Parallel evaluation of arithmetic circuits

Nathalie REVOL and Jean-Louis ROCH

LMC-IMAG, 100 rue des Mathématiques, BP 53X, 38041 Grenoble Cedex, FRANCE

Nathalie.Revol@imag.fr, Jean-Louis.Roch@imag.fr

In this paper, a generic algorithm designed for the parallel evaluation of arithmetic circuits is given. This algorithm can be used in the domain of VLSI design, in order to get tight upper bounds on the computing time of a circuit. It can also be used in automatic parallelization of numerical programs, as a guide for the detection of some predefinite schemes such as dot-products or reductions. More generally, the (theoretical) algorithm presented in section 2 evaluates very quickly arithmetic straight-line programs, and its evaluation time serves as a good upper bound. This algorithm generalizes Miller, Ramachandran and Kaltofen's algorithm [18] in the sense it deals with a great variety of algebraic structures: semi-rings, rings or lattices. Our contribution resides on the one hand in a new bound for the evaluation of circuits over lattices, which improves previous results [19], and on the other hand in the unified formulation for the evaluation algorithm. This algorithm runs in $\mathcal{O}(\min(\log n + \log d) \log n, (h_a + \log n) \log n))$ parallel time, d being the "algebraic degree" (in an extended sense) of the circuit and h_a the maximal number of alternances of $\oplus$ and $\otimes$ on a path of the circuit if the $\oplus$ and $\otimes$ operations define a lattice, with $M(n)$ processors, where $M(n)$ is the number of processors necessary for the multiplication of two $n \times n$ matrices in the structure in $\mathcal{O}(\log n)$ parallel time. After presenting this algorithm, its efficiency is shown on particular cases: taking as input a simple and sequential algorithm, it can be used as a "compiler" to produce a sorting circuit as fast as Cole's circuit, with logarithmic depth, or an adder equivalent to Brent and Kung's adder in terms of size and depth. These academic examples confirm the relevance of the algorithm presented here in the area of conception of fast VLSI arithmetic operators.

1 Introduction

In the domain of VLSI design, two questions naturally arise: the first one is to design a circuit computing the solution of a given problem, the second one is

Preprint submitted to Elsevier Preprint *16 April 1996*

to determine whether this circuit can be improved. It appears that in this area, the computation time of a circuit is an important criterion. Our works allows one to derive tight upper bounds on the computation time of a multi-valued boolean function.

Actually, in order to measure the quality of a VLSI circuit, two measures are used (cf. [16]): the first one is A, the area of the chip surface that is taken up by the electronic components devoted to the considered computation; the second one is the time, T, which represents the number of clock cycles spent in a computation. It is assumed that the components of the circuit are totally synchronous. Even if this assumption is not very realistic, it is interesting since it permits to derive tight upper bounds on the complexity of actual circuits. These two quantities can be combined in order to get a new measure of the quality of a circuit, which exhibits the trade-off between the area A and the time T. It appears that with respect to the quantity AT, circuits are optimal that are not considered to be good VLSI circuits; eventually, the quantity which is widely used is AT^2. Thus it is particularly interesting to get good upper bounds on the time T, both in terms of performance (for the user of the circuit) and of hardware cost (AT^2 cost).

Since our works focuses on T, an apparent limitation is that the quantity A is often overestimated. It appears that there exists circuits achieving our bounds on T with reasonable A; for instance, we derive automatically a logarithmic time bound from the boolean equations modeling the addition of two n-bit integers, and it is well-known that Brent and Kung [3] have proposed an adder with logarithmic time and small (linear) area. (This formed a test case for our work). Another apparent weakness of our result is that it gives asymptotic bounds on T. However, the constants are small (between 1 and 2). Moreover, our algorithm takes benefit of the algebraic properties of the boolean operations in order to derive the upper bounds, and in fact the result indicates for instance whether the associativity should be used in order to reorganize the computations or the commutativity could help more fruitfully.

Briefly stated, our work takes as input boolean equations describing the computations and describes how they can be evaluated quickly in parallel. The time of this evaluation is very often a tight upper bound on the time needed by a circuit to perform these computations; the way the boolean operations are performed often indicates how to design a real VLSI circuit achieving this time complexity with a reasonable area A. In the last section of this paper, some upper bounds for already known problems illustrate this point.

To replace this in a more general framework, we consider the problem of the evaluation of arithmetic circuits when the $+$ and $\times$ operations define various algebraic structures such as semi-rings, rings or lattices. The boolean algebra, which is the basic algebraic structure used for VLSI design, constitutes in fact a particular application. The term of "circuit" will be used in a more general meaning than the VLSI one; actually the parallel complexity theory is based on the notion of – uniform – boolean [2,7,21] and arithmetic [9] circuits, also called straight-line programs. We present in this paper a generic algorithm for the parallel evaluation of arithmetic circuits when the underlying algebraic structure is commutative and is either a semi-ring, a ring or a lattice. Our first contribution is a unified presentation of known and new algorithms designed for each case.

The boolean circuits constitute a theoretical model for parallel computations and the complexity classes NC^k, NC are defined upon this model [7]. Thus, any parallel computation is equivalent to the parallel evaluation of the corresponding boolean circuit. Since the boolean algebra $(\{True, False\}, \vee, \wedge)$ is a lattice, algebraic properties can be taken into account in order to speed up the parallel evaluation of boolean circuits. Our other contribution consists in a new algorithm for the parallel evaluation of lattice circuits; its complexity improves the best known results [19]. We introduce a new measure, the maximal number of alternances, to which the complexity is related. It has to be noticed that Ladner has shown that the boolean circuit evaluation problem is $\mathcal{P}$-complete.

In the framework of parallel evaluation, two cases can be distinguished: expressions and circuits. An expression is a formula where every variable and every intermediate result can serve only once as an operand. It can be represented as a tree, and optimal algorithms exist with a $EREW_S(\frac{n}{\log n}, \log n)$ complexity [1] [1,4,10,15,17], where n is the number of nodes in this tree. Note that this problem is NC^1-complete [14].

The evaluation of an arithmetic circuit with operations in a commutative semi-ring $(SR, +, \times, 0, 1)$ can be done by Miller, Ramachandran and Kaltofen's algorithm [2] [18]. It has a complexity of $CREW_{SR}(M(n), \log n(\log n + \log d))$, where d denotes the arithmetic degree of the circuit and n the number of nodes. It consists in applying repeatedly a sequence of three procedures. The first one groups two successive + nodes into one, the second evaluates + nodes having their operands evaluated, and the last one evaluates or shunts $\times$ nodes

[1] The notation $EREW_S(nb\,of\,proc, time)$ is due to Karp and Ramachandran [14]; it means that the computation time of the algorithm on a EREW (Exclusive Read Exclusive Write) PRAM is $\mathcal{O}(time)$, where an operation on the structure S is done in one unit of time, while the number of processors required to achieve this time is $\mathcal{O}(nb\,of\,proc)$. The notation $CREW_S(nb\,of\,proc, time)$ just differs on the parallel machine performing the computation: it is a CREW (Concurrent Read Exclusive Write) PRAM. Similarly, when the parallel machine is a CRCW (Concurrent Read Concurrent Write) PRAM, the notation $CRCW_S(nbofproc, time)$ will be employed. For this last case, it does not matter which kind of CRCW-PRAM is used.

[2] In the following, Miller, Ramachandran and Kaltofen will be abbreviated in MRK.

(the **shunt** is the equivalent of the **rake** of Kosaraju and Delcher [15], or of the **prune** of Cole and Vishkin [4]).

A lattice $(L, \oplus, \otimes)$ is a set L in which two internal operations, $\oplus$ and $\otimes$, are commutative and associative and satisfy the absorption law: $\forall a, b \in L$, $(a \oplus b) \otimes a = (a \otimes b) \oplus a = a$. In this paper we shall restrict the work to distributive lattices. Miller and Teng [19] proposed two algorithms for the contraction of distributed lattice circuits. They are based on MRK's algorithm [18], designed for circuits with operations in a semi-ring. The first one consists in two simultaneous executions of this algorithm, one considering $\oplus$ as the addition and $\otimes$ as the multiplication, and the other inverting the roles of $\oplus$ and $\otimes$. The first having completed its computation stops the second. The other algorithm consists in applying the basic procedure of contraction twice at each step, the first one with $\otimes$ as multiplication, the second with $\oplus$ as multiplication.
If $d_\oplus$ stands for the arithmetic degree of the circuit when $\oplus$ represents the addition and $\otimes$ the multiplication, and $d_\otimes$ is the degree when the roles of $\oplus$ and $\otimes$ are inverted, d being the minimum of $d_\oplus$ and $d_\otimes$, and if n is the number of nodes in the circuit, then the complexity of Miller and Teng's algorithms is :

$$CREW_L(M(n), \log n(\log n + \log d)).$$

The main benefit of Miller and Teng's algorithms is their simplicity, since you just have to run MRK's algorithm twice. However, this simplicity is counterbalanced by a loss of performance. Actually, its major drawback comes from the difference of treatment between $\oplus$ and $\otimes$, whereas in a lattice they have exactly the same properties.

The algorithm presented in this paper (section 2) is a generalization of MRK's algorithm; on the one hand, it is designed to handle algebraic structures of different kinds; on the other hand, it fully exploits the symmetric properties of $\oplus$ and $\otimes$ in the lattice case. It is composed of four procedures. The first operation, called **Group**, groups $+$ nodes by means of matrix multiplications, and it groups both $\oplus$ and $\otimes$ nodes in the lattice case. The second one, **Eval**, evaluates $+$ (resp. $\oplus$) nodes as well $\times$ (resp. $\otimes$) nodes having their operands already evaluated. The third one is a partial evaluation of the nodes having some of their operands evaluated; its name is **PartialEval**. A generalization of the **Shunt** is then performed on $\times$ nodes, or on $\oplus$ and $\otimes$ nodes in the lattice case, suppressing chains of unary $\times$ (resp. $\oplus$ and $\otimes$) nodes; thus it is called **Suppress**.

This algorithm can be modulated in order to be a simple extension of the tree contraction technique, to correspond to MRK's algorithm for circuits over semi-rings, or to be a very efficient algorithm in the lattice case. Its complexity is thus the same as MRK's complexity in the semi-ring case.

For the lattice case, this algorithm has a first complexity upper bound of $CREW_L(M(n), \log n(\log n + \log d))$, which means that it is (at least) as efficient as Miller and Teng's algorithms. Another upper bound is $CREW_L(M(n), (h_a + \log n)\log n)$, where h_a is the maximal number of alternances of $\oplus$ and $\otimes$ on any path between a value node and a result node in the circuit. Proofs of these bounds are to be found in section 3.

In section 4, some applications of this algorithm are presented. These examples are classical test problems, in order to check if the evaluation algorithm achieves good performances on well-known problems. The first one illustrates the power of this algorithm as complexity predictor: it gives a $\mathcal{O}(\log n)$ time complexity on a $CRCW-PRAM$ for the sort, when given as input the insertion sort algorithm. The second problem is the addition and the multiplication of two n-bit numbers: the practical complexity matches the theoretical one, since the experimental time is logarithmic; this algorithm used as a compiler produces in this case an adder of linear width and logarithmic depth starting from the boolean equations of the addition, this means that it produces automatically an adder equivalent to Brent and Kung's adder [3]. Lastly, the limits of our algorithm are given: for the $\mathcal{P}$-complete lexicographic maximal independent set problem [6], it evaluates the corresponding circuit in linear parallel time. Since the evaluation algorithm gives satisfying results on these problems, some real applications are considered as future work.

2 Algorithm

2.1 Definitions and notations

A commutative semi-ring $(SR, +, \times, 0, 1)$ is a set SR in which two internal operations, $+$ and $\times$, are associative and commutative, have a unit element (0 for $+$ and 1 for $\times$) and $\times$ is distributive with respect to $+$. In a commutative ring, every element has an inverse for $+$.

A lattice $(L, \oplus, \otimes)$ is a set L in which two internal operations, $\oplus$ and $\otimes$, are commutative and associative and satisfy the absorption law: $\forall a, b \in L, (a \oplus b) \otimes a = (a \otimes b) \oplus a = a$. From the absorption law, we can deduce the idempotency of $\oplus$ and $\otimes$: $\forall a \in L,\ a \oplus a = a \otimes a = a$. In this paper, we will restrict the work in two directions: on the one hand, we consider only distributive lattices, *i.e.* lattices where $\otimes$ is distributive with respect to $\oplus$, which implies that $\oplus$ is also distributive with respect to $\otimes$; on the other hand, we limit ourselves to lattices with a greatest element e (a unit element for $\otimes$) and a smallest element ε (a unit element for $\oplus$). Actually, this second assumption is not restrictive, since it is possible to add dummy e and ε elements to L; we also have $\forall x \in L$, $e \oplus x = e$ and $\varepsilon \otimes x = \varepsilon$ because of the absorption law.

Notations: in what follows, S stands for an arbitrary algebraic structure, whereas L stands for a lattice, and $+$ and $\times$ represent operations from a semi-ring or a ring, $\oplus$ and $\otimes$ stand for lattice operations.

In each of these structures, an arithmetic operation (either an addition or a multiplication) is assumed to be performed in unit time. In the case of a totally ordered lattice, it happens that the addition or multiplication of n elements can be performed in constant time with $\mathcal{O}(n^2)$ processors on a $CRCW-PRAM$.

In the following, arithmetic circuits are represented by DAGs. The vertices are labeled as leaves, $+$ or $\times$ nodes, or $\oplus$ or $\otimes$ nodes. The out-degree of leaves is 0, the out-degree of $+$, $\oplus$ and $\otimes$ nodes is ≥ 1 (operations of variable arity), the out-degree of $\times$ nodes is ≤ 2. The edges are directed top-down, from the operator to the operands; they are weighted with $(a \times x)$-like linear functions or $(a \otimes x \oplus b)$-like affine functions.

Notations: v and w denote nodes of the DAG, usually with w representing any child of v. The adjacency matrix associated to the DAG is denoted by U; its coefficients represent the linear or affine functions weighting the edges: if the function is linear, it is simply represented as a single coefficient whereas an affine function is represented as a pair of coefficients.

The four basic operations of the contraction algorithm are detailed in the following paragraphs.

2.2 Parallel evaluation algorithm

Eval

The most obvious procedure is **Eval**: when a node knows the value of all its operands, it computes a value and becomes a leaf, *i.e.* it disconnects itself from its children.

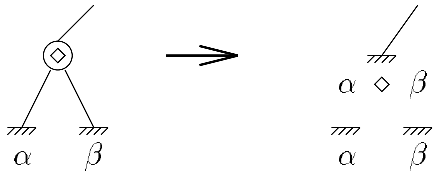

Fig. 1. *The* **Eval** *procedure.*

This can be done with $CREW_S(n^2, \log n)$ complexity and $CRCW_L(n^2, 1)$ complexity in a totally ordered lattice.

Group

We generalize the "MM" operation of MRK's algorithm. In the latter, they use only linear functions on the edges, and they group $+$ operations by matrix products (which correspond to one step of a transitive closure computation), so as to transform two successive $+$ nodes into a single one, and to compute the result of n-ary $+$.

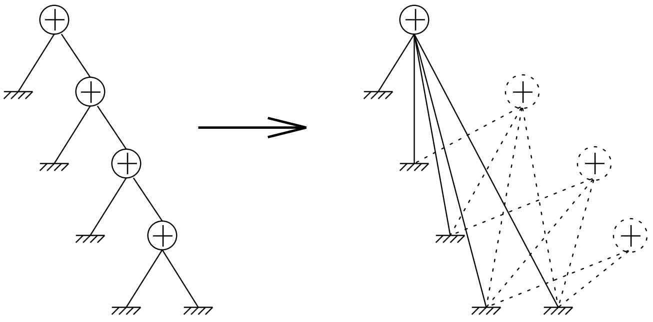

Fig. 2. *MRK's* **Group**.

They define two matrices from the adjacency matrix:

- U^{++} is the matrix of weights of the edges between two $+$ nodes,
- $U^{+\cdot}$ is the matrix of weights of the edges from a $+$ node to a $\times$ node or to a leaf.

MRK's "MM" operation can then be defined as

$$\begin{cases} & U^{+\cdot} \longleftarrow U^{++}.U^{+\cdot} + U^{+\cdot} \\ \text{followed by} & U^{++} \longleftarrow U^{++}.U^{++} \end{cases}$$

As far as lattices are concerned, since $\oplus$ and $\otimes$ are symmetric, $\otimes$ nodes are grouped as well as $\oplus$, and thus we define four auxiliary matrices from the adjacency matrix U:

- $U^{\oplus\oplus}$ (resp. $U^{\otimes\otimes}$) is the matrix of weights of the edges between two $\oplus$ (resp. $\otimes$) nodes,
- $U^{\oplus\cdot}$ (resp. $U^{\otimes\cdot}$) is the matrix of weights of the edges from a $\oplus$ (resp. $\otimes$) node to a $\otimes$ (resp. $\oplus$) node or to a leaf.

The grouping operation can be written as matrices operations:

$$\begin{cases} & U^{\oplus\cdot} \longleftarrow U^{\oplus\oplus}.U^{\oplus\cdot} + U^{\oplus\cdot} \\ \text{followed by} & U^{\oplus\oplus} \longleftarrow U^{\oplus\oplus}.U^{\oplus\oplus} \end{cases}$$

and

$$\begin{cases} & U^{\otimes\cdot} \longleftarrow U^{\otimes\otimes}.U^{\otimes\cdot} + U^{\otimes\cdot} \\ \text{followed by} & U^{\otimes\otimes} \longleftarrow U^{\otimes\otimes}.U^{\otimes\otimes} \end{cases}$$

Since the coefficients of the matrices represent affine functions, usual matrix products can not be used and have to be slightly modified:

$$(A.B)_{i,j} = \bigoplus_{k=1}^{n} A_{i,k}(x) \circ B_{k,j}(x)$$

when $\oplus$ nodes are grouped, and

$$(A.B)_{i,j} = \bigotimes_{k=1}^{n} A_{i,k}(x) \circ B_{k,j}(x)$$

when $\otimes$ nodes are grouped.
The **Group** procedure has the same complexity as a matrix product:

$$CREW_S(M(n), \log n),$$

where $M(n)$ is the minimal number of processors required to multiply two $n \times n$ matrices in time $\mathcal{O}(\log n)$, and $CRCW_L(M'(n), 1)$ in a totally ordered lattice, where $M'(n)$ is the minimal number of processors required to multiply two $n \times n$ matrices in time $\mathcal{O}(1)$. $M(n)$ is roughly bounded by $\mathcal{O}(n^3)$, and $M'(n)$ by $\mathcal{O}(n^4)$.

PartialEval

This procedure and the following one form a generalization of MRK's **Shunt** of $\times$ nodes; they allow one to shunt both $\oplus$ and $\otimes$ nodes.

Any node having leaves children computes its partial result and puts the result on one (or any in the lattice case[3]) edge to a non-leaf child if one exists; otherwise the node keeps only one child (a leaf) and puts its value on the edge between them.

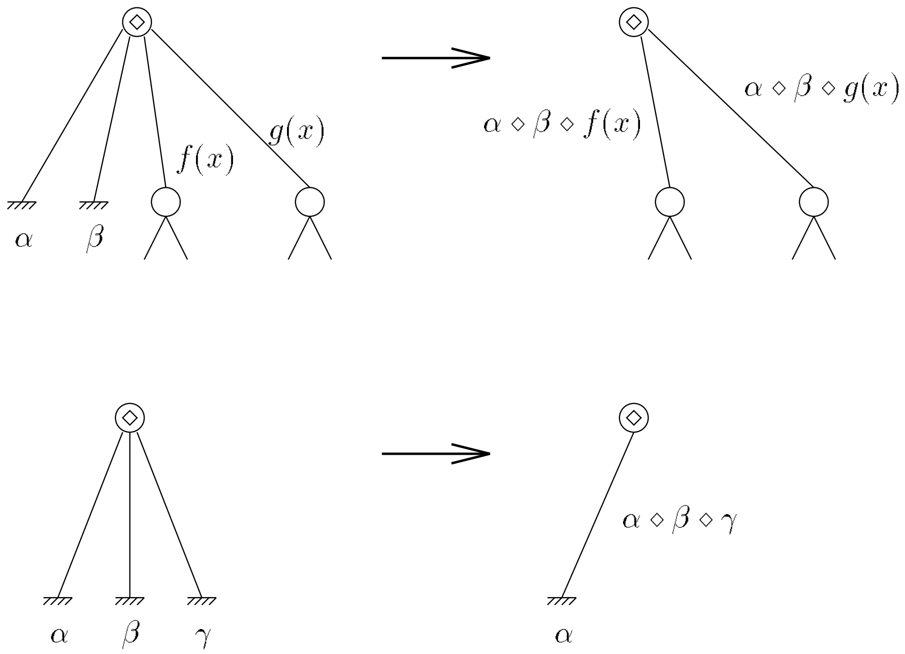

Fig. 3. *The partial evaluation procedure.*

This procedure has a $CREW_S(n^2, \log n)$ complexity, and a $CRCW_L(n^2, 1)$ complexity in a totally ordered lattice.

Suppress

The previous procedure may have created unary nodes. We now "compress" the chains of unary nodes (only the $\times$ nodes in the semi-ring case, both $\oplus$ and $\otimes$ nodes in the lattice case). A pointer-jumping technique is used; it consists in repeating the following process until no node has a unary child: each node which has a unary child disconnects itself from this child and connects to the only grand-child originated from the unary child. For the whole evaluation of the DAG, the total cost of **Suppress** is $CREW_S(n, \log n)$ and $CRCW_S(n, \log n)$.

The combination of **PartialEval** and **Suppress** corresponds to the **Shunt** procedure of MRK.

[3] using the idempotency of $\oplus$ and $\otimes$.

Algorithm

The main procedure of the evaluation algorithm consists in a preprocessing and then in applying the "**Group - Eval - PartialEval - Suppress**" sequence as many times as required.

Algorithm 1 (*DAG evaluation*)

Preprocessing

Group**: do* $\lceil \log n \rceil$ *times* **Group**

Suppress

repeat **Phase**

Group

$\oplus$ *nodes*

$\otimes$ *nodes*

Eval

all nodes in parallel

PartialEval

all nodes in parallel

Suppress

every chain of unary nodes

until every node is evaluated.

end **Phase**

In the next section, an upper bound of the number of applications of **Phase** is given.

Remark
For the CREW version of this algorithm, the read/write protocol is the following: we work with two copies of the DAG, an old copy used to read the old values and a new, modified copy, so as to avoid write conflicts; the nodes and the adjacency matrices are then updated. This allows to perform in parallel the matrices products, using old matrices and then updating them.

3 Complexity

In this section, an upper bound of the complexity of the previous algorithm is given. In the lattice case, it is split into three parts. The first one involves an algebraic measure, depending on the function computed by the DAG: $h_\oplus$

corresponds to the height defined by MRK, when $\oplus$ represents the addition and $\otimes$ the multiplication. The second one inverts the roles of $\oplus$ and $\otimes$ and involves $h_{\otimes}$, the corresponding height. For the last part of the proof, we introduce h_a, the maximal number of alternances of $\oplus$ and $\otimes$ on any path of the DAG. Using these measures, we put in evidence three different upper bounds. Two characteristic quantities have to be defined upon the DAG: the degree of a DAG and its number of alternances.

Definition 1 *Let's define $d_{\oplus}$, $d_{\otimes}$ and d as follows:*

$$d_{\oplus}(v) = \begin{cases} 1 & \textit{if } v \textit{ is a leaf,} \\ \sum_w d_{\oplus}(w \textit{ child of } v) & \textit{if } v \textit{ is a } \oplus \textit{ node,} \\ \max_w (d_{\oplus}(w \textit{ child of } v) & \textit{if } v \textit{ is a } \otimes \textit{ node,} \end{cases}$$

$d_{\oplus}$ of a DAG is the maximum of the $d_{\oplus}$ degrees of its nodes,

$$d_{\otimes}(v) = \begin{cases} 1 & \textit{if } v \textit{ is a leaf,} \\ \sum_w d_{\otimes}(w \textit{ child of } v) & \textit{if } v \textit{ is a } \otimes \textit{ node,} \\ \max_w (d_{\otimes}(w \textit{child of } v) & \textit{if } v \textit{ is a } \oplus \textit{ node,} \end{cases}$$

$d_{\otimes}$ of a DAG is the maximum of the $d_{\otimes}$ degrees of its nodes.

The degree d of a DAG is the minimum of its $d_{\oplus}$ and $d_{\otimes}$ degrees in the lattice case, $d = d_{\times}$ otherwise.

This notion of degree does not exactly meet the usual definition of "algebraic degree": a DAG computing $(x + x^2) - (x^2)$ computes the identity function, whose algebraic degree is equal to one, whereas the degree of the DAG is 2. This definition permits to manage such pathological cases; however, d can be thought as the usual degree in a first approach.

Definition 2 *We then define h_a as the maximal number of alternances on a path from an output node to a leaf of $\oplus$ and $\otimes$ nodes*[4] *.*

[4] This notion of alternance should not be confused with the notion of alternance defined for alternative Turing machine: in the latter context, the notion of alternance refers to the number of random choices made during a given execution.

More formally, $h_a(v)$ is equal to 1 if v is a leaf,

$$h_a(v) = \max \begin{pmatrix} h_a(w \otimes \ \textit{child of } v) + 1, \\ h_a(w \oplus \ \textit{child of } v), \\ h_a(w \textit{ leaf child of } v) + 1) \end{pmatrix}$$

if v is a $\oplus$ node, and

$$h_a(v) = \max \begin{pmatrix} h_a(w \oplus \ \textit{child of } v) + 1, \\ h_a(w \otimes \ \textit{child of } v), \\ h_a(w \textit{ leaf child of } v) + 1) \end{pmatrix}$$

if v is a $\otimes$ node. The number of alternances h_a of a circuit is the maximum of the h_a numbers of alternances of its nodes in the lattice case, $h_a = +\infty$ otherwise.

These two quantities permit to measure the parallel complexity of the DAG, as shown in the following theorem.

Theorem 1 *An upper bound for the complexity of algorithm 1 is :*

$$CREW_L(M(n),\ \log(n) * \min(\log(nd), h_a + \log(n)))$$

and

$$CRCW_L(M(n),\ \min(\log(nd), h_a + \log(n)))$$

Proof

An easy point to prove is the complexity of the procedure **Phase**: the complexity of the procedure **Phase** is $CREW(M(n), \log n)$ and $CRCW$ $(M'(n), 1)$ in a totally ordered lattice. □

The number of applications of the procedure **Phase** is a little bit more tricky to establish. First of all, an upper bound is established using a new quantity, the height of a DAG (this first upper bound is based on MRK's proof).

Let's define $h_\oplus$ as follows: v being a node of the DAG, let $h_\oplus(v)$ be 1 if v is a leaf,

$$h_\oplus(v) = \max_w \begin{pmatrix} h_\oplus(w \oplus \text{ child}) + \frac{1}{2}, \\ h_\oplus(w \otimes \text{ child}), \\ h_\oplus(w \text{ leaf child}) \end{pmatrix}$$

if v is a $\oplus$ node, and

$$h_\oplus(v) = \sum_w h_\oplus(w \text{ child})$$

if v is a $\otimes$ node.

A dominant child w of a $\oplus$ node v is a child such that $h_\oplus(v) = h_\oplus(w) + \frac{1}{2}$ if w is a $\oplus$ node, $h_\oplus(v) = h_\oplus(w)$ otherwise. $h_\otimes(v)$ is the height obtained by exchanging $\oplus$ and $\otimes$ in the previous definition.

The height $h_\oplus$ (resp. $h_\otimes$) of a DAG is the maximum of the heights of its nodes.

We assume that there is no unary node in the entry DAG (thanks to the preprocessing ending up with **Suppress**). Following MRK proof's scheme, let's show that $h_\oplus$ is divided by 2 by one application of **Phase**. Let's consider **Phase**, **Group**, **Eval**,... as maps of circuits to circuits. They are surjective on nodes, and modify only the edge structure. We denote by $U = (X, E)$ a circuit and by $U' = (X', E')$, with $X' = X$, its image by **Phase**, by v a node in X and by $v' \in X'$ its image, and by U_v and $U'_{v'}$ the subcircuits they induce (i.e. the subcircuits computing their value). w' will represent any child of v', and w its antecedent by **Phase**.

Let v be a $\otimes$ node having one $\oplus$ child w. If w', its image, is not a child of v', the only possibility is that w has become a unary node with child x after **PartialEval**, and has been suppressed by **Suppress**. Thus the $v - w$ edge has been replaced by only one edge $v' - x'$. From this point we deduce :

Corollary 1 *If v' is a $\otimes$ node, then its height is $h_\oplus(v') = \sum h_\oplus(w' \text{ child of } v')$ and the height of its antecedent is $h_\oplus(v) \geq \sum h_\oplus(w \text{ antecedent of } w')$.*

The same holds when $\oplus$ and $\otimes$ are exchanged.

Theorem 2 *If U and U' are arithmetic circuits, if v' is a node of U' which is neither a leaf nor an output node (a node without parents) and v is its antecedent, then $h_\oplus(v) \geq 2h_\oplus(v')$.*

Proof
Let's prove it by induction on the size of $U'_{v'}$ the subcircuit induced by v' in U'.

– Initialization. Let v' be neither a leaf nor an output node of U', and let v' have only leaves children.

If v' is a $\oplus$ node, $h_\oplus(v') = 1$. Let's prove that $h_\oplus(v) \geq 2$ by reducing it to the absurd. If $h_\oplus(v) < 2$, either $h_\oplus(v) = 1$ or $h_\oplus(v) = \frac{3}{2}$. If $h_\oplus(v) = 1$, since there is no unary node, v is either a leaf or a $\oplus$ node whose children are leaves. After **Eval**, v is a leaf, and thus v' is a leaf, which is opposite to the assumption. If $h_\oplus(v) = \frac{3}{2}$, then v is a $\oplus$ node whose children are either $\oplus$ nodes with only leaves children, or and possibly leaves. After **Group**, every child of v is a leaf, and after **Eval** v itself is also a leaf. Thus, v' is a leaf, and this is a contradiction. The case where v' is a $\oplus$ node is completed.

If v' is a $\otimes$ node, $h_\oplus(v') = \sharp(w'$ child of $v')$. It is enough to prove that every w antecedent of w' has a height ≥ 2, thanks to corollary 1. If $h_\oplus(w) < 2$, $h_\oplus(w) = 1$ or $h_\oplus(w) = \frac{3}{2}$. If $h_\oplus(w) = \frac{3}{2}$, then (cf previous case) after **Group** w is a $\oplus$ node having only leaves children. If $h_\oplus(w) = 1$, after **Group** w is also a $\oplus$ node having only leaves children. In both cases, after **Eval** w is a leaf. Since w' is a child of v', w must be the only child of v after **PartialEval**. Thus, v is a unary node and after **Suppress** it is disconnected from its parents. This means that v' is an output node, which is a contradiction. Hence, every w antecedent of w' child of v' has a height ≥ 2, and $h_\oplus(v) \geq \sum h_\oplus(w) \geq 2 \sum h_\oplus(w') = 2h_\oplus(v')$ (by corollary 1).

Our induction is correctly founded, let's treat the general case now.

– General Case. The induction hypothesis is that for any subcircuit U_w of size $\leq k$ (i.e. with a number of nodes $\leq k$), and such that w', the image of w, is neither a leaf nor an output node, the height of w is divided by (at least) 2 by **Phase**.

Let v be a node such that U_v is of size $k+1$, let's prove that $h_\oplus(v)$ is divided by 2 by **Phase**.

· If v is a $\oplus$ node, v' is also a $\oplus$ node. Let w' be a dominant child of v' and w its antecedent. If w' is a $\otimes$ node, then $h_\oplus(v') = h_\oplus(w') \leq \frac{1}{2}h_\oplus(w)$ by a straightforward application of the induction hypothesis, and since w is a descendant of v, $h_\oplus(w) \leq h_\oplus(v)$. Thus, $2h_\oplus(v') \leq h_\oplus(v)$.

If w' is a $\oplus$ node, we only need to show that there exists a path of length at least 2 between v and w. If such a path does not exist, then, v and w are adjacent, and **Group** is disconnecting them. Hence, we have $h_\oplus(w) + 1 \leq h_\oplus(v)$, and $h_\oplus(v') = h_\oplus(w') + \frac{1}{2} \leq \frac{1}{2}(h_\oplus(w) + 1) \leq \frac{1}{2}h_\oplus(v)$.

In conclusion, the property is true for $\oplus$ nodes.

· If v is a $\otimes$ node, v' is also a $\otimes$ node. $h_\oplus(v') = \sum h_\oplus(w'$ child of $v')$. Thanks to corollary 1, we only need to prove that for each w', $h_\oplus(w') \leq \frac{1}{2}h_\oplus(w)$. w' is not an output node since it has a parent v'. If w' is not a leaf, then, the induction hypothesis applies, and $h_\oplus(w') \leq \frac{1}{2}h_\oplus(w)$.

The only delicate case occurs when w' is a leaf. Let's show that $h_\oplus(w) \geq 2$. By a mean of contradiction, if $h_\oplus(w) < 2$, as for the initial case, v' would be an output node. This is a contradiction with the initial assumption on v'. Hence, we deduce that $h_\oplus(w) \geq 2$ if w' is a leaf. So $h_\oplus(v') =$

$\sum h_\oplus(w' \text{ child of } v') \leq \frac{1}{2}\sum h_\oplus(w) \leq h_\oplus(v)$.

By this induction, we proved that for any node in the DAG which is not transformed into a leaf or an output node by **Phase**, its height $h_\oplus$ is divided by 2. □

Let $h = \min(h_\oplus, h_\otimes)$.

Theorem 3 *Each application of* **Phase** *on a circuit divides its height* $h = \min(h_\oplus, h_\otimes)$ *by 2.*

Proof

It is true for $h_\oplus$. Since algorithm 1 deals with $\oplus$ and $\otimes$ in a symmetric manner, it is also true for $h_\otimes$; hence, it is true for h the minimum of $h_\oplus$ and $h_\otimes$.
□

By this theorem, after $\lceil \log_2 h \rceil$ applications of **Phase**, a circuit of height h is transformed into a circuit with only leaves and output nodes. One **Eval** is enough to evaluate every node. (Note that the preprocessing does not increase the height, and thus its influence can be neglected).

MRK proved that $h_\oplus \leq \frac{1}{2}e_\oplus d_\oplus + d_\oplus$ where $e_\oplus$ is the number of $\oplus - \oplus$ edges. Similarly, $h_\otimes \leq \frac{1}{2}e_\otimes d_\otimes + d_\otimes$ where $e_\otimes$ is the number of $\otimes - \otimes$ edges.

Thus, after $\lceil \log_2 h \rceil + 1 = \mathcal{O}(\log(nd))$ applications of **Phase**, a circuit of degree d with n nodes is evaluated.

Let's tackle the last part of the proof. Intuitively, it is clear that what prevents the **Phase** procedure to work more is the alternance of $\oplus$ and $\otimes$ nodes.

Let's show that at most $(h_a + \log n)$ applications of **Phase** suffice to evaluate the DAG.

Firstly, none of the **Group**, **Eval**, **PartialEval** or **Suppress** procedures increase h_a.

Secondly, the preprocessing plays an important role: **Group*** computes a transitive closure of $\oplus$ nodes and $\otimes$ nodes. After **Group***, h_a is the length of the longest path of the DAG +1. Obviously, the parallel evaluation time of a DAG is less than the length of its longest path: h_a applications of **Eval** are enough to evaluate the DAG, *a fortiori* h_a applications of **Phase** suffice to evaluate it.

If these results are put together, they involve theorem 1: at most $\mathcal{O}(\min(\log(nd), h_a + \log(n)))$ applications of **Phase** are enough in order to evaluate every node of the DAG.

Remark
The complexity of the preprocessing is bounded by the complexity of $\log n$ **Phase**; the preprocessing can even be replaced by $\log n$ applications of **Phase** if an homogeneous algorithm is preferred instead.

4 Applications

The aim of the following examples is only to illustrate the efficiency of algorithm 1. They have been chosen because their time complexity is already well-known and thus the comparison between the best implementation and the performances of our algorithm is possible. The results we obtain are encouraging. Real applications are mentioned at the end of this section and will constitute our future work.

4.1 Addition and multiplication of two n-bit numbers

In order to illustrate the complexity of algorithm 1, we have simulated its execution and counted the number of applications of **Phase**. The input straight-line programs are classical "paper-and-pencil" algorithms for infinite precision - either integer or fixed-point real - arithmetic, described with boolean gates. Let's consider the addition of two integers, the adaptation to real fixed-point addition being straightforward. Let $a = (a_{n-1} \ldots a_0)$ and $b = (b_{n-1} \ldots b_0)$ two n-bit integers. Introducing carries c_i, the equations defining the result $r = (r_{n-1} \ldots r_0)$, that we assume to be given in the VLSI specification, are the following:

$$\begin{aligned} c_0 &= 0, \\ c_i &= (a_i \wedge b_i) \vee ((a_i \vee b_i) \wedge c_{i-1}), \end{aligned}$$

$$\begin{aligned} r_i &= [(a_i \wedge \bar{b}_i) \vee [(\bar{a}_i \wedge b_i)] \wedge \bar{c}_{i-1}) \vee [(\bar{a}_i \wedge \bar{b}_i) \vee (a_i \wedge b_i)] \wedge c_{i-1} \\ r_n &= c_{n-1} \quad (r_n \text{ is an Overflow Flag}). \end{aligned}$$

It may be noticed that we consider here the VLSI specification as simple as possible.
The corresponding boolean circuit has $\mathcal{O}(n)$ nodes. Its degree is linear in the n,

and h_a is also linear. Thus, the predicted complexity is $CRCW_L(M'(n), \log n)$ [5]. On figure 4, we can check that this complexity is achieved, and that the constants are small. Since the parallel time is logarithmic for the tree contraction technique, and since this algorithm requires only a linear number of processors, this result means that an adder with a linear number of gates and a logarithmic delay exists. It can even be built if this contraction algorithm is used as a compiler instead of being used as an interpreter. An adder with the same properties has been proposed by Brent and Kung [3]. Our algorithm presents the advantage that it can build the circuit automatically from the classical boolean equations of the addition.

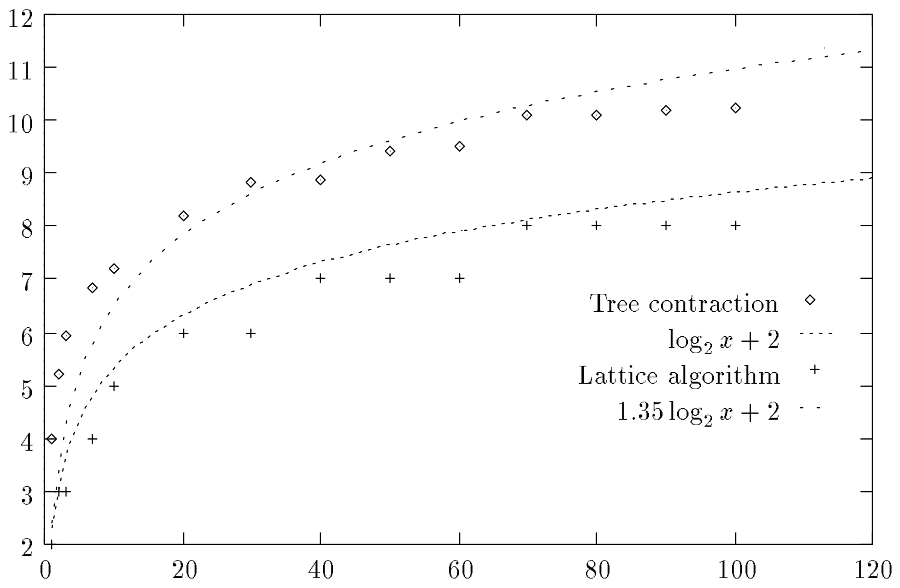

Fig. 4. *Experimental results for the addition.*

The same results (logarithmic time, small constants) occur when the boolean circuit for the multiplication of two n-bit numbers is evaluated [20].

To obtain these results, only one test with arbitrary inputs has been done, in order to determine the number of steps needed to compute the result; indeed, there is no trick using the actual values of the inputs in the algorithm; thus, the number of steps is the same, whatever the inputs are.

The addition requires the NOT operator. Goldschlager [11] has proven that a boolean circuit with NOT gates can be transformed into a monotone boolean circuit (without NOT gates). In fact we did not use this transformation; instead, we slightly modified the algorithm to be able to treat the NOT operation: the weighting functions are $True$, $False$, x, $\neg x$ (or more generally $((a \otimes x) \oplus (b \otimes \neg x) \oplus c)$-like functions); since NOT is a unary operation,

[5] In the Boolean algebra, $M'(n) = \mathcal{O}(n^3)$.

the NOT nodes are suppressed; since $a \oplus (\neg(b \otimes c)) = a \oplus (\neg b) \oplus (\neg c)$, $\oplus$ and $\otimes$ nodes are grouped when they are connected by a $\neg x$ edge; since $a \oplus (\neg(b \oplus c)) = a \oplus ((\neg b) \otimes (\neg c))$, $\oplus$ nodes are not grouped when they are connected by a $\neg x$ edge, neither are the $\otimes$ nodes. The evaluation time is the same for the monotone circuit as for the circuit with NOT gates; we can thus save half of the memory.

4.2 Parallel sort

Algorithm 1 can be used as a predictor of parallel complexity: the computation of the degree and the maximal number of alternances of a program provides an estimation of its parallel complexity which is often non trivial. For instance, let us consider the problem of sorting an array of n distinct numbers $x_1, \ldots x_n$ and of placing them in an array $y_1, \ldots y_n$ in increasing order. The structure $(\mathbb{R} \bigcup \{+\infty\} \bigcup \{-\infty\}, min, Max, +\infty, -\infty)$ is a distributive lattice. The easiest way to solve this problem is to program an insertion sort. An obvious parallelization of the insertion procedure leads to the following algorithm, the upper index denoting the step:

Algorithm 2 (*Insertion sort)*

for $i = 1$ *to* n *do*
  if $i = 1$ *then*
    $[y_1^{(1)}] \leftarrow [x_1]$
  if $i = 2$ *then*
    $[y_1^{(2)}, y_2^{(2)}] \leftarrow [min(x_1, x_2), Max(x_1, x_2)]$
  if $i > 2$ *then*
    $y_1^{(i)} \leftarrow min(x_i, y_1^{(i-1)})$
    $y_i^{(i)} \leftarrow Max(y_{i-1}^{(i-1)}, x_i)$
    for $j = 2$ *to* $i - 1$ *do*
      $y_j^{(i)} \leftarrow Max(y_{j-1}^{(i-1)}, min(y_j^{(i-1)}, x_i))$

Actually, the computation of $y_j^{(i)}$ from $y_{j-1}^{(i-1)}$, $y_j^{(i-1)}$ and x_i corresponds to the computation of the second element of the sorted array containing these three elements.
The maximal number of alternances of the corresponding circuit is at least linear: $h_a \geq h_a(y_2^{(n)})$ and $h_a(y_2^{(i)}) = 2 + h_a(y_2^{(i-1)}) = 2(i-1)$. The degree d_{Max}

is exponential: for $2 \le i \le n$,

$$\begin{cases} d_{Max}(y_1^{(i)}) = d_{Max}(y_1^{(i-1)}) \\ d_{Max}(y_j^{(i)}) = d_{Max}(y_j^{(i-1)}) + d_{Max}(y_{j-1}^{(i-1)}),\ 2 \le j \le i-1 \\ d_{Max}(y_i^{(i)}) = d_{Max}(y_i^{(i-1)}) + 1 \end{cases}$$

and they correspond to the $\binom{i}{j}$. The degree d_{min} is linear: for $2 \le i \le n$,

$$\begin{cases} d_{min}(y_1^{(i)}) = d_{min}(y_1^{(i-1)}) + 1 \\ d_{min}(y_j^{(i)}) = \max(d_{min}(y_j^{(i-1)}) + 1, d_{min}(y_{j-1}^{(i-1)})),\ 2 \le j \le i-1 \\ d_{min}(y_i^{(i)}) = d_{min}(y_i^{(i-1)}) \end{cases}$$

Thus, algorithm 1 evaluates the insertion sort circuit in logarithmic time on a $CRCW-PRAM$. The parallel complexity of the best known algorithms [12] for this problem is thus automatically predicted, using a simple and *a priori* not highly parallel algorithm.

4.3 A $\mathcal{P}$-complete boolean circuit

A problem of particular interest is the lexicographic maximal independent set problem, denoted by LMIS; actually, it is a $\mathcal{P}$-complete problem. As a matter of fact, it happens that the LMIS is one of our worst cases.
The LMIS problem is the following: let $G = (V, E)$ be a graph, and V be a linearly ordered set: $V = \{v_1, \ldots, v_n\}$ with $v_1 > v_2 > \ldots > v_n$. The LMIS problem consists in determining a maximal set of vertices that form an empty subgraph; this set must be maximal for the order on V [6].
A greedy sequential algorithm gives the solution, and the corresponding boolean circuit has an exponential degree, and a linear h_a, the complexity bound for its parallel evaluation using algorithm 1 is thus $\mathcal{O}(n)$.

4.4 Application fields: circuits for fixed-point arithmetic

First of all, some other boolean circuits can be studied in the same way as the addition or multiplication circuits. Thus an estimation of their depth can be easily obtained. If this estimation is good, it is then possible to "compile" the

[6] Without the lexicographic order constraint, this problem can be solved efficiently in $\mathcal{O}(\log^2(n))$ parallel time.

original circuit into another one, achieving the depth bound: algorithm 1 has to be applied "symbolically" on the circuit, i.e. each operation such as *disconnect from the child and connect to a grand-child* has to be physically realized, but no evaluation is performed. Furthermore, for the addition and multiplication circuits, it appears experimentally that the adjacency matrices are very sparse. Thus, if only the useful computations are performed for the **Group** operations, the number of simultaneous operations decreases significantly, *i.e.* the size of the circuit is small. For the addition circuit, only a linear number of operations are performed at each step, and then the size of the "compiled" circuit is small compared to the $\mathcal{O}(n^3)$ theoretical bound.
It seems that the usual arithmetic operations - implemented either by power series or in a "CORDIC way" for instance - present the same characteristics: very good theoretical time, theoretically overestimated size which appears to be reasonable in practice. In such cases, algorithm 1 can be used as a preprocessor for VLSI design, in order to build a circuit with a good cost.

In other areas, algorithm 1 cannot be used because it evaluates a straight-line program with too many processors. However, the complexity result of section 3 can be used in order to estimate an upper bound of the parallel time required to solve a problem, as a predictor. Such areas include for instance graph theory: problems such as the computation of connected components are expressed in terms of lattice operations (min - max), optimization problems or reliability studies require either lattice (min - max) or semi-ring (max - $+$, Δ - $\times$) operations and finally enumeration problems use a semi-ring. Lastly, simulations of discrete events systems (modeled by timed Petri nets for instance) perform computations in the semi-ring $(\mathbb{R}, \max, +)$.

5 Conclusion

In this paper, an algorithm for the parallel contraction of arithmetic circuits has been presented. Firstly, it unifies the various algorithms designed for different algebraic structures. Secondly, it improves previous algorithms by the use of the lattice's algebraic properties, and by a symmetric treatment of the lattice's $\oplus$ and $\otimes$ operations. Its complexity is $CREW_S(M(n), \log n * \min(\log n + \log d, h_a + \log n))$, where d is the algebraic degree of the circuit, and h_a is the maximal number of alternances of $\oplus$ and $\otimes$ in the DAG in the lattice case, $+\infty$ otherwise. In most problems, this bound is rather tight, thus this algorithm appears as a predictor for the time complexity of an algorithm, and as an indicator of the algebraic properties that have to be taken into account, in order to reach this time. The mapping issue is not covered by this approach: firstly, the material resources criterion is not minimized (the number of processors in a parallel computation or the area of a VLSI circuit); secondly, the problem of reusing the processing components is not considered. Further work has to be done in order to attain a trade-off between time and material resources - reflecting the AT^2 measure of quality in VLSI design.

More generally, this easy-to-compute complexity estimation provided by algorithm 1 is particularly interesting: it can be used to detect the existence of reductions in numerical programs: actually, a constant h_a for instance means that one operation prevails, and that reductions based on this operation probably exist. A linear $d_\oplus$ means that reductions of "dot-product" kind are worth to be searched, whereas a linear $d_\otimes$ indicates that $p \leftarrow p \otimes x_i \oplus y_i$ patterns are preferably to be looked for. This is valid even if the $\oplus$ and $\otimes$ operations do not define a lattice. Thus, the complexity results established in §3 can be integrated in automatic parallelizing tools, in order to guide the detection of reductions: indeed, the reduction procedures are now integrated in most of the parallel languages (HPF, MPI, ...) because they are performed efficiently on most of the highly parallel arithmetic units. Since the detection algorithms are rather costly and cannot be applied to the whole numerical program to be parallelized, they really need such an expert tool to guide them.

References

[1] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2), pp 287–302, June 1989.

[2] A. Borodin. On relating time and space to size and depth. *SIAM J. on Computing*, 6(4), pp 733–744, December 1977.

[3] R.P. Brent and H.T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C31(3), pp 260–264, 1982.

[4] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In 27^{th}*IEEE Symposium on Foundations of Computer Science*, pp 478–491, 1986.

[5] R. Cole and U. Vishkin. Approximate parallel scheduling. Part I : the basic technique with a pplications to optimal list ranking in logarithmic time. *SIAM J. on Computing*, 17(1), pp 128–142, February 1988.

[6] S. A. Cook. Towards a complexity theory of synchronous parallel computations. *Enseignement Mathématique*, 27, pp 99–124, 1981.

[7] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64, pp 2–22, 1985.

[8] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progression (full paper). *Journal of Symbolic Computation*, 9(3), pp 251–280, March 1990.

[9] J. von zur Gathen, G. Seroussi. Boolean circuits versus arithmetic circuits. *Proc.* 6^{th} *Int. Conf. Computer Science, Santiago, Chile*, pp 171–184, 1986.

[10] A. Gibbons and W. Rytter. Optimal parallel algorithms for dynamic expression evaluation and context-free recognition. *LNCS, VLSI Algorithms and Architecture*, (319), pp 32–45, 1988.

[11] L. M. Goldschlager. The monotone and planar circuit value problems are log space complete for P. *SIGACT News*, 9(2), pp 25–29, 1977.

[12] J. JáJá. *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[13] E. Kaltofen. Greatest common divisors of polynomials given by straight-line programs. *J.ACM*, 35(1), pp 231–264, January 1988.

[14] R. M. Karp and V. Ramachandran. *Handbook of Theoretical Computer Science, J. van Leeuwen, editor*, chapter Parallel Algorithms for Shared-Memory Machines, pp 869–941. Elsevier Science Publishers, B.V., 1990.

[15] S. R. Kosaraju and A. L. Delcher. Optimal parallel evaluation of tree-structured computations by raking. *LNCS, VLSI Algorithms and Architecture*, (319), pp 103–110, 1988.

[16] T. Lengauer *Handbook of Theoretical Computer Science, J. van Leeuwen, editor*, chapter VLSI Theory, pp 835–868. Elsevier Science Publishers, B.V., 1990.

[17] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *IEEE,* 26^{th} *IEEE Symposium on Foundations of Computer Science*, pp 478–489, 1985.

[18] G. L. Miller, V. Ramachandran, and E. Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM J. of Computing*, 17(4), pp 687–695, 1988.

[19] G. L. Miller and S.-H. Teng. Dynamic parallel complexity of computational circuits. *J.ACM*, pp 254–263, 1987.

[20] N. Revol. Complexité de l'évaluation parallèle de circuits arithmétiques. Ph.D. Thesis, LMC, INPG, 1994.

[21] W.L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22(3), pp 365–383, June 1981.

[22] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, Band 13(Heft 4), pp 354–356, 1969.

[23] V. Strassen. Vermeidung von Divisionen. *Journal für Mathematik*, Band 264, pp 184–202, 1973.