PARALLELISATION OF EQUATION-BASED SIMULATION PROGRAMS USING KERNEL CODE GENERATION TECHNIQUES[*]

DRAGAN D. NIKOLIĆ[†]

Abstract. In this work, a methodology for efficient evaluation of model equations in equation-based simulation programs is presented. A typical equation-based model consists of a coupled set of partial-differential equations (often referred to as kernel equations) such as mass, heat and momentum balances and transport equations, and additional algebraic and ordinary differential equations (auxiliary equations) such as boundary conditions and process performance indicators. The methodology implements the concept of kernel equations which represent a group of identical mathematical expressions operating on different variables and provides: (a) an API for specification of kernel equations, (b) a method for generation of the source code for kernels in multiple languages targeting different APIs/frameworks, and (c) a method for evaluation of coupled kernels and auxiliary equations on multiple computing devices/architectures. The auxiliary equations are typically evaluated using general purpose processors while kernels are evaluated on streaming processors/accelerators or heterogeneous systems. The methodology is implemented as a part of the Open Compute Stack (OpenCS) framework. The kernels approach is applied to simulation of two large-scale models. Two performance critical phases of the numerical solution are analysed: evaluation of residuals and evaluation of derivatives (used for a preconditioner). Execution times of individual phases and the overall performance are compared and discussed. The performance of the kernels concept are compared to the performance of the existing approach where all equations are treated individually and evaluated using the Compute Stack Machine byte-code instructions, and to the tailor-made C implementations.

Key words. modelling, differential-algebraic equations, HPC, OpenCL, OpenMP, OpenCS

MSC codes. 34M04, 35M04, 68U04

1. Introduction.

Large scale systems of non-linear (partial-)differential and algebraic equations (ODE or DAE) are found in many engineering problems. Equation-based approach is one of the efficient methods to solve this kind of models. Equation-based simulation, optimisation, parameter estimation and sensitivity analysis software requires a numerical integration of a system of differential-algebraic equations and integration of sensitivity equations by a suitable ODE/DAE solver. An efficient evaluation of model equations is of utmost importance since it can require up to 85% of the total integration time. The focus in this work is on an efficient parallel evaluation of residuals, derivatives (used as a preconditioner or a Jacobian matrix) and sensitivity residuals on various computing platforms. Since most of the modern computers and many specially designed clusters are equipped with additional stream processors/accelerators such as Graphics Processing Units (GPU) and Field Programmable Gate Arrays (FPGA), the simulation software must be specially designed to effectively take advantage of multiple architectures.

In general, a typical equation-based model consists of a coupled set of partial-differential equations and auxiliary algebraic and ordinary differential equations. An individual partial-differential equation is often treated as a distinct group of equations - a kernel equation. Kernel equations represent a homogeneous group of identical mathematical expressions operating on different variables and can be efficiently evaluated on streaming processors/accelerators or heterogeneous systems. A typical example could be mass, heat and momentum balance equations and various scalar/vector transport equations. The auxiliary equations represent, for instance, boundary conditions, phase equilibria, connectivity between units and various process performance

[*]Submitted for review

[†]DAE Tools Project, Belgrade, Serbia(dnikolic@daetools.com, http://daetools.sourceforge.io)

indicators. The auxiliary equations are typically evaluated on general purpose processors (in general, they cannot be evaluated on streaming processors/accelerators). A detailed discussion of capabilities and limitations of the available approaches for model specification and development of large-scale simulation programs are given in [20, 21, 23].

Numerous software packages and libraries implement various methods for specification of model equations that can be evaluated on multiple computing devices. Libraries for Finite Element Analysis (FEA) and Computational Fluid Dynamics (CFD) such as deal.II [6], libMesh [15], and OpenFOAM [34], and Computer Aided Engineering (CAE) software for Finite Element Analysis and Computational Fluid Dynamics such as HyperWorks [4], STAR-CCM+ and STAR-CD [31], COMSOL Multiphysics [9], ANSYS Fluent/CFX [5] and Abaqus [10] employ various, mostly proprietary, methods for generation and evaluation of kernels on different compute devices. Maxeler Technologies [17] utilise Java language based framework for generation of kernels for Data Flow Engines. In addition, there is a large number of software projects dealing with code generation and transformation of model equations into a code suitable for parallel evaluation on different HPC platforms/architectures. OpenFPM is a scalable open framework for particle and particle-mesh codes on parallel computers [13]. The same authors introduce a C++ expression system to implement numerical simulations of continuum biological hydrodynamics [32]. The expression system allows specification of partial differential equations in near-mathematical notation. Simulations can be parallelised on multi-processor computer systems. Devito [16] is a framework capable of generating highly-optimized code given symbolic equations expressed in Python, specialized in, but not limited to, affine (stencil) codes. It automates stencil computations from a high-level mathematical syntax. A high-performance vendor-agnostic method for massively parallel solving of ensembles of ordinary differential equations and stochastic differential equations (SDEs) on GPUs was presented in [35]. The method is integrated with a widely used differential equation solver library in a high-level language (Julia's DifferentialEquations.jl) and enables GPU acceleration and automatically generates optimized GPU kernels. Bempp-cl [7] is an open-source boundary element method (BEM) library that can be used to assemble all the standard integral kernels for Laplace, Helmholtz, modified Helmholtz, and Maxwell problems. It uses PyOpenCL to just-in-time compile its computational kernels on a wide range of CPU and GPU devices and modern architectures. Chaste (Cancer, Heart And Soft Tissue Environment) [18] - is an open source C++ library for the computational simulation of mathematical models developed for physiology and biology. The code provides modules for handling common scientific computing components, such as meshes and solvers for ordinary and partial differential equations (ODEs/PDEs). Exasim [36] is an open-source software for generating high-order discontinuous Galerkin codes to numerically solve parametrised partial differential equations. The software combines high-level and low-level languages to construct parametrised PDE models via Julia, Python or Matlab scripts and produce high-performance C++ codes for solving the PDE models on CPU and NVidia GPU processors with distributed memory. MOD2IR [19] is an open-source code generation pipeline for NMODL (NEURON MODeling Language, a domain specific language for neuroscience). MOD2IR leverages the LLVM toolchain to target multiple CPU and GPU hardware platforms. Generating LLVM IR allows the vector extensions of modern CPU architectures to be targeted directly, producing optimized SIMD code. Nektar++ is an open-source software framework designed to support the development of high-performance scalable solvers for partial differential equations using the spectral/hp element method.

It runs on general purpose processors and utilises MPI for parallelisation. Other approaches include using modern frameworks such as SYCL for solution of systems of partial differential equations. Solving Maxwell's equation by using a Discontinuous Galerkin time-domain solver implemented in C ++ 17 and SYCL was presented in [2]. Implementation of the nodal discontinuous Galerkin time domain method (NDGTD) and SYCL code generation for Multigrid Methods (fast and scalable numerical solvers for partial differential equations) is given in [11]. Finally, there is a number of projects for development of open-source software for solution of partial-diiferential equations that utilise various compute platforms and architectures such as: DPIVSoft-OpenCL [3], MaMICo [14], MOOSE [25], pyBaram [24], life^x [1], and MultiFEBE [8].

The methodology for specification of model equations in the Open Compute Stack (OpenCS) framework is presented in [22]. In the OpenCS framework, equations can be evaluated using the Compute Stack Machine on all types of compute devices. However, the performance is still an issue since the equations are evaluated using the byte-code instructions. Compared to the models implemented in C/C++ and compiled into the machine code, the compute Stack Machine performance is lower. Therefore, the evaluation performance needs to be further improved. In this work, the existing methodology available in the Open Compute Stack framework [22] has been extended using the concept of kernel equations. This way, models can contain a set of coupled kernels and auxiliary equations that can be evaluated independently. The proposed methodology consists of the following components: (a) an API for specification of kernel equations, (b) a method for generation of the source code for kernels in multiple languages targeting different APIs/frameworks, and (c) a method for evaluation of coupled kernels and auxiliary equations on multiple computing devices/architectures.

An API for specification of kernel equations is very simple and it is straightforward to transform existing models. Kernel equations can generate the source code for compute kernel functions in: (a) C/C++ for compilation into a shared library, and (b) OpenCL C/C++ for streaming processors/accelerators. The information about kernel equations are stored into the Compute Stack while the source code is saved in the file system. Since only variable indexes are stored the size of the resulting Compute Stack is significantly lower.

Kernel equations can be evaluated in a very efficient way using generated kernel functions on most existing computing platforms including general purpose processors, GPUs, FPGAs and other accelerators. As a result, performance of evaluation of model equations is significantly improved (approximately an order of magnitude faster than the Compute Stack Machine). The OpenCS framework can utilise multiple streaming processors/accelerators. Each device can evaluate one or more kernels (the list of kernels are specified as an input parameter). Kernel equations are evaluated using the Compute Stack evaluators that execute compute kernel functions from the generated code. At the moment two kernel evaluators are available: (1) the generated kernels compiled into a shared library are evaluated in parallel using the OpenMP framework, and (2) the generated OpenCL kernels are executed by the OpenCL kernel evaluator.

The article is organised in the following way. First, the required data structures, an API and implementation details are presented. Then, the proposed methodology is applied to two large-scale problems. The simulation results, an overall performance and performance of individual phases are analysed and discussed in details. Finally, a summary of the most important capabilities of the methodology and directions for future work are given in the last section.

2. Methods. The kernel equations concept is implemented as a part of the OpenCS framework [22] and based on the previously developed methodology for parallel evaluation of general systems of differential and algebraic equations on shared and distributed memory systems and heterogeneous setups presented in [21, 23]. In the OpenCS framework, in the current approach, model equations are specified in a symbolic form and transformed using the operator overloading technique into the byte-code instructions. Byte-code instructions are then stored as an array of binary data (a Compute Stack) for direct evaluation on all platforms with no additional processing nor compilation steps. A limited set of byte-code instructions is utilised (only memory access to supplied data arrays and unary and binary mathematical operations). Individual equations (Compute Stacks) are evaluated by a stack machine (Compute Stack Machine) using the Last In First Out (LIFO) queue. Systems of equations are evaluated in parallel using a Compute Stack Evaluator interface which manages the Compute Stack Machine kernels. Two APIs/frameworks are used for parallelism: (a) the Open Multi-Processing (OpenMP) API for parallelisation on general purpose processors (multi-core CPUs), and (b) the Open Computing Language (OpenCL) framework for parallelisation on streaming processors (GPU) and heterogeneous systems (CPU+GPU). Switching to a different computing device for evaluation of equations is straightforward and controlled by an input parameter.

In general, models can contain a coupled set of kernel equations and auxiliary algebraic and ordinary differential equations. In the new approach, all equations must belong to one of equation groups or kernels. Auxiliary equations must be assigned to a single or multiple groups of equations. Kernels by default represent a group of equations. Compute Stack Evaluators now operate on groups of equations and kernels - not the plain range of equations. Kernels can generate C/C++ source code targeting different APIs/frameworks and hardware/accelerators.

The following data structures were introduced to support the concept of kernel equations: `csGroup_t`, `csKernel_t`, `csEquation_t` and `csKernelGenerator_t`. `csGroup_t` data structure contains an ID and the name of the group. `csKernel_t` data structure inherits `csGroup_t` and in addition contains an array of equations, a map of variable indexes in each equation, number of equations and produces a list of source code generators for each supported language/API. `csEquation_t` data structure contains a reference to a group (or kernel), the mathematical expression of the equation and the target variable index. `csKernelGenerator_t` interface is utilised for generation (and automatic compilation, if required) of the kernel source code.

OpenCS kernels during the equations creation phase analyse the supplied equations, gather required information and generate: (a) the Compute Stack byte-code instructions with variable indexes for evaluation using the generated kernel functions, and (b) the C/C++ source code for evaluation functions. For each equation, three separate arrays of indexes are created for variables, time derivatives and degrees of freedom appearing in the expression (degrees of freedom represent system variables that may vary independently, i.e. they are set at the beginning and may be changed during simulation). The arrays are populated with indexes in the order of appearance in the mathematical expression. This way, the mapping between an index in the array (the local index) and the overall index of the model variable is established. Variable indexes are stored in the Compute Stack as ordinary byte-code instructions. This method for storing variable indexes of kernel equations allows for partitioning of the system of equations for simulation on distributed memory systems using the Message Passing Interface (MPI).

Kernels source code is generated in the following way. A single expression is

used as a basis for all equations. That expression is transformed into a form which utilises only local indexes. The generated source code for each kernel is located in the directory with the same name as a kernel name. For instance, in the Case Study 1, the kernels are available in the *Brusselator_u* and *Brusselator_v* sub-directories. Each directory contains *kernel.h*, *kernel.cpp* and *kernel_c_interface.h* files for compilation into a shared library. Compilation of shared library kernels is CMake-based and automatically performed by a generator. A set of .cmake files is generated with options for each supported compiler. In addition, the OpenCL source code is generated in the *OpenCL* and *OpenCL_C99* sub-directories with the OpenCL kernels in C++ and C99, respectively. Both C++ and C99 versions are generated since some devices support only OpenCL C specification. Each sub-directory contains *kernels.cl* file with OpenCL kernels and a set of files with the user defined functions, OpenCL compiler and linker options, preprocessor definitions, include and library directories and link libraries. OpenCL kernels are compiled by the OpenCL run-time.

Evaluation of model equations is group/kernel-based and the modified OpenCS interface (`csComputeStackEvaluator_t`) is used [21]. Each evaluator represents a physical compute device and can evaluate a single or multiple groups of equations or kernels. A typical equation-based model and evaluators setup is given in Fig. 1. The model contains groups with boundary conditions, phase equilibria, connectivity equations and process performance, and multiple kernels for mass, heat and momentum balances and multiple scalar/vector transport equations. A small number of auxiliary equations is evaluated on a single processor using the Compute Stack Machine. The kernels are generated for mass, heat and momentum balance equations and additional transport equations. Mass balance kernels (i.e. the total and two components balances) are evaluated using the OpenCL evaluator on a discrete GPU 1. The momentum balance kernels (x, y and z velocity components) are evaluated using the OpenMP evaluator on multiple CPUs. The heat balance and additional two transport equations kernels are evaluated using the OpenCL evaluator on a discrete GPU 2. The goal is to distribute the compute kernels among compute devices so that each device receives the workload proportional to its compute capabilities. Specification of equation evaluators is an input parameter for simulation located in the configuration file (*simulation_options.json*). A sample specification for the typical equation-based model is given in the Listing 1.

An API for specification of model equations using the kernel concept is very simple. In the existing concept, all equations are placed into a single array and added to a model. In the new kernel concept, each equation must be placed into a group of equations or into a kernel. These two concepts are illustrated in Listings 2 and 3. They show excerpts from the Case Study 1 for two approaches, respectively.

To illustrate the concept, a very simple one-dimensional Laplace equation (heat conduction problem) can be used:

$$\frac{dT}{dt} - k\frac{\partial^2 T}{\partial x^2} = 0,\ \forall i \in [1, N_x - 1] \quad (1)$$

where T is the temperature, k is thermal conductivity and N_x is the number of points in the x domain.

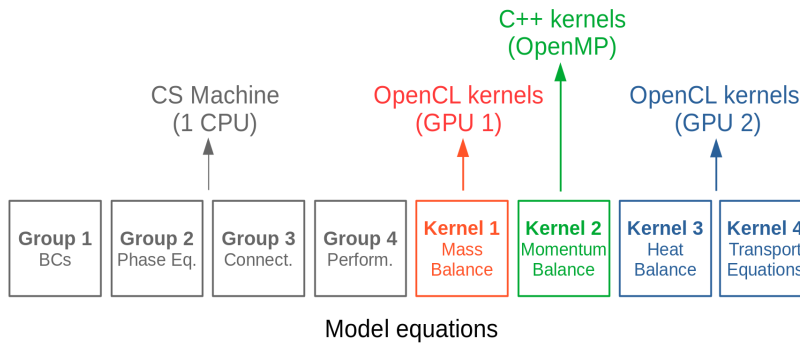

Figure 1: A typical equation-based model and setup of Compute Stack Evaluators. The model contains: (1) multiple kernels for mass (three components), heat and momentum balances (x, y and z components) and two scalar transport equations, and (2) groups of equations for boundary conditions, phase equilibria, connectivity equations and process performance. Groups and kernels are evaluated using different compute devices and the workload is approximately equally distributed.

Listing 1: Evaluator options for a typical equation-based model

```
"Evaluators": {
    "Device_0": {
        "Library": "Sequential",
        "Name": "sequential",
        "Groups": [
            "BoundaryConditions", "PhaseEquilibria",
            "Connectivity",       "ProcessPerformance"
        ]
    },
    "Device_1": {
        "Library": "Kernels_OpenCL",
        "Name": "kOpenCL1",
        "Parameters": {
            "platformID": 0,
            "deviceID": 0
        },
        "Kernels": [
            "MassBalance_C1", "MassBalance_C2", "MassBalance_total"
        ]
    },
    "Device_2": {
        "Library": "Kernels_OpenMP",
        "Name": "kOpenMP",
        "Parameters": {
            "numThreads": 0
        },
        "Kernels": [
            "MomentumBalance_x", "MomentumBalance_y", "MomentumBalance_z"
        ]
    },
    "Device_3": {
        "Library": "Kernels_OpenCL",
        "Name": "kOpenCL2",
        "Parameters": {
            "platformID": 1,
            "deviceID": 1
        },
        "Kernels": [
            "HeatBalance", "TransportEquation_1", "TransportEquation_2"
        ]
    }
}
```

Listing 2: Specification of model equations in the existing concept (Case Study 1)

```
std::vector<csNumber_t> equations;
for(int x = 0; x < Nx; x++)
  for(int y = 0; y < Ny; y++)
    if(x == 0 || x == Nx-1 || y == 0 || y == Ny-1)
    {
      // Boundary points:
      csNumber_t bc = du_dx(x,y) - u_flux_bc;

      equations.push_back(bc);
    }
    else
    {
      // Interior points:
      csNumber_t u_pde = du_dt(x,y)
                       - eps1 * (d2u_dx2(x,y) + d2u_dy2(x,y))
                       - (u(x,y)*u(x,y)*v(x,y) - (B+1)*u(x,y) + A);

      equations.push_back(u_pde);
    }
...
modelBuilder.SetModelEquations(equations);
```

Listing 3: Specification of model equations in the kernel concept (Case Study 1)

```
std::vector<csEquation_t> equations;
std::vector<csKernel_t>   kernels;
csGroup_t  group_BCs("BoundaryConditions", 1);
csKernel_t kernel_u ("Brusselator_u",      2);

for(int x = 0; x < Nx; x++)
  for(int y = 0; y < Ny; y++)
    if(x == 0 || x == Nx-1 || y == 0 || y == Ny-1)
    {
      // Add boundary points to the "BoundaryConditions" group:
      csEquation_t equation(&group_BCs);

      csNumber_t bc = du_dx(x,y) - u_flux_bc;

      equation[ u(x,y) ] = bc;
      equations.push_back(equation);
    }
    else
    {
      // Add interior points to the "Brusselator_u" kernel:
      csEquation_t equation_u(&kernel_u);

      csNumber_t u_pde = du_dt(x,y)
                       - eps1 * (d2u_dx2(x,y) + d2u_dy2(x,y))
                       - (u(x,y)*u(x,y)*v(x,y) - (B+1)*u(x,y) + A);

      equation_u[ u(x,y) ] = u_pde;
      kernel_u.AddEquation(equation_u);
    }

kernels.push_back(kernel_u);
...
modelBuilder.SetModelEquations(equations, kernels);
```

The discretised form of the equation is presented in the Listing 4 (in pseudo-code). The generated kernel is given in the Listing 5, where k and dx are floating point constants (values specified in the simulation) and x_t, x and y are arrays of variable values, time derivatives and degrees of freedom appearing in the equation, respectively. The generated index arrays in this example are [[2, 1, 0], [3, 2, 1], ..., [N_x-1, N_x-2, N_x-3]] for variables and [[1], [2], ..., [$N_x - 1$]] for time derivatives. During evaluation, the values of variables, time derivatives and degrees of freedom are obtained from the solver data arrays using the stored overall indexes.

Listing 4: The discretised form of the heat conduction equation (pseudo-code)

```
for x in range(1, Nx-1):
  dT_dt[x] + k * (T[x+1] - 2*T[x] + T[x-1]) / (dx*dx) = 0
```

Listing 5: Generated C++ kernel functions for the heat conduction equation

```
double residual_Conduction(double time, double* x_t, double* x, double* y)
{
    return x_t[0] - k * (x[0] - 2 * x[1] + x[2]) / (dx*dx);
}

adouble deriv_Conduction(double time, adouble* x_t, adouble* x, adouble* y)
{
    return x_t[0] - k * (x[0] - 2 * x[1] + x[2]) / (dx*dx);
}
```

3. Case Studies. The performance of the kernels approach has been evaluated by benchmarking two large-scale models: (Case 1) a process of auto-catalytic chemical reaction with oscillations known as the Brusselator PDE, and (Case 2) the Chapman mechanism for ozone kinetics arising in atmospheric simulations.

3.1. Case 1: transient two-dimensional diffusion-reaction equations, structured grid. The model describes the process of auto-catalytic chemical reaction with oscillations known as the Brusselator PDE. The net reaction is $A + B \rightarrow D + E$ with transient appearance of intermediates X and Y, where A and B are reactants and D and E are products [33]. The model is originally implemented using SUNDIALS IDAS suite [30]. Under conditions where components A and B are in vast excess during the chemical reaction the system dynamics is described by the following equations (DAE system):

$$\begin{aligned} \frac{du}{dt} - k_1 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - R_u(u, v, t) = 0 \\ \frac{dv}{dt} - k_2 \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - R_v(u, v, t) = 0 \end{aligned} \tag{2}$$

where the reaction rates R_u and R_v are defined as:

$$\begin{aligned} R_u(u, v, t) &= u^2 v - (B + 1)u + A \\ R_v(u, v, t) &= -u^2 v + Bu \end{aligned} \tag{3}$$

Here, k_1 and k_2 are diffusion constants, A and B are the concentrations of components A and B, and u and v are concentration of intermediaries X and Y. The equations are distributed on a square domain $x \in [0, 10]$ and $y \in [0, 10]$ and discretised by central differences on a uniform 1000x1000 spatial mesh resulting in 2,000,000 unknowns. The boundary conditions are homogeneous Neumann (no normal flux at boundaries). The initial conditions are given by: $u(x, y, t_0) = 1.0 - 0.5 \cos(\pi y)$ and $v(x, y, t_0) = 3.5 - 2.5 \cos(\pi x)$. The concentrations of components A and B and the diffusion constants are held constant ($A = 1$, $B = 3.4$, $k_1 = k_2 = 0.002$). The relative and absolute tolerances for all unknowns are set to 10^{-5}. The system is simulated for 10 seconds and the outputs are taken every 0.1 second. The C++ source code and the compilation and usage instructions are given in the Supplemental Listing SM1.

The DAE system is integrated in time using the variable-step variable-order backward differentiation formula from SUNDIALS IDAS solver [12]. Systems of linear equations are solved using the SUNDIALS GMRES solver and the IFPACK ILU [26] preconditioner from Trilinos suite (in the original IDAS model the band-block-diagonal preconditioner has been applied). The input parameters for the IFPACK preconditioner are $k = 3$, $\rho = 1.0$, $\alpha = 0.1$ and $\omega = 0.5$ where k is the fill-in factor, ρ is the relative threshold, α is the absolute threshold and ω is the relax value.

3.2. Case 2: transient two-dimensional convection-diffusion-reaction equations, structured grid. The model describes the Chapman mechanism for ozone kinetics arising in atmospheric simulations [27]. The reaction involves three components: ozone singlet (O), ozone (O_3) and oxygen (O_2), where the first two reactions are photo-chemical and contain diurnal rate coefficients. The model is originally presented in [37] and implemented using SUNDIALS CVodes suite [29]. The system dynamics is described by the following equations (ODE system):

$$\frac{dc_i}{dt} = K_h \frac{\partial^2 c_i}{\partial x^2} + \frac{\partial}{\partial y}\left(K_v(y)\frac{\partial c_i}{\partial y}\right) + V\frac{\partial c_i}{\partial x} + R_i(c_1, c_2, t),\ \forall i \in \{1, 2\} \tag{4}$$

where the reaction rates R_1 and R_2 are given by:

$$\begin{aligned} R_1(c_1, c_2, t) &= -q_1 c_1 c_3 - q_2 c_1 c_2 + 2q_3(t)c_3 + q_4(t)c_2 \\ R_2(c_1, c_2, t) &= q_1 c_1 c_3 - q_2 c_1 c_2 - q_4(t)c_2 \end{aligned} \tag{5}$$

Here, c_1, c_2 and c_3 are concentrations of O, O_3 and O_2, respectively, q_1, q_2, q_3 and q_4 are reaction rate coefficients, V is velocity, and K_h and K_v are diffusion coefficients. The numerical values of the input parameters are: $V = 10^{-3}$, $K_h = 4 \cdot 10^{-6}$ and $K_v(y) = 10^{-8}\exp(0.2y)$. $K_v(y)$ is a function of the position (y coordinate) and a special care must be taken when generating the source code for kernels. q_1, q_2 and c_3 are constant ($q_1 = 1.63 \cdot 10^{-16}$, $q_2 = 4.66 \cdot 10^{-16}$, $c_3 = 3.7 \cdot 10^{16}$) while q_3 and q_4 vary diurnally:

$$\begin{aligned} q_3(t) &= \begin{cases} \exp\left(-A_3/\sin(\omega t)\right), & \text{if} \sin(\omega t) > 0 \\ 0, & \text{otherwise} \end{cases} \\ q_4(t) &= \begin{cases} \exp\left(-A_4/\sin(\omega t)\right), & \text{if} \sin(\omega t) > 0 \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{6}$$

where $\omega = \pi/43200$ and A_3 and A_4 are coefficients ($A_3 = 22.62$, $A_4 = 7.601$). The equations are distributed on a square domain: $x \in [0, 20]$ km and $y \in [30, 50]$ km and discretised by central differences on a uniform 1000x500 spatial mesh resulting in 1,000,000 unknowns. In the original CVodes model the equations were also discretised using the central differences, except for the advection term where a biased 3-point difference formula was used. The boundary conditions are homogeneous Neumann (no normal flux at boundaries). The initial conditions are given by:

$$\begin{aligned} c_1(x, y, t_0) &= 10^6\alpha(x)\beta(y) \\ c_2(x, y, t_0) &= 10^{12}\alpha(x)\beta(y) \\ \alpha(x) &= 1 - (0.1(x - x_{mid}))^2 + 0.5(0.1(x - x_{mid}))^4 \\ \beta(y) &= 1 - (0.1(y - y_{mid}))^2 + 0.5(0.1(y - y_{mid}))^4 \end{aligned} \tag{7}$$

where $x_{mid} = (0+20)/2 = 10$ and $y_{mid} = (30+50)/2 = 40$ are mid points of the x, y domains. The relative and absolute tolerances for all unknowns are set to 10^{-5}. The system is integrated for 86,400 seconds (1 day) and the outputs are taken every 360 seconds. The C++ source code and the compilation and usage instructions are given in the Supplemental Listing SM2.

The ODE system is integrated in time using the variable-step variable-order backward differentiation formula available in SUNDIALS CVodes solver [28]. Systems of linear equations are solved using the SUNDIALS generalised minimal residual solver and the IFPACK ILU [26] preconditioner from Trilinos suite (in the original CVodes model the 2x2 block-diagonal preconditioner has been applied). The input parameters for the IFPACK preconditioner for all runs are $k = 1$, $\rho = 1.0$, $\alpha = 10^{-5}$ and $\omega = 0.0$.

3.3. Run-time setup. The model in Case 1 (DAE system) contains one group with boundary condition equations ("BoundaryConditions") and two kernels of equal size with equations for interior points of transport equations for u and v reaction components ("Brusselator_u" and "Brusselator_v"). The model in Case 2 (ODE system) contains one group with boundary condition equations ("BoundaryConditions") and two kernels of equal size with equations for interior points of transport equations for C_1 and C_2 reaction components ("DiurnalKinetics_C1" and "DiurnalKinetics_C2"). The generated source code for all kernels is given in the Supplemental Listing SM3, located in the directories with the same name as kernels. Four different runs were performed for each case study:

Cx-CSM-SEQ Compute Stack Machine implementation
(evaluation on a single processor)

Cx-CSM-OMP Compute Stack Machine implementation
(evaluation using the OpenMP framework)

Cx-SEQ Kernels implementation
(the group evaluated on a single processor,
kernels evaluated *one by one* on another processor)

Cx-OMP Kernels implementation
(the group evaluated on a single processor,
kernels evaluated *one by one* using OpenMP)

where Cx is C1 for Case Study 1 and C2 for Case Study 2. Simulation runs for Case Studies 1 and 2 are given in Table 1:

Table 1: Simulation runs for Cases 1 and 2

Run	Simulation	Evaluation of model equations
C1-CSM-SEQ	1 CPU	CS Machine, Sequential
C1-CSM-OMP	16 CPUs	CS Machine, OpenMP
C1-SEQ	1 CPU	Kernels, Sequential
C1-OMP	16 CPUs	Kernels, OpenMP
C2-CSM-SEQ	1 CPU	CS Machine, Sequential
C2-CSM-OMP	16 CPUs	CS Machine, OpenMP
C2-SEQ	1 CPU	Kernels, Sequential
C2-OMP	16 CPUs	Kernels, OpenMP

In addition, the performance of OpenCS kernels were compared to tailor-made C kernel implementations for both case studies (single processor runs only). The source code of the C implementations are adapted from the original *idasBruss_kry_bbd_p.c*

IDAS example implementation [12] and *cvsDiurnal_kry.c* CVodes example implementation [28] and given in Supplemental Listing SM4. The simulation setup, compilation and run options are given in the Supplementary Materials SM1 and SM2.

The simulations were carried out in 64-bit Windows 11, compiled using the vc++ 19.34 compiler and run using OpenCS 2.1.0. The hardware configuration consists of the AMD Ryzen CPU with 8 cores/16 threads at 4GHz and 16 GB of RAM.

4. Results. The number of equations (N_{eq}), the number of non-zero items in the Jacobian matrix (the total number $N_{nz} = \sum_{i=1}^{N_{eq}} N_{nz}[i]$), the number of Compute Stack items (the total number $N_{cs} = \sum_{i=1}^{N_{eq}} N_{cs}[i]$ and the average number per equation $N_{cs/equation}$) and the average number of Compute Stack items for evaluation of a single row of the Jacobian matrix ($N_{cs/jacob_row} = \frac{1}{N_{eq}} \sum_{i=1}^{N_{eq}} N_{nz}[i] N_{cs}[i]$) for the sequential runs in both cases are given in Table 2. Since only variable indexes are stored in the Compute Stack, its size in kernel versions is significantly lower: 3.76 times in Case 1 and 8.42 times in Case 2.

Four main phases of the numerical solution are analysed:

1. *EvalEqns* – evaluation of equations (residuals or right-hand side).
2. *EvalDerivs* – evaluation of derivatives (for a preconditioner).
3. *LinSysSetup* – evaluation of derivatives + compute preconditioner
4. *LinSysSolve* – apply preconditioner
5. *Integration* – integration of the system of equations in time.

In SUNDIALS the difference quotient approximation is used in the *LinSysSolve* phase and requires an additional call to the *EvalEqns* function.

The total integration time, the duration of individual phases of the numerical solution and the number of function calls of each phase are presented in Table 3 for Case 1 and Table 6 for Case 2. The duration of individual phases given as a percentage of the duration of the total integration time are presented in Table 4 for Case 1 and Table 7 for Case 2. The speed-ups of individual phases of the numerical solution, the maximum theoretical overall speed-ups and the achieved overall simulation speed-ups are given in Table 5 for Case 1 and Table 8 for Case 2. The number of linear solver iterations and the number of evaluate function calls is different for Compute Stack Machine and Kernel versions since the equations are differently arranged in kernel versions. The equations are grouped and sorted by a group ID producing different linear systems. However, the numerical results are identical.

The maximum theoretical speed-up for evaluation of model equations is 16 for both cases (the number of cores). The maximum theoretical overall simulation speed-ups can be calculated from the Amdahl's law using the data from Tables 3 and 6: $1/\left(1 - p + p/s\right)$, where p is the portion of the solution that can be parallelised and s is the maximum theoretical speed-up. The sequential runs C1-CSM-SEQ and C2-CSM-SEQ are used as a basis. In both cases only the evaluation of model equations and derivatives can be performed in parallel. Combined, they amount to 68.5% (Case 1) and 89.1% (Case 2) of the total integration time in sequential runs. Hence, the maximum theoretical overall speed-ups are: 2.79 for Case 1 and 6.07 for Case 2.

The simulation results for the comparison of OpenCS kernels with the tailor-made C implementations for both case studies (single processor runs only) are given in Tables SM1 and SM2 in the Supplementary Materials, respectively.

Table 2: Workload-related properties for Case 1 and Case 2.

	N_{eq}	N_{nz}	N_{cs}	$N_{cs/equation}$	$N_{cs/jacob_row}$
C1-CSM	2,000,000	11,976,024	67,832,168	33.92	203.09
C1	2,000,000	11,976,024	18,039,960	9.02	54.01
C2-CSM	1,000,000	5,988,000	69,928,000	69.93	418.73
C2	1,000,000	5,988,000	8,305,496	8.31	58.03

Table 3: Case 1: duration of individual phases in seconds and the number of function calls (given in the brackets).

Phase	C1-CSM-SEQ	C1-CSM-OMP	C1-SEQ	C1-OMP
EvalEqns	710.20 (2449)	145.74 (2449)	44.14 (2446)	29.91 (2446)
EvalDerivs	49.77 (19)	13.60 (19)	16.34 (19)	7.56 (19)
LinSysSetup	89.79 (19)	52.50 (19)	51.59 (19)	42.39 (19)
LinSysSolve	647.03 (973)	307.96 (973)	267.45 (973)	250.09 (973)
Integration	1093.60 (973)	497.89 (973)	414.96 (973)	386.09 (973)

Table 4: Case 1: duration of individual phases in percents.

Phase	C1-CSM-SEQ	C1-CSM-OMP	C1-SEQ	C1-OMP
EvalEqns	64.00	28.76	10.41	7.59
EvalDerivs	4.49	2.68	3.85	1.92
LinSysSetup	8.09	10.36	12.17	10.75
LinSysSolve	58.31	60.78	63.08	63.43

Table 5: Case 1: Speed-ups for individual phases and the achieved overall speed-up.

Phase	C1-CSM-SEQ	C1-CSM-OMP	C1-SEQ	C1-OMP
EvalEqns	1.00	4.87	16.07	23.72
EvalDerivs	1.00	3.66	3.05	6.59
LinSysSetup	1.00	1.71	1.74	2.12
LinSysSolve	1.00	2.10	2.42	2.59
Overall (achieved)	1.00	2.21	2.64	2.83

Table 6: Case 2: duration of individual phases in seconds and the number of function calls (given in the brackets).

Phase	C2-CSM-SEQ	C2-CSM-OMP	C2-SEQ	C2-OMP
EvalEqns	1661.35 (4046)	286.33 (4046)	264.95 (3783)	54.12 (3783)
EvalDerivs	339.29 (135)	78.30 (135)	105.37 (115)	36.15 (115)
LinSysSetup	388.42 (135)	128.18 (135)	158.97 (115)	90.27 (115)
LinSysSolve	1084.95 (1753)	300.38 (1753)	308.28 (1432)	168.55 (1432)
Integration	2244.07 (1756)	612.31 (1756)	619.22 (1435)	328.94 (1435)

Table 7: Case 2: duration of individual phases in percents.

Phase	C2-CSM-SEQ	C2-CSM-OMP	C2-SEQ	C2-OMP
EvalEqns	73.98	46.64	42.67	16.37
EvalDerivs	15.11	12.75	16.97	10.93
LinSysSetup	17.30	20.88	25.61	27.30
LinSysSolve	48.31	48.93	49.65	50.98

Table 8: Case 2: Speed-ups for individual phases and the achieved overall speed-up.

Phase	C2-CSM-SEQ	C2-CSM-OMP	C2-SEQ	C2-OMP
EvalEqns	1.00	5.80	5.86	28.70
EvalDerivs	1.00	4.33	2.74	7.99
LinSysSetup	1.00	3.03	2.08	3.67
LinSysSolve	1.00	3.61	2.87	5.26
Overall (achieved)	1.00	3.66	2.96	5.58

5. Discussion. For sequential runs, the observed speed-ups in evaluation of model equations in kernel versions in the *EvalEqns* phase are 16.07 and 5.86 for Case 1 and Case 2, respectively. On the other hand, the observed speed-ups in evaluation of model equations in kernel versions in the *EvalDerivs* phase are 3.05 and 2.74 for Case 1 and Case 2, respectively. Therefore, the compiled C++ kernels are approximately an order of magnitude faster for evaluation of model equations and three times faster for evaluation of derivatives.

Regarding the benchmarks with the tailor-made C implementation for single processor runs, the performance is significantly higher in the C implementations. In average, in Case 1, Brusselator u and v kernels are 3.0 and 3.7 times faster, and in Case 2, DiurnalKinetics C1 and C2 kernels are 26.6 and 32.6 times faster in the C implementation, respectively. In both cases the better performance is due to the hand-made code optimisations (in particular for Case 2). Also automatic compiler code optimisations can be more aggressive and more effective since the loop counts are known in advance. In both cases, the code is auto-vectorised by the compiler, too. The full analysis for both case studies are given in the Supplementary Materials SM1 and SM2.

In parallel OpenMP runs, the observed speed-ups in evaluation of model equations using Compute Stack Machine are as follows: 4.87 in Case 1 and 5.80 in Case 2 (the *EvalEqns* phase), and 3.66 in Case 1 and 4.33 in Case 2 (the *EvalDerivs* phase). Speed-ups in kernel versions are: 23.72 in Case 1 and 28.70 in Case 2 (the *EvalEqns* phase), and 6.59 in Case 1 and 7.99 in Case 2 (the *EvalDerivs* phase). Theoretically, since there are no dependency nor data exchange between processing elements, evaluation of equations should scale linearly with the increase in the number of threads (16 in this work). The observed speed-ups are somewhat lower than expected.

The achieved overall simulation speed-ups in Case 1 are: 2.21, 2.64 and 2.83, and in Case 2: 3.66, 2.96 and 5.58 for runs CSM-OMP, SEQ and OMP, respectively. The maximum theoretical overall simulation speed-ups are 2.79 for Case 1 and 6.07 for Case 2. The achieved overall speed-ups are approaching the maximum theoretical ones for OpenMP kernel versions.

6. Conclusions. A methodology for efficient evaluation of model equations in equation-based simulation programs has been presented in this work. The main idea and implementation details of the kernel equations concept have been described. Kernel equations represent a homogeneous group of identical expressions operating on different variable values and can be efficiently evaluated on both general purpose processors and streaming processors / accelerators. The methodology has been implemented as a part of the Open Compute Stack (OpenCS) framework and provides: (a) an API for specification of kernel equations, (b) a method for generation of the source code for kernels in multiple languages targeting different APIs/frameworks, and (c) a method for evaluation of coupled kernels and auxiliary equations on multiple computing devices / architectures. Kernel equations can generate C/C++ source code for compilation into a shared library, and OpenCL C/C++ source code for streaming processors / accelerators. The code is automatically compiled by the framework. To perform simulation, the OpenCS framework can utilise multiple compute devices where each device evaluates one or more kernels. Setup of compute devices to be used for simulation is an input parameter.

The capabilities of the framework have been illustrated using two large scale problems. The concept of equation kernels offers a significant improvement in the simulation performance. The compiled C++ kernels are an order of magnitude faster for evaluation of model equations ($6 - 16$ times) and three times faster for evaluation of derivatives compared to the Compute Stack Machine. The achieved overall speedups are approaching the maximum theoretical ones for parallel kernel versions using the OpenMP framework.

Performance of OpenCS C++ kernels have been compared to the tailor-made C implementations. The tailor-made models provide significant performance gains (more than an order of magnitude in some cases) with some limitations such as difficulties to obtain analytical derivatives, a difficult and error-prone procedure and simulations limited to general-purpose processors only.

The future work will focus on further improvements of the performance (i.e. using vector extensions such as AVX512 and SVE2), reduction of memory requirements, support for additional architectures used in HPC (such as ARM64), and kernel generators and Compute Stack Evaluator implementations for additional types of computing devices (such as FPGA and other accelerators).

REFERENCES

[1] P. C. Africa, $life^x$, 2022, https://lifex.gitlab.io/lifex.
[2] A. Afzal, C. Schmitt, S. Alhaddad, Y. Grynko, J. Teich, J. Forstner, and F. Hannig, *Solving Maxwell's Equations with Modern C++ and SYCL: A Case Study*, in 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2018, pp. 1–8, https://doi.org/10.1109/ASAP.2018.8445127.
[3] J. Aguilar-Cabello, L. Parras, and C. del Pino, *DPIVSoft-OpenCL*, 2022, https://gitlab.com/jacabello/dpivsoft_python.
[4] Altair, *HyperWorks*, 2018, https://altairhyperworks.com.
[5] Ansys, Inc., *ANSYS Fluent*, 2018, http://www.ansys.com.
[6] W. Bangerth, R. Hartmann, and G. Kanschat, *deal.II – a general purpose object oriented finite element library*, ACM Trans. Math. Softw., 33 (2007), pp. 24/1–24/27.
[7] T. Betcke and M. W. Scroggs, *Bempp-cl: A fast python based just-in-time compiling boundary element library.*, Journal of Open Source Software, 6 (2021), p. 2879, https://doi.org/10.21105/joss.02879.
[8] J. D. Bordón, G. M. Álamo, L. A. Padrón, J. J. Aznárez, and O. Maeso, *MultiFEBE, url = https://github.com/mmc-siani-es/MultiFEBE, version = 2.0.0, year = 2022.*
[9] COMSOL, Inc., *COMSOL Multiphysics*, 2018, http://www.comsol.com.

[10] Dassault Systemes, *Abaqus*, 2018, http://www.simulia.com.
[11] S. Groth, C. Schmitt, J. Teich, and F. Hannig, *SYCL Code Generation for Multigrid Methods*, in Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems, SCOPES '19, New York, NY, USA, 2019, Association for Computing Machinery, p. 41–44, https://doi.org/10.1145/3323439.3323984.
[12] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, *SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers*, ACM Trans. Math. Softw., 31 (2005), pp. 363–396, https://doi.org/10.1145/1089014.1089020.
[13] P. Incardona, A. Leo, Y. Zaluzhnyi, R. Ramaswamy, and I. F. Sbalzarini, *OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers*, Computer Physics Communications, 241 (2019), pp. 155–177, https://doi.org/10.1016/j.cpc.2019.03.007.
[14] P. Jarmatz, H. Wittenberg, V. Jafari, A. D. Sharma, F. Maurer, N. Wittmer, and P. Neumann, *MaMiCo*, 2022, https://hsu-hpc.github.io/MaMiCo.
[15] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey, *libMesh: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations*, Engineering with Computers, 22 (2006), pp. 237–254, https://doi.org/10.1007/s00366-006-0049-3.
[16] F. Luporini, M. Louboutin, M. Lange, N. Kukreja, P. Witte, J. Hückelheim, C. Yount, P. H. J. Kelly, F. J. Herrmann, and G. J. Gorman, *Architecture and performance of devito, a system for automated stencil computation*, ACM Trans. Math. Softw., 46 (2020), https://doi.org/10.1145/3374916.
[17] Maxeler Technologies, Inc., *Data Flow Engines*, 2023, https://www.maxeler.com.
[18] G. R. Mirams, C. J. Arthurs, M. O. Bernabeu, R. Bordas, J. Cooper, A. Corrias, Y. Davit, S.-J. Dunn, A. G. Fletcher, D. G. Harvey, M. E. Marsh, J. M. Osborne, P. Pathmanathan, J. Pitt-Francis, J. Southern, N. Zemzemi, and D. J. Gavaghan, *Chaste: An open source c++ library for computational physiology and biology*, PLOS Computational Biology, 9 (2013), pp. 1–8, https://doi.org/10.1371/journal.pcbi.1002970.
[19] G. Mitenkov, I. Magkanaris, O. Awile, P. Kumbhar, F. Schürmann, and A. F. Donaldson, *MOD2IR: High-performance code generation for a biophysically detailed neuronal simulation dsl*, in Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, CC 2023, New York, NY, USA, 2023, Association for Computing Machinery, p. 203–215, https://doi.org/10.1145/3578360.3580268.
[20] D. D. Nikolić, *DAE Tools: Equation-based object-oriented modelling, simulation and optimisation software*, PeerJ Computer Science, 2 (2016), p. e54, https://doi.org/10.7717/peerj-cs.54.
[21] D. D. Nikolić, *Parallelisation of equation-based simulation programs on heterogeneous computing systems*, PeerJ Computer Science, 4 (2018), p. e160, https://doi.org/10.7717/peerj-cs.160.
[22] D. D. Nikolić, *OpenCS: a framework for parallelisation of equation-based simulation programs*, PeerJ Computer Science, (2023), https://doi.org/submitted-for-review, https://daetools.sourceforge.io/docs/parallelisation-opencs-preprint.pdf.
[23] D. D. Nikolić, *Parallelisation of equation-based simulation programs on distributed memory systems*, SIAM J. Sci. Comput., (2023), https://doi.org/submitted-for-review, https://daetools.sourceforge.io/docs/parallelisation-dsm-preprint.pdf.
[24] J. S. Park, *pyBaram*, 2022, https://aadl_inha.gitlab.io/pyBaram.
[25] D. R. Permann, Cody J. Gaston, D. Andrš, R. W. Carlsen, F. Kong, A. D. Lindsay, J. M. Miller, J. W. Peterson, A. E. Slaughter, R. H. Stogner, and R. C. Martineau, *MOOSE*, 2022, https://mooseframework.inl.gov.
[26] M. Sala and M. Heroux, *Robust algebraic preconditioners with IFPACK 3.0*, Tech. Report SAND-0662, Sandia National Laboratories, 2005.
[27] W. E. Schiesser and L. Lapidus, Academic Press, Inc, New York, United States, 1976.
[28] R. Serban and A. C. Hindmarsh, *CVODES, the sensitivity-enabled ODE solver in SUNDIALS*, Proceedings of the 5th International Conference on Multibody Systems, Nonlinear Dynamics and Control, Long Beach, CA, USA, 2005.
[29] R. Serban and A. C. Hindmarsh, *Example programs for CVODES v2.8.2*, Tech. Report UCRL-SM-208115, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2015, https://computation.llnl.gov/sites/default/files/public/cvs_examples.pdf.
[30] R. Serban and A. C. Hindmarsh, *Example programs for IDAS v1.3.0*, Tech. Report LLNL-TR-437091, Center for Applied Scientific Computing, Lawrence Livermore National Labo-

ratory, 2016, https://computation.llnl.gov/sites/default/files/public/idas_examples.pdf.
[31] Siemens, *Multidisciplinary Design Exploration*, 2018, https://mdx.plm.automation.siemens.com.
[32] A. Singh, P. Incardona, and I. F. Sbalzarini, *A c++ expression system for partial differential equations enables generic simulations of biological hydrodynamics*, The European Physical Journal E, 44 (2021), p. 117, https://doi.org/10.1140/epje/s10189-021-00121-x.
[33] S. H. Strogatz, CRC Press LLC, Boca Raton, Florida, USA, 1994.
[34] The OpenFOAM Foundation, *OpenFOAM*, 2018, http://www.openfoam.org.
[35] U. Utkarsh, V. Churavy, Y. Ma, T. Besard, T. Gymnich, A. Gerlach, A. Edelman, and C. Rackauckas, *Automated translation and accelerated solving of differential equations on multiple gpu platforms*, (2023), https://www.researchgate.net/publication/370058411_Automated_Translation_and_Accelerated_Solving_of_Differential_Equations_on_Multiple_GPU_Platforms.
[36] J. Vila-Pérez, R. L. Van Heyningen, N.-C. Nguyen, and J. Peraire, *Exasim: Generating discontinuous galerkin codes for numerical solutions of partial differential equations on graphics processors*, SoftwareX, 20 (2022), p. 101212, https://doi.org/10.1016/j.softx.2022.101212.
[37] M. R. Wittman, *Testing of PVODE, a parallel ode solver*, Tech. Report UCRL-ID-125562, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 1996.