Silicon Cartography

Constraint-First Programming for Spatial Chips

Flyxion

Independent Researcher

May 2026

Abstract

Contemporary compilation assumes a computational substrate that is abundant, uniform, and energetically indifferent. This assumption fails for a growing class of spatial chips: multi-core mesh architectures, neuromorphic arrays, FPGAs, and low-power embedded systems in which energy, memory locality, routing topology, and inter-node communication are the dominant engineering parameters. We introduce *Silicon Cartography*, a constraint-first programming discipline and corresponding compiler architecture that begins not with a program but with an interrogation of the machine. The hardware is modeled as a constrained spatial computer—a metrized graph $\mathcal{H} = (N, E, \mathcal{A}, \mathcal{M}, \mathcal{C}, \Phi)$ encoding nodes, communication edges, instruction algebras, memory geometry, hard constraints, and physical cost functionals. A program is then a mapping problem: factor a desired computation into locally self-contained kernels, place those kernels onto the discovered hardware geometry, route communication, and schedule wake/sleep behavior so that only necessary physical transitions occur.

We develop this framework across five interrelated dimensions. First, we establish the formal structure of the hardware graph and its compilation functor, including feasibility, optimality, and compositionality results. Second, we develop a taxonomy of fractal procedural chip topologies—fractal island meshes, tree-of-meshes, Hilbert-curve chips, Sierpinski compute fabrics, dragon-curve pipelines, and Apollonian heterogeneous fabrics—each with characteristic locality properties and programming disciplines. Third, we analyze the communication manifold structure of spatial chips through graph Laplacians, spectral connectivity, and synchronization sheaves, connecting deadlock detection to Čech cohomology obstruction classes. Fourth, we introduce a procedural topology language that separates hardware description from compilation policy, enabling systematic design-space exploration. Fifth, we develop a visualization framework that represents the six interacting dimensions of spatial chip behavior—topology, activity, energy, communication, memory, and time—through activity heatmaps, event wave propagation, causal braid diagrams, multiscale semantic zooming, and energy landscape terrain maps. Finally, we ground the energy-minimality criterion in thermodynamic computation theory, connecting Moore's principle that every unnecessary transistor transition is waste to Landauer's fundamental bound on the thermodynamic cost of logically irreversible operations.

Contents

Motivation: The Return of Physical Constraints

The dominant tradition in software engineering treats hardware as a uniform abstraction. A program is written against an idealized machine—sequential, byte-addressable, with abundant RAM and an optimizing compiler mediating between intent and execution. This tradition emerged from a historically contingent fact: from roughly 1970 to 2010, transistor density, clock speed, and memory bandwidth improved rapidly enough that programmers could afford to ignore the physical substrate. Abstraction layers—operating systems, virtual machines, garbage collectors, high-level languages—accumulated precisely because the hardware could absorb their overhead.

That era is structurally over in several important engineering domains. Edge inference, wearable computing, embedded intelligence, sensor swarms, and large-scale datacenter deployments each face a common bottleneck: energy. As Chuck Moore observed in his documentation of the GreenArrays architecture, a 144-node mesh of tiny Forth computers, power consumption is no longer a secondary engineering parameter but the primary one. Even two logically equivalent delay loops may consume measurably different energy because one toggles more decode circuitry than the other. At the spatial and temporal scales of modern embedded systems, every unnecessary transistor transition carries a thermodynamic cost.

The implications for programming methodology are radical. If energy is primary, then the compiler's job is not to translate syntax while optimizing for speed. It is to find a physical placement of computation that minimizes unnecessary state transitions, routes communication along low-cost paths, keeps idle hardware in low-energy dormancy, and exploits the specific geometry of the chip's memory, bus, and interconnect topology. This requires the compiler—and the programmer—to begin by knowing the machine.

Silicon Cartography is the discipline of beginning there: with an interrogation of the hardware geometry before a single line of code is written.

The Hardware Model

Spatial Chips as Constraint Graphs

We define a *spatial chip* as any computational substrate in which the topology of processing units, their communication paths, and their local memory resources are physically fixed and non-uniform. GreenArrays-style mesh processors, FPGAs, neuromorphic arrays, and many modern embedded SoCs satisfy this description. The key property is that computation has *location*: a kernel running on node n_i has access to different resources at different costs than the same kernel running on node n_j, and communication between distant nodes is more expensive than communication between neighbors.

Definition 1 (Hardware Graph)**.** *A hardware graph is a tuple*

$$H = (N,\, E,\, I,\, M,\, C,\, P)$$

where:

- N *is a finite set of computational nodes (cores, processing elements);*
- $E \subseteq N \times N$ *is a set of directed communication edges representing physical links, buses, DMA channels, or shared memory paths;*
- $I : N \to \mathcal{P}(\mathit{Instr})$ *assigns to each node a set of executable instructions or operations;*
- $M : N \to \mathbb{N}$ *assigns to each node a local memory capacity (words or bytes);*
- C *is a set of hard constraints: maximum fanout, routing exclusions, timing windows, voltage domains, synchronization requirements; and*
- $P : N \cup E \times \mathit{Op} \to \mathbb{R}_{\geq 0}$ *is a physical cost model assigning energy, latency, heat, and contention costs to operations and communication events.*

The hardware graph is not a datasheet. It is an *executable constraint map*: a structure that can be queried to answer questions of the form "can computation k run at node n?", "what does it cost to send message m along edge e?", and "which nodes can sleep while kernel k' executes?". The first task of the Silicon Cartographer system is to construct this map.

The Cartographer Layer

Construction of H may proceed by several means depending on the chip and available tooling. For chips with published hardware description languages (VHDL, SystemVerilog, device tree specifications), the cartographer parses these to extract node counts, interconnect topology, instruction sets, and power domains. For underdocumented or novel chips, it may issue low-level probe sequences—latency measurements, instruction timing loops, memory bandwidth tests, power consumption sampling via external instrumentation—to empirically derive the cost model P.

The result in all cases is a machine-readable description of the chip's *physical possibility space*: what computations are locally admissible at each node, what communication patterns are feasible across each edge, and what physical costs attach to each operation. This is the foundation on which all subsequent compilation rests.

Node States and Energy Topology

An important dimension of the hardware model that static datasheets tend to omit is the *energy topology*: the graph of power states that each node may occupy and the transition costs between them. For a GreenArrays node, this includes at minimum a full-execution state, a port-blocked waiting state (listening for a message from a neighbor), and a deep-idle state. The waiting state consumes dramatically less energy than execu-

tion but more than deep idle. Programming the chip well means keeping most nodes in low-energy states most of the time, with brief excursions into execution triggered by communication events.

We formalize this as a finite automaton $\mathcal{A}_n = (Q_n, \Sigma_n, \delta_n, q_0^n)$ for each node n, where Q_n is the set of power states, Σ_n is the set of triggering events (instruction issue, message arrival, clock interrupt), δ_n is the transition function, and q_0^n is the default idle state. The cost model P then includes the energy cost of each transition $\delta_n(q, \sigma) = q'$ as $P_{\text{trans}}(n, q, \sigma)$.

Minimizing total energy expenditure over a computation trace τ becomes:

$$\text{minimize} \quad \sum_{n \in N} \sum_{t} P_{\text{trans}}(n,\, q_n(t),\, \sigma_n(t)) + P_{\text{exec}}(n,\, \text{instr}_n(t)) + \sum_{e \in E} P_{\text{comm}}(e,\, \text{msg}_e(t))$$

subject to the constraint that the computation trace produces the correct output.

The Program Model

Kernels, Dependencies, and Requirements

Given a high-level task specification, the program decomposer factors it into a *program graph*:

Definition 2 (Program Graph)**.** *A program graph is a tuple*

$$G = (K,\, D,\, R)$$

where:

- *K is a finite set of kernels—self-contained computational units that operate on local data, produce outputs, and communicate through well-defined message interfaces;*
- *$D \subseteq K \times K$ is a directed graph of data dependencies: an edge $(k_i, k_j) \in D$ indicates that k_j requires output from k_i; and*
- *$R : K \to \mathcal{C}$ assigns to each kernel its resource requirements: instruction types needed, minimum local memory, required timing windows, accuracy or reliability constraints.*

Crucially, kernels are designed to be *locally self-contained*. Each kernel communicates only through its message interface—it does not assume a shared global address space. This discipline directly mirrors the computational model of spatial chips, where no global memory exists and all inter-node communication is explicit.

The Decomposer

The decomposer's task is to factor a task specification into G in a way that respects the chip's physical geometry. This requires answering a set of questions that are typically

invisible in mainstream programming:

- What data can remain *entirely local* to a single node, never appearing on any communication link?
- What data *must move*, and along which communication paths?
- Which kernels can remain dormant until triggered by a message, and what is the expected message rate?
- What can be *preloaded* onto a node's stack or local memory before computation begins?
- What can be *delegated to a neighbor* acting as a smart memory or preprocessor, reducing the computation that the primary node must perform?
- What computation can be *avoided entirely* by exploiting structure in the data (sparsity, symmetry, known ranges)?

These questions are not optimizations applied after a naive decomposition. They are the primary structure of the decomposition itself. On a spatial chip, a design that requires dense global communication is not merely inefficient—it may be physically infeasible.

The Compilation Functor

Placement and Routing

The compilation step finds a mapping $\varphi : G \to H$ that places kernels onto nodes, routes dependencies onto communication edges, and assigns wake/sleep schedules consistent with the chip's energy topology.

Definition 3 (Compilation Mapping)**.** *A valid compilation of G into H is a pair (φ_K, φ_D) where:*

- *$\varphi_K : K \to N$ maps each kernel to a node such that $R(k) \subseteq I(\varphi_K(k))$ (the node can execute the kernel) and the memory required by k does not exceed $M(\varphi_K(k))$;*
- *$\varphi_D : D \to \text{Paths}(E)$ maps each dependency (k_i, k_j) to a directed path in E from $\varphi_K(k_i)$ to $\varphi_K(k_j)$;*
- *for each edge $e \in E$, the aggregate communication load from all paths $\varphi_D(d)$ passing through e does not exceed e's bandwidth constraint; and*
- *all timing and synchronization constraints in C are satisfied by the induced schedule.*

The cost of a compilation is:

$$\text{Cost}(\varphi) = \alpha \cdot E_{\text{exec}}(\varphi) + \beta \cdot E_{\text{comm}}(\varphi) + \gamma \cdot L_{\text{critical}}(\varphi) + \delta \cdot M_{\text{pressure}}(\varphi) + \varepsilon \cdot S_{\text{waste}}(\varphi)$$

where E_{exec} is total execution energy, E_{comm} is total communication energy, L_{critical} is critical-path latency, M_{pressure} is peak memory pressure, S_{waste} is synchronization waste (time spent in active waiting rather than dormancy), and $\alpha, \beta, \gamma, \delta, \varepsilon \geq 0$ are domain-specific weighting coefficients.

For Chuck Moore's computational philosophy, the dominant constraint is $\alpha + \beta \gg \gamma$: energy dominates over speed. For hard real-time systems, γ dominates. For memory-constrained microcontrollers, δ dominates. The cost structure is chip-class-specific, and the cartographer derives appropriate defaults from the hardware model.

Code Generation

Once a valid compilation φ has been found, code generation is chip-specific but structurally uniform. For each node $n \in N$, the code generator emits a program that:

1. preloads any required data onto n's local stack or memory;
2. enters a low-energy waiting state, blocked on the first expected input message;
3. upon message arrival, executes the assigned kernel and produces output messages;
4. routes output messages to φ_D-specified neighbors; and
5. returns to the waiting state.

For GreenArrays-style chips, this generates colorForth or Forth primitives that directly encode the stack operations and port reads/writes. For FPGAs, it generates dataflow graphs or HDL modules. For GPUs, it generates CUDA or OpenCL kernels with appropriate memory hierarchy awareness. For neuromorphic chips, it generates spike-routing configurations. The *shape* of the generated code varies; the *structure* of the generation process is common.

Admissibility Geometry and RSVP Correspondence

The Chip as an Admissibility Field

The hardware graph H defines not merely a set of constraints but a *topology of reachability*: the set of computations that are physically possible, at what cost, and along what paths. In the language of the RSVP (Relativistic Scalar-Vector Plenum) framework, H defines an *admissibility field* $\mathcal{A}_H$ over the space of computation-placements.

A computation-placement is a pair (G, φ): a program graph together with a specific mapping into the hardware. The admissibility field assigns to each such pair a value in $[0, 1]$ reflecting whether the placement is feasible within the chip's hard constraints, with the value decreasing as constraints are approached and reaching zero outside the feasible region.

Definition 4 (Admissibility Field)**.** *The admissibility field of a hardware graph H is a function*

$$\mathcal{A}_H : \{(G, \varphi)\} \to [0, 1]$$

defined by:

$$\mathcal{A}_H(G, \varphi) = \prod_{c \in C} \sigma_c(G, \varphi)$$

where σ_c is a soft satisfaction measure for constraint c, equal to 1 when c is fully satisfied and 0 when violated. Hard constraints enter as $\{0, 1\}$-valued factors; soft constraints (energy budgets, latency preferences) enter as smoothly interpolated values.

Compilation is then constrained projection: finding the highest-quality placement within the admissible region,

$$\varphi^* = \arg \max_{\varphi : \mathcal{A}_H(G,\varphi)=1} \big(-\text{Cost}(\varphi)\big).$$

Instruction Execution as Local Entropy Expenditure

The RSVP framework treats entropy not as a global thermodynamic quantity but as a local, field-valued measure of irreversible state transitions. Instruction execution on a spatial chip is a direct physical instantiation of this idea: each instruction toggles a set of transistors, dissipating energy proportional to the number of $0 \to 1$ and $1 \to 0$ transitions in the affected registers, buses, and decode circuitry. The chip's physical cost model P is essentially a local entropy expenditure map.

Idle states correspond to low-transition basins: the node is in a metastable configuration that changes slowly or not at all, minimizing entropy production. Communication events are brief, spatially localized entropy expenditures that propagate along admissibility channels (the edges E) to trigger state changes in neighboring nodes.

Stack locality in Forth-style architectures corresponds to compressed trajectory memory: the computation's recent history is encoded in the top few cells of the stack, which are physically implemented as fast registers, minimizing the energy cost of accessing state from the recent past.

Blocking Behavior as Sparse Event Activation

The blocking behavior that Moore describes—a node enters a low-energy waiting state until a message arrives from a neighbor, then briefly executes and returns to dormancy—is structurally identical to sparse-event cognition models in neuromorphic computing and asynchronous neural signaling. In both cases, the energy cost of a computation is proportional to the rate and magnitude of state-change events, not to the passage of time per se.

In RSVP terms, the computation trace τ is a sequence of brief, spatially localized trajectory segments separated by intervals of near-zero field activity. The chip is not a continuously executing machine; it is a field of cooperative local operators that activate opportunistically when boundary conditions (incoming messages, clock edges, sensor events) make activation necessary.

Proposition 1 (Energy Minimality as Trajectory Sparsity)**.** *A computation τ on hardware graph H is energy-minimal if and only if the induced activation sequence $\mathcal{E}(\tau) = \{(n, t) : n \text{ executes at time } t\}$ is sparse in both the node dimension (few nodes active at any moment) and the temporal dimension (each active node's active intervals are brief relative to its dormant intervals).*

This is not merely a performance observation. It is a statement about the physical structure of good programs on spatial chips: a well-written program on a GreenArrays-like machine looks like a sparse activation field, not a dense computation trace.

Compiler Architecture

The Four Layers

The Silicon Cartography compiler is organized into four layers, each with a well-defined input/output contract.

Layer 1: Hardware Cartographer. Accepts chip description (HDL, device tree, empirical probe results) and produces the hardware graph $H = (N, E, I, M, C, P)$. Responsibilities: topology extraction, instruction set parsing, memory map construction, power-state automaton derivation, cost model calibration.

Layer 2: Capability Model. Refines H into an executable constraint map: a queryable data structure supporting questions about local admissibility, edge capacity, power-state transitions, and routing feasibility. This layer is the foundation of all subsequent optimization; it must be consistent and queryable in logarithmic time.

Layer 3: Program Decomposer. Accepts a high-level task specification and the capability model, and produces a program graph $G = (K, D, R)$ with kernel interfaces designed to fit the chip's communication model. The decomposer applies locality analysis (what can stay local?), communication analysis (what must move and how often?), and sparsity analysis (what computation can be avoided?).

Layer 4: Placement, Routing, and Code Generator. Solves the constrained optimization problem to find $\varphi^* : G \to H$, then emits chip-specific code for each assigned node. The optimization may use constraint programming, simulated annealing, or reinforcement learning depending on chip complexity; the code generator is chip-class-specific.

The Compiler's Role

Chuck Moore's insistence that the compiler "merely translates" and performs no optimization is not a limitation of Forth but a design philosophy: the programmer should understand the hardware geometry directly, not delegate that understanding to an optimizer. Silicon Cartography is consistent with this philosophy at the strategic level while acknowledging that on complex multi-node chips, the placement and routing problem

is too large for unaided human cognition.

The resolution is that the compiler's optimization is *structural*, not *semantic*. It does not rewrite the kernel logic; it finds the best physical placement of kernels that the programmer has already designed to be hardware-aware. The programmer still reasons about locality, communication, and energy. The compiler automates the combinatorial search over physical placements.

In this sense, the Silicon Cartographer is less like a traditional optimizing compiler and more like a *constraint solver over hardware geometry*: it finds the mapping φ that satisfies all hard constraints and minimizes the energy cost model, but it trusts the programmer to have designed kernels that are worth placing well.

Target Chip Classes

The Silicon Cartography framework is designed to generalize across chip classes, with chip-specific code generation at Layer 4. We briefly characterize the four primary targets.

Mesh Processors (GreenArrays-style)

The GreenArrays GA144 contains 144 F18A cores in a mesh, each with 64 words of RAM, an 18-bit instruction word, a data stack, and a return stack. Communication is synchronous: a read from a port blocks until a write arrives from the neighboring node, and vice versa. This blocking synchrony is the chip's primary coordination mechanism.

For this class, the cartographer extracts the mesh topology as a grid graph, assigns Forth/colorForth instruction sets to each node, and models communication as synchronous rendezvous with energy cost proportional to the number of transitions on the shared port wire. The code generator emits colorForth-style source for each node.

FPGAs

FPGAs present a different constraint geometry: the computation itself is spatially configurable, but routing resources are finite and congestion in the interconnect fabric is a primary bottleneck. The cartographer models the FPGA fabric as a graph of configurable logic blocks, DSP slices, block RAMs, and routing switches with congestion-dependent latency and energy costs. The code generator emits HDL or a high-level synthesis specification.

GPUs

GPUs offer massive parallelism but impose strict requirements on memory access patterns: coalesced global memory access, shared memory bank-conflict avoidance, and occupancy optimization. The cartographer models the GPU as a hierarchical graph of streaming multiprocessors, each with shared memory and a warp-parallel execution model. The program decomposer must respect the SIMT execution model. The code generator emits CUDA or OpenCL kernels.

Neuromorphic Chips

Neuromorphic chips (Intel Loihi, IBM TrueNorth, SpiNNaker) compute through spike propagation on fixed or configurable connection graphs. Energy is consumed primarily at spike events, making sparsity of activation the central optimization target. The cartographer models the neural network graph with synaptic weights and routing tables. The code generator emits spike-routing configurations and neuron parameter settings.

The Ecological Metaphor

Moore's language suggests an ecological rather than mechanical understanding of the chip: nodes occupy niches, communication channels are migration paths, energy is a shared resource to be conserved, and the programmer's job is not to command execution but to design an ecosystem in which the right computations emerge from local interactions.

This metaphor is not merely rhetorical. It suggests a generalization of the Silicon Cartography framework toward *machine ecology*: the study of how computational load distributes across a spatial chip under realistic workloads, how communication patterns create congestion or underutilization in particular chip regions, and how programming choices shift the equilibrium of this distribution.

A machine ecology analysis of a compiled program would track:

- *Node utilization*: the fraction of time each node spends in each power state, revealing hot spots and idle regions;
- *Edge saturation*: the fraction of available bandwidth consumed by each communication link, revealing routing bottlenecks;
- *Entropy production*: the total energy expenditure per node per unit time, identifying which program components are thermodynamically expensive; and
- *Dormancy patterns*: the temporal distribution of idle intervals, characterizing how well the program exploits the chip's low-energy states.

This analysis closes the loop with the hardware cartographer: the measured execution ecology of a deployed program refines the cost model P, improving future compila-

tions.

Formal Properties

Feasibility

Not every program graph G admits a valid compilation into every hardware graph H. Feasibility requires that the kernel resource requirements R are satisfiable within the node capabilities I and M, that data dependencies D can be routed through the edge set E without violating bandwidth constraints, and that all timing constraints in C can be met. Feasibility checking is in general NP-complete (by reduction to graph coloring), but for structured chip topologies—grids, trees, hypercubes— polynomial-time algorithms exist.

Theorem 1 (Feasibility on Grid Meshes)**.** *For a program graph $G = (K, D, R)$ and a grid mesh hardware graph H with uniform node capabilities and edge capacities, feasibility is decidable in polynomial time in $|K|$ and $|N|$ when all kernels require the same instruction types and $|D| = O(|K|)$.*

We omit the proof here; it follows from the fact that routing on a grid mesh with bounded-degree program graphs reduces to a planar embedding problem.

Optimality

Finding φ^* that minimizes Cost(φ) over all valid compilations is in general NP-hard (by reduction to quadratic assignment). In practice, for small chips (fewer than a few hundred nodes), exact solutions via integer programming are feasible. For larger chips, heuristic methods—simulated annealing, genetic algorithms, message-passing belief propagation—find near-optimal placements efficiently.

Compositionality

A key property of the program graph formalism is compositionality: if G_1 and G_2 are program graphs with compatible message interfaces, their composition $G_1 \otimes G_2$ is a well-formed program graph. Valid compilations of G_1 and G_2 into disjoint subsets of N compose into a valid compilation of $G_1 \otimes G_2$, provided the routing paths for inter-graph communication are feasible. This allows incremental compilation and modular system design on spatial chips.

Fractal Topology Families

Beyond the Uniform Grid

The GA144 is a flat rectangular mesh: eight rows by eighteen columns, with each F18A node communicating through four blocking ports to its cardinal neighbors. This topology is elegant and minimal, but it represents only one point in a much larger space of spatial chip architectures. The Silicon Cartography framework is deliberately topology-agnostic: the hardware graph H accommodates any connected graph, and the compilation functor adapts accordingly. This section develops a family of procedurally generated chip topologies that extend the flat mesh toward recursive, hierarchical, and fractal structures, each with characteristic locality properties and programming disciplines.

Fractal Island Mesh

The simplest departure from the flat grid is recursive subdivision. Divide the node budget into a $k \times k$ array of *islands*, each containing a small local mesh of worker nodes. Islands connect through dedicated *bridge nodes* whose sole function is inter-island routing. The resulting two-level hierarchy is described by the generator

$$N_{\text{island}} = N_{\text{local}}^{(k \times k)} \cup N_{\text{bridge}},$$

where N_{bridge} forms a secondary grid over the island topology. Bridge nodes carry higher-bandwidth, higher-energy communication budgets than local worker nodes, and may be assigned larger local memory to buffer inter-island traffic.

The programming discipline that this topology enforces is the central principle of Silicon Cartography in recursive form: keep fast loops inside islands, export only summaries across bridges, and never move raw state unless unavoidable. Algorithms that exhibit local coherence and occasional long-range exchange—field relaxation, graph diffusion, image processing, sparse neural inference—map naturally onto this structure. The island boundary acts as an admissibility barrier, and the compiler's locality preservation constraint

$$d_G(\varphi(k_i), \varphi(k_j)) \leq L \cdot \|\sigma(k_i) - \sigma(k_j)\|$$

is applied at two scales: within an island for local kernels and across islands for summary reduction kernels.

The recursion may be applied again: a level-3 architecture organizes islands of islands, with super-bridge nodes handling inter-cluster routing. This produces a three-level hierarchy that mirrors the cache hierarchy of conventional processors but with explicit programmer control over every level of the communication geometry.

Tree-of-Meshes Architecture

An alternative to recursive subdivision is the tree-of-meshes: a rooted tree in which leaves perform sensing or local computation, branch nodes perform reduction and routing, and the root node coordinates global state. Each leaf cluster contains a small flat mesh; sibling clusters may carry optional lateral cross-links with probability p, creating a small-world connectivity structure on top of the tree backbone.

The generator is:

1. Instantiate one root node with elevated memory $M_{\text{root}} \gg M_{\text{leaf}}$.
2. Attach k branch nodes, each with intermediate memory M_{branch}.
3. Attach k leaf-cluster meshes to each branch node.
4. With probability p, add lateral links between sibling clusters.

This architecture is suited to reductions: many local readings aggregate into regional summaries and ultimately into global decisions. Sensor swarms, audio analysis pipelines, distributed event detection, and biological-style hierarchical inference all exhibit this pattern. The tree structure makes the communication pattern maximally explicit: every upward edge is a reduction, every downward edge is a control signal, and lateral edges are optional coherence corrections.

The hardware graph for this topology has bounded degree at leaf nodes (four neighbors in the local mesh plus one upward branch link) and higher degree at branch and root nodes (up to k downward links plus one upward link). The cost model must assign appropriately higher energy budgets to branch and root nodes while enforcing that leaf nodes spend the vast majority of their time dormant.

Hilbert-Curve Chip

A Hilbert curve generates a bijective map between the integers $\{0, \ldots, 2^{2n} - 1\}$ and the cells of a $2^n \times 2^n$ grid that preserves locality: cells nearby in the linear index are nearby in the grid. A Hilbert-curve chip is physically a flat grid, but the compiler treats node addresses as Hilbert-curve indices, so that streaming access along the logical order corresponds to physically local communication.

The Hilbert ordering is generated recursively. For a 4×4 grid the ordering visits cells in the U-shaped pattern characteristic of the space-filling curve, and at each recursive subdivision the pattern rotates and reflects to maintain the locality property. The result is that any contiguous segment of the linear traversal corresponds to a roughly square subregion of the grid, giving optimal locality for both streaming and tiled computation.

For the compiler, this means that data with one-dimensional serial structure— image scanlines, audio sample streams, compressed symbol sequences—should be mapped onto the chip along the Hilbert index order. Operations that require spatial locality

(field relaxation, image tile convolution, compression context modeling) automatically achieve physical locality as a consequence. The programmer specifies the semantic ordering of data; the compiler uses the Hilbert bijection to find the corresponding physical placement.

Sierpinski Compute Fabric

The Sierpinski-fabric architecture deliberately removes portions of the grid at multiple scales. Given a base square lattice, the generator recursively excises the center subregion at each level of subdivision, leaving a fractal structure of connected compute cells separated by voids. These voids are not defects; they serve as routing reservoirs, thermal gaps, analog interface corridors, or dedicated memory channels.

At level m the node set is:

$$S_{m+1} = \bigcup_{i=1}^{3} f_i(S_m), \qquad f_i(x) = \tfrac{1}{2}x + v_i,$$

where v_1, v_2, v_3 are the vertices of the base triangle and each f_i is a contraction by factor $\frac{1}{2}$. The resulting communication graph has a characteristic fractal dimension $d_f = \log 3 / \log 2 \approx 1.585$ and a degree distribution that is highly non-uniform: nodes near the boundary of each sub-triangle have higher degree than interior nodes.

The power-density motivation is central. On conventional dense grids, active regions near each other produce overlapping thermal fields that raise the local temperature, increasing leakage current and degrading reliability. The Sierpinski voids provide recursive thermal separation between active subregions, allowing higher per-node power budgets in each compute cell without exceeding global thermal limits. Programming for this topology concentrates dense computation within the connected compute cells and uses boundary nodes as the only communication points between sub-regions.

Dragon-Curve Pipeline

The dragon-curve architecture is a folded asynchronous pipeline. A long linear computation—a filter chain, a signal processing cascade, an event detection hierarchy—is physically folded into a compact recursive curve that fits within the chip's spatial bounds. The folding preserves the logical order of the pipeline while minimizing the physical wire length between adjacent pipeline stages.

The dragon curve is generated by repeatedly replacing each directed segment with a right-angled pair of segments, alternating orientation at each level. The result is a space-filling path with the property that each segment is adjacent to the next in the logical pipeline ordering and also physically close to two or three other pipeline stages. This locality enables a "pass-forward" programming discipline: each node wakes when a value arrives from its predecessor, performs a local transform, forwards the result to its

successor, and returns to dormancy. Feedback is used only at the recursive fold points, where the curve doubles back on itself.

This architecture is ideal for streaming computations where the data flow is fundamentally one-directional but the chip is two-dimensional: audio sample processing, image scanline filtering, sensor event pipelines, and data compression.

Apollonian Heterogeneous Fabric

The Apollonian architecture abandons node uniformity entirely. Drawing on the geometry of circle packing, it places large high-capability nodes at the center of the chip and recursively smaller specialized nodes around them. The largest central region handles memory-intensive or globally coordinating computation. Medium regions handle DSP, routing, and analog interfacing. Small peripheral nodes handle preprocessing and event filtering. Tiny always-on nodes at the outermost periphery perform wake-up detection at near-zero power.

The programming rule is: use tiny nodes for event detection, use small nodes for preprocessing, use medium nodes to aggregate and route, and wake large nodes only when the accumulated evidence makes full processing necessary. This matches the inference pattern of biological sensory systems, in which peripheral neurons perform massive signal reduction before forwarding only salient events to higher processing regions.

Communication Manifold and Spectral Structure

The Chip as a Metrized Graph

The hardware graph $G = (N, E)$ is not merely topological. It is *metrized*: each edge carries a cost that reflects the physical properties of the wire, buffer, or bus it represents. Define a weighted edge cost

$$w(i, j) = \alpha \ell_{ij} + \beta \tau_{ij} + \gamma \epsilon_{ij},$$

where ℓ_{ij} is the geometric wire length, τ_{ij} is the communication latency, and ϵ_{ij} is the energy required for a single data transfer, with $\alpha, \beta, \gamma \geq 0$ weighting coefficients. This induces a communication metric over all admissible routing paths:

$$d_G(i, j) = \inf_{\pi : i \rightsquigarrow j} \sum_{e \in \pi} w(e).$$

The metric d_G is not symmetric in general: a chip may have asymmetric buses, directional links, or one-way DMA channels. For undirected substrates, d_G is a proper metric and equips the node set with a metric space structure that captures the energetic geog-

raphy of communication.

Graph Laplacian and Diffusion

Define the graph Laplacian $L = D - A$ where A is the weighted adjacency matrix and D is the degree matrix with $D_{ii} = \sum_j w(i,j)$. The Laplacian governs how disturbances propagate across the chip. A message broadcast or synchronization signal injected at node n_0 diffuses according to:

$$\frac{\partial u}{\partial t} = -Lu, \qquad u(0) = \delta_{n_0}.$$

The eigenvalues $0 = \lambda_0 \leq \lambda_1 \leq \cdots \leq \lambda_{|N|-1}$ of L characterize the chip's communication geometry. The algebraic connectivity λ_1 (Fiedler value) measures how well-connected the chip is: a small λ_1 indicates a bottleneck or near-partition, while a large λ_1 indicates high global connectivity. For fractal architectures, the spectrum of L encodes the self-similar structure of the topology and can be computed recursively from the spectra of smaller instances.

The spectral decomposition of the chip graph informs compilation directly. Kernels that must communicate frequently should be placed on nodes that are spectrally close (small eigenvector distance in the low-frequency subspace). Dormant regions correspond to parts of the chip that decouple from the active subgraph, appearing as near-zero entries in the relevant eigenvectors.

Synchronization Sheaves and Global Sections

For asynchronous chips, communication events are not clock-driven but handshake-driven. A port read on node i blocks until a port write arrives from node j. This synchronization constraint can be formalized using sheaf theory.

Define a presheaf F over the open sets of the chip graph G, assigning to each subgraph $U \subseteq G$ the local state space $F(U) = \prod_{i \in U} S_i$ of all nodes in U. For overlapping subgraphs U and V with non-empty intersection, the restriction maps require consistency on the shared boundary:

$$s_U\big|_{U \cap V} = s_V\big|_{U \cap V}.$$

A valid distributed computation is then a global section $s \in \Gamma(G, F)$: an assignment of local states to every node that is consistent across all communication interfaces simultaneously. Deadlock corresponds to the absence of a global section: some combination of blocking port states admits no consistent resolution. The sheaf-theoretic formulation makes this failure mode precise and suggests detection algorithms based on Čech cohomology obstruction classes in $H^1(G, F)$.

Iterated Function Systems and Recursive Chip Generation

For fractal chip topologies, the node set itself is generated by an iterated function system $\mathcal{F} = \{f_1, \ldots, f_k\}$ acting on an embedding space $X \subseteq \mathbb{R}^2$. Each f_i is a contraction map. The node set at recursion depth m is:

$$N_m = \bigcup_{i_1,\ldots,i_m} f_{i_1} \circ \cdots \circ f_{i_m}(N_0).$$

The communication graph at depth m is the union of the images of the depth-$(m-1)$ graph under each f_i, together with the inter-image bridge edges that connect adjacent subregions. This recursive structure means that the hardware graph H_m satisfies a self-similarity relation:

$$\mathbf{Chip}_m \simeq \coprod_{i=1}^{k} \mathbf{Chip}_{m-1}^{(i)},$$

where the coproduct is in the category whose objects are spatial chip regions and whose morphisms are admissible communication transformations. Compilation for fractal chips exploits this recursive structure: a kernel decomposition for depth m is obtained by recursively solving the depth-$(m-1)$ problem for each sub-region and then solving the inter-region routing problem for the bridge nodes.

Procedural Topology Language

A Unified Descriptor Schema

The taxonomy of chip topologies developed in Section 5 suggests a general *procedural topology language*: a declarative schema in which a chip architecture is specified not by enumerating nodes and edges but by giving a generator, a recursion depth, a node-type assignment rule, and a cost model. The compiler then instantiates the hardware graph H from this specification before beginning placement.

A chip specification in this language has five components. The *topology generator* is a rule for constructing the node set and edge set: flat grid, recursive island, Hilbert curve, Sierpinski subdivision, dragon-curve pipeline, or a composition thereof. The *node capability assignment* maps node positions in the generated topology to instruction sets, memory capacities, and power domains, potentially varying by position (as in the Apollonian fabric where node size decreases with distance from the center). The *communication geometry* specifies the edge types: blocking synchronous ports, buffered asynchronous channels, shared-memory regions, or broadcast buses, each with its own cost function $w(i, j)$. The *energy model* assigns active and idle power to each node type and transition type. The *I/O boundary* identifies which nodes interface with external pins, analog

inputs, clock sources, or off-chip memory.

Together these five components fully determine the hardware graph H and its cost model P, from which the cartographer layer constructs the executable constraint map.

Compiler Policy Parameters

Beyond the hardware specification, the procedural topology language includes a set of *compiler policy parameters* that configure the optimization behavior of the placement and routing layer:

- `minimize_instruction_count`: prefer placements that require fewer total instructions by exploiting hardware-specific primitives;
- `minimize_message_distance`: place communicating kernel pairs on neighboring or nearby nodes;
- `prefer_sleep_over_polling`: replace active waiting loops with blocking port reads wherever the communication pattern permits;
- `preserve_locality`: enforce the Lipschitz condition $d_G(\varphi(k_i), \varphi(k_j)) \leq L\|\sigma(k_i) - \sigma(k_j)\|$ with a specified Lipschitz constant L; and
- `thermal_separation`: apply a minimum distance constraint $d_G(\varphi(k_i), \varphi(k_j)) \geq d_{\text{min}}$ for kernel pairs whose simultaneous execution would exceed the local thermal budget.

These parameters allow domain-specific tuning. A sensor swarm application might prioritize sleep maximization and message-distance minimization. A hard real-time control system might override sleep preferences to guarantee bounded latency. A thermally constrained wearable device might impose strict thermal separation constraints. The procedural topology language thus separates hardware description from compilation policy, making both independently reusable.

Visualization Framework

The Inadequacy of Block Diagrams

A spatial chip in the sense defined here is not a collection of processors connected by labeled arrows. It is a dynamic energetic manifold with locality structure, recursive topology, wake/sleep state transitions, communication flow, and admissibility constraints that vary continuously across the substrate. A conventional block diagram is structurally inadequate to represent these six interacting dimensions simultaneously:

$$(\text{topology},\ \text{activity},\ \text{energy},\ \text{communication},\ \text{memory},\ \text{time}).$$

A visualization framework suited to spatial chips must represent all six dimensions,

ideally at multiple spatial and temporal scales. This section develops such a framework.

Spatial Graph Embedding with Physical Metrics

The foundational layer is a spatial graph embedding $\varphi : N \to \mathbb{R}^2$ that places each node at its physical silicon location rather than at an arbitrary graph-layout position. Node size encodes memory capacity:

$$r_i \propto \log(1 + m_i),$$

so that memory-rich nodes (branch and root nodes in a tree-of-meshes, central nodes in an Apollonian fabric) are visually prominent. Edge thickness encodes bandwidth:

$$w_{ij} \propto b_{ij},$$

so that high-bandwidth bridges appear as thick channels and low-bandwidth point-to-point links appear as thin filaments. The resulting image is not an abstract graph diagram but a topographic map of the chip's resource geography.

Activity Heatmaps and Dormancy Fields

Activity visualization extends the spatial embedding with a temporal dimension. Define the time-averaged activity of node i over a computation trace of duration T:

$$\bar{\chi}_i = \frac{1}{T} \int_0^T \chi_i(t)\, dt,$$

where $\chi_i(t) \in \{0, 1\}$ is the activity indicator. Rendering $\bar{\chi}_i$ as a color field over the spatial embedding produces an *activity heatmap*: high-activity nodes appear in warm colors, dormant nodes appear dark. A critical insight for these architectures is that darkness is success. An activity heatmap for a well-compiled program on a GreenArrays-like chip should show most nodes as dark, with brief bright pulses propagating along the communication paths defined by the program's data flow.

Thermal accumulation introduces a related field:

$$T(x, t) = \int_0^t k_{\text{diff}}(x, x') \cdot \rho_e(x', t')\, dx'\, dt',$$

where ρ_e is the local energy dissipation density and k_{diff} is a thermal diffusion kernel. Rendering $T(x, t)$ as a slowly evolving overlay on the activity map reveals thermal hotspots that may not be visible in the instantaneous activity alone.

Event Wave Propagation

For asynchronous chips, clock-cycle diagrams are meaningless: there is no global clock. Instead, the appropriate temporal visualization is the *event wave*: only actual state transitions are rendered. A transition at node i at time t appears as a brief luminous pulse at position $\varphi(i)$. The pulse decays with a time constant τ_{vis} chosen to make individual events visible without creating persistent glow. When the computation involves a message cascade (one node's output triggering a neighbor's computation triggering the next neighbor's), the resulting visualization shows a wave of pulses propagating across the chip surface, resembling neural spike propagation, mycelial signaling, or fluid turbulence.

Repeated high-traffic communication paths accumulate visible luminosity over many computation cycles, forming what might be called *semantic highways*: the chip's dominant information arteries become geometrically visible without any explicit routing annotation.

Causal Braid Diagrams

For programs with complex synchronization structure, the event-wave visualization obscures the causal order of transitions. A *causal braid diagram* makes causality explicit by assigning each node a vertical strand in a spacetime diagram (time increasing downward) and representing communication events as crossings between strands. The braid structure reveals synchronization dependencies, bottlenecks, deadlock cycles, burstiness, and idle gaps in a form that is directly analogous to string diagrams in monoidal category theory.

For asynchronous systems, causal braid diagrams are vastly more informative than clock-cycle diagrams, because they show the actual causal dependency structure of the computation rather than its alignment to an external time reference. A computation with no unnecessary synchronization appears as a braid with no unnecessary crossings; redundant synchronization points appear as spurious entanglements that a compiler optimization pass can remove.

Multiscale Semantic Zooming for Fractal Topologies

For fractal chip topologies, a static rendering at any single scale is inadequate: the recursive structure that defines the architecture's properties appears only across multiple scales simultaneously. A *multiscale semantic zoom* renders the chip at multiple levels of detail simultaneously, with the visible level of detail varying continuously as the viewer zooms in or out.

At the coarsest zoom level, entire islands or subtrees appear as single representative nodes, with inter-island communication edges rendered as thick channels. At interme-

diate zoom, the internal structure of each island becomes visible, with local communication patterns emerging. At the finest zoom level, individual stack states and instruction sequences appear. This is directly analogous to geographic information systems in which political boundaries, cities, roads, and buildings emerge at successive zoom levels.

For fractal topologies generated by iterated function systems, the self-similarity of the structure means that the same visual grammar applies at every zoom level: the viewer exploring a Sierpinski-fabric chip sees the same triangular-void pattern at the scale of the whole chip, at the scale of a sub-region, and at the scale of an individual compute cell's neighborhood.

Energy Landscape Terrain

Perhaps the most powerful single visualization for constraint-aware compilation is the *energy landscape terrain*: a three-dimensional surface over the chip's spatial embedding in which the height $z(x, y, t)$ equals the instantaneous energy dissipation rate $E(x, y, t)$. Rendering this surface as a dynamic terrain map transforms the chip into a living landscape: active computation appears as mountains and peaks, dormant regions appear as valleys and plains, and communication events appear as traveling ridges.

Efficient programs produce sparse ripples and localized transient peaks that quickly relax back to the valley floor. Inefficient programs produce flat global elevation (all nodes consuming energy simultaneously), persistent plateaus (nodes stuck in polling loops), or chaotic turbulence (synchronization failures causing repeated retries). The terrain metaphor makes the energy cost of programming decisions immediately visible and viscerally comprehensible.

This is a direct geometric expression of Moore's core principle: unnecessary switching is waste, and waste is visible as unnecessary height in the energy landscape. A programmer who can see the terrain their program creates on a spatial chip can reason about energy in the same intuitive way that a civil engineer reasons about drainage and runoff.

Compiler Placement Geometry Overlay

A final visualization layer represents the compiler's output directly: the mapping $\varphi : K \to N$ from computational kernels to physical nodes. Each kernel is rendered as a colored territorial overlay on the chip's spatial embedding, with the kernel's color encoding its computational role (sensing, reduction, output, control). Data dependencies between kernels appear as elastic bands stretching between their territorial boundaries.

Compilation quality is then visually immediate. A placement with good locality shows each kernel's territory as a compact, convex region, with short elastic bands connecting communicating kernels. A placement with poor locality shows kernels stretched across

the chip, with long elastic bands crossing many intermediate nodes. Thermal hotspots appear as multiple high-energy kernels occupying adjacent territories. Synchronization bottlenecks appear as many elastic bands converging on a single bridge node.

This overlay closes the loop between the cartographer's hardware model and the programmer's algorithmic intent: it shows, in physical terms, exactly where the computation lives and how information moves between its components.

Spacetime Activity and the Thermodynamic Cost of Computation

Activity Volume

A spatially distributed computation induces a *spacetime activity volume*: the integral over time of the number of active nodes,

$$\mathcal{V} = \int_0^T |A(t)|\, dt,$$

where $A(t) \subseteq N$ is the set of nodes transitioning at time t. Minimizing $\mathcal{V}$ is equivalent to minimizing total dynamic energy expenditure when per-node transition costs are uniform. For non-uniform costs, the weighted activity volume

$$\mathcal{V}_w = \int_0^T \sum_{i \in A(t)} \eta_i(t)\, dt$$

is the appropriate objective, where $\eta_i(t)$ is the instantaneous transition energy of node i. This is the quantity that the cost functional $J(\varphi)$ approximates in the static compilation setting.

The activity volume formulation makes explicit what mainstream software metrics obscure: the relevant measure of a computation's cost is not its wall-clock duration (which conflates execution time with idle time) and not its instruction count (which does not account for spatial parallelism) but the total spacetime mass of its physical transitions.

Sparse Activation as a Design Criterion

The proposition established in Section 4—that energy minimality is equivalent to trajectory sparsity—extends naturally to the spacetime framework. A computation is *maximally sparse* when at each instant t the active node set $A(t)$ is as small as possible while still making progress toward the desired output. This corresponds to the ideal of asynchronous event-driven computation: no node executes unless a communication event has made its execution necessary.

Achieving maximal sparsity requires that the program graph G be decomposed into ker-

nels with minimal data dependencies, that the compilation mapping φ places communicating kernels on neighboring nodes (minimizing the number of intermediate routing hops that must activate), and that the code generation layer replaces all polling loops with blocking port reads. These three requirements—minimal dependency graph, locality-preserving placement, blocking communication—are the core conditions of the Silicon Cartography discipline.

Connection to Thermodynamic Computation Theory

The identification of computation with physical state transitions connects Silicon Cartography to the theoretical foundations of thermodynamic computation. Landauer's principle establishes that erasing one bit of information requires a minimum energy dissipation of $k_B T \ln 2$, where k_B is Boltzmann's constant and T is the physical temperature. This bound applies to any logically irreversible operation: any computation that discards information must pay a thermodynamic price.

For practical computing on spatial chips, Landauer's bound is not yet the limiting factor—switching energy in current CMOS technology exceeds the Landauer limit by several orders of magnitude. But the direction of approach is clear: as transistors shrink and switching energies fall, the thermodynamic cost of logically irreversible operations will become increasingly significant. Programming disciplines that minimize unnecessary state transitions—like the one Moore describes and Silicon Cartography formalizes—are not merely practical optimizations for the present. They are preparations for a computational regime in which thermodynamic cost is not an engineering approximation but the fundamental constraint.

In this sense, Moore's moral language about waste is not rhetorical. Every unnecessary transistor transition is a physical irreversibility—a small but real contribution to the entropy of the universe. A programmer who takes this seriously is not performing premature optimization; they are acknowledging the physical reality of computation.

Discussion

Recovery of Physical Meaning

The central claim of Silicon Cartography is that energy-minimal computing on spatial chips recovers a form of *physical meaning* in programs that mainstream software engineering has erased. In a conventional compiled language, the connection between a line of source code and the physical transistor transitions it induces is opaque: it passes through an optimizing compiler, an instruction scheduler, a microarchitectural pipeline, a cache hierarchy, and a memory controller, each of which transforms the computation in ways invisible to the programmer.

On a spatial chip programmed through Silicon Cartography, the connection is trans-

parent. Each kernel is physically located at a node. Each communication is a physical wire transition. Each idle period is a physical dormancy. The cost model P is not an approximation of something deeper; it is the direct physical reality of the computation.

This transparency is not merely aesthetically satisfying. It is practically necessary for programming chips at the energy frontier. A programmer who cannot reason about the physical cost of their design choices cannot make them well.

From Sequential Control to Constraint Propagation

Traditional programming languages model computation as the ordered execution of symbolic instructions over centralized mutable state. This perspective emerged from serial machine architectures in which global synchronization was technologically convenient, and it has been inherited by nearly every layer of the mainstream software stack—operating systems, garbage collectors, memory models, sequential consistency protocols. The question that rarely gets asked is whether this inheritance reflects something necessary about computation or something contingent about a particular engineering era.

The framework proposed here suggests the latter. Computation, viewed physically, is constrained propagation across admissibility manifolds: programs are not sequences of instructions but structured activation geometries embedded within hardware topology. Under this interpretation, control flow becomes activation flow, memory locality becomes geometric proximity, synchronization becomes compatibility of propagation fronts, and scheduling becomes manifold navigation under energetic constraints.

The distinction is not merely semantic. Sequential models externalize coordination into scheduling mechanisms, synchronization primitives, cache coherence protocols, and global execution semantics. Constraint-propagation models internalize coordination into the admissibility structure itself. This internalization eliminates an entire class of coordination overhead that the sequential model must introduce and then laboriously manage. The resulting architecture more closely resembles naturally distributed systems—vascular networks, neural tissue, reaction-diffusion systems—where coherent global behavior emerges through local propagation rules and energetic admissibility conditions rather than centralized orchestration.

This shift has a historical parallel in physics: the transition from force-centric to field-centric formulations. Just as modern physics replaced action-at-a-distance mechanisms with continuous field descriptions, future computational systems may increasingly replace imperative control structures with distributed admissibility geometries governing local activation propagation. The field is not weaker than the force; it is a more faithful account of what is actually happening.

An important strand of theoretical computer science has argued for a related reconception from the symbolic side: that concurrency is simpler and more primitive than sequentiality, and that sequential execution is best understood as a particular serial-

ization imposed over a fundamentally distributed substrate. Process-network semantics, communicating-process algebras, dataflow models, and invocation-based process expression all move in this direction at the symbolic level. The present framework extends that reconception into explicitly physical territory: spatial embedding, thermodynamic cost, energy topology, routing metrics, and hardware admissibility constraints transform the philosophical argument into an engineering discipline with concrete implementation consequences.

The Thermodynamic Cost of Serialization

Sequential execution imposes a thermodynamic burden that abstract computational models routinely conceal. Serialization forces computational regions that could otherwise evolve independently to synchronize around a globally ordered execution schedule. This synchronization introduces energetic overhead through clock distribution, pipeline stalling, cache coherence maintenance, global arbitration, and repeated state serialization and deserialization. In sparse activation systems operating near the energy frontier, these coordination costs can exceed the energy spent on useful computation by orders of magnitude.

The present framework treats serialization not as a neutral abstraction but as a physically consequential geometric restriction. A globally ordered execution schedule constrains the dimensionality of allowable activation configurations: at each moment, only the single active node specified by the program counter may transition, while all other nodes remain artificially frozen. This is an extreme restriction of the accessible manifold of low-energy computational states.

By contrast, asynchronous sparse activation permits computational activity to distribute across all locally admissible regions simultaneously. Energy expenditure concentrates near active computational frontiers while dormant regions remain quiescent. The spacetime activity volume $\mathcal{V}_w$ is minimized not by accelerating a single execution thread but by allowing the computation's natural distributed geometry to evolve without artificial serialization constraints.

This observation suggests that many contemporary inefficiencies in large-scale embedded and distributed computation may not arise primarily from transistor limitations or memory bandwidth—the usual suspects—but from geometric mismatches between sequential symbolic abstractions and the distributed thermodynamic structure of physical hardware. Addressing these mismatches requires not faster serialization but the dissolution of unnecessary serialization: a return to programming disciplines that negotiate with the physical geometry of computation rather than abstracting it away.

Three Senses of Locality

The compiler policy parameter `preserve_locality` encodes a Lipschitz condition $d_G(\varphi(k_i), \varphi(k_j)) \leq L\|\sigma(k_i) - \sigma(k_j)\|$ that constrains the mapping from semantic to physical distance. This constraint conceals an important conceptual distinction among three senses of locality that the framework simultaneously manages.

Physical locality is the most direct: two nodes are physically local when they are adjacent or nearby in the hardware graph G, meaning that communication between them costs little energy and imposes minimal latency. Physical locality is determined entirely by chip geometry and is independent of the computation being performed.

Logical locality is a property of the program graph G: two kernels are logically local when they share a direct data dependency—when one kernel's output is the immediate input of the other. Logical locality is determined by the structure of the computation and is independent of the hardware.

Semantic locality is the subtler quantity: two kernels are semantically local when they operate on conceptually related data or perform functionally coherent operations, even if they are not directly connected in the dependency graph. A sensor kernel and its associated calibration kernel are semantically local even if a reduction kernel sits between them in the dependency chain.

The Lipschitz condition above maps semantic distance $\|\sigma(k_i) - \sigma(k_j)\|$ to physical distance d_G, requiring that the placement preserve semantic geometry. This is the deepest of the three locality conditions because it connects algorithmic intent to physical substrate: a well-compiled program is one in which the chip's physical geometry reflects the semantic structure of the computation it implements.

This connection has broad resonance. In sparse representation learning, the goal is to find embeddings in which semantic similarity maps to geometric proximity in a low-dimensional space. In biological cortical organization, functionally related neurons are spatially clustered, and this clustering reduces the wiring length and energy cost of correlated activations. In graph partitioning, the goal is to assign computationally related tasks to the same processor to minimize inter-processor communication. Silicon Cartography unifies these concerns under a single geometric discipline: the placement φ is an embedding of the program's semantic geometry into the chip's physical geometry, and compilation quality is measured by how faithfully this embedding preserves the relevant metric structure.

Causal Geometry and Temporal Admissibility

Time has been implicit throughout this framework in traces, schedules, and activity volumes. For asynchronous spatial chips, where no global clock coordinates execution, a more explicit treatment of the temporal structure of computation is necessary. The appropriate formalism is a causal partial order.

Definition 5 (Causal Precedence)**.** *Given a computation trace on hardware graph H, define the causal precedence relation $\prec$ on execution events (k_i, t_i) by:*

$$(k_i, t_i) \prec (k_j, t_j)$$

if and only if kernel k_j depends on output from k_i and the information emitted by k_i at time t_i can physically reach $\varphi(k_j)$ before time t_j, i.e. $t_j \geq t_i + d_G(\varphi(k_i), \varphi(k_j))/c_{\max}$, where $c_{\max}$ is the maximum signal propagation speed on the chip.

The causal precedence relation defines a partial order on execution events: it is irreflexive, asymmetric, and transitive. The forward causal cone of an event (k_i, t_i) is the set of all events that can be influenced by it:

$$\mathcal{C}^+(k_i, t_i) = \{(k_j, t_j) \mid (k_i, t_i) \prec (k_j, t_j)\}.$$

The causal cone is determined jointly by the dependency structure of G and the communication metric d_G of H. Two events are *causally disjoint* when neither lies in the other's causal cone; causally disjoint events may execute simultaneously without coordination overhead, because no information can flow between them within the relevant time interval.

This formulation makes the hardware graph into something closer to a finite asynchronous spacetime manifold. Physical distance in d_G bounds the speed of causal propagation; the chip's topology determines which computations can run concurrently and which must be sequenced. A placement φ that maximizes the causal disjointness of computationally independent kernels minimizes unnecessary serialization and therefore minimizes the thermodynamic cost of coordination.

Deadlock can now be characterized geometrically: a deadlock occurs when a set of events $\{(k_i, t_i)\}$ form a cycle in the causal precedence relation, so that each event is waiting for a predecessor that is itself waiting. This is equivalent to the non-existence of a global section in the synchronization sheaf F—consistent with the Čech cohomology obstruction interpretation developed in Section 6. The causal and sheaf-theoretic analyses thus converge on the same necessary condition for deadlock-free computation.

Temporal admissibility adds a further constraint to the compilation problem. A valid compilation must not only satisfy spatial admissibility (kernel k at node n with $R(k) \subseteq I(n)$) but also temporal admissibility: the causal precedence constraints must be satisfiable within the timing windows specified in C. Bounded-latency proofs for safety-critical systems can then be stated as geometric assertions about the causal cone structure of the compiled program: a kernel k_j meets its deadline if and only if all predecessors in $\mathcal{C}^-(k_j, t_j^{\max})$ can reach $\varphi(k_j)$ by time $t_j^{\max}$.

A Worked Miniature Example

To anchor the preceding abstractions, consider a concrete miniature instantiation of the framework on a nine-node island mesh.

Hardware. The hardware graph H_9 consists of nine F18A-style nodes arranged in a 3×3 grid. Interior node n_5 (center) connects to all four cardinal neighbors. Corner nodes (n_1, n_3, n_7, n_9) each have two neighbors. Edge nodes (n_2, n_4, n_6, n_8) each have three neighbors. Node n_1 serves as the external input port; node n_9 serves as the external output port. Each edge has uniform energy cost $w = 1$ and latency $\tau = 1$. Each node has 64 words of local memory and supports stack-machine instruction execution.

Program. The task is a sensor-filter-reduce-output pipeline: a scalar sensor reading enters at n_1, passes through a digital filter (a five-tap FIR computation requiring five multiply-accumulate operations), is reduced by a threshold comparison, and the binary result is emitted at n_9.

The program graph G has four kernels: k_{sense} (sample and forward the input), k_{filter} (accumulate five taps), k_{reduce} (threshold comparison), and k_{out} (format and emit output). The dependency chain is linear: $k_{\text{sense}} \to k_{\text{filter}} \to k_{\text{reduce}} \to k_{\text{out}}$.

Valid placement. The locality-preserving placement assigns: $\varphi(k_{\text{sense}}) = n_1$, $\varphi(k_{\text{filter}}) = n_2$, $\varphi(k_{\text{reduce}}) = n_3$, $\varphi(k_{\text{out}}) = n_6$. Each communication hop traverses one edge; total communication cost is $3w = 3$. Nodes n_4, n_5, n_7, n_8, n_9 remain dormant throughout. The activity volume for a single pipeline invocation is $\mathcal{V} = 4$ (four kernel executions) plus 3 (three intermediate communication events) $= 7$ node-time units.

Invalid placement. An alternative placement assigns $\varphi(k_{\text{filter}}) = n_7$, routing the sensor output from n_1 to n_7 through n_4 and n_7 (two hops, passing through n_4) and the filtered output back from n_7 to n_3 via n_4 or n_8 (two more hops). The intermediate routing nodes n_4 must activate to relay messages even though they perform no computation. Total communication cost rises to $5w = 5$; the activity volume rises to approximately 11 node-time units due to relay activations. This placement also violates the thermal-separation constraint if k_{filter} and k_{sense} were assigned to the same voltage domain boundary.

Energy comparison. The valid placement achieves an activity volume of 7 units; the invalid placement achieves approximately 11 units—a 57% increase in energy expenditure for identical computational output. On a chip running at 10^9 pipeline invocations per second, this difference corresponds to a sustained power increase of several milliwatts, which is significant at the energy budget of a sensor swarm node. The activity

heatmap for the valid placement shows five dark (dormant) nodes and four briefly active nodes arranged along the top edge of the grid; the invalid placement shows six active nodes including two relay activations whose sole contribution is message forwarding.

Relation to Existing Frameworks

Silicon Cartography does not emerge from a vacuum. Several prior and parallel research traditions have addressed aspects of the problem it addresses, and positioning the framework relative to these traditions is necessary to identify what is genuinely new.

Dataflow programming languages—from Dennis's original data flow procedure language through Kahn process networks and Lucid—decompose computation into stream-processing networks in which nodes activate when input data is available. This is structurally similar to the kernel-decomposition model, and the blocking communication discipline of GA144-style chips is directly implementable as a Kahn network. The difference is that Silicon Cartography adds explicit physical embedding: the placement of dataflow nodes onto chip geometry with energy-cost optimization is not addressed by dataflow semantics, which abstract away the physical substrate entirely.

Communicating sequential processes and related process algebras (CCS, π-calculus) provide compositional models of concurrent communication that closely parallel the synchronization sheaf formulation. The handshake admissibility condition $r_i(t) \wedge w_j(t) = 1$ is a direct instance of CSP synchronization. The difference again is physicality: process algebras operate on abstract named channels; Silicon Cartography operates on metrized hardware edges with energy, latency, and congestion.

Systolic arrays formalize the mapping of regular array computations onto spatially regular processor meshes, achieving high throughput through careful data scheduling and local communication. The systolic principle—that data should flow rhythmically through a regular array without requiring global control—is closely related to the streaming computation model of Hilbert-curve and dragon-curve chips. The difference is that systolic arrays require globally synchronous clocking, while Silicon Cartography targets asynchronous chips and prioritizes energy minimality over throughput maximization.

Network-on-chip research addresses the design and optimization of the on-chip communication fabric for many-core processors, including topology selection, routing algorithms, and arbitration policies. The hardware graph formalism directly subsumes the network-on-chip problem as a special case: the chip's communication topology is modeled as a metrized graph with bandwidth, latency, and contention constraints. The difference is that network-on-chip research typically assumes a fixed program structure and optimizes the network for it, while Silicon Cartography co-optimizes program decomposition and network placement simultaneously.

High-level synthesis tools (Halide, TVM, XLA) optimize the mapping of array computations onto hardware targets including GPUs, FPGAs, and custom accelerators. These tools perform schedule and placement optimization within their respective computational models. Silicon Cartography generalizes beyond regular array computation to arbitrary program graphs, handles asynchronous communication explicitly, and treats energy as the primary optimization target rather than throughput or latency.

Place-and-route tools in VLSI design solve the problem of assigning logic gates to physical locations and routing wires between them—a problem that is structurally similar to the compilation mapping $\varphi : G \to H$. The key difference is level of abstraction: place-and-route operates at the level of logic gates and wires; Silicon Cartography operates at the level of computational kernels and communication protocols. The formalisms are compatible and a Silicon Cartography compiler could in principle generate a place-and-route specification for an FPGA or custom ASIC.

The genuinely novel synthesis offered by Silicon Cartography is the unification of these concerns under a single geometric-thermodynamic ontology: spatial compilation, thermodynamic cost modeling, admissibility geometry, fractal topology generation, synchronization sheaves, causal partial orders, and visualization as energetic cartography. No existing framework addresses all of these simultaneously, and the mutual reinforcement among them—particularly the way that fractal topology generation, spectral analysis, sheaf-theoretic deadlock detection, and energy landscape visualization all operate on the same underlying hardware graph H—is the framework's most distinctive structural contribution.

Limitations and Open Problems

The framework as presented assumes that the hardware graph H is fully known at compile time. For reconfigurable chips (FPGAs, some neuromorphic systems), H may itself be a design variable, opening a joint optimization problem over chip configuration and program placement. Extending Silicon Cartography to this setting requires a richer formalism in which the compilation functor maps into a space of (hardware configuration, program placement) pairs.

A second open problem is the handling of dynamic workloads: inputs that vary in size, shape, or arrival rate over time. The static compilation model produces a fixed placement; adapting it dynamically requires either pre-compiling a family of placements indexed by workload parameters, or developing online replanning algorithms that can migrate kernels across nodes at runtime.

A third open problem concerns the procedural topology language itself: given a computational workload and an energy budget, what chip topology should be generated? The inverse design problem—find $\mathcal{F}$ and recursion depth m such that the induced hardware graph H_m admits a low-cost compilation of a given program class $\mathcal{G}$—is structurally a program synthesis problem over a topology grammar, and is not addressed by any of

the existing tools reviewed above.

Finally, the causal geometry framework raises the question of reversible computation. Fredkin and Toffoli demonstrated that logically reversible computation can in principle approach the Landauer limit of zero dissipation per operation. Integrating reversible computation primitives into the Silicon Cartography kernel model—so that kernels can be annotated as reversible where the computation permits—would allow the framework to reason about the thermodynamic floor of a given computation, not merely its current energy cost.

Conclusion

Silicon Cartography proposes a discipline inversion for programming spatial chips: begin with the machine, not the program. By modeling the hardware as a metrized constraint graph, decomposing tasks into locally self-contained kernels with explicit communication interfaces, and finding energy-minimal placements through constrained optimization over the chip's admissibility field, it recovers the direct negotiation between algorithm and matter that abstraction-heavy software engineering has systematically obscured.

The framework operates simultaneously at three intellectual levels that mutually constrain one another. At the practical level it provides a four-layer compiler architecture with concrete algorithms for hardware cartography, capability modeling, program decomposition, and placement routing. At the mathematical level it furnishes a unified set of tools—metrized hardware graphs, graph Laplacians, synchronization sheaves, iterated function systems for fractal topology generation, causal partial orders, and spacetime activity integrals—that make each architectural decision formally precise. At the philosophical level it argues that the dominant tradition of sequential symbolic computation is not a neutral description of what computation is, but a historically contingent engineering choice whose thermodynamic costs become increasingly visible as computing moves toward the energy frontier.

The fractal topology families developed here—island meshes, tree-of-meshes, Hilbert-curve chips, Sierpinski compute fabrics, dragon-curve pipelines, Apollonian fabrics—demonstrate that the hardware graph formalism is not limited to the flat uniform grid of the GA144 but encompasses a wide design space of spatial substrates, each with characteristic spectral properties, locality structures, and programming disciplines. The procedural topology language provides a common descriptor schema for this design space, separating hardware geometry from compilation policy and enabling systematic exploration of chip-program co-design.

The visualization framework translates the abstract mathematical structure of spatial computation into directly perceivable form: activity heatmaps make dormancy visible, event wave propagation makes causal structure visible, energy landscape terrain makes thermodynamic cost visible, and causal braid diagrams make synchronization

structure visible. Together these representations constitute a cartographic discipline in the literal sense: they map the invisible energetic geography of a running computation onto a form that can be directly navigated.

The connection to thermodynamic computation theory—Landauer's principle, conservative logic, the activity volume integral as a measure of irreversibility expenditure—grounds the energy-minimality criterion in physics rather than engineering preference. Moore's moral language about waste is, in the end, thermodynamically precise: every unnecessary transistor transition is a small irreversibility, a minor but real entropic cost paid by the physical universe for the privilege of discarding information. A programming discipline that takes this seriously is not performing premature optimization. It is computing honestly.

Acknowledgments

The author thanks the extended discussions around GreenArrays documentation, Forth philosophy, and constraint-first computing that informed this framework.

References

[1] C. Moore, "GreenArrays Architecture and colorForth Programming," unpublished lecture notes and documentation, GreenArrays Inc., 2013.

[2] C. Moore, "Programming a Problem-Oriented Language," unpublished manuscript, 1974.

[3] GreenArrays Inc., "GA144 Product Brief," 2011. `http://www.greenarraychips.com`

[4] K. M. Fant, *Computer Science Reconsidered: The Invocation Model of Process Expression*, Wiley-IEEE Computer Society Press, 2005.

[5] K. M. Fant and S. A. Brandt, "Null Convention Logic: A Complete and Consistent Logic for Asynchronous Digital Circuit Synthesis," in *Proceedings of the International Conference on Application Specific Array Processors*, 1985.

[6] A. J. Martin, "The Limitations to Delay-Insensitivity in Asynchronous Circuits," in *Advanced Research in VLSI*, MIT Press, 1990, pp. 263–278.

[7] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design: A Systems Perspective*, Kluwer Academic Publishers, 2001.

[8] J. B. Dennis, "First Version of a Data Flow Procedure Language," in *Programming Symposium*, Lecture Notes in Computer Science, vol. 19, Springer, 1974, pp. 362–376.

[9] W. W. Wadge and E. A. Ashcroft, *Lucid, the Dataflow Programming Language*, Academic Press, 1985.

[10] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proceedings of IFIP Congress 74*, North-Holland, 1974, pp. 471–475.

[11] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, 21(8):666–677, 1978.

[12] R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[13] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, 21(8):613–641, 1978.

[14] R. Landauer, "Irreversibility and Heat Generation in the Computing Process," *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[15] C. H. Bennett, "The Thermodynamics of Computation—A Review," *International Journal of Theoretical Physics*, 21(12):905–940, 1982.

[16] E. Fredkin and T. Toffoli, "Conservative Logic," *International Journal of Theoretical Physics*, 21(3–4):219–253, 1982.

[17] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed., Morgan Kaufmann, 2019.

[19] W. J. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks," in *Proceedings of the 38th Design Automation Conference*, ACM/IEEE, 2001, pp. 684–689.

[20] L. Benini and G. De Micheli, "Networks on Chips: A New SoC Paradigm," *Computer*, 35(1):70–78, 2002.

[21] M. Fiedler, "Algebraic Connectivity of Graphs," *Czechoslovak Mathematical Journal*, 23(2):298–305, 1973.

[22] F. R. K. Chung, *Spectral Graph Theory*, American Mathematical Society, 1997.

[23] M. F. Barnsley, *Fractals Everywhere*, Academic Press, 1988.

[24] H. Sagan, *Space-Filling Curves*, Springer, 1994.

[25] D. Hilbert, "Über die stetige Abbildung einer Linie auf ein Flächenstück," *Mathematische Annalen*, 38(3):459–460, 1891.

[26] D. J. Watts and S. H. Strogatz, "Collective Dynamics of Small-World Networks," *Nature*, 393:440–442, 1998.

[27] D. I. Spivak, *Category Theory for the Sciences*, MIT Press, 2014.

[28] S. Mac Lane, *Categories for the Working Mathematician*, 2nd ed., Springer, 1998.

[29] R. Ghrist, *Elementary Applied Topology*, Createspace, 2014.

[30] J. Curry, R. Ghrist, and M. Robinson, "Euler Calculus with Applications to Signals and Sensing," *Proceedings of Symposia in Applied Mathematics*, 70:75–145, 2012.

[31] M. Robinson, *Topological Signal Processing*, Springer, 2014.

[32] P. A. Merolla et al., "A Million Spiking-Neuron Integrated Circuit with a Scalable Communication Network and Interface," *Science*, 345(6197):668–673, 2014.

[33] M. Davies et al., "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning," *IEEE Micro*, 38(1):82–99, 2018.

[34] S. Furber et al., "The SpiNNaker Project," *Proceedings of the IEEE*, 102(5):652–665, 2014.

[35] P. M. Kogge et al., "Processing in Memory: Chips to Petaflops," Workshop on Mixing Logic and DRAM, 1994.

[36] C. E. Leiserson and J. B. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, 6(1):5–35, 1991.

[37] A. Juarrero, *Dynamics in Action: Intentional Behavior as a Complex System*, MIT Press, 1999.

[38] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.