Cellato: a DSL for Cellular Automata based on C++ Template Meta-programming

Matyáš Brabec
Jiří Klepl
Martin Kruliš
brabec@d3s.mff.cuni.cz
klepl@d3s.mff.cuni.cz
krulis@d3s.mff.cuni.cz
Department of Distributed and Dependable Systems, Charles University
Prague, Czech Republic

Abstract

Cellato is a tool with embedded DSL in C++ that leverages template meta-programming to define and execute cellular automata (CA) via concise type-level expressions that are specialized into efficient kernels at compile time. Its modular architecture decouples the Algorithm (rules), Evaluator (per-cell update), Layout (memory representation), and Traverser (grid iterator), allowing users to mix and match components without altering rule definitions. We demonstrate Cellato on Conway's Game of Life, Forest Fire, Wireworld, and the Greenberg-Hastings excitable medium, which cover binary as well as multi-state models and Moore to von Neumann neighborhoods. We experimented with three memory layouts (standard arrays, bit-packed arrays, and bit-planes) for transparent bit-level encodings and data-parallel optimizations. Targeting both CPU and GPU back-ends, Cellato delivers performance on par with hand-tuned code, while its zero-overhead abstractions, flexible scheduling, and portable optimizations provide a robust foundation for high-performance CA computations.

CCS Concepts

• Software and its engineering → Domain specific languages; **Frameworks**; **• Computing methodologies → Simulation languages**; **• Theory of computation → Massively parallel algorithms**.

Keywords

Domain-Specific Language, Cellular Automata, Bit-Packing, C++ Template Metaprogramming, CUDA

ACM Reference Format:
Matyáš Brabec, Jiří Klepl, and Martin Kruliš. 2018. Cellato: a DSL for Cellular Automata based on C++ Template Meta-programming. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Cellular Automata (CA) are discrete computational models consisting of grids of cells, each in a finite number of states, evolving synchronously according to simple local rules. Despite their simplicity, CA have numerous applications across scientific and engineering disciplines, including physics, biology, computer science, and environmental modeling [15, 19, 21]. They are especially valued for their capability to simulate complex dynamical systems and emergent phenomena via local interactions. Although CA are used in many domains, their underlying implementations often share many commonalities; thus, they can benefit from common abstractions, frameworks, and optimizations.

Traditional approaches frequently tightly couple algorithm logic, evaluation strategies, and data layouts [5, 8]. This coupling complicates or even prevents code reuse and duplicates programmers' work, particularly in the context of performance optimizations and elaborate parallelization. Furthermore, this tight coupling restricts researchers and developers aiming to explore new rules or optimize performance without re-engineering substantial parts of the underlying codebase.

To overcome these limitations, we propose a **Cell**ular **A**bstraction **To**ol (Cellato) with an embedded domain-specific language (DSL) for Cellular Automata implemented in C++ using advanced template meta-programming techniques. The Cellato DSL decouples the specification of CA rules from evaluation strategies and data storage. Users can abstractly express CA algorithms, facilitating independent selection and optimization of data layouts and traversal methods. Furthermore, Cellato provides prototype implementations for simulating specified CA using various data layouts or executors that demonstrate portability across CPU and GPU platforms.

We have validated Cellato using four well-known CA models: Conway's Game of Life [6], the Forest Fire automaton [5], Wireworld [7], and the Greenberg-Hastings automaton [9]. These diverse examples demonstrate the flexibility and expressiveness of Cellato, enabling users to seamlessly define various CA rules without modifying evaluation or traversal logic. Additionally, we use these four models as running examples to illustrate the DSL syntax and capabilities.

The main contributions of this work include:

- A flexible template-based DSL enabling abstract and concise definitions of CA algorithms.
- Seamless support for various data layouts (standard arrays, bit-packed arrays, bit-planes).
- Efficient and modular evaluation strategies utilizing visitor patterns and traversers suitable for both CPU and GPU execution.
- A proof-of-concept implementation that can be used for further experimentation and CA optimizations.
- Comprehensive evaluation comparing our proposed abstraction with existing tools for implementing CA or stencil algorithms in general.

Our prototype implementation of the Cellato framework as well as complete implementations of the model CA and all experimental data are available on GitHub[1]. The source codes are available under the MIT license, allowing free utilization in future research.

The remainder of this paper is structured as follows. Section 2 provides background information on related topics. The automata selected as running examples are detailed in Section 3. Section 4 describes the Cellato DSL using the four running examples, and Section 5 presents the implementation details of the layout encoders and CA evaluators. The evaluation and comparison with existing DSL and CA tools are summarized in Section 6. Section 7 summarizes related work and Section 8 concludes our paper.

2 Background

A cellular automaton (CA) can be viewed as a special case of a *stencil* computation. At every discrete time step, an identical local operator (the *stencil*) reads the states of a fixed neighborhood $\mathcal{N}$ surrounding each lattice point and writes a new state for that point. Formally, for a d-dimensional grid with the finite state set $S = \{s_0, \ldots, s_{k-1}\}$, the update rule is a function $f : S^{|\mathcal{N}|} \to S$ applied simultaneously to all cells:

$$c^{t+1}(\mathbf{x}) = f\big(\{\, c^t(\mathbf{x}+\mathbf{n}) \mid \mathbf{n} \in \mathcal{N} \,\}\big).$$

Because the computation is (i) *embarrassingly parallel* (assuming separate buffers for input and output grids) and (ii) involves only a handful of arithmetic or logical operations per input value, real-world CA codes are often **memory-bound**. Meaning that if GPU cores access global memory inefficiently (e.g., through unaligned accesses or with insufficient reuse via either caches or shared memory) memory throughput can become the primary bottleneck of the computation. However, when memory access is properly optimized, the number of arithmetic and logical operations begins to impact performance as well – especially in case of more complex cellular automata.

2.1 Standard optimization techniques

The classical optimization for stencil codes is *temporal blocking*. In this optimization, small tiles (that can fit caches) are processed so multiple time steps are aggregated together [20]. Production frameworks such as Halide, ExaStencils, and GridTools offer rich schedule languages to explore that space automatically [1, 11, 14].

For cellular automata, however, an additional (and often more impactful) optimization exists: *state encoding*. Most CA uses only a few distinct states – for instance, Game of Life has two, Forest-Fire and Wireworld have four, and our version of Greenberg-Hastings uses eight. Packing each state into its minimum bit width (1–3 bits in the presented cases) can reduce memory traffic by up to an order of magnitude [5, 8]. Even more aggressive is a *bit-plane* layout that stores individual bits of the state in separate dense bit-arrays. This maximizes SIMD utilization at the expense of more complex indexing.

Existing stencil DSLs largely ignore this dimension, and their generated code assumes a scalar or vector element size chosen once per grid. Changing the memory representation of a cell, therefore, requires manual modifications throughout the code base. We endeavor to eliminate this coupling and make the CA specification independent of the selected encoding.

2.2 DSLs and frameworks for stencils and CA

Domain-specific languages (DSLs) for stencil and CA specifications fall into three broad categories:

External languages. Languages such as ExaStencils [11] are written in their own syntax, then compiled to C/CUDA/HIP or directly to machine code. They expose high-level scheduling primitives but offer no mechanism to specialize the *in-cell* data layout.

Embedded run-time languages. Languages like Halide [14] or Tiramisu [2] embed the DSL in C++, allowing users to write rules as C++ functions. The DSL compiler then generates optimized code for the target device. They are convenient and offer a rich set of scheduling primitives, but they sacrifice interoperability with the host language. Similarly to external DSLs, they offer no mechanism to specialize the in-cell data layout.

Embedded compile-time languages. A third family leverages the type system of the host language itself to represent computations. Blitz++ [17], Boost.YAP [10] and various finite-difference libraries are notable examples in C++. Our proposed tool, Cellato, belongs in this category since the entire CA rule is a single *type-level expression.* No external parser or compiler plugin is needed, and the C++ compiler becomes also the optimizer.

Annotation-based frameworks. Slightly aside from DSLs stands frameworks that use *annotation* systems to mark loops or functions with special directives. Frameworks like AN5D [12] or Loopy [13] embed scheduling directives as an otherwise ordinary code. A separate code-generator back-end rewrites the marked loops into optimized kernels. Although convenient, annotation systems still treat every cell as a fixed-width scalar.

2.3 C++ templates

Modern C++ offers two complementary compile-time mechanisms:

- **Templates** are primarily designed for generic programming – allowing functions and classes to operate on types and values in a flexible, reusable way. However, their capabilities go far beyond that. Since templates can be nested, specialized, and parameterized by types, values, or even other templates, they can, in principle, be used to perform arbitrary computation during compilation.

[1]https://github.com/matyas-brabec/2025-icooolps-cellato

- `constexpr` **functions** evaluate at compile time whenever their inputs are compile-time constants, producing ordinary run-time values otherwise.

Both mechanisms adhere to the *zero-overhead principle*: once the compiler inlines and optimizes away the meta-programming scaffolding, the generated machine code is indistinguishable from hand-written special-purpose C. In the context of cellular automata, this means we can:

(1) encode the transition rule, neighborhood shape, and even compile-time numeric constants *as types*;
(2) pattern-match those types to autogenerate evaluator kernels for CPU, blocked CPU, CUDA, and future back-ends;
(3) and select alternative memory layouts by recompiling, with no edits to the code that specifies CA rules.

3 Implemented Cellular Automata

We have selected four well-known cellular automata as running examples and CA representatives for evaluation. These models vary in their neighborhood shapes, state counts, and rule structures, allowing us to demonstrate the flexibility and expressiveness of our Cellato DSL. All presented CA models use 2D cell grids, which is the minimal dimensionality for demonstrating the DSL capabilities; however, all the techniques discussed in this section can be easily extended to higher dimensions. The Forest-Fire model uses the von Neumann neighborhood (4 neighbors), while the other three use the Moore neighborhood (8 neighbors).

3.1 Conway's Game of Life

Perhaps the best-known CA is Conway's Game of Life [6] (GoL). It operates on a 2D grid with a **binary state set**, where each cell is *dead* or *alive*. The CA employs the following rules that determine the next state of each cell based on its **Moore neighborhood** (the eight surrounding cells):

- A cell becomes *alive* if exactly three of its eight neighbors are alive (reproduction).
- A cell dies if there are fewer than two (not enough social interactions) or more than three alive neighbors (starvation caused by overpopulation).
- A cell stays *alive* if it has two or three alive neighbors (thriving conditions).

The Game of Life is a classic example of a CA that exhibits complex emergent behavior from simple rules. Despite its simplicity (only two states handled by three simple rules), it can simulate quite elaborate patterns and animations. It is also popular since these patterns can be easily visualized.

From the implementation perspective, the most intriguing fact is that the state can be represented by a single bit. This allows us to use a bit-packed array layout, where each cell is stored as a single bit, significantly reducing memory footprint. More specifically, we can pack various tiles of cells into a single word (e.g., we can pack 1×64 or 8×8 cells into a single 64-bit word). Additionally, the rules can be expressed as binary operations, enabling efficient use of bitwise instructions [8] in a SIMD-like manner. Alternatively, specialized instructions such as `popcount`, which counts the number of set bits in a word, can be used to quickly compute the number of alive neighbors.

3.2 Forest-Fire model

Our second example is the Forest-Fire model [5], which simulates the spread of fire in a forest. The version we have adopted for our paper has **four states** that indicate the main contents of the area represented by the cell: *tree*, *fire*, *ash*, and *empty*. The state changes are governed by the following rules:

- A *tree* with at least one burning neighbor ignites and turns to *fire*.
- A burning tree (*fire*) turns into *ash*.
- *Ash* decays to an *empty* cell if it has no *fire* cell neighbors.

This model uses the **von Neumann neighborhood** (four cells adjacent by full side), allowing us to demonstrate the independence of our proposed language on the neighborhood shape.

The four states can be represented by 2 bits, which is still small enough to benefit from bit-packing. On the other hand, packing multiple bits opens other possibilities, namely the bit-plane encoding. The bit-planes use a similar idea as the structure-of-arrays (SoA) layout, where each field of a structure is stored in a separate array (rather than having the structures compact as in the array-of-structures representation). In this case, we store individual bits in separate bit-packed words. This approach is beneficial if the state transformation can be implemented solely by bitwise logical operations, thus exploiting SIMD processing at the bit level.

Stochastic Fire. As an alternative, one can consider a probabilistic Forest-Fire variant in which two additional random events occur at each time step:

- *Spontaneous ignition*: each *tree* cell ignites with a small probability f, even if none of its neighbors is burning.
- *Tree regrowth*: each *empty* cell sprouts a new *tree* with probability g.

In this nondeterministic model, the core neighbor-driven ignition and state-decay rules remain the same.

3.3 Wireworld

Wireworld is a cellular automaton originally proposed by Brian Silverman almost four decades ago [7]. It simulates electron movements in electrical circuits, including transistors and logical gates, and it can be used for very complex simulations since it is Turing-complete. Like the previous example, it uses **four states**: *empty*, *electron-head*, *electron-tail*, and *conductor*. The state changes are as follows:

- An *empty* cell remains *empty* (all the time).
- An *electron-head* becomes an *electron-tail* (basic movement).
- An *electron-tail* becomes a *conductor*.
- A *conductor* becomes an *electron-head* if one or two neighboring cells contain *electron-heads*; otherwise, it remains a *conductor*.

As with the Forest Fire model, this automaton encodes its states using two bits per cell. However, it relies on the **Moore neighborhood**, considering all eight surrounding cells – similarly to the Game of Life.

3.4 Excitable medium model

An excitable medium is a nonlinear dynamical system that can exhibit wave-like behavior. It is often used to model physical, biological, or chemical processes. Discrete approximations of this system can use cellular automata for modeling. The Greenberg-Hastings (GH) model [9] is one of the first examples of a cellular automaton that models an excitable medium using three states: *quiescent* (resting), *excited*, and *refractory* (cooling down). The transition rules for the states are as follows:

- A *quiescent* cell becomes *excited* if at least one of its neighbors is *excited*.
- An *excited* cell transitions to the *refractory* state.
- A *refractory* cell returns to the *quiescent* state.

The Greenberg-Hastings model works with any neighborhood; we have selected the Moore neighborhood, as it is more challenging for some of the techniques presented than the von Neumann neighborhood and thus serves as a more interesting test case. Furthermore, the GH model can be easily extended to use more refractory states (so it better simulates the continuous excitable medium systems), which makes it a good candidate for demonstrating the flexibility of Cellato. Having k refractory states $r_1, \ldots, r_k$, the state transitions need to be modified slightly:

- An *excited* cell transitions to the *refractory* state r_1.
- A *refractory* state r_i $(i < k)$ transitions to state r_{i+1}.
- A *refractory* state r_k transitions back to the *quiescent* state.

For our demonstration and verification, we have selected $k = 6$, so the total number of **states is** 8; thus, each state can be encoded using 3 bits. However, we can choose any other number for k even if the total number of states is not a power of two (in such case, the memory representation will be inevitably suboptimal).

4 DSL Design and Syntax

Cellato DSL can be understood as a *type-level expression language*, which encodes the transition rule of a cellular automaton as a C++ type for a single cell. This type-based rule is later applied uniformly across the entire grid. The type itself is constructed from predefined nodes, each representing fundamental operations such as state constants, neighbor access, and logical evaluations. Let us present the most common nodes.

```
template <auto value>
struct constant { /* ... */ };

template <auto value>
struct state_constant { /* ... */ };
```

Listing 1: Overview of nodes

Listing 1 presents `constant` and `state_constant` nodes, which represent compile-time constants. The `constant` node is used for constant values that play some role in the CA rules (like 2 or 3, which indicate ideal numbers of neighbors in the GoL automaton). The `state_constant` refers to a value representing one of the CA states.

```
template <int X_offset, int Y_offset>
struct neighbor_at { /* ... */ };

template <typename S, typename N>
struct count_neighbors { /* ... */ };
```

Listing 2: Overview of nodes

Listing 2 demonstrates nodes that are used to access neighboring cells. The `neighbor_at` node allows access to a neighbor cell value positioned relatively at `<X_offset,Y_offset>` from the current cell. The `count_neighbors` node counts how many neighbors of a certain state S are present in the neighborhood defined by shape N. Unlike other nodes, this one is more high-level – resembling a function rather than a simple atomic operation – and could, in principle, be expressed using a combination of lower-level nodes. However, since counting neighbors is a common operation in many cellular automata, we have introduced it as a dedicated node, allowing for potential optimizations during evaluation.

```
template <typename C, typename T, typename E>
struct if_then_else { /* ... */ };

template <typename L, typename R>
struct equals { /* ... */  }; // + and_, or_ ...
```

Listing 3: Overview of nodes

Listing 3 presents the most typical representatives of conditional nodes. The `if_then_else` node represents a ternary operator, which evaluates the condition C and returns either the type T if the condition is true or the type E if it is false. The `equals` node checks whether two types are equal. Similarly to `equals`, we can define logical operators like `and_` or `or_`.

4.1 Supporting types

C++ types inherently use prefix notation, which is not intuitive for all users. We address this issue by introducing additional supporting types, allowing users to write types in a more familiar and readable form. Underlying this syntactic enhancement is a simple renaming of existing nodes through type aliases. Users are also free to introduce their own syntactic constructs, provided they eventually resolve to our basic nodes. Crucially, these aliases incur no additional overhead as they are interpreted by the compiler directly as type-based expressions. An example of a supporting type is a parentheses-like notation shown below:

```
template <
  typename L,
  template <typename, typename> class Op,
  typename R>
using p = Op<L, R>;

// later we can use parentheses p it like this:
using is_alive = p<current_state, equals, alive>;
```

Listing 4: Supporting type for implementing parentheses

This syntax significantly improves readability by allowing operations to be written more naturally.

Another critical construct in cellular automata rules is conditional branching. The ternary operator `if_then_else` becomes cumbersome with deeply nested expressions. To simplify this, we provide a supporting type resembling the standard **if-else** statements found in imperative languages:

```
using some_rule =
  if_</* bool expression */>::then_<
      /* value if the expression evaluates to true */
  >::
  elif_</* another bool expression */>::then_<
      /* value if the second expression evaluates to true */
      /* multiple elifs allowed */
  >::
```

```
else_<
    /* value if all expressions evaluate to false */
>;
```

Listing 5: Supporting types for if-then-else

Similarly to functional languages, all branches must be explicitly defined. Internally, this syntax is an alias for the original `if_then_else` node, enforcing completeness.

Note that although these aliases introduce no runtime overhead, they can increase the complexity of template expansion, potentially leading to longer compilation times. However, cellular automaton rules are typically short, so the additional cost is unlikely to be significant – especially when compared to the compile-time visitor discussed in Section 5.2.1. In any case, the trade-off is well justified, as the improvement in code readability and usability outweighs the modest impact on compilation time. Without such syntactic sugar, the prefix notation would render the language practically unusable for most programmers.

4.2 Implementation of cellular automata rules

Having introduced the basic nodes and supporting syntactic structures, we demonstrate their usage through practical examples. While the following examples follow our recommended conventions, users may freely adapt or extend these patterns as long as the resulting rule is composed solely of the predefined nodes.

4.2.1 Conway's Game of Life. First, we define necessary constants, clearly distinguishing *state constants* (e.g., `alive`, `dead`) from *numeric constants* (e.g., `c_2`, `c_3`). State constants represent discrete cell states, whereas numeric constants are utilized for comparisons. This distinction prevents accidental misuse:

```
enum class cell_state { dead, alive };

using alive = state_constant<cell_state::alive>;
using dead  = state_constant<cell_state::dead>;

using c_2 = constant<2>;
using c_3 = constant<3>;
```

Listing 6: Defining GoL states and constants

Type aliases enhance readability, aligning with patterns found in other functional languages. Therefore, we define predicates that determine the current cell's state. The `current_state` node refers explicitly to the cell at position (0,0), the cell being evaluated:

```
using is_alive = p<current_state, equals, alive>;
using is_dead  = p<current_state, equals, dead>;
```

Listing 7: Cell state predicates

Counting neighbors is another essential operation in GoL rules. Although users could define neighbor-counting explicitly using the `neighbor_at` node, our provided `count_neighbors` node simplifies the notation and potentially optimizes compilation:

```
using alive_cnt = count_neighbors<alive, moore_8_neighbors>;

using has2      = p<alive_cnt, equals, c_2>;
using has3      = p<alive_cnt, equals, c_3>;
using has3_or_2 = p<has2, or_, has3>;
```

Listing 8: Counting neighbors

The `alive_cnt` uses the previously unspecified `moore_8_neighbors` type. This is a tag type (an empty `struct`) that is later used in a CA evaluator to determine how to iterate over the neighboring cells. The evaluators use template specialization based on these tags to select the right implementation (at compile time). Analogically, we declare the `von_neumann_4_neighbors` tag type for the von Neumann neighborhood.

Finally, we define Conway's Game of Life as follows. As a reminder, an alive cell remains alive if it has exactly two or three living neighbors, and a dead cell turns alive if it has exactly three living neighbors.

```
using gol_rule =
  if_< is_alive >::then_<
      if_< has3_or_2 >::then_< alive >::else_< dead >
  >::else_<
      if_< has3 >::then_< alive >::else_< dead >
  >
```

Listing 9: Final type-level rule

4.2.2 Forest Fire. The Forest Fire CA introduces a more diverse state set: *empty*, *tree*, *fire*, and *ash*. Furthermore, it is based on examining the von Neumann neighborhood instead of the Moore neighborhood. This contrast to the previously described GoL CA demonstrates the flexibility of our neighbor-counting abstraction and the ease of defining rules with multiple conditions.

```
using fire_cnt = count_neighbors<fire, von_neumann_4_neighbors>;
using has_fire_neighbors = p<fire_cnt, greater_than, c_0>;
```

Listing 10: Counting fire neighbors

Listing 10 introduces the `von_neumann_4_neighbors` tag type for the neighborhood specification and the `greater_than` operator. The `fire` and `c_0` are state and numeric constants, respectively.

The key predicate checks for the presence of any neighboring fire cells (see Listing 10). The final rule expresses the different transitions, including re-ignition from neighboring flames, as shown in Listing 11:

```
using fire_algorithm =
    if_< cell_is_fire >::then_< ash >::
    elif_< cell_is_ash >::then_<
        if_< has_fire_neighbors >::then_< ash >::else_< empty >
    >::
    elif_< cell_is_tree >::then_<
        if_< has_fire_neighbors >::then_< fire >::else_< tree >
    >::
    else_< empty >; // If cell is empty, it remains empty
```

Listing 11: Forest Fire rule

We selected the deterministic Forest Fire variant since stochastic transitions are not yet supported. However, Cellato could be extended to handle nondeterministic rules by introducing a new AST node — e.g., `random<Distribution>` — that produces values from a user-defined distribution.

4.2.3 Wireworld. Wireworld is a four-state automaton used to model digital circuits. Each cell can be *empty*, an *electron head*, an *electron tail*, or a *conductor*. The update rule is based on a simple sequence of transitions, but it also includes a numeric condition: a conductor turns into an electron head only if it has exactly one or two electron head neighbors. The rule itself is then a straightforward sequence of state transitions.

```
using wire_algorithm =
    if_< cell_is_electron_head >::then_<
        electron_tail
    >::elif_< cell_is_electron_tail >::then_<
        conductor
    >::elif_< cell_is_conductor >::then_<
        if_< has_one_or_two_electron_head_neighbors >::then_<
            electron_head
        >::else_<
            conductor
        >
    >::else_< empty >; // If cell is empty, it remains empty
```

Listing 12: Wireworld rules

The `has_one_or_two_electron_head_neighbors` predicate has a similar definition to the `has3_or_2` predicate defined in Listing 8.

4.2.4 Greenberg-Hastings Model. The Greenberg-Hastings Model illustrates how the DSL can express rules with larger and more structured state machines. It cycles through a series of refractory states before returning to quiescence. We use eight distinct states to demonstrate the DSL's ability to support automata that require multiple bits for encoding:

```
using gh_rule =
    if_< is_quiescent >::then_<
        if_< has_excited_neighbors >::then_<
          excited
        >::else_<
          quiescent
        >
    >::
    elif_< cell_is_excited >::then_< refractory_1 >::
    elif_< is_refractory_1 >::then_< refractory_2 >::
    elif_< is_refractory_2 >::then_< refractory_3 >::
    elif_< is_refractory_3 >::then_< refractory_4 >::
    elif_< is_refractory_4 >::then_< refractory_5 >::
    elif_< is_refractory_5 >::then_< refractory_6 >::
    else_< quiescent >;
```

Listing 13: Greenberg-Hastings rule

5 Cellato Framework

Once the transition *rule* (i.e., the static representation of the CA rules) is defined as a type-level expression, the runtime of our framework is required to evaluate the rule for each cell (thus computing its new state) and store the new state according to the chosen memory layout. The traversal of the grid (evaluation of the rule) can be performed on various target (parallel) platforms, such as multi-core CPU and many-core GPU, or distributed on a cluster by MPI. We organize our Cellato framework into four modular components that cover these responsibilities:

(1) **Algorithm**: the type-level rule itself.
(2) **Evaluator**: a policy class that applies the Algorithm to a single cell.
(3) **Layout**: a grid data structure defining how cells are stored and accessed in memory.
(4) **Traverser**: the driver that walks over every cell in a specified order and execution context.

The layout and the evaluator are tightly coupled, as detailed below.

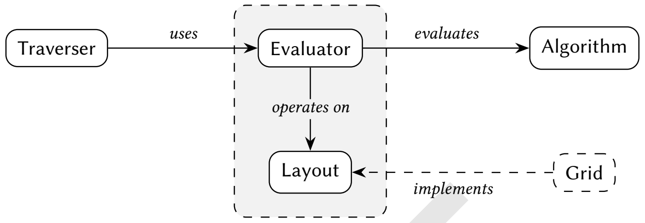

Figure 1: The execution architecture

5.1 Architecture

Figure 1 summarizes the architecture of our framework and the dependencies between its components. The grey box highlights the *Evaluator-Layout* pair, while the traverser and the Algorithm remain independent. We describe the components in more detail in the following.

5.1.1 Evaluators. An *evaluator* is a class template parameterized by the Algorithm (the type representing the encoded CA rules), exposing one static method:

```
new_cell = evaluator<Algorithm>::evaluate(/* cell id, grid */)
```

The `evaluate` method accepts a cell identifier and a grid that provides neighbor access. It then returns the `new_state`, which is then stored back into the grid.

5.1.2 Layouts. A layout determines both the ordering and internal memory representation of the cells (e.g., bit-packing). To demonstrate the complete independence of the *Algorithm* from the layout, we define three distinct layouts that represent three different approaches.

- **Standard** — a baseline layout that stores cells in their original representation (as enums) in the row-major order.
- **Bit Array** — many cellular automata require far fewer bits per cell than provided by `int` or `char` (e.g., Game of Life requires one bit, Greenberg-Hastings requires three bits). This layout eliminates wasted bits by packing multiple cells into a single word (either 32 or 64 bits), treating memory as an array of bits.
- **Bit Planes** — the most exotic layout we tested. If an automaton requires `n` bits per cell, the grid is represented as `n` separate arrays, each storing one bit from each cell. This layout is essentially a structure of arrays (SoA) rather than an array of structures (AoS), which describes the previous two cases.

In our implementation, we tightly couple the memory layout and the evaluator. While this approach necessitates implementing a distinct evaluator for each layout, it enables us to fully leverage layout-specific optimizations within the evaluators.

5.1.3 Traversers. A traverser is entirely independent of the Algorithm, evaluator, and layout; it only depends on the `grid` interface and the evaluator's interface (static `evaluate` method). Its responsibility is to iterate over the cells and apply the evaluation. The traverser can iterate the grid in any order or even in parallel.

```
template <typename evaluator_t, typename grid_t>
grid_t get_updated_grid(grid_t grid, int steps) {
    // ...
    for (int i = 0; i < steps; i++) {
        for (var cell_id : grid.cell_references()) {
            next_grid[cell_id] =
                evaluator::evaluate(/* cell_id, grid */)
        }
        // ...
    }
    // ...
}
```

Listing 14: Naïve sequential implementation of a traverser

Although we have implemented several traversers ourselves (particularly a parallel CPU traverser and a GPU-accelerated CUDA traverser), this component is heavily decoupled from the rest of the DSL, allowing users to define their own traversal strategy or to integrate other frameworks (such as GridTools) that address efficient grid traversal.

5.2 Implementation details

In the previous sections, we defined the four modular components of our architecture. Implementing the grid, the underlying layouts, and the traversers is relatively straightforward, but handling a type-level expression both at compile time and at runtime requires a more careful design.

5.2.1 Compile-Time visitor pattern. Since the CA rules are encoded as one C++ type (an AST), we face the classic choice of how to traverse and evaluate that tree. Embedding `evaluate` methods directly in each node would couple the AST to a single evaluation strategy.

Instead, we adopt a compile-time *visitor* pattern. The various `evaluator<...>` specializations play the role of visitors, and partial template specializations stand in for the usual `accept` methods, which are dispatched purely at compile time. This implementation is not a traditional double-dispatch visitor known from the runtime OOP but rather a zero-overhead, purely static dispatch where, for each AST node type `ast_node`, a full specialization of the *evaluator* is provided:

```
template <typename ast_node>
struct evaluator<ast_node> {
    static auto evaluate(/* cell_id, grid */) {
        // Specific logic for the node here
    }
};
```

Listing 15: AST node template specialization of evaluator

The compiler picks the correct `evaluate` at compile time based on the `ast_node` template parameter. For example, evaluating an `if_then_else` node translates down to:

```
template<typename cond, typename then, typename else>
struct evaluator<if_then_else<cond, then, else>> {

  static auto evaluate(/* cell_id, grid */) {
    if ( evaluator<cond>::evaluate(/* cell_id, grid */) ) {
      return evaluator<then>::evaluate(/* cell_id, grid */);
    } else {
      return evaluator<else>::evaluate(/* cell_id, grid */);
    }
  }
};
```

Listing 16: Compile-time visitor for `if_then_else`

This specialization needs to be defined for every common node type introduced at the beginning of Section 4. The supporting structures do not require specializations as they are processed directly by the compiler.

However, C++ partial template specializations unlock even finer-grained customizations. For example, the same `count_neighbors<state , neighborhood_tag>` node can dispatch to different evaluators based on the neighborhood tag:

```
template<typename cell_state>
struct evaluator<
    count_neighbors<cell_state, moore_8_neighbors> {
  static auto evaluate(/* ... */) { /* logic for 8 neighbors */ }
};

template<typename cell_state>
struct evaluator<
    count_neighbors<cell_state, von_neumann_4_neighbors>{
  static auto evaluate(/* ... */) { /* logic for 4 neighbors */ }
};
```

Listing 17: Specialized neighbor-count evaluators

We can apply the same technique to more complex AST patterns. For instance, when a `greater_than<constant<0>, count_neighbors<...>>` appears, it is semantically equivalent to finding at least one node with the given state (which is a simpler operation than counting). By fusing the comparison and counting into a single specialization, we can terminate the iteration across the neighbors the moment the target state is found, which results in a tighter, more efficient evaluator:

```
template<typename cell_state>
struct evaluator<
    greater_than<
      constant<0>,
      count_neighbors<cell_state, moore_8_neighbors>> {
  static auto evaluate(/* ... */) { /* specialized logic */ }
};
```

Listing 18: Fused count-and-compare evaluator

Visitor's return value. It may seem as though this compile-time visitor should produce a new cell state—but since it executes entirely during compilation, long before any actual grid or cell exists, it cannot compute concrete values. Instead, each partial specialization contributes a snippet of code: the body of an `evaluate` function for its AST node. When the compiler instantiates and inlines those specializations, it assembles them into a single, fully-formed runtime function. In effect, the visitor returns not a cell value but the specialized evaluation logic itself.

Because all dispatching is resolved at compile time and each `evaluate` specialization is inlined, the resulting binary should perform identically to a hand-written implementation.

5.2.2 Example. Consider a simple rule defined in Listing 19, which flips the state of a GoL cell (from *dead* to *alive* or vice versa).

```
using flip_rule = if_then_else<
    equals<neighbor_at<0, 0>, state_constant<cell_state::alive>>,
    state_constant<cell_state::alive>,
    state_constant<cell_state::dead>
>;

/* ... */ = evaluator<flip_rule>::evaluate(/* ... */);
```

Listing 19: Flip rule example

At compile time, this expands to nested `evaluator` specializations. Below is the resulting code that has been expanded from the specializations. The parts of code are colored based on the specific node in the AST that has contributed them:

```
cell_state resulting_evaluate(/* cell, grid */) {
    if (grid[cell.x + 0, cell.y + 0] == cell_state::alive) {
        return cell_state::alive;
    } else {
        return cell_state::dead;
    }
}
```

Listing 20: Result of the visitor evaluation

5.2.3 Extensibility via partial specialization. Unlike a classic OOP visitor, where every new traversed type (in our case, an AST node) forces the programmer to add an implementation, stub, or boilerplate method to every visitor class, the compile-time approach only requires an evaluator specialization for those types (nodes) that have been actually used (in the *Algorithm*). The users can introduce new node types freely, and existing evaluators continue to compile and work without change.

When an unsupported AST node is used in the *Algorithm*, the compiler will emit an error. This makes it easy to create highly specialized evaluators that handle a subset of the language with maximal performance without polluting the codebase with unused methods or risking silent failures.

5.3 Advanced evaluators

Until now, we have described only the simple evaluator, which operates on a standard grid stored as an array of enum states; to showcase the full power of Cellato DSL, we have implemented more sophisticated evaluators that exploit alternative memory layouts and bit-level parallelism.

We will not explore every detail of these optimizations (e.g., the SIMD bit manipulations in the bit-plane evaluator); instead, we demonstrate the flexibility gains these custom evaluators enable.

5.3.1 Bit-array evaluator. In the bit-array layout, multiple cell states are packed into a single machine word (either a 32- or 64-bit unsigned integer). Depending on the number of bits required per state, some bits may serve as padding (if the number of bits per state does not divide the size of the word). To translate between the enum-based representation and the packed format, the user provides a *state dictionary* that enumerates all possible states. With the dictionary in place, the grid class can pack and unpack words to and from the original row-major format.

```
using fire_dictionary = state_dictionary<
    cell_state::empty,
    cell_state::tree,
    cell_state::fire,
    cell_state::ash
>;
```

Listing 21: State dictionary for Forest-Fire CA

This dictionary could also be inferred automatically using a static visitor pattern applied to the Algorithm expression at compile time; however, when not all states appear explicitly (e.g., a rule that merely shifts the values of the grid without explicitly naming all cell states), the complete dictionary cannot be deduced. Therefore, we have opted for an explicit dictionary in all cases.

The bit-array evaluator then treats each word as a mini-grid: it iterates over its subcells, invokes a subcell evaluator—identical to the simple evaluator except for a custom `neighbor_at` that reads bits and repacks the results into the output word. From the traverser's perspective, the grid simply appears smaller along one dimension, and the traverser code requires no changes.

5.3.2 Bit-plane evaluator. The bit-plane layout extends this concept further: for an n-bit state, it allocates n independent bit arrays, each storing one bit-plane for all cells. This design enables explicit SIMD-style evaluation since bitwise operations on an entire plane process many cells in parallel without relying on compiler autovectorization.

As before, the user defines the same `state_dictionary` and only needs to swap evaluator and layout types. The Algorithm and traverser remain unchanged, underscoring the complete decoupling of the rule logic, memory representation, and execution strategy.

6 Evaluation

As shown in Section 5, Cellato DSL cleanly decouples the three main components of cellular automata, computation rule set, memory representation, and grid traversal. To assess its expressiveness and flexibility, we compare it against four prominent stencil and DSL frameworks: Kokkos [16], GridTools [1], Halide [14], and AN5D [12]. These frameworks were chosen for their industrial maturity and the diversity of their rule-definition syntaxes; details can be found in Section 7. We focus on four key capabilities that users should be able to adjust without modifying the core automaton rule:

- **Platform independence:** support for CPU, CUDA, MPI, and possibly other methods of execution.
- **Memory layout independence:**
  - Changing array major order (row- vs. column-major).
  - Bit-packed array encoding.
  - More exotic layouts (like bit planes).
- **Execution model independence:** the ability to express explicit vectorization or hierarchical parallelism.
- **Compilation independence:** whether external tools beyond common C++ compiler and well-established libraries like CUDA or MPI are required.

We use the Game of Life as a comparative example since its rule is well-known and simple to reason about.

Kokkos

Kokkos provides multi-device support for back-ends – including OpenMP and pthreads for CPU or CUDA and HIP for GPU – via a rather generic `parallel_for` interface. Execution is controlled via execution policies like `MDRangePolicy`, which allow for explicit tiling and specifying the execution space (e.g., CUDA). Memory order agnosticism is achieved through Kokkos `View` abstractions that can express multi-dimensional arrays with arbitrary strides and extents, usually using the `LayoutLeft` (column-major) or `LayoutRight` (row-major) policies. The framework does not natively support bit-packed arrays or bit-planes.

Listing 22 shows a Kokkos kernel for the Game of Life specified as a lambda expression executed in the CUDA execution space. It performs computation on a 2D region with one-cell-wide padding around the grid. It requires linking against the Kokkos library and compiling with the compiler for the target platform (e.g., NVCC for CUDA). The framework does not provide fine-grained control over the optimization of the stencil evaluation, but it automatizes necessary data movements between the host and device and allows for specifying the tiling of the computation.

```
Kokkos::parallel_for(
  "GameOfLife",
  Kokkos::MDRangePolicy<Cuda>(
    {{1,1}}, {{width-1,height-1}}
  ),
  KOKKOS_LAMBDA(int x, int y){
    auto n = grid(x-1,y-1) + ... + grid(x+1,y+1);
    next(x,y) = grid(x,y)
      ? (n==2 || n==3)
      : (n==3);
  }
);
```

Listing 22: Kokkos Game of Life implementation

GridTools

GridTools employs C++ templates to generate platform-specific code at compile time. It supports CPU (OpenMP) and GPU backends. GridTools uses a functor-based approach with a generic functor with input and output accessors that specify the extents of the accessed data (neighbors). The framework supports multi-dimensional layouts with specified strides and extents, allowing for flexible memory representations, but it does not natively support bit-packed arrays or bit-planes. The whole framework is header-only, and the code generation is done at compile time using C++ metaprogramming techniques. The target platform is determined implicitly by the compiler used to compile the code (e.g., NVCC for CUDA).

Listing 23 shows a GridTools functor for the Game of Life, which defines the stencil operation using input and output accessors. The `in` accessor specifies a 3×3 neighborhood around the current cell. The framework can optimize a chain of such functors by automatizing the tiling and caching of intermediate results but does not provide fine-grained control over the optimization of the stencil evaluation.

```
struct GameOfLife {
  using in  = in_accessor<0,extent<-1,1,-1,1>>;
  using out = out_accessor<1>;
  using parms = make_param_list<in,out>;

  template<class Eval>
  void apply(Eval&& eval){
    auto n = eval(in(-1,-1) + ... + in(1,1));
    if (eval(in(0,0)) == 1)
      eval(out(0,0)) = (n == 2 || n == 3);
    else
      eval(out(0,0)) = (n == 3);
  }
};
```

Listing 23: GridTools Game of Life functor

Halide

Halide expresses computation as a pipeline of `Func` objects, with strictly separated scheduling directives controlling execution. It supports CPU and GPU (CUDA, OpenCL). The framework uses a domain-specific language to define the computation and scheduling of the computation pipeline. The schedule directives applied to the `Func` objects control how the computation is executed, including explicit control over parallelism, vectorization, and data caching.

The framework is built on top of a polyhedral model and the LLVM compiler infrastructure. Halide is specific in that the whole program is defined within C++ regardless of the target platform. However, its compilation requires the Halide compiler and runtime, and the DSL is quite limited in expressivity and integration with other C++ code. Although Halide is powerful for image processing tasks, it is not suitable for computing cellular automata as it does not natively support computing over bit-packed arrays or bit-planes. It automatically outperforms naïve cellular automata implementations by employing a polyhedral model to optimize the computation; however, its performance does not reach that of state-of-the-art implementations such as that by Fujita et al. [8], even with user-directed optimizations.

Listing 24 shows a simple Halide pipeline for the Game of Life and schedule directives that define how the computation is mapped onto the GPU.

```
Func neighbors, step;
neighbors(x,y) = input(x-1,y-1) + ... + input(x+1,y+1);
step(x,y) = select(input(x,y)==1,
  select((neighbors==2)||(neighbors==3),1,0),
  select(neighbors==3,1,0)
);
step.compute_root()
    .gpu_tile(x,y, xi, yi, xo, yo, 16,16);
```

Listing 24: Halide Game of Life pipeline

AN5D

AN5D is a source-to-source framework that optimizes a given C stencil kernel annotated with `#pragma scop` directives. The user marks the innermost loops of the stencil with these pragmas, and AN5D generates optimized CUDA code using temporal and spatial blocking at the register level. The framework is based on the polyhedral model and automatically performs loop transformations to optimize the stencil code. However, it is designed to work with fixed C-style row-major arrays, which limits its flexibility in terms of memory layout, and it requires an external AN5D translator to generate the optimized CUDA code. The framework does not natively support bit-packed arrays or bit-planes, nor does it provide explicit vectorization directives.

Listing 25 shows a simple AN5D-annotated kernel for the Game of Life, where the `rule_table` is a precomputed lookup table for the next state based on the current state and the sum of its neighbors. The pragmas `#pragma scop` and `#pragma endscop` delimit the region to be optimized by AN5D.

```
// ...

#pragma scop
  for (int t = 0; t < TIMESTEP; ++t)
    for (int i = 1; i < SIDE-1; ++i)
      for (int j = 1; j < SIDE-1; ++j)
      grid[(t+1)%2][i][j] =
        rule_table[ grid[t%2][i][j] ]
                  [ /* sum of 8 neighbors */ ];
#pragma endscop

// ...
```

Listing 25: AN5D Game of Life kernel

Cellato

Cellato was designed from the ground up to satisfy the four key capabilities without modifying the core automaton rule. In all examples, the same rule definition can run on CPU, GPU, or any other execution context simply by selecting a different layout, evaluator, or traverser. The framework supports both bit-packed arrays and bit planes, allowing for efficient memory representation of cellular automata. The rule logic is agnostic to the underlying data layout, meaning that the same rule set can be executed on different memory layouts without modification.

The evaluator design can explicitly introduce SIMD-level vectorization (even for bit-packed arrays via bitwise operations). The framework is implemented as a pure C++ header-only library that can be compiled with any standard C++ compiler or NVCC for CUDA. It does not require any external code generation or JIT compilation tools, and it is designed to be easily integrated into existing C++ codebases.

6.1 Summarizing the comparison

Table 1 summarizes the key capabilities of each framework. Columns correspond to features critical for stencil-based applications — specifically, the trade-off between expressiveness and optimization potential. In addition to standard properties like platform independence, we also assess cellular-automata-specific enhancements, such as explicit bit level SIMD on bit-packed representations for the Game of Life [8].

Platform independence The same specification of the cellular automaton logical rules can be directly compiled for different platforms (CPU, GPU, etc.) without any substantial changes to the code.

Bit-packed The framework supports computing the cellular automaton on bit-packed arrays, where each cell is represented by the least possible amount of bits.

Bit-planes The framework supports computing the cellular automaton on bit-planes — i.e., a bit-packed representation that uses separate bit-arrays for individual bits.

Vectorization The framework supports explicit vectorization of the cellular automaton evaluation without relying on compiler auto-vectorization and without modifying the rule logic specification. For cellular automata with very few states (which is often the case), the best performance can be achieved if the framework can introduce bit-level parallelism in the evaluation, as SIMD-level vectorization is still subject to memory bandwidth limitations.

Native compilation The framework is implemented as a pure C++/CUDA library without requiring any external code generation or JIT compilation tools.

Optimization The framework supports specifying optimizations of the cellular automaton evaluation without modifying the rule logic specification. This includes optimizations such as tiling, caching, or other performance improvements that reduce the computational cost of evaluating the cellular automaton.

6.2 Performance Evaluation

In addition to feature comparisons, we conducted a brief performance study of our standard evaluator against a hand-written baseline. The goal was to determine whether the template-based approach incurs any significant overhead. Because a hand-coding of the bit-array and bit-plane evaluators for each automaton would be prohibitively laborious, we limited the scope of this comparison to the standard layout and evaluator.

6.2.1 Experimental setup. All benchmarks were executed on a Rocky Linux server equipped with an NVIDIA H100 GPU (80 GB, Compute Capability 9.0) and AMD EPYC 9454 48-core processors. GPU workloads used CUDA 12.8, while the CPU code was built with GCC 13.2 — both compiled with maximum optimization flags enabled.

The baseline is a straightforward, hand-written implementation of each automaton that reproduces the logic obtained by unrolling the type-level expression in Cellato. We mirror the exact sequence of conditionals and neighborhood computations to match not only semantic behavior but also the order of evaluation and branching structure.

The Cellato implementation follows the code presented in Section 4. Both implementations and all measured data are available in our replication package[2].

6.2.2 Measurement. Figure 2 presents our experimental data; the left panel shows CPU results, and the right panel shows GPU results. Each bar represents the normalized time per cell update (lower is better).

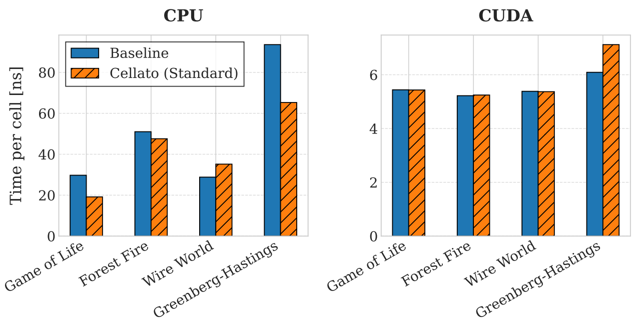

Figure 2: Normalized update time per cell on CPU (left) and GPU (right)

The performance of Cellato closely matches the baseline in most cases. The only exception is the Greenberg-Hastings model, where we observe a noticeable divergence: on the CPU, Cellato outperforms the baseline, whereas on the GPU, the baseline is faster. We

[2]https://github.com/matyas-brabec/2025-icooolps-cellato

Framework	Platform	Bit-packed	Bit-planes	Vectorization	Native compilation	Optimization
Cellato	✓	✓	✓	bit level	✓	✓
Kokkos	✓	✗	✗	✗	✓	only tiling
GridTools	✓	✗	✗	✗	✓	only caching
Halide	✓	✗	✗	SIMD	✗	✓
an5d	✓	✗	✗	✗	✗	✓

Table 1: Comparison of DSL frameworks in key capabilities

attribute this discrepancy to differences in compiler optimizations (GCC for the CPU tests and NVCC for CUDA). The Greenberg-Hastings kernel contains the most complex branching logic and state transitions, so small differences in code generation can lead to measurable performance variations. Because both compilers were invoked with maximum optimization flags, the exact machine code produced was effectively non-deterministic from the developer's perspective.

In all other cases, the timing difference between the baseline and Cellato is negligible (within a few percent), confirming that our template meta-programmed evaluator indeed provides zero-overhead abstraction, albeit it influences the compiler optimization heuristics.

7 Related Work

Numerous frameworks aim to simplify the development of high-performance stencil computations on CPUs and GPUs. While most are designed for general-purpose stencil operations, none focus on the unique characteristics and optimization opportunities of Cellular Automata (CA).

A prominent class of solutions relies on external code generation tools, such as AN5D [12], which is part of the PPCG framework and uses polyhedral compilation [18] to transform annotated C code with `#pragma` directives into optimized C or CUDA kernels. These tools apply scheduling and tiling transformations to improve locality and parallelism but lack support for low-level memory layout choices like bit-packing. As demonstrated by Cagigas et al. [5], such frameworks often fail to match the performance of hand-optimized CA implementations.

Another major group consists of C++-embedded DSLs with runtime code generation, such as Halide [14] and Tiramisu [2]. These systems separate algorithm specification from execution strategy via symbolic function pipelines and schedules, and internally leverage the polyhedral model to optimize loop nests. While powerful for image-processing and deep learning workloads, they are not well-suited to CA-specific update rules or representations, and they lack support for both the bit-packed and bit planes representation.

Frameworks more closely aligned with our approach use compile-time C++ metaprogramming, such as GridTools [1], Kokkos [16], RAJA [3], and Thrust [4]. GridTools – originally developed for weather and climate simulations – allows users to define stencil computations as multi-stage pipelines and supports caching across stages. Kokkos and RAJA offer device-agnostic abstractions over parallel execution via execution policies and multi-dimensional data containers (views), but their expressiveness is limited to coarse-grained parallelism and tiling. None of these frameworks provide dedicated support for CA-specific techniques such as SIMD-style rule evaluation or compact bit-level encodings.

Compared to general-purpose frameworks, Cellato is tailored specifically for cellular automata. It encodes update rules as C++ type-level ASTs, enabling complete separation of rule logic from memory layout and traversal. This design allows users to switch between layouts or evaluators without modifying the rule code – something not feasible in systems where rules are embedded in lambdas or compiled functions with fixed data representations.

8 Conclusion

We have introduced Cellato (Cellular Abstraction Tool), a C++ framework with embedded DSL for cellular automata that leverages template meta-programming to turn concise, type-level rule definitions into zero-overhead, highly efficient kernels at compile time. By cleanly separating four major concerns — *Algorithm* (CA rules), *Evaluator* (cell update), *Layout* (memory organization), and *Traverser* (iteration strategy) — Cellato lets users experiment with each aspect independently (e.g., testing new optimizations without ever touching the CA rules logic).

To showcase its expressiveness, we encoded four canonical models (Conway's Game of Life, Forest Fire, Wireworld, and Greenberg-Hastings) which cover binary to multi-state rules and both Moore and von Neumann neighborhoods. We then demonstrated three memory layouts (standard array, packed bit array, and bit-planes) and implemented matching evaluators, including an explicit SIMD bit-plane version that does not rely on compiler auto-vectorization. In each case, the same rule definition and traverser code remain unchanged.

Under the hood, a novel compile-time visitor pattern traverses the type-level AST and dispatches each node to its specialized evaluator via template specializations. This replaces classic run-time double-dispatch, ensuring that adding new node types never breaks existing evaluators, and unhandled nodes simply produce a compile-time error only if they appear in the AST. Performance measurements on both CPU and GPU back-ends confirm that this compile-time approach provides zero-overhead abstractions; however, it may affect compiler optimization decisions.

Finally, we compared Cellato against mature CA frameworks and found that it matches (or even exceeds) their core capabilities when focusing on flexibility and code reuse, since it uniquely allows users to swap memory layouts and evaluators seamlessly without manual code rewrites.

In our future research, we plan to use Cellato to experiment with low-level CA optimizations and improving parallel and distributed processing. This includes implementing distributed traverser based on MPI framework. Furthermore, we started experimenting with LLM integration that would enable automated translation of existing CA implementations and specifications into Cellato DSL code or pre-generating the right snippets of code when the user writes the rules manually.

Acknowledgments

We used AI tools for grammar and style checking and for simple text transformations that aimed at clarification.

References

[1] Anton Afanasyev, Mauro Bianco, Lukas Mosimann, Carlos Osuna, Felix Thaler, Hannes Vogt, Oliver Fuhrer, Joost VandeVondele, and Thomas C Schulthess. 2021. Gridtools: A framework for portable weather and climate applications. *SoftwareX* 15 (2021), 100707.

[2] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

[3] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable performance for large-scale scientific applications. In *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE, 71–81.

[4] Nathan Bell and Jared Hoberock. 2012. Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*. Elsevier, 359–371.

[5] Daniel Cagigas-Muñiz, Fernando Diaz-del Rio, Jose Luis Sevillano-Ramos, and Jose-Luis Guisado-Lizar. 2022. Efficient simulation execution of cellular automata on GPU. *Simulation Modelling Practice and Theory* 118 (2022), 102519.

[6] John Conway et al. 1970. The game of life. *Scientific American* 223, 4 (1970), 4.

[7] AK Dewdney. 1990. The cellular automata programs that create wireworld, rugworld and other diversions. *Scientific American* 262, 1 (1990), 146–149.

[8] Toru Fujita, Koji Nakano, and Yasuaki Ito. 2016. Fast simulation of Conway's Game of Life using bitwise parallel bulk computation on a GPU. *International Journal of Foundations of Computer Science* 27, 08 (2016), 981–1003.

[9] JM Greenberg, BD Hassard, and SP Hastings. 1978. Pattern formation and periodic structures in systems modeled by reaction-diffusion equations. (1978).

[10] T. Zachary Laine. 2018. Boost YAP: Yet Another Preprocessor. https://www.boost.org/doc/libs/release/doc/html/yap.html

[11] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rüde, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, et al. 2014. ExaStencils: Advanced stencil-code engineering. In *Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II 20*. Springer, 553–564.

[12] Kazuaki Matsumura, Hamid Reza Zohouri, Mohamed Wahib, Toshio Endo, and Satoshi Matsuoka. 2020. AN5D: automated stencil framework for high-degree temporal blocking on GPUs. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. ACM New York, NY, USA, 199–211.

[13] Kedar S Namjoshi and Nimit Singhania. 2016. Loopy: Programmable and formally verified loop transformations. In *International Static Analysis Symposium*. Springer, 383–402.

[14] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM* 61, 1 (2017), 106–115.

[15] Adrian S Sabau, Lang Yuan, Jean-Luc Fattebert, and John A Turner. 2023. An OpenMP GPU-offload implementation of a non-equilibrium solidification cellular automata model for additive manufacturing. *Computer Physics Communications* 284 (2023), 108605.

[16] Christian R Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S Hollman, Dan Ibanez, et al. 2021. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 805–817.

[17] Todd Veldhuizen and Blitz++ contributors. 2019. Blitz++: Multi-Dimensional Array Library for C++. https://github.com/blitzpp/blitz

[18] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–23.

[19] Jingjing Wang, Hongji Meng, Jian Yang, and Zhi Xie. 2023. Multi-GPU accelerated cellular automaton model for simulating the solidification structure of continuous casting bloom. *The Journal of Supercomputing* 79, 5 (2023), 4870–4894.

[20] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann, and Holger Fehske. 2009. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, Vol. 1. IEEE, 579–586.

[21] Haoming Zhuang, Xiaoping Liu, Xun Liang, Yuchao Yan, Jinqiang He, Yiling Cai, Changjiang Wu, Xinchang Zhang, and Honghui Zhang. 2022. Tensor-CA: A high-performance cellular automata model for land use simulation based on vectorization and GPU. *Transactions in GIS* 26, 2 (2022), 755–778.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009