Supporting Information

We evaluated DSDStudio's performance on six hardware platforms spanning integrated graphics, mainstream GPUs, and high-performance discrete GPUs (Table S1). Synthetic ODE systems were generated with species counts ranging from 4 to 16384, each species participating in up to 4 reactions with randomly assigned rate constants and initial concentrations. Simulations ran for a final time of 5 arbitrary units with absolute tolerance 10^{-6} and relative tolerance 10^{-3}. Each configuration was tested three times, with the best time recorded to minimize thermal throttling effects. Speedup is defined as CPU execution time divided by GPU execution time.

Table S1: Hardware platforms used for performance evaluation.

Platform	CPU	GPU	GPU Type
P1	11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz	NVIDIA GeForce RTX 3060 Laptop	Discrete (NVIDIA)
P2	Intel(R) Core(TM) i7-14700KF	NVIDIA GeForce RTX 5070 Ti	Discrete (NVIDIA)
P3	12th Gen Intel(R) Core(TM) i3-12100F	NVIDIA GeForce RTX 2070	Discrete (NVIDIA)
P4	12th Gen Intel(R) Core(TM) i7-12700H	NVIDIA GeForce RTX 2050	Discrete (NVIDIA)
P5	Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz	NVIDIA GeForce MX250	Discrete (NVIDIA)
P6	AMD Ryzen 7 7735H with Radeon Graphics	AMD Radeon(TM) Graphics	Integrated (AMD)

Tables S2-S7 present detailed performance measurements for hardware Platforms 1 through 6.

Table S2: Performance results for Platform 1.

Species	**CPU Time (ms)**	**GPU Time (ms)**	**Speedup**
4	1	104	0.009615
8	5	117	0.042735
16	12	126	0.095238
32	29	132	0.219697
64	51	130	0.392308
128	130	148	0.878378
256	328	171	1.918129
512	798	248	3.217742
1024	1866	304	6.138158
2048	3441	316	10.889241
4096	9195	382	24.070681
8192	22745	483	47.091097
16384	52365	773	67.742561

DSDStudio follows a modular MVVM architecture comprising five core components (Figure S1).

Table S3: Performance results for Platform 2.

Species	CPU Time (ms)	GPU Time (ms)	Speedup
4	1	105	0.009524
8	3	107	0.028037
16	8	120	0.066667
32	21	127	0.165354
64	35	122	0.286885
128	98	142	0.690141
256	232	143	1.622378
512	546	178	3.067416
1024	1176	222	5.297297
2048	2276	227	10.026432
4096	5171	301	17.179402
8192	11844	394	30.060914
16384	27229	470	57.934043

Table S4: Performance results for Platform 3.

Species	CPU Time (ms)	GPU Time (ms)	Speedup
4	1	130	0.007692
8	4	139	0.028777
16	12	147	0.081633
32	29	149	0.194631
64	46	158	0.291139
128	131	157	0.834395
256	324	182	1.780220
512	745	247	3.016194
1024	1645	303	5.429043
2048	3263	390	8.366667
4096	7992	441	18.122449
8192	16975	521	32.581574
16384	35471	805	44.063354

Table S5: Performance results for Platform 4.

Species	CPU Time (ms)	GPU Time (ms)	Speedup
4	1	110	0.009091
8	3	116	0.025862
16	8	128	0.062500
32	30	145	0.206897
64	61	165	0.369697
128	126	153	0.823529
256	319	176	1.812500
512	687	227	3.026432
1024	1365	265	5.150943
2048	3181	300	10.603333
4096	7236	376	19.244681
8192	16075	623	25.802568
16384	39793	1282	31.039782

Table S6: Performance results for Platform 5.

Species	**CPU Time (ms)**	**GPU Time (ms)**	**Speedup**
4	2	136	0.014706
8	8	144	0.055556
16	15	158	0.094937
32	47	178	0.264045
64	78	178	0.438202
128	186	183	1.016393
256	394	194	2.030928
512	913	228	4.004386
1024	1989	262	7.591603
2048	5011	430	11.653488
4096	13906	704	19.752841
8192	30184	1337	22.575916
16384	64748	2641	24.516471

Table S7: Performance results for Platform 6.

Species	**CPU Time (ms)**	**GPU Time (ms)**	**Speedup**
4	9	1702	0.005288
8	9	1555	0.005788
16	20	1622	0.012330
32	59	1712	0.034463
64	139	1757	0.079112
128	354	1773	0.199662
256	856	1747	0.489983
512	1753	1855	0.945013
1024	3813	1982	1.923814
2048	9681	1975	4.901772
4096	24080	2202	10.935513
8192	53567	3056	17.528469
16384	119695	4458	26.849484

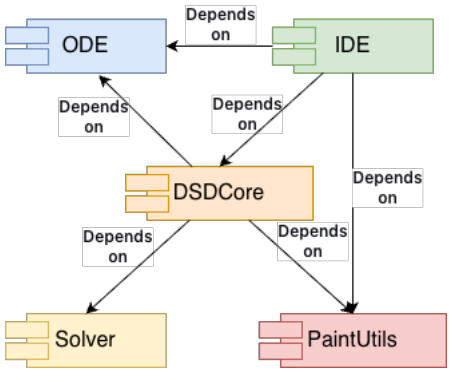
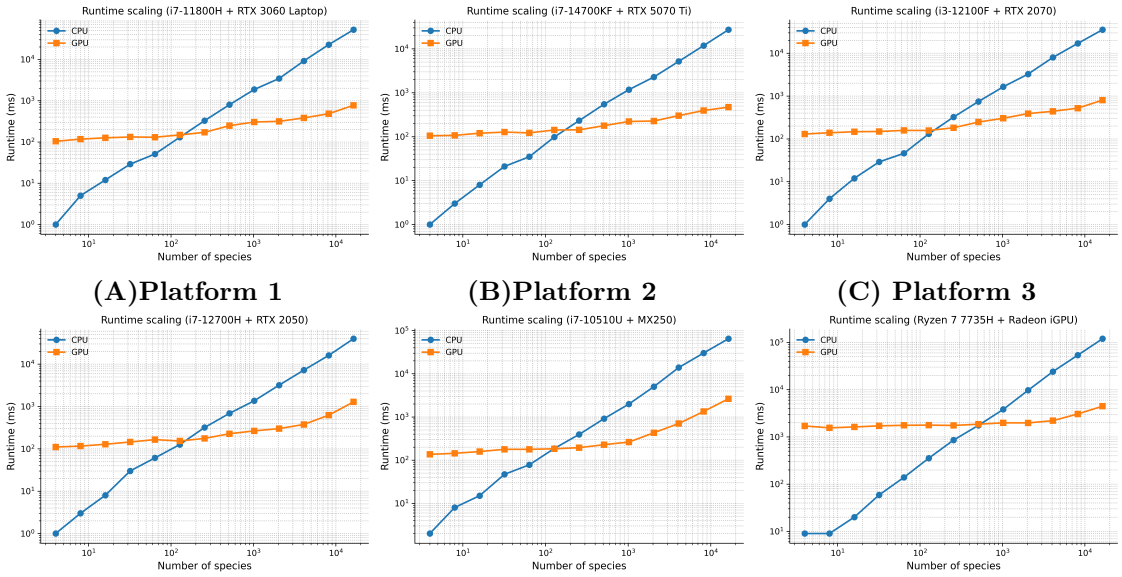


Figure S1: System architecture component diagram

Figure S2 presents the speedup curves for all six platforms. For small systems (< 100 species), GPU overhead dominates, and CPU execution is faster—GPU kernel launch latency and memory transfer costs outweigh the computational benefits. However, once the system exceeds 200–500 species, GPU acceleration becomes increasingly advantageous across all platforms.



(A)Platform 1 **(B)Platform 2** **(C) Platform 3**

(D) Platform 4 **(E) Platform 5** **(F) Platform 6**

Figure S2: Each subfigure shows the speedup (CPU time/GPU time) as a function of species count for one platform.

Figure S3 shows the class hierarchy of the ODE module, which is responsible for converting chemical reactions into ordinary differential equations and preparing them for numerical solving. The module consists of five main classes: `Reaction` stores reaction rules with rate constants; `ODEterm` represents individual terms in an ODE with coefficients and exponents; `ODEequation` collects all terms for a specific species; `ODEsystem` manages the complete equation system and provides evaluation functions; and `GPU_ODE_Data` prepares optimized data structures for GPU execution. Figure S4 illustrates the Model-View-ViewModel architecture implemented in DSDStudio, which separates the user interface from the business logic through a mediating ViewModel layer. The Model-View-ViewModel pattern separates the user interface (View) from the business logic (Model) through a mediating ViewModel layer. This design enhances maintainability, facilitates unit testing, and enables independent development of the graphical interface and computational core.

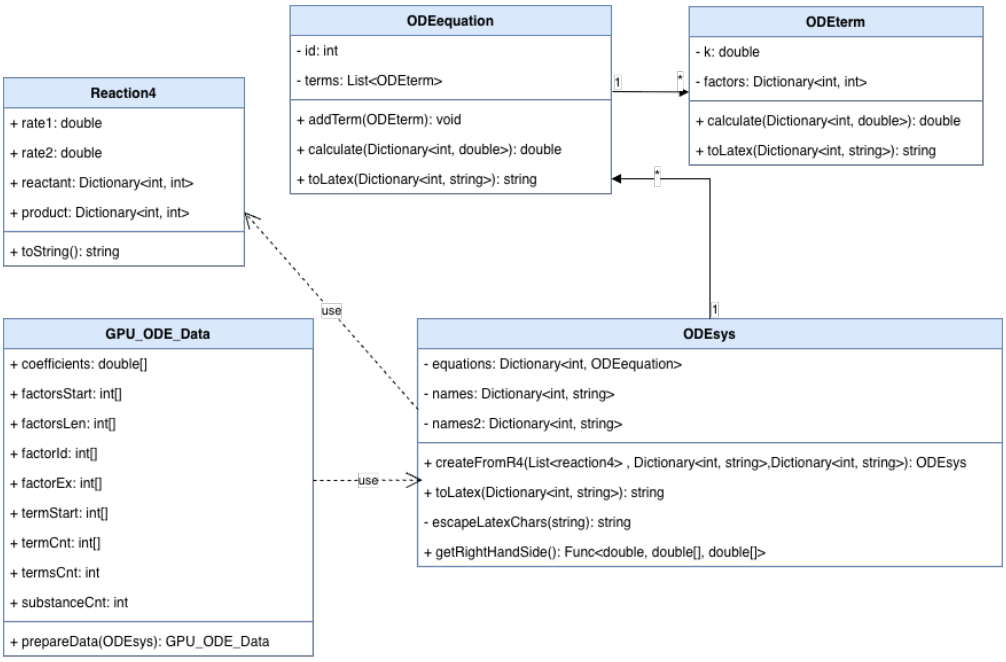


Figure S3: Class diagram of the ODE module in DSDStudio.

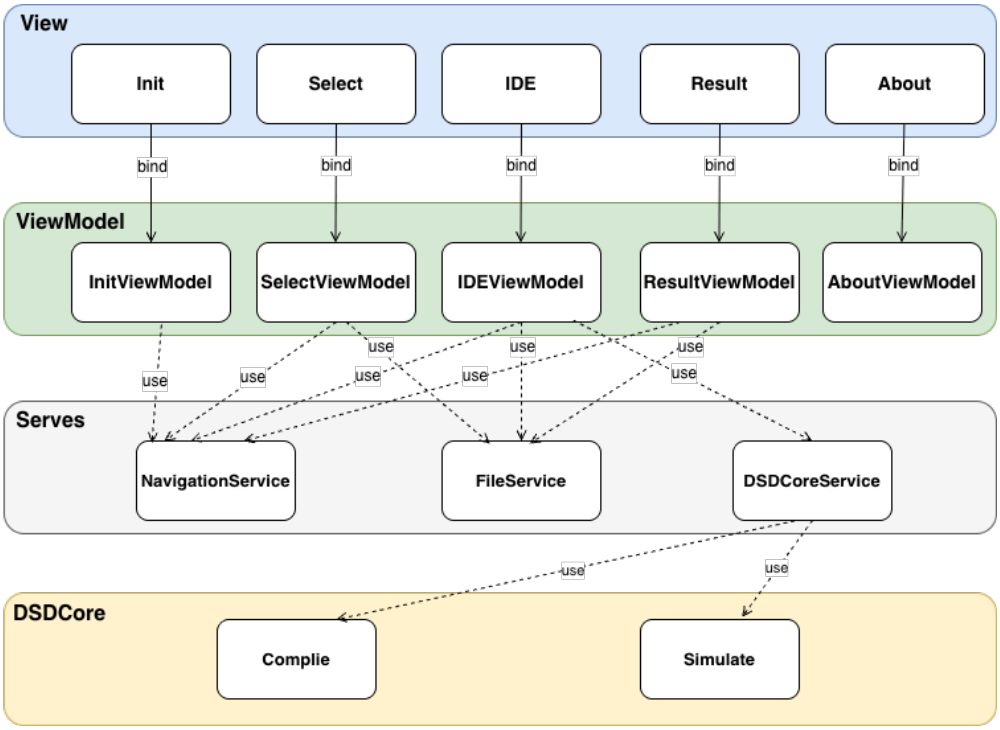


Figure S4: MVVM architecture of DSDStudio.

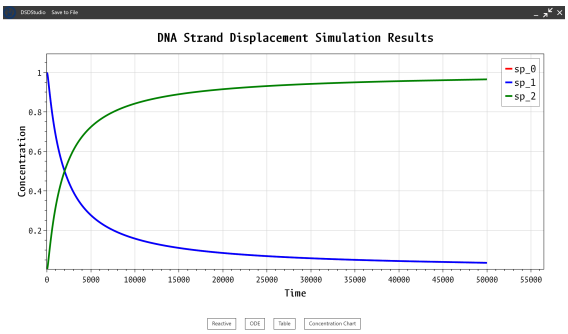
To verify simulation correctness, we compared DSDStudio's output with Visual DSD on three canonical examples.

0.1 DNA Hybridization

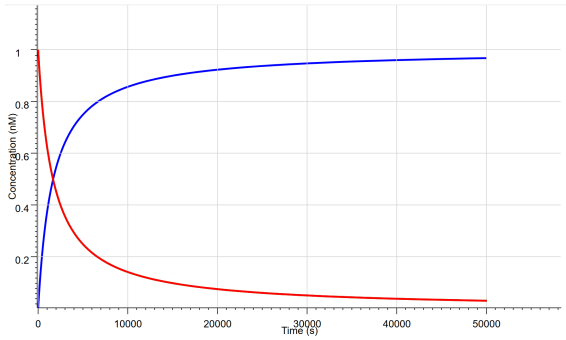
The reversible hybridization reaction

$$\langle a\hat{}b\hat{}c\rangle + \{da\hat{} * b\hat{}*\} \rightleftharpoons \{d\}[a\hat{}b\hat{}]\langle c\rangle \tag{S1}$$

was simulated with forward rate constant 0.0006 and reverse rate constant 0.0006. Figure S5 shows the concentration profiles from DSDStudio and Visual DSD, demonstrating perfect agreement. The exponential approach to equilibrium matches theoretical predictions, validating the reversible reaction implementation.



(A) DSDStudio



(B) Visual DSD

Figure S5: DNA hybridization reaction concentration profiles. The reversible reaction shows an exponential approach to equilibrium, with DSDStudio results (A) matching Visual DSD reference curves (B).

0.2 Three-Way Branch Migration

The irreversible displacement reaction

$$\langle a\hat{}b\rangle + \{a\hat{}*\}[a\hat{}b\hat{}*] \rightarrow \{a\hat{}*\}[ab] + \langle b\hat{}*\rangle \tag{S2}$$

was simulated with a rate constant of 0.0006. Figure S6 shows the characteristic dynamics where the invading strand gradually displaces the incumbent, with all four species evolving identically in both simulators. The absence of a reverse reaction produces a monotonic decay of reactants and growth of products.

0.3 AND Logic Gate

A two-input AND gate was implemented following standard DSD design patterns. Figure S7 illustrates the complete design workflow in DSDStudio. A: DSD code defines input strands and gate complex; B: DSDStudio automatically generates the reaction network; C: corresponding ODEs are derived via mass action kinetics.

The circuit produces output only when both inputs are present. Figure S7D shows DSDStudio simulation results: output concentration reaches input levels only in the High-High condition, remaining at baseline otherwise. Figure S7E shows Visual DSD reference simulations under identical conditions; the dynamics match perfectly, validating the compiler's handling of multi-step reaction pathways.

Algorithm S1 and S2 describe the GPU-accelerated Dormand-Prince adaptive step-size solver, which parallelizes rate calculations across all species within each time step using two CUDA kernels.

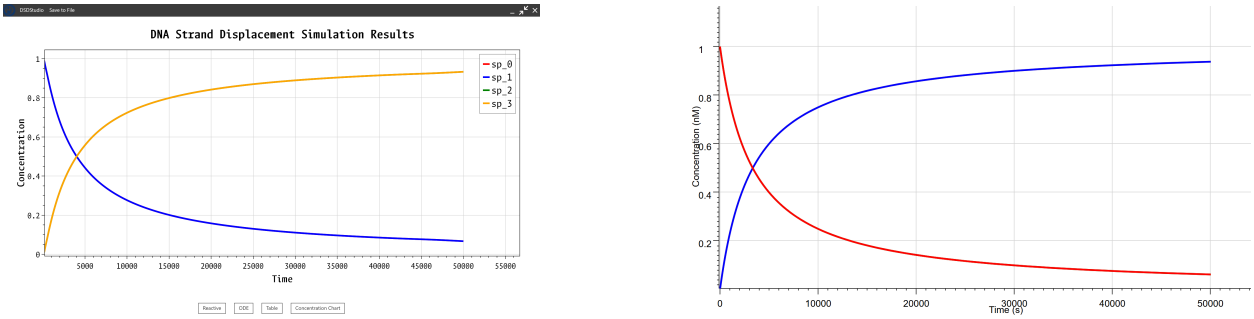
Algorithm S1 GPU-Accelerated Dormand-Prince Method

function GPURK45_SOLVE(initialConcentrations, tStart, tEnd, initialStep, tol, safetyFactor)
  results ← new List
  timePoints ← new List
  currentState ← CopyToDeviceMemory(initialConcentrations)
  timePoints.add(tStart)
  $t \leftarrow$ tStart
  $dt \leftarrow$ initialStep
  while $t <$ tEnd **do**
    if $t + dt >$ tEnd **then**
      $dt \leftarrow$ tEnd − t
    end if
    stepAccepted ← false
    while ¬ stepAccepted **do**
      $k_1 \leftarrow$ CALCULATERATES(currentState, 0)
      for stage ← 1 to 6 **do**
        tempState ← COMPUTETEMPSTATE(currentState, $k_1, \ldots, k_{\text{stage}}$, a[stage], dt)
        $k_{\text{stage}+1} \leftarrow$ CALCULATERATES(tempState, 1)
      end for
      $y_{\text{next}} \leftarrow$ currentState + $dt \cdot (b_5[0] \cdot k_1 + b_5[1] \cdot k_2 + \ldots + b_5[5] \cdot k_6)$
      $y_{\text{err}} \leftarrow dt \cdot ((b_5[0] - b_4[0]) \cdot k_1 + \ldots + (b_5[5] - b_4[5]) \cdot k_6)$
      $error \leftarrow$ MAX(ABS($y_{\text{err}}[i]$)/(ABS(currentState[i]) + ϵ))
      if $error >$ tol **then**
        $dt \leftarrow$ safetyFactor $\cdot\ dt \cdot$ (tol/error)$^{0.2}$
        $dt \leftarrow$ CLAMP(dt, $0.1 \cdot dt$, $5.0 \cdot dt$)
      else
        stepAccepted ← true
        currentState ← y_{next}
        $t \leftarrow t + dt$
        results.add(currentState)
        timePoints.add(t)
      end if
    end while
  end while
  return (timePoints, results)
end function

Algorithm S2 GPU-Accelerated Dormand-Prince Method - continued

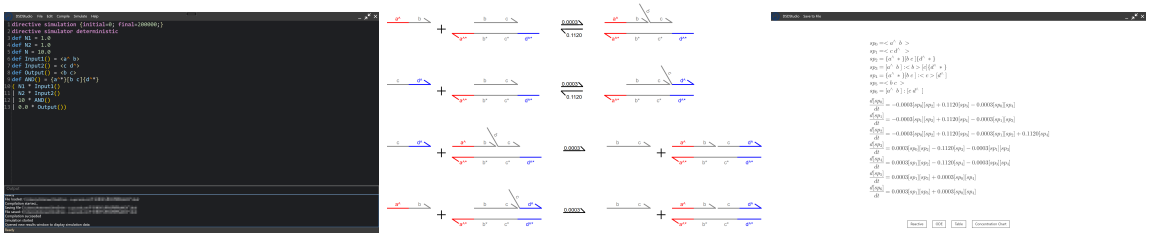
```
function CalculateRatesKernel(concentrations, tempState, calculationType)
    for parallel each substanceIdx in 0 to substanceCount − 1 do
        rate ← 0.0
        for termIdx ← 0 to equationTermCount[substanceIdx] − 1 do
            coefficient ← termCoefficients[termStart + termIdx]
            termValue ← coefficient
            for factorIdx ← 0 to factorCount − 1 do
                substanceId ← factorSubstances[factorIdx]
                exponent ← factorExponents[factorIdx]
                concentration ←
                if calculationType = 0 then
                    concentrations[substanceId]
                else
                    tempState[substanceId]
                end if
                termValue ← termValue · concentration^exponent
            end for
            rate ← rate + termValue
        end for
        rates[substanceIdx] ← rate
    end for
end function
function ComputeTempStateKernel(originalState, k_values, a_coefficients, dt, stage)

    for parallel each i in 0 to count − 1 do
        tempState[i] ← originalState[i]
        for j ← 0 to stage − 1 do
            tempState[i] ← tempState[i] +
                    dt · a_coefficients[stage · 6 + j] · k_values[j · count + i]
        end for
    end for
end function
```

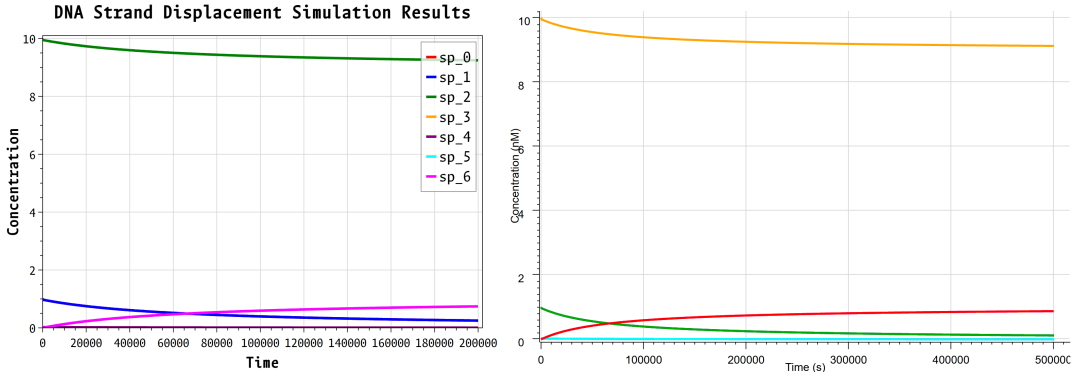


(A) DSDStudio **(B)** Visual DSD

Figure S6: Three-way branch migration reaction. The invading strand displaces the incumbent, with DSD-Studio results (A) matching Visual DSD reference curves (B).



(A) DSDStudio input code **(B)** Generated reactions **(C)** ODE system



(D) DSDStudio **(E)** Visual DSD

Figure S7: Complete AND logic gate design workflow in DSDStudio.