RAPID-2, An Object Oriented Associative Memory Applicable to Genome Data Processing

Denis Archambaud, Pascal Faudemay
MASI Laboratory - UPMC
4 place Jussieu, 75252 Paris Cedex 05, France
email : archambaud@masi.ibp.fr

Abstract

The RAPID-2 system is a set-associative memory organized into virtual circuits. Such an architecture implies a high flexibility for data storage (structured data with various types and sizes are supported), and a high-speed execution possibilities since set associativity involves massive parallelism and an SIMD organization.

These two features make Rapid-2 a general purpose circuit that can speed-up applications needing parallel processing over data. Some genome specific applications can benefit from Rapid-2 to make possible some tasks that required too much execution time so far.

DNA or proteins sequences alignment is a very important application in nowadays research in biology. Standard sequences alignment usually entails software implementation with disastrous execution time (typically one year). A parallel implementation can solve such a problem. We present the Rapid-2 circuit architecture and functionalities and we detail the implementation of genome sequences alignment. We programmed different algorithms with progressive complexity, adapted to different contexts and requirements.

The first method we programmed is a mere distance calculation between a pattern and a sequence. This distance correspond to the number of pairs to change or delete or insert in the pattern to make it matching with the reference sequence. This method needs n^2 comparisons (n being the number of pairs in the sequence), a systolic implementation using Rapid-2 parallelism reduces the complexity to 2.n comparisons. We then improved the distance formula to run the Needleman-Wunsch algorithm with gap detection and pair phases calculation. The Dayhoff mutation matrix encoding the distance between all possible pairs can be handled in the circuit with high parallelism. Such complex formulas entirely run in the memory, with minimum transfer between the host computer and the memory. Execution time evaluations of all these methods are presented. We consider that Rapid-2 could improve nowadays software implementation of a factor 100.

I - Introduction

Many data processing applications need intensive calculation over numerous data. This is the case for knowledge bases, large databases, genomics, image processing and many more topics. To increase the calculus power, we can either improve the performances of processors themselves - but this implies to build a new computer and all its relevant architecture - or design a system that could be connected to a computer and would speed-up its operations. The advantages of the second solution is its low-price conception and the possible compatibility with multiple workstation types.

Associative memories are based on an access to the data depending on their content rather than their address. Parallelism involved in such memories is based on an SIMD organization (Simple Instruction for Multiple Data): the memory is composed of an arbitrary number of identical cells executing the same instruction in parallel. Opposite to the Content Addressable Memories (CAM) which only perform equality comparisons, associative memories perform complex retrieval (data selection according to its content) and calculations in parallel. Such an architecture is cascadable, it is possible to enlarge its capacity with future technologies. Associative memories do not support structured data with various types and sizes. Architectures designed so far with such a concept are dedicated to specific applications as artificial intelligence circuits [Ogura 89].

Set-associative memories allow to perform parallel calculations with faster requests over data. Hashing is used to determine the storage location of data depending on their content, the memory is divided into sets containing data with same hashing value. This implies a reduced access time to a peculiar data since the selection only occurs in one set. Set-associative memory is used for many specific applications, such as accelerators for electronic dictionaries [Motomura 90]. However, classical set-associative memories only support one type of data and one type of query.

Rapid-2 is an object oriented set-associative memory composed of VLSI components. The user can program the circuits and send it requests or operations which will be executed in parallel. Such a system accelerates tasks needing some parallel processing over data. Architectures built so far have been massively applied to specific contexts (cache management, dictionaries, AI...). This partly results from small memory capacities and from the fact that no data structures were supported. The Rapid-2 memory space is organized into virtual circuits allowing to store structured data with different types and sizes. The flexibility of the Rapid-2 system gives it a general capability for calculus acceleration in a data parallel approach.

Genome applications is a major scientific challenge which implies very large amounts of calculation. The main genome application consists in performing the alignment of proteins or DNA sequences, i.e. evaluating distances between substrings of two sequences. Such a task is well adapted to Rapid-2 since it requires a massive processing over data. Nowadays algorithms have a redhibitory execution time with software implementation on workstations (typically one year, [Gonnet 92]). As the genome global database grows day after day, more and more alignments are needed. With too slow processing, alignment could become the bottleneck of the on-going research in bio-chemistry, an hardware accelerator could solve this execution time problem.

Rapid-2 can be programmed in accordance to the sequence alignment method used in genomics research, any specificity or modification in the algorithm can be applied when using the circuit. We programmed some known algorithms, from the mere distance calculation to the Needleman-Wunsch algorithm with Dayhoff mutation matrix utilization [Needleman 70].

We first present the Rapid-2 circuit architecture and functionalities, its logical and physical organization, its main devices connected with this special architecture, and the instruction set. We then present the implementation of the distance calculation with program listing and time analysis. More specific genome calculation algorithms with progressive complexity are then discussed.

II - Circuit Organization and Functionalities

The Rapid-2 system is a memory board intended to be connected to a host computer. This memory supports structured data and performs massively parallel operations upon stored data. We present in that chapter the logical organization of the system (regardless the hardware constraints), and the functionalities related to such an architecture.

Data Hierarchy

The memory can be seen as an array in which each column (called a cell) contains 64 words, and each line (called a page) is made up of a word of each cell. Thus, there are 64 pages in the memory, and far more cells. The capacity of a N cells memory is 64.N words. The word size is 52 bits : 32 data bits plus 20 tag bits, tags are flags dedicated to a data word, it enables to manage structured data and to record some events on a data.

The data stored in the memory are structured into a seven-level hierarchy which names are (from top level to the bottom) container, segment, set, data, block, element and word.

- **Data** is the atomic object that the user can store in the memory. It is composed of one block in monoblock mode, and of an arbitrary number of blocks in multiblock mode.

- **Set** - a set is composed of a number of successive pages. When storing data in the memory, an hashcode is calculated from a key (usually a part of the datum). The data are always stored in the set corresponding to their hashcode (this is the set-associative approach).

- **Block** is a datum or a part of a datum contained in one cell. In monoblock mode, each block is a datum.

- **Element** is a subdivision of a datum identified by an element-number (5 bits in the tag). The user defines the element-number of each word and can run selections upon them.

- **Container** is a virtual circuit in which the user defines the structure (size of a set, number of sets, and consequently number of possible hash values). Completely different data can be stored in Rapid-2 in different containers with different storage format. A container is accessed by its first page number. The maximum number of containers is 32.

- **Segment** is a subdivision of a container, identified by a segment-number (5 bits in the tag). The user can manage segments as containers having the same number of sets.

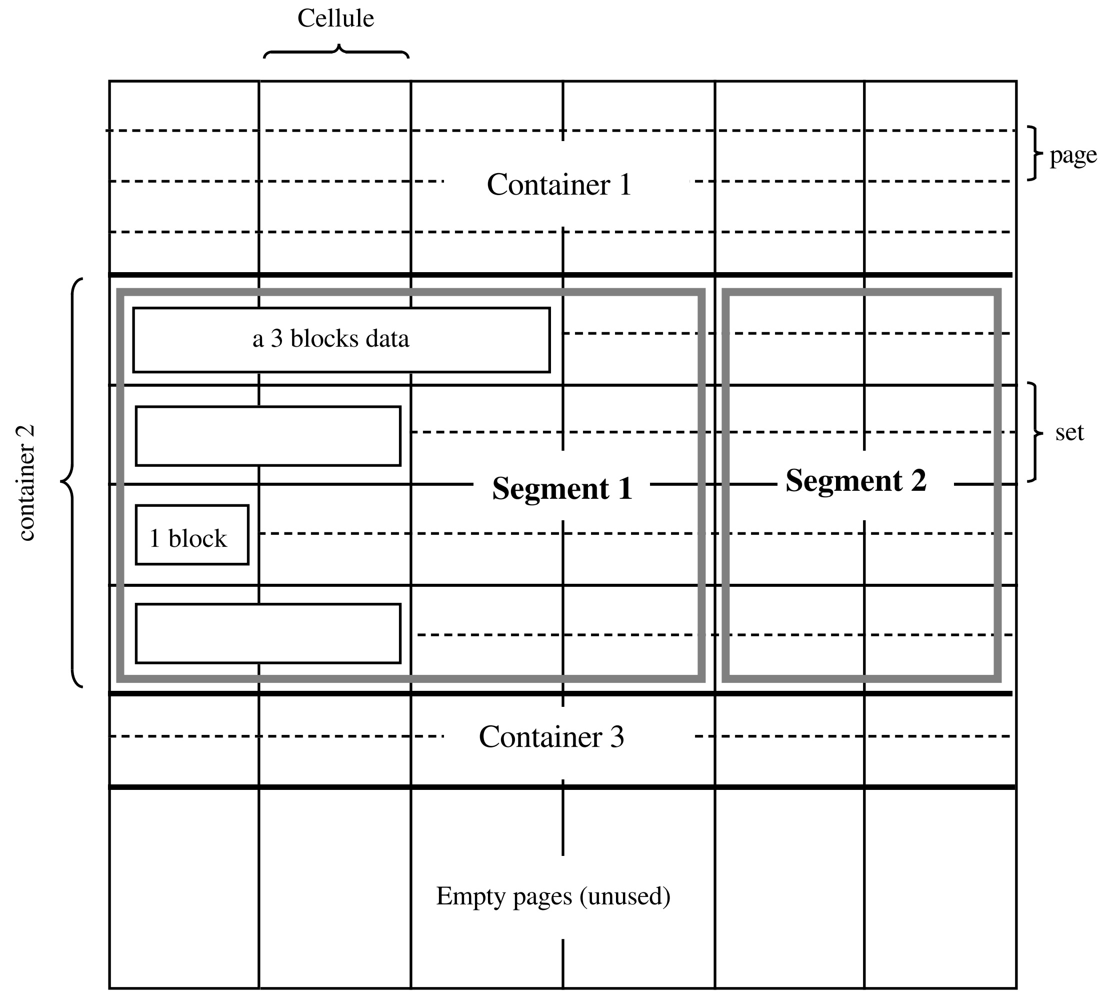


Figure 1 - Paginated structure of the Rapid-2 memory

The user accesses a container by indicating the number of its first page. The segment is accessed by a selection upon the segment number. The set is accessed by hashing a key. The peculiar word of a data is accessed by an offset. The address formula for a page access is then :

*page number = container's first page + set number * block_size + offset*

where the set number is the hash code, the block size is the number of pages of a set.

In order to optimize the access speed, the user indicates directly *page = container's first page + offset,* the formula is then *page number = page+ set number * block_size* . Block size can only be an integer power of 2, from 2 to 32, so that the multiplication is performed by a mere logical shift. Figure 1 shows the structure levels in some containers.

Data types

Different data types can be stored in the circuit, this can be considered as a part of the data structure since the physical storage format depends on the data type. Data type is not connected with data hierarchy (except for texts), it can be different from a word to another in the same block. Rapid-2 can deal with four different data types:

- **Integers** - the 32 data bits are used to store a signed integer from -2,147,483,648 to +2,147,483,647. The most significant bit is used to code the sign (0 means negative, 1 means positive). This implies simplifications when carrying out comparisons since a signed integer and an unsigned integer are equally ordered (i.e. comparing two signed integers as unsigned integers yields a correct result).

- **Bytes** - the 32 data bits are divided into four 8-bit unsigned integers. The cell architecture is designed to perform fundamental operations (addition, substraction...) upon four bytes as upon one 32-bit integer - in other words, when dealing with bytes, the parallelism is multiplied by four.

- **Trits** - a trit is a "trinary digit", i.e. a digit equal to one of the three possible states true, false, or unspecified. Each 32 trits value requires two words : the first 32 bits code the value (true/false) and the following 32 bits code the mask (specified or not). Such a format implies specific logical operations and hashing.

- **Text** - the data words are used to store successive bytes of a text. The format is equivalent to the type Byte, however text implies a special storage format in the data structure : a block occupies always 2 pages, contains 8 characters. In that mode, data, set, and hashing are not available. The text is wrapped, this means that blocks are stored from the left cell to the right, when the last right cell is reached, the following blocks are stored on the two pages below. This data type has been implemented to make Rapid-2 able to store efficiently long texts, since this is a very common object in databases.

Cell Functionalities

The cell is the associative unit, it means that the most optimized it will be (concerning size and consumption), the greater shall be the number of cells. Therefore, the cells contain neither sequencer nor instruction-decoder nor devices that could be shared between cells. The cell actually contains a register file (64 words of 52 bits), auxiliary registers (addressable by the user as 4 auxiliary pages for temporary operations), an arithmetical and logical unit (ALU) performing integer addition and substraction and standard logical operations, a token unit (which will be explained further), some shifters mounted on the auxiliary registers, a very simple address driver, and a limits management unit (cf hashing capabilities).

The identical cells in memory receive the same orders from a sequencer generating a Single Instruction for Multiple Data in the cells (SIMD organization). The parallelism factor is equal to the number of cells in the memory. Each cell can either handle internal operations, or exchange data with its neighbors, or exchange data with the user via the data bus. When dealing with characters or 8-bit integers, each 32-bit cell is composed of four 8-bit virtual cells so that the parallelism is multiplied by a factor 4.

The cells are connected by a monodimentional mesh. We shall consider an horizontal connection in which the "first cell" is on the uttermost left, and the "last cell" is on the uttermost right. We can therefore define some oriented actions, as shifting operations consisting in transferring a datum from a cell to its right or left neighbor.

The cell interface is composed of an address bus (to code the page number), a data bus, and two 52-bit mesh for shifting data, one connected to the previous cell and one connected to the next cell. The token state is also output to the token trees.

Token unit

Each cell has a token which is a flag that determines whether the cell is active or not while executing an instruction. This device is necessary, due to the SIMD organization of the system : the single instruction is sent in all the cells but executed only in the ones whose token is set. Therefore, each action in the memory is a succession of cell selections (token affectations) and operation executions.

Since the token unit involves important capabilities in such an architecture, it is a complex part of the circuit. The tokens can be affected regarding the result of a comparison on data or tag, so that it can indicates the positions of searched words or the positions related to the structure (i.e. data beginning/end). Token transformations are possible too, such as shift left, shift right, propagation, tokens copying into a tag bit, or tokens derivation. Some of these transformations are performed by binary trees connected to all the tokens : an Anderson tree [Anderson 74] is used to perform arbitration (i.e. find first/last set token), an *or* tree indicates whether one token at least is set or not, some trees are also implemented to perform local propagation (limited to one datum).

Hashing capabilities

Hashing provides a way to increase the time performances when searching in the memory. Searching a word in a container without hashing device would imply comparisons with all the words of all the pages. Hashing reduces the number of pages to search in because it places similar data in a given page. More precisely, a container is divided in sets. When storing a datum, the hasher yields an hashcode from its key (i.e. some words representative of the datum), the datum is then stored in the set whose number is determined by the hashcode. When looking for a datum in that container, its hashcode indicates in which set we have to search. In Rapid-2, hashing is carried out by hardware and can be parameterized by the user. The hashcode size can be defined from 1 bit (2 sets) to 5 bits (32 sets). Two hash types are available: folding and intervals.

Folding is a *xor* function of different bits of the key which returns a bitstring. Four variants of the function can be used, depending on the distribution of the *hot bits* (i.e. the bits which probability to be different from one key to another is the highest). This ensures the best hashcodes distribution and therefore the highest sets filling efficiency.

When hashing by intervals, the data range is divided into intervals, thus the hashcode is the number of the interval which includes the key. Each cell can store one limit (and the corresponding hashcode), so that the key is compared in parallel with all the limits. The first successful comparison on the left is then selected and the corresponding hash-code is yielded. Several limits sets can be stored in the Rapid-2 system, the user can therefore define different hashing functions adapted to the data specificities.

Instruction set

A high-level instruction set makes Rapid-2 easy to interface with a software layer. An instruction is a 32-bit word sent to the circuit and interpreted by an hardware sequencer that generates commands for the cells. Figure 2 presents the instruction set.

Instruction COPY is able to transfer a datum from a cell to the next or previous one. This possibility is very important because it enables the programming of systolic algorithms, usually applied to bidimensional problems (Rapid-2 memory can easily encode a bidimensional array). The program presented in chapter V uses a systolic method to solve a bidimensional problem.

INSTRUCTION SET

Update instructions

LOAD :	following data are stored into the circuit
MODIFY :	selected data are replaced in parallel
SUPPRESS :	selected data are suppressed
INSERT :	each data is inserted after its predecessor
CHANGE-LIMITS :	loads limits for hashing by intervals

Retrieval instructions

SELECT :	selects the data which satisfy some data value
TAG-SELECT :	selects the data which satisfy some tag values
READ :	returns the previously selected data with their tags

Redistribution instructions

REHASH :	calculates a new hashing value for each data
REDISTRIBUTE :	redistributes data between sets after re-hashing

Integer arithmetic and binary operations

AGGREGATE :	calculates an aggregate (count, sum, min, max) on virtual classes
OPERATION :	performs an arithmetic or binary operation on a page in parallel (add, sub, mul, div, shift, rotate, and, or, xor, not, comparisons)

Register copying

ALTER-TOKEN :	operation on tokens, such as shift, propagate, set, reset...
COPY :	copies an element of a selected data into a register, and conversely
MOVE-CONT :	moves a container towards empty pages (containers reorganization).

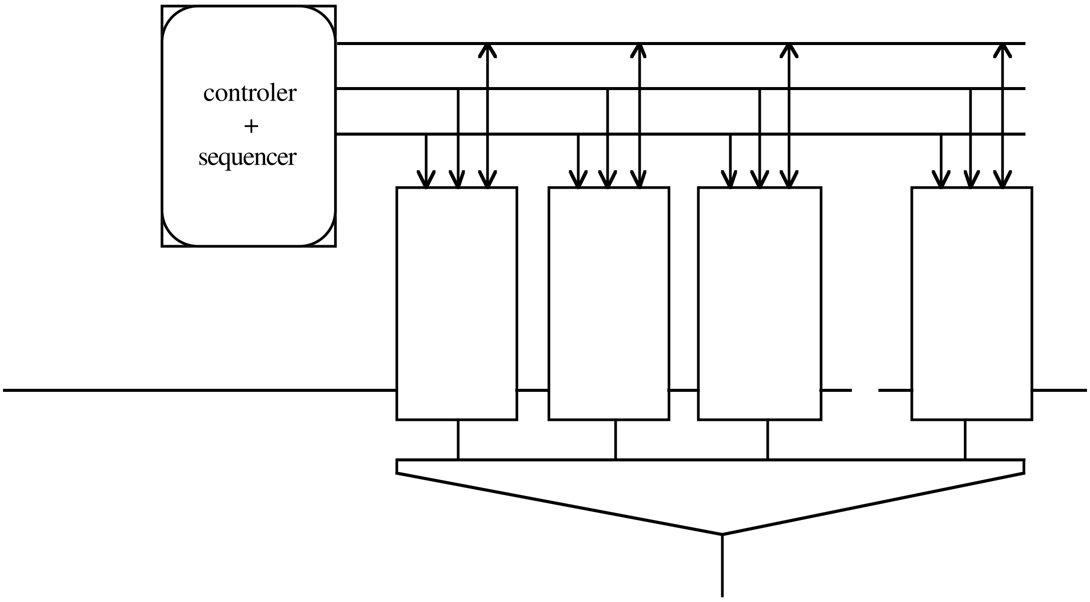
Figure 2

III - Circuits architecture

With nowadays technology, the whole memory cannot be contained in a single circuit. Therefore, the Rapid-2 is a board connecting several associative memory circuits and one arbiter circuit. This chapter describes its organization. The memory circuit is composed of a sequencer and multiple cells. The number of cells can be increased with growing integration level. The behavior of the sequencers is the same in each circuit (they are duplicates). The arbiter is necessary to complete the token-trees and will be used too for the possible mesh connection at extremities.

Basic component architecture

Each basic component is a part of the memory. It contains 16 cells, a sequencer, a controller, and the local token trees. Figure 3 shows the connection of these elements. The sequencer is an automata generating bus μinst (micro-instruction), the controller contains the user accessible registers and the device calculating the page number from parameters page, hashcode and block size. There are four 32-bit registers in the controller, addressable by the user (using address bus). When the user wants to run an instruction, he writes its code on register Instruction, when he wants to send data, he writes on registers Data, Wordtag (for changing tags), or Mask (for trits).



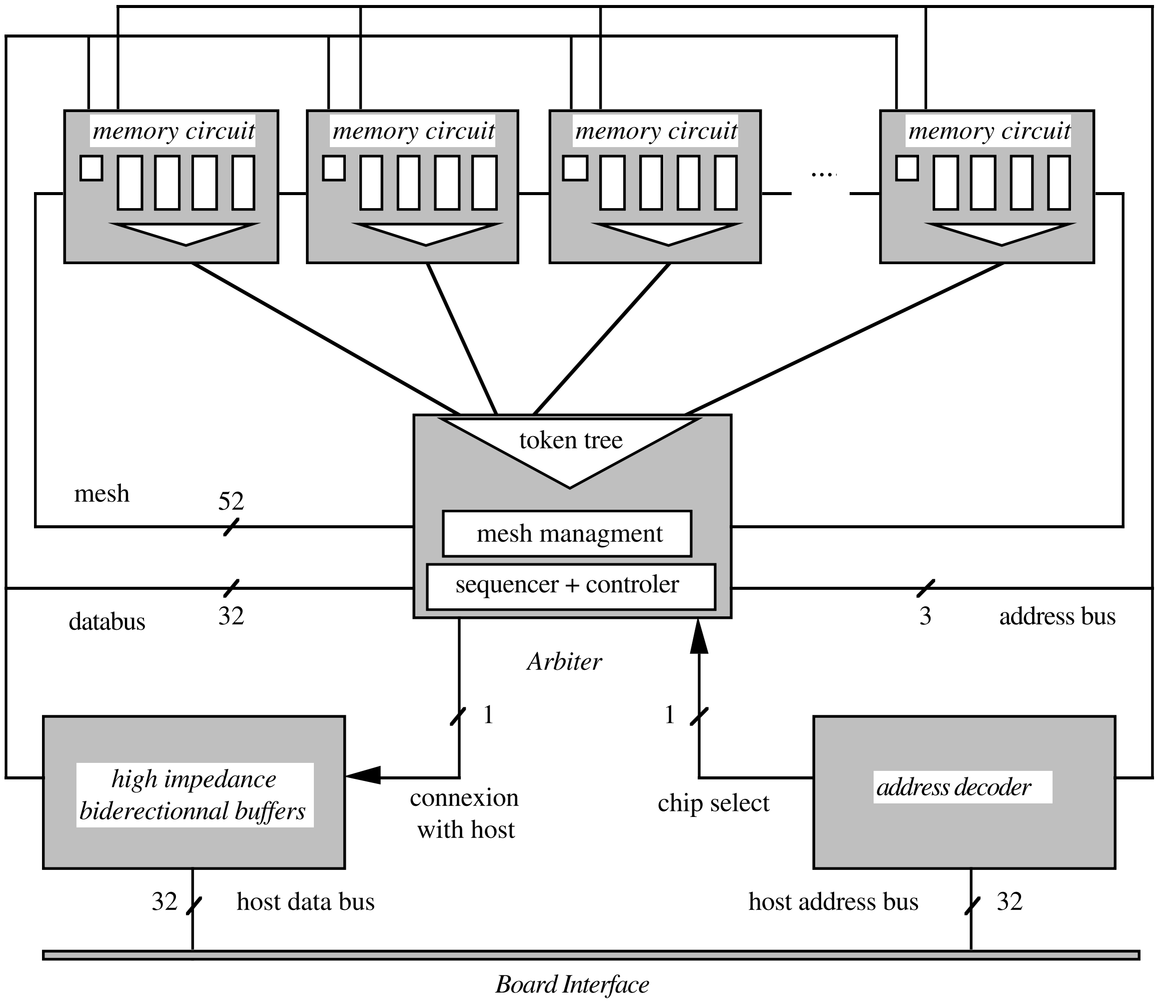


Figure 4 - Board architecture

Feasibility

We aim at a 16 memory circuits board, with 16 cells in each one, i.e. 256 cells memory. Such an expectation is based on different circuits implemented in the MASI laboratory, and particularly a DLXm implementation using a 32-bit DataPath and a Mealy automata (64 states). We estimate that a cell will require less than 50,000 transistors, i.e. 10,200,000 λ^2. The sequencer should not be bigger than three cells, so a 16 cells memory could be achieved in a 1 cm^2 circuit using CMOS 0.8 µm technology. The memory circuit and the arbiter will have approximately the same sizes, the memory being core-limited and the arbiter being pad-limited.

The Rapid-2 board will then feature a 64 Kilo-bytes memory (not counting tag) and will be designed to support a 30 MHz clock rate. The Alliance CAD tool developed at the MASI laboratory will be used for logical simulation (VHDL), layout generation, and behavioral verification [Greiner 92].

IV - Some possible applications for the Rapid-2 memory

Although RAPID-2 was initially dedicated to database operations, its flexibility enlarges its applications on several domains such as text processing, image processing, neural networks, object MMU, and genomics calculations.

Database cache

Rapid-2 is an efficient objects cache that can accelerate object oriented database operations because its architecture supports objects collections with various types and sizes [Faudemay 92]. The circuits can either perform integer or logical operations on appropriate fields (elements in data), or run pattern searching in parts of text. Data migration implies to know whether a segment is present in the circuit or not. This is managed by using a container storing segment descriptors [Lavington 90]. An LRU policy may be used, using tags to encode the objects age or segment age.

Object MMU

Rapid-2 can be used as an object-oriented memory management unit (MMU) attached to a disk cache. Object oriented MMUs can manage variable size objects and map disk files or large disk segments into main memory. A single basic component (containing 16 cells) could be used to map 512 segments into main memory. Parallelism and SIMD organization accelerate the address searching, and thus the cache management. A basic component can be used as a stand-alone circuit as it contains all the devices required to handle the cells : a sequencer, a controller ensuring the interface with the host computer, and a local token tree that as not to be connected to the arbiter's token tree since it is unique.

Neural networks

An artificial neural network (ANN) consists of many processing elements (neurons), each one connected to many others. Neurons are possibly organized into layers, the first layer neurons being connected to the system inputs, and the last layer yielding the output informations. The ANN can learn and adapt by changing the synaptic weights of its neurons. Neural networks implement a kind of massive parallelism that can be carried out either by using analog transistors (but problems arise to build accurate resistors to represent the weights [Hecht-Nielsen 88]), or binary logic that enables high accuracy in calculus (this implies complex neuron structure, e.g. 55.000 transistors per neuron [Yasunaga 93]).

A Rapid-2 cell can have a neuron behavior, interconnection between neurons being performed by a systolic algorithm using the mesh. Rapid-2 can thus contain up to 1024 neurons with 8-bit weights and up to 60 synaptic inputs for each neuron (Yasunaga's wafer scale integrated system is composed of 288 digital neurons with 16-bit weights). Implementation of an ANN with 16-bit or 32-bit weights is also possible. The programming flexibility of Rapid-2 enables to implement differents networks and learning algorithms (Widrow-Hoff, perceptron, back-prop and so on).

Image processing

Images are usually coded as bitmap arrays, each pixel (picture element, i.e. point in the array) coding a colour. A usual image processing consists in changing the colour of each pixel, depending on the pixels in its vicinity. Such algorithms - as convolution matrix for edge detection - are easily applicable to the Rapid-2 architecture because the access to the vicinity of each pixel can be handled by the mesh (at least for one dimension in the array), and the parallelism can transform the complexity from n^2 to n (considering n*n pixels pictures). Edge detection may be followed by a vectorization that transforms the bitmap array into a vector image.

Rapid-2 can also store images coded as vector collections. Comparisons between images can be performed in parallel, by comparing their vectors as we usually compare two texts by comparing their characters. Furthermore, Rapid-2 makes possible approximate image comparisons, images can be retrieved even in case of noise, translation, distortion, changes in scale and so on.

Text processing

When dealing with text, each cell usually storing 32-bit words can be considered as four parallel 8-bit cells with their own ALU and token resources. ACSII texts are stored in the consecutive pages of the memory.

With a naïve algorithm, searching a pattern of length p in a text of length t consists in looking for the first character of the pattern, then the second, and so on. Each matching character is marked, so that a pattern match is detected by p successive character matching. That involves $p.t$ elementary comparisons in a software implementation, since searching one character in the text requires t comparisons. Some refined algorithms using more memory space can run the search task with only $p+t$ or $p.t^{-1}.\log t$ instructions [Crochemore 92].

In the Rapid-2 memory, we compare each pattern character with all the text characters at the same time, the matches are reported in the tokens. After each character search, the token is shifted one character to the right so that the next comparison will be held only in the previously correct parts of the text. At the end of the process, the tokens state indicates the perfect matches (figure 5). Such an implementation only needs $2.p$ instructions to yield the result. Since p is usually small compared with t, the task is well sped-up by the parallelism.

Before the end of the process, the tokens state indicates how many matches are still possible in the text so far. Therefore, if none of the tokens is activated, it means that the pattern does not exist in the text, so the user can stop the operation before the end. Such a situation is detected by the *or* binary tree implemented in the token tree.

A problem arises when the text is long enough so that it cannot be contained in one page. In this situation, the pattern must be searched in all the pages storing the text, one page after the other.

Pattern overlapping two pages can be detected only if the shifted tokens at the end of the page are memorized and fed back in the beginning of the next page when searching in it. The devices needed to run this DPPM algorithm (Data Parallel Pattern Matching, [Herman 91]) are implemented in Rapid-2 circuits and offer transparent string searching for the user. Let ℓ be the length of a page, the number of operations needed is then $2 . p . t / \ell$

```
                b  a  b  a  b  i  b  o  b  a  b  y  b  u  b  a
initial state   x  x  x  x  x  x  x  x  x  x  x  x  x  x  x  x
select 'b'      x     x     x     x     x     x     x     x
shift token        x     x     x     x     x     x     x     x
select 'a'         x     x                 x                 x
shift token           x     x                 x
select 'b'            x     x                 x
shift token              x     x                 x
select 'y'                                       x
```

The pattern 'baby' is searched in the string. The token state codes the matching of a character and enables its possible selection.

When starting, all the tokens are marked so that the first `select` concerns every characters in the text. After that selection, the tokens are shifted and thus 'a' is searched only on characters preceded by 'b'. At the end of the process, the only remaining token indicates the last character of a perfect match (i.e. 'y' preceded by 'b', 'a', and 'b').

Figure 5 - An example of pattern searching in a text

If the text can not be stored in the entire memory (which capacity is 65,536 characters for a 256 cells configuration), it is necessary to load the successive parts of the text, and to search in each one. Loading the entire memory can not be parallized and requires 550 µs, the search speed is then less favourable. As an example, searching a 10 characters pattern in a 65,000 characters text requires $2 . p . t / \ell =$ 1280 instructions, i.e. 40 µs. The same task into a 650,000 characters text (160 pages book equivalent) requires 10*40 µs plus 10*550 µs, i.e. 5.9 ms.

Pattern searching with wildcards can be performed too. The common '?' wildcard (i.e. any character) correspond to a single shift right of the token without any comparison. The '*' wildcard (i.e. arbitrary length text sequence) correspond to a right propagation of the tokens which can be performed in 2 cycles, using the token tree. More complex wildcards correspond to multiple comparison connected by *and* or *or* without shifting.

Combining data hashing with text storing and searching can increase the speed of specific applications as dictionaries [Motomura 90]. In such context, the text is composed of objects that can be stored in different parts of the memory according to their hashcode. Thus searching a certain pattern can be reduced to the memory zone containing the same hashcode than the pattern, dividing the execution time by the number of possible hashcodes (maximum 32 in the current version).

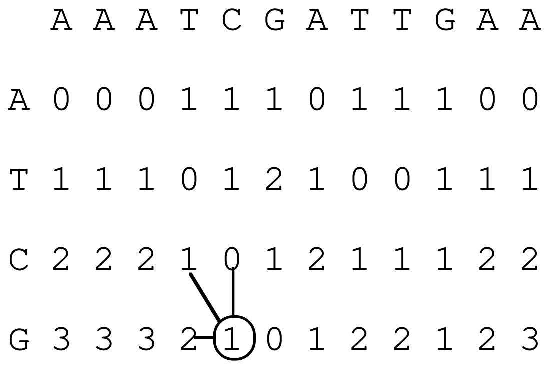
V - A fundamental operation in genomics: approximate searching

Searching similarities between DNA or proteins sequences is a very important application in nowadays research in bio-chemistry. Genome sequences involve an important amount of data, and therefore a long and complex processing whose execution time is usually redhibitory in traditional software implementation. As genome projects around the world produce data at a growing rate, genome sequences processing could become the bottleneck of these initiatives. Hardware accelerators is a solution to this problem. In this chapter, we describe an algorithm that performs an approximate matching between a pattern and a stored text. Parallelism and SIMD orientation are well adapted to problems such as matching proteins or DNA sequences. A sequence can be coded as a text, each character coding a amino acid, or base pair.

Distance calculation between a pattern and a text - general method

Approximate searching consists in calculating - for each text character - a distance between the pattern and the part of text preceding the character. This distance represents the number of characters to insert or delete or change in the pattern to make it matching.

The distances are calculated using a matrix whose columns are connected to the text characters and lines to the pattern characters. The matrix values are calculated from the first line to the last, from the left column to the right one. Each value is equal to the minimum of the three closer values upward, backward and diagonally. Furthermore, this value is incremented if the pattern character connected with the current line is different from the text character connected with the current column (penalty). The last line of the matrix contains - after computation - the distances for all the text characters. Figure 6 is an example of the matrix calculation.



	A	A	A	T	C	G	A	T	T	G	A	A
A	0	0	0	1	1	1	0	1	1	1	0	0
T	1	1	1	0	1	2	1	0	0	1	1	1
C	2	2	2	1	0	1	2	1	1	1	2	2
G	3	3	3	2	1	0	1	2	2	1	2	3

Figure 6 : Distance calculation between text AAATCGATTGAA and pattern ATCG

For instance, the matrix element corresponding to the letter 'G' in the pattern and the letter 'C' in the text has the value 1 because values upward, backward and diagonally are respectively 0, 2 and 1. The result is the minimum (i.e. 0) incremented since G≠C, hence 1.

Initial conditions

When computing on the first matrix line, values upward and diagonally must be zero. When computing on the first matrix column, value backward must be maximum.

Adaptation to the Rapid-2 architecture

The parallel architecture is well adapted to a systolic implementation. The pattern characters are successively aligned to each text character by shifting them to the right (figure 7), each alignment state is called a **step**. At each pattern character we associate several values : the pattern character itself (P), values upward (U), backward (B), diagonally (D), and the result (R).

```
A A A T C G A T T G A A
A
T A
C T A
G C T A
  G C T A
    G C T A
      G C T A
         ...
```

Figure 7 : Pattern shifting for the systolic algorithm

At each step, value R is calculated in parallel in each pattern character with formula $\mathcal{R} = min\ (\mathcal{U}, \mathcal{B}, \mathcal{D})$. R is incremented when the pattern character is different from the corresponding text character.

Figure 8 describes the values transfers during a systolic step. Here are the details concerning pattern character 'G' on the the text character 'C' (situation of figure 6) :

α- The Backward value is the Result of pattern character 'G' aligned with text character 'T' at time t-1 (i.e. 2, see figure 6)

β- The Upward value is the Result of character 'C' aligned with text character 'C' at time t-1 (i.e. 0)

γ- The Diagonal value is the Result of character 'C' aligned with text character 'T' at time t-2 (i.e. 1, see figure 6); but this R value has been transferred in B value of character 'C' while previous shift (γ' on figure 8), therefore D value is B value of character 'C' aligned with text character 'C' at time t-1

The value R in character 'G' at time t is then $\mathcal{R} = min\ (\mathcal{B}, \mathcal{U}, \mathcal{D}) + 1$ as 'G'≠'C'

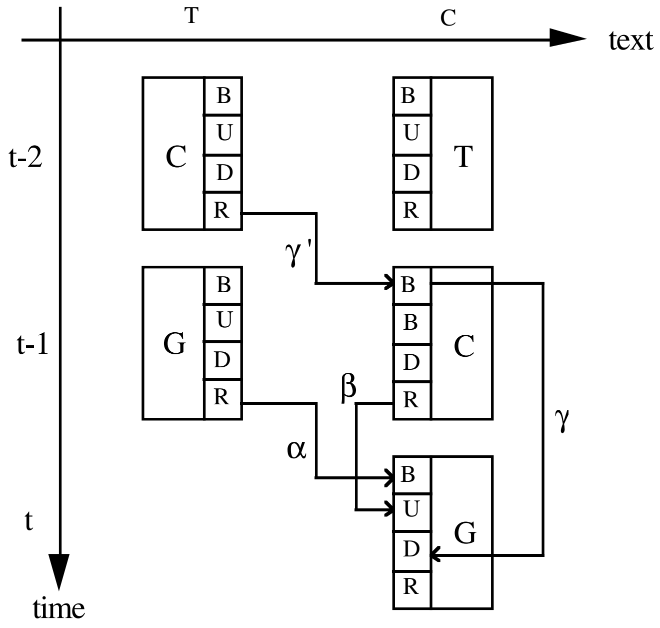


Figure 8 : Values transfers concerning character G

Implementation in Rapid-2 data structure

We use 5 pages in the circuit : the first page stores the text (page T), other pages store values P, B, D and R. The systolic step is detailed in figure 9 :

page T is shifted to the right
page B is copied to page D (corresponding to γ in figure 8)
page R is right shifted and copied to page B (corresponding to α in figure 8)
minimum of values in pages D, R, B is copied to page R
values R are incremented where P≠T

The β transfer (on figure 8) has not to be run because page U is replaced by page R so that formula $\mathcal{R} = min\ (\mathcal{B}, \mathcal{U}, \mathcal{D}) + 1$ is transformed into $\mathcal{R} = min\ (\mathcal{B}, \mathcal{R}, \mathcal{D}) + 1$

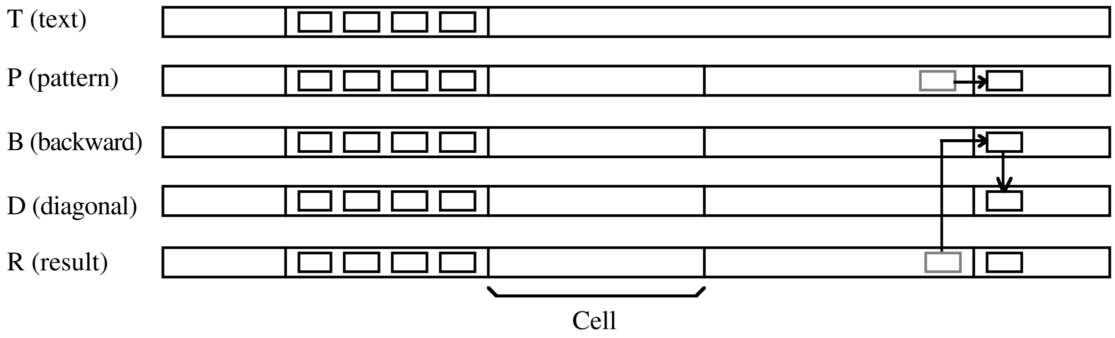


Figure 9 : Rapid-2 pages organization for distance calculation

Systolic step programming

Figure 10 lists the Rapid-2 instructions to run one step. We used a pseudo-language to make the program easier to understand. This algorithm has been simulated by software. Figure 12 displays the content of the pages at each step of a distance calculation.

High-level instructions make possible a very clear programming : only 16 instructions are necessary.

- Instruction `Alter-Token` is used either to set every token `(op=set)`, so that the next instruction could be executed in each cell, or to store the tokens to a page tag bit (`op=copy token to flag`), or to recover that bit (`op=copy flag to token`).
- Instruction `Operation` with operand `rr8` performs a 8-bit right rotate. This is necessary to run a one-byte right shift.
- Instruction `Copy` can only perform 32 bits shift (transfer from a cell to its neighbor). Hence, to run a 8-bit right shift, we first run a 8-bit right rotate and then we shift right the most significant byte of each 32-bit words. That's why `Operation-rr8` is always followed by `Copy-shift-right.` The shift right instruction is applied on the most significant byte of each word because `Operation-rr8` sets adequately the tokens.

```
//  B is transferred to D (γ transfer in figure 8)
ALTER-TOKEN (op = set)
COPY (source = page B ; target = page D ; shift = 0 ; byte = 0)

//  R is shifted and transferred to  B (α transfer in figure 8)
OPERATION (source = page R ; target = page B ; op = rr8 ; byte = 1)
COPY (source = page B ; target = page B ; shift = right ; byte = 1)

//  P is right shifted
ALTER-TOKEN (op = set)
OPERATION (source = page P ; target = page P ; op = rr8 ; byte = 1)
COPY (source = page P ; target = page P ; shift = right ; byte = 1)

//  minimum calculation
ALTER-TOKEN (op = copy flag to token ; page = page P)
OPERATION (source1 = page B ; source2 = page R ; op = less_than ; byte = 1)
COPY (source = page B ; target = page R ; shift = 0 ; byte =1)
ALTER-TOKEN (op = copy flag to token ; page = page P)
OPERATION (source1 = page D ; source2 = page R ; op = less_than ; byte = 1)
COPY (source = page D ; target = page R ; shift = 0 ; byte =1)

// conditional incrementation
ALTER-TOKEN (op = copy flag to token ; page = page P)
OPERATION (source1 = page P ; source2 = page T ; op = not_equal ; byte = 1)
OPERATION (source1 = page R ; source2 = reg DATAIN; target = page R ; op = add ; byte = 1)
// the user must write 1 to the DATAIN register (via the host data bus) to run the inc.
```

Figure 10 : systolic step program

Execution time evaluation

Each Rapid-2 instruction has a specific sequencing depending on the architecture and management by the automata. The timing of each instructions is outside the framework of this paper, however we present in figure 11 a time diagram of the systolic step based on the instructions timing. The *pipe-lining* possibilities (starting an instruction before the end of the previous one) is an important factor of optimization. Pipe-line is used provided there is neither collision (i.e. a resource is required by both tasks at the same time) nor dependency (i.e. the second task needs the first task result to run correctly). The diagram shows that a systolic step requires 30 cycles evaluated to 1µs execution time.

Running a distance calculation between a text of length t and a pattern of length p requires $p+t$ systolic steps. Pages used in the circuit must be initialized before running the algorithm : pages D and R must be set to 0, and page B must be set to maximum (255). This could be executed in no more than 6 cycles. After running the $p+t$ steps, the user must read the results in page R. This requires around 256 cycles. Thus, the global execution time is $262 + (p + t) * 30$ cycles, with $t<1024$ because of the memory capacity (this constraint will be discussed and bypassed further).

```
alter token (set)                     IX
copy (mem -> reg)                     .IXXX
operation (unary mem -> mem)          . IXXX
copy (mem -> mem)                     .   IXXX
alter token (set)                     .     IX
operation (unary mem -> mem)          .      IDXX
copy (mem -> mem)                     .        IXXX
alter token (copy flag2tkn)           .          IXXX
operation (binary mem,mem -> void)    .            IXXXX
copy (mem -> mem)                     .               IXXX
alter token (copy flag2tkn)           .                 IXXX
operation (binary reg,mem -> void)    .                   IXXX
copy (reg -> mem)                     .                     IXX
alter token (copy flag2tkn)           .                      IXXX
operation (binary mem,mem -> void)    .                        IXXXX
operation (binary reg,mem -> void)    .                           IXXX
next step : alter token (set)         .                            .IX
                                      ----------- 30 cycles --------

user data flow                        IIIXIXIIBIXIXIXXIXIXIXIIXIXXIX
```

Letters X, I and D represent the machine cycles. Horizontal axis is time.
Letters in the same column represent pipe-lined cycles. The maximum pipe-line depth is 3.

`X:`	internal calculation cycle with no communication with the user
`I:`	instruction cycle: the user writes the instruction code to the circuits
`D:`	Data input cycle : the user writes the pattern character to be introduced at the beginning of page P while right shift.

Figure 11 : The systolic step timing diagram

```
step 0                  step 6                  step 11
T  AAATCGATTGAA         T  AAATCGATTGAA         T  AAATCGATTGAA
P  A                    P     GCTA              P          GCTA
B  900000000000         B  993311100000         B  999993202110
D  099999999999         D  993202099999         D  999933112110
V  000000000000         V  933111000000         V  999932021109
R  000000000000         R  333202000000         R  333210122110

step 1                  step 7                  step 12
T  AAATCGATTGAA         T  AAATCGATTGAA         T  AAATCGATTGAA
P  TA                   P      GCTA             P            GCT
B  909999999999         B  999320209999         B  999993311211
D  100000000000         D  993311110000         D  999993202121
V  099999999999         V  993202099999         V  999933112110
R  100000000000         R  333211110000         R  333210122121

step 2                  step 8                  step 13
T  AAATCGATTGAA         T  AAATCGATTGAA         T  AAATCGATTGAA
P  CTA                  P       GCTA            P             GC
B  910000000000         B  999331111000         B  999999320212
D  210999999999         D  999320201999         D  999993311222
V  100000000000         V  993311110000         V  999993202121
R  210000000000         R  333210201000         R  333210122122

step 3                  step 9                  step 14
T  AAATCGATTGAA         T  AAATCGATTGAA         T  AAATCGATTGAA
P  GCTA                 P        GCTA           P              G
B  921099999999         B  999932020199         B  999999331122
D  321100000000         D  999331110100         D  999999320213
V  210999999999         V  999320201999         V  999993311222
R  321100000000         R  333210110100         R  333210122123

step 4                  step 10                  step 15
T  AAATCGATTGAA         T  AAATCGATTGAA         T  AAATCGATTGAA
P   GCTA                P         GCTA          P
B  932110000000         B  999933111010         B  999999932021
D  932019999999         D  999932021109         D  999999331122
V  321100000000         V  999331110100         V  999999320213
R  332010000000         R  333210121100         R  333210122123

step 5
T  AAATCGATTGAA
P    GCTA
B  993201999999
D  933111000000
V  932019999999
R  333111000000
```

Figure 12 :

Simulation program output showing the pages contents during
the distance calculation of example in figure 6.

The R page (result) at step 15 contains the distances.
Page B has been initialized at maximum 9, only for display esthetic

VI - Genome specificities

We present in that chapter some improvements of the distance calculation algorithm dedicated to the genome applications. We shall extend the text length - which was limited so far to 1024 characters. The distances were 8-bit integers but can be extended to 16-bit integers. Specific genome formulas can be adapted too, such as bases dependent penalty or Dayhoff matrix [Needleman 70].

Distance calculation on extended texts

Running the distance calculation as in chapter V requires only 4 pages in the memory : one for the text and 3 for values P, B and R. Values D can be coded in one auxiliary page since these values are not needed after steps computation.

It is possible to declare 16 containers of 4 pages, thus the circuit can contain a 16,384 character text. The distance calculation algorithm is run successively on each container :

- at the beginning, the pattern is contained in the first container, therefore the algorithm is only run in the first container
- after a number of right-shifts, the pattern overlaps the first and the second container, then the distance must be calculated on both containers. Each pattern characters (and values) shifted at the end of the first container pages must be introduced at the beginning of the next container pages. This wrapping problem is solved by using the mesh management unit in the arbiter (see figure 4) which is able to memorize the content of the last memory cell (shifted via the mesh) and to feed back the first cell with that content via the first cell mesh.
- depending on the pattern length, the pattern will successively overlap a number of containers, then the algorithm will have to be run on every container containing the pattern.
- at the end of computation, the pattern is contained in the last container only, and the algorithm is only run on that container.

A typical genome application would be to calculate the distance between a 16,384 characters text and a 16,384 characters pattern. A memory page contains 1024 characters.

- During 1024 first right-shifts, the pattern only covers the first container, so 1024 steps are necessary.
- During the 1024 following right-shifts, the pattern covers both first and second containers, 2048 steps are then necessary: 1024 on the first container, and 1024 on the second.
- We then need 3072; 4086; 5110; ...; 16384; 15360; 14036; ...; 2048; and 1024 steps

Actually we have a double arithmetic series which sum is 17*16*1024 = 278,528 steps.
The distance calculation requires then 9.3 ms.

Distance calculation on very long texts

When the text is longer than 16,384 characters, the only solution is to load successive parts of the text in the memory and to run a distance calculation upon each part. The problem of pattern overlapping is solved by memorizing (in an outside memory) the shifted values output from the last cell. When shifting the pages of following part, the first cell values are initialized with the memorized values. Figure 9 clearly shows that the only pages which need to be right-shifted are pages P and R. Page P is easily fed by the user since he *knows* the pattern. Page R has to be memorized as explained afterwards.

Figure 13 shows an example of such a method : we calculate the distances of figure 6 example by first loading a 6 characters part, and then loading the 6 following characters (as though the page size was 6 characters).

Part 1, the user memorizes the right-shift output :

```
                          A A A T C G
                        A 0 0 0 1 1 1 → value 1 is output
                        T 1 1 1 0 1 2 → value 2 is output
                        C 2 2 2 1 0 1 → value 1 is output
                        G 3 3 3 2 1 0 → value 0 is output
```

Part 2, the user feeds back the R value to make the first column calculation valid :

```
                          A T T G A A
value 1 is fed back → A 0 1 1 1 0 0
value 2 is fed back → T 1 0 0 1 1 1
value 1 is fed back → C 2 1 1 1 2 2
value 0 is fed back → G 1 2 2 1 2 3
```

Figure 13 : Distance calculation on two text parts

A typical genome application would be to calculate the distance between a 8 million characters text and a 8 million characters pattern [Gonet 92]. The user must then divide the text in $8,000,000/16,384 = 489$ parts. Loading a part requires less than 16,384 cycles (i.e. 0.6 ms). Calculating the distances for this part requires 128 seconds (i.e. 128 million steps, as calculated in previous section, the pattern being present on all 16 containers during about 8 million steps). Reading the result distances requires around 4096 cycles (i.e. 0.1 ms). Therefore the complete application requires about $489 * 128\ s = 18$ hours, provided the host computer throughput is sufficient to feed Rapid-2 with instructions and data.

Complex penalty

The distance is calculated by adding the penalties encountered while shifting the pattern. The penalty presented at chapter V consists in incrementing the distance, however some genome application need some more complex penalty formula.

A general formula could be [Needleman 70] : $distance = a.g + b.\ell + \Sigma\ f(\varphi)$

a is an extra penalty for each gap beginning (a gap is a connected mismatches zone)
g is the number of encountered gaps
b is the gap penalty
l is the gap length
φ is the gap phase, i.e. the character absolute position modulo 3
f is a function defining the penalty depending on the phase

The chapter V distance calculation correspond the configuration $a = 0; b = 1; f() = 0$.

This formula implies some modifications in the implementation :

- for each mismatch ($P \neq T$), constant b must be added (instead of the former incrementation)
- for each mismatch following an equality ($P=T$), constant a must be added to the distance. In the circuit, this situation can be detected by saving the token state in the tag of a page, so that each token can be compared with its former state.
- for each mismatch, the phase $f()$ must be added to the distance. Function $f()$ shall be loaded in a special F page, each F page character codes the f value corresponding to its position. For each mismatch, value in page F (at the same position) is added to the distance.

Figure 14 lists the step program for Rapid-2. The step requires 41 machine cycles - result based on a timing diagram similar to the figure 11. Therefore, with such a distance formula, all previous time estimations must be increased of about 37%.

```
// B is transferred to D
ALTER-TOKEN (op = set)
COPY (source = page B ; target = page D ; shift = 0 ; byte = 0)
// R is shifted and transferred to B, P is right shifted
OPERATION (source = page R ; target = page B ; op = rr8 ; byte = 1)
COPY (source = page B ; target = page B ; shift = right ; byte = 1)
ALTER-TOKEN (op = set)
OPERATION (source = page P ; target = page P ; op = rr8 ; byte = 1)
COPY (source = page P ; target = page P ; shift = right ; byte = 1)
// minimum calculation
ALTER-TOKEN (op = copy flag to token ; page = page P)
OPERATION (source1 = page B ; source2 = page R ; op = less_than ; byte = 1)
COPY (source = page B ; target = page R ; shift = 0 ; byte =1)
ALTER-TOKEN (op = copy flag to token ; page = page P)
OPERATION (source1 = page D ; source2 = page R ; op = less_than ; byte = 1)
COPY (source = page D ; target = page R ; shift = 0 ; byte =1)
// additions, in each cell, auxiliary register WORK0 contains b, WORK1 contains a
ALTER-TOKEN (op = copy flag to token ; page = page P)
OPERATION (source1 = page P ; source2 = page T ; op = not_equal ; byte = 1)
OPERATION (source1 = page R ; source2 = reg WORK0; target = page R ; op = add ; byte = 1)
OPERATION (source1 = page R ; source2 = reg page F; target = page R ; op = add ; byte = 1)
TAG_SELECT (page = page T ; field = flag ; value = 1 ; op = 1SO1 ; UFHC = 0 ; Single-Word)
OPERATION (source1 = page R ; source2 = reg WORK1; target = page R ; op = add ; byte = 1)
// save mismatches for gap beginning operations
OPERATION (source1 = page P ; source2 = page T ; op = not_equal ; byte = 1)
ALTER-TOKEN (op = copy token to  flag; page = page T)
```

Figure 14 : Systolic step instructions for complex distance calculation

Bases dependent penalty

The base dependent penalty is an improvement of the distance formula specifying that b depends on the characters connected to the mismatch. The b value depends on character in the text (T) and character in the pattern (P). Values are coded in a matrix, genome applications typically need a 20 by 20 triangular matrix, called Dayhoff matrix [Gonnet 92].

In Rapid-2 implementation, modifications are implied in figure 14 listing, at line :

```
OPERATION (source1 = page R ; source2 = reg WORK0; target = page R ; op = add ; byte = 1)
```

This must be replaced by an access to the correct matrix value, and the addition of this value in page R. The only way to perform a parallel access to the matrix is to store it entirely in each cell. Such a redundancy requires 50 pages in the memory. The bidimensional matrix $\mathcal{M}(c,t)$ is stored as a monodimensional string $\mathcal{N}(x)$, we have the formula :

$$\mathcal{M}(c,t) = \mathcal{N}(c + (t-1) + (t-2) + \ldots + 2 + 1) = \mathcal{N}(c + t.(t-1)/2) = \mathcal{N}(c + (t^2 + t)/2)$$

Thus, for each mismatch between character c in the pattern and t int the text, the value to be added is the byte number $c + (t^2 + t)/2$ in the cell's zone reserved for the matrix. As t is a constant for each position, value $t^2 + t$ shall be stored in a special page.

The program for Rapid-2 has been written but not yet simulated. As there is one matrix in each cell, and four character in a cell, the execution time will be 4 times longer. Furthermore, only three containers can be loaded in the memory (see *Distance calculation on extented texts* section).

16 bits arithmetics

For higher precision distances, it can be worth coding the values on 16 bits rather than 8. The circuit is not designed to perform 16 bits operations in parallel. Therefore, operations will require 2 cycles. A new storing configuration is needed :

- page T : one page
- page P : one page
- page B : two pages, B1 for the most significant byte and B0 for the least significant byte
- page D : two pages, D1 for the most significant byte and D0 for the least significant byte
- page R : two pages, R1 for the most significant byte and R0 for the least significant byte
- matrix : if used, it will be stored as specified before, values still being stored as 8-bit integers

Modifications in the algorithm implementation : all the algorithms seen before use elementary operations which are transfer, equality test, inequality test, and addition. We list here the differences between a 16-bit implementation and a 8-bit implementation :

Transfer : 2 pages have to be transferred instead of 1, i.e. 2 instructions COPY instead of 1

Inequality : Testing the 16 bits relation $\mathcal{A}<\mathcal{B}$ with 8 bits comparators implies the formula

$$[\ msb(\mathcal{A}) < msb(\mathcal{B})\ ]\ or\ \{\ [\ msb(\mathcal{A}) = msb(\mathcal{B})\ ]\ and\ [\ lsb(\mathcal{A}) < lsb(\mathcal{B})\ ]\ \}$$

the complexity is multiplied by 3, separated by AND and OR connectors that can be run with instruction SELECT.

Equality : $\mathcal{A} = \mathcal{B}$ when $[\ msb(\mathcal{A}) = msb(\mathcal{B})\ ]\ and\ [\ lsb(\mathcal{A}) = lsb(\mathcal{B})\ ]$

then the complexity is multiplied by 2.

Addition : operation $\mathcal{C} = \mathcal{A} + \mathcal{B}$, with $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ being 16 bits integers, becomes :

$lsb(\mathcal{C}) = lsb(\mathcal{A}) + lsb(\mathcal{B})$; carry → token

if token is set, $msb(\mathcal{C})$ is incremented, otherwise $msb(\mathcal{C})=0$

$msb(\mathcal{C})$ += $msb(\mathcal{A})$; $msb(\mathcal{C})$ += $msb(\mathcal{B})$

complexity is multiplied by 4.

The 16 bits step has been programmed for Rapid-2 with the complex distance formula, figure 15 lists the instructions of the step. The time diagram of the step instructions gives an estimation time of 70 cycles (i.e. 2.3 µs). This increases execution time by 70%.

```
// B is transferred to D
ALTER-TOKEN (op = set)
COPY (source = page B0 ; target = page D0 ; shift = 0 ; byte = 0)
COPY (source = page B1 ; target = page D1 ; shift = 0 ; byte = 0)
// R is shifted and transferred to B
OPERATION (source = page R0 ; target = page B0 ; op = rr8 ; byte = 1)
OPERATION (source = page R1 ; target = page B1 ; op = rr8 ; byte = 1)
COPY (source = page B0 ; target = page B0 ; shift = right ; byte = 1)
COPY (source = page B1 ; target = page B1 ; shift = right ; byte = 1)
// P is right-shifted
ALTER-TOKEN (op = set)
OPERATION (source = page P ; target = page P ; op = rr8 ; byte = 1)
COPY (source = page P ; target = page P ; shift = right ; byte = 1)
// minimum calculation
ALTER-TOKEN (op = copy flag to token ; page = page P)
OPERATION (source1 = page B0 ; source2 = page R0 ; op = less_than ; connect = AND)
OPERATION (source1 = page B1 ; source2 = page R1 ; op = equal ; connect = AND)
OPERATION (source1 = page B1 ; source2 = page R1 ; op = less_than ; connect = OR)
COPY (source = page B ; target = page R ; shift = 0 ; byte =1)
ALTER-TOKEN (op = copy flag to token ; page = page P)
OPERATION (source1 = page D0 ; source2 = page R0 ; op = less_than ; connect = AND)
OPERATION (source1 = page D1 ; source2 = page R1 ; op = equal ; connect = AND)
OPERATION (source1 = page D1 ; source2 = page R1 ; op = less_than ; connect = OR)
COPY (source = page D ; target = page R ; shift = 0 ; byte =1)
// additions
ALTER-TOKEN (op = copy flag to token ; page = page P)
OPERATION (source1 = page P ; source2 = page T ; op = equal ; byte = 1)
ALTER-TOKEN (op = copy token to  flag; page = page R0)  // temporary save the tokens
// addition 16 bits + 8 bits : add lsb, increments msb if ovrf; WORK3 must contain 1
OPERATION (src1 = page R0 ; src2 = reg WORK0; tgt = page R0 ; op = add)
TAG_SELECT (page = page R0 ; field = flag_ovrf ; op = 1SO1 ; UFHC = 0 ; SINGLE-WORD)
OPERATION (src1 = page R1 ; src2 = reg WORK3; tgt = page R1 ; op = add)
ALTER-TOKEN (op = copy flag to token ; page = page R0)
OPERATION (src1 = page R0 ; src2 = reg WORK2; tgt = page R0 ; op = add)
TAG_SELECT (page = page R0 ; field = flag_ovrf ; op = 1SO1 ; UFHC = 0 ; SINGLE-WORD)
OPERATION (src1 = page R1 ; src2 = reg WORK3; tgt = page R1 ; op = add)
ALTER-TOKEN (op = copy flag to token ; page = page R0)
TAG_SELECT (page = page T ; field = flag ; value = 1 ; op = 1SO1 ; UFHC = 0 ; SINGLE-WORD)
OPERATION (src1 = page R0 ; src2 = reg WORK1; tgt = page R0 ; op = add)
```