POLITECNICO di MILANO

Algorithms on Strings in OpenCL for Many-Core Platforms

A.A. 2011/2012

Author:
Alberto De Natale
Matr. 739859

Supervisor:
Dr. Giovanni Agosta

Contents

List of Figures **2**

1 Background **3**
1.1 Definitions . . . 3
1.2 The Problem Definition [2] . . . 5
1.2.1 The Dictionary Automaton . . . 5
1.2.2 Build the Trie Automaton . . . 5
1.2.3 The Dictionary Automaton . . . 6
1.2.4 Building the Dictionary Automaton . . . 7

2 Hardware and Software Platform [3] **11**
2.1 P2012 . . . 11
2.1.1 Introduction . . . 11
2.1.2 Fabric Level . . . 12
2.1.3 The multi-core computing cluster . . . 13
2.2 Programming on P2012 . . . 16
2.2.1 OpenCL . . . 16

3 Algorithms on String **19**
3.1 The framework . . . 19
3.2 The Algorithm Parallelized . . . 20
3.2.1 The initialization of the memory on the stack . . . 21
3.2.2 The first loop . . . 21
3.2.3 The second loop . . . 21

4 Results **23**
4.1 The DNA Model . . . 23
4.1.1 The DNA Structure . . . 23
4.1.2 The Model [6] . . . 23

5 References **27**

A Code **28**
A.1 Automata_Utilities.h . . . 28
A.2 Automata_Utilities.c . . . 28
A.3 Automata_Functions.c . . . 31

List of Figures

1 The DNA model 24
2 IUPAC Codes 24
3 Results with 300 nodes 25
4 Results with 3000 nodes 26
5 Results with 6000 nodes 26
6 Results with 9000 nodes 26

1 Background

1.1 Definitions

A *graph* is defined to be a couple (N, E) where N is a set of elements called nodes and E is a set of elements called arcs.
A *weighted graph* is a graph where for every arc $e \in E$ exists a real number c called *cost* $\in \mathbb{R}$.
A node n_1 is said to be a *successor* of a node n_0 if exists an arc e_{01}, from n_0 to n_1 in E.
A *search problem* [1] on a graph can be formalized with the following elements:

- the initial node $n_0 \in N$
- the successor function $f : N \times E \to N^n$ that associates to a node the set of nodes reachable
- the goal function $g : N \to \{0, 1\}$ indicates if a node is a goal node.
- the path cost function $h : N \times N \to \mathbb{R}$
- the solution, a path from n_0 to a node in $\{n \in N \mid g(n) = 1\}$

A *search tree* is a data structure acted to represent data contained in a graph during a search. Its elements are said *nodes*. Every node in a search tree corresponds to a node in the graph.
A node is *expanded* when all successors in the graph are *generated*
The *node* of a search tree is a data structure containing:

- the reference to the state in the graph at which the node corresponds
- the node that generated this node

For what concern the analysis of sequences of strings it's common adopt some basic notions that will now be introduced.
An *alphabet* [2]{a, b, ...} is a finite nonempty set whose elements are called letters.
A *string* or a *word* on an alphabet A is a character sequence.
The *empty string* is a string of lenth zero and is indicated as ϵ.
A *language* L is a set of strings.
The set of all string on an alphabet A can be denoted by A^*, using the *kleene operator* * so defined:

$$A_0 = \epsilon$$

$$A_{i+1} = \{wv \mid w \in A_i \wedge v \in A\}$$
$$A^* = \bigcup_{i \in \mathbb{N}} A_i$$

The *kleen plus operator* + is defined as:

$$A^+ = \bigcup_{i \in \mathbb{N} \setminus \{0\}} A_i$$

Any subset of A^* is a *language* on the alphabet A.
The *length of a string* x [2] is defined as the length of the sequence associated with the string x and it's denoted as $|x|$.
Two strings are equals if and only if:

$$|x| = |y| \wedge x[i] = y[i]$$

for $i \in \{0, 1, ..., |x| - 1\}$.
A string x is a *factor of* y [2] if there exists two strings u and v such that $y = uxv$. When $u = \epsilon$ x is a *prefix of* y and when $v = \epsilon$ x is *a suffix of* y.
The sets of prefixes, suffixes of the strings of a language X are in turn languages and will be denoted respectively as Pref(X), Suff(X).

A *finite-state machine (FSM)* [4]or *finite-state automaton* (plural: *automata*), is a mathematical model used to design computer programs and digital logic circuits. It is conceived as an abstract machine that can be in one of a finite number of states. The machine can be only one state at a time called the *current state*. It can change from one state to another when initiated by a triggering event or condition, this is called a transition. A particular FSM is defined by a list of the possible transition states from each current state, and the triggering condition for each transition.
A deterministic finite state machine or acceptor deterministic finite state machine is a quintuple , where:

- A is the input alphabet.
- Q is a finite, non-empty set of states.
- q_0 is the initial state, an element of Q.
- δ is the state-transition function:$\delta : Q \times A \to Q$, eventually partial and implementable as a set $F \subseteq Q \times A \times Q$.

- T is the set of final states, a subset of Q.

For the automata used here, every state belonging to Q will be rappresentative of a string $s \in A^*$.

1.2 The Problem Definition [2]

1.2.1 The Dictionary Automaton

A *dictionary* $X \subseteq A^*$*on* A is finite nonempty language not containing the empty string ϵ.
The problem here addressed concerns with the search, on a given text y, of a sequence of words that can be modelled throught the concept of dictionary. The automaton that recognize a dictionary is composed as follow:

- The set of states is Pref(X).
- The initial state corresponds to the empty string ϵ.
- The set of terminal states is the corresponding to a string belonging to X.
- The connections are of the form (u, a, ua), where the successor states are obtained appending a character $a \in A$ to a string $u \in A^*$.

The automaton so defined is said $T(X)$, is deterministic and recognizes X. $T(X)$ is said the *trie* of the dictionary X.

The problem introducted at the beginning of the section, can be so redefined as the problem of recognizing the language A^*X in a *text*, modelled with a string $y \in A^*$.

1.2.2 Build the Trie Automaton

The algorithm that allow to build the automaton is the following:

```
1 M ← newAutomaton()
2 foreach x ∈ X do
3         s1 ← initial[M]
4         foreach a ∈ x do
5                 s2 ← target(s1, a)
6                 if s2=nil then
7                    s2←newState()
8                    M ← M∪ nextState
9                 s1 ← s2
10        terminal(s2) ←TRUE
11 return M
```

The algorithm begins its computation creating a new empty automaton (line 1). From the initial state ϵ, loops around every word in the dictionary (line2). For every character in the word, queries with the function target() if already exists a state that models the string builded (line 5). If it doesn't exist, the function creates a new state and inserts it into the automata (lines 7,8). Then the algorithm restarts looping from that state (line 9) until it doesn't reach the end of the dictionary. Every state ending a word, is marked as terminal (line 10).

1.2.3 The Dictionary Automaton

The automaton capable to recognize A^*X will now be introduced.
The function h:

$$h : A^* \to Pref(X)$$

where $h(u)$ is the longest suffix of u that belongs to Pref(X). For every $(u, a) \in A^* \times A$:

$$h(ua) = \begin{cases} ua & \text{if } ua \in Pref(X) \\ h(f(u)a) & \text{if } u \neq \epsilon \wedge ua \notin Pref(X) \\ \epsilon & \text{otherwise} \end{cases}$$

The function f:

$$f : A^* \to Pref(X)$$

where $f(u)$ is the longest *proper* suffix of u that belongs $Pref(X)$. For the same input string u the same dictionary X, the function f doesn't map to the same values for those strings $u \in Pref(X)$.
For every $(u, a) \in A^* \times A$:

$$f(ua) = \begin{cases} h(f(u)a) & \text{if } u \neq \epsilon \\ \epsilon & \text{otherwise} \end{cases}$$

Follows that $\forall u \in A^*$:

$$u \in A^*X \Leftrightarrow \ u \in X \vee u \neq \epsilon \wedge f(u) \in A^*X$$

Let $D(x)$ be the automaton whose:

- set of states is $Pref(X)$
- initial state is the empty string ϵ
- set of terminal states is $Pref(X) \cap A^*X$
- arcs are of the form $(u, a, h(ua))$

The automaton recognizes A^*X. $D(X)$ is called the dictionary automaton.

1.2.4 Building the Dictionary Automaton

The problem of building the dictionary automaton can be well reducted to a search problem on a graph $G(S, E)$, where every node of the graph represents a string. It can be formalized with these basic elements:

- the initial state is the one corresponding to an empty string ϵ
- the successor function is implemented throught the trie automaton, the action set T is the alphabet set A
- the goal states are the ones corresponding to

$$u \in X \vee u \neq \epsilon \wedge f(u) \in A^*X$$

- the path cost function costant for every connection in the graph
- the solution is a collection of paths

A general search algorithm could be:

```
1 F ← emptyQueue()
2 enqueue(F, s_0)
3 while not isEmpty(F) do
4         p ← removeFront(F)
5         processState(p)
6         expandState(F, p)
7 return
```

The expandState() function is:

```
1 expandState(F, p)
2 foreach e in E do
3         q ← f(s, t)
4         if (q! = nil) then
5           enqueue(F, q)
6 return
```

The expandState() function is responsible for calculating the set of nodes reachable from the node passed as input. The collection of nodes waiting to be expanded is called $fridge$ or $frontier$. It is here organized as a *queue* on which the following operations are valid:

- emptyQueue(*queue*) creates an empty queue
- isEmpty(*queue*) returns true if there are no more elements in the queue
- removeFront(*queue*) removes the element at the front of the queue
- enqueue(*queue*, *states*) inserts a set of states into the queue

There are six kinds of search implementable using this basic algorithm structure and they all differs by the *order* in which states into the queue are visited:

- breadth-first search
- uniform cost search
- depth-first search
- depth-limited search
- iterative deepening search

The breadth-first is a searching strategy in which the root node is expanded first, then all the nodes generated by the root are expanded next, and their successors, and so on. All the nodes at depth d in the search trie are expanded before the nodes at depth $d + 1$. Breadth-first search algorithm can be implemented with a queuing function that puts newly generated states at the end of the queue, after the previously generated.

The dictionary automaton can be implemented substituting the search tree with the $Trie$ automaton:

buildDictionary()
1 $M \leftarrow$ Trie(X)
2 $q_0 \leftarrow initial[M]$
3 $F \leftarrow$ Empty-Queue()
4 enqueue(F,(q_0))
5 *while* not isEmpty(F) *do*
6 $p \leftarrow$ removeFront(F)
7 processNode(p)
8 expandNode(F, p)
9 *return* M

expandNode(F, p)
1 *foreach* $a \in A$ *do*
2 $q \leftarrow target(p, a)$
3 *if* $(q! = nil)$ *then*
4 enqueue(F, q)
5 *return*

The algorithm simply navigates the $Trie$ and recognizes every string $u \in$ Pref(X). Now the automata has to recognize those $u \neq \epsilon \wedge f(u) \in A^*X$. For doing this, the queue will not just take memory of the nodes expanded but about the couple $(u, f(u))$. The final algorithm becomes:

buildDictionary()
1 $M \leftarrow$ Trie(X)
2 $q_0 \leftarrow initial[M]$
3 $F \leftarrow$ Empty-Queue()
4 *for* $a \in A$ *do*
5 q $\leftarrow$ target(q_0, a)
6 *if* $(q = nil)$ *then*
7 $succ(q_0) \leftarrow succ(q_0) \cup (a, q_0)$
8 *else*
9 enqueue$\{(F, (q, q_0))\}$
8 *while* not empty-Queue(F) *do*
9 $(p, r) \leftarrow$ removeFront(F)
10 processNode(p, r)
11 expandNode($F, (p, r)$)
12 *return* M

In lines 4-7 an initialization passage was added, adding a connection to the initial state for those letters for which a next state wasn't defined and populating the queue with the first level of the Trie. The queue is populated of couples (q,s).

```
expandNode(F, p)
1 foreach a ∈ A do
2         q ← target(p, a)
3         s ← target(r, a)
4         if (q! = nil) then
5            enqueue(F, q)
6         else succ(p) ← succ(p) ∪ {(a, s)}
7 return
```

At line 3 has been added the code to memorize the state corresponding to $f(u)$. If there's not a connection outing from p means that the input string doesn't belong to X but possibly to A^*X. So the computation restarts from $f(u)$ and a connection to the state corresponding to it is added (line 6).

```
processNode(F, p)
1 if (termina(r)) then
2    terminal(p) ← TRUE
3 return
```

if a string belongs to X means that it belongs to A^*X too. So if $f(u)$ arrives in a terminal state also the string u must be in a terminal state.

2 Hardware and Software Platform [3]

2.1 P2012

2.1.1 Introduction

The transition to multi-core is almost a forced choice to escape the silicon efficiency crisis caused by the looming power wall. While multi-core architectures are common in general-purpose and domain-specific computing, there is no one-size-fits-all solution.

General purpose multi-cores are still designed to deliver outstanding single-thread performance under very general conditions in terms of workload mix, memory footprint, run-time environment, legacy code compatibility. The landscape is quite different when focusing on domain-specific acceleration, which is driven by the growing demand for high-end value-added functionality. These applications are extremely demanding from a computational viewpoint: multi-GOPS performance requirements are not uncommon. In this context, computing engines have traditionally been implemented as hardwired functional units, but this scenario is changing under the pressure for increased flexibility.
Today, we see a trend towards many-core fabrics, with a throughput-oriented memory hierarchy featuring software-controlled local memories, FIFOs and specialized DMA engines. As a result, a SoC platform today is a highly heterogeneous system. It integrates a general-purpose multi-core CPU, and a number of domain-specific many-core subsystems.

P2012 is an area and power-efficient many-core computing fabric, and it provides an architectural harness that eases integration of hardwired accelerators. The P2012 computing fabric is highly modular, as it is based on multiple clusters implemented with independent power and clock domains, enabling aggressive fine-grained power, reliability and variability management. Clusters are connected via a high-performance fully-asynchronous network-on-chip (NoC), which provides scalable bandwidth, power efficiency and robust communication across different power and clock domains. Each cluster features up to 16 tightly-coupled processors sharing multi-banked level-1 instruction and data memories, a multi-channel advanced DMA engine, and specialized hardware for synchronization and scheduling acceleration. P2012 achieves extreme area and energy efficiency by aggressive exploitation of domain-specific acceleration at the processor and cluster level. Each processor can be specialized at design time with modular extensions (vector units, floating-point unit, special-purpose instructions). Clusters can easily become heterogeneous computing engines thanks to the integration

of coarse-grained hardware processing elements that implement those functions for which a software implementation would be inefficient (e.g. bitwise stream manipulations). Hardware-software interaction is facilitated by the local and global interconnect which efficiently supports point-to-point stream communication.

The ultimate goal of P2012 is to fill the area and power efficiency gap between general-purpose embedded CPUs and fully hardwired application accelerators. This ambitious goal is achieved through a mix of three key elements:

- modular hardware building blocks (cluster tiles)
- support for hardware specialization at multiple levels of granularity
- a programming and platform customization framework which supports migration of software functions into hardwired functional units without requiring extensive application-level rework. In the next sections we will provide details on the P2012 hardware and software architecture, with an emphasis on these three major points.

2.1.2 Fabric Level

Fabric-level communication is based on an asynchronous Network-on-Chip (NoC) organized in a 2D mesh structure. The routers of this NoC are implemented in a Quasi-Delay-Insensitive (QDI) asynchronous (clock-less) logic. They provide a natural Globally Asynchronous Locally Synchronous (GALS) scheme by isolating the clusters logically and electrically. Asynchronous-to-Synchronous interfaces based on FIFOs connect the NoC to the different clusters. Following the GALS interface, a Network Interface (NI) is used as a logical link between a cluster and the NoC. It is used for encapsulating the address-based protocol of the cluster into a packet-based NoC-compatible protocol. It also gives access to the main clock, variability and power (CVP) controller that is used to control a power management harness.

The clocking scheme is based on an innovative Frequency Locked Loop (FLL) unit which provides frequencies in the range of 500 Mhz to 1.5 Ghz with less than 50 ps jitter and can be reprogrammed in less than 200ns. This very small clock generator can be duplicated for the different blocks of the cluster, enabling a dynamic fine-grain frequency adaptation capability.

One important characteristic of the fabric is that all local storage at the cluster level is visible in a global memory map, which also includes memory-mapped peripherals. The global, aliasing-free visibility of local resources is

a key design choice as it facilitates code development, porting and maintenance. However, in this non-uniform memory architecture (NUMA), remote memories (off- cluster or off-fabric) are expensive to access. For this reason, DMA engines are available for hardware- accelerated global memory transfers. At the fabric level, a configurable number of I/O channels, implemented via multiple DMAs, can be used for connecting the fabric to the rest of the SoC. Finally, a fabric controller serves as the control interface between the SoC and the fabric. Its role is to isolate the fabric world from the SoC world, and controlling the boot and application mapping sequences, as well as the general power management scheme. The modularity of the fabric is given by the NoC and the power and clock domain isolation among different tiles. Tiles need not to be homogeneous: as we will see in the next section, each cluster can be populated with a different number of Hardware Processing Elements (HWPEs) and a different number of tightly coupled processors (the Encore16 block in the figure represents a maximal configuration with 16 processors). The back-end of the physical design process is facilitated by the pure GALS strategy, enabling good performance scalability. The width of the data paths between the routers can be adapted to the required application throughput and to the number of clusters. Finally, the distributed power scheme implemented emphasizes scalability and modularity by avoiding centralized control bottlenecks.

2.1.3 The multi-core computing cluster

The P2012 cluster aggregates a multi-core computing engine, called ENCore and a cluster controller. The ENCore cluster includes a number of processing elements (PEs) varying from 1 to 16 (the maximal configuration). Each PE is built with a highly configurable and extensible processor called the STxP70-V4. It is a cost effective and customizable 32-bit RISC core supported by a comprehensive state-of- the-art development toolset.

The STxP70-V4 architecture is a 32-bit load/store architecture with a variable- length instruction-set encoding (16, 32 or 48-bit), allowing manipulation of 32-bit, 16-bit or 8-bit data words and supporting predicated execution to minimize the branch penalty effect. The STxP70-V4 core is implemented with a 7-stage pipeline, enabling a processor clock frequency exceeding 600 MHz (in a low- power 32nm CMOS process) and it can execute up to two instructions per clock cycle (dual issue). The P2012 baseline STxP70 core is configured with a 32-entry register file, two zero-cycle overhead hardware loops, a 32-bit hardware multiplier, a second 32-bit core ALU and an interrupt controller with 16 maskable interrupt lines and 1 non-maskable interrupt.

The STxP70-4 architecture offers a simple way to extend the basic STxP70 instruction set with application- specific instructions. In the current embodiment of P2012, each STxP70 PE includes up to two extensions suitable for video and image processing. The first one, called VECx, is a SIMD vectorial extension operating on 128-bit vectors of fixed point data types. It can be used to perform 8-way operations on 16-bit datatypes, or 4-way operations on 32-bit datatypes and implements 241 instructions. The second extension is called BSPx, and was designed to accelerate bit and bit-field manipulations frequently encountered in parsing video bitstreams. It implements 34 instructions. Both extensions feature a dedicated register file to minimize interference and access time. With 16 processing elements, a P2012 cluster can deliver up to 150 programmable GOPS peak. An STxP70-4 version with a floating-point unit extension is also available. In this configuration, one single ENCore16 cluster can execute up to 16 floating point operations per cycle. The dual-issue architecture of STxP70-4 ensure that this FLOP rate is sustainable as loads, stores and integer operations can be issued in parallel with the FLOPs. In addition, since all cores have independent instruction issue pipelines, there is no single-instruction, multiple-data restriction on execution, which is a common restriction of GPUs. This greatly simplifies application porting and development.

The ENCore16 processing elements do not have private data caches or memories, therefore avoiding memory coherency overhead. Instead, the processing elements can directly access a L1-shared program cache (P$) and a L1-shared data memory (TCDM). Each core therefore has two 64-bit ports to the shared memories, a read-only instruction port and a read/write data port. The P$ is a 256 KB, 64- bank, direct mapped cache memory while the TCDM is a 256-KB, 32-bank memory. The P$ and the TCDM have been architected with a banking factor (number of memory banks divided by the number of processor) of 4 and 2, respectively. This reduces memory bank conflicts whenever several PEs attempt to access the same memory bank simultaneously. The P$ and TCDM can therefore support a throughput of one instruction fetch and one data access per PE on each clock cycle. The performance of most digital systems today is limited by the performance of their interconnection between logic and memory, rather than the performance of the logic or memory itself. The ENCore16 architecture solves this problem by using a high-performance network to interconnect its processing elements and on-chip shared memory banks.

ENCore provides runtime acceleration by the means of a Hardware Synchronizer (HWS). The HWS includes a synchronization module, an event and an interrupt generator, a dynamic allocator, and a fault-tolerance module. The synchronization module provides a hardware-supported acceleration of

various synchronization mechanisms: semaphores, mutexes, barriers, joins, etc. The event and interrupt generators allow any processor to generate an event or interrupt (a PE sees the event as a low-priority interrupt) to any ENCore16 PE. To avoid systematic polling operations, automatic notification is made possible via the programmable notifier which can send trigger events on specific synchronization values. The Interrupt generator also contains a System Messenger providing a set of mail-boxes to generate interrupt notifications to a target PE, whenever a new message is stored. The Dynamic Allocator allows the system to dynamically assign a task to an available PE. It is able to check the resource states (PE fault status, dynamic interrupt priority level table, etc.) and to allocate the best processor to execute the task. The fault- tolerance module gives access to the faulty states of cluster STxP70 cores and controls a set of watchdog timers for detecting violations of execution deadlines in the application due to transient or permanent fault occurrences in the cluster. The ENCore16 interfaces with the rest of the system through the ENCore16 interface including a 128-bit data interface to transfer data blocks from the external memory to the internal memory and vice versa, a 128-bit instruction interface to transfer instruction blocks from the external memory to the P$ using the P$ refill engine; one In/Out interface decomposed into one 64-bit data interface and one 32-bit interface enabling data access by the processing elements to/from the external bus system; finally one 32-bit In/Out interface allowing remote accesses of the HWS. Each interface works without any dependency to the other interfaces.

The cluster controller (CC) consists of a cluster processor sub-system, a DMA sub-system, a CC interconnect, and three interfaces: one to ENCore16, one to the asynchronous network and one to the rest of the SoC. The CC contains a cluster processor designed around a STxP70-V4 dual-issue core without extensions and with 16-KB of program cache and 16-KB of local data memory. The cluster processor, in conjunction with its cluster controller peripherals, is in charge of booting and initializing the ENCore16 PE. It also performs application deployment on the ENCore16 PEs, some of the error handling, as well as energy management through the Clock Variability and Power (CVP) module. The CVP exposes process, variability and temperature sensors information and generates clocks for the different domains.

The DMA sub-system is made of 3 independent DMA channels offering a maximum bandwidth of 7.6 GB/s. It performs the data block transfers from the external memory to the internal memory and vice versa while the various PE are operating. The CC interconnect supports intra and inter-cluster communication. The system interface depends on the system constraints and is therefore configurable with regard to the number of master and slave ports. The interface to the asynchronous network is made of a Stream In/Out

interface and a Network Interface providing a bridge between the synchronous and asynchronous domains.

2.2 Programming on P2012

The Platform 2012 programming tools and runtime support three main classes of platform programming models (PPMs):

- Native Programming Layer is an API which is closely coupled to the platform capabilities. It allows the highest level of control on application-to-resource mapping at the expense of abstraction and platform independence.
- Standards-based Programming Models are based on industry standards that can be implemented effectively on the P2012 platform. They can be used by both third-party and in-house programmers.
- Advanced Programming Models provide a midway productivity/performance tradeoff between the above models. They are built on well-defined parallel programming abstractions that can be efficiently mapped on P2012 capabilities while offering substantial automation tools for exploring different mapping solutions.

The present work will focus its attenction on the Standards programming models with particular regard to OpenCL and its implementation on P2012.

2.2.1 OpenCL

The OpenCL standard is originally aimed at programming the emerging intersection between GPU and CPU. By targeting scalability, P2012 shares structural properties with GPU, in particular the multi-cluster organization as well as a cluster local shared memory. By embedding STxP70 computational cores associated to a 128 bit SIMD vector unit, capable of both executing efficiently control code and SIMD instructions, P2012 shares properties with CPU. Therefore, OpenCL is a programming model that naturally fits with the P2012 architecture. OpenCL distinguishes the host processor running the main application from devices such as P2012, which run computational kernels. Using the OpenCL API, the host application can control the execution of kernels and can communicate with them through global data buffers. Kernels are programmed using the OpenCL C language, derived for ISO C99, which provides built-in vector data types and a set of computational and kernel management functions. Kernels can be executed

on the device in two ways: either as a Task which is a single instance of the kernel, or as a ND-Range where multiple instances of same kernel, called work-items, run concurrently in a N- dimensional computation domain. A ND-Range can have from one to three dimensions, which matches the structure of media data. Task, ND-Range and other commands such as buffer copies can be organized into a graph which gives a second level of parallelism at coarser grain. A ND-Range is itself structured into a set of work groups. Work-items within a work-group may interact by sharing data in the local memory and synchronizing with a work-group barrier. Work-items may also copy data between global and local memories with asynchronous copies and then, at their level, parallelize data transfers and computation.

P2012 efficiently supports both OpenCL task- and data-parallel execution models. In task-level parallelism, each PE can naturally have an independent instruction flow. On the other hand, the shared program cache of Encore16 clusters allows efficient execution of ND-Ranges, since the shared kernel code is fetched only once for all cores and potential divergent execution paths of work-items will not create any overhead. OpenCL work-groups provide a good abstraction of the multi-cluster structure of P2012 for the programmer, with the local memory well-matched to the cluster shared memory. OpenCL C built-in vector types are mapped to VECx vector units. OpenCL atomic built-in functions and barriers are efficiently implemented with the P2012 hardware synchronizer. In addition, STMicroelectronics provides a set of OpenCL extensions that allow to fully leverage P2012, in particular a set multi-dimensional asynchronous memory transfer functions, as well as a set of built-in functions to access specific STxP70 and VECx functionalities.

To illustrate the use OpenCL for P2012, we take as example the Temporal Mosquito Noise Reduction (TMNR) application which aims at reducing a specific noise introduced by video compression. The TMNR algorithm works natively on blocks of NxN pixels, applying a complex sequence of filters to each block. This application is largely data parallel since each block can be processed independently and then concurrently. In such an application, good management of the flow of data is fundamental for both performance and energy efficiency. In particular, data transfers must be done concurrently with computation with the DMA to hide memory latency.

If we consider a single ENCore16 cluster, one can then create a 4x4 2D-Range, matching both the image and ENCore16 properties. The kernel receives as parameters pointers to input and output images and is programmed to process 1/16th of the image corresponding to its position in the 2D-Range. It will then process thousands of blocks in 1080p HD format, for example. The kernel fetches its input data blocks in step-by-step fashion using OpenCL asynchronous copies, which uses the DMA, while processing previous fetched

data and copying output block in a classical pipeline way. Figure 9 illustrates the TMNR processing of an individual data block, the relationship between image structure, ND-range and ENCore16, as well as a conceptual chronogram of the execution of a kernel. Native data blocks may be coalesced into larger blocks in order to optimize DMA transfers, as well as the memory bandwidth since data blocks overlap. For this application, STMicroelectronics has extended the OpenCL asynchronous copy mechanism to support multiple dimensions.

3 Algorithms on String

3.1 The framework

The framework implemented supplies all code needed for creating algorithms on strings and is composed by three headers.
The "automata_utilities.h" contains the code for creating and manipulating automata. The functions here declared are implemented in "automata_utilities.c". To make independent the actions connecting two states from the possible alphabets and languages adopted, making the automata structure easily recyclable for different alphabets and languages, a mapping function has been adopted:

$$m : A \to \mathbb{N}$$

The function *getCharacterId* allow to retrieve the id connected to a given letter of the alphabet.
The structure acted to contain this information is the *AlphabetMapping* structure: this structure contains the information needed by the map function and is composed by an array of chars and a space of memory dedicated to memorize the length of it. The correspondence is between the position into the array and the letter memorized into.
The dictionary is memorized into the Dictionary structure: this structure contains the strings into an array of pointer to char. Furthermore this structure contains the size of the dictionary.
The function *printAutomataToConsole* allows to print the automata to console.
The automata is contained into the *Automata structure*. The state set Q is implemented with an array of size N of pointers to array of int values. Each element of the array points to another array of cardinality $|A|$ and for every letter l in A memorizes the state where this letter conducts. This is implemented throught a correspondence between physical position in the double array (i,j) and the couple (stateId, actionId). The physical dimension in memory of the array is saved into the variable *stateSetSize* while the number of state effectively in the automata are contained into *stateSetCardinality*. The structure has been chosen to grant compactness of the information expecially when loaded in central memory.
An array of int is acted to memorize if a state is a terminal state always throught a mapping between physical position and stateId. A positive value memorized will indicate the existence of a terminal state at the selected id.
The function *initAutomata* allocates the storage for the automata with a given initial number of states and initializes it.

The function *addConnection* allows, given the state from which to depart, to add a connection to a new state and return an id meaning the new state. If $|Q| > N$ ($stateSetCardinality > stateSetSize$) the function allocates dynamically new space for the automata. The set of the terminal states is then updated to hold in memory trace of the new state.
The "automata_functions.h" header file contains functions that take an automata and returns an automata properly modified to accomplish some aim, like recognizing the language A^*X. The functions here declared are implemented in "automata_functions.c".
Into the "automata_functions.h" the functions on strings are declared. Generally, a function here declared, receives an automata as input and returns an automata suitably modified. The function $buildTrie$ implements the algorithm buildTrie() while the function $initAutomata$: implements the algorithm for building the dictionary automaton discussed in this work.

3.2 The Algorithm Parallelized

The algorithm has been revised to extract the maximum parallelism available. The result is composed of two sections: a serial one containing code acted to instanciate the automata and manage the results of the other one, that makes extensive use of OpenCL code. The use of OpenCL and the presence of more compute units enable the possibility to visit the tree exploring more paths in the same time and then more nodes at everystep. For every trie automata given as input, the algorithm has to produce the same result of the serial algorithm seen in 1.2.4 and then the need of mantaining the equivalence of the final results. It's possible to visit more nodes at the same level at the same time, but the level order in the trie has to be preserved: it's not possible to visit a node at the level $l+1$ before having visited the nodes at level l because the computation in a node at level l makes use of informations computed at level l, in particular the new connections created at the previous levels. It would eventually be possible in a pure breadth-first algorithm.
This can be translated into the need to mantain the visiting order into the trie and then to implement a less strinct LIFO policy, at tree level rather then at node level. The final program realized can be splitted in four sections:

- the OpenCL initialization procedure
- the initialization of the memory on the stack of the process. This step is necessary because OpenCL has not visibility on the heap.
- the first loop of the algorithm
- the second loop of the algorithm

3.2.1 The initialization of the memory on the stack

The initialization consists in the implementation of a mapping function:

$$g : \mathbb{N}^2 \to \mathbb{N}$$

so that for every couple (i,j) in the automata structure, corresponds to a one and only one index into the stack automata structure, that is a compact, consecutive block of memory. This is duty to the need of OpenCL to work on the stack memory of a process. The function definition is:

$$g(i, j) = i * |A| + j$$

The same work is done for the terminal state structure but the mapping doesn't affect the indexing.

3.2.2 The first loop

The first loop is managed via OpenCL. The number of kernel activated are $|A|$. It is passed the automata present on the stack into a buffer and another buffer containing the space for the queue initialized with the first level of the trie. Every kernel has to compute the couple (p, r) relative to the root node. For every kernel are reserved the indexes $kernelid * 2$ and $kernelid * 2 + 1$ on the out queue buffer.

3.2.3 The second loop

The second loop computation is distributed along the platform. Part is accomplished into the main device and partially inside the clusters.
The kernel code receveis the stack automata into a buffer, the current queue containing the states of certain level of the trie, the outing trie empty, the set of terminal states and the alphabet size. Every kernel computes the effect of a letter identified by $id2$ on a state in the queue identified by $id1$. The number of kernel activated are in this way $|A| * b$ where b is the maximum number of states into a trie level.
Every kernel has its space reserved both on the current queue than on the outing queue.
On the current queue the space reserved to a kerned is identified by the pair $(id1, id2)$ is $(id1 * 2, id1 * 2 + 1)$ while on the outing queue is $(id1 * |A| * 2 + id2 * 2, id1 * |A| * 2 + id2 * 2 + 1)$. The first term of the sum is $(id1 * |A| * 2)$ and indentifies the block of memory on the outing queue corresponding to the state $id1$ expanded.
The second term of the sum is $(id2 * 2)$ and is acted to identify into the space

reserved to the state expanded the space reserved to the letter tested on the state. If a couple $(id1, id2)$ doesn't produce any result, a -1 value is filled into the queue.
The code on the device loops launch the kernel code, waits for its end and then reorders the queue. To do this the algorithm scan the queue looking for the first position empty and marks it into a variable called $first$ that has the aim to take in memory the position of the next free position consecutive to the block of the values already ordered. Then the algorithm repositionates every value into the position marked by the first variable. The algorithm doesn't need to be activated at the end of every step of the extern cycle.

4 Results

The algorithm has been tested on a DNA representative dictionary with the following results.[6] [7] [8]

4.1 The DNA Model

4.1.1 The DNA Structure

Deoxyribonucleic acid (DNA) is a nucleic acid containing the genetic instructions used in the development and functioning of all known living organisms (with the exception of RNA viruses). The DNA segments carrying this genetic information are called genes.
DNA consists of two long polymers of simple units called nucleotides, with backbones made of sugars and phosphate groups joined by ester bonds. These two strands run in opposite directions to each other and are therefore antiparallel. Attached to each sugar is one of four types of molecules called nucleobases. It is the sequence of these four nucleobases along the backbone that encodes information. This information is read using the genetic code, which specifies the sequence of the amino acids within proteins. The code is read by copying stretches of DNA into the related nucleic acid RNA in a process called transcription. Within cells DNA is organized into long structures called chromosomes. During cell division these chromosomes are duplicated in the process of DNA replication, providing each cell its own complete set of chromosomes. Within the chromosomes, chromatin proteins such as histones compact and organize DNA. These compact structures guide the interactions between DNA and other proteins, helping control which parts of the DNA are transcribed.
The dictionary adopted for testing is a collection of 3000 DNA segments all of the same length l. The segment is modelled as a sequence of nucleotids and every nucleotid is modelled with the generic IUPAC nucleotide code reported in table .

A nucleotide is composed of a nucleobase, a five-carbon sugar and one phosphate group. Nucleic acids are polymeric macromolecules made from nucleotide monomers. In DNA, the purine bases are adenine and guanine, while the pyrimidines are thymine and cytosine.

4.1.2 The Model [6]

The dictionary adopted for testing is a collection of 3000 DNA segments all of the same length l. The segment is modelled as a sequence of nucleotids

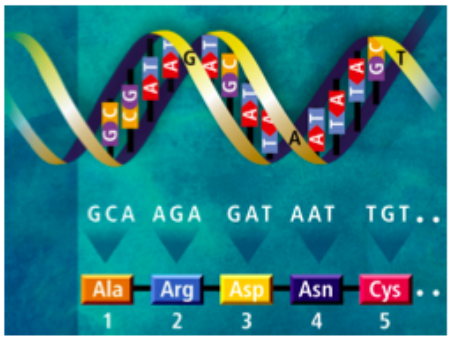


Figure 1: The DNA model

IUPAC nucleotide code	Base
A	Adenine
C	Cytosine
G	Guanine
T (or U)	Thymine (or Uracil)
R	A or G [puRine]
Y	C or T (U) [pYrimidine]
S	G or C
W	A or T (U)
K	G or T (U)
M	A or C
B	C or G or T (U)
D	A or G or T (U)
H	A or C or T (U)
V	A or C or G
N	any base
. or -	gap

Figure 2: IUPAC Codes

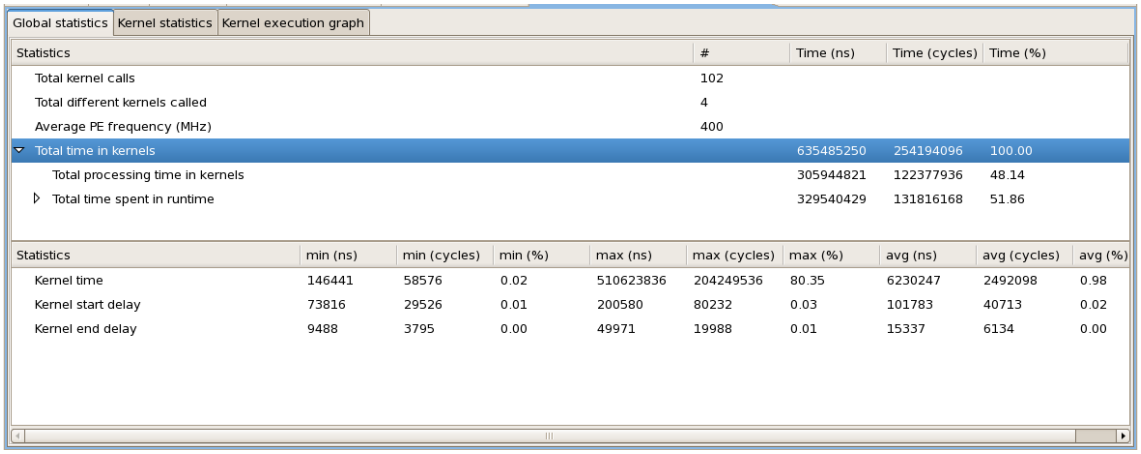
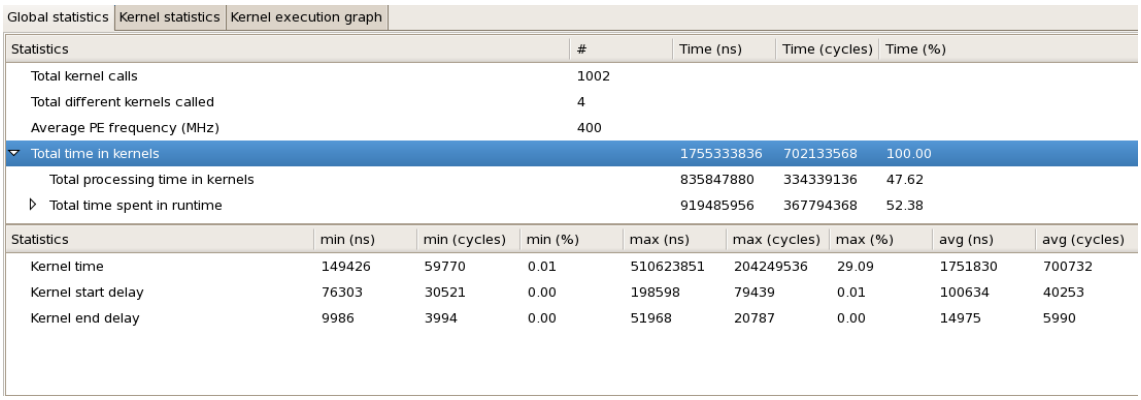


Figure 3: Results with 300 nodes

and every nucleotid is modelled with the generic IUPAC nucleotide code reported in fig.2.
To build the dictionary, a segment of code that create a random one has been implemented. The algorithm have been tested with segement of code of $l \in \{300, 3000, 6000, 9000\}$ characters. A major size could not be adopted because of the stack segment dimension. For every test a collection of measures have been caputered. It's here reported the time spent, passed inside the host and inside the PE's of P2012.

The dictionary adopted for testing is a collection of 3000 DNA segments all of the same length l. The segment is modelled as a sequence of nucleotids and every nucleotid is modelled with the generic IUPAC nucleotide code reported in table . The dictionary adopted for testing is a collection of 3000 DNA segments all of the same length l. The segment is modelled as a sequence of nucleotids and every nucleotid is modelled with the generic IUPAC nucleotide code reported in table .

The number of calls to kernel have been 102 with 300 nodes, 1002 with 3000 nodes, 2001 with 6000 nodes, 4500 with 9000 nodes. So every every kernel have worked on an average of three nodes (3PEs working for call) reduced to two in the case of 9000 nodes. The time spent is of 0.6ms for 300 nodes, 1.7ms for 3000 nodes, 3.5ms for 6000 nodes and 7.29 ms for 9000 nodes. So the the growing of the time spent is not linear. The complete results are reported in fig.3, fig.4, fig.5, fig. 6.

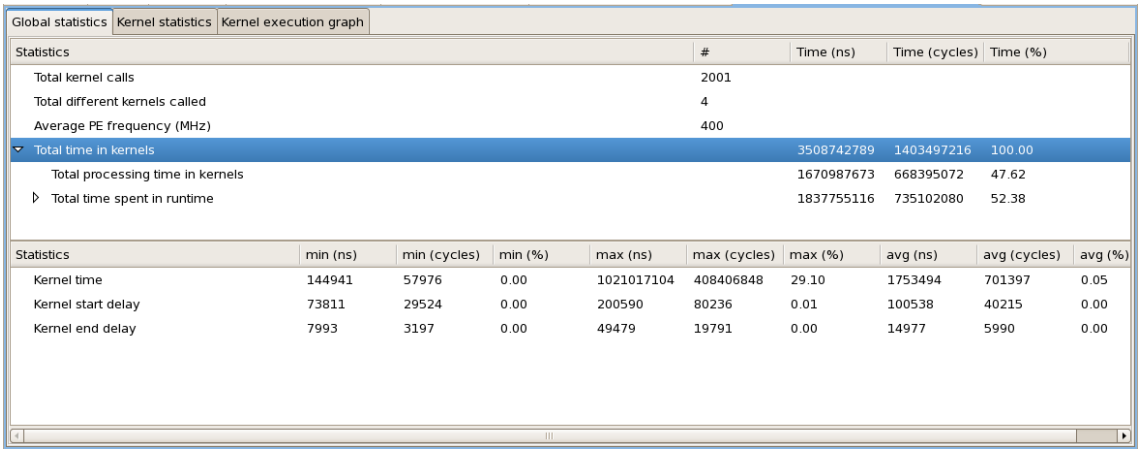


Global statistics | Kernel statistics | Kernel execution graph

Statistics	#	Time (ns)	Time (cycles)	Time (%)
Total kernel calls	1002			
Total different kernels called	4			
Average PE frequency (MHz)	400			
Total time in kernels		1755333836	702133568	100.00
Total processing time in kernels		835847880	334339136	47.62
Total time spent in runtime		919485956	367794368	52.38

Statistics	min (ns)	min (cycles)	min (%)	max (ns)	max (cycles)	max (%)	avg (ns)	avg (cycles)
Kernel time	149426	59770	0.01	510623851	204249536	29.09	1751830	700732
Kernel start delay	76303	30521	0.00	198598	79439	0.01	100634	40253
Kernel end delay	9986	3994	0.00	51968	20787	0.00	14975	5990

Figure 4: Results with 3000 nodes

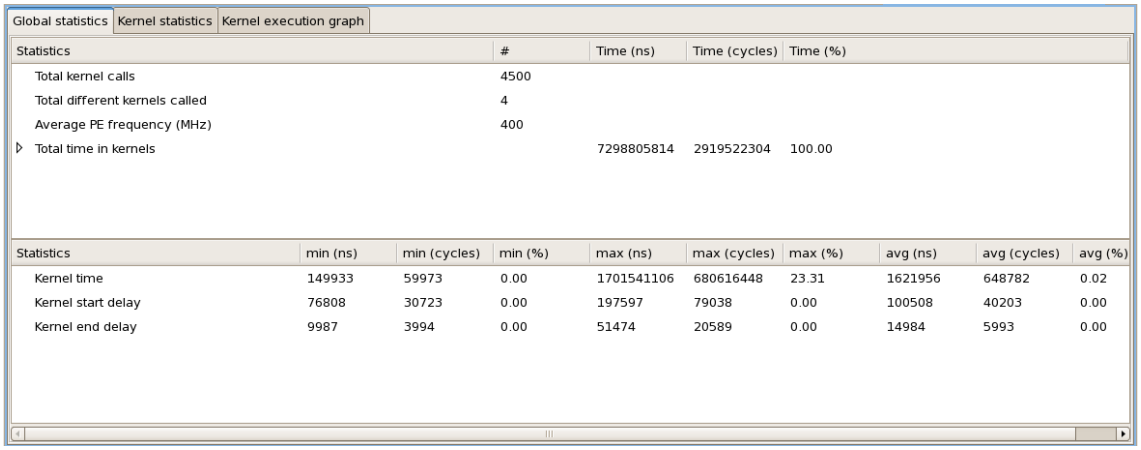


Global statistics | Kernel statistics | Kernel execution graph

Statistics	#	Time (ns)	Time (cycles)	Time (%)
Total kernel calls	2001			
Total different kernels called	4			
Average PE frequency (MHz)	400			
Total time in kernels		3508742789	1403497216	100.00
Total processing time in kernels		1670987673	668395072	47.62
Total time spent in runtime		1837755116	735102080	52.38

Statistics	min (ns)	min (cycles)	min (%)	max (ns)	max (cycles)	max (%)	avg (ns)	avg (cycles)	avg (%)
Kernel time	144941	57976	0.00	1021017104	408406848	29.10	1753494	701397	0.05
Kernel start delay	73811	29524	0.00	200590	80236	0.01	100538	40215	0.00
Kernel end delay	7993	3197	0.00	49479	19791	0.00	14977	5990	0.00

Figure 5: Results with 6000 nodes



Global statistics | Kernel statistics | Kernel execution graph

Statistics	#	Time (ns)	Time (cycles)	Time (%)
Total kernel calls	4500			
Total different kernels called	4			
Average PE frequency (MHz)	400			
Total time in kernels		7298805814	2919522304	100.00

Statistics	min (ns)	min (cycles)	min (%)	max (ns)	max (cycles)	max (%)	avg (ns)	avg (cycles)	avg (%)
Kernel time	149933	59973	0.00	1701541106	680616448	23.31	1621956	648782	0.02
Kernel start delay	76808	30723	0.00	197597	79038	0.00	100508	40203	0.00
Kernel end delay	9987	3994	0.00	51474	20589	0.00	14984	5993	0.00

Figure 6: Results with 9000 nodes

5 References

[1] Stuart Russell, Peter Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, 1st Edition, 1995.

[2] Maxime Crochemore, Christophe Hancart, Thierry Lecroq, *Algorithms On Strings*. Cambrige University Press, New York, 1st Edition, 2007.

[3] STMicroelectronics and CEA, *Platform 2012: A Many-core programmable accelerator for Ultra- Efficient Embedded Computing in Nanometer Technology*. November 2010

[4] Wikipedia, *Finite-state machine*. March 2012, url: `http://en.wikipedia.org/wiki/Finite-state_machine`

[5] Wikipedia, *Depth-first search*. March 2012, url: `http://en.wikipedia.org/wiki/Depth-first_search`

[6] Wikipedia, *DNA*. March 2012, url: `http://en.wikipedia.org/wiki/DNA`

[7] Wikipedia, *Nucleic acid*. March 2012, url: `http://en.wikipedia.org/wiki/Nucleic_acid`

[8] Wikipedia, *Nucleic acid*. March 2012, url: `http://en.wikipedia.org/wiki/Nucleotide`

A Code

A.1 Automata_Utilities.h

```
typedef struct Dictionary
{
        char** dictionary;
        int size;
} Dictionary;


typedef struct AlphabetMapping
{
        char* map;
        int size;
} AlphabetMapping;

int getCharacterId(AlphabetMapping* mapping, char
   currentCharacter);

typedef struct
{
        int** stateSet;
        bool*  terminalStates;
        int   stateSetCardinality;
        int   stateSetSize;
        int   actionContainerSize;
}Automata;

errorType initAutomata(Automata* automata, int
   initialCardinality, int alphabetCardinality);
int addConnection(Automata* automata, int firstStateId,
    int characterId);
void printAutomataToConsole(Automata* automata,
   Dictionary* dictionary, AlphabetMapping* mapping);

#endif /* TYPE_DEFINITIONS_H_ */
```

A.2 Automata_Utilities.c

```
int getCharacterId(AlphabetMapping* mapping, char
   currentCharacter)
{
        int id=0;
        while(mapping->map[id]!=currentCharacter)
                id++;
        if(id!=mapping->size)
                return id;
        else return -1;
};


errorType initAutomata(Automata* automata, int
   initialCardinality, int alphabetCardinality)
{
        int i,j=0;
        if(initialCardinality==0)
                initialCardinality=100;
        automata->stateSet=(int**) malloc(
           initialCardinality*sizeof(int*));
        automata->terminalStates=(bool*) malloc(
           initialCardinality*sizeof(bool));
        automata->stateSetCardinality=1;
        automata->stateSetSize=initialCardinality;
        for(i=0;i<automata->stateSetSize;i++)
        {
                automata->terminalStates[i]=0;
                automata->stateSet[i]=(int*) malloc(
                   alphabetCardinality*sizeof(int));
                for(j=0;j<alphabetCardinality;j++)
                {
                        automata->stateSet[i][j]=-1;
                }
        }
        automata->actionContainerSize=
           alphabetCardinality;
        printf("Trie memory allocated\n");
        return SUCCESS;
};

int addConnection(Automata* automata, int firstStateId,
```

```
    int characterId)
{
        if(automata->stateSetCardinality>automata->
            stateSetSize)
        {
                int i,j=0;
                int** temp=(int**)realloc(automata->
                    stateSet,(automata->stateSetSize)*2*
                    sizeof(int*));
                bool* tempFinalStates=(bool*)realloc(
                    automata->terminalStates,(automata->
                    stateSetSize+firstStateId)*2*sizeof(
                    bool));
                if(temp==NULL|| tempFinalStates==NULL)
                {
                        return -1;
                }
                automata->stateSet=temp;
                automata->terminalStates=
                    tempFinalStates;
                automata->stateSetSize*=2;
                for(i=automata->stateSetCardinality;i<
                    automata->stateSetSize;i++)
                {
                        automata->terminalStates[i]=0;
                        automata->stateSet[i]=(int*)
                            malloc(automata->
                            actionContainerSize*sizeof(
                            int));
                        for(j=0;j<automata->
                            actionContainerSize;j++)
                        {
                                automata->stateSet[i][j
                                    ]=-1;
                        }
                }

        }
        automata->stateSet[firstStateId][characterId]=
            automata->stateSetCardinality;
        automata->stateSetCardinality++;
```

```
        return automata->stateSet[firstStateId][
            characterId];
};

void printAutomataToConsole(Automata* automata,
    Dictionary* dictionary, AlphabetMapping* mapping)
{
        int i=0,j=0;
        for(i=0;i<automata->stateSetCardinality;i++)
        {
                printf("\nState: %d",i);
                if(automata->terminalStates[i])
                        printf(": Terminal state\n");
                else printf("\n");
                for(j=0;j<automata->actionContainerSize
                    ;j++)
                {
                        if(automata->stateSet[i][j
                            ]!=-1)
                                printf("Action %c led
                                    to state %d\n",
                                    mapping->map[j],
                                    automata->stateSet[i
                                    ][j]);
                }
        }
}
```

A.3 Automata_Functions.c

```
errorType buildTrie(Automata* automata, Dictionary*
    dictionary, AlphabetMapping* mapping)
{
        int wordIndex=0, currentStateId=0, nextStateId
            =0,characterPositionInString=0;
        char currentCharacter=0;
        for(wordIndex=0;wordIndex<dictionary->size;
            wordIndex++)
        {
```

```
                currentStateId=0;
                characterPositionInString=0;
                currentCharacter=dictionary->dictionary
                    [wordIndex][0];
                while(currentCharacter!='\0')
                {
                        int characterId=getCharacterId(
                            mapping, currentCharacter);
                        nextStateId=automata->stateSet[
                            currentStateId][characterId
                            ];
                        if (nextStateId==-1)
                                nextStateId=
                                    addConnection(
                                    automata,
                                    currentStateId,
                                    characterId);
                        currentStateId=nextStateId;
                        characterPositionInString++;
                        currentCharacter=dictionary->
                            dictionary[wordIndex][
                            characterPositionInString];
                }
                automata->terminalStates[currentStateId
                    ]=1;
        }
        return SUCCESS;
};

errorType buildDMA(Automata* trie, Dictionary*
    dictionary, AlphabetMapping* mapping)
{
        const char* loop=
                                "__kernel void
                                    initQueue(__global
                                    int *queueBuffer)\n
                                    "\
                "{\n "\
                "    int id=get_global_id(0);\n"\
                "    queueBuffer[id]=-1;\n"\
                "}\n"\
```

```
"\n"\
                "__kernel void
                   initAutomata(
                   __global int *
                   automataBuffer)\n"\
"{\n "\
"    int id=get_global_id(0);\n"\
"    automataBuffer[id]=-1;\n"\
"}\n"\
"\n"\
"__kernel void firstLoop(__global int *
   automata, __global int *outQueue)\n
   "\
"{\n"\
"  int id = get_global_id(0);\n "\
"  int nextState=automata[id];\n"\
"  if(nextState==-1)\n{\n"\
"     outQueue[id*2]=-1;\n"\
"     outQueue[id*2+1]=-1;\n"\
"     automata[id]=0;}"\
"  else {outQueue[id*2] = nextState;
   outQueue[id*2+1]=0;} \n"\
"}\n"\
"\n"\
"__kernel void secondLoop(__global int
   *automata, __global int *firstQueue,
   __global int *secondQueue,__global
   int *terminalStates,__global int *
   alphabetSize, __global uint *counter
   )\n"\
"{\n "\
"  int q=0, s=0, p=0, r=0, id1=
   get_global_id(0), id2=get_global_id
   (1);  \n"\
"  p=firstQueue[id1*2];"\
"  r=firstQueue[id1*2+1]; \n"\
"  if(p!=r)\n{ \n"\
"                atomic_inc(counter);\n
   "
"        if(terminalStates[r]==1)\n"\
"                        terminalStates[
```

```
        p]=1;\n"\
"           q=automata[p*alphabetSize[0]+
        id2];\n"\
"                   s=automata[r*
        alphabetSize[0]+id2];\n"\
"           if(q==-1)\n{\n"\
"               automata[p*alphabetSize[0]+
        id2]=s;\n"\
"               secondQueue[id1*
        alphabetSize[0]*2+id2*2]=-1;\n"\
"                           secondQueue[id1
        *alphabetSize[0]*2+id2*2+1]=-1;\n}\n
        "\
"                   else\n{ \n"\
"               secondQueue[id1*
        alphabetSize[0]*2+id2*2]=q;\n"\
"                           secondQueue[id1
        *alphabetSize[0]*2+id2*2+1]=s;\n}\n
        "\
" }\n"\
" else\n{\n  \n"\
"           secondQueue[id1*alphabetSize
        [0]*2+id2*2]=-1;\n"\
"                   secondQueue[id1*
        alphabetSize[0]*2+id2*2+1]=-1;\n}\n
        "\
"    \n"\
"}\n"\
"\n";

cl_context context;
cl_context_properties properties[3];
cl_kernel initQueue, initAutomata, firstLoop,
   secondLoop;
cl_command_queue command_queue;
cl_program program;
cl_int err;
cl_uint num_of_platforms=0;
cl_platform_id platform_id;
cl_device_id device_id;
cl_uint num_of_devices=0;
```

```
cl_mem automataBuffer, firstQueueBuffer,
    secondQueueBuffer, alphabetSizeBuffer,
    counterBuffer, terminalStatesBuffer;
size_t global[2], globalInitQueue,
    globalInitAutomata, globalFirstLoop;


int queue[MAX_STATES_FOR_LEVEL_TREE*
    ALPHABET_SIZE*VALUES_FOR_STATE]={0};
int automata[MAX_STATES_IN_AUTOMATA]={0};
int terminalStates[MAX_STATES_IN_AUTOMATA]={0};
int i,j,k,totStatesProcessed=1;


if (clGetPlatformIDs(1, &platform_id, &
    num_of_platforms)!= CL_SUCCESS)
{
        printf("Unable to get platform_id\n");
        return GENERAL_ERROR;
}

if (clGetDeviceIDs(platform_id,
    CL_DEVICE_TYPE_ALL, 1, &device_id,
        &num_of_devices) != CL_SUCCESS)
{
        printf("Unable to get a device id\n");
        return GENERAL_ERROR;
}
else {
        printf("Device retrieved\n");
}

properties[0]= CL_CONTEXT_PLATFORM;
properties[1]= (cl_context_properties)
    platform_id;
properties[2]= 0;

context = clCreateContext(properties,1,&
    device_id,NULL,NULL,&err);

command_queue = clCreateCommandQueue(context,
```

```
    device_id, 0, &err);


program = clCreateProgramWithSource(context,1,(
    const char **)
        &loop, NULL, &err);
int error=clBuildProgram(program, 0, NULL, NULL
    , NULL, NULL);

if (error != CL_SUCCESS)
{
        printf("Error building program %d\n",
            error);
        return GENERAL_ERROR;
}

automataBuffer = clCreateBuffer(context,
    CL_MEM_READ_WRITE,
        sizeof(int) * MAX_STATES_IN_AUTOMATA,
            NULL, NULL);
firstQueueBuffer = clCreateBuffer(context,
    CL_MEM_READ_WRITE,
        sizeof(int) * MAX_STATES_FOR_LEVEL_TREE
            *ALPHABET_SIZE*VALUES_FOR_STATE,
            NULL, NULL);

initQueue = clCreateKernel(program, "initQueue
    ", &err);
clSetKernelArg(initQueue, 0, sizeof(cl_mem), &
    firstQueueBuffer);
globalInitQueue=MAX_STATES_FOR_LEVEL_TREE*
    ALPHABET_SIZE*VALUES_FOR_STATE;


initAutomata = clCreateKernel(program, "
    initAutomata", &err);
clSetKernelArg(initAutomata, 0, sizeof(cl_mem),
     &automataBuffer);
globalInitAutomata=MAX_STATES_IN_AUTOMATA;

firstLoop = clCreateKernel(program, "firstLoop
```

```
    ", &err);
clSetKernelArg(firstLoop, 0, sizeof(cl_mem), &
    automataBuffer);
clSetKernelArg(firstLoop, 1, sizeof(cl_mem), &
    firstQueueBuffer);
globalFirstLoop=ALPHABET_SIZE;

secondLoop = clCreateKernel(program, "
    secondLoop", &err);
int alphSize=ALPHABET_SIZE;
secondQueueBuffer = clCreateBuffer(context,
    CL_MEM_READ_WRITE,
          sizeof(int) * MAX_STATES_FOR_LEVEL_TREE
              *ALPHABET_SIZE*VALUES_FOR_STATE,
              NULL, NULL);
alphabetSizeBuffer = clCreateBuffer(context,
    CL_MEM_READ_WRITE,
          sizeof(int), NULL, NULL);
counterBuffer = clCreateBuffer(context,
    CL_MEM_READ_WRITE,
          sizeof(int), NULL, NULL);
terminalStatesBuffer=clCreateBuffer(context,
    CL_MEM_READ_WRITE,
          sizeof(int) * MAX_STATES_IN_AUTOMATA,
              NULL, NULL);
secondQueueBuffer = clCreateBuffer(context,
    CL_MEM_READ_WRITE,
          sizeof(int) * MAX_STATES_FOR_LEVEL_TREE
              *ALPHABET_SIZE*VALUES_FOR_STATE,
              NULL, NULL);
alphabetSizeBuffer = clCreateBuffer(context,
    CL_MEM_READ_WRITE,
          sizeof(int), NULL, NULL);
counterBuffer = clCreateBuffer(context,
    CL_MEM_READ_WRITE,
          sizeof(int), NULL, NULL);
terminalStatesBuffer=clCreateBuffer(context,
    CL_MEM_READ_WRITE,
          sizeof(int) * MAX_STATES_IN_AUTOMATA,
              NULL, NULL);
clEnqueueWriteBuffer(command_queue,
```

```
    alphabetSizeBuffer, CL_TRUE, 0,
            sizeof(int) , &alphSize, 0, NULL, NULL)
                ;
clEnqueueWriteBuffer(command_queue,
    counterBuffer, CL_TRUE, 0,
            sizeof(int) , &totStatesProcessed, 0,
                NULL, NULL);
clSetKernelArg(secondLoop, 0, sizeof(cl_mem), &
    automataBuffer);
clSetKernelArg(secondLoop, 1, sizeof(cl_mem), &
    firstQueueBuffer);
clSetKernelArg(secondLoop, 2, sizeof(cl_mem), &
    secondQueueBuffer);
clSetKernelArg(secondLoop, 4, sizeof(cl_mem), &
    alphabetSizeBuffer);
clSetKernelArg(secondLoop, 5, sizeof(cl_mem), &
    counterBuffer);
clSetKernelArg(secondLoop, 3, sizeof(cl_mem), &
    terminalStatesBuffer);
global[0]=MAX_STATES_FOR_LEVEL_TREE;
global[1]=ALPHABET_SIZE;


clEnqueueNDRangeKernel(command_queue,
    initAutomata, 1, NULL, &globalInitAutomata,
            NULL, 0, NULL, NULL);
clEnqueueNDRangeKernel(command_queue, initQueue
    , 1, NULL, &globalInitQueue,
            NULL, 0, NULL, NULL);

clFinish(command_queue);
clEnqueueReadBuffer(command_queue,
    automataBuffer, CL_TRUE, 0,
            sizeof(int) * MAX_STATES_IN_AUTOMATA,
                automata, 0, NULL, NULL);

k=0;
// Init terminal states automata
for(i=0;i<trie->stateSetCardinality;i++)
{
            terminalStates[i]=trie->terminalStates[
```

```
            i];
        for(j=0;j<trie->actionContainerSize;j
            ++)
        {
                automata[k]=trie->stateSet[i][j
                    ];
                k++;
        }
}

clEnqueueWriteBuffer(command_queue,
    automataBuffer, CL_TRUE, 0,
        sizeof(int) * MAX_STATES_IN_AUTOMATA,
            /*trie->stateSet[0]*/automata, 0,
            NULL, NULL);

clEnqueueNDRangeKernel(command_queue, firstLoop
    , 1, NULL, &globalFirstLoop,
        NULL, 0, NULL, NULL);

clEnqueueReadBuffer(command_queue,
    firstQueueBuffer, CL_TRUE, 0,
        sizeof(int) * MAX_STATES_FOR_LEVEL_TREE
            *ALPHABET_SIZE*VALUES_FOR_STATE,
            queue, 0, NULL, NULL);

clFinish(command_queue);

int first=-2;
for(i=0;i<MAX_STATES_FOR_LEVEL_TREE*
    ALPHABET_SIZE*VALUES_FOR_STATE;i+=2)
{
        if(queue[i]==-1)
        {
                if(first==-2 || i==0)
                        first=i;
        }
        if(queue[i]!=-1)
                if(first!=-2)
                {
                        queue[first]=queue[i];
```

```
                        queue[first+1]=queue[i
                            +1];
                        queue[i]=-1;
                        queue[i+1]=-1;
                        first+=2;
                    }
}
clEnqueueWriteBuffer(command_queue,
    firstQueueBuffer, CL_TRUE, 0,
            sizeof(int) * MAX_STATES_FOR_LEVEL_TREE
                *ALPHABET_SIZE*VALUES_FOR_STATE,
                queue, 0, NULL, NULL);
clEnqueueWriteBuffer(command_queue,
    secondQueueBuffer, CL_TRUE, 0,
            sizeof(int) * MAX_STATES_FOR_LEVEL_TREE
                *ALPHABET_SIZE*VALUES_FOR_STATE,
                queue, 0, NULL, NULL);

clEnqueueWriteBuffer(command_queue,
    terminalStatesBuffer, CL_TRUE, 0,
            sizeof(int)* MAX_STATES_IN_AUTOMATA , &
                terminalStates, 0, NULL, NULL);


            while(1)
            {
                    clEnqueueNDRangeKernel(
                        command_queue, secondLoop,
                        2, NULL, global,
                                NULL, 0, NULL, NULL);
                    clFinish(command_queue);

                    clEnqueueReadBuffer(
                        command_queue,
                        automataBuffer, CL_TRUE, 0,
                                sizeof(int) *
                                    MAX_STATES_IN_AUTOMATA
                                    , automata, 0, NULL,
                                     NULL);
                    clEnqueueReadBuffer(
                        command_queue,
```

```
    secondQueueBuffer, CL_TRUE,
    0,
            sizeof(int) *
                MAX_STATES_FOR_LEVEL_TREE
                *ALPHABET_SIZE*
                VALUES_FOR_STATE,
                queue, 0, NULL, NULL
                );
clEnqueueReadBuffer(
    command_queue,
    terminalStatesBuffer,
    CL_TRUE, 0,
            sizeof(int) *
                MAX_STATES_IN_AUTOMATA
                , terminalStates, 0,
                 NULL, NULL);
int first=-2;
for(i=0;i<
    MAX_STATES_FOR_LEVEL_TREE*
    ALPHABET_SIZE*
    VALUES_FOR_STATE;i+=2)
{
if(queue[i]==-1)
{
            if(first==-2 || i==0)
                        first=i;
}
if(queue[i]!=-1)
            if(first!=-2)
            {
                        queue[first]=
                            queue[i];
                        queue[first+1]=
                            queue[i+1];
                        queue[i]=-1;
                        queue[i+1]=-1;
                        first+=2;
            }
}
if(first==0 && queue[0]==-1)
            break;
```

```
            clEnqueueWriteBuffer(
                command_queue,
                firstQueueBuffer, CL_TRUE,
                0,
                        sizeof(int) *
                            MAX_STATES_FOR_LEVEL_TREE
                            *ALPHABET_SIZE*
                            VALUES_FOR_STATE,
                            queue, 0, NULL, NULL
                            );


}
clEnqueueReadBuffer(command_queue,
    automataBuffer, CL_TRUE, 0,
            sizeof(int) *
                MAX_STATES_IN_AUTOMATA,
                automata, 0, NULL, NULL);
clEnqueueReadBuffer(command_queue,
    terminalStatesBuffer, CL_TRUE, 0,
            sizeof(int),&terminalStates ,
                0, NULL, NULL);

k=0;
for(i=0;i<trie->stateSetCardinality;i
    ++)
{
            trie->terminalStates[i]=
                terminalStates[i];
            for(j=0;j<trie->
                actionContainerSize;j++)
            {
                        trie->stateSet[i][j]=
                            automata[k];
                        k++;
            }
}


clReleaseMemObject(automataBuffer);
clReleaseMemObject(firstQueueBuffer);
```

```
                clReleaseMemObject(secondQueueBuffer);
                clReleaseProgram(program);
                clReleaseKernel(secondLoop);
                clReleaseCommandQueue(command_queue);
                clReleaseContext(context);


                return SUCCESS;
};
```