Accelerating the solution of large number of delay differential equations with GPUs

Author: Dániel Nagy
Co-author: Dr. Ferenc Hegedűs

GPU days 2021

10. November 2021

MŰEGYETEM 1782

Hidrodinamikai Rendszerek Tanszék

HDR

Új Nemzeti Kiválóság Program

NEMZETI KUTATÁSI, FEJLESZTÉSI ÉS INNOVÁCIÓS HIVATAL

Some applications of delay differential equations

Field	**Cause of the delay**	**Practical example**
Modelling of epidemics	incubation and recovery time	forecasting the spread of a disease

Some applications of delay differential equations

Field	Cause of the delay	Practical example
Modelling of epidemics	incubation and recovery time	forecasting the spread of a disease
Computer control	processing time of the computers	avoiding the stability loss caused by the processing time

Some applications of delay differential equations

Field	Cause of the delay	Practical example
Modelling of epidemics	incubation and recovery time	forecasting the spread of a disease
Computer control	processing time of the computers	avoiding the stability loss caused by the processing time
Human balancing	processing time of the brain (reaction time)	rod balancing on a finger

Some applications of delay differential equations

Field	Cause of the delay	Practical example
Modelling of epidemics	incubation and recovery time	forecasting the spread of a disease
Computer control	processing time of the computers	avoiding the stability loss caused by the processing time
Human balancing	processing time of the brain (reaction time)	rod balancing on a finger
Sonochemistry	finite wave propagation velocity in fluids	simulation of sonochemical reactors

Main motivation: Solution of delay differential equations in the field of sonochemistry (Sonochemistry Research Group - Technical University of Budapest)

General form

Delay Differential Equation (DDE)

$$\begin{cases} \dot{\boldsymbol{x}}(t) & = & \boldsymbol{f}\big(t, \boldsymbol{x}(t), \boldsymbol{x}(t-\tau_1), \boldsymbol{x}(t-\tau_2) \ldots, \boldsymbol{x}(t-\tau_n)\big) \\ \boldsymbol{x}(t < t_0) & = & \boldsymbol{\eta}(t) \end{cases}$$

- $x \in \mathbb{R}^m$ are the dependent variables
- m is the system size
- $\boldsymbol{\eta}(t)$ is the initial function
- $\tau_i = \tau_i(t, \boldsymbol{x}); \quad i = 1 \ldots n$ are the delays

General form

Delay Differential Equation (DDE)

$$\begin{cases} \dot{\boldsymbol{x}}(t) & = & \boldsymbol{f}\big(t, \boldsymbol{x}(t), \boldsymbol{x}(t-\tau_1), \boldsymbol{x}(t-\tau_2) \ldots, \boldsymbol{x}(t-\tau_n)\big) \\ \boldsymbol{x}(t < t_0) & = & \boldsymbol{\eta}(t) \end{cases}$$

- $x \in \mathbb{R}^m$ are the dependent variables
- m is the system size
- $\boldsymbol{\eta}(t)$ is the initial function
- $\tau_i = \tau_i(t, \boldsymbol{x}); \quad i = 1 \ldots n$ are the delays

I only discuss constant delay

Efficient numerical solution

1. pth order Runge–Kutta (RK) method
2. $(p-1)$th order interpolation to calculate past values[1]
3. Interpolation without minimal extra calculations
    - Hermite interpolation (from steps and derivatives)
    - Continuous extension of the underlying RK method (from stages)

[1] Alfredo Bellen and Marino Zennaro. *Numerical methods for delay differential equations*. Oxford university press, 2013.

Per-thread approach

Extremely efficient in case of ordinary differential equations[2]

- Each ODE assigned to a thread
- Each ODE has different parameters
- Each thread solves the ODE with an RK method
- Each thread uses the same fixed timestep

[2] Nagy Dániel, Plavecz Lambert, and Hegedűs Ferenc. "The art of solving a large number of non-stiff, low-dimensional ordinary differential equation systems on GPUs and CPUs". In: *Communications in Nonlinear Science and Numerical Simulation* preprint (2020).

Per-thread approach

Extremely efficient in case of ordinary differential equations[2]

- Each ODE assigned to a thread
- Each ODE has different parameters
- Each thread solves the ODE with an RK method
- Each thread uses the same fixed timestep
- Same code inside the kernel as on a CPU
- No communication between threads

[2] Dániel, Lambert, and Ferenc, "The art of solving a large number of non-stiff, low-dimensional ordinary differential equation systems on GPUs and CPUs".

Per-thread approach

Extremely efficient in case of ordinary differential equations[2]

- Each ODE assigned to a thread
- Each ODE has different parameters
- Each thread solves the ODE with an RK method
- Each thread uses the same fixed timestep
- Same code inside the kernel as on a CPU
- No communication between threads
- Problem with branches (if-else statements)

[2] Dániel, Lambert, and Ferenc, “The art of solving a large number of non-stiff, low-dimensional ordinary differential equation systems on GPUs and CPUs”.

Per-thread approach

Extremely efficient in case of ordinary differential equations[2]

- Each ODE assigned to a thread
- Each ODE has different parameters
- Each thread solves the ODE with an RK method
- Each thread uses the same fixed timestep
- Same code inside the kernel as on a CPU
- No communication between threads
- Problem with branches (if-else statements)

Alternative approach: using the GPU for vector and matrix operations.

[2] Dániel, Lambert, and Ferenc, "The art of solving a large number of non-stiff, low-dimensional ordinary differential equation systems on GPUs and CPUs".

Interpolation of memory usage.

- Interpolation between past values (because of the delay)

Interpolation of memory usage.

- Interpolation between past values (because of the delay)
  1. Data written to the global memory after each step – Save actual state
  2. Data read from the global memory before each step – Read previous state

Interpolation of memory usage.

- Interpolation between past values (because of the delay)
  1. Data written to the global memory after each step – Save actual state
  2. Data read from the global memory before each step – Read previous state
- Allocates a lot of global memory
- Example: 10000 first order DDEs each with 10000 steps $\rightarrow$ allocates 1.5 GB memory
- Solution for fixed timestep: circular memory, overwriting values which won't be used anymore

Interpolation of memory usage.

- Interpolation between past values (because of the delay)
  1. Data written to the global memory after each step – Save actual state
  2. Data read from the global memory before each step – Read previous state
- Allocates a lot of global memory
- Example: 10000 first order DDEs each with 10000 steps $\rightarrow$ allocates 1.5 GB memory
- Solution for fixed timestep: circular memory, overwriting values which won't be used anymore

Algorithm

- General purpose DDE solver in the MPGOS[3] package
  - Arbitrary number of dependent variables
  - Arbitrary number of constant delays
  - Arbitrary number of parameters
- Written in CUDA C++

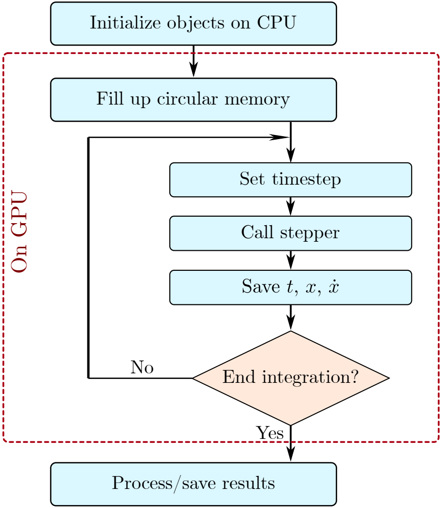
[3] Ferenc Hegedűs. *Massively Parallel GPU-ODE Solver (MPGOS)*. URL: https://www.gpuode.com/.

Algorithm

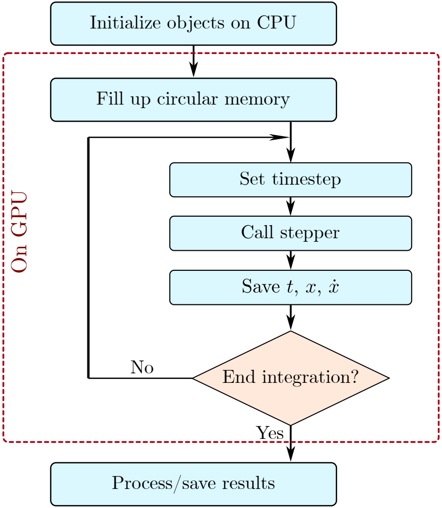
- General purpose DDE solver in the MPGOS[3] package
  - Arbitrary number of dependent variables
  - Arbitrary number of constant delays
  - Arbitrary number of parameters
- Written in CUDA C++
- Methods used
  - 4th order traditional explicit Runge–Kutta method
  - 3rd order Hermite-interpolation

[3] Hegedűs, *Massively Parallel GPU-ODE Solver (MPGOS)*.

Main steps

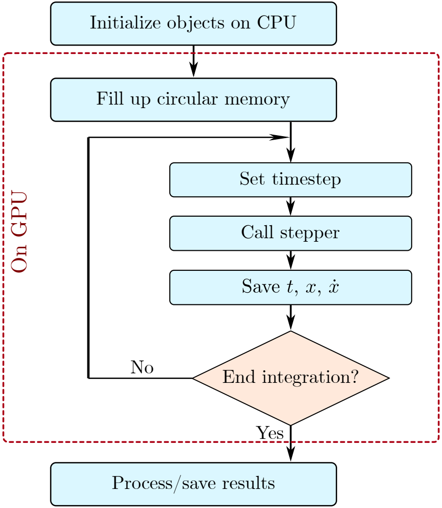

Initialize objects on CPU
On GPU
Fill up circular memory
Set timestep
Call stepper
Save t, x, $\dot{x}$
End integration?
No
Yes
Process/save results

Main steps



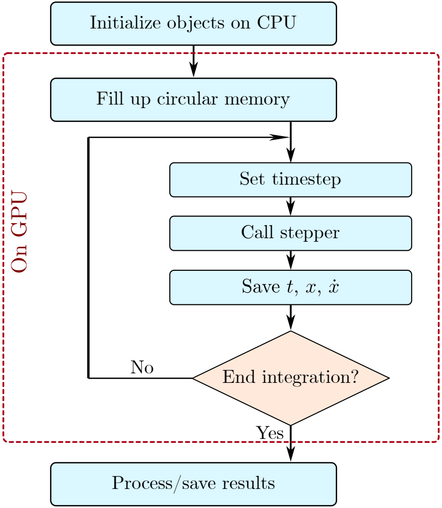
- Initialization on CPU
  - Set solution domain and timestep
  - Set delays to variables and initial functions
  - Set control parameters

Main steps



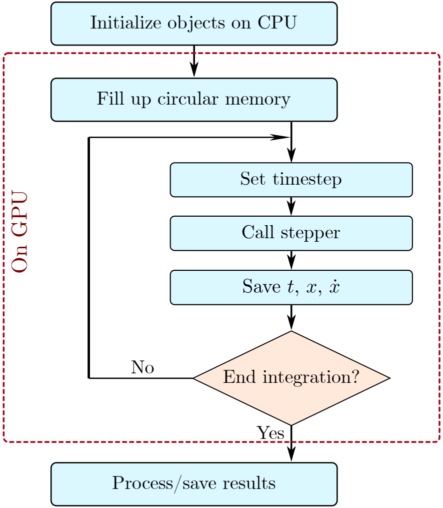
- Initialization on CPU
  - Set solution domain and timestep
  - Set delays to variables and initial functions
  - Set control parameters
- Fill up circular memory on GPU
  - Dense output only to delayed variables
  - Circular memory is initialized with discrete points of the initial function
  - Homogenous code

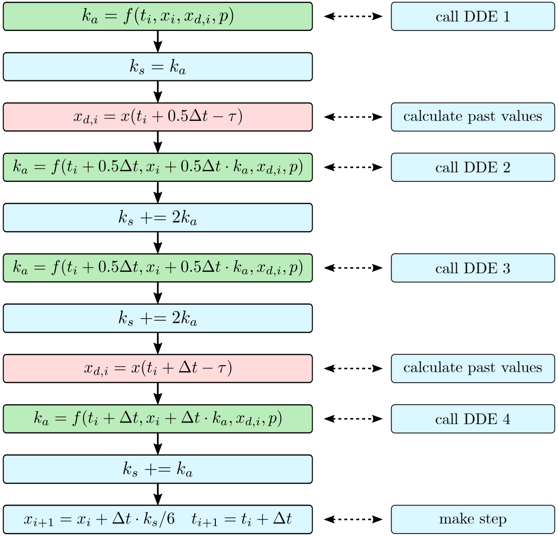
Main steps



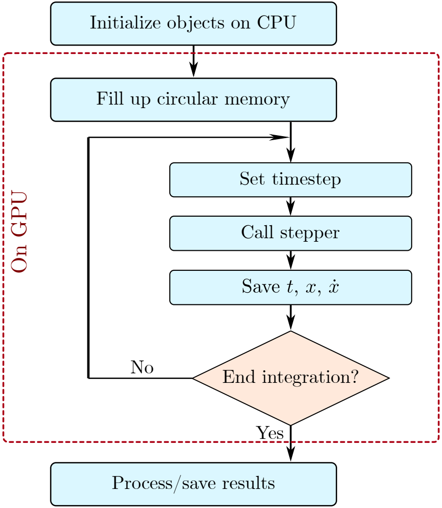
- Initialization on CPU
  - Set solution domain and timestep
  - Set delays to variables and initial functions
  - Set control parameters
- Fill up circular memory on GPU
  - Dense output only to delayed variables
  - Circular memory is initialized with discrete points of the initial function
  - Homogenous code
- Set timestep
  - Usually constant except for the end
  - May be changed due to event handling (not implemented yet)

Call stepper (4 stage but only 2 interpolation)



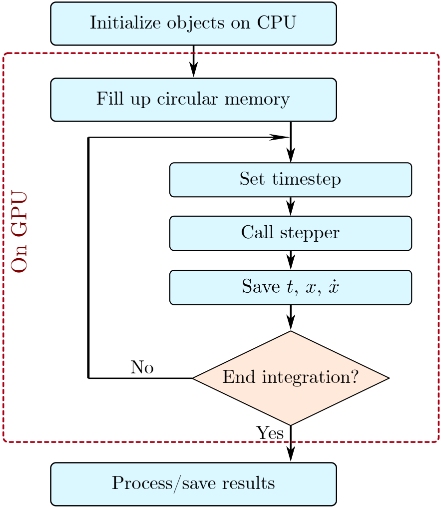


Main steps



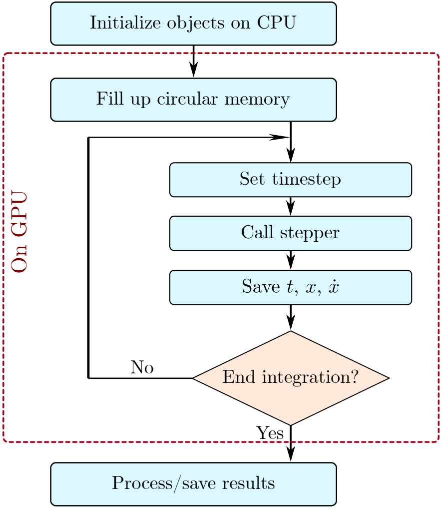
- Save $t, x, \dot{x}$
  - Saving the results of the step for later interpolation
  - Only for variables with dense output
  - Call user defined function (find local max/min)
  - Aligned and coalesced memory access

Main steps



- Save $t, x, \dot{x}$
  - Saving the results of the step for later interpolation
  - Only for variables with dense output
  - Call user defined function (find local max/min)
  - Aligned and coalesced memory access
- End integration?
  - Endtime is reached or not
  - If not next iteration
  - If yes exit the kernel

Main steps



- Save $t, x, \dot{x}$
  - Saving the results of the step for later interpolation
  - Only for variables with dense output
  - Call user defined function (find local max/min)
  - Aligned and coalesced memory access
- End integration?
  - Endtime is reached or not
  - If not next iteration
  - If yes exit the kernel
- Process/save results
  - Copy data to CPU (final steps, circular memory, user defined outputs)

Testing the performance

Codes[4]

- MPGOS (algorithm described earlier, general)
- Problem specific GPU codes (not general)
- Problem specific CPU codes (not general)
- Commercial programs (Julia only on CPU)

[4] Nagy Dániel. *DDE solver tests*. URL: https://github.com/nnagyd/DDE_solver_tests.

Testing the performance

Codes[4]

- MPGOS (algorithm described earlier, general)
- Problem specific GPU codes (not general)
- Problem specific CPU codes (not general)
- Commercial programs (Julia only on CPU)

Hardware

- NvidiaGTX Titan Black (1882 GFLOPS)
- Intel Core i7-10510U (39.2 GFLOPS)

[4] Dániel, *DDE solver tests*.

Testing the performance

Codes[4]

- MPGOS (algorithm described earlier, general)
- Problem specific GPU codes (not general)
- Problem specific CPU codes (not general)
- Commercial programs (Julia only on CPU)

Hardware

- NvidiaGTX Titan Black (1882 GFLOPS)
- Intel Core i7-10510U (39.2 GFLOPS)

Test problems

- Delayed logistic equation

$$x'(t) = x(t) \cdot \left[p - x(t-1)\right]$$
$$x(t \leq 0) = \eta(t) = 1.5 - \cos(t)$$

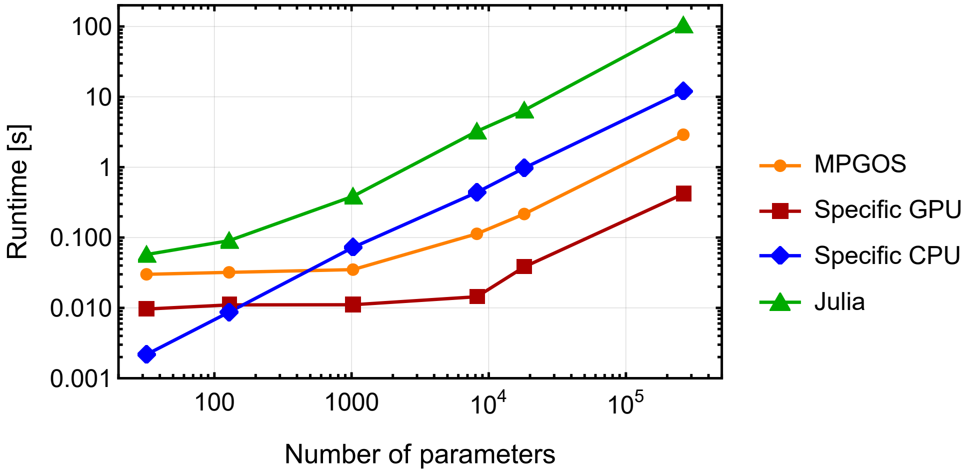
  N_p is the number of different p parameters to test. Solution with 10000 timesteps

- Delayed Lorenz equation

[4] Dániel, *DDE solver tests*.

Runtime comparison on the Logistic equation

Logistic equation (1st order, 2 arithmetic operations and 1 delay)



Analysing GPU code performance

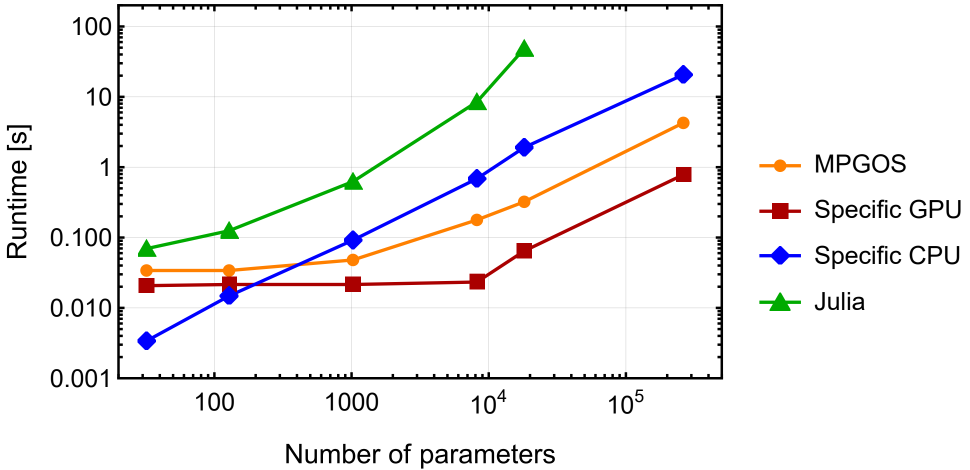
Logistic equation (1st order, 2 arithmetic operations and 1 delay)

For $N_p = 262144$

Metric	Specific GPU	MPGOS GPU
Runtime [s]	0.413	2.896
Threads	32×8192	16×16384
Blocks	128	128
Achieved occupancy	0.27	0.29
Eligible Warps Per Cycle	3.8	1.26
Memory bandwidth [%]	64	60
Global Memory Load Efficient [%]	95.6	100
Global Memory Store Efficient [%]	95.6	100
Double FLOP Efficiency [%]	26.6	3.83

Runtime comparison on the Lorenz equation

Lorenz equation (3rd order, 9 arithmetic operations and 1 delay)



Analysing GPU code performance

Lorenz equation (3rd order, 9 arithmetic operations and 1 delay)

For $N_p = 262144$

Metric	Specific GPU	MPGOS GPU
Runtime [s]	0.560	4.013
Threads	4×65536	2×131072
Blocks	128	128
Achieved occupancy	0.43	0.24
Memory bandwidth [%]	75	61
Global Memory Load Efficient [%]	91.7	100
Global Memory Store Efficient [%]	95.6	100
Double FLOP Efficiency [%]	44.0	4.9

Summary of the performance test

Performance

- Problem specific GPU solver is 30× faster than problem specific CPU solver (50× double GFLOPS)

Summary of the performance test

Performance

- Problem specific GPU solver is $30\times$ faster than problem specific CPU solver ($50\times$ double GFLOPS)
- MPGOS is $30\times$ faster than Julia (fastest commercial CPU solver)
- MPGOS is $6-8\times$ slower than optimal
- Larger systems has better performance, because it requires more arithmetic operation

Summary of the performance test

Performance

- Problem specific GPU solver is 30× faster than problem specific CPU solver (50× double GFLOPS)
- MPGOS is 30× faster than Julia (fastest commercial CPU solver)
- MPGOS is $6 - 8\times$ slower than optimal
- Larger systems has better performance, because it requires more arithmetic operation

Possible MPGOS improvements

- Improving performance
- Event handling
- Adaptive methods

Adaptive Runge–Kutta solver on GPU

Problem

- Each thread has a different timestep
- Each thread reads past values with different indices
- Those indices are far in the memory $\rightarrow$ extremely low performance

Adaptive Runge–Kutta solver on GPU

Problem

- Each thread has a different timestep
- Each thread reads past values with different indices
- Those indices are far in the memory $\rightarrow$ extremely low performance

Solution: Heterogenous CPU-GPU solver

- Control logic on CPU
- A kernel is called for each stage (called several times in a step)
- A kernel only performs arithmetic calculations $\rightarrow$ high efficency
- Efficiency is lost because data must be copied between the CPU and GPU $\rightarrow$ Overlaping calculations

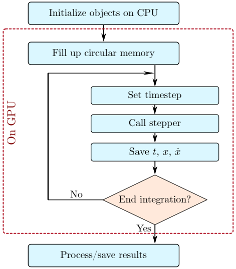
Accelerating DDE solvers on GPUs

Efficient algorithm

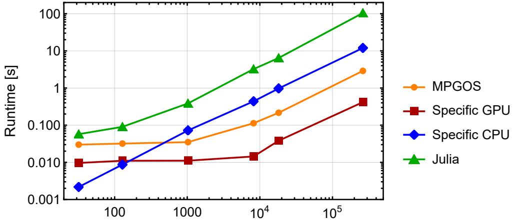
- 4th order ERK
- 3rd order Hermite interpolation
- Interpolation without extra calculations

Per thread approach

- Each ODE assigned to a thread
- Each ODE has different parameters



Runtime comparision



Code Metrics

Metric	Specific GPU	MPGOS GPU
Runtime [s]	0.413	2.896
Threads	32×8192	16×16384
Blocks	128	128
Achieved occupancy	0.27	0.29
Eligible Warps Per Cycle	3.8	1.26
Memory bandwidth [%]	64	60
Global Memory Load Efficient [%]	95.6	100
Global Memory Store Efficient [%]	95.6	100
Double FLOP Efficiency [%]	26.6	3.83

Thank you for your attention!

- Bellen, Alfredo and Marino Zennaro. *Numerical methods for delay differential equations*. Oxford university press, 2013.
- Dániel, Nagy, Plavecz Lambert, and Hegedűs Ferenc. "The art of solving a large number of non-stiff, low-dimensional ordinary differential equation systems on GPUs and CPUs". In: *Communications in Nonlinear Science and Numerical Simulation* preprint (2020).
- Dániel, Nagy. *DDE solver tests*. URL: https://github.com/nnagyd/DDE_solver_tests.
- Hegedűs, Ferenc. *Massively Parallel GPU-ODE Solver (MPGOS)*. URL: https://www.gpuode.com/.