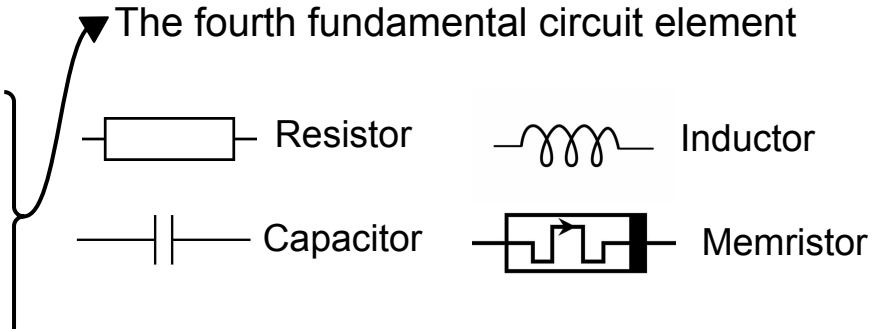
High Performance Computing

Parallelization of differential equation solver with OpenMP and CUDA

Lada Karimullina, Nikita Khoroshavtsev

31 may 2024

What is Memristor?

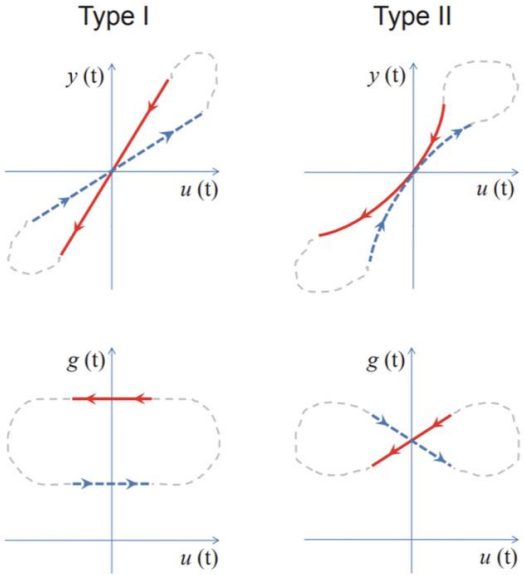


Memristor is an electric element which provided the last vacant connection between voltage, current, and their time-domain integrals, namely between flux and charge

Mathematically memristor is a device whose dynamic is defined as system of equations:

$$\begin{cases} y(t) = g(x)u(t), & \text{port equation} \\ \frac{d}{dt}x(t) = u(t). & \text{state equation} \end{cases}$$

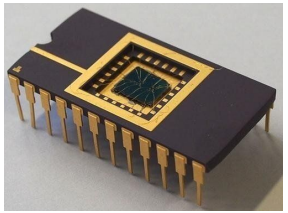
y(t) - response variable (e.g. current), u(t) - excitation variable (e.g. voltage), g(x) - memristance, x(t) - state variable, intrinsic property of the system



Pinched Hysteresis Loop

phenomenon of hysteresis with respect to the excitation and the response variable

It is a chip!



Chua L. Memristor-the missing circuit element //IEEE Transactions on circuit theory. – 1971. – Т. 18. – №. 5. – С. 507-519.

Pershin Y. V., Di Ventra M. Memory effects in complex materials and nanoscale systems //Advances in Physics. – 2011. – Т. 60. – №. 2. – С. 145-227

Memristor is a structural unit of Random Resistive Access Memory (RRAM)

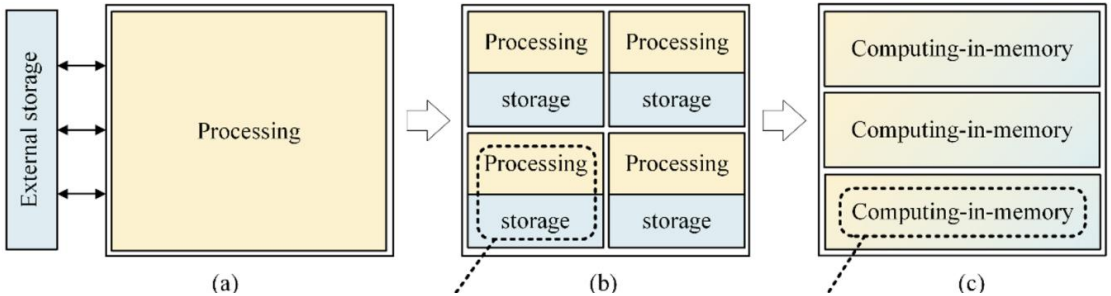


Fig. 2 The development of the computing structure.[21] (a) von Neumann architecture. (b) Near-memory computing structure. (c) Computing-in-memory structure.

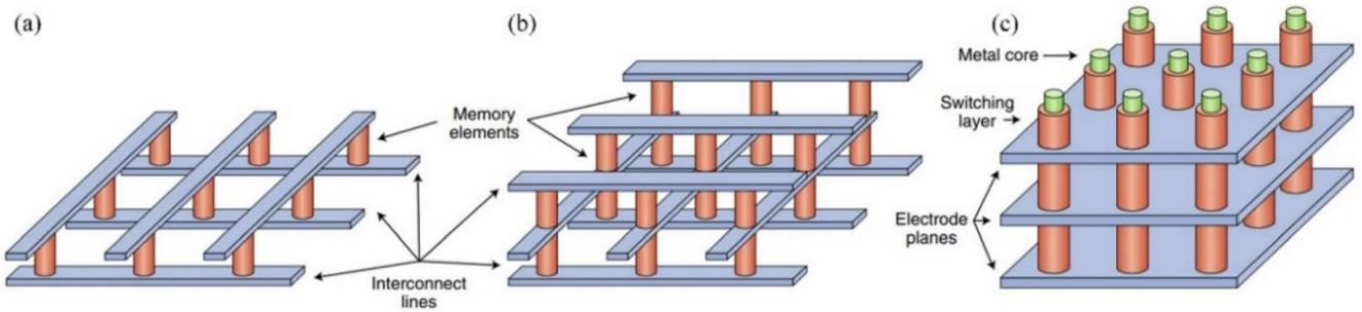
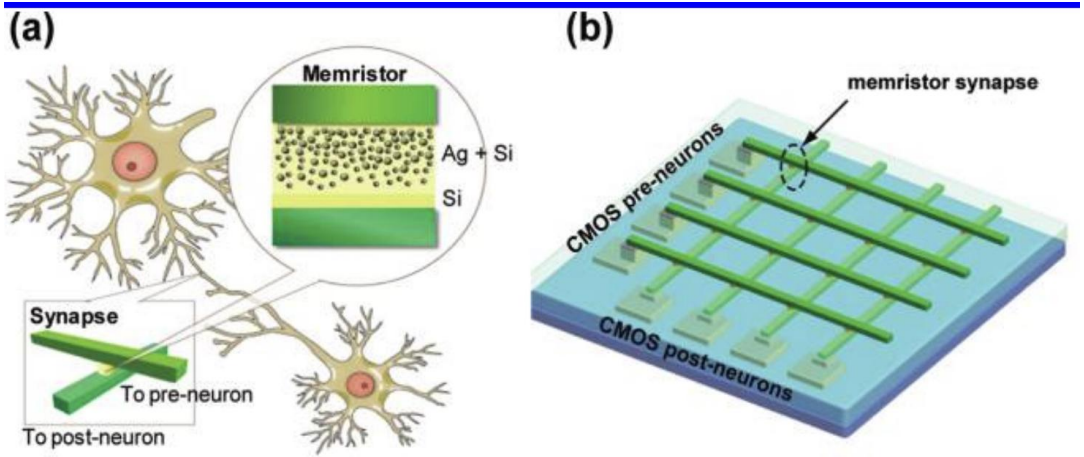


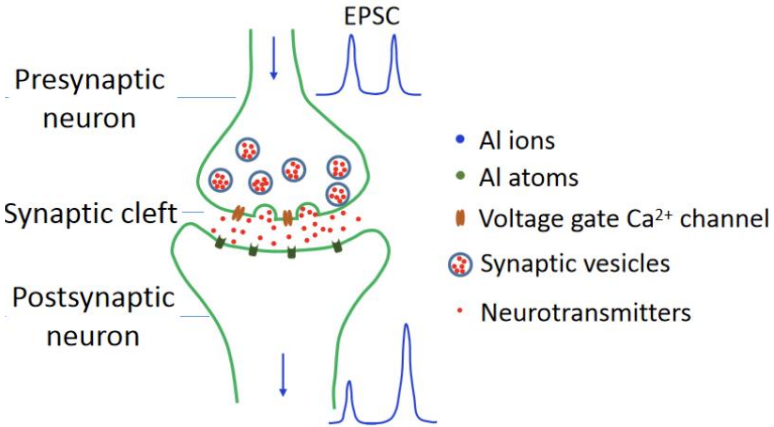
Fig. 4 RRAM across-array architecture and scaling.[25] (a) RRAM across-array structure with a single memory layer. (b) Horizontal stacked 3D across-array structure. (c) Vertical 3D across-array structure.

Wang C. et al. Research progress in architecture and application of RRAM with computing-in-memory //Nanoscale Advances. – 2023. – Т. 5. – №. 6. – С. 1559-1573.

Diffusive memristors

Diffusive memristors are volatile resistance switches that go to a low resistance state when an electrical stimulus is applied but automatically relax to the original high resistance state if the stimulus is removed.



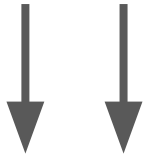


Jo S. H. et al. Nanoscale memristor device as synapse in neuromorphic systems //Nano letters. – 2010. – Т. 10. – №. 4. – С. 1297-1301.

Sueoka B. et al. A synaptic memristor based on natural organic honey with neural facilitation //Organic Electronics. – 2022. – Т. 109. – С. 106622.

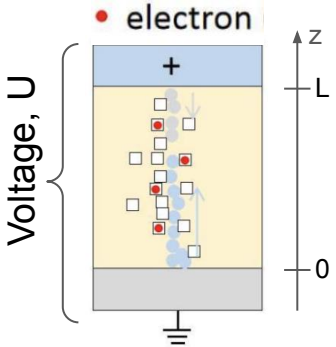
Disadvantage of diffusive memristors and solution

There is no physical model that can predict the properties a memristor will have based on its material and the applied voltage. Another issue of the diffusive memristor is the non-uniform and non-controllable relaxation time (the time it takes to go from low to high resistance states), which limits the wide-range adoption of such devices in large arrays for real-world applications



The solution is to formulate a specific and realistic physical model, that describes the microscopic processes taking place in memristive devices, and exploit it to obtain current-voltage and current- temperature dependencies of a device with given characteristics, e. g. Fermi energy, thickness and area of the device. Thus, one could understand an impact of a specific characteristic on the device's operation, which in turn results in a possibility to reproduce memristive devices with desired functionality

What model is supposed to be simulated and how?



Electron tunneling:

$$\hat{H} = -\frac{\hbar^2}{2m_e}\nabla^2 + U + \sum_{j=1}^{m} \hat{u}(\boldsymbol{r}, \boldsymbol{r}_j), \quad 0 \le z, z_j \le L$$

$$\hat{H}\psi = \varepsilon\psi$$

$\hat{u}(\boldsymbol{r}, \boldsymbol{r}_j)$ - operator of local perturbation produced by an impurity at point rj

ε - energy of electrons incident on the electrode

Diffusion of the metal ions:

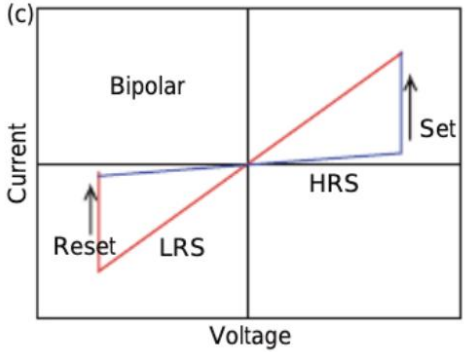
$$\partial_t n = D\partial_{zz}^2 n - \frac{\mu e V}{L}\partial_z n - 4\pi D c b n$$

D - diffusion coefficient

$\mu = \frac{D}{kT}$ - ionic mobility

V - bias, e - electron charge, c - concentration of the dangling bonds

Dynamic switching process of the memory system



Simulation setup:

Electron tunneling through flat parallel contact metal-insulator-metal (Ag−α-Si−Pt)

Length L

The barrier $U(z) = U_0 - (U_0 / L)\, z$

Tunneling through insulator occurs via so-called impurity leakage channels. These impurities are randomly dispersed throughout the insulator's volume - solve with DFT

What does the system of partial differential equations describe?

$$
\begin{cases}
\partial_t n_l = D_l \partial^2_{zz} n_l, \ -L \le z \le 0 \\
\partial_t n_r = D_r \partial^2_{zz} n_r - \frac{\mu e V_{\text{set}}}{L} \partial_z n_r, \ 0 \le z \le 2L \\
n_l(0,t) = n_r(0,t), \ D_l \partial_z n_l(0,t) = D_r \partial_z n_r(0,t) - \frac{\mu e V_{\text{set}}}{L} \partial_z n_r(0,t) \\
n_l(-L,t) = n_0, \ n_r(2L,t) = 0 \\
n_l(z,0) = \frac{n_0}{2}\big(1 - \tanh(Cz)\big), \ -L \le z \le 0 \\
n_r(z,0) = \frac{n_0}{2}\big(1 - \tanh(Cz)\big), \ 0 \le z \le 2L
\end{cases}
$$

The system consists of one-dimensional diffusion equation with the drift term induced by the applied voltage. It describes phenomenon of the diffusion of particles that form percolation channels

D is a diffusion coefficient, $\mu = \frac{D}{kT}$ is an ionic mobility, V is the bias, L is the length between two electrodes, e is the electron charge, c is the concentration of the dangling bonds.

Thomas Algorithm

```
void mul_by_tridiag(double* res, double* arr, int len_arr, float lower_d, float d, float upper_d) {
    // Handle the first element separately
    res[0] = d * arr[0] + upper_d * arr[1];

    // Handle the last element separately
    res[len_arr-1] = lower_d * arr[len_arr-2] + d * arr[len_arr-1];

    // Handle the elements in between
    for (int i = 1; i < len_arr - 1; i++) {
        res[i] = lower_d * arr[i-1] + d * arr[i] + upper_d * arr[i+1];
    }
}
```

- Tridiagonal Matrix: The matrix used in mul_by_tridiag has non-zero values only on the main diagonal and the diagonals immediately above and below it. This matrix is not explicitly stored but is defined by the values lower_d, d, and upper_d.
- Matrix-Vector Multiplication: The mul_by_tridiag function computes the product of this implicit tridiagonal matrix and a vector efficiently, leveraging the tridiagonal structure to reduce computational complexity.
- Diffusion Calculation: The inter_diffusion function uses this multiplication as part of a larger algorithm to model diffusion, adjusting boundary conditions and solving the resulting system of equations using forward and backward substitution

```
void inter_diffusion(double* B, double* alpha, double* beta, double* n, int ny, float s, float a)
{
    // Multiply by tridiagonal matrix
    mul_by_tridiag(B, &n[1], ny - 2, 0.5 * s * (1 - a), 1 - s, 0.5 * s * (1 + a));

    // Adjust boundary conditions
    B[0] = B[0] + 0.5 * s * (n[0] + n[0]) * (1 - a);
    B[ny-3] = B[ny-3] + 0.5 * s * (n[ny-1] + n[ny-1]) * (1 + a);

    // Initialize alpha and beta
    alpha[0] = (s * (1 + a) / 2) / (1 + s);
    beta[0] = B[0] / (1 + s);

    // Forward substitution
    for (int i = 1; i < ny - 3; i++) {
        alpha[i] = (s * (1 + a) / 2) / (1 + s - s * (1 - a) / 2 * alpha[i - 1]);
        beta[i] = (s * (1 - a) / 2 * beta[i - 1] + B[i]) / (1 + s - s * (1 - a) / 2 * alpha[i - 1]);
    }

    // Backward substitution
    n[ny - 2] = (B[ny - 3] + s * (1 - a) / 2 * beta[ny - 4]) / (1 + s - s * (1 - a) / 2 * alpha[ny - 4]);
    for (int j = ny - 3; j > 0; j--) {
        n[j] = alpha[j - 1] * n[j + 1] + beta[j - 1];
    }
}
```

$$\begin{cases} \partial_t n_l = D_l \partial_{zz} n_l, & -L \le z \le 0 \\ \partial_t n_r = D_r \partial_{zz} n_r - \frac{\mu e V_{set}}{L} \partial_z n_r, & 0 \le z \le 2L \end{cases}$$

Discretization: $z_i = z_0 + i\Delta z \quad \text{for} \quad i = 0, 1, \ldots, N \qquad t^k = t_0 + k\Delta t \quad \text{for} \quad k = 0, 1, \ldots, M$

Finite Difference Approximation: $\partial_{zz} n_i^k \approx \frac{n_{i+1}^k - 2n_i^k + n_{i-1}^k}{(\Delta z)^2} \qquad \partial_t n_i^k \approx \frac{n_i^{k+1} - n_i^k}{\Delta t}$

Left Region

$$\frac{n_i^{k+1} - n_i^k}{\Delta t} = D_l \frac{n_{i+1}^k - 2n_i^k + n_{i-1}^k}{(\Delta z)^2}$$

Rearranging gives:

$$n_i^{k+1} = n_i^k + \frac{D_l \Delta t}{(\Delta z)^2} \left(n_{i+1}^k - 2n_i^k + n_{i-1}^k \right)$$

Let $s = \frac{D_l \Delta t}{(\Delta z)^2}$:

$$n_i^{k+1} = n_i^k + s \left(n_{i+1}^k - 2n_i^k + n_{i-1}^k \right)$$

Right Region

The right region has an additional term for drift:

$$\frac{n_i^{k+1} - n_i^k}{\Delta t} = D_r \frac{n_{i+1}^k - 2n_i^k + n_{i-1}^k}{(\Delta z)^2} - \frac{\mu e V_{set}}{L} \frac{n_{i+1}^k - n_{i-1}^k}{2\Delta z}$$

Rearranging gives:

$$n_i^{k+1} = n_i^k + s_r \left(n_{i+1}^k - 2n_i^k + n_{i-1}^k \right) - \frac{\mu e V_{set} \Delta t}{2L\Delta z} \left(n_{i+1}^k - n_{i-1}^k \right)$$

Let $s_r = \frac{D_r \Delta t}{(\Delta z)^2}$ and $a = \frac{\mu e V_{set} \Delta t}{2L\Delta z}$:

$$n_i^{k+1} = n_i^k + s_r \left(n_{i+1}^k - 2n_i^k + n_{i-1}^k \right) - a \left(n_{i+1}^k - n_{i-1}^k \right)$$

The tridiagonal matrix A_l for the left region can be represented as:

$$A_l = \begin{bmatrix} 1-2s & s & 0 & \cdots & 0 \\ s & 1-2s & s & \cdots & 0 \\ 0 & s & 1-2s & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1-2s \end{bmatrix}$$

The tridiagonal matrix A_r for the right region can be represented as:

$$A_r = \begin{bmatrix} 1-2s_r+a & s_r-a & 0 & \cdots & 0 \\ s_r+a & 1-2s_r & s_r-a & \cdots & 0 \\ 0 & s_r+a & 1-2s_r & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1-2s_r \end{bmatrix}$$

For each time step, we solve the linear system $A\mathbf{n}^{(k+1)} = \mathbf{b}$

```
void mul_by_tridiag(double* res, const double* arr, int len_arr, float lower_d, float d, float upper_d) {
    int tid;
    res[0] = d * arr[0] + upper_d * arr[1];
    res[len_arr - 1] = lower_d * arr[len_arr - 2] + d * arr[len_arr - 1];

    #pragma omp parallel for schedule(static) shared(d, upper_d, lower_d, res, arr)
    for (int i = 1; i < len_arr - 1; i++) {
        //tid = omp_get_thread_num();
        res[i] = lower_d * arr[i - 1] + d * arr[i] + upper_d * arr[i + 1];
        //printf("tid = %d, res[%d] = %f\n", tid, i, res[i]);
    }
}
```

```
void inter_diffusion(double* B, double* alpha, double* beta, double* n, int ny, float s, float a) {
   mul_by_tridiag(B, &n[1], ny - 2, 0.5 * s * (1 - a), 1 - s, 0.5 * s * (1 + a));
   B[0] = B[0] + 0.5 * s * (n[0] + n[0]) * (1 - a);
   B[ny - 3] = B[ny - 3] + 0.5 * s * (n[ny - 1] + n[ny - 1]) * (1 + a);

   alpha[0] = (s * (1 + a) / 2) / (1 + s);
   beta[0] = B[0] / (1 + s);

   for (int i = 1; i < ny - 3; i++) {
       alpha[i] = (s * (1 + a) / 2) / (1 + s - s * (1 - a) / 2 * alpha[i - 1]);
       beta[i] = (s * (1 - a) / 2 * beta[i - 1] + B[i]) / (1 + s - s * (1 - a) / 2 * alpha[i - 1]);
   }

   n[ny - 2] = (B[ny - 3] + s * (1 - a) / 2 * beta[ny - 4]) / (1 + s - s * (1 - a) / 2 * alpha[ny - 4]);

   #pragma omp parallel for schedule(static) shared(alpha, beta)
   for (int j = ny - 3; j > 0; j--) {
       n[j] = alpha[j - 1] * n[j + 1] + beta[j - 1];
   }
}
```

adding OpenMP

Our pipeline

- Where it would be possible to make parallel cycles in the serial version of the solver

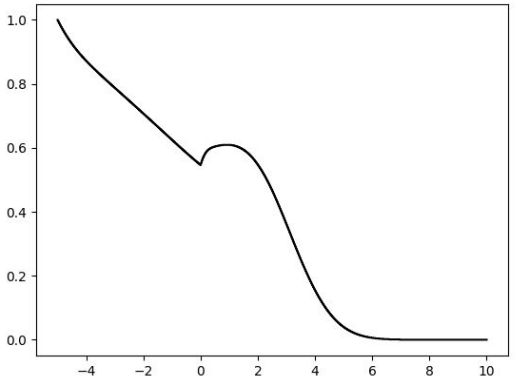
- Measure time for serial and OpenMP implementations

- Try to optimize algorithm on CUDA with changing the array size
- Measure time for CUDA implementation

Results of serial and OpenMP implementations

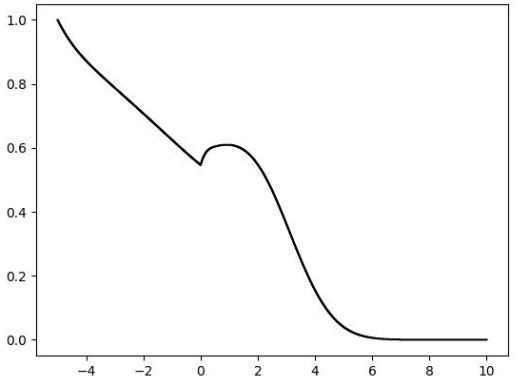
Serial

```
D_left = 0.400, num of timesteps: 1000002, a = -0.002
Using precomputed profile
Dumped in a file a profile for 1.00 seconds

real    6m16.150s
user    6m12.108s
sys     0m0.077s
```



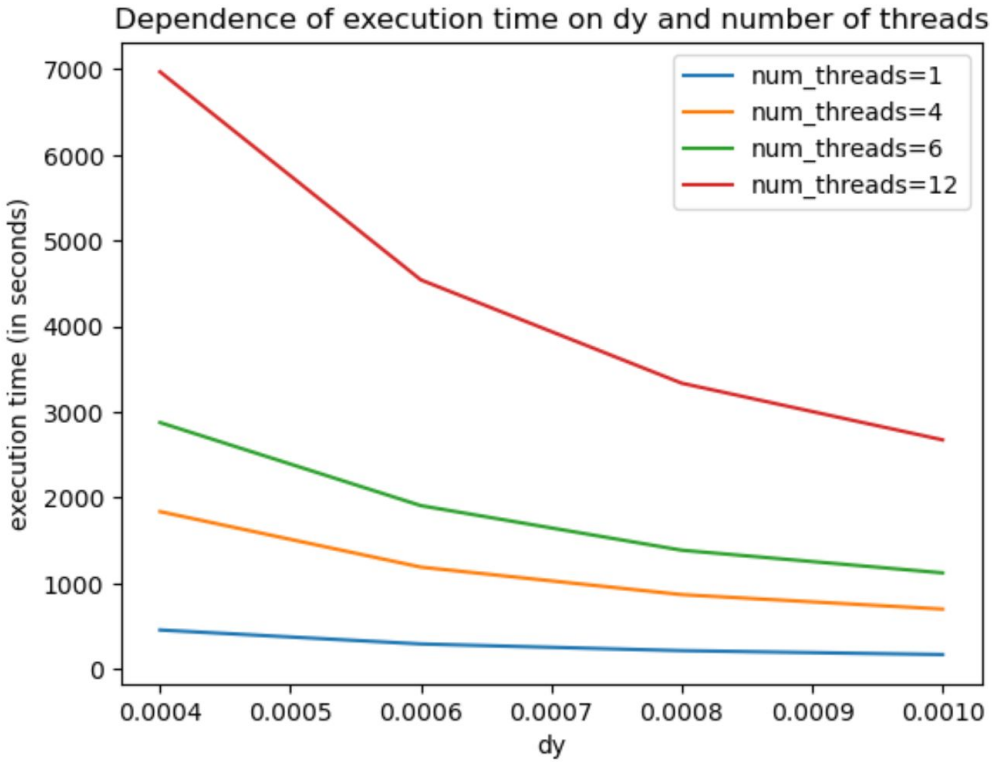
Parallel

```
D_left = 0.400, num of timesteps: 1000002, a = -0.002
Using precomputed profile
Dumped in a file a profile for 1.00 seconds

real    9m1.489s
user    15m44.981s
sys     0m2.515s
```



Results of OpenMP implementation

Parallel



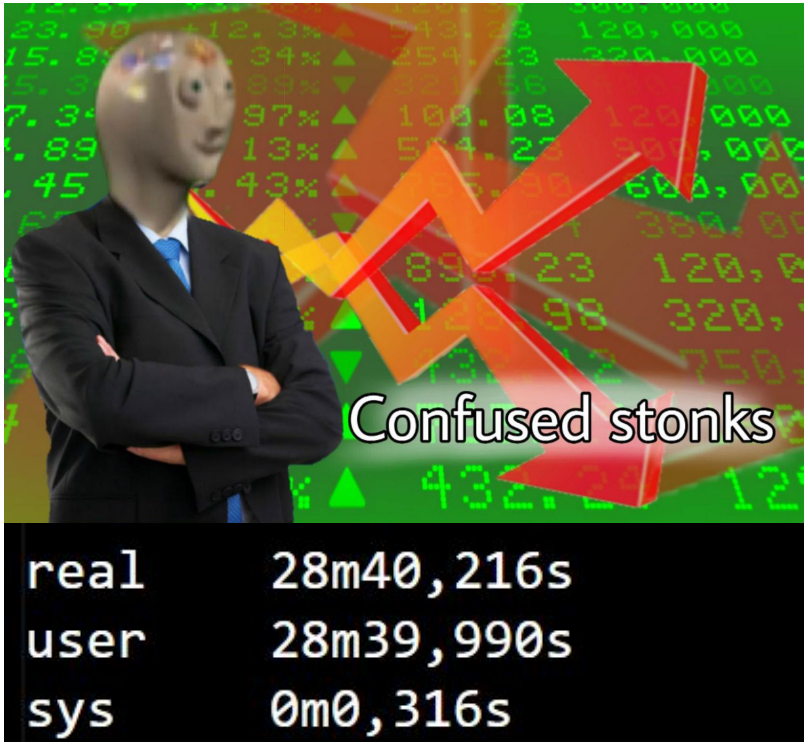
Cuda optimization

- Started from first version of code
- All arrays on CUDA
- Absolutely devastating results

Improvements?

Cuda optimization. How to improve?

- Larger array size?
- Do only some operations on GPU
- More parallelizable algorithm itself



Conclusion

- Parallelization with OpenMP did not get any improvements. The reasons could be following:
  - Creating and managing threads incurs overhead. If the parallel regions are not sufficiently large or computationally intensive, the overhead cam outweigh the benefits of parallelism
  - The function **inter_diffusion** involves several operations that are serialized (e.g., tridiagonal matrix multiplication, and the loops updating alpha and beta). These operations involve dependencies that limit parallelism - that is why possible solution could be to try implementing MPI Thomas algorithm
- Parallelization with Cuda did not help either
  - Cuda hates sequential execution and addressing by index
  - Cuda likes large arrays and threaded operations on them