Master Thesis

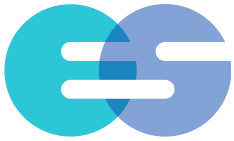
Dennis Keil

Graph Algorithms for a MISD Computer Architecture (Micron Automata Processor)

December 2016

Supervised by: Prof. Dr. Karl-Heinz Zimmermann
Prof. Dr. Heiko Falk

Hamburg University of Technology
Institute of Embedded Systems
Am Schwarzenberg-Campus 3 (E)
21073 Hamburg
Germany



Abstract

This master thesis describes the Micron Automata Processor, a novel co-processor based on the MISD (Multiple Instruction Single Data) architecture. It was developed for highly parallelized execution of non-deterministic finite automata and provides effective solutions for the modeling of regular expressions and pattern matching. This makes it a promising tool for handling a number of NP-complete problems which cannot be solved efficiently using existing processors.

A well-known problem of this kind is the clique problem which is about finding complete subgraphs in a graph. This document presents two algorithms for solving this problem using the automata processor. A brute-force algorithm for finding all k-cliques and a branch-and-bound algorithm for finding the maximum clique are described.

The presented algorithms are implemented as automata and compared to existing CPU-based solutions.

Ackknowledgements

I wish to express my sincere thanks to Prof. Dr. Zimmermann for his ongoing support during the time of the research project. Also many thanks to Wolfgang Brandt and Prof. Dr. Heiko Falk for sharing expertise and valuable guidance. Furthermore I want to thank my parents for their encouragement and support.

Declaration of Authorship

I declare that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other University.

Formulations and ideas taken from other sources are cited as such. This work has not been published.

Date: _____ Signature: _____

Contents

List of Figures

List of Algorithms

List of Tables

1 Introduction

1.1 Motivation

Many difficult computing problems require high-speed search and analysis of complex data streams. This functionality cannot be implemented well by traditional CPU and memory systems. Therefore Micron Technology Inc., one of the world's leading manufacturers of semiconductors, developed the Automata Processor (AP) based on the MISD (Multiple Instruction Single Data) architecture.

The AP is a co-processor that enables direct implementation of several thousand non-deterministic finite automata and works on large streams of unstructured data. It provides effective solutions for the modeling of regular expressions and pattern matching. The automata processor is easier to program than existing FPGA-based hardware while delivering better performance. As such it enables fast computation of problems from various fields such as computational biology, graph analysis, network security and more.

1.2 Objectives

The task of this work is to measure the acceleration potential of the Automata Processor for graph analysis problems. For this purpose the NP-complete clique problem is chosen which is about finding complete subgraphs in a graph. For solution of this problem the *Brute-Force* and the *Branch-and-Bound* algorithm are developed and implemented as automata. Finally the performance of these automata is analysed and compared to existing CPU-based solutions.

1.3 Structure

Chapter 2 describes *non-deterministic and deterministic automata* and their implementation on generic sequential processors.

Chapter 3 introduces the Micron Automata Processor (AP). First the hierarchical hardware layout which enables direct implementation of automata is presented. Then the Automata Network Markup Language for description of automata is introduced. Afterwards the workflow of designing and running automata is described and important command line tools are explained. The whole workflow can be done programatically using two APIs which are described next. Finally guidelines for optimization of automata for the hardware layout of the AP are described.

Chapter 4 gives a short introduction to *graph theory* and introduces the *clique problem*.

Chapter 5 presents the *Brute-Force* algorithm for finding all k-cliques of a graph using an automata on the AP.

Chapter 6 presents the *Branch-and-Bound* algorithm for finding the maximum clique of a graph. An automaton for solving the problem is described and optimized for the hardware structure of the AP. The resource usage and performance of the automaton are analysed and compared to existing CPU-based solutions.

Chapter 8 deals with possible improvements of the algorithms and automata.

Chapter 9 conclusions are presented.

2 Automata

In this section we introduce two variants of finite automata and describe some of their properties. Furthermore we describe implementations on sequential processors and their performance.

2.1 Deterministic and Non-Deterministic Automata

Definition 2.1 (Deterministic Finite automaton (DFA)) *[18]*
A deterministic finite automaton (DFA) is a 5-tuple (Q, Σ, T, S, F) where

- *Q is a finite set of states,*
- *Σ is a finite set of symbols called alphabet,*
- *$T \subseteq (Q \times \Sigma) \times Q$ is a transition function,*
- *$q_0 \in Q$ is the start state, and*
- *$F \subseteq Q$ is a set of accepting states.*

An automaton runs on a given *input sequence* consisting of symbols from the alphabet Σ. At the beginning of the sequence the automaton is in the start state q_0. It then incrementally consumes one symbol after another from the sequence and updates its state according to its transition function T. After reading the whole sequence the automaton terminates in the so called final state. If this state is in the set of accepting states F, the input sequence is accepted by the automaton. Otherwise it is rejected. The set of all input sequences that are accepted by the automaton is called the *language* recognized by the automaton.

An automaton can be represented as a *state-transition diagram* (see figure 1). It contains a labeled node for each state in Q (here $S_1, \ldots, S_5$).

The nodes are connected according to the transition function. For each state q in Q and each input symbol a in Σ, let $T(q, a) = p$. Then the transition diagram has an arrow from node q to node p, labeled a.

There is an arrow into the start date (here S_1) and nodes corresponding to accepting states F are marked by a double circle (here S_3, S_5, S_6, S_7).

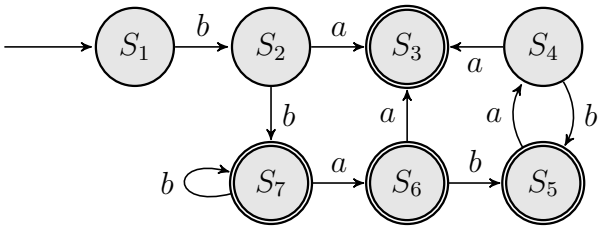


Figure 1: State-transition diagram of a DFA

There exist another group of automata called *non-deterministic automata* which have a transition relation instead of a transition function. This means there may exist multiple transition options from one state when a specific input symbol is read by the automata. In the diagram (figure 2) this will lead to multiple arrows with the same label exiting the same node (here S_3, b).

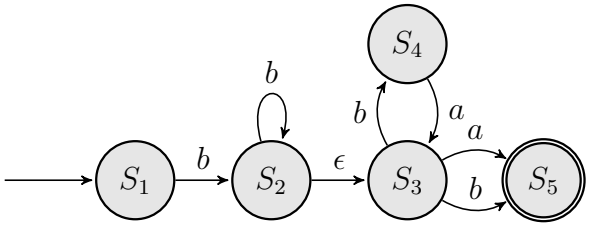


Figure 2: State-transition diagram of a NFA

Furthermore there exist ϵ-transitions (here from S_2 to S_3) which enable the automaton to change states without consumption of an input symbol. The ϵ-closure $\Gamma(S_i)$ of a state S_i is the state itself and all states that are reachable from S_i by following ϵ transitions (e.g. $\Gamma(S_2) = \{S_2, S_3\}$).

2.2 Implementation on CPU

Automata are usually modeled on von Nuemann sequential processors (CPUs). These processors are not optimized for such calculations yielding up to exponential processing and storage complexity depending on the type of automata.

An NFA can be modeled on a CPU by defining the set of states Q and their connections Σ. The number of states is $n = |Q|$.

During execution the CPU maintains the list of active states in memory. At each time step the CPU reads a smbol from the input sequence and compares it to every outgoing connection of every active state. These comparisions cannot be done in parallel but instead have to be done sequentially. In the worst case all states are active and each state is connected to each other yielding a processing complexity of $O(n^2)$. The storage complexity is $O(n)$ as only the active states have to be maintained in memory.

	Processing Complexity	Storage Cost
CPU - NFA	$O(n^2)$	$O(n)$
CPU - DFA	$O(1)$	$O(2^n)$

Table 1: Complexity for automata implementation on CPU [17]

A DFA can have only a single active state at each time step and each state can only transition to one other state given a specific input symbol. This is why the implementation of a DFA needs one comparision when reading a symbol yielding a processing complexity of $O(1)$. As a DFA has a state for each possible combination of active states of the equivalent NFA the storage complexity is much higher. In the worst case the DFA has a state for each combinations of states of the NFA which is equivalent to the power set of the states of the NFA. That is why the storage complexity is $O(2^n)$.

3 Micron Automata Processor (AP)

The Micron Automata Processor is based on the MISD (Multiple Instruction Single Data) architecture. It was developed for highly parallelized execution of non-deterministic finite automata and provides effective solutions for the modeling of regular expressions and pattern matching.

The automata processor was designed as an acceleration device rather than a standalone utility. It provides a PCI Express interface for connection to a host application on a CPU-based system. The host appplication can stream input symbols to the AP and read from output buffer of the AP aynchronously.

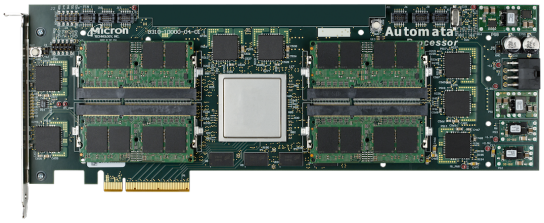


Figure 3: Micron Automata Processor PCIe Card [9]

The AP enables direct implementation of NFAs with a size of $O(n)$ where n is the number of states of the NFA. In contrast to sequential processors, the AP can transmit an input symbol throughout the complete unit in one clock cycle. This enables the processor to compute all state changes in parallel yielding a processing cost of $O(1)$. The constant processing cost regardless of automata complexity is a great advantage over generic CPU solutions.

	Processing Complexity	Storage Cost
CPU - NFA	$O(n^2)$	$O(n)$
CPU - DFA	$O(1)$	$O(2^n)$
AP - NFA	$O(1)$	$O(n)$

Table 2: Complexity for automata implementation on CPU vs. AP [17]

Micron Technology is a leading manufacturer of Dynamic Random Access Memory (DRAM). This technology is slightly modified to form the foundation of the AP. In a DRAM chip a memory adress and operation can be broadcasted to every memory cell on every clock cycle. In the AP these memory cells are replaced by processing elements which receive a symbol from the input sequence on every clock cycle. One clock cycle lasts $t_{clock} = 7.45 \cdot 10^{-9}\,\mathrm{s}$ yielding a clock rate of 134.23 MHz. As one character is one byte the AP has a processing rate of $134.23\,\mathrm{MHz} \cdot 8\,\mathrm{bit} = 1.074\,\mathrm{Gbps}$. The elements in the AP are connected through a programmable routing network (ARM) and can activate each other in every clock cycle. When an element reports a match it notifies the output handling unit which streams the results to the host application.

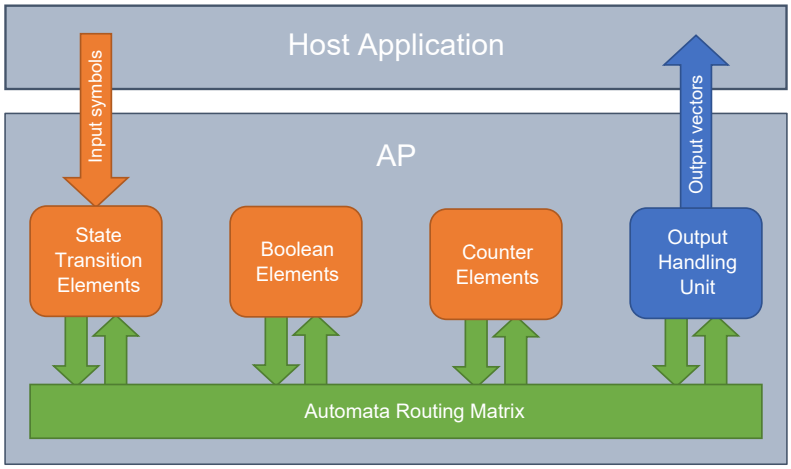


Figure 4: Interface between CPU and AP [11]

For the programming of the AP there exists a Software Development Kit (SDK) which can be downloaded from an online developer portal [5]. The SDK enables definition and compilation of an automaton and loading it onto an AP. Afterwards all runtime operations like streaming of input sequence and reading of output buffer can be handled. Furthermore the SDK provides the ability to simulate the execution of automata with limited complexity. There does also exist a visual interface called *AP Workbench* that can be used for creation and simulation of automata.

3.1 Automata Elements

The fundamental building blocks of automata on the AP are State Transition Elements (STE). These elements can be either active or inactive and describe the state of the automata. The STEs are connected to the input sequence and to each other through a routing matrix. This enables state changes of an automaton based on current state and input symbol. As such any automaton can be implemented on the AP as long as it does not exceed the available number of STEs or capability of the routing system. The AP also has two additional element types, *counter elements* and *boolean elements*. By using these elements for counting and boolean logic instead of STEs more complex automata can fit on the AP. The AP Chip has 49 152 STEs, 768 counter elements and 2 304 boolean elements.

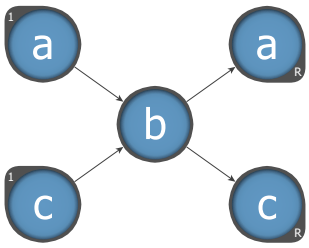


Figure 5: Automaton with 5 State Transition Elements (STE)

The execution of an automata on the AP works as follows. In the beginning all STEs are inactive, except those marked as starting elements. Then at each clock cycle one symbol is consumed from the input stream and broadcasted to all STEs. Each STE compares this symbol to its set of symbols it accepts. If an STE is active and accepts the current input symbol, then all STEs that are connected through the routing matrix are activated for the next clock cycle. This process terminates when there are no active STEs or the input stream ends.

For output generation STEs can be marked as *reporting elements* and associated with a report code $R \in [0, 2^{64}]$. If a reporting STE is active and accepts the current input symbol, it generates an output vector with its ID and the current stream offset. This vector is transmitted through the routing matrix to the output buffer where it can be read asynchronously by the host application.

Figure 6: Reporting STE

Every STE has an associated *start type* which specifies on which symbol cycles it is active. The following start types are available:

- None - The STE can only be activated by other STEs in the automaton.
- Start-of-data - The STE is only active on the first symbol cycle. It can also be activated by other STEs in the automaton in remaining symbol cycles.
- All-input - The STE is active on all symbol cycles.

Figure 7: Start types *none*, *start-of-data*, *all-input*

3.2 Hierarchical Structure

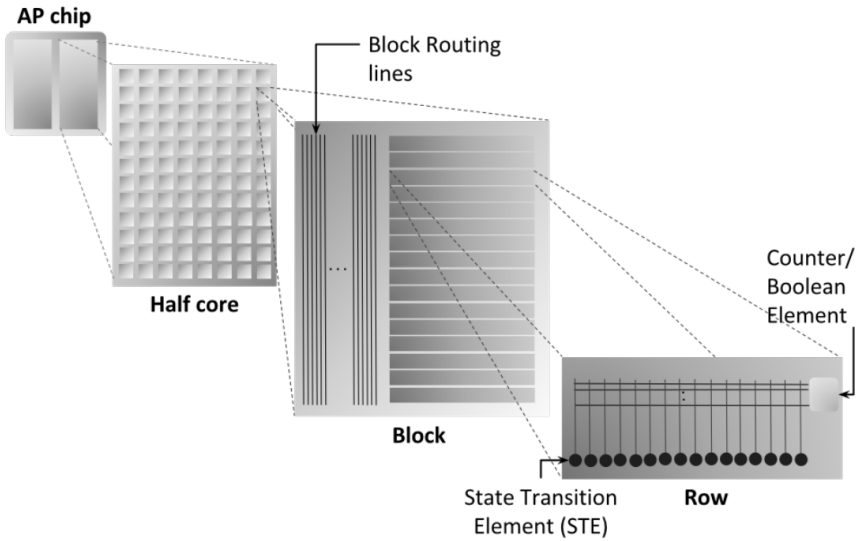


Figure 8: Hierarchical layout of an AP chip [16]

The state transition, counter and boolean elements on an AP chip are arranged hierarchically into rows, blocks, and half-cores. A row contains 16 STEs and one boolean or counter element. 16 rows form a block whereby 12 rows have a boolean element and 4 rows have a counter element. 96 blocks are arranged in a grid to form a half-core. Two half-cores are on one AP chip.

The routing capability reduces as we move up the hierarchy. The STEs in a row can be connected to each other and to the boolean or counter element in the row. To connect STEs from different rows, there exist 24 block routing lines that are shared by the 16 rows in a block. The connectivity between blocks is even more limited. STEs can only be connected to other STEs of the neighbouring 8 blocks in a limited number. Half-cores do not have a connection possibility. Therefore the upper limit on the size of an automaton is the size of a half-core.

The manufacturer Micron sells multiple variants of AP boards with up to 48 AP chips. These AP chips are arranged into *ranks* consisting of up to 8 chips. A large automaton that does not fit into a single AP chip can be split onto 2, 4 or 8 AP chips in a rank forming a *logical core*. All AP chips of a logical core receive the same data stream from the high-speed intra-rank bus. Therefore a logical core of 8 AP chips has a processing rate of 1 Gbps. However if all patterns can be fit inside a single chip, then they can be replicated on all 8 chips on the rank and 8 different streams can be processed in parallel yielding a processing rate of 8 Gbps.

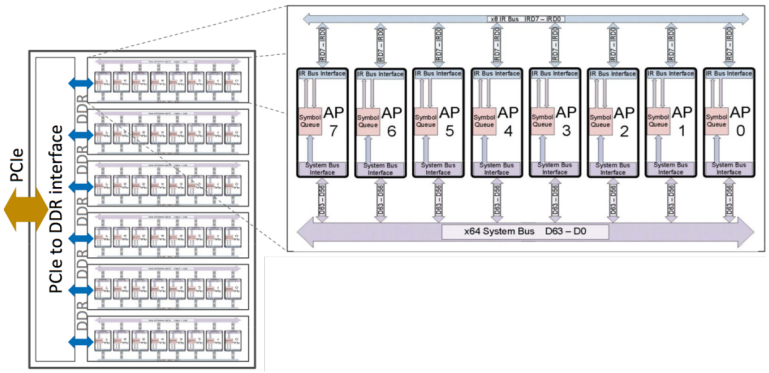


Figure 9: Hierarchical layout of AP board with 48 AP chips [11]

The smallest AP board contains two AP chips which are organized into a single rank. The second AP board contains 32 AP chips organized into 4 ranks containing 8 chips each. The third AP board is the largest of the three and contains 48 AP chips organized into 6 ranks.

AP Chips	Interface	STE	Counter	Boolean	Processing Rate
2	USB	98 304	1 536	4 608	1 - 2 Gbps
32	PCIe	1 572 864	24 576	73 728	1 - 32 Gbps
48	PCIe	2 359 296	36 864	110 592	1 - 48 Gbps

Table 3: Comparison of available AP boards [16]

3.3 Automata Network Markup Language (ANML)

Micron developed the Automata Network Markup Language (ANML) for designing automata. It is an XML-based language that is used to configure processing elements such as STEs, counter elements and boolean elements and connect them to form an automaton.

3.3.1 Translating NFAs to ANML-NFAs

Every NFA can be converted to an equivalent ANML-NFA for implementation on the AP using the following steps:

- For each transition function $T(q, a) = p$ of the NFA, create a STE with the set of accepted symbols a.
- For each state connect all STEs that represent ingoing edges of this state with all STE that represent outgoing edges of this state.
- Each STE that represents outgoing edges of the start state is marked as start STE.
- Each STE that represents ingoing edges of a final state is marked as reporting STE.

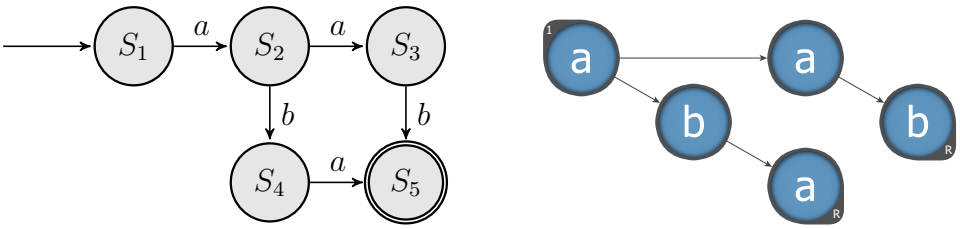


Figure 10: NFA and equivalent ANML-NFA

3.3.2 ϵ-transitions

The conversion of a NFA with ϵ-transitions requires some additional steps. After converting the NFA using the steps from the previous section, the ϵ-closure $\Gamma(u)$ of all states is determined. This information is required for the next 3 steps:

- Connect all STEs that represent an ingoing edge of state u with all STEs that represent outgoing edges of state v for each $v \in \Gamma(u)$.
- Mark all STE as start elements that represent outgoing edges of the state v for each $v \in \Gamma(u)$ where u is the start state.
- Mark all STE as reporting elements that represent ingoing edges of the state u if there is a $v \in \Gamma(u)$ that is an accepting state.

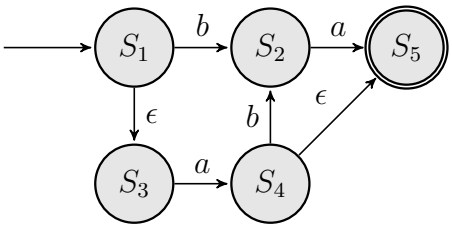


Figure 11: NFA with ϵ-transitions

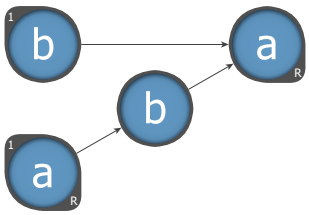


Figure 12: Equivalent ANML-NFA

3.3.3 Counter

The *counter element* is a special element that is used for counting events such as multiple occurences of a pattern in the input sequence. It does not directly match against symbols from the input sequence but instead is used in conjunction with STEs which are connected to the ports of the counter element.

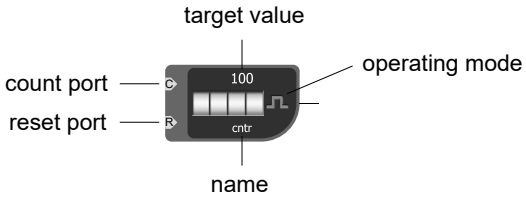


Figure 13: Counter

There exist two input ports, the *count* port and the *reset* port. Each time the count port is activated, the internal value of the counter is incremented. Each time the reset port is activated, the internal value is set to 0. The counter has one output port which is activated when the internal value reaches a specified target value $t \in [1, 2^{48}]$. There are 3 *operating modes* latch, pulse and roll. In *latch* mode the output port stays active once it has been activated until the counter is reset or the input stream ends. In *pulse* mode the output port is only active only during the clock cycle in which the target value was reached. The *roll* mode is the same as the pulse mode except that it also resets the counter.

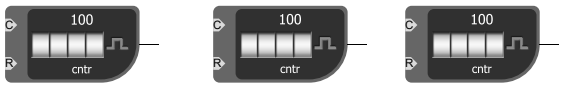


Figure 14: Operating modes *latch*, *pulse*, *roll*

A counter can be used for example in pattern matching of a fixed number of symbols. Figure 15 shows an automaton matching one a followed by 4 bs followed by one a.

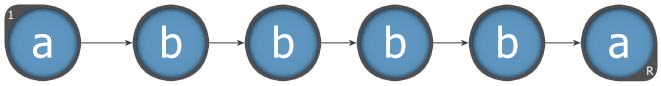


Figure 15: Automaton matching *abbbba*

By replacing the four STEs in the middle with a counter element (operating mode: *pulse*, target value: 4), the same functionality can be achieved.

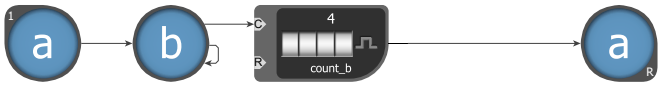


Figure 16: Automaton matching *abbbba* using counter

Another example of using the counter element is shown in figure 17. The automaton reports on every fourth b.

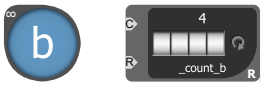


Figure 17: Automaton matching every fourth b using counter

The use of counters can strongly reduce the number of STEs needed for an automaton.

3.3.4 Boolean

The *boolean element* is a special element that is used for boolean logic. It does not directly match against symbols from the input sequence but instead is used in conjunction with STEs which are connected to the ports of the boolean element.

The following types of boolean elements are supported: [9]

- Inverter (single input terminal accepting a single activation signal)
- OR, AND, NAND, NOR (single input terminals accepting multiple activation signals)
- POS, SOP, NPOS, NSOP (multiple input terminals accepting multiple activation signals)

An inverter element inverts an activation signal. It inverts non-activation into activation even when the input STE is not testing input symbols against its symbol set.

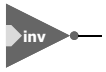


Figure 18: Inverter

The AND, OR, NAND, and NOR elements combine activation values and produce a high event when the boolean value computed by the element is equivalent to 1.

Figure 19: AND, OR, NAND, and NOR

A POS is the product (AND) of sum (OR) terms while a SOP is the sum (OR) of product (AND) terms. The elements NPOS and NSOP are based on POS and SOP but have their activation value inverted.

The elements can be represented as combinations of OR and AND gates as shown in figure 20.

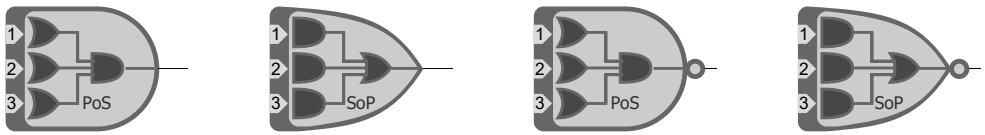


Figure 20: POS, SOP, NPOS, NSOP

3.3.5 Macro

A *macro* has a user defined number of input and output ports which are connected to an internal structure consisting of processing elements. As such it works as a container for processing elements and enables the hierarchical design of automata.

Inside a macro the symbol sets of STEs and target values of counter elements can be specified as parameters $\%P_i$. A default value for each parameter has to be defined. The macro can later be instantiated in multiple places in an automaton with different parameters.

3.4 Workflow

The development of automata for the AP is done in two phases. In the *design* phase the automata are defined, simulated and debugged on a CPU. Then they are compiled for the AP. In the *runtime* phase the automata are loaded into the AP. Afterwards the input sequence is streamed to the AP and all automata are executed in parallel. The output buffer of the AP can be read by the host application.

3.4.1 Design Phase

- Construction - The automaton is defined by writing ANML or regular expressions (PCRE).

- Simulation - The automaton can be simulated on a generic CPU. Debugging functionality such as stepwise forward and backward execution on the input sequence is provided. The automaton network is visualized as a graph and nodes are colored depending on their state. Report events are listed and their offset on the input sequence is shown.

- Compilation - The ANML or PCRE defintion of the automaton is compiled for the hardware resulting in a binary automaton file. During this process required processing elements are determined and mapped to a physical location on the AP hardware.

3.4.2 Runtime Phase

- Loading - The binary automaton file is programmed into the AP.

- Execution - The input sequence is streamed to the AP and the automaton is executed. The host application asynchronously reads output vectors from the output buffer of the AP.

3.4.3 Tools

The Software Development Kit (SDK) provides a visual interface called *AP Workbench* that can be used for creation and simulation of an automaton. All steps of the design and runtime phase can be controlled using the workbench. Furthermore the SDK contains following command line tools:

- AP Compile - Compilation of ANML file or regular expressions to binary automaton.
- AP Emulate - Simulation of an automaton on generic CPU. Provided with an input sequence and binary automaton this tool will output all report events generated during execution.
- AP Admin - Utility functions such as listing properties of a binary automaton or extracting a subgraph of an automaton.

3.5 API

Micron provides software developers with APIs that can be used as an alternative to the AP workbench and command line tools. The APIs encapsulate functionality of the design and runtime phase. Bindings for the programming languages C, python and Java are provided.

3.5.1 ANML API

The ANML API encapsulates functionality for the design phase. It can be used to define and compile automata.

For definition of an automaton a ANML object needs to be created using the function *AP_CreateANML()*. This object serves as a container object for one or more automata. The automata can either be loaded from a file with *AP_LoadAnmlObjects()* or new automata can be created with *AP_Create AutomataNetwork()*. Using the the function *AP_AddAnmlElement()* ANML elements such as STE, boolean and counter can be added to an automaton. The processing elements can be connected using the functions *AP_AddAnmlEdge()* and *AP_AddAnmlEdgeEx()*. Finally the automata can be compiled for execution in the simulator or the AP using the function *AP_CompileAnml()*. After compilation all internal memory used by the API can be freed by calling *AP_DestroyAnml()*.

3.5.2 Runtime API

The runtime API encapsulates functionality for the runtime phase. It can be used to configure the driver, load and unload automata and read output events.

Using the function *AP_Load()* an automaton can be loaded and relocated into the AP. From that point on the automaton is refered to as *runtime-object*.

The runtime API uses a software abstraction called *flow* for streaming input sequences to runtime objects. A flow is opened by calling the function

AP_OpenFlow() with a runtime object. Next, the flow is provided with an input sequence and streamed to the AP using the function *AP_ScanFlows()*. This starts the execution of the automaton on the input sequence. To retrieve output vectors from the output buffer the function *AP_GetMatches()* can then be used asynchonously. The function *AP_Wait()* enables blocking of a host application thread until results are available.

Every flow that is created has a data write overhead which becomes less significant with increasing data size. As such it is recommended to stream data sets of size 32 KB and above for maximum performance.

The API can also be used to change symbol sets of STEs and target values of counters using the functions *AP_SetSymbol()* and *AP_SetCounterTarget()* . The changes are applied at runtime by calling the function *AP_Reload()*. The recompilation of the automaton is not required.

3.6 Guidelines and Optimizations

This section describes some guidelines to optimize automata for the hardware layout of the AP.

3.6.1 Modulization

The AP enables encapsulation of processing elements into container elements called macros. This way reusable components can be created and compiled. These can afterwards be parametrized and combined to form large automata.

For example the k-clique macro in section 5 and the clique extension macros in section 6 are replicated in large quantities on the AP for the generation of automata.

3.6.2 Reprogramming

The compilation of an ANML automaton involves place-and-route algorithms that are computationally expensive. Depending on the complexity of the automaton, the compilation may take from a few milliseconds up to multiple hours. Therefore it is recommended to compile automata beforehand and save them as binary files. This way they can be loaded into the AP at runtime which takes 0.05 s.

The precompilation of automata can be performed for applications if the automaton design is independent of actual problem instances. For example there may exist a generic rule set for network intrusion detection that can be programmed into an automaton. By compiling this automaton once into a binary file, it can afterwards be rapidly distributed and loaded into APs in various datacenters quickly.

For automata that have a fixed structure and only vary slightly depending on problem instances, macros may be used. These contain placeholder values for the symbol sets of STEs and target values of counters that can be replaced at runtime. This way they can be adapted rapidly to the problem at hand.

3.6.3 Output Processing

The AP has six *output regions* that each have an *output event memory* which can store 1 024 *output vectors*. An output vector is generated each time one or more reporting STEs in an output region match the current input symbol. It is then stored in the output event memory in the same symbol cycle. However reading the output buffer by the host application requires multiple symbol cycles depending on the number of output regions in which events are generated and the output vector length.

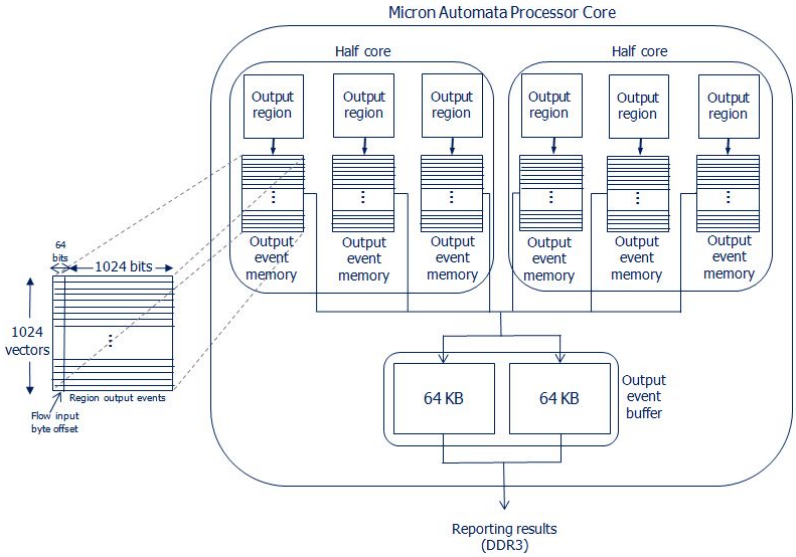


Figure 21: Micron Automata Processor Core [9]

The maximum output vector length is 1 024 bits. During compilation the place and route algorithms try to position the processing elements such that smaller output vectors can be used. Result of this process is a parameter called *event vector division* that defines the possible reduction of event vector length. The event vector division is the same for all regions and has possible values of: 1 (no reduction), 1.33, 2, 4, 8, and 16. Therefore output vector length is between 1 024 bits / 1 = 1 024 bits and 1 024 bits / 16 = 64 bits.

Event vector division	1	1.33	2	4	8	16
Output vector length	1 024	768	512	256	128	64
Transfer time t_{single}	40	30	20	10	5	2.5

Table 4: Event vector division, output vector length and transfer time

The output vectors have to be transfered from the output event memories to the output buffer of the AP to be read out by the host application. For transmission of event vectors of all regions a 15 symbol cycles overhead occurs. The transfer time of one output vector t_{single} depends on the output vector length as shown in table 4. Empy regions require 2 symbol cycles.

$$t_{out} = (15 + t_{single} \cdot r + (6 - r) \cdot 2) \cdot t_{clock}$$

Every symbol cycle in which output events are generated introduces above transfer time. To reduce the number of transfers needed, it is recommended to put as many output-generating events into the same symbol-cycle as possible.

3.6.4 Resource Usage

The hierarchical layers of an AP chip such as rows, blocks half-cores have different physical routing capabilities. While STEs in a row can be connected to each other, the connection of STE between rows is limited by the number of available block routing lines. The next higher layer places even more restrictions as STEs in one block can only be connected to STEs in neighbouring blocks. Connecting STEs in different half-cores is not possible.

The routing capabilities restrict the complexity of automata design for the AP. Many small automata that fit inside a single row and are not connected to each other can be implemented with high resource utilization. However as automata size and interconnectivity increases the resource utilization decreases. As such it is recommended to design automata with small densely connected subgraphs that are connected to each other by few connection lines.

4 Graph Theory and Clique Problem

4.1 Graph

A *graph* X consists of a *node* set $V(X)$ and an *edge* set $E(X)$, where an edge is an unordered pair of distinct nodes of X. We will usually use xy rather than $\{x, y\}$ to denote an edge. If xy is an edge, then we say that x and y are *adjacent* or that y is a *neighbour* of x, and denote this by writing x $\sim$ y. [4]

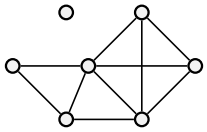


Figure 22: Graph

A subgraph of a graph X is a graph Y such that [4]

$$V(Y) \subseteq V(X), \quad E(Y) \subseteq E(X).$$

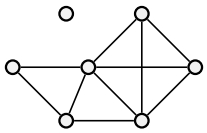


Figure 23: Graph X

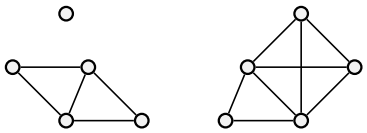


Figure 24: Some subgraphs of X

4.2 Graph Representation

The AP can only process strings over an alphabet size of 256, containing symbols 0 through 255. Therefore each node in an input graph is represented by a unique integer ID between 0 and 255. An edge of the graph is represented by the IDs of its source and destination nodes whereby the smaller ID is listed first.

The graph in Figure 25 has the nodes 0, 1, 2, 3, 4, 5 and the edges 0 1, 0 4, 1 2, 1 3, 1 4, 2 3, 2 4, 3 4, 4 5.

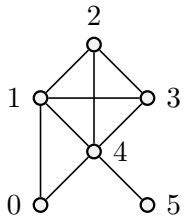


Figure 25: Graph with 6 nodes

4.3 Clique

A clique is a subgraph that is complete [4]. That means all nodes of a clique are connected to each other.

The figures 26-28 show all 2-cliques, 3-cliques and 4-cliques of a graph with 6 nodes.

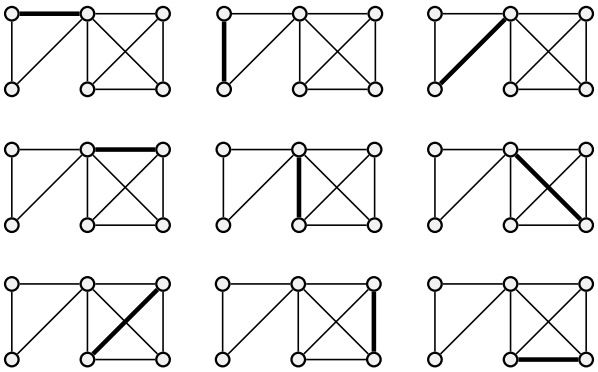


Figure 26: 2-cliques

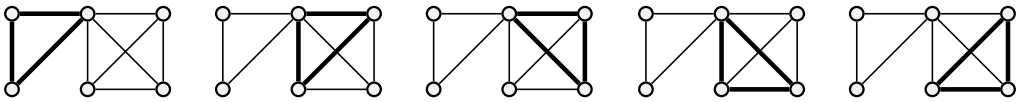


Figure 27: 3-cliques

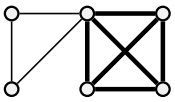


Figure 28: 4-cliques

4.4 Clique Problem

The *clique problem* is about finding subsets of nodes where each node is connected to each other. A such subset is also called complete subgraph.

There exist different formulations of the clique problem:

- Finding a maximum clique (a clique with the largest possible number of nodes)
- Finding a maximum weight clique
- Finding all maximal cliques (cliques that cannot be enlarged)
- The decision problem of testing whether a graph contains a clique larger than a given size

The clique problem is one of Karp's 21 NP-complete problems [6].

Solving the clique problem for different types of graphs was subject of the second DIMACS Implementation Challenge (1992-1993) [5]. The graphs used as benchmarks for the challenge are still available [8].

4.4.1 k-Clique

For *finding cliques of size* k in a graph the brute-force algorithm can be used. This algorithm enumerates all subgraphs of size k and checks for each one whether it is a clique. This is done by checking the subgraph for completeness.

The number of subgraphs of size k of a graph with n nodes is $\binom{n}{k}$. Therefore the enumeration of all subgraphs is only feasible for graphs with a few dozen nodes.

In section 5 an automaton implementing the brute-force algorithm is designed which has linear runtime for small graphs.

4.4.2 Maximum clique

The *maximum clique problem* is about finding a clique with the largest possible number of nodes. The size of the maximum clique is also refered to as *clique number* of a graph.

The Bron–Kerbosch algorithm [1] can be used to list all maximal cliques of an arbitrary graph in worst-case optimal time. By choosing the largest clique of this list, the maxmium clique can be found.

There exist also exist a number of heuristic algorithms for finding the maximum clique [19]. These algorithms are based on methods such as branch-and-bound [2], local search [7], and constraint programming [15]. Furthermore there has been research on novel techniques for solving the clique problem including DNA computing [12, 20] and adiabatic quantum computation [3].

In section 6 an automaton implementing a branch-and-bound algorithm is designed.

5 Brute-Force Solution

In this section we will introduce the *Brute-Force* algorithm for finding all k-cliques of a graph. Next, an automaton implementing the algorithm is described and adapted to the structure of the AP. Finally implementation is shown and resource usage and performance is analysed.

5.1 Algorithm

The brute force algorithm finds a k-clique in a graph with n nodes by systematically checking all $\binom{n}{k}$ subgraphs with k nodes for completeness.

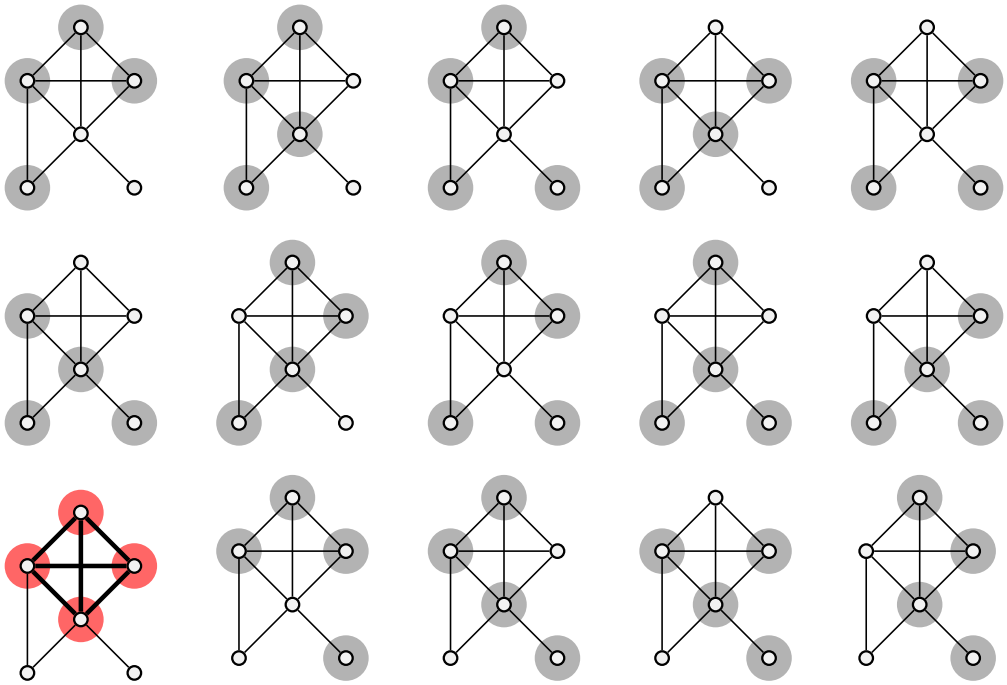


Figure 29: All $\binom{6}{4} = 15$ subgraphs of a graph

In figure 29 a 4-clique is found in a graph with 6 nodes by checking all $\binom{6}{4} = 15$ subgraphs for completeness.

5.2 k-Cliques Automaton

The k-cliques automaton for a graph with n nodes consists of $\binom{n}{k}$ k-clique macros each checking a possible subgraph for completeness. The input sequence streamed to the AP consists of all edges of the graph:

$e_{1,x}$	$e_{1,y}$	$\cdots$	$e_{n,x}$	$e_{n,y}$	FF

Table 5: Input sequence consisting of all edges of the graph

5.2.1 k-Clique Macro

The k-clique macro checks whether a graph contains all edges for a specific k-clique. It is parametrized with k nodes $\%P_0, \ldots, \%P_{k-1}$ and looks for occurences of all edges $\{\, \%P_i\%P_j \,|\, i < j \,\}$ in the input sequence.

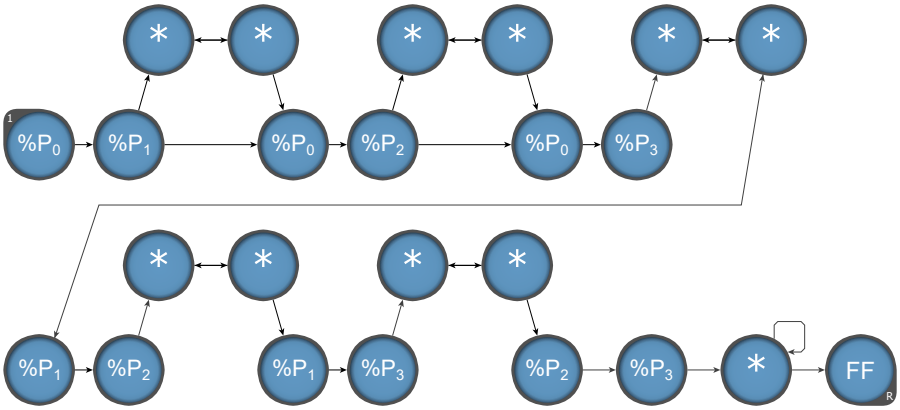


Figure 30: 4-Clique Macro

If all edges are found the macro reports a match at the end of the input sequence. The nodes forming the clique are encoded in the report code R of the macro. R is a bitstring of length 16 with positions $\%P_0, \ldots, \%P_{k-1}$ set to 1. For example the report code of a 4-clique macro for 0 2 3 6 has the report code $0000\,0000\,0100\,1101_2 = 77$.

A 4-clique automaton for a graph with 6 nodes contains $\binom{6}{4} = 15$ macros and as such is too big too be displayed in this thesis. However input sequence and reports for a 4-cliques automaton running on a sample graph (figure 31) are shown below.

Input	0	1	0	4	1	2	1	3	1	4	2	3	2	4	3	4	4	5	FF
Report																			30

Table 6: Input sequence and reports for 4-cliques automaton

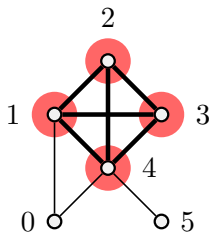


Figure 31: Graph with 6 nodes

The report code $30 = 0000\,0000\,0001\,1110_2$ denotes that the clique 1 2 3 4 has been found.

5.3 Resource Usage

The k-cliques automaton for a graph with n nodes consists of $\binom{n}{k}$ k-clique macros each checking for $\binom{k}{2}$ edges. Each edge check uses 4 STEs.

$$S_{macro} = \binom{k}{2} \cdot 4 \qquad S_{automaton} = \binom{n}{k} \cdot \binom{k}{2} \cdot 4$$

The maximum number of STEs on an AP chip is 59 152. The following table lists the maximum number of nodes n_{max} a graph may have such that a k-clique automaton does still fit on a chip.

k	n_{max}	$S_{automaton}$
3	30	48 720
4	16	43 680
5	13	51 480
6	11	27 720
7	11	27 720
8	11	18 480

Table 7: Graph size n_{max} for automaton finding k-cliques

As is evident from this table, the brute-force approach is severely limited by its combinatorial size requirements. Only graphs with a few dozen nodes can be processed.

5.4 Runtime

The compilation of the automaton takes a few minutes on an Intel Core i5 depending on clique size k and node count n. The automaton is independent of the graphs that it runs on and can be precompiled. As such the compilation time has not been considered in runtime calculation.

The runtime consists of loading the automaton, streaming all edges of the graph and reading output is

$$t_{run} = t_{load} + t_{edges} + t_{out}$$

The time for loading an arbitrary automaton into the AP $t_{load} = 0.05\,\mathrm{s}$.

Every streamed edge consists of two symbols and the end of the edge list is marked with the symbol FF. This yields a runtime $t_{edges} = (2\,e + 1) \cdot t_{clock}$.

The time for reading the output vectors from the output buffer has been described in a previous section:

$$t_{out} = (15 + t_{single} \cdot r + (6 - r) \cdot 2) \cdot t_{clock}$$

As the complete output of the automaton is generated on the last symbol cycle it is highly probable that one event vector in each of the 6 regions is created. Therefore the time for reading the output buffer is $t_{out} = (15 + 40 \cdot 6) \cdot t_{clock}$.

$$t_{run} = 0.05\,\mathrm{s} + (2\,e + 1) \cdot t_{clock} + 255 \cdot t_{clock}$$

As the runtime os only dependent on the number of edges, the worst runtime is achieved on fully connected graphs. The highest node count for which a k-clique automaton can be implemented is 30. A fully connected graph with 30 nodes has $\binom{30}{2} = 435$ edges. This results in a runtime

$$t_{max} = 0.05\,\mathrm{s} + (2 \cdot 435 + 1) \cdot t_{clock} + 255 \cdot t_{clock} = 0.050\,008\,4\,\mathrm{s}$$

6 Branch-and-Bound Solution

In this section we will introduce the recursive *Branch-and-Bound* algorithm for iteratively extending cliques to find the maximum clique of a graph. Next, an automaton for clique extension is described and adapted to the structure of the AP. Finally implementation is shown and resource usage and performance is analysed.

6.1 Algorithm

A simple exact algorithm for solving the maximum clique problem is shown in Algorithm 1. It operates by extending a known clique by one node each iteration.

```
1: function CLIQUE(set C, set P)
2:     if |C| > |C*| then
3:         C* ← C
4:     end if
5:     if |C| + |P| > |C*| then
6:         for all p ∈ P in predetermined order do
7:             P ← P \ {p}
8:             C' ← C ∪ {p}
9:             P' ← P ∩ N(p)
10:            CLIQUE(C', P')
11:        end for
12:    end if
13: end function
```

Algorithm 1: Finding the maximum clique C^* [2]

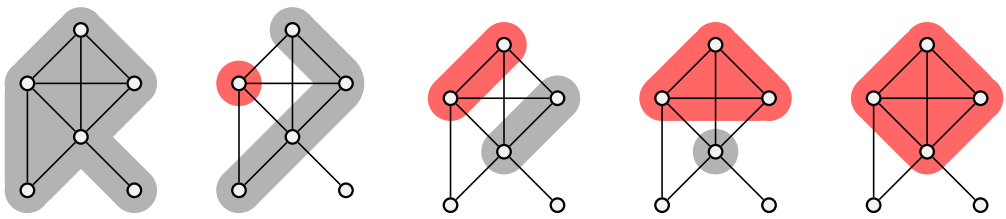
The first parameter of the *Clique(C,P)* function is the clique set C that contains all nodes of a clique that should be extended. The second argument is the candidate set P of nodes that are connected to each node in the clique and as such may extend the clique.

The algorithm recursively calls the function *Clique(C,P)* starting with an empty clique $C = \varnothing$ and $P = V$. To track the best solution found so far a global variable C^* is used (line 2-4).

The first component of the *Clique()* function is the bounding condition (line 5) which determines whether the clique C should be extended or not. As clique C can be extended by at most all nodes in P (if they are connected to each other) it can grow to a maximum size of $|C| + |P|$. Therefore further computation is only needed if this size is greater than the current best solution $|C^*|$.

Another important part is the branching procedure (line 6). It determines in which order the clique C is extended by the candidates $p \in P$. A simple branching strategy is ordering the vertices by ascending degree. But there also exist more sophisticated branching strategies using e.g. vertex coloring.

The remaining part of the function deals with the extension of a clique with a candidate $p \in P$ (lines 7-10). Therefore the candidate p is removed from the candidate set P. Then the new clique C' is defined by adding p to the current clique C. A new candidate set P' is defined containing all nodes of the old candidate set which are neighbours of p. Finally the function calls itself with the new arguments *Clique(C', P')*.



(a) 0-clique (b) 1-clique (c) 2-clique (d) 3-clique (e) 4-clique

Figure 32: Execution path resulting in a 4-clique (C: red, P: grey)

6.2 Clique Extension Automaton

The clique extension automaton contains one *clique extension macro* for each node x of the graph. The macro checks whether a clique that is streamed to the AP can be extended by its node. The input sequence streamed to the AP contain cliques c_x delimited by the symbol FF.

Input	FF	$c_{1,1}$	$\cdots$	$c_{1,m}$	FF	$c_{2,1}$	$\cdots$	$c_{2,n}$	FF	$\cdots$

Table 8: Input sequence

6.2.1 Clique Extension Macro

The *clique extension macro* for a node x is activated by the clique delimiter FF. Then it stays active as long as nodes are streamed which are neighbours of x. If all nodes of the clique are neighbours of x, the macro is still active when the next clique delimiter is streamed and thus generates a report event with the code x. If any of the nodes in the clique are not adjacent to x the macro gets inactive and does not report.

To avoid duplicate enumeration of cliques the macro only reports if all current nodes in the clique are smaller than x.

$$\%\text{P1} = \{n \in N(x) \mid n < x\}$$

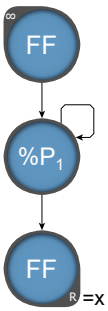


Figure 33: Clique extension macro for node x

An example for a clique extension automaton for a graph with 6 nodes can be seen in figure 34. It has 5 clique extension macros for the nodes 1-5. As the parameter %P1 for the node $x = 0$ equals the empty set it would never match and is excluded.

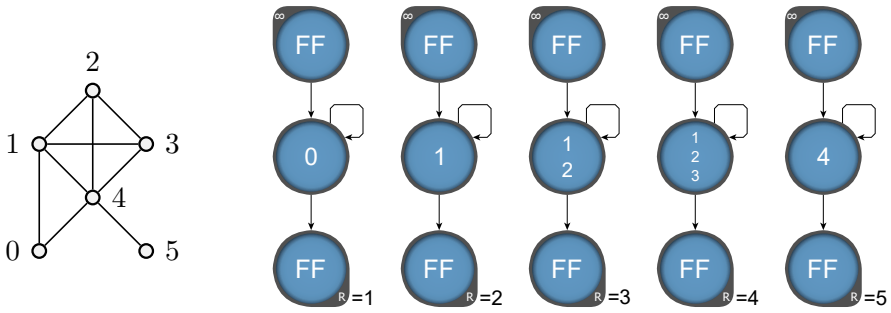


Figure 34: Graph and parametrized clique extension automaton

The following tables show the input sequence and reports for the automata and graph in figure 34.

Input	FF	0	FF	1	FF	2	FF	3	FF	4	FF	5	FF
Report			1, 4		2, 3, 4		3, 4		4		5		

Table 9: Input sequence for 1-cliques and report events

Input	FF	0	1	FF	0	4	FF	1	2	FF	1	3	FF	···
Report				4						3, 4			4	···

Table 10: Input sequence for 2-cliques and report events

6.2.2 Clique Extension Macro With Compression

The runtime of the AP is above all dependent on the length of the input sequence. In the clique extension automaton this sequence consists of the cliques that should be extended. As the size of the cliques grows, the input stream gets larger and performance decreases.

The input sequence can be compressed by merging common nodes of the cliques. For example two cliques c_1 and c_2 have 2 common nodes z_1, z_2.

c_1	1	2	3
c_2	1	2	4

z	1	2

Table 11: Common nodes z_1, z_2 of two cliques c_1 and c_2

The input sequence can now be started with the common nodes z followed by a new delimiter FE. Then remaining parts of c_1 and c_2 follow delimited by FD.

Input	FF	z_1	$\cdots$	z_k	FE	$c_{1,k+1}$	$\cdots$	$c_{1,n}$	FD	$c_{2,k+1}$	$\cdots$	$c_{2,m}$	FF	$\cdots$

Table 12: Input sequence with compression

For enabling compression three STEs are added to the clique extension macro as shown in figure 35.

After verifiying that all nodes z_i are neighbours of x the automaton enters a special state. From there it activates itself each time a FE or FD is streamed.

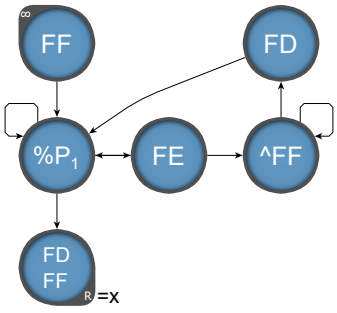


Figure 35: Clique extension macro for node x with compression

The following tables show the same cliques streaming without and with compression. It can be seen that compression is able to reduce the length of the input stream.

Input	FF	1	2	3	FF	1	2	4	FF	···
Report					4					···

Table 13: Input sequence and report events without compression

Input	FF	1	2	FE	3	FD	4	FF	···
Report						4			···

Table 14: Input sequence and report events with compression

In test series the compression was able to reduce the length of the input sequences to about half their size.

6.2.3 Clique Extension Macro With Output Aggregation

The macros described in the previous sections report after every streamed clique. As described in section 3.6.3 this frequent output generation can slow down execution of the automaton. The input sequence is therefore modified as follows. The symbol FC is added after each o-th clique. The macro is then modified to generate output only on this new symbol. A value of $o = 50$ has been determined in a series of tests as sufficient to reduce output generation to a reasonable amount.

Input	FF	···	FF	···	FC	FF	···	FF	···	FC	FF	···

Table 15: Input sequence for $o = 2$

For enabling output aggregation, multiple STEs are added to the clique extension macro depending on o as shown in figure 36.

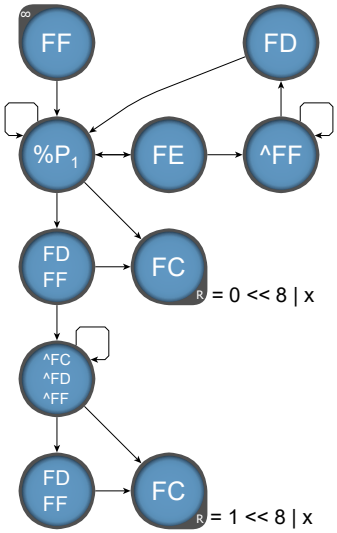


Figure 36: Clique extension macro for node x with output aggregation $o = 2$

The report code of the reporting elements is determined as follows. The node ID x occupies the lowest 8 bit. The offset o is shifted 8 bits to the left to occupy the next higher 8 bits of the report code $R_o = o << 8 \,|\, x$.

The following tables show the same cliques streaming without and with output aggregation.

Input	FF	0	1	FF	0	4	FF	1	2	FF	$\cdots$
Report				4						3, 4	$\cdots$

Table 16: Input sequence and report events without output aggregation

Input	FF	0	1	FF	0	4	FF	1	2	FC	FF	$\cdots$
Report										516, 3, 4		$\cdots$

Table 17: Input sequence and report events for $o = 3$

The report codes 3 and 4 have an offset $o = 0$ which means that they belong to the last streamed clique 1 2 before the output delimiter FC. The report code $516 = 2 << 8 \,|\, 4$ has an offset $o = 2$ and as such belongs to the clique 0 1.

6.3 Resource Usage

The size of the clique extension macro with output aggregation depends on the number of output levels o. The smallest macro with one output level has 7 STEs. Every additional output level increases the STE count by 3.

$$S_{macro} = 7 + 3 \cdot (o - 1)$$

A clique extension automaton has one macro for every node in a graph.

$$S_{automaton} = n \cdot (7 + 3 \cdot (o - 1))$$

The AP can process strings over an alphabet of size 256. The symbols FF, FE, FD, FC are used for delimiting cliques, compression and output aggregation. The remaining $256 - 4 = 252$ symbols can be used for representing nodes. As such the maximum automata size is

$$S_{automaton-max} = 252 \cdot (7 + 3 \cdot (50 - 1)) = 38\,962$$

This automaton uses $38\,808/49\,152 = 79\,\%$ of the available STEs on an AP chip.

6.4 Runtime

The compilation of the automaton takes a few minutes on an Intel Core i5 depending on node count n and output aggregation o. The automaton structure is independent of the graph that it runs on and can be precompiled. As such the compilation time has not been considered in runtime calculation.

The runtime consists of loading and parametrizing the automaton, streaming all cliques and delimiters and reading output:

$$t_{run} = t_{load} + t_{cliques} + t_{FC} + t_{FF} + t_{out}$$

The time for loading an arbitrary automata into the AP and parametrizing the macros is $t_{load} = 0.05\,\mathrm{s}$.

Every clique that should be extended is streamed to the AP. Therefore amount of symbols to be streamed is the sum of the length of all cliques.

$$t_{cliques} = (|c_1| + \cdots + |c_m|) \cdot t_{clock}$$

Every o-th clique the symbol FC is streamed that notifies the automaton to generate output.

$$t_{FC} = \frac{m}{o} \cdot t_{clock}$$

Before every clique a delimiter FF is streamed. After last clique also one FF is streamed.

$$t_{FF} = (m + 1) \cdot t_{clock}$$

The automaton is designed such that output generation during streaming does not affect performance. Therefore only the time for reading the output vectors that are generated on the last symbol cycle is important:

$$t_{out} = 255 \cdot t_{clock}$$

The final formula for the determining the runtime is:

$$t_{run} = 0.05\,\mathrm{s} + ((|c_1| + \cdots + |c_m|) + \tfrac{m}{o} + m + 1) \cdot t_{clock} + 255 \cdot t_{clock}$$

For using the formula the lengths of the cliques $c_1, \ldots, c_m$ have to be known. An algorithm for determining this information is described in the next section.

6.5 Host Application

The following algorithm describes a host application for the automaton.

```
1: function Main(Graph, L)
2:     File ← CompileAutomaton(Graph)
3:     Cliques ← GetNodes(Graph)
4:     while Cliques != ∅ do
5:         InputSequence ← GatherCliques(Cliques, L)
6:         ReportEvents ← ApEmulate(File, InputSequence)
7:         ProcessEvents(ReportEvents)
8:     end while
9: end function
```

Algorithm 2: Host application for the clique extension automaton

The first parameter is a data structure containing the nodes and edges of the graph. The second parameter is the length L of the input sequences that are streamed to the AP.

The program creates a clique extension automaton with output aggregation $o = 50$ for the graph. This automaton is compiled into a binary file which can later be used for execution. Next, the cliques list is filled with all nodes of the graph as 1-cliques. The program then enters a loop:

The cliques list is filtered by the bounding condition and sorted by the branching rule. Then cliques are taken from the list and assembled into an input sequence until the sequence reaches the length L.

Next, the command line tool AP Emulate from the Automata SDK is called with the binary automaton file and the input sequence. The tool emulates the automaton and returns a list of all report events.

By processing the report events the program gets information about each extended clique and candidates for its further extension. This information is added to the cliques list. If a new best solution has been found a global variable is updated.

The program terminates when no cliques are left to be extended.

6.6 Performance

The performance of the clique extension automaton is determined by running algorithm 2 from the previous section. The lengths $|c_1| + \cdots + |c_m|$ of the enumerated cliques are then used to calculate the runtime t_{run} of the automaton (see section 6.4). Afterwards the host application overhead t_{host} is calculated. It consists of the time needed for composition of the input sequences (line 5) and processing of report events (line 7). Finally the complete execution time is calculated:

$$t_{execution} = t_{run} + t_{host}$$

Using this method the execution time of a clique extension automaton with output aggregation $o = 50$ has been determined for test graphs with up to $n = 70$ nodes and input sequence lengths $L = 1\,\text{KB}$, $16\,\text{KB}$ and $32\,\text{KB}$. The results are shown in columns 1-3 in table 18 splitted in $t_{run} + t_{host}$. In column 4 the execution time of algorithm 1 on a CPU is shown for comparison. For empty cells execution times have not been determined yet. The input sequence was not compressed because that would only lower t_{run} slightly while strongly increasing t_{host}.

	AP + Host App. Overhead			CPU
	$L = 1\,\text{KB}$	$L = 16\,\text{KB}$	$L = 32\,\text{KB}$	
$n = 40$	0.052 + 0.300	0.054 + 0.400	0.056 + 0.600	0.180
$n = 50$	0.072 + 2.030	0.076 + 2.470	0.081 + 2.880	0.770
$n = 60$	0.219 + 17.250	0.223 + 17.080	0.228 + 18.070	5.370
$n = 70$	0.610 + 54.940			19.140

Table 18: Execution time in s

Both platforms use algorithms based on the branch-and-bound design paradigm with the same branching and bounding rules. Also for both benchmarks the same system (Intel Core i5 760, 2.80 GHz, 16 GB RAM) was used.

The test graphs were generated using *ggen* [10], a program that was used to generate graphs of the c family for the Second DIMACS Implementation Challenge (1992-1993) [8]. The following parameters were used:

Seed	74 328 432
Number of vertices	n
Max number of edges	10 000
Edge probability	0.90
Data structure	dense

Table 19: Parameters for generating test graphs with *ggen* [10]

This resulted in the following test graphs:

n	edge count	edge density	max degree	avg degree	min degree
40	703	0.90	38	35.15	32
50	1 114	0.90	49	44.56	41
60	1 599	0.90	58	53.30	49
70	2 173	0.90	66	62.09	56

Table 20: Statistics for test graphs

7 Source Code and External Code

The source code is available at the following repository:
`https://github.com/sedk1661/graph-algorithms-micron-ap`.

The following external packages were used in this project:

- combinatoricslib [13] for enumerating combinations $\binom{n}{k}$
- pengyifan-commons [14] for managing cliques using an efficient tree data structure

8 Future Work

8.1 Brute-Force Solution

The k-clique automaton is severely limited by its size requirements. Even for small values of clique size k and graph node count n thousands of k-clique macros are needed to check for all possible cliques. As many cliques share common nodes and thus check for existence of the same edges, there will exist some duplication in the final automata.

For example the 3-clique macros for 0 1 2 and 0 1 3 both start by checking for the edge 0 1. By merging this part of the macros, duplicate processing elements can be removed.

8.2 Branch-and-Bound Solution

The performance of the clique extension automata is greatly affected by the performance of the host processor. Currently the construction of the input sequence and the processing of report events by the CPU take much more time than the execution of the automaton on the AP. The algorithms used for these processes need to be improved. Furthermore mutlithreading capabilities of the CPU should be used.

The clique extension automata only implements a part of the Branch-and-Bound algorithm namely the extension of a k-clique to a $(k + 1)$-clique. The branching and bounding rules however are executed on the CPU. Therefore the overall performance of the clique extension automata will improve by advanced branching and bounding methods which prune the search space effectively.

There also exists the idea of creating one clique extension macro for each edge in the input graph. Then the macro would check whether each streamed clique can be extended by both nodes of the edge. This would result in extending cliques twice as fast, but also increase automaton size requirements.

9 Conclusion

In this thesis the Micron Automata Processor has been introduced as a promising tool for solving computationally expensive problems. Then two algorithms to solve the clique problem were described and implemented as automata.

The brute-force algorithm has been shown to be only suitable for very small graphs up to 30 nodes due to its fast growing size complexity. The branch-and-bound algorithm is theoretically able to process graphs with up to 252 nodes. However it is currently limited by the host application performance.

As described in the previous section the automata and host application can still be improved in various ways.

Bibliography

[1] Bron, C. and Kerbosch, J. (1973). Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577.

[2] Carraghan, R. and Pardalos, P. M. (1990). An exact algorithm for the maximum clique problem. *Oper. Res. Lett.*, 9(6):375–382.

[3] Childs, A. M., Farhi, E., Goldstone, J., and Gutmann, S. (2002). Finding cliques by quantum adiabatic evolution. *Quantum Info. Comput.*, 2(3):181–191.

[4] Godsil, C. and Royle, G. (2001). *Algebraic Graph Theory*. Springer.

[5] Johnson, D. and Trick, M. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*. American Mathematical Society.

[6] Karp, R. (1972). Reducibility among combinatorial problems. In Miller, R. and Thatcher, J., editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press.

[7] Katayama, K., Hamamoto, A., and Narihisa, H. (2005). An effective local search for the maximum clique problem. *Information Processing Letters*, 95(5):503–511.

[8] Mascia, F. Dimacs benchmark set. `http://iridia.ulb.ac.be/~fmascia/maximum_clique/DIMACS-benchmark`.

[9] Micron Technology, Inc. Micron automata processing. `http://www.micronautomata.com/`.

[10] Morgenstern, C. ggen. `http://iridia.ulb.ac.be/~fmascia/files/ggen.tar.bz2`.

[11] Noyes, H. (2014). Micron's automata processor architecture: Reconfigurable and massively parallel automata processing.

[12] Ouyang, Q., Kaplan, P. D., Liu, S., and Libchaber, A. (1997). Dna solution of the maximal clique problem. *Science*, 278(5337):446–449.

[13] Paukov, D. combinatoricslib. `https://github.com/dpaukov/combinatoricslib`.

[14] Peng, Y. pengyifan-commons. `https://github.com/yfpeng/pengyifan-commons`.

[15] Régin, J.-C. (2003). *Using Constraint Programming to Solve the Maximum Clique Problem*, pages 634–648. Springer Berlin Heidelberg, Berlin, Heidelberg.

[16] Roy, I. (2015). Algorithmic techniques for the micron automata processor.

[17] Sabotta, C. R. (2013). Advantages and challenges of programming the micron automata processor. *Graduate Theses and Dissertations*.

[18] Watson, B. W. (1995). *A taxonomy of finite automata construction algorithms*.

[19] Wu, Q. and Hao, J.-K. (2015). A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693 – 709.

[20] Zimmermann, K., Martinez-Perez, I., Ignatova, Z., and Gong, Z. (2006). Solving the maximum clique problem via dna hairpin formation.