theory of computing*, pages 122–134. ACM press, 1974.

[PS76] Vaughan R. Pratt and Larry J. Stockmeyer. A characterisation of the power of vector machines. *Journal of Computer and Systems Sciences*, 12:198–221, 1976.

[Qui94] Michael J. Quinn. *Parallel computing: theory and practice.* McGraw-Hill series in Computer Science. Networks, parallel and distributed computing. McGraw-Hill, Maidenhead, Berkshire, UK, second edition, 1994.

[Rei93] John H. Reif, editor. *Synthesis of parallel algorithms.* Morgan Kaufmann, 1993.

[Roj96] Raúl Rojas. Conditional branching is not necessary for universal computation in von Neumann computers. *Journal of Universal Computer Science*, 2(11):756–768, 1996.

[RT97] John H. Reif and Akhilesh Tyagi. Efficient parallel algorithms for optical computing with the discrete Fourier transform (DFT) primitive. *Applied optics*, 36(29):7327–7340, October 1997.

[Sav76] John E. Savage. *The complexity of Computing.* Wiley, 1976.

[Sav98] John E. Savage. *Models of computation: Exploring the power of computing.* Addison Wesley, 1998.

[Sho94] Peter Shor. Algorithms for quantum computation: Discrete logarithms and factoring. *In Proceedings 35th Annual Symposium on Foundations Computer Science*, pages 124–134, 1994.

[Sie99] Hava T. Siegelmann. *Neural networks and analog computation: beyond the Turing limit.* Progress in Theoretical Computer Science. Birkhäuser, Boston, 1999.

[Sim77] Janos Simon. On feasible numbers. In *Proc. 9th Annual ACM Symposium on Theory of Computing*, pages 195–207. ACM, 1977.

[Sos03] Petr Sosík. The computational power of cell division in P systems: Beating down parallel computers? *Natural Computing*, 2(3):287–298, 2003.

[SS71] Arnold Schönhage and Volker Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7(3-4):281–292, 1971.

[SS79] Walter J. Savitch and Michael J. Stimson. Time-bounded random access machines with parallel processing. *Journal of the ACM*, 26:103–118, 1979.

[SS91] Hava T. Siegelmann and Eduardo D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80, 1991.

[SS94] Hava T. Siegelmann and Eduardo D. Sontag. Analog computation via neural networks. *Theoretical Computer Science*, 131(2):331–360, September 1994.

[Sta04] Mike Stannett. Hypercomputational models. In Teuscher [Teu04].

[Teu04] Christof Teuscher, editor. *Alan Turing: life and legacy of a great thinker*. Springer, Berlin, 2004.

[TLR92] Jerry L. Trahan, Michael C. Loui, and Vijaya Ramachandran. Multiplication, division and shift instructions in parallel random access machines. *Theoretical Computer Science*, 100:1–44, 1992.

[TvEB93] John Tromp and Peter van Emde Boas. Associative storage modification machines. In Klaus Ambos-Spies, Steven Homer, and Uwe Schöning, editors, *Complexity theory: current research*, pages 291–313. Cambridge University Press, 1993.

[Van92] Anthony VanderLugt. *Optical Signal Processing*. Wiley Series in Pure and Applied Optics. Wiley, New York, 1992.

[vEB90] Peter van Emde Boas. *Machine models and simulations*. Volume A of van Leeuwen [vL90], 1990.

[vL90] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume A. Elsevier, Amsterdam, 1990.

[vLW87] Jan van Leeuwen and Jiří Wiedermann. Array processing machines. *BIT*, 27:25–43, 1987.

[Vol99] Heribert Vollmer. *Introduction to circuit complexity: A uniform approach*. EATCS Texts in Theoretical Computer Science. Springer, 1999.

[WC98] Colin P. Williams and Scott H. Clearwater. *Explorations in quantum computing*. Springer, 1998.

[Wea89] H. Joseph Weaver. *Theory of Discrete and Continuous Fourier Analysis.* Wiley, 1989.

[Weg87] Igno Wegener. *The Complexity of Boolean Functions.* Wiley-Teubnre, 1987.

[Wei00] Klaus Weihrauch. *Computable Analysis: An Introduction.* Texts in Theoretical Computer Science. Springer, Berlin, 2000.

[WG05a] Damien Woods and J. Paul Gibson. Complexity of continuous space machine operations. In S. B. Cooper, B. Löewe, and L. Torenvliet, editors, *New Computational Paradigms, First Conference on Computability in Europe, CiE 2005*, volume 3526 of *Lecture Notes in Computer Science*, pages 540–551, Amsterdam, June 2005. Springer.

[WG05b] Damien Woods and J. Paul Gibson. Lower bounds on the computational power of an optical model of computation. In Cristian S. Calude, Michael J. Dinneen, Gheorge Păun, Mario J. Pérez-Jiménez, and Grzegorz Rozenberg, editors, *Fourth International Conference on Unconventional Computation (UC'05)*, volume 3699 of *Lecture Notes in Computer Science*, pages 237–250, Sevilla, October 2005. Springer.

[Wie84] Jiří Wiedermann. Parallel Turing machines. Technical Report RUU-CS-84-11, Department of Computer Science, University of Utrecht, 1984.

[WN05] Damien Woods and Thomas J. Naughton. An optical model of computation. *Theoretical Computer Science*, 334(1–3):227–258, April 2005.

[Woo] Damien Woods. Upper bounds on the computational power of an optical model of computation. In *16*$^{\text{th}}$ *International Symposium on Algorithms and Computation (ISAAC 2005)*, Lecture Notes in Computer Science, Sanya, China. Springer. To appear.

[WW86] Klaus Wagner and Gerd Wechsung. *Computational Complexity.* Reidel, 1986.

[YJY01] Francis T. S. Yu, Suganda Jutamulia, and Shizhuo Yin, editors. *Introduction to information optics.* Academic Press, 2001.

[Yok02] Takashi Yokomori. Molecular computing paradigm – toward freedom from Turing's charm. *Natural computing*, 1(4):333–390, 2002.