Can algebra be an API for parallel computing?

The owl of Minerva spreads its wings only with the falling of the dusk

Martin Berger

University of Sussex & Montanarius Ltd

Bertinoro, September 16, 2026

From repeated multiplication to repeated squaring

The direct calculation of x^8 is

$$x^8 = \underbrace{x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x}_{\text{7 multiplications}}$$

Repeated squaring:

$$\underbrace{\begin{array}{l}\text{let } a = x \cdot x \text{ in} \\ \text{let } b = a \cdot a \text{ in} \\ \text{let } c = b \cdot b \text{ in } c\end{array}}_{\text{3 multiplications}}$$

Repeated squaring works in general semigroups.

From repeated multiplication to repeated squaring

Volume 2 / **Seminumerical Algorithms**

THE ART OF COMPUTER PROGRAMMING

THIRD EDITION

Note: just one operation (semigroup) [16, §4.6.3]

From repeated multiplication to repeated squaring

Not trivial in general, e.g., x^{15}:

$$\underbrace{\begin{array}{l}\text{let } a = x \cdot x \text{ in}\\ \text{let } b = a \cdot a \text{ in}\\ \text{let } c = b \cdot b \text{ in}\\ \text{let } d = a \cdot b \text{ in}\\ \text{let } e = c \cdot d \text{ in}\\ \text{let } f = e \cdot x \text{ in } f\end{array}}_{\text{6 multiplications}}$$

$$\underbrace{\begin{array}{l}\text{let } a = x \cdot x \text{ in}\\ \text{let } b = a \cdot x \text{ in}\\ \text{let } c = b \cdot b \text{ in}\\ \text{let } d = c \cdot c \text{ in}\\ \text{let } e = b \cdot d \text{ in } e\end{array}}_{\text{5 multiplications}}$$

From repeated multiplication to repeated squaring

Not trivial in general, e.g., x^{15}:

$$\begin{aligned}
&\text{let } a = x \cdot x \text{ in} \\
&\text{let } b = a \cdot a \text{ in} \\
&\text{let } c = b \cdot b \text{ in} \\
&\text{let } d = a \cdot b \text{ in} \\
&\text{let } e = c \cdot d \text{ in} \\
&\underbrace{\text{let } f = e \cdot x \text{ in } f}_{\text{6 multiplications}}
\end{aligned}$$

$$\begin{aligned}
&\text{let } a = x \cdot x \text{ in} \\
&\text{let } b = a \cdot x \text{ in} \\
&\text{let } c = b \cdot b \text{ in} \\
&\text{let } d = c \cdot c \text{ in} \\
&\underbrace{\text{let } e = b \cdot d \text{ in } e}_{\text{5 multiplications}}
\end{aligned}$$

Aside: the (decision version of the) problem of finding the shortest chain for x^n (given binary n, k, is $\ell(n) \leq k$?) is neither known NP-hard, nor known in P. cf. [10, 4].

Exponential propagation

From: LTL learning on GPUs (with M. Valizadeh, N. Fijalkow) [31, 32], see also [1].

```
def eventually_naive(input_matrix):
    matrix = []
    for bv_orig in input_matrix:
        bv = bv_orig
        bv = bv | (bv_orig << 1)
        bv = bv | (bv_orig << 2)
        bv = bv | (bv_orig << 3)
        bv = bv | (bv_orig << 4)
        bv = bv | (bv_orig << 5)
        ...
        bv = bv | (bv_orig << 62)
        bv = bv | (bv_orig << 63)
        matrix.append(bv)
    return matrix
```

```
def eventually(input_matrix):
    matrix = []
    for bv in input_matrix:
        bv = bv | (bv << 1)
        bv = bv | (bv << 2)
        bv = bv | (bv << 4)
        bv = bv | (bv << 8)
        bv = bv | (bv << 16)
        bv = bv | (bv << 32)
        matrix.append(bv)
    return matrix
```

Bit-order convention used: left shift moves truth from later trace positions towards earlier positions.
Here `bv` is length 64 bitvector

Let's compare repeated squaring and exponential propagation

```
# Exponentiation by naive multiplication
power = x
span = 1
while span < N:
   power = power * x
   span += 1
```

```
# Exponentiation by repeated squaring
# N is a power of two
power = x
span = 1
while span < N:
   power = power * power
   span *= 2
```

```
# Naive (linear) eventually
goal = q_bits   # Formula as bitvector
span = 1
while span < width:
   goal = goal | (q_bits << span)
   span += 1
```

```
# Eventually by exponential propagation
# Width is a power of two
goal = q_bits
span = 1
while span < width:
   goal = goal | (goal << span)
   span *= 2
```

For simplicity, the code assumes fixed-width words whose overflowing bits are discarded.

General form: two operations $\star$ and τ and an equation

Ordinary repeated squaring uses a single function (semigroup multiplication) and

$$x^k \star x^k = x^{2k}.$$

Can we lift this to two operations? (For LTL: $\vee$ (disjunction) and $\ll$ (left shift))

General form: two operations $\star$ and τ and an equation

$$A_0 = \epsilon \qquad A_N = a \star \tau(a) \star \tau^2(a) \star \cdots \star \tau^{N-1}(a)$$

($N > 0$). If we had:

$$A_{m+n} = A_m \star \tau^m(A_n),$$

then we could get the doubling law

$$A_{2k} = A_k \star \tau^k(A_k).$$

Here

- Exponentiation: $\epsilon = 1$, $a = x$, $\star$ is multiplication, τ is the identity function
- Eventually $F\phi$ in (finite-trace) LTL: ϵ is the all-false bitvector, a is the bitvector representing formula ϕ, $\star$ is $\vee$ (disjunction), τ is logical left shift $\ll$

General form: two operations $\star$ and τ and an equation

We used

- Exponentiation: $\epsilon = 1$, $a = x$, $\star$ is multiplication, τ is the identity function
- Eventually $F\phi$ in (finite-trace) LTL: ϵ is the all-false bitvector, a is the bitvector representing formula ϕ, $\star$ is $\vee$ (disjunction), τ is logical left shift $\ll$

Let's analyse this:

- Exponentiation: $(\mathbb{N}, \cdot, 1)$ is a monoid, seed $a = x$ is in $\mathbb{N}$, and $\tau = \mathrm{id}$ is a monoid endomorphism on $\mathbb{N}$.
- Finite Eventually: $(\mathrm{BitVec}_N, \vee, [0, 0, ..., 0])$ is a monoid, seed $a = \mathrm{bits}(\phi)$ is in BitVec_N, and $\tau = \ll$ is a monoid endomorphism on BitVec_N.

Goal

Generalise powering by repeated squaring from one operation to a binary operation interleaved with a unary operation.

Ideally, we can recover both algorithms from the previous slides.

What abstract mathematical structure lets us do this?

Seeded monoid endomorphism (SME)

A *seeded monoid endomorphism* is a triple (M, τ, a) such that

- $M = (M, \star, \epsilon)$ is a monoid
- $\tau : M \to M$ is an endomorphism: $\tau(x \star y) = \tau(x) \star \tau(y)$ for all $x, y \in M$ and $\tau(\epsilon) = \epsilon$.
- $a \in M$

Note that we do not require $\tau(a) = a$ (but we allow it).

We call τ the *rebasing* operation

The cocycle law

To reach our goal (generalise powering by repeated squaring) we need, given an SME (M, τ, a), this to hold:

$$A_0 = \epsilon \qquad A_1 = a \qquad A_{j+k} = A_j \star \tau^j(A_k)$$

The cocycle law

To reach our goal (generalise powering by repeated squaring) we need, given an SME (M, τ, a), this to hold:

$$A_0 = \epsilon \qquad A_1 = a \qquad A_{j+k} = A_j \star \tau^j(A_k)$$

Theorem (Cocycle law)

For every SME and all $j, k \in \mathbb{N}$,

$$A_{j+k} = A_j \star \tau^j(A_k). \tag{1}$$

No commutativity or idempotence is assumed: order matters

Question for audience: who proved this first?

Exponentiation by repeated squaring in general SMEs

The cocycle law for SME (M, τ, a) can be packaged as an ordinary monoid power. For $x, y \in M$ and $m, n \in \mathbb{N}$ define

$$\langle x, m\rangle \diamond \langle y, n\rangle = \langle x \star \tau^m(y), m + n\rangle. \tag{2}$$

(This is a semidirect product $M \rtimes_\tau \mathbb{N}$.) Left component does interleaved $\star$ and τ, right component controls recursion.

Endomorphism preservation makes $\diamond$ associative, with unit $\langle \epsilon, 0\rangle$. Then clearly:

$$\langle A_k, k\rangle^2 = \langle A_k, k\rangle \diamond \langle A_k, k\rangle = \langle A_k \star \tau^k(A_k), 2k\rangle = \langle A_{2k}, 2k\rangle. \tag{3}$$

Exponentiation by repeated squaring in general SMEs

Define the empty block and the one-factor generator by

$$B_0 = \langle \epsilon, 0 \rangle, \qquad g = B_1 = \langle a, 1 \rangle.$$

Then, for every $n \in \mathbb{N}$,

$$B_n = g^n = \langle A_n, n \rangle.$$

The cocycle law says precisely $B_m \diamond B_n = B_{m+n}$. This gives us:

- Linear evaluation: $B_{k+1} = B_k \diamond B_1$.
- Doubling: $B_{2k} = B_k \diamond B_k$.
- Arbitrary addition-chain evaluation: $B_{p+q} = B_p \diamond B_q$.

Summary

SMEs let us see an apparently heterogeneous product

$$a \star \tau(a) \star \cdots \star \tau^{n-1}(a)$$

as an ordinary power of one element in a larger monoid. Standard powering addition chains (cf. Knuth's TAOCP) can therefore be reused unchanged.

Note that standard exponentiation by repeated squaring is the degenerate case ($\tau = id$), where $M \rtimes_{id} \mathbb{N}$ is just the direct product and the SME machinery adds nothing.

The SME interface in Scala

```
trait Monoid[K]:
    def identity: K
    def combine(left: K, right: K): K

trait EndomorphismAction[Op, K]:
    def identity: Op
    def compose(left: Op, right: Op): Op
    def apply(operator: Op, value: K): K

case class SeededEndomorphism[Op, K](
    monoid: Monoid[K],
    operators: EndomorphismAction[Op, K],
    tau: Op,
    seed: K
)
```

An instance: packed Boolean Eventually

```
case class BlockSummary[Op, K](
    aggregate: K,
    power: Op,
    span: BigInt
) derives CanEqual

def eventuallyProblem(seed: BitTrace):
    SeededEndomorphism[BitShift, BitTrace] =
    SeededEndomorphism(
        OrTraceMonoid(seed.width),
        bitShiftAction,
        BitShift(BigInt(1)),
        seed
    )
```

Here $K =$ `BitTrace`, `combine` is bitwise OR, and `BitShift`(k) applies a zero-padded shift by k. The initial block represents

`BlockSummary(seed, BitShift(1), 1)` $\quad\longleftrightarrow\quad B_1 = \langle a, 1 \rangle.$

NB: Scala does not enforce the monoid/action laws. Lean supplies the mathematical proofs.

Exponential propagation (2)

Above we computed $F\phi$ using exponential propagation, that turns out to be powering by repeated squaring, via SMEs. What about pUq, the binary temporal operator from LTL? In [32] we derived by hand:

```
def until(input_matrix_1, input_matrix_2):
    matrix = []
    for i in range(len(input_matrix_1)):
        bv1 = input_matrix_1[i]
        bv2 = input_matrix_2[i]
        bv2 = bv2 | (bv1 & (bv2 << 1))
        bv1 = bv1 & (bv1 << 1)
        bv2 = bv2 | (bv1 & (bv2 << 2))
        bv1 = bv1 & (bv1 << 2)
        bv2 = bv2 | (bv1 & (bv2 << 4))
        bv1 = bv1 & (bv1 << 4)
        bv2 = bv2 | (bv1 & (bv2 << 8))
        bv1 = bv1 & (bv1 << 8)
        bv2 = bv2 | (bv1 & (bv2 << 16))
        bv1 = bv1 & (bv1 << 16)
        bv2 = bv2 | (bv1 & (bv2 << 32))
        matrix.append(bv2)
    return matrix
```

Note: we now have 3 operators: conjunction, disjunction and left shift. Can SMEs handle this?

LTL's Until as an SME

For $p \mathbf{U} q$ define (assuming traces of length N):

- $M = \mathrm{BitVec}_N \times \mathrm{BitVec}_N$
- $(g, w) \odot (h, v) = \big(g \wedge h,\ w \vee (g \wedge v)\big)$ pointwise
- $\epsilon_\odot = (\mathsf{true}, \mathsf{false})$
- Rebasing: $\tau(g, w) = \big(\sigma_1(g), \sigma_0(w)\big)$, where σ_1 left-shifts with true padding and σ_0 left-shifts with false padding.

We can now define the SME for Until as $(M, \tau, (p, q))$. With

$$A_0 = \epsilon_\odot \qquad\qquad A_k = (G_k, W_k),$$

as before

$$W_N = \mathsf{bits}(p \mathbf{U} q).$$

Repeated squaring in the induced semidirect-product monoid $M \rtimes_\tau \mathbb{N}$ therefore gives, essentially, the hand-derived algorithm above.

For simplicity, we assume that formulae p, q are already bitvectors. We also ignore G_N.

Wait a minute:

Wait a minute:

We know that associative operations expose multiple forms of parallelism.

Wait a minute: prefix scans?

An algebra of scans

Ralf Hinze

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
ralf@informatik.uni-bonn.de
http://www.informatik.uni-bonn.de/~ralf/

Abstract. A parallel prefix circuit takes n inputs $x_1, x_2, \ldots, x_n$ and produces the n outputs $x_1, x_1 \circ x_2, \ldots, x_1 \circ x_2 \circ \cdots \circ x_n$, where '$\circ$' is an arbitrary associative binary operation. Parallel prefix circuits and their counterparts in software, parallel prefix computations or scans, have numerous applications ranging from fast integer addition over parallel sort-

Parallel Prefix Computation

RICHARD E. LADNER AND MICHAEL J. FISCHER

University of Washington, Seattle, Washington

ABSTRACT The prefix problem is to compute all the products $x_1 * x_2 * \cdots * x_k$ for $1 \le k \le n$, where $*$ is an associative operation. A recursive construction is used to obtain a product circuit for solving the prefix problem which has depth exactly $\lceil \log_2 n \rceil$ and size bounded by $4n$. An application yields fast, small Boolean circuits to simulate finite-state transducers. By simulating a sequential adder, a Boolean circuit which has depth $2\lceil \log_2 n \rceil + 2$ and size bounded by $14n$ is obtained for n-bit binary addition. The size can be decreased significantly by permuting the depth to increase by an additive constant

KEY WORDS AND PHRASES automaton, binary addition, circuit, combinational complexity, depth, fanout, parallelism, size, transducer

CR CATEGORIES 5.22, 5.25, 6.1, 6.32

1. *Introduction*

Many algorithmic problems are easy to solve sequentially with finite memory. Examples are the addition of two binary numbers and the division of a binary number by a constant. By way of contrast, efficient parallel solutions to these same problems (restricted to inputs

1

Prefix Sums and Their Applications

Guy E. Blelloch

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Ordinary scans: arbitrary inputs, every prefix

Let $(M, \star, \epsilon)$ be a monoid and let $x_0, x_1, \ldots, x_{N-1} \in M$ be arbitrary. Write

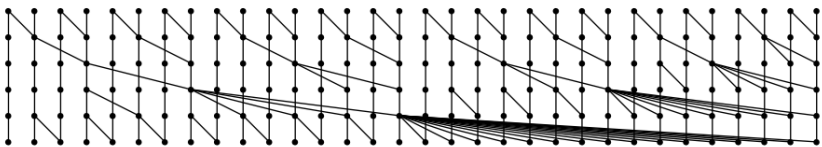
$$P_0 = \epsilon, \qquad P_n = x_0 \star x_1 \star \cdots \star x_{n-1}.$$

The (inclusive) prefix scan returns

$$\mathrm{scan}(x_0, \ldots, x_{N-1}) = (P_1, P_2, \ldots, P_N).$$

What associativity provides

Serial, balanced, and parallel-prefix circuits may parenthesise and share adjacent products differently while returning the same complete prefix vector. Highly useful for parallelisation:



[19, 3, 15]

Scans vs SMEs

$$\text{Scan:} \qquad A_0 = \epsilon \qquad A_n = x_0 \star x_1 \star x_2 \star \cdots \star x_{n-1}$$

$$\text{SME:} \qquad A_0 = \epsilon \qquad A_n = a \star \tau(a) \star \tau^2(a) \star \cdots \star \tau^{n-1}(a)$$

In other words, in general scans, each element can be arbitrary, while SMEs elements are all τ-**orbits of the seed** a.

Substituting the orbit factors into an ordinary scan gives

$$\text{scan}\big(a, \tau(a), \ldots, \tau^{N-1}(a)\big) = \big(A_1, A_2, \ldots, A_N\big).$$

We will come back to scans below

What extra information does an SME provide?

For arbitrary scan inputs, define the length-n interval product

$$C_{i,n} = x_i \star x_{i+1} \star \cdots \star x_{i+n-1}.$$

Associativity gives adjacent composition:

$$C_{i,m+n} = C_{i,m} \star C_{i+m,n}.$$

In an SME, $x_i = \tau^i(a)$, so we additionally know

$$C_{i,n} = \tau^i(A_n).$$

The additional information

Every length-n interval product is a rebase of the same canonical block A_n.

$$A_n \xrightarrow{\tau^i} C_{i,n} \qquad \text{representing the interval } [i, i+n).$$

General scans can compose adjacent blocks but cannot generally derive their values from one canonical block.

What can a compiler do with the additional structure?

Recall

$$A_N = a \star \tau(a) \star \cdots \star \tau^{N-1}(a) \qquad B_n = (A_n, n), \qquad B_m \diamond B_n = B_{m+n}.$$

The required outputs determine the legal schedule:

$$\begin{aligned} (A_1, \ldots, A_N) &\implies \text{prefix scan,} \\ A_N &\implies \text{powering by repeated squaring, or addition chain,} \\ \{A_n \mid n \in H\} &\implies \text{shared multi-output span DAG.} \end{aligned}$$

For example, if only A_8 is required:

$$B_2 = B_1 \diamond B_1, \qquad B_4 = B_2 \diamond B_2, \qquad B_8 = B_4 \diamond B_4.$$

Compiler opportunities

The compiler may share canonical blocks, rebase them to required intervals, omit unrequested prefixes, and select a validated span DAG.

When does the additional structure pay?

An SME schedule is potentially useful when all four conditions hold:

- **Few required outputs:** one A_n, or a sparse set of A_i, rather than every prefix.
- **Succinct summaries:** the representation of A_n is substantially smaller than the orbit fragment it summarises. In particular: does not grow (much) throughout the SEM computation.
- **Closed operations:** $\star$ and τ^k operate directly on that representation without expanding the underlying factors.
- **Favourable target costs:** the saved block operations outweigh rebasing, storage, communication, and preprocessing.

All obtain in LTL & integer exponentiation

What SMEs do not give the compiler

Consider again

```
# Exponentiation by repeated squaring
# N is a power of two
power = x
span = 1
while span < N:
   power = power * power
   span *= 2
```

```
# Eventually by exponential propagation
# Width is a power of two
goal = q_bits
span = 1
while span < width:
   goal = goal | (goal << span)
   span *= 2
```

Note: all use an accumulator with destructive update. This, however, is not possible in general, e.g. optimal x^{15}.

$$
\begin{aligned}
&\text{let } a = x \cdot x \text{ in} \\
&\text{let } b = a \cdot x \text{ in} \\
&\text{let } c = b \cdot b \text{ in} \\
&\text{let } d = c \cdot c \text{ in} \\
&\text{let } e = b \cdot d \text{ in } e
\end{aligned}
$$

What SMEs do not give the compiler

Pure doubling has a chain-shaped dependency graph:

$$B_1 \longrightarrow B_2 \longrightarrow B_4 \longrightarrow B_8 \longrightarrow B_{16}.$$

Only the latest logical block need remain live.

A general addition-chain or sparse multi-horizon programme may branch:

$$\begin{aligned} B_1 \longrightarrow B_2 \longrightarrow B_4 \longrightarrow B_8, \\ \searrow B_6 \longrightarrow B_{10}. \end{aligned}$$

Previously computed blocks may then have to remain live for later reuse.

Required storage is joint property of

$$\text{SME algebra} + \text{schedule DAG} + \text{representation}$$

SMEs vs compiler: summary

SME gives additional equations that hold (and can be used in optimisation)

SME does not give

- Schedule
- Data representation
- Primitive instructions

So what, Part 2?

Are there many interesting applications?

Three canonical SMEs induced by a semiring

Select x in a semiring $(R, +, \cdot, 0, 1)$.

$$\begin{array}{lll} \textbf{Multiplicative} & (R, \cdot, 1; \mathrm{id}, x) & A_N = x^N, \\ \textbf{Additive left} & (R, +, 0; \lambda_x, 1) & A_N = \sum_{k<N} x^k, \\ \textbf{Additive right} & (R, +, 0; \rho_x, 1) & A_N = \sum_{k<N} x^k, \end{array}$$

where

$$\lambda_x(y) = x \cdot y, \qquad \rho_x(y) = y \cdot x.$$

Important distinction:

The two additive routes have the same aggregate, but retain different action orientations: $x^j A_k$ versus $A_k x^j$.

Which examples use the canonical routes?

Canonical route	Direct instances
Multiplicative: $A_N = x^N$	Ordinary powers, matrix powers for exact-length paths and weighted-automaton length totals, regular-language and Kleene-algebra powers, fixed-width truncated-series powers such as $(1+z)^N$.
Additive left: $A_N = \sum_{k<N} x^k$	Bounded path closure, natural counting, min-plus, max-times, max–min and language-valued matrix closures, finite regular-language and Kleene-algebra closure.
Additive right: $A_N = \sum_{k<N} x^k$	The right-oriented version of those bounded closures, our path presentation uses $\tau(Y) = YP$.

Applications I: temporal logic and signals

- Boolean finite-trace LTL: X, F, G, U, R [7, 32]
- Boolean LTL on ultimately periodic lasso traces [27]
- Lattice-valued finite-trace F/G [23] (replacing Boolean truth values by values in a bounded distributive lattice for quantitative monitoring of cyber-physical systems)
- Discrete sampled STL (Signal Temporal Logic) robustness: bounded F, G, and exclusive U [9, 8]

Applications II: paths and finite reductions

- Exact-length path and witness counting [21]
- Modular counts and unnormalised rational witness totals [14]
- Bounded min-plus cost-to-reach [21]
- Max-times most-likely reduction sequences [11]
- Max–min bottleneck routes and fixed-topology stability [30]
- Chronological rule-word languages [11]

Applications III: automata, routing, and tropical problems

- Truncated formal-power-series prefixes and bounded witness-length distributions [11]
- Finite weighted automata and bounded-length acceptance [11]
- Finite parametric tropical optimal paths [14, 2]
- Finite distributive algebraic routing [30]

Applications IV: recursion and program semantics

- Ground finite-affine weighted recursion [6]
- Graded bounded-duplication certificates [25]
- First-order graded substitution [26]
- Simply typed first-order graded beta evaluation [20]
- Nested first-order graded program evaluation [20, 26]
- Non-recursive higher-order graded arrows [20, 25]

Applications V: graphs, control, and stateful iteration

- Exact type-aggregated inference for stochastic Kronecker graph models [22]
- Finite-horizon controllability Gramians [29, 28]
- Functional-graph pointer chasing and binary lifting [13, 24]
- Deterministic output-carrying transducers [11, 24]
- Fixed linear and affine recurrences [17, 24]

Work in progress (e.g., on compilation):

- Fourier/Vandermonde and DFT-matrix orbit aggregation [5]
- Certified finite polynomial and Newton corrections [12]
- Regular-language closure and regular expressions [18]

Applications

Whatever else SMEs may be useful for, they unify some correctness work.

So what, Part 3?

Are there many interesting accelerators?

Opportunities for accelerators for SMEs?

The most important difference is that an arbitrary-input scan cannot assume any two operands are related.

$$x \star y,$$

whereas SME doubling combines related operands,

$$x \star \tau^k(x).$$

An accelerator may therefore load x once, derive the second operand through a shift, permutation, sparse transformation or structured matrix action, and fuse the result. The "Eventually" in LTL is a clean example:

$$g \leftarrow g \vee (g \ll k).$$

There is no need to materialise the shifted bitvector as a separate scan input.

What SME structure offers an accelerator

SME information	Accelerator opportunity	Qualification
$x_i = \tau^i(a)$	Generate factors from one seed instead of materialising them	Useful only when τ^i is executable succinctly
$C_{i,k} = \tau^i(A_k)$	Compute one canonical block per span and rebase it	Rebasing must be cheaper than recomputation or loading
$A_{2k} = A_k \star \tau^k(A_k)$	Use doubling and addition-chain schedules	Best for selected n in A_n, not necessarily complete scans
Related operands in doubling	Fuse action and combination as $D_k(x) = x \star \tau^k(x)$	Requires a suitable target instruction or kernel
Explicit selected horizon	Avoid constructing all prefixes	Cannot help when every prefix is required
Fixed composite summary	Keep a small tuple in registers or local memory	The represented summary must actually remain small
Known spans and actions	Use static tiling, addressing and synchronisation	Dynamic horizons may require more general control

Caveat:

The SME laws do not say anything about data representation and cost of operations.

Two very tentative experiments

Compiling from SME to tensor cores.

Finite-Horizon Gramians on Matrix-Multiplication Accelerators

MARTIN BERGER, Montanarius Ltd, UK and University of Sussex, UK

Matrix-multiplication accelerators promise high throughput when a workload contains large, reusable dense products, but that promise does not determine which algorithmic formulation should be used. We study finite-horizon discrete-time controllability Gramians as a deliberately ordinary-arithmetic case: the mathematical operators are real matrix addition and multiplication, without a semiring substitution or nonlinear decoder. The same Gramian admits a sequential recurrence, a logarithmic-depth Smith-style composition, and a factorized controllability-matrix form. This creates a useful regime question involving horizon, state dimension, input rank, dynamics, batch size, precision, and data movement. The current manuscript defines that comparison and its evidence requirements, proves its finite-horizon identities, and validates four portable formulations against an independent exact reference on untimed structured and generated corpora. We then execute a frozen strict-FP32 study of Metal Performance Primitives TensorOps on one Apple M5 Pro against matched MPS and OpenBLAS baselines. Of 230 MPP route–cell comparisons, only 32 satisfy the preregistered

Evaluating LTL on matrix-multiplication accelerators

Mojtaba Valizadeh Martin Berger

Abstract

Matrix engines make affine computation cheap, but finite-trace linear temporal logic (LTL) is governed by ordered

How can a compiler preserve finite-trace semantics while selecting exact affine maps with epilogues, fixed-block scans, or vector fallbacks under explicit representation, precision, and cost

Comparison with Maslen / Rockmore

Our SME (M, τ, a)	Maslen–Rockmore
carrier M	carrier A
monoid product $\star$	vector product $\star : A \times A \to A$
rebasing τ	basic shift L_1
orbit τ^i	shift or lag operator L_i
seed a	$a \in A$
span-summary (A_i, i)	state (i, A_i)
$M \rtimes_\tau \mathbb{N}$	$\mathbb{Z}_{>0} \ltimes_L A$

"How to Compute a Moving Sum" by Maslen / Rockmore: https://arxiv.org/abs/2509.00537

Our generic Scala & Lean evaluators are executable, zero-horizon-completed variant of Maslen/Rockmore's semidirect-product powering procedure [24, Theorem 11.9 and Algorithms 11.11 and 11.14, pp. 104–107, Chapter 15, pp. 132–133]. Their `window_compose` operation corresponds to our block-summary composition, while their `binary_exponentiate` supplies the same accumulator-and-squaring schedule.

We verify this in Lean.

Conclusion

Conclusion

Talk asked: Can algebra be an API for parallel computing?

Conclusion

Talk asked: Can algebra be an API for parallel computing?

Answer: yes, sometimes!

Conclusion

Talk asked: Can algebra be an API for parallel computing?

Answer: yes, sometimes!

What is the role of the SME algebra?

- Vocabulary for unification
- Additional information for compiler/accelerator
- Interesting new question for pure mathematics

Conclusion

Talk asked: Can algebra be an API for parallel computing?

Answer: yes, sometimes!

What is the role of the SME algebra?

- Vocabulary for unification
- Additional information for compiler/accelerator
- Interesting new question for pure mathematics

Or: Knuth could have invented our fast LTL algorithms ...

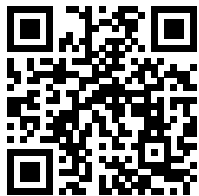
... or: I have not read enough algorithms & maths papers ...

... "the owl of Minerva spreads its wings only with the falling of the dusk"

Thank you!

Contact information & publications

Martin Berger



References I

[1] S. E. Anderson.
Bit twiddling hacks: Round up to the next highest power of 2.
2005.

[2] D. Barbarossa and P. Pistone.
Tropical mathematics and the lambda-calculus I: Metric and differential analysis of effectful programs.
In *32nd EACSL Annual Conference on Computer Science Logic (CSL 2024)*, volume 288 of *Leibniz International Proceedings in Informatics*, pages 14:1–14:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

[3] G. E. Blelloch.
Prefix sums and their applications.
Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Nov. 1990.

[4] K. Casel, H. Fernau, S. Gaspers, B. Gras, and M. L. Schmid.
On the Complexity of the Smallest Grammar Problem over Fixed Alphabets.
Theory of Computing Systems, 65(2):344–409, 2021.

[5] J. W. Cooley and J. W. Tukey.
An algorithm for the machine calculation of complex Fourier series.
Mathematics of Computation, 19(90):297–301, 1965.

[6] U. Dal Lago, G. Fiorillo, and P. Pistone.
On Higher-Order Probabilistic Verification via the Weighted Relational Model of Linear Logic, 2026.

[7] G. De Giacomo and M. Y. Vardi.
Linear Temporal Logic and Linear Dynamic Logic on Finite Traces.
In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, pages 854–860, Beijing, China, 2013. AAAI Press.

[8] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia.
Robust online monitoring of signal temporal logic.
In *Runtime Verification*, volume 9333 of *Lecture Notes in Computer Science*, pages 55–70, Cham, 2015. Springer.

[9] A. Donzé and O. Maler.
Robust satisfaction of temporal logic over real-valued signals.
In *Formal Modeling and Analysis of Timed Systems*, volume 6246 of *Lecture Notes in Computer Science*, pages 92–106, Berlin, Heidelberg, 2010. Springer.

References II

[10] P. Downey, B. Leong, and R. Sethi.
Computing sequences with addition chains.
SIAM Journal on Computing, 10(3):638–646, 1981.

[11] M. Droste and W. Kuich.
Semirings and formal power series.
In M. Droste, W. Kuich, and H. Vogler, editors, *Handbook of Weighted Automata*, pages 3–28. Springer, Berlin, Heidelberg, 2009.

[12] J. Esparza, S. Kiefer, and M. Luttenberger.
Newtonian program analysis.
Journal of the ACM, 57(6):1–47, 2010.

[13] P. Flajolet and A. M. Odlyzko.
Random mapping statistics.
In *Advances in Cryptology—EUROCRYPT '89*, volume 434 of *Lecture Notes in Computer Science*, pages 329–354, Berlin, Heidelberg, 1990. Springer.

[14] J. S. Golan.
Semirings and their Applications.
Kluwer Academic Publishers, Dordrecht, 1999.

[15] R. Hinze.
An algebra of scans.
In D. Kozen and C. Shankland, editors, *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 186–210. Springer, 2004.

[16] D. E. Knuth.
The Art of Computer Programming, Volume 2: Seminumerical Algorithms.
Addison-Wesley, Reading, MA, 3 edition, 1997.
Section 4.6.3, Evaluation of Powers, pp. 461–485.

[17] P. M. Kogge and H. S. Stone.
A parallel algorithm for the efficient solution of a general class of recurrence equations.
IEEE Transactions on Computers, C-22(8):786–793, 1973.

References III

[18] D. Kozen.
A completeness theorem for Kleene algebras and the algebra of regular events.
Information and Computation, 110(2):366–390, 1994.

[19] R. E. Ladner and M. J. Fischer.
Parallel prefix computation.
Journal of the ACM, 27(4):831–838, 1980.

[20] J. Laird, G. Manzonetto, G. McCusker, and M. Pagani.
Weighted relational models of typed lambda-calculi.
In *2013 28th Annual IEEE/ACM Symposium on Logic in Computer Science*, pages 301–310. IEEE, 2013.

[21] D. J. Lehmann.
Algebraic structures for transitive closure.
Theoretical Computer Science, 4(1):59–76, 1977.

[22] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani.
Kronecker graphs: An approach to modeling networks.
Journal of Machine Learning Research, 11:985–1042, 2010.

[23] K. Mamouras, A. Chattopadhyay, and Z. Wang.
Algebraic quantitative semantics for efficient online temporal monitoring.
In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 12651 of *Lecture Notes in Computer Science*, pages 330–348, Cham, 2021. Springer.

[24] D. K. Maslen and D. N. Rockmore.
How to compute a moving sum: Windowed recurrences—a monograph, 2026.
Version 2, corrected 8 February 2026.

[25] B. Moon, H. E. III, and D. Orchard.
Graded modal dependent type theory, 2021.

[26] D. Orchard, V.-B. Liepelt, and H. Eades.
Quantitative program reasoning with graded modal types.
Proceedings of the ACM on Programming Languages, 3(ICFP), 2019.

References IV

[27] A. Pnueli.
The temporal logic of programs.
In *18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57. IEEE, 1977.

[28] M. Sadkane.
A low-rank Krylov squared Smith method for large-scale discrete-time Lyapunov equations.
Linear Algebra and its Applications, 436(8):2807–2827, 2012.

[29] R. A. Smith.
Matrix equation $XA + BX = C$.
SIAM Journal on Applied Mathematics, 16(1):198–201, 1968.

[30] J. a. L. Sobrinho.
An algebraic theory of dynamic network routing.
IEEE/ACM Transactions on Networking, 13(5):1160–1173, 2005.

[31] M. Valizadeh and M. Berger.
Search-Based Regular Expression Inference on a GPU.
Proc. ACM Program. Lang., 7(PLDI), jun 2023.
Draft available at `https://arxiv.org/abs/2305.18575`, implementation: `https://github.com/MojtabaValizadeh/paresy`.

[32] M. Valizadeh, N. Fijalkow, and M. Berger.
LTL learning on GPUs.
In *Computer Aided Verification*, volume 14683 of *Lecture Notes in Computer Science*, pages 209–231, Cham, 2024. Springer.