ACCELERATING CONSTRAINT AUTOMATA COMPOSITION WITH GPGPU PARALLELIZATION

January 2012

Gunnar K. Vilbergsson
Master of Science in Computer Science

ACCELERATING CONSTRAINT AUTOMATA COMPOSITION WITH GPGPU PARALLELIZATION

Gunnar K. Vilbergsson
Master of Science
Computer Science
January 2012
School of Computer Science
Reykjavík University

M.Sc. PROJECT REPORT

ISSN 1670-8539

Accelerating Constraint Automata Composition with GPGPU Parallelization

by

Gunnar K. Vilbergsson

Project report submitted to the School of Computer Science
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science in **Computer Science**

January 2012

Project Report Committee:

Marjan Sirjani, Supervisor
Supervisor, Reykjavik University

Emanuela Merelli
Co-Supervisor, University of Camerino

Luca Aceto
Professor, Reykjavik University

Luca Tesei
Assistant Professor, University of Camerino

The undersigned hereby certify that they recommend to the School of Computer Science at Reykjavík University for acceptance this project report entitled **Accelerating Constraint Automata Composition with GPGPU Parallelization** submitted by **Gunnar K. Vilbergsson** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

_____

Date

_____

Marjan Sirjani, Supervisor
Supervisor, Reykjavik University

_____

Emanuela Merelli
Co-Supervisor, University of Camerino

_____

Luca Aceto
Professor, Reykjavik University

_____

Luca Tesei
Assistant Professor, University of Camerino

_____

Date

_____

Gunnar K. Vilbergsson
Master of Science

Accelerating Constraint Automata Composition with GPGPU Parallelization

Gunnar K. Vilbergsson

January 2012

Abstract

One of the principle challenges of Constraint Automata composition is the rapid growth of the state space and the difficulty inherent in processing very large state spaces both in terms of space as well as computation time. We show that the method outlined here goes some way in tackling both these issues by making it possible to process the composition in parallel using GPGPU programming. We also show how, using the methods put forth to make the GPGPU parallelization possible, it is possible to distribute the composition of Constraint Automata across many nodes.

Hröðun Samsetningar Þvingunarstöðuvéla með GPGPU Samhliðun

Gunnar K. Vilbergsson

Janúar 2012

Útdráttur

Ein helsta áskorunin þegar kemur að samsetningu þvingunarstöðuvéla er hversu hratt stöðunum fjölgar og vandamál sem upp koma við meðhöndlun mikils stöðufjölda bæði þegar kemur að gagnamagni og vinnslutíma. Við sýnum að aðferðin sem er útlistuð hér nýtist til að samþætta þvingunarstöðuvélar samhliða með GPGPU forritun. Við sýnum einnig hvernig hægt er, með aðferðunum sem gera GPGPU samhliðunina mögulega, að dreifa samþáttun þvingunarstöðuvéla á margar vélar.

This work is dedicated to my parents who through their love and support have inspired me to push myself further and to my girlfriend Eva without whose love, support and confidence this project would never have been finished.

Thanks go first to my teacher Marjan Sirjani for her invaluable help with this project. Without her none of this would have been possible. I would like to thank Emanuela Merelli and Luca Tesei for their kind hospitality and invaluable help. I would also like to thank Grímur Tómas Tómasson whose patient help with an unfamiliar programming language was a large contributing factor to the success of this project. I would like to thank my very good friend Martha Dís Brandt for her help with making this report a little more readable. Also deserving of thanks are my contemporary students, especially Björn Jónsson who by lending an ear to my ramblings were of invaluable help when nothing seemed to be working.
Finally my friends and family deserve great thanks for all their love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As CPU designers hit the physical limits of what processors can do with respect to clock speed and computational ability the importance of parallelization grows exponentially. It is a fact that the advancement of computing power comes no longer from architectural changes or the increase of clock frequencies. Moore's Law states that the number of transistors that can be placed inexpensively on an integrated circuit doubles every two years. As chipmakers are faced with the limits of current microchip manufacturing techinques, Moore's Law is now being kept alive by adding more cores to processors. This means that any increases in computation speed must come from the parallelization of computing tasks since single threaded tasks can, by definition, not run on more than one core. Currently the strongest gains are being had with GPGPU (General Purpose computation on Graphics Processing Units). The most widespread API for writing GPGPU enabled code, currently, is the CUDA API from NVIDIA.

Several computational tasks are said to be "embarrassingly parallel" in that they can be parallelized with little or no effort. There are, however, many more problems that are not so easily parallelized. Some of these tasks can however be made significantly parallelizeable by some simple changes to either algorithms or data structure. The complexity of these changes can span a wide gamut ranging from minor tweaks to complete rewrites.

The Reo coordination language (Arbab, 2004) has Constraint Automata as its formal semantics. This means that for a Reo network to be reasoned about computationally it must be converted into a Constraint Automaton that represents said Reo network. Since Reo networks are made up of simple components which all have their own Constraint Automata representation any Constraint Automata representation of a Reo network must be created by compositing the component Constraint Automata of the networks. This

often leads to very large Constraint Automata which are very costly both in terms of memory and computation since the state space growth is geometric.

The problem of compositing Constraint Automata (Arbab, Baier, Rutten, & Sirjani, 2004) is one of those problems which may appear to be hard to parallelize. However with a change of data representation and some simple changes to the algorithm, the problem of composing Constraint Automatons becomes highly parallel. Once the problem of composition has been made parallel, computational models such as the massively parallel execution on GPGPU hardware can be introduced to exploit the parallelism for great gains in efficiency. More classically parallel execution models such as grid computation and distributed computation also become feasible. This paper will however focus on the GPGPU side, specifically the CUDA API from NVIDIA.

Current approaches (Pourvatan & Rouhy, 2007) to generating the product of Constraint Automata are inherently single threaded where the Constraint Automata are presented as lists of states, names, transitions and data constraints. In fact, each iteration of the algorithm relies on the work done by the previous iteration. By changing the representations of the Constraint Automata to adjacency matrices we eliminate the need for each iteration to rely on any previous iteration. This is achieved by having each cell in the adjacency matrix contain all pertinent information about each transition. With this we can then use what is essentially matrix product, albeit with a fair bit of extra logic, to produce the product of two Constraint Automata. The challenge is creating an algorithm that will consistently follow the rules of Constraint Automata composition while also scaling up to very large automata.

To address these challenges the following has been done:

- Identify the challenges to parallelization inherent in current methods
- Identify necessary changes to enable parallelization
- Implement a new data structure to represent Constraint Automata
- Create an algorithm for composing Constraint Automata

The creation of a new data representation for Constraint Automata as well as a new algorithm for Constraint Automata composition using this data representation means that composition can be achieved at greater speed using parallelization. Beyond that, composition can be distributed across an arbitrary number of nodes without any cross node communication. This means that there will be no need for computation to halt while waiting for synchronization. This can clearly increase throughput considerably.

Chapter 2

Background

This work is motivated by the need for an efficient way to use Reo to model real systems. Since Reo has Constraint Automata as its formal semantics it is very important that Constraint Automata can be composited in an efficient manner. This is because all Reo circuits can be represented as Constraint Automata to enable us to reason about their behavior computationally. That means that it is very important to be able to efficiently composite Constraint Automata into larger automatons that represent Reo circuits. This is where CUDA comes in.

2.1 Reo

From (Arbab, 2004):

> Reo is a coordination model and as such has very little to say about the computational entities whose activities it coordinates. These entities can be fragments or modules of sequential code, passive or active objects, threads, processes, agents, or software components. Without loss of generality, we refer to these entities as component instances in Reo. From the point of view of Reo, a system consists of a number of component instances executing at one or more locations, communicating through connectors that coordinate their activities. This is shown in Figure 2.1, where component instances are represented as boxes, channels as straight lines, and connectors are delineated by dashed lines. Each connector in Reo is, in turn, constructed compositionally out of simpler connectors, which are ultimately composed out of channels.

This is why each dashed closed curve representing a connector in Figure 2.1 contains only a set of channels connected together in a specific topology.

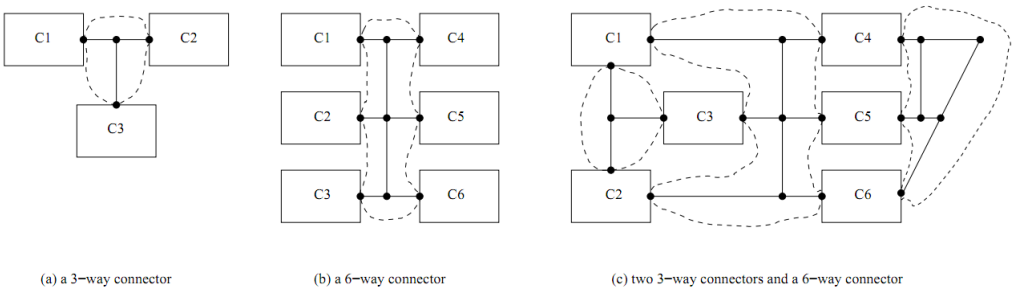

Figure 2.1: Components and Connectors

A *component instance*, p, is a non-empty set of active entities (e.g., processes, agents, threads, actors, etc.) whose only means of communication with the entities outside of this set is through input/output operations that they perform on a (dynamic) set of channel ends that are $connected$ to p. The communication among the active entities inside a component instance, and the mechanisms used for this communication, are of no interest. Likewise, Reo is oblivious to the synchronization, mutual exclusion, and coordination that may have to take place among the active entities inside a component instance for their proper utilization of the channel ends that are connected to that component instance. All these details are internal to a component instance and, thus, irrelevant. What is relevant is only the inter-component-instance communication which takes place exclusively through channels that comprise Reo connectors. Indeed, the constituents inside a component instance may themselves be other component instances that are connected by Reo connectors.

Components are software implementations, instances of which can be executed on either physical or logical devices. Thus components are abstract types that describe the properties of their instances.

Physical or logical devices where active entities execute are called locations. Such entities may include virtual machines, processes or even actual computers. Entities also do not need to be confined to single computers so an entity can be a grid of computers. There is no locality constraint on entities so an entity can be distributed over a large geographic area, thus a highly distributed grid could still be an entity. The configuration of device is considered an internal detail of the component instance and therefore Reo is oblivious to it. An instance of a distributed component will however always have a unique location associated with it. There can be zero or more component instances executing at

a given location and instances can move between locations while they execute. Reo is concerned with locations only as far as inter-component communication optimization is concerned. A channel is the only primitive communication medium between two component instances. Channels have their own unique identities and are dynamically created and automatically garbage collected.

Channels in themselves have no direction. Each channel in Reo however has exactly two directed ends, each with its own identity, through which components refer to and manipulate the data they carry. Channel ends are either sources or sinks. Source ends accept data into channels and sink ends dispense data out of channels. Any of a component instance's active entities can use any channel ends that are known to that component instance in Reo operations.

Channels in Reo are exclusively for data transfer using input/output operations of their ends. Only component instances, or entities inside component instances, connected to channel ends can perform operations on said channel ends. While several component instances may know the identity of a channel end only one component instance can be connected to any channel end at any given time. The connection of a channel end to a component instance is a logical notion and therefore independent of the locations of either the channel end or the component instance. Whereas a component instance sharing a location with a channel end may be able to more efficiently manipulate said channel end, co-location is not a prerequisite for any such manipulation.

Components and channels in Reo can both be considered to be mobile. Component instances may move from one location to another during their lifetime but when this happens the channel connection topology remains intact. Channel ends may also be moved by active entities inside component instances for whatever reason without changing the connection topology. Irrespective of this channel ends may be disconnected from component instances and perhaps connected to another component instance, altering the connection topology at any time.

2.2 Constraint Automata

To ensure that Reo components can be reasoned about by machines a concrete operational model is needed. For Reo this model is Constraint Automata. The CA states stand for the possible configurations of Reo channels, such as the contents of a FIFO buffer. The transitions of the Constraint Automata however describe the possible data flow through the channel and its effect on the configuration of the Reo channel and thus the state of the

Constraint Automata. The operational semantics of Reo, as described in (Arbab, 2004) can be reformulated in terms of constraint automata.

In (Arbab et al., 2004) Constraint Automata are defined as follows:

> A constraint automaton (over the data domain Data) is a tuple A$A = (Q, N, \rightarrow, Q_0)$ where
> - Q is a finite set of states
> - N is a finite set of names
> - $\rightarrow$ is a finite subset of $Q \times 2^N \times DC \times Q$, called the transition relation of A
>
> $Q_0 \subseteq Q$ is the set of initial states. We call N the name set and g the guard of the transition. For every transition $(q, N, g, p) \in \rightarrow$ we require that $(1)N \neq 0$ and $(2)g \in DC(N, Data)$.

2.3 CUDA

CUDA is an acronym for Compute Unified Device Architecture. CUDA was developed by NVIDIA, the world's leading creator of graphics cards and released on February 15th 2007 as a way to facilitate GPGPU, or General Purpose Graphics Processing Unit, programming. Prior to the release of CUDA researchers interested in harnessing the power of graphics cards to do parallel computation had to essentially trick 3D APIs to do the computational work they wanted done by using texture operations on matrices containing not textures but actual computational data. Obviously this was very challenging and proved a great limitation to the growth of GPGPU programming. CUDA completely changes the landscape of GPGPU programming by giving programmers a C like API. This meant that programming for the GPU became possible for anyone with knowledge of the C programming language and enough patience to learn the particulars of CUDA.

2.3.1 CUDA Applications

The applications for CUDA have been incredibly varied, ranging from physics acceleration of video games in the form of PhysX from NVIDIA to Geographic Information Systems (GIS) and medical imaging. Recently CUDA has even been used for breaking encryption at incredible speeds by speeding up the brute-force guessing of encryption keys (Gómez et al., 2010).

2.3.2 The Power of CUDA

The power of CUDA comes not only from having hundreds of processors and incredibly wide memory busses allowing very fast memory access but also from CUDA's execution model. A fast CUDA program will be designed to run over thousands of threads. This means that even running on hardware that has hundreds of processors context switches are necessary. GPU hardware is designed to be able to context switch very fast, so fast in fact that the performance impact of context switching is almost negligible. Switching threads out one by one would clearly not be feasible, since that would place a huge bookkeeping load on the system. This is why threads in CUDA are arranged into blocks. These blocks are then switched out rather than the individual threads. Since context switches need to happen anyway, it would be good to be able to use context switches to achieve even greater performance. In fact this is what CUDA does. When a block of threads needs to wait for a memory operation, it is switched out immediately and another block that is all ready to go is run in its place. This means that in the theoretical "perfect" program memory access is effectively nearly instantaneous since each block's memory accessing is done while another block is being run.

2.3.3 CUDA Programming Challenges

There are many challenges facing CUDA programmers. The principal of them being the stochastic order of execution, the fact that synchronization and locking have drastic performance implications and the fact that whenever data in the GPU context must be made available to the CPU context and vice versa said data must be transferred across the PCI-Express bus. The PCI-Express bus is, relative to the CPU memory bus and especially the GPU memory bus, very slow. The CUDA driver, the bit of code that essentially sits between the CPU context and the GPU context will decide which blocks to run first based on a black-box algorithm. Thus the CUDA developer can and should not concern himself with execution order and therefore must design his program in such a way that no matter which part of the input data is processed first the outcome of the program will not change. To illustrate the potential performance degradation caused by locks imagine one thread causing thousands of other threads to hold effectively turning the parallel processing behemoth that is the modern graphics card into what is effectively a rather weak single-threaded processor. Finally to underline the difference between PCI-Express and memory busses current graphics cards such as the NVIDIA GeForce 580 GTX have a memory bus capable of a theoretic maximum of 192.4 GB/sec (Corporation, 2011) while the PCI-Express 2.0 bus for that same card is capable of a relatively slow 16 GB/sec

(Group, 2011), a difference of more than an order of magnitude. It is therefore fairly obvious that the designer should avoid at all costs excessive transfers between the CPU and GPU contexts, as such transfers can have a serious negative impact on performance.

Chapter 3

The Method

To be able to utilize the power of CUDA parallelization a new kind of algorithm was needed. One that could be run on hundreds or even thousand of processors concurrently. This required an entirely new approach, both in terms of algorithm design and data representation. In this chapter we cover the challenges encountered and the solutions to those challenges as well as some of the broader ramifications of those solutions.

3.1 Algorithm Design

In designing an algorithm to run in a CUDA kernel it is necessary to be mindful of the architecture and runtime behavior of CUDA. Rather than one thread running in a serial manner there are hundreds or even thousands of threads running in parallel at any given moment during execution. This poses some interesting design challenges.

- The data must be laid out in such a manner that is we imagine the data needed for a single iteration to be a conceptual whole, let's call this entity a chunk, each chunk must be accessible independently from any other chunk.
- It can not matter whether a chunk's neighbor has already been processed or not, nor can the order in which the chunks are processed matter, nor the number of chunks being processed concurrently.
- While locks and atomic transactions are possible within CUDA they incur a large performance penalty and should therefore be avoided.

Current approaches to Constraint Automata composition (Pourvatan & Rouhy, 2007) rely heavily on the results of previous iterations for correctness and speed. Previous ap-

proaches have for instance split the data representation of Constraint Automata into what basically amounts to a collection of lists. The algorithm would then run searches through those lists and mark off transitions that had been checked and manipulated, thus previous implementations were clearly unsuitable. A massively parallel CUDA implementation can not use such structures or methods because of the previously discussed need for atomicity in iterations. Thus an entirely new approach had to be devised that would no only avoid these issues but also do so in such a way as to increase performance.

The central innovation of this project is the representation of Constraint Automata as adjacency matrices. With the use of adjacency matrices much of the data, which in previous work, is represented with lists of transitions, states and destinations is inherent in the very structure of the data representation. When generating a transition in the output constraint automaton adjacency matrix for a product it is only necessary to access two transitions, one in each of the input Constraint Automata adjacency matrices, in order to gain a complete picture of the transitions that need to be considered for the product. This certainty is due to the structure of the algebraic representation introduced in (Pourvatan & Rouhy, 2007) which ensures that searching for contradictory transitions is not necessary since any contradiction is encoded into the transition. In essence the way this works is that rather than each transition simply having the names of the constraints that allow that transition to fire also included are the constraints that must not be met in order to allow that transition to fire. This means that a search through the set of transitions for a contradictory transition is not necessary. While this constituted a great efficiency increase in the serial algorithm it is fairly essential for this work since while it would probably be possible without the algebraic representation it would be so inefficient as to probably very nearly negate any performance increase.

Due to this highly efficient encoding the algorithm itself is quite simple. In fact the only real computation the algorithm does is to ascertain whether there are any contradicting labels in the transitions being joined and if there are to drop the transition since there can clearly be no transition with contradicting labels.

3.1.1 Encoding

The data that needs to be encoded for each transition is its origin, destination and the names of the constraints of the activation of that edge. Since a constraint automaton can have more than one transition between each state but an adjacency matrix only has one entry for each adjacency the word edge shall be used for each adjacency matrix entry

and all transitions within said entry. This means that the word edge covers all transitions between a pair of states.

Each edge is represented as a list of names for its transitions. If an edge has more than one transition such as the example of a sync channel (figure 3.2) a separator is inserted and then the next transition's names are listed for all transitions in the edge. The final name for the final transition is then followed by a terminator character.

This structure was selected because of its space efficiency as well as the greater ease offered when programming the addressing and manipulation of multiple transitions.

The origin and destination of each edge is not included in the encoding of the transitions since they ares implicit in the structure of the adjacency matrix, namely the position of each transition in the adjacency matrix indicates what its origin is and what its destination is.

Using an adjacency matrix as opposed to a series of lists as in (Pourvatan & Rouhy, 2007) for instance therefore not only enables far greater atomicity of calculation but can also afford significant space saving. This is especially true when combined with sparse matrix representation such as is described in (Bošnački, Edelkamp, & Sulewski, 2009). Such representation eliminates dead entries in the adjacency matrix which would otherwise consume space needlessly.

3.1.2 Examples

Let's compare the representation of a constraint automaton for a primitive Reo channel known as a FIFO1 buffer. This channel can store one bit of data and forward it once it's output end is activated. The constraint automaton for the FIFO1 channel has two states, two names and in the algebraic representation four transitions. The adjacency matrix of this constraint automaton would be a 2×2 matrix which, since Constraint Automata are in essence directed graphs, can be asymmetrical.

If we look at the constraint automaton in Figure 3.1 we see it has two states, 1 and 2, an edge leading from state 1 to state 2 labeled $a\overline{b}$, a self loop on 1 labeled $\overline{a}\overline{b}$, an edge from 2 to 1 labeled $\overline{a}b$ and finally a self loop on 2 labeled $\overline{a}\overline{b}$. Note how transition A from state 1 to state 2 is in cell (1,2) and similarly transition B is in cell (2,1). The conventional representation would note the structure in a manner similar to this textual description in that there would be a list of states, transitions, labels and so forth. In the adjacency matrix representation however the structure of the graph would be inferred from the placement of entries in the matrix. Since state 1 has a self loop there would be an entry in cell (1,1)

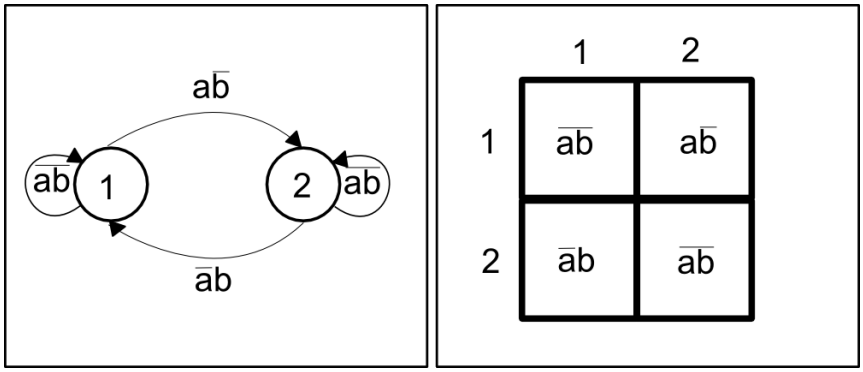

Figure 3.1: The graph and adjacency matrix representations of a FIFO1 Reo channel.

with the labels $\overline{a}\overline{b}$, since there is an edge from state 1 to state 2 there would be an entry in cell (1,2) with the labels $a\overline{b}$, since there is an edge from state 2 to state 1 there would be an entry in cell (2,1) with the label $\overline{a}b$ and finally the self loop on state 2 would be represented by an entry in the cell (2,2) with the labels $\overline{a}\overline{b}$.

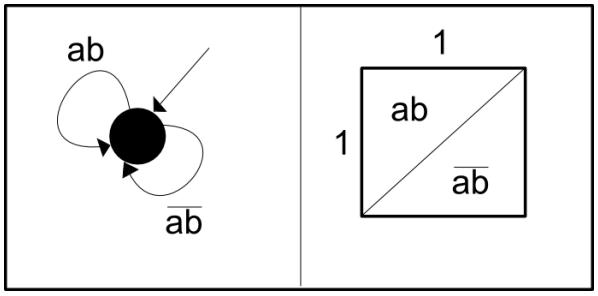

Figure 3.2: The graph and adjacency matrix representations of a SYNC Reo channel.

Similarly if a constraint automaton has more than one edge between any two states or more than one self loop on any state as in Figure 3.2 this is represented simply by having more than one entry in each cell. Thus if we were representing the sync Reo channel, which has one state and two self loops labeled ab and $\overline{a}\overline{b}$ respectively, we would use a $1x1$ matrix and in cell (1,1) we would put an entry with the labels ab and another with the label $\overline{a}\overline{b}$.

This data layout allows an algorithm that behaves much like a matrix product. Happily the vector hardware on graphics cards was in fact designed to handle matrix operations very efficiently since their normal job is to process textures. These are composed of pixels arranged into matrices, into screen renderings which are also pixel matrices. This means that matrix products are as close to the perfect job to port to CUDA as can be imagined. So perfect in fact that the very first demonstration programs any student studying CUDA writes will almost certainly include a matrix product.

The algorithm we developed is in fact a modified version of a matrix product algorithm for CUDA. Rather than each cell in the output matrix being the sum of the products of a whole row and a whole column, calculating the value of a cell in the output matrix only requires accessing one cell in the left hand side input matrix and the corresponding cell in the right hand side matrix. This obviously affords great efficiency gains as well.

The computation would then proceed in iterations where with each iteration the number of constraint automatons would be halved until we are left with a single adjacency matrix representing the union of the input Constraint Automata.

3.1.3 Broader Ramifications

What may not be apparent is the effect that the adjacency matrix representation will have on non-CUDA parallelization of Constraint Automata product calculation.

The fact that the representation of each state is very much independent of its neighboring states means that the calculation of Constraint Automata products can be split among many computers since the representation of each automaton can be split down into an arbitrary number of sections without affecting calculation efficiency.

This opens up huge possibilities for processing the gargantuan state spaces that the state explosion inherent to Constraint Automata products due to its exponential state growth.

To illustrate this point let's look at the FIFO1 constraint automaton mentioned earlier. This constraint automaton has 4 transitions to begin with. Once we have combined 2 of these the resulting constraint automaton has 16 states. Combine two of those and we have 256 transitions. Combining two of those gives us 65,536 transitions. This is where we start to really see the bite of state explosion since combining two of the previously discussed constraint automatons yields 4,294,967,296 transitions. That's over 4 billion transitions. A naive calculation assuming a size of 30 bytes for each transition gives us a total size of just under 129 gigabytes. While this is clearly way outside the capacity of any single graphics card the discrete nature of the data representation means that, with the ubiquitous availability of terabyte level hard disk drives, even consumer grade computers will be able to process Constraint Automata of this magnitude by simply partitioning the output Constraint Automaton's adjacency matrix representation, as well as the input Constraint Automata's adjacency matrices and processing one partition at a time by swapping out the relevant data on the card. We can also assume that hard drive latencies will not be much of an issue since even consumer grade modern computers have between 2 and 8 gigabytes of RAM meaning that while one partition is being processed on the GPU

the previous partition's output can be written to disk and any necessary input data can be cached up in RAM ready to be swapped onto the GPU for processing.

The design of the our structure does by no means isolate its usefulness to graphics cards since the layout of the data structure makes it ideally suitable for grid and cloud computation. The atomic nature of the data structure means that it could be spread out across an arbitrary number of nodes with absolutely no communication or synchronization and fairly little data replication needed. Each node would be allocated a portion of the output data structure as well as the portions of the input data structures needed to build the output. There would inevitably be some overlapping of input data. This means several nodes would need to access the same data in the input data structures. The fact that the input data structures are not changed at all allows for portions of the input data structures to simply be duplicated across nodes. This means that even though the product of two of the previously discussed 4,3 billion edge constraint automatons would be ludicrously large it could, in theory, be calculated given a large enough grid of computers and enough processing time. The calculation of the next step up however would probably outlast the sun so we will not consider it.

3.2 Implementation

The implementation is effectively split into two parts, namely the host code and the GPGPU code, henceforth known as the kernel according to GPGPU programming convention. The host code takes care of memory allocation and deallocation, initializing the data structures, splitting the task up into blocks, allocating thread count for blocks and in general doing all the housekeeping and management. The main kernel on the other hand contains the functional portion of the code as it is responsible for the actual calculations and implements the algorithm itself. The main kernel also calls a few helper functions that for one reason or another did not belong inside the kernel itself. A pseudo code representation of the kernel follows.

Algorithm 1 Main kernel function

```
init coordWidth
init coordHeight
for x = 0 to inputWidth do
  for y = 0 to inputHeight do
    if itercount == threadIdx.x then
      coordWidth.x ← x
      coordWidth.y ← y
    end if
    if iterCount == threadIdx.y then
      coordHeight.x ← x
      coordHeight.y ← y
    end if
    iterCount++
  end for
end for
outerIterator ← 0
innerIterator ← 0
prodIterator ← 0
isNotValue ← 0
while inputArrLeft[(coordWidth.x×2)+coordHeight.x].names[outerIterator] ≠
0 do
  while inputArrRight[(coordWidth.y × 2) +
  coordHeight.y].names[innerIterator] ≠ 0 do
    isNotValue ← isNotValue + isNot(inputArrLeft[(coordWidth.x × 2) +
    coordHeight.x].names[outerIterator], inputArrRight[(coordWidth.y × 2) +
    coordHeight.y].names[innerIterator])
    innerIterator++
  end while
  innerIterator ← 0
  outerIterator++
end while
```

Algorithm 2 Main kernel function cont.

if $isNotValue == 0$ **then**
  $outerIterator \leftarrow 0$
  $innerEterator \leftarrow 0$
  while $inputArrLeft[(coordWidth.x \times 2) + coordHeight.x].names[outerIterator] \neq 0$ **do**
    while $inputArrRight[(coordWidth.y \times 2) + coordHeight.y].names[innerIterator] \neq 0$ **do**
      $outputArr[threadIdx.x \times 4 + threadIdx.y].names[prodIterator++] \leftarrow inputArrLeft[(coordWidth.x \times 2) + coordHeight.x].names[outerIterator]$
      $outputArr[threadIdx.x \times 4 + threadIdx.y].names[prodIterator++] \leftarrow inputArrRight[(coordWidth.y \times 2) + coordHeight.y].names[innerIterator]$
      $innerIterator++$
    end while
    $innerIterator \leftarrow 0$
    $outerIterator++$
  end while
  $outputArr[threadIdx.x \times 4 + threadIdx.y].names[prodIterator++] \leftarrow 0$
  $outputArr[threadIdx.x \times 4 + threadIdx.y] \leftarrow dupRemove(outputArr[threadIdx.x \times 4 + threadIdx.y])$
end if

Algorithm 3 dupRemove() function which remove duplicate transition labels

while $in.names[nameCount] \neq 0$ **do**
  $nameCount++$
end while
if $nameCount > 0$ **then**
  for $i = 0$ **to** $nameCount - 1$ **do**
    for $j = i + 1$ **to** $nameCount$ **do**
      if $in.names[i] - in.names[j] == 0$ **then**
        $removePlaces[removeCount] \leftarrow j$
        $removeCount++$
      end if
    end for
  end for
  for $x = removeCount - 1$ **to** 0 **do**
    $nameCount \leftarrow nameCount - 1$
    for $j = removePlaces[x]$ **to** $j < nameCount$ **do**
      $in.names[j] \leftarrow in.names[j + 1]$
    end for
  end for
  $in.names[nameCount] \leftarrow 0$
end if
return in

Algorithm 4 isNot() function which checks whether the two input labels are contradictory by checking if the two input characters are upper and lowercase versions of the same character since NOTed characters are represented as upper case. In ASCII the difference between upper and lowercase is 32

$result \leftarrow left - right$
$result \leftarrow absolute(result)$
if $result == 32$ **then**
  return 1
else
  return 0
end if

3.2.1 Architecture

To implement the algorithm for joining two Constraint Automata first the data structure meant to hold the adjacency matrices had to be decided upon. For simplicity and robustness it was decided to forgo a 2 dimensional array for a 1 dimensional array. Normally such a decision would have to be balanced against the extra calculations needed to address a 1 dimensional array as if it were a 2 dimensional array. However due to the algorithm being very memory bound it was decided that the few extra calculations needed for addressing would not be a factor.

Since the addressing of the component transitions of each combined transition is not straightforward for each combined transition the positions it should look at for its component transitions is calculated on a per transition basis in a space vs. time trade-off once again due to the fact that the algorithm is memory bound and a data structure that would hold the positions of each combined transitions component transitions would either have to be stored in global memory, which is slow, or duplicated into shared memory for each block which is inefficient space-wise.

Chapter 4

Experiments

To benchmark the performance of the GPU versus the CPU versions of the algorithm the combination of two four state FIFO1, themselves the results of the composition of a pair of two state FIFO1s, was repeated up to a thousand times. The rationale behind this decision was as follows. Firstly anything smaller would have caused the multiprocessors in the CUDA cards to have been largely idle and would therefore have lead to an unfair test and anything larger would not have fitted into the available on board memory of the cards. Secondly fewer iterations would have pushed the limits of reliable timing. In fact a preliminary test involving 5 iterations was over in a matter of microseconds and the timing mechanisms used were not designed to be able to accurately measure such short durations accurately and thus repeatability would have suffered.

4.1 Experimental Results

To assess the performance of the algorithm, two versions were implemented. One version was written in C using the CUDA API from graphics card manufacturer NVIDIA to run on their graphics cards. A second reference version was written in C but this one was made to run on normal processors. Both versions were written without the aid of any libraries or special purpose enhanced data types to eliminate any possible discrepancies and make the comparison as fair as possible.

It should be noted that due to time constraints the CUDA code was written without any sort of performance optimization and that much headroom is available as far as performance optimization is concerned in the CUDA code. We believe that with proper performance optimization, performance can by increased significantly. In fact in (Nickolls,

Buck, Garland, & Skadron, 2008) performance was increased significantly by using the shared memory built into each CUDA multiprocessor. This shared memory operates at processor speed, like the cache on a traditional microprocessor, and that alone should therefore increase performance significantly. There are also many other techniques, many of them discussed in (Nickolls et al., 2008), that could be utilized to gain further improvements.

Despite the CUDA code having been written in a very inefficient manner it still manages to outperform the CPU code by up to a factor of 4 as can be seen in Table 4.1 and Table 4.2.

Table 4.1: GPU vs. CPU Averages

Iterations	100	200	300	400	500
GTX570 GPU Averages	1,5566304	3,06376	4,574966	6,063792	7,623904
E8400 CPU Averages	3,92294	7,812669	11,67134	15,66299	19,54998
T7100 CPU Average	6,6391209	13,28748	19,33361	26,15898	32,22918

Table 4.2: GPU vs. CPU Averages cont.d

Iterations	600	700	800	900	1000
GTX570 GPU Averages	9,1789376	10,61772	12,06964	13,70077	15,2143
E8400 CPU Averages	23,3996744	27,24991	31,23446	35,27192	39,13096
T7100 CPU Average	39,850584	44,03374	51,96439	57,65711	68,8065

The benchmarks were run on three distinct platforms, each belonging to a separate performance category. These were the NVIDIA GeForce GTX570 graphics card, the Intel E8400 processor and the Intel T7100 processor. The specifications of these platforms are listed in Table 4.3. These specifications underline the significance of the findings of this project. Note that the E8400 has more than twice the clock rate of the GTX570 and only about 30 percent slower memory interface and yet Tables 4.1 and 4.2 show the GTX570 is roughly twice as fast through all the tests as the E8400. Finally since both the GTX570 and the E8400 are fairly high performance parts the T7100 was added as representative of a lower performance bracket. Despite having a faster core clock than the GTX570 the T7100 was roughly four times slower than the GTX570.

Both the GTX570 and the E8400 returned beautifully consistent benchmark results. In fact Figure 4.1 shows both of them having very straight lines indicating very consistent data and therefore a valid test. The T7100 line however is less consistent and this is believed to be either a result of background processes in the Windows operating system or perhaps more aggressive power saving settings since the T7100 is a laptop part. In

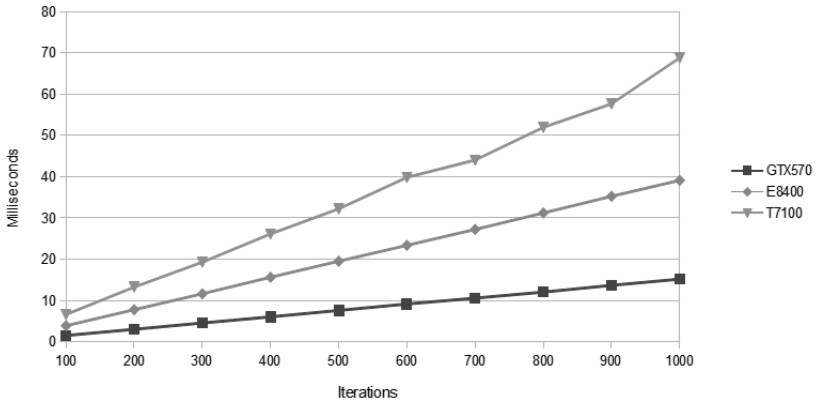

Figure 4.1: GPU vs. CPU Performance

Table 4.3: Benchmarking Platforms

Platform	NVIDIA GeForce GTX570 GPU	Intel E8400 CPU	Intel T7100 CPU
Number of Processing Cores	480	2	2
Processing Core Speed (MHz)	1464	3000	1800
Memory Speed (MHz)	1900	1333	800
Memory Size (MB)	1280	2048	2048

fact the effects of power saving technologies such as clock speed manipulation are very evident in the raw run times in Tables 4.6, 4.7, 4.8 and 4.9. Note how the first few runs are almost always slower than the following. According to (Hoban, 2010) this is because operating systems poll CPU demand in time slices and apparently these time slices are large enough that the first few runs are over before the operating system switches to a higher performance state. Due to this effect, the CPU benchmarks were run 15 times whereas the GPU benchmarks were run 10 times since the GPU did not exhibit this behavior. The first five runs were then ignored when calculating average run times for the CPU benchmarks.

Table 4.4: GTX570 Run Times in Milliseconds

Iterations	100	200	300	400	500
	1,564896	3,058496	4,571968	6,024704	7,647776
	1,565088	3,061344	4,585344	6,090752	7,587424
	1,5488	3,0536	4,595648	6,089408	7,517376
	1,547552	3,07216	4,581312	6,070944	7,655008
	1,5464	3,054976	4,562976	6,089984	7,60496
	1,566112	3,066912	4,56736	5,94176	7,583424
	1,544736	3,081856	4,563584	6,072128	7,70144
	1,567392	3,072032	4,570144	6,091872	7,671712
	1,56672	3,098432	4,582944	6,070976	7,671712
	1,548608	3,017792	4,568384	6,095392	7,598208
Average	**1,5566304**	**3,06376**	**4,5749664**	**6,063792**	**7,623904**

Table 4.5: GTX570 Run Times in Milliseconds cont.d

Iterations	600	700	800	900	1000
	9,180928	10,600256	12,092288	13,677216	15,239872
	9,284416	10,637792	12,072768	13,687808	15,21536
	9,112384	10,605536	12,094688	13,72528	15,202816
	9,115712	10,633216	12,060384	13,710528	15,222368
	9,143456	10,600736	12,096576	13,683904	15,1976
	9,115328	10,59696	12,065216	13,703616	15,20752
	9,093152	10,658688	12,05552	13,691584	15,212864
	9,117504	10,632704	12,072224	13,74896	15,20352
	9,512416	10,621248	12,013408	13,650208	15,240704
	9,11408	10,59008	12,073312	13,728608	15,200416
Average	**9,1789376**	**10,6177216**	**12,0696384**	**13,7007712**	**15,214304**

Table 4.6: E8400 Run Times in Milliseconds

Iterations	100	200	300	400	500
	3,872048	11,745507	12,121311	20,905439	19,646273
	3,883312	9,625848	11,585082	15,595368	19,666753
	3,902085	7,915443	11,61785	15,701181	19,73024
	3,87785	7,808265	11,775885	15,627795	19,639446
	3,91369	7,814751	11,708302	15,753404	19,536365
	3,907205	7,844105	11,722638	15,757159	19,442499
	3,919834	7,750581	11,674852	15,696743	19,671531
	3,895258	7,803145	11,684068	15,830886	19,586199
	3,957721	7,748191	11,637988	15,60595	19,396078
	3,870341	7,794271	11,545488	15,610728	19,608727
	4,072067	7,74785	11,645497	15,77798	19,63023
	3,893893	7,856393	11,803533	15,55987	19,490626
	3,916421	7,82738	11,635599	15,63428	19,590295
	3,895941	7,905203	11,656761	15,586835	19,583809
	3,900719	7,849566	11,706937	15,569427	19,499842
Average	**3,92294**	**7,8126685**	**11,6713361**	**15,6629858**	**19,5499836**

Table 4.7: E8400 Run Times in Milliseconds cont.d

Iterations	600	700	800	900	1000
	33,623054	27,41392	31,359695	42,697654	44,242514
	23,516614	27,453514	31,510221	35,057323	39,124611
	23,700249	27,485941	31,676449	35,961506	39,089113
	23,705028	27,692445	31,326927	35,164159	39,205507
	23,462684	27,24974	31,192102	35,062102	38,850182
	23,505009	27,194786	31,251835	35,206143	38,972037
	23,491355	27,298892	31,26856	35,116032	39,009242
	23,391004	27,355894	31,189713	36,007586	38,785329
	23,26403	27,232674	31,204731	35,14641	38,94473
	23,24867	27,266124	31,193809	35,199316	39,099353
	23,316595	27,317324	31,161382	35,321171	39,299031
	23,425479	27,197858	31,517389	35,151189	39,265922
	23,485212	27,220386	31,22999	35,106474	38,979546
	23,368477	27,203661	31,113937	34,985302	39,059417
	23,500913	27,211511	31,213264	35,479548	39,894993
Average	**23,3996744**	**27,249911**	**31,234461**	**35,2719171**	**39,13096**

Table 4.8: T7100 Run Times in milliseconds

Iterations	100	200	300	400	500
	8,08405	13,142357	30,084007	26,661791	33,019616
	6,697251	13,312355	21,524189	26,492363	33,505652
	6,230611	13,250175	19,618268	26,656086	33,421223
	6,653325	13,432723	18,83445	26,896252	33,533034
	6,734902	13,149203	19,55894	26,443303	32,976831
	6,484468	13,309503	19,55894	26,871722	33,251795
	6,579736	13,140646	19,610282	26,442733	32,011606
	6,79366	13,417891	18,477339	26,510618	30,708095
	6,655607	13,290107	19,727797	26,430183	31,163325
	6,620809	13,132089	19,526994	26,712562	31,578623
	6,706949	13,136652	19,752898	25,897939	31,885533
	6,697251	13,140075	18,287945	24,744461	33,623167
	6,628225	13,276416	19,935446	25,645224	32,340763
	6,551783	13,555373	18,481333	26,341761	32,526735
	6,672721	13,476079	19,97709	25,992636	33,202165
Average	**6,6391209**	**13,2874831**	**19,3336064**	**26,1589839**	**32,2291807**

Table 4.9: T7100 Run Times in milliseconds cont.d

Iterations	600	700	800	900	1000
	40,456759	44,452291	61,428739	68,251493	77,737744
	38,570804	45,618321	50,879143	55,912349	61,553101
	41,236014	46,313146	52,923116	58,847388	61,843467
	40,612496	47,105521	51,766215	59,425269	64,115056
	41,262255	46,092376	52,802178	58,386453	62,625573
	39,594788	46,179087	53,262543	56,432613	63,98556
	40,979305	44,302259	50,18774	59,003695	64,148143
	40,119615	43,478509	50,896827	58,51823	65,707222
	40,257097	43,813942	49,974386	56,353318	66,629663
	39,960455	43,187572	52,407987	55,74121	65,702088
	40,24854	43,88525	52,855801	55,693861	67,245194
	39,360897	44,213837	53,061739	55,972819	81,027612
	39,099625	44,865308	51,806147	56,899253	67,010162
	39,391132	42,810495	52,155271	60,490325	69,444904
	39,494386	43,601159	53,035498	61,465819	77,164427
Average	**39,850584**	**44,0337418**	**51,9643939**	**57,6571143**	**68,8064975**

Chapter 5

Conclusions

5.1 Conclusions and Future Work

It has been shown that there is real benefit in parallelizing the composition of Constraint Automata. A performance benefit of 100 percent and up in terms of speed of calculation has been shown. Beyond pure speed it has been shown that one of the greatest obstacles to the composition of large Constraint Automata, namely state explosion, can be overcome with the methods detailed here. By splitting the task of combining Constraint Automata over multiple machines the limitations of memory addressing and hardware costs may be ameliorated by distributing the task over many machines. In fact by combining the two core methods detailed here it is possible to tackle both limitations of large Constraint Automata combination. Running through millions of states takes a long time but by having many machines working on the problem will considerably shorten computational time. Storing millions of states takes up a lot of space but distributing the state-space over many machines makes it possible to avoid the hard limits of memory addressing. The atomicity of data introduced here will even make caching data to storage easier and more efficient by making data accesses more predictable.

Even though only one type of Constraint Automaton was used in testing due to time constraints other types should give very similar results due to the structure of the algorithm and the data representation. No matter what kind of Constraint Automaton is used the only differences will be the sparseness of the adjacency matrix and the number of transitions in each cell. Transition counts greater than one mean that there are more transitions to combine and thus more work for each thread to do. However since the algorithm does no calculation more strenuous than a value comparison and is therefore clearly memory bound this should not have a noticeable effect. The sparseness of the adjacency matrix will

also have very little impact on the speed of calculation beyond offering greater potential for optimization since once a thread hits a null transition it has nothing to do. This is due to the fact that with the current data representation each and every cell in the adjacency matrix is checked but if sparse matrix representation were to be used that would mean that only non-empty cells would be checked. This would then mean that more sparse adjacency matrices would be processed more rapidly.

5.2 Future Work

This work barely scratches the surface of the possibilities of parallelization. Future directions of research include distributing the combination of Constraint Automata over multiple nodes, exploring the feasibility of bypassing the memory limitations of current GPUs by storing the main Constraint Automata data structure in the memory of the host machine and then moving it portion by portion on to the GPU for processing or even a combination of both.

Clearly further research is needed.

Bibliography

Arbab, F. (2004, June). Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, *14*, 329–366. Available from http://dl.acm.org/citation.cfm?id=992032.992035

Arbab, F., Baier, C., Rutten, J., & Sirjani, M. (2004). Modeling component connectors in reo by constraint automata: (extended abstract). *Electronic Notes in Theoretical Computer Science*, *97*(0), 25 - 46. Available from http://www.sciencedirect.com/science/article/pii/S157106610405039X (Proceedings of FOCLASA 2003, the Foundations of Coordination Languages and Software Architectures, a satellite event of CONCUR 2003)

Bošnački, D., Edelkamp, S., & Sulewski, D. (2009). Efficient probabilistic model checking on general purpose graphics processors. In C. Pasareanu (Ed.), *Model checking software* (Vol. 5578, p. 32-49). Springer Berlin / Heidelberg. Available from http://dx.doi.org/10.1007/978-3-642-02652-2_7

Corporation, N. (2011, September). *Geforce 580 gtx specifications.* http://www.geforce.com/Hardware/GPUs/geforce-gtx-580/specifications.

Gómez, J., Montoya, F., Benedicto, R., Jimenez, A., Gil, C., & Alcayde, A. (2010). Cryptanalysis of hash functions using advanced multiprocessing. In A. de Leon F. de Carvalho, S. Rodríguez-González, J. De Paz Santana, & J. Rodríguez (Eds.), *Distributed computing and artificial intelligence* (Vol. 79, p. 221-228). Springer Berlin / Heidelberg. Available from http://dx.doi.org/10.1007/978-3-642-14883-5_29

Group, P. S. I. (2011, September). *Pcie 3.0 faq.* http://www.pcisig.com/news_room/faqs/pcie3.0_faq/PCI_Express_3_0_FAQ_06092011.pdf.

Hoban, A. (2010, 05). *Designing real time systems on embedded intel architecture processors* (Tech. Rep.). Intel. Available from http://download.intel.com/

design/intarch/papers/323671.pdf

Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008, March). Scalable parallel programming with cuda. *Queue*, *6*, 40–53. Available from http://doi.acm.org/10.1145/1365490.1365500

Pourvatan, B., & Rouhy, N. (2007). An alternative algorithm for constraint automata product. In *Proceedings of the 2007 international conference on fundamentals of software engineering* (pp. 412–422). Berlin, Heidelberg: Springer-Verlag. Available from http://dl.acm.org/citation.cfm?id=1775223.1775252

School of Computer Science
Reykjavík University
Menntavegi 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.reykjavikuniversity.is
ISSN 1670-8539