Parallelisation of equation-based simulation programs on heterogeneous computing systems

Dragan D. Nikolić

DAE Tools Project, Belgrade, Serbia, http://www.daetools.com

Corresponding author:

Dragan D. Nikolić

Email address: dnikolic@daetools.com

ABSTRACT

Numerical solution of equation-based simulations requires computationally intensive tasks such as evaluation of model equations, linear algebra operations and solution of systems of linear equations. The focus in this work is on parallel evaluation of model equations on shared memory systems such as general purpose processors (multi-core CPUs and manycore devices), streaming processors (GPU and FPGA) and heterogeneous systems. The current approaches for evaluation of model equations are reviewed and their capabilities and shortcomings analysed. Since stream computing differs from traditional computing in that the system processes a sequential stream of elements, equations must be transformed into a data structure suitable for both types. The postfix notation expression stacks are recognised as a platform and programming language independent method to describe, store in computer memory and evaluate general systems of differential and algebraic equations of any size. Each mathematical operation and its operands are described by a specially designed data structure, and every equation is transformed into an array of these structures (a Compute Stack). Compute Stacks are evaluated by a stack machine using a LIFO queue. The stack machine is implemented in the DAE Tools modelling software in the C99 language using two APIs/frameworks for parallelism. The OpenMP API is used for parallelisation on general purpose processors, and the OpenCL framework is used for parallelisation on streaming processors and heterogeneous systems. The performance of the sequential Compute Stack approach is compared to the direct C++ implementation and to the previous approach that uses evaluation trees. The new approach is 45% slower than the C++ implementation and more than five times faster than the previous one. The OpenMP and OpenCL implementations are tested on three medium-scale models using a multi-core CPU, a discrete GPU, an integrated GPU and heterogeneous computing setups. Execution times are compared and analysed and the advantages of the OpenCL implementation running on a discrete GPU and heterogeneous systems are discussed. It is found that the evaluation of model equations using the parallel OpenCL implementation running on a discrete GPU is up to twelve times faster than the sequential version while the overall simulation speed-up gained is more than three times.

INTRODUCTION

Most of engineering problems can be described by a system of non-linear (partial-)differential and algebraic equations. One of the methods to solve this type of problems is by using the Equation-

40 Based approach (Barton and Pantelides, 1993). According to this approach, all equations and
41 variables which constitute the model representing the process are generated and gathered together.
42 Then, equations are solved simultaneously using a suitable mathematical algorithm (Morton,
43 2003). In the equation-based approach equations are given in an implicit form as functions of
44 state variables and their derivatives, degrees of freedom (the system variables that may vary
45 independently), and parameters. A typical Equation-Based simulation, optimisation, parameter
46 estimation and sensitivity analysis software requires a numerical integration of a system of
47 differential-algebraic equations (DAE) and integration of sensitivity equations (if local gradient
48 based sensitivities are requested) by a suitable DAE solver. DAE solvers perform the following
49 computationally intensive tasks (Fig. 1): (1) numerical integration in time (requires evaluation
50 of equations residuals), (2) linear algebra operations (mostly BLAS L1 vector operations and
51 some of BLAS L2 matrix-vector operations), (3) solution of systems of linear equations (direct
52 solvers require evaluation of a Jacobian matrix while the iterative ones need a preconditioner),
53 (4) integration of sensitivity equations (local gradient-based sensitivity analysis methods such as
54 the Forward and Adjoint methods require evaluation of sensitivity residuals). Parallelisation of
55 these tasks lead to a faster numerical solution.

56   A large number of state of the art parallel linear algebra libraries are freely available such
57 as AMD CPU Libraries (Advanced Micro Devices, Inc., 2018) and Intel Math Kernel Library
58 (Intel Corporation, 2018a) and several parallel linear solvers such as SuperLU_MT (Li, 2005),
59 PARDISO (Schenk et al., 2007) and Intel Pardiso and Iterative Sparse Solvers (Intel Corporation,
60 2018b). Thus, the focus in this is work is on parallel evaluation of residuals, Jacobian matrix
61 and sensitivity residuals on shared memory computing platforms: (a) general purpose processors
62 (traditional multi-core CPUs and manycore devices i.e. Xeon Phi), (b) streaming processors
63 (General Purpose Graphics Processing Units, GPGPU and Field Programmable Gate Arrays,
64 FPGA), and (c) heterogeneous computing systems (i.e. CPU+GPU and CPU+FPGA).
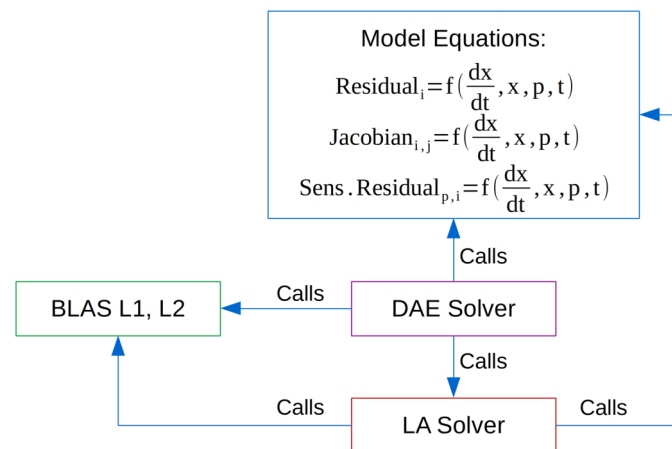
Figure 1. Computationally intensive tasks performed by DAE solvers

65   In general, model equations can be specified using the following approaches (again, only
66 the problems that can be described by a system of partial or ordinary differential and algebraic
67 equations are of interest):

68 1. Using general-purpose programming languages such as C/C++ or Fortran to implement
69   user-supplied functions for one of available suites for scientific applications such as SUN-

DIALS (Hindmarsh et al., 2005), Trilinos (Heroux et al., 2005) and PETSC (Balay et al., 2015)

2. Using modelling languages such as Ascend (Piela et al., 1991), gPROMS (Barton and Pantelides, 1994), Modelica (Fritzson and Engelson, 1998) and Dymola (Elmqvist, 1978)

3. Using multi-paradigm numerical languages such as Matlab (The MathWorks, Inc., 2018), Mathematica (Wolfram Research, Inc., 2015) and Maple (Waterloo Maple, Inc., 2018)

4. Using higher-level fourth-generation languages (Python) and the modelling concepts provided by a software such as DAE Tools (Nikolić, 2016)

5. Using Computer Aided Engineering (CAE) software for finite element analysis and computational fluid dynamics such as COMSOL Multiphysics (COMSOL, Inc., 2018), ANSYS Fluent/CFX (Ansys, Inc., 2018), Abaqus (Dassault Systemes, 2018), deal.II (Bangerth et al., 2007), libMesh (Kirk et al., 2006) and OpenFOAM (The OpenFOAM Foundation, 2018)

The compiled user-supplied functions often produce the fastest computation. However, the process is error prone and it is difficult to specify expressions for analytical derivatives required for Jacobian/preconditioner matrix and sensitivity residuals. The source code of modelling languages is typically parsed into an Abstract Syntax Tree (AST). The produced AST can be transformed into a simulator-specific data structure or used to generate C source code as in OpenModelica (Fritzson et al., 2005) and JModelica (Akesson et al., 2010). Other modelling software such as DAE Tools (Nikolić, 2016) use the operator overloading technique to produce a tree-like data structure called an Evaluation Tree (ET). CAE software perform a discretisation of Partial Differential Equations (PDE) on a specified grid. On unstructured grids, the results of discretisation using the Finite Element (FE) or Finite Volume (FV) methods are the mass and stiffness matrices and load vectors. On structured grids, the results of discretisation using the Finite Difference (FD) method are the stencil data (nodes arrangement and their coefficients). Then, matrix-vector/matrix-matrix operations or stencil codes are used to obtain the results.

Evaluation of user-supplied functions, automatically generated source code and simulator-specific data structures can be easily performed on general purpose processors, either sequentially or in parallel. The parallel evaluation is fairly straightforward: every process/thread operates on its part of the array of equations. However, evaluation on streaming processors is rather difficult. Stream computing differs from traditional computing in that the system processes a sequential stream of elements: a kernel is executed on each element of the input stream and the result stored in an output stream. Hence, applications must be specially designed to effectively take advantage of the technology. For instance, the tree-like data structures are represented by non-contiguous blocks of memory and their evaluation typically requires recursive function calls which are not supported on all types of streaming processors. On the other hand, kernels for matrix-vector and matrix-matrix operations as well as stencil codes are widely available for virtually all API and hardware platforms. However, the general non-linear DAE systems cannot be represented in this way.

The main idea in this work is to implement a method to transform a general set of non-linear differential-algebraic equations into a stream of data that can be processed in parallel on both general purpose and streaming processors. In contrast to problems that can be described by a single system of finite element/finite volume/finite difference equations the focus in this work is on DAE systems that might include mixed/multiple coupled FE/FV/FD equations with additional ordinary differential and algebraic equations. Such mixed non-linear systems of equations are

115 often found in multi-scale models and typically cannot be represented using a single mass/stiffness
116 matrix and load vector nor evaluated using the stencil kernels.

117   The proposed approach is to transform equation expressions from infix notation into a Reverse
118 Polish Notation (postfix notation) expression stack. For instance, the expression in infix notation
119 $x + y$ can be represented in postfix notation as $xy+$ and stored in computer memory as an array
120 of three items: $[x, y, binary\ operator\ +]$. Expression stacks can be evaluated by a stack machine
121 using a LIFO (Last In First Out) queue and in general are available for all platforms (due to their
122 simplicity). Therefore, the postfix notation expression stacks are found as an efficient method to
123 describe, store in computer memory and evaluate general DAE systems on all computing devices.
124 Each mathematical operation and its operands are described by a specially designed data structure.
125 This structure is very compact (to reduce memory transfers) and contains the op code and other
126 data required for evaluation within the DAE Tools context. Every equation is represented by an
127 array of these structures (called a Compute Stack in DAE Tools terminology). Since it is just
128 a stream of binary data it can be read and used from any programming language. Evaluation
129 of equations using the postfix notation expression stacks is obviously slower than compiled C
130 code. However, the memory latency can be hidden in streaming processors by employing a large
131 number of light-weight 'threads' with fast context switching (as shown in the Results section).

132   Implementation of a Compute Stack Machine requires an Application Programming Interface
133 (API) and a runtime library to work on target platforms. There is a large number of available
134 software API, libraries and languages that support shared memory multiprocessing and listing
135 them all is out of scope of this work. Open Multi-Processing (OpenMP, http://www.openmp.org)
136 has been chosen as one of the most widely used API for general purpose processors on shared
137 memory systems. Open Computing Language (OpenCL, https://www.khronos.org/opencl) has
138 been selected for streaming processors since it supports all hardware platforms of interest
139 (CPU, GPGPU, and FPGA) while its main competitor NVidia CUDA is available only on
140 NVidia hardware. Regarding the programming language, OpenMP supports C/C++ and Fortran
141 while OpenCL supports a static subset of C99 and a static subset of C++14 (since version 2.1).
142 Therefore, C99 is the most suitable language for implementation as it is supported by both
143 frameworks and all versions of the standards.

144   An overview of DAE Tools simulation utilising compute stack machines on different comput-
145 ing hardware is shown in Fig. 2. At the moment, simulations are executed on the host computer
146 and the integration in time and solution of linear systems are performed on the CPU. The addi-
147 tional computing devices are used as accelerators for evaluation of model equations. A modelling
148 software generates an array of Compute Stacks (one for every equation). All required data arrays
149 are sent to a Compute Stack Evaluator (a class providing a common interface for evaluation of
150 residuals, Jacobian and sensitivity residuals). The Compute Stack Evaluator implementation
151 launches multiple OpenMP threads or OpenCL kernels, each of them running a Compute Stack
152 Machine on the selected hardware.

153   An implementation of a Compute Stack Machine in C99 using OpenCL framework provides
154 the following functionality/benefits: (a) parallel evaluation of any system of equations of any
155 size, (b) support for all important computing platforms (including heterogeneous systems), (c)
156 superior performance of streaming processors compared to the evaluation on general purpose
157 processors, (d) straightforward scaling up to any number of equations, and (e) straightforward
158 switching to a different computing platform.

159   The article is organised in the following way. First, the required data structures and the
160 OpenMP and OpenCL implementations in C99 are presented. Then, three medium scale models
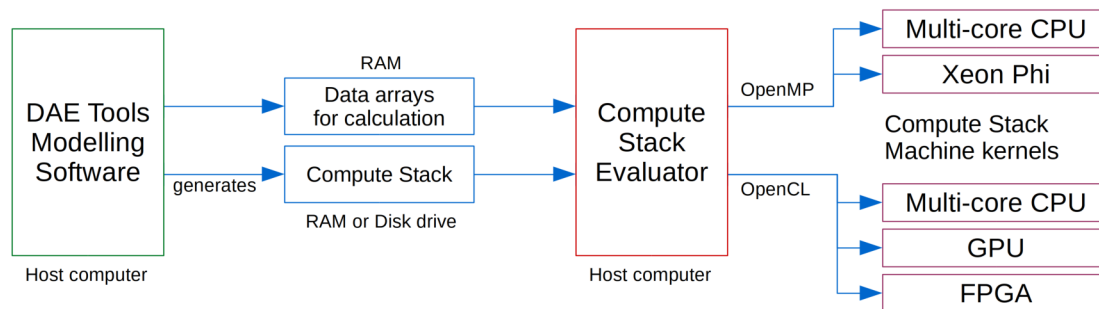
Figure 2. DAE Tools simulation using a Compute Stack Evaluator and compute kernels

161 are benchmarked using the following computation setups: (1) sequential and OpenMP runs
162 on multi-core CPU using the Evaluation Tree approach, (2) sequential and OpenMP runs on
163 multi-core CPU using the Compute Stack approach, (3) OpenCL runs on multi-core CPU, discrete
164 NVidia GPU and integrated Intel GPU using the Compute Stack approach, (4) OpenCL runs on
165 heterogeneous multi-core CPU/discrete GPU system using the Compute Stack approach, and (5)
166 OpenCL runs on heterogeneous multi-core CPU/discrete GPU/integrated GPU system using the
167 Compute Stack approach. The integration times are compared and analysed. Finally, a summary
168 of the most important capabilities and benefits of the proposed implementation and directions for
169 future work are given in the last section.

IMPLEMENTATION
170

Data Structures
171

172 In DAE Tools, the operator overloading technique is used to transform the model equations
173 given in implicit (acausal) form into a form suitable for evaluation. The standard mathematical
174 operations and functions are re-defined to operate on a modified ADOL-C (Walther and Griewank,
175 2012) class adouble, which has been extended to contain simulator-specific information. In the
176 existing approach, equations are transformed into a tree-like data structure called an Evaluation
177 Tree. Evaluation Trees consist of a hierarchy of nodes where each node represents either an
178 operand data or a mathematical operation. All node classes are derived from the base *adNode*
179 abstract C++ class and in a simplified form given in the source code listing 1. *adNode* class defines
180 an interface for operations on Evaluation Trees such as evaluation, units consistency check and
181 export into the Latex or MathML format. Operand data nodes contain information about constants,
182 parameters, variables and their time derivatives (*adConstantNode*, *adRuntimeVariableNode* and
183 *adRuntimeTimeDerivativeNode* classes). Mathematical operation nodes represent unary (+, -) and
184 binary operators (+, -, *, /, **) and standard mathematical unary and binary functions available
185 in <math.h>: sqrt, pow, log, log10, exp, min, max, floor, ceil, abs, sin, cos, tan, asin, acos, atan,
186 sinh, cosh, tanh, asinh, acosh, atanh, atan2 and erf (*adUnaryNode* and *adUnaryNode* classes).
187 Unary nodes contain an operand node and an unary function to perform. Binary nodes contain
188 two operand nodes (left and right) and a binary function to perform. During operations on an
189 Evaluation Tree the node tree structure is traversed by calling functions on the top level node and
190 on operand nodes in a recursive fashion.
191 However, recursive function calls are not supported by the OpenCL framework and a non-
192 contiguous layout of nodes in memory is not compatible with the concept of OpenCL memory

193 buffers. Therefore, the model equations need to be transformed into a suitable form. In this work,
194 the expression stacks in Reverse Polish Notation (postfix notation) are proposed as a platform and
195 programming language independent method to describe, store in computer memory and evaluate
196 general systems of differential and algebraic equations of any size. An expression stack data
197 structure in DAE Tools is called a Compute Stack. It represents an array of *adComputeStackItem_t*
198 objects given in the source code listing 2. Transformation of Evaluation Trees is performed in a
199 similar way as their evaluation. The Compute Stack arrays are automatically generated in DAE
200 Tools software by traversing down the hierarchy of nodes in the Evaluation Tree data structure
201 and adding the *adComputeStackItem_t* objects to an array in a predefined order for all operations
202 and their operands.

203   The *adComputeStackItem_t* structure is very compact: its size is only 16 bytes (when aligned)
204 and contains the following information: *opCode* (1 byte) is an integer defining an operation to
205 be performed; *function* (1 byte) is an integer defining the mathematical operator/function for
206 unary and binary items; *resultLocation* (1 byte) is an integer defining a location where to store
207 the results (lvalue, rvalue or value LIFO stack); *size* (4 bytes) is an integer defining the size of
208 the Compute Stack and it is always stored in the first item of the array; *data* (8 bytes) is a union
209 holding a float *value* for constants, or *indexes* structure for variable values and time derivatives,
210 or *dof_indexes* structure for values of degrees of freedom. As an example, the equation specified
211 in implicit form:

$$\frac{x[0] - x[1]}{x[2] - x[3]} + 1.2 \cdot sin(x[0]) = 0 \tag{1}$$

212 can be represented as an Evaluation Tree in Fig. 3 and as a Compute Stack in Fig. 4 (where
213 *oi* represents the variable overallIndex and *bi* its blockIndex used to identify variables in data
214 arrays).
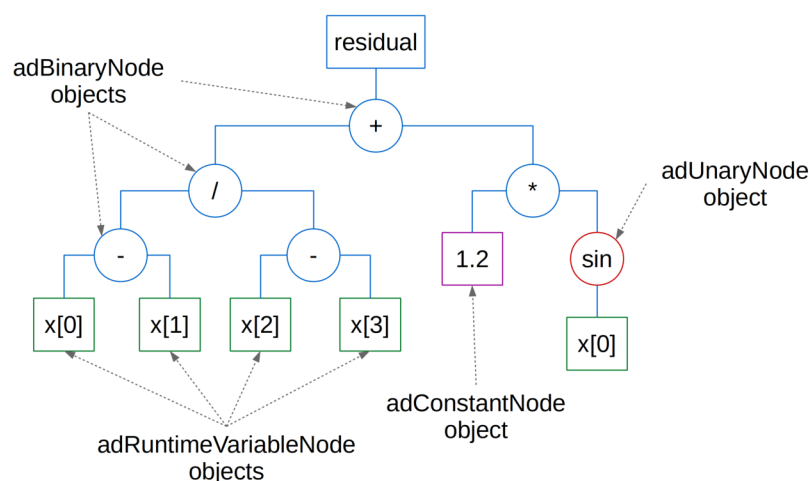
Figure 3. The Evaluation Tree data structure representing eq. 1.

	0	1	2	3	4	5	6	7	8	9	10	11
opCode:	Variable value	Variable value	Binary operator	Variable value	Variable value	Binary operator	Binary operator	Constant value	Variable value	Unary function	Binary operator	Binary operator
function:			Minus			Minus	Divide			sin	Multi	Plus
data.value:								1.2				
data.indexes:	oi = 0 bi = 0	oi = 1 bi = 1		oi = 2 bi = 2	oi = 3 bi = 3				oi = 0 bi = 0			

Figure 4. The Compute Stack data structure representing eq. 1.

Evaluation of a Single Equation

Evaluation of a single Evaluation Tree is performed in a recursive fashion starting with the top level node and traversing down the tree by processing its operand nodes.

A single Compute Stack is evaluated by a Compute Stack Machine using a LIFO queue. As an input, the Compute Stack Machine that calculates an equation residual (depicted in Fig. 5) requires the *daeComputeStackEvaluationContext_t* object, a Compute Stack array as a stream of contiguous data and random access data arrays with variable values (x), time derivatives (dx/dt) and degrees of freedom (y). *daeComputeStackEvaluationContext_t* structure contains all run-time information required by a Compute Stack Machine such as the current time, number of variables and sizes of input arrays. Evaluation of a Jacobian item requires an additional input array of *adJacobianMatrixItem_t* structures, while evaluation of a sensitivity residual requires two additional input arrays: sensitivity values and sensitivity derivatives.

An implementation of a Compute Stack Machine is fairly simple since it performs only a limited number of operations. It uses a LIFO queue composed of *adouble_t* objects. The *adouble_t* structure is used for auto-differentiation and holds a value and a derivative data. The Compute Stack Machine is implemented as a function *evaluateComputeStack* in C99. The algorithm (in a simplified form for clarity) is given in the source code listing 3 while the auxiliary data structures in the source code listing 2.
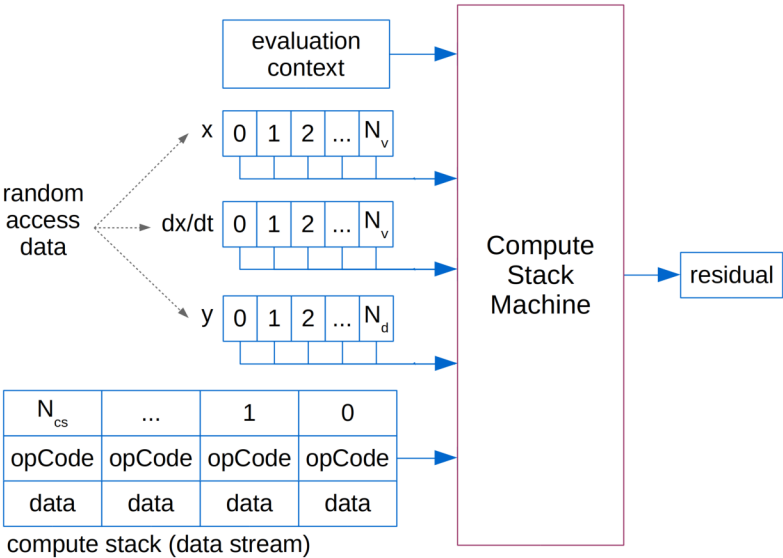
Figure 5. The Compute Stack Machine

Evaluation of Systems of Equations

Systems of equations defined using the Evaluation Tree approach are stored in memory as an array of smart pointers to *adNode*-derived class objects. Parallel evaluation of systems of equations using the Evaluation Tree approach is rather straightforward: a single OpenMP parallel for loop is used to launch a team of threads, where each thread processes its chunk of equations. The C++ implementation is given in the source code listing 4.

Systems of equations defined using the Compute Stack approach are stored in memory as a single one-dimensional array of *adComputeStackItem_t* objects populated with Compute Stacks from all equations. The reason is OpenCL memory model where memory buffers can only hold contiguous blocks of generic data. Therefore, multidimensional arrays must be represented as one-dimensional ones. Since the size of individual Compute Stacks is not constant an additional array of integers are required that mark a starting index for each equation's Compute Stack. In addition, the information required to evaluate a sparse Jacobian matrix are stored as an array of *adJacobianMatrixItem_t* objects. The *adJacobianMatrixItem_t* structure is given in the source code listing 2 and contains the equation index and the variable block and overall indexes which identify a row and a column of the item in the Jacobian matrix and the variable with respect to which a derivative is requested.

Parallel evaluation of systems of equations using the Compute Stack approach is performed through a common interface called a Compute Stack Evaluator. The interface is specified in the *adComputeStackEvaluator_t* abstract C++ class given in the source code listing 5. Compute Stack Evaluator implementations utilising a particular framework and/or targeting different computing devices are developed by deriving a new class from the base class. *adComputeStackEvaluator_t* class interface is designed with a support for heterogeneous computing in mind. Basically, it can contain nested evaluators targeting different computing hardware, each evaluator processing the specified chunk of equations. Two different Compute Stack Evaluator implementations have been developed: (a) using OpenMP API, and (b) using OpenCL framework. OpenCL implementation can use a single or multiple OpenCL-enabled devices for heterogeneous computing. All implementations use the Compute Stack Machine function presented in the source code listing 3. OpenMP implementation is given in the source code listing 6 and OpenCL kernel in the source code listing 7.

In the OpenMP implementation, every thread evaluates a chunk of the total number of equations, one at the time. In the OpenCL implementation, every thread evaluates only a single equation. Each thread executes a for loop where mathematical operations are performed in a single IF block controlled by the op_code (as given in the source code listing 3). On GPUs, the thread divergence should not ever occur since the control flow does not depend on the thread ID and all threads execute the very same block of code (which does contain conditional jumps). At the moment, it is left to OpenCL compilers to optimise the IF block for a particular platform.

The OpenCL implementation of the Compute Stack Evaluator interface requires memory transfers between the CPU and the device: (1) the Compute Stack array, the active equation set indexes and the Jacobian information data are copied to a device only once during the initialisation ($16N_{cs} + 4N_{eq} + 12N_{nz}$ bytes), (2) variable values and derivatives are copied before every evaluation ($16N_{eq}$ bytes), (3) residuals and Jacobian values are copied from the device after every evaluation ($8N_{eq}$ bytes for residuals and $8N_{nz}$ bytes for Jacobian evaluation). Here, N_{eq} is the number of equations/variables in the DAE system, N_{cs} is the size of the ComputeStack array (for all equations), and N_{nz} is the number of non-zero items in the Jacobian matrix. The current OpenCL implementation always explicitly copies the data between CPU-owned and

279 device-owned buffers although in some instances where a computing device shares memory with
280 the CPU (i.e. integrated GPU) it is possible to use a zero-copy buffer to avoid movement of data
281 between different locations in the same memory.

Listing 1. Evaluation Tree related C++ classes (simplified)

```cpp
/* Base abstract class */
class adNode
{
public:
    virtual ~adNode(){}
    virtual adouble_t Evaluate(const daeExecutionContext* EC) const = 0;
    virtual quantity  GetQuantity() const = 0;
    virtual string    SaveAsLatex(const daeNodeSaveAsContext* c) const = 0;
};

/* Constants */
class adConstantNode : public adNode
{
public:
    quantity value;
};

/* Variable values */
class adRuntimeVariableNode : public adNode
{
public:
    size_t overallIndex;
    size_t blockIndex;
};

/* Variable time derivatives */
class adRuntimeTimeDerivativeNode : public adNode
{
public:
    size_t order;
    size_t overallIndex;
    size_t blockIndex;
};

/* Unary operators and functions: -, sqrt, log, log10, exp, sin, cos, ... */
class adUnaryNode : public adNode
{
public:
    adNode*           operand;
    daeeUnaryFunctions function;
};

/* Binary operators and functions: +, -, *, /, **, pow, atan2, min, max */
class adBinaryNode : public adNode
{
public:
    adNode*           leftOperand;
    adNode*           rightOperand;
    daeeBinaryFunctions function;
};
```

Listing 2. Compute Stack related C99 data structures

```
334
335    /* op codes */
336    typedef enum
337    {
338        eOP_Unknown = 0,
339        eOP_Constant,
340        eOP_Time,
341        eOP_InverseTimeStep,
342        eOP_Variable,
343        eOP_DegreeOfFreedom,
344        eOP_TimeDerivative,
345        eOP_Unary,
346        eOP_Binary
347    }daeeOpCode;
348
349    /* real_t is defined as a single or a double precision floating point type. */
350    #define real_t double
351
352    typedef struct adComputeStackItem_
353    {
354        uint8_t  opCode;         /* Operation to perform */
355        uint8_t  function;       /* Unary or binary function code */
356        uint8_t  resultLocation; /* lvalue, value or rvalue LIFO stacks */
357        uint32_t size;           /* Size of the compute stack array */
358        union data_
359        {
360            real_t value;          /* For constants */
361            struct dof_indexes_    /* For degrees of freedom */
362            {
363                uint32_t overallIndex;
364                uint32_t dofIndex;
365            }dof_indexes;
366            struct indexes_        /* For variables (algebraic and differential) */
367            {
368                uint32_t overallIndex;
369                uint32_t blockIndex;
370            }indexes;
371        }data;
372    }adComputeStackItem_t;
373
374    /* Auxiliary data structures */
375    typedef struct adouble_
376    {
377        real_t value;
378        real_t derivative;
379    } adouble_t;
380
381    typedef struct daeComputeStackEvaluationContext_
382    {
383        real_t   currentTime;
384        real_t   inverseTimeStep;
385        uint32_t equationCalculationMode;
386        uint32_t sensitivityParameterIndex;
387        uint32_t jacobianIndex;
388        uint32_t numberOfVariables;
389        uint32_t startEquationIndex;
390        uint32_t numberOfEquations;
391        uint32_t startJacobianIndex;
392        uint32_t numberOfJacobianItems;
393        uint32_t numberOfDOFs;
394        uint32_t numberOfComputeStackItems;
395        uint32_t valuesStackSize;
396        uint32_t lvaluesStackSize;
397        uint32_t rvaluesStackSize;
398    }daeComputeStackEvaluationContext_t;
399
400    typedef struct adJacobianMatrixItem_
401    {
402        uint32_t equationIndex;
403        uint32_t overallIndex;
404        uint32_t blockIndex;
405    } adJacobianMatrixItem_t;
406
```

Listing 3. Compute Stack Machine implementation in C99 (a simplified algorithm)

```
407
408   adouble_t evaluateComputeStack(const adComputeStackItem_t*        computeStack,
409                                  daeComputeStackEvaluationContext_t EC,
410                                  const real_t*                      dofs,
411                                  const real_t*                      values,
412                                  const real_t*                      timeDerivatives,
413                                  const real_t*                      svalues,
414                                  const real_t*                      sdvalues)
415   {
416       for(int i = 0; i < computeStackSize; i++)
417       {
418           /* Take the item from the array. */
419           const adComputeStackItem_t item = computeStack[i];
420
421           /* Initialise a temporary adouble_t object to zero. */
422           adouble_t result = {0.0, 0.0};
423
424           if(item.opCode == eOP_Constant)
425           {
426               /* result.value is set to item.data.value
427                * result.derivative is zero. */
428           }
429           else if(item.opCode == eOP_Time)
430           {
431               /* result.value is set to EC.currentTime
432                * result.derivative is zero. */
433           }
434           else if(item.opCode == eOP_InverseTimeStep)
435           {
436               /* result.value is set to EC.inverseTimeStep
437                * result.derivative is zero. */
438           }
439           else if(item.opCode == eOP_Variable)
440           {
441               /* result.value is a variable value and set to values[blockIndex]
442                * result.derivative is set to:
443                *  - 1.0 if overallIndex == EC.jacobianIndex
444                *  - svalues[blockIndex] if overallIndex == EC.sensitivityParameterIndex
445                *  - otherwise zero */
446           }
447           else if(item.opCode == eOP_TimeDerivative)
448           {
449               /* result.value is a variable derivative and set to timeDerivatives[blockIndex]
450                * result.derivative is set to:
451                *  - EC.inverseTimeStep if overallIndex == EC.jacobianIndex
452                *  - sdvalues[blockIndex] if overallIndex == EC.sensitivityParameterIndex
453                *  - otherwise zero */
454           }
455           else if(item.opCode == eOP_DegreeOfFreedom)
456           {
457               /* result.value is a degree of freedom time and set to dofs[dofIndex]
458                * result.derivative is set to:
459                *  - 1.0 if overallIndex == EC.sensitivityParameterIndex
460                *  - otherwise zero */
461           }
462           else if(item.opCode == eOP_Unary)
463           {
464               /* The operand value is popped from the stack,
465                * unary function performed and the result pushed onto the stack. */
466           }
467           else if(item.opCode == eOP_Binary)
468           {
469               /* The left and right operands are popped from the stack,
470                * binary function performed and the result pushed onto the stack. */
471           }
472
473           /* Push the result onto the stack. */
474       }
475
476       /* Pop the final result from the stack and return. */
477
478   }
```

Listing 4. Evaluation of residuals using the Evaluation Tree approach and OpenMP API

```
479
480  /* A system of equations is stored as an array: std::vector<adNodePtr> evaluationTrees. */
481
482  /* Set number of threads (or leave the default value). */
483  omp_set_num_threads(num_threads);
484
485  /* Start the parallel OpenMP for loop. */
486  #pragma omp parallel for firstprivate(EC)
487  for(int ei = 0; ei < numberOfEquations; ei++)
488  {
489      adNode* node = evaluationTrees[ei];
490      /* The flag EC.equationCalculationMode defines the operation to perform:
491       * evaluation of residuals, Jacobian matrix or sensitivity residuals.
492       * Here, we calculate residuals. */
493      adouble_t residual = node->Evaluate(&EC);
494      residuals[ei] = residual.value;
495  }
496
```

Listing 5. Compute Stack Evaluator abstract C++ class

```
497
498  class adComputeStackEvaluator_t
499  {
500  public:
501      virtual ~adComputeStackEvaluator_t(){}
502
503      /* Compute Stack related arrays are copied to the compute device
504       * only once (during the initialisation phase). */
505      virtual void Initialize(bool                     calculateSensitivities,
506                              size_t                    numberOfVariables,
507                              size_t                    numberOfEquationsToProcess,
508                              size_t                    numberOfDOFs,
509                              size_t                    numberOfComputeStackItems,
510                              size_t                    numberOfJacobianItems,
511                              size_t                    numberOfJacobianItemsToProcess,
512                              adComputeStackItem_t*     computeStacks,
513                              uint32_t*                 activeEquationSetIndexes,
514                              adJacobianMatrixItem_t* computeStackJacobianItems) = 0;
515
516      /* The arrays required for evaluation must be copied to the device
517       * before every computation. */
518      virtual void EvaluateResiduals(daeComputeStackEvaluationContext_t EC,
519                              real_t*                   dofs,
520                              real_t*                   values,
521                              real_t*                   timeDerivatives,
522                              real_t*                   residuals) = 0;
523
524      virtual void EvaluateJacobian(daeComputeStackEvaluationContext_t EC,
525                              real_t*                   dofs,
526                              real_t*                   values,
527                              real_t*                   timeDerivatives,
528                              real_t*                   jacobianItems) = 0;
529
530      virtual void EvaluateSensitivityResiduals(daeComputeStackEvaluationContext_t EC,
531                                      real_t*                   dofs,
532                                      real_t*                   values,
533                                      real_t*                   timeDerivatives,
534                                      real_t*                   svalues,
535                                      real_t*                   sdvalues,
536                                      real_t*                   sresiduals) = 0;
537  };
538
```

Listing 6. Evaluation of residuals using the Compute Stack approach and OpenMP API

```
539
540  void EvaluateResiduals(const adComputeStackItem_t*        computeStacks,
541                         const uint32_t*                   activeEquationSetIndexes,
542                         daeComputeStackEvaluationContext_t EC,
543                         const real_t*                     dofs,
544                         const real_t*                     values,
545                         const real_t*                     timeDerivatives,
546                         const real_t*                     residuals)
547  {
548      /* Systems of equations are represented as a single array populated with
549       * Compute Stacks from all equations:
550       *   adComputeStackItem_t computeStacks[numberOfComputeStackItems].
551       * Starting indexes for each equation's Compute Stack are stored in:
552       *   uint32_t activeEquationSetIndexes[numberOfEquations]. */
553
554      /* Set number of threads (or leave the default value). */
555      omp_set_num_threads(num_threads);
556
557      /* Start the parallel OpenMP for loop. */
558      #pragma omp parallel for firstprivate(EC)
559      for(int ei = 0; ei < numberOfEquations; ei++)
560      {
561          /* Locate the current equation stack in the array of all compute stacks. */
562          uint32_t firstIndex                    = activeEquationSetIndexes[ei];
563          const adComputeStackItem_t* computeStack = &computeStacks[firstIndex];
564
565          /* Evaluate the compute stack. */
566          adouble_t residual = evaluateComputeStack(computeStack,
567                                                    EC,
568                                                    dofs,
569                                                    values,
570                                                    timeDerivatives,
571                                                    NULL,
572                                                    NULL);
573
574          /* Set the value in the residuals array. */
575          residuals[ei] = residual.value;
576      }
577
578  }
```

Listing 7. OpenCL kernel for evaluation of a single residual using the Compute Stack approach

```
579
580  void __kernel EvaluateResidual(__global adComputeStackItem_t*     computeStacks,
581                                 __global uint32_t*                 activeEquationSetIndexes,
582                                 daeComputeStackEvaluationContext_t EC,
583                                 __global real_t*                   dofs,
584                                 __global real_t*                   values,
585                                 __global real_t*                   timeDerivatives,
586                                 __global real_t*                   residuals)
587  {
588      /* Get the global thread id. */
589      int residualIndex = get_global_id(0);
590      /* Get the equation index to process by this thread. */
591      int equationIndex = EC.startEquationIndex + get_global_id(0);
592
593      /* Locate the current equation stack in the array of all compute stacks. */
594      uint32_t firstIndex                    = activeEquationSetIndexes[equationIndex];
595      __global adComputeStackItem_t* computeStack = &computeStacks[firstIndex];
596
597      /* Evaluate the compute stack. */
598      __global real_t* null_data = 0;
599      adouble_t residual = evaluateComputeStack(computeStack,
600                                                EC,
601                                                dofs,
602                                                values,
603                                                timeDerivatives,
604                                                null_data,
605                                                null_data);
606
607      /* Set the value in the residuals array. */
608      residuals[residualIndex] = residual.value;
609
610  }
```

BENCHMARKS

The performance of Evaluation Tree and Compute Stack approaches were evaluated by benchmarking the numerical solutions of three medium-scale models. The list and the description of all computing setups are given in Table 1. Cases 3 and 4 use the old Evaluation Tree approach and all other use the Compute Stack approach. Cases 2 and 4 utilise the OpenMP API while Cases 5-12 utilise the OpenCL framework. The heterogeneous cases used different proportions of the work done by individual OpenCL devices: (Case 8) 30% on the CPU and 70% on the NVidia GPU, (Case 9) 50% on the CPU and 50% on the NVidia GPU, (Case 10) 70% on the CPU and 30% on the NVidia GPU, (Case 11) 40% on the CPU, 40% on the NVidia GPU and 20% on the Indel HD GPU, and (Case 12) 30% on the CPU, 60% on the NVidia GPU and 10% on the Indel HD GPU. The hardware configuration consists of Intel i7-6700HQ CPU (4 cores/8 threads at 2.6 GHz, 8 GB of RAM, 34.32 GFLOPS peak double precision, 128-bit bus width and 34.1 GB/s max memory bandwidth), an integrated Intel HD 530 GPU (24 execution units at 350 MHz, 256 MB of RAM, 33.6 GFLOPS peak double precision), and a discrete NVidia GeForce GTX 950M GPU (640 execution units at 914 MHz, 2 GB of RAM, 36.56 GFLOPS peak double precision, 128-bit bus width and 32 GB/s max memory bandwidth). The number of equations (N_{eq}), the number of non-zero items in the Jacobian matrix (the total number $N_{nz} = \sum_{i=1}^{N_{eq}} N_{nz}[i]$ and the average number per equation $N_{nz/equation}$), 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 all benchmarks are given in Table 2.

All benchmark models were implemented in Python using the DAE Tools v1.8.0 software (Nikolić, 2016). DAE Tools is free software released under the GNU General Public Licence. The source code, the installation packages and more information about the software can be found on the website (http://www.daetools.com). For integration of DAE systems in time the software uses the variable-step variable-order backward differentiation formula available in SUNDIALS IDAS DAE solver (Hindmarsh et al., 2005). Systems of linear equations are solved using SuperLU and SuperLU_MT solvers (Li, 2005). The benchmarks are carried out in 64-bit Debian Stretch GNU/Linux and compiled using the gcc 6.3 compiler, OpenMP 4.5 from the GOMP library, OpenCL 1.2 from NVidia CUDA 9.0 with v384.90 display driver, and OpenCL 2.0 from Intel GPU/CPU driver package for Linux r5.0.

Table 1. Computational setups

Case	Name	Description
1.	CPU Seq	Sequential on Intel CPU
2.	CPU OpenMP	OpenMP on Intel CPU
3.	CPU Seq (ET)	Sequential using the Evaluation Tree approach on Intel CPU
4.	CPU OpenMP (ET)	OpenMP using the Evaluation Tree approach on Intel CPU
5.	CPU OpenCL	OpenCL on Intel CPU
6.	GPU (HD)	OpenCL on Intel HD GPU
7.	GPU	OpenCL on NVidia GPU
8.	CPU+GPU (1)	OpenCL on Intel CPU (30%) and NVidia GPU (70%)
9.	CPU+GPU (2)	OpenCL on Intel CPU (50%) and NVidia GPU (50%)
10.	CPU+GPU (3)	OpenCL on Intel CPU (70%) and NVidia GPU (30%)
11.	CPU+GPUx2 (1)	OpenCL on Intel CPU (40%), NVidia GPU (40%) and Intel GPU (20%)
12.	CPU+GPUx2 (2)	OpenCL on Intel CPU (30%), NVidia GPU (60%) and Intel GPU (10%)

Table 2. Size of the benchmarked models

	N_{eq}	N_{nz}	N_{cs}	$N_{nz/equation}$	$N_{cs/equation}$	$N_{cs/jacob\_row}$
Benchmark 1	46,948	205,218	9,199,459	4.37	196	1,475
Benchmark 2	20,000	355,216	15,554,304	17.76	778	13,902
Benchmark 3	36,864	952,278	7,122,536	25.83	193	5,318

Benchmark 1: Transient two-dimensional Burgers equations, structured grid

The model describes a viscous fluid flow presented in Appendix C of Salari and Knupp (2000) and defined by the Burgers equations given in eq. 2. The Burgers equations are spatially discretised using the centered Finite Difference Method.

$$\frac{du}{dt} + \frac{d(uu)}{dx} + \frac{d(uv)}{dy} - v\left(\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2}\right) = S_u(x,y)$$
$$\frac{dv}{dt} + \frac{d(vu)}{dx} + \frac{d(vv)}{dy} - v\left(\frac{d^2v}{dx^2} + \frac{d^2v}{dy^2}\right) = S_v(x,y)$$

(2)

Here, u and v are two components of the velocity field, v is kinematic viscosity, and S_u and S_v are the source terms.

This problem uses a formal code verification technique: the *Method of Manufactured Solutions* (Salari and Knupp, 2000) for checking the correctness of the numerical software. The method tests the software capabilities in full generality and therefore results in a high degree of confidence that all coding mistakes which prevent the equations being solved correctly have been identified. The procedure is the following: (1) a function representing the solution is selected (the manufactured solution), (2) the new source term for the original problem is computed, and (3) the original problem is solved with the new source term. The computed numerical solution should be equal to the manufactured one. In this work, the quality of the solutions are checked using the most rigorous acceptance criteria available: the normalised global errors and the order of accuracy.

15/30

657   The manufactured solution in this example is given by:

$$u_m(x,y) = u_0 \left(sin(x^2 + y^2 + w_0 t) + \varepsilon \right)$$
$$v_m(x,y) = v_0 \left(cos(x^2 + y^2 + w_0 t) + \varepsilon \right)$$
(3)

658   and the new source terms are:

$$S_u(x,y) = \frac{du_m}{dt} + \frac{d(u_m u_m)}{dx} + \frac{d(u_m v_m)}{dy} - \nu \left(\frac{d^2 u_m}{dx^2} + \frac{d^2 u_m}{dy^2} \right)$$
$$S_v(x,y) = \frac{dv_m}{dt} + \frac{d(v_m u_m)}{dx} + \frac{d(v_m v_m)}{dy} - \nu \left(\frac{d^2 v_m}{dx^2} + \frac{d^2 v_m}{dy^2} \right)$$
(4)

659 Equations are distributed on a two-dimensional rectangular grid (-0.1, 0.7)x(0.2, 0.8) with 120x96
660 grid points. Input parameters are: $u_0 = 1.0\frac{m}{s}$, $v_0 = 1.0\frac{m}{s}$, $w_0 = 0.1$, $\nu = 0.7\frac{m^2}{s}$, $\varepsilon = 0.001$.
661 The Dirichlet boundary conditions on all four edges are set using the manufactured solution.
662 The initial conditions are set using the manufactured solution with $w_0 = 0.0$. The system is
663 integrated for 90 seconds and the outputs are taken every 2 seconds. The source code is given
664 in the supplemental source code listing 1 (benchmark_1.py file) and on DAE Tools website
665 (http://www.daetools.com/docs/tutorials-cv.html#tutorial-cv-4).
666   The normalised global errors ($\|E_u\|$ for the horizontal and $\|E_v\|$ for the vertical velocity com-
667 ponent) and orders of accuracy (p_u for the horizontal and p_v for the vertical velocity component)
668 for the steady-state solution and different mesh sizes (grids used: 10x8, 20x16, 40x32, and 80x64)
669 are given in Table 3. The results show a decrease in the normalised global error with the decrease
670 in mesh size and the order of accuracy approaching the theoretical order 2, as expected.

Table 3. The normalised global errors and the orders of accuracy for the steady-state solution

Mesh:	10x8	20x16	40x32	80x64
$\|Eu\|$	1.92e-04	4.80e-05	1.20e-05	3.00e-06
$\|Ev\|$	4.80e-04	1.20e-04	3.00e-05	7.50e-06
p_u		1.996	1.999	2.000
p_v		1.994	1.999	2.000

671   In addition, for benchmarking purposes, the same model is re-implemented in plain C++. The
672 functions for evaluation of residuals and the Jacobian matrix use the same auto-differentiation
673 method as in the other approaches: operator overloading technique adopted from ADOL-C.
674 The time for a sequential evaluation of 1000 residuals and Jacobian matrix in a single loop
675 is measured and compared for both the C++ and the Compute Stack implementations. This
676 way the effect of other parts of the simulation code is minimised and only evaluation of model
677 equations is benchmarked. The source code is given in the supplemental source code listing 1
678 (cxx-cs-benchmark directory). The compilation instructions are located in the read_me.txt file.
679 The software loads the Compute Stack and Jacobian data from the input data files, calls the C++
680 and ComputeStack evaluation functions in a loop for the given number of iterations and records
681 the execution times.

Benchmark 2: Transient two-dimensional Cahn-Hilliard equations, unstructured grid

The model describes the process of phase separation, where two components of a binary mixture separate and form domains pure in each component. The problem is defined by the Cahn-Hilliard equations given in eq. 5.

$$\frac{dc}{dt} = D\nabla^2\mu$$
$$\mu = c^3 - c - \gamma\nabla^2 c \tag{5}$$

Here, D is the diffusion coefficient, c is the concentration, μ is the chemical potential and $\sqrt{\gamma}$ defines the length of the transition regions between the domains. The mesh is a simple square (0,100)x(0,100) with 100x100 elements. Input parameters are $D = 1$ and $\gamma = 1$. Boundary conditions for both c an μ are insulated boundary conditions (no flux on boundaries). Initial conditions are set to $c(0) = 0.5 + c_{noise}$ where the noise c_{noise} is specified using the normal distribution with standard deviation of 0.1. The system is integrated for 500 seconds and the outputs are taken every 5 seconds. The source code is given in the supplemental source code listing 2 (benchmark_2.py file) and on DAE Tools website (http://www.daetools.com/docs/tutorials-fe.html#tutorial-dealii-3).

The Cahn-Hilliard equations are spatially discretised using the Finite Elements Method. In DAE Tools, deal.II (http://dealii.org) library is utilised for low-level tasks such as mesh loading, management of finite element spaces, degrees of freedom, assembly of the system stiffness and mass matrices and the system load vector, and setting the boundary conditions. The assembled system matrices are used to generate a set of equations in the following form:

$$[M]\{\dot{x}\} + [A]\{x\} - \{F\} = 0 \tag{6}$$

where x and $\dot{x}$ are vectors of state variables and their derivatives, M and A are mass and stiffness matrices and F is the load vector. The generated set of equations (in general case a DAE system) are solved together with the rest of equations in the model.

Benchmark 3: Transient two-dimensional Stokes equations, unstructured grid

The model describes a fluid flow caused by the differences in buoyancy as a result of temperature differences (Stokes flow). Parts of the fluid that are hotter have a lower density and rise up while parts that are cooler are denser and sink down with gravity. The differences to the original problem are that the grid is not adaptive and no stabilisation method is used. The problem is defined by the Stokes equations given in eq. 7. The Stokes equations are spatially discretised using the Finite Elements Method.

$$-\nabla \cdot (2\eta\varepsilon(\mathbf{u})) + \nabla p = -\rho\beta T\mathbf{g}$$
$$-\nabla \cdot \mathbf{u} = 0$$
$$\frac{dT}{dt} + \mathbf{u} \cdot \nabla T - \nabla \cdot k\nabla T = \gamma \tag{7}$$

Here, $\mathbf{u}$ is the velocity field, p is the pressure, ε is the symmetric gradient of the velocity and T is temperature. The mesh is a simple square (0,1)x(0,1) with 96x96 elements. Input parameters are

713 $\kappa = 2 \cdot 10^{-5}$, $\beta = 10.0$, $\eta = 1.0$ and $\rho = 1.0$. The source term in the heat balance equation (γ) is
714 defined as three point-like heat sources near the bottom of the grid with the coordinates (0.3, 0.1),
715 (0.45, 0.1) and (0.75, 0.1), the radius $r = 1/32$ and the value 1.0. Boundary conditions are no-slip
716 no-flux conditions for the velocity, no-flux conditions for the pressure, and an insulated boundary
717 condition for the temperature. Initial conditions are set to 0.0 for the temperature and 0.0 for the
718 velocity. The system is integrated for 50 seconds and the outputs are taken every 0.5 seconds.
719 The source code is given in the supplemental source code listing 3 (benchmark_3.py file) and
720 on DAE Tools website (http://www.daetools.com/docs/tutorials-fe.html#tutorial-dealii-7). The
721 problem is originally defined in the step-31 deal.II tutorial (https://www.dealii.org/8.5.0/doxygen/
722 deal.II/step_31.html).

RESULTS

724 The comparison of the performance of the Compute Stack, the Evaluation Tree and the direct
725 C++ implementations of the Benchmark 1 model are given in Table 4. The time required to copy
726 the inputs (variable values and derivatives) to OpenCL buffers and results from OpenCL buffers,
727 given as percentage of the total time for evaluation of residuals, is presented in Table 5.
728   Execution times for different phases of the numerical solution given as percents of the total
729 integration time for the base case, Case 1 (CPU Seq) are given in Fig. 6. Four phases have
730 been analysed: (1) time for evaluation of residuals, (2) time spent in a linear solver (excluding
731 evaluation of a Jacobian matrix), (3) time for evaluation of a Jacobian matrix, and (4) time spent
732 in a DAE solver. Clearly, the first two benchmarks are dominated by the execution time of
733 evaluation functions (approximately 85% of the total time). On the other hand, Benchmark 3 is
734 dominated by the time spent in the linear solver, mostly for the sparse matrix factorisation (71%
735 of the total time). This becomes more understandable if the numbers of non-zero items in the
736 Jacobian matrix in Table 2 are compared: the number of items per equation in Benchmark 1 is
737 approximately 4 while in Benchmark 3 it is 26. Therefore, both effects, evaluation of equations
738 and solution of linear systems, need to be considered in Benchmark 3.
739   The maximum theoretical speed-ups for evaluation of model equations (compared to the se-
740 quential Case 1) can be estimated using the maximum peak performance for individual platforms.
741 For instance, in Case 7 the theoretical speed-up is 36.56 GFlops/(34.32 GFlops/8 cores) = 8.52,
742 while for Cases 2, 4 and 5 it is equal to the number of cores (8 in this work). The maximum
743 theoretical overall simulation speed-ups (as a result of parallelisation of only model equations)
744 compared to the sequential Case 1 can be calculated from the Amdahl's law using the data from
745 Fig. 6 and the maximum peak performance for individual platforms: $1/(1-p+p/s)$, where
746 p is the portion of the solution that can be parallelised (in this case the sum of the time spent
747 for evaluation of residuals and the Jacobian) and s is the maximum theoretical speed-up for
748 evaluation of model equations. The maximum theoretical overall and speed-ups for evaluation of
749 model equations are summarised in the Table 6.
750   The speed-ups achieved in the Compute Stack approach compared to the old Evaluation
751 Tree one, measured during the simulations, are summarised in Table 7. The maximum overall
752 speed-ups observed in Case 7 (NVidia GPU) compared to Cases 1 (CPU Seq) and 4 (CPU
753 OpenMP (ET)) are given in Table 8. Execution times (in seconds) and speed-ups of individual
754 phases for the single-threaded linear solver (SuperLU) for Benchmark 1 are presented in Tables 9
755 and 10, for Benchmark 2 in Tables 11 and 12, and for Benchmark 3 in Tables 13 and 14. The
756 speed-ups presented are given relative to the base case, Case 1 (sequential run on the CPU).

18/30

In addition, all runs were repeated using the multi-threaded SuperLU_MT linear solver using OpenMP API. The time spent in the linear solver and the overall speed-ups (compared to the base case where the serial linear solver is used) for Benchmarks 1, 2 and 3 are presented in Tables 15, 16, and 17, respectively.

Verification of the Compute Stack approach using the Method of Manufactured Solutions is performed in Benchmark 1 and both OpenMP and OpenCL implementations perform as expected. Additional code verification tests can be found on DAE Tools website in the Code Verification Tests section (http://www.daetools.com/docs/tutorials-cv.html).

Table 4. Performance comparison of the ComputeStack, the Evaluation Tree and the C++ implementations of Benchmark 1 model

	EvaluateResiduals			EvaluateJacobian		
	C++	CS	ET	C++	CS	ET
Average time/call, ms	48.10	69.70	364.30	353.30	507.57	1676.56
Relative to C++	1.00	1.45	7.57	1.00	1.44	4.75
Relative to CS	0.69	1.00	5.23	0.69	1.00	3.30

Table 5. Time required for the data transfer to/from OpenCL buffers, given as percentage of the total time for evaluation of residuals

		Case 5	Case 6	Case 7
Benchmark 1:	Copy inputs	1.23	3.39	1.99
	Copy outputs	0.49	2.52	1.19
	Total	1.72	5.91	3.18
Benchmark 2:	Copy inputs	0.45	1.10	0.61
	Copy outputs	0.18	0.86	0.35
	Total	0.63	1.96	0.96
Benchmark 3:	Copy inputs	2.30	2.44	2.04
	Copy outputs	0.66	1.69	1.27
	Total	2.96	4.13	3.31
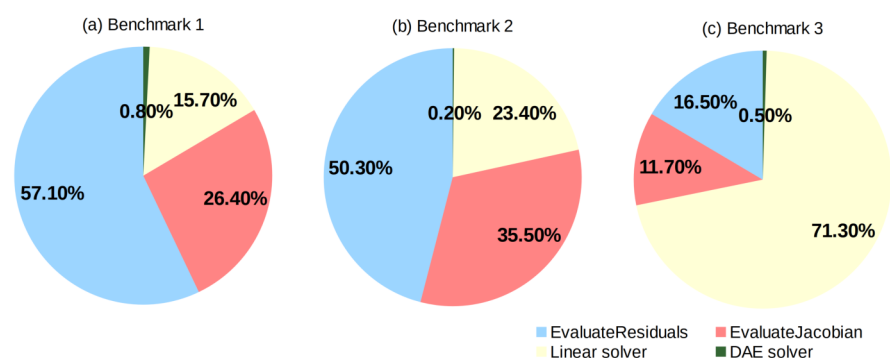
Figure 6. Case 1 - Percentage of the integration time for individual phases of the solution.

Table 6. The maximum theoretical speed-ups for evaluation of model equations and the maximum overall simulation speed-ups (relative to the sequential Case 1)

Case	Maximum overall simulation speed-up			Maximum speed-up for model equations
	Benchmark 1	Benchmark 2	Benchmark 3	
2, 4, 5	3.71	4.01	1.33	8.00
6	3.68	3.98	1.32	7.83
7	3.80	4.12	1.33	8.52
8, 9, 10	4.64	5.16	1.36	16.52
11, 12	5.02	5.64	1.37	24.35

Table 7. Speed-up observed in the ComputeStack compared to the Evaluation Tree approach

	EvaluateResiduals			EvaluateJacobian		
Benchmark:	1	2	3	1	2	3
CPU Seq vs. CPU Seq (ET)	6.25	4.55	7.69	4.00	2.63	4.00
CPU OpenMP vs. CPU OpenMP (ET)	4.29	3.54	5.74	4.24	2.41	2.99

Table 8. Maximal overall speed-up observed in Case 7 (NVidia GPU) relative to Cases 1 (CPU Seq) and 4 (CPU OpenMP (ET))

	Benchmark:	1	2	3
GPU vs. CPU Seq		2.84	3.16	1.26
GPU vs. CPU OpenMP (ET)		4.66	3.95	1.75

Table 9. Benchmark 1 - Execution times in seconds for individual phases of the numerical solution

Case	Run	EvaluateResiduals	EvaluateJacobian	Linear solver	DAE solver
1.	CPU Seq	19.57 (57.1%)	9.06 (26.4%)	5.39 (15.7%)	0.28 (0.8%)
2.	CPU OpenMP	8.94 (48.2%)	2.67 (14.4%)	6.64 (35.8%)	0.31 (1.7%)
3.	CPU Seq (ET)	121.51 (74.4%)	35.91 (22.0%)	5.52 (3.4%)	0.32 (0.2%)
4.	CPU OpenMP (ET)	38.53 (68.6%)	11.28 (20.1%)	5.99 (10.7%)	0.33 (0.6%)
5.	CPU OpenCL	8.64 (46.3%)	4.34 (23.2%)	5.40 (28.9%)	0.29 (1.6%)
6.	GPU (HD)	14.64 (47.6%)	5.63 (18.3%)	9.64 (31.3%)	0.87 (2.8%)
7.	GPU	4.68 (38.8%)	1.98 (16.5%)	5.11 (42.4%)	0.29 (2.4%)
8.	CPU+GPU (1)	7.03 (41.5%)	2.68 (15.8%)	6.94 (41.0%)	0.29 (1.7%)
9.	CPU+GPU (2)	9.32 (52.0%)	2.91 (16.2%)	5.40 (30.1%)	0.30 (1.7%)
10.	CPU+GPU (3)	9.37 (49.6%)	3.80 (20.1%)	5.41 (28.7%)	0.30 (1.6%)
11.	CPU+GPUx2 (1)	8.45 (47.2%)	2.35 (13.1%)	6.71 (37.5%)	0.39 (2.2%)
12.	CPU+GPUx2 (2)	7.37 (43.4%)	2.11 (12.4%)	6.76 (39.8%)	0.75 (4.4%)

Table 10. Benchmark 1 - Speed-up achieved in the evaluate functions and the overall speed-up

Case	Run	EvaluateResiduals	EvaluateJacobian	Overall
1.	CPU Seq	1.00	1.00	1.00
2.	CPU OpenMP	2.19	3.39	1.85
3.	CPU Seq (ET)	0.16	0.25	0.21
4.	CPU OpenMP (ET)	0.51	0.80	0.61
5.	CPU OpenCL	2.27	2.09	1.84
6.	GPU (HD)	1.34	1.61	1.11
7.	GPU	4.18	4.56	2.84
8.	CPU+GPU (1)	2.79	3.38	2.02
9.	CPU+GPU (2)	2.10	3.11	1.91
10.	CPU+GPU (3)	2.09	2.38	1.82
11.	CPU+GPUx2 (1)	2.32	3.86	1.92
12.	CPU+GPUx2 (2)	2.65	4.29	2.02

Table 11. Benchmark 2 - Execution times in seconds for individual phases of the numerical solution

Case	Run	EvaluateResiduals	EvaluateJacobian	Linear solver	DAE solver
1.	CPU Seq	84.20 (50.3%)	59.37 (35.5%)	23.43 (14.0%)	0.38 (0.2%)
2.	CPU OpenMP	34.44 (42.1%)	19.42 (23.7%)	27.57 (33.7%)	0.44 (0.5%)
3.	CPU Seq (ET)	377.63 (67.3%)	157.98 (28.2%)	24.70 (4.4%)	0.43 (0.1%)
4.	CPU OpenMP (ET)	122.70 (61.8%)	46.90 (23.6%)	28.32 (14.3%)	0.47 (0.2%)
5.	CPU OpenCL	40.63 (42.8%)	29.00 (30.5%)	24.94 (26.3%)	0.41 (0.4%)
6.	GPU (HD)	74.52 (46.4%)	39.43 (24.5%)	45.76 (28.5%)	1.02 (0.6%)
7.	GPU	24.71 (46.6%)	5.01 (9.5%)	22.90 (43.2%)	0.38 (0.7%)
8.	CPU+GPU (1)	25.35 (42.9%)	7.83 (13.2%)	25.06 (42.4%)	0.86 (1.5%)
9.	CPU+GPU (2)	25.59 (41.3%)	10.38 (16.8%)	25.54 (41.2%)	0.44 (0.7%)
10.	CPU+GPU (3)	24.42 (37.2%)	15.33 (23.3%)	25.49 (38.8%)	0.46 (0.7%)
11.	CPU+GPUx2 (1)	33.33 (44.2%)	16.17 (21.4%)	25.53 (33.8%)	0.45 (0.6%)
12.	CPU+GPUx2 (2)	23.52 (40.1%)	9.27 (15.8%)	25.30 (43.2%)	0.50 (0.8%)

Table 12. Benchmark 2 - Speed-up achieved in the evaluate functions and the overall speed-up

Case	Run	EvaluateResiduals	EvaluateJacobian	Overall
1.	CPU Seq	1.00	1.00	1.00
2.	CPU OpenMP	2.44	3.06	2.04
3.	CPU Seq (ET)	0.22	0.38	0.30
4.	CPU OpenMP (ET)	0.69	1.27	0.84
5.	CPU OpenCL	2.07	2.05	1.76
6.	GPU (HD)	1.13	1.51	1.04
7.	GPU	3.41	11.85	3.16
8.	CPU+GPU (1)	3.32	7.59	2.83
9.	CPU+GPU (2)	3.29	5.72	2.70
10.	CPU+GPU (3)	3.45	3.87	2.55
11.	CPU+GPUx2 (1)	2.53	3.67	2.22
12.	CPU+GPUx2 (2)	3.58	6.40	2.86

Table 13. Benchmark 3 - Execution times in seconds for individual phases of the numerical solution

Case	Run	EvaluateResiduals	EvaluateJacobian	Linear solver	DAE solver
1.	CPU Seq	16.33 (16.5%)	11.57 (11.7%)	70.65 (71.3%)	0.53 (0.5%)
2.	CPU OpenMP	8.12 (9.4%)	5.22 (6.0%)	72.43 (83.8%)	0.65 (0.7%)
3.	CPU Seq (ET)	121.76 (49.6%)	47.20 (19.2%)	76.05 (31.0%)	0.58 (0.2%)
4.	CPU OpenMP (ET)	46.11 (33.6%)	15.72 (11.5%)	74.77 (54.5%)	0.63 (0.5%)
5.	CPU OpenCL	8.46 (9.7%)	6.13 (7.0%)	72.31 (82.7%)	0.56 (0.6%)
6.	GPU (HD)	21.99 (19.0%)	11.59 (10.0%)	81.27 (70.3%)	0.68 (0.6%)
7.	GPU	6.57 (8.3%)	1.59 (2.0%)	70.11 (89.0%)	0.50 (0.6%)
8.	CPU+GPU (1)	7.27 (9.1%)	1.52 (1.9%)	70.24 (88.3%)	0.54 (0.7%)
9.	CPU+GPU (2)	6.53 (7.9%)	3.11 (3.8%)	72.71 (87.7%)	0.55 (0.7%)
10.	CPU+GPU (3)	7.17 (8.5%)	3.96 (4.7%)	72.76 (86.2%)	0.53 (0.6%)
11.	CPU+GPUx2 (1)	21.24 (20.7%)	8.27 (8.1%)	71.57 (69.9%)	1.35 (1.3%)
12.	CPU+GPUx2 (2)	18.91 (19.9%)	4.00 (4.2%)	71.20 (74.9%)	0.91 (1.0%)

Table 14. Benchmark 3 - Speed-up achieved in the evaluate functions and the overall speed-up

Case	Run	EvaluateResiduals	EvaluateJacobian	Overall
1.	CPU Seq	1.00	1.00	1.00
2.	CPU OpenMP	2.01	2.21	1.15
3.	CPU Seq (ET)	0.13	0.25	0.40
4.	CPU OpenMP (ET)	0.35	0.74	0.72
5.	CPU OpenCL	1.93	1.89	1.13
6.	GPU (HD)	0.74	1.00	0.86
7.	GPU	2.49	7.29	1.26
8.	CPU+GPU (1)	2.25	7.59	1.25
9.	CPU+GPU (2)	2.50	3.71	1.20
10.	CPU+GPU (3)	2.28	2.92	1.17
11.	CPU+GPUx2 (1)	0.77	1.40	0.97
12.	CPU+GPUx2 (2)	0.86	2.89	1.04

Table 15. Benchmark 1 - The time in seconds spent in the multi-threaded linear solver and the overall speed-up

Case	Run	Linear solver (MT)	Overall speed-up
1.	CPU Seq	4.94 (14.4%)	1.01
2.	CPU OpenMP	5.53 (30.8%)	1.91
3.	CPU Seq (ET)	5.26 (3.2%)	0.21
4.	CPU OpenMP (ET)	5.55 (10.0%)	0.62
5.	CPU OpenCL	5.10 (26.2%)	1.76
6.	GPU (HD)	9.14 (30.3%)	1.14
7.	GPU	5.06 (42.1%)	2.86
8.	CPU+GPU (1)	5.17 (29.2%)	1.94
9.	CPU+GPU (2)	5.20 (28.1%)	1.85
10.	CPU+GPU (3)	5.44 (27.0%)	1.70
11.	CPU+GPUx2 (1)	5.09 (30.5%)	2.06
12.	CPU+GPUx2 (2)	5.26 (35.0%)	2.28

Table 16. Benchmark 2 - The time in seconds spent in the multi-threaded linear solver and the overall speed-up

Case	Run	Linear solver (MT)	Overall speed-up
1.	CPU Seq	16.56 (10.4%)	1.05
2.	CPU OpenMP	18.97 (27.8%)	2.45
3.	CPU Seq (ET)	17.09 (3.4%)	0.33
4.	CPU OpenMP (ET)	20.64 (11.3%)	0.92
5.	CPU OpenCL	16.72 (19.1%)	1.91
6.	GPU (HD)	37.02 (24.5%)	1.11
7.	GPU	16.55 (35.4%)	3.58
8.	CPU+GPU (1)	17.18 (34.5%)	3.36
9.	CPU+GPU (2)	18.34 (38.6%)	3.53
10.	CPU+GPU (3)	17.70 (29.4%)	2.78
11.	CPU+GPUx2 (1)	21.21 (29.6%)	2.34
12.	CPU+GPUx2 (2)	17.93 (35.3%)	3.29

Table 17. Benchmark 3 - The time in seconds spent in the multi-threaded linear solver and the overall speed-up

Case	Run	Linear solver (MT)	Overall speed-up
1.	CPU Seq	38.32 (57.3%)	1.48
2.	CPU OpenMP	42.65 (75.8%)	1.76
3.	CPU Seq (ET)	41.02 (21.1%)	0.51
4.	CPU OpenMP (ET)	41.81 (44.1%)	1.04
5.	CPU OpenCL	39.70 (72.2%)	1.80
6.	GPU (HD)	49.87 (60.4%)	1.20
7.	GPU	39.16 (81.9%)	2.07
8.	CPU+GPU (1)	37.90 (79.8%)	2.04
9.	CPU+GPU (2)	39.96 (79.3%)	1.97
10.	CPU+GPU (3)	40.11 (78.3%)	1.93
11.	CPU+GPUx2 (1)	40.02 (60.9%)	1.51
12.	CPU+GPUx2 (2)	40.26 (59.2%)	1.46

DISCUSSION

The performance of the Compute Stack approach was first compared to the direct C++ implementation of the Benchmark 1 model (Table 4). It was found that the Compute Stack approach is 45% slower for evaluation of residuals and 44% for evaluation of the Jacobian. It must be kept in mind that these results are approximate since neither the Compute Stack nor the C++ implementation are optimised for maximum performance. Although the Compute Stack approach can never achieve better performance than the direct C++ implementation, the Compute Stack Machine and Evaluator can be optimised for each platform. At the moment, the same code is used for all platforms.

The Compute Stack approach brings significant performance improvements compared to the Evaluation Tree approach. The speed-ups for both sequential and parallel cases using identical hardware and API are given in Table 7. Evaluation of residuals is 4.5-7.7 times faster for sequential runs and 3.5-5.7 times for OpenMP runs, while evaluation of Jacobian matrix is 2.6-4.0 times faster for sequential runs and 2.4-4.2 times for OpenMP runs (both using 8 threads). It is difficult to attribute the causes for the performance improvement in a definitive way. However, the possible reasons could be the fact that Evaluation Trees are represented by a large number of non-contiguous blocks of memory, occupy almost an order of magnitude more memory, and require millions of calls to virtual functions that must be resolved at runtime.

Regarding the effect of the memory transfer between the CPU and the computing devices, it can be seen from the Table 5 that it does not significantly affects performance: it requires 0.6-3.0% of the time for evaluation of residuals for Intel CPUs, 2.0-5.9% for the integrated Intel HD GPU and 1.0-3.3% for the discrete NVidia GPU. The reason is that the most of the data are copied to a device only once during the initialisation and only the inputs (variable values and derivatives) and results (residuals or Jacobian) need to be transferred during every evaluation of equations. Furthermore, the time for memory transfer can in some cases be reduced by using the pinned/mapped memory buffers or even completely avoided if a computing device shares memory with the CPU (i.e. integrated GPU).

The general trend in all Benchmarks (Tables 10, 12 and 14) is that the best performance is always achieved in Case 7 where a single discrete NVidia GPU was used: the achieved speed-ups in Benchmarks 1-3 are 4.18, 3.41 and 2.49 for evaluation of residuals and 4.56, 11.85 and 7.29 for evaluation of the Jacobian. The maximum theoretical speed-up in Case 7 is 8.52 (Table 6). The speed-ups for evaluation of the Jacobian are always higher than speed-ups for evaluation of residuals since a much larger amount of computation is required and the hardware is better utilised. This is especially evident in Cases 7-9 where all or a larger portion of the calculations are done on a discrete GPU. The computation load for evaluation of residuals is proportional to the average number of Compute Stack items per equation ($N_{cs/equation}$ in Table 2): 196, 778 and 193 for Benchmarks 1-3, respectively. The computation load for evaluation of the Jacobian is proportional to the average number of Compute Stack items for evaluation of a single row in the Jacobian matrix ($N_{cs/jacob\_row}$ in Table 2): 1475, 13902 and 5318 for Benchmarks 1-3, respectively. Consequently, the hardware resources per single evaluation are significantly better utilised. In addition, the speed-up for evaluation of the Jacobian in Benchmark 2 (11.85) is higher than the theoretical one (8.52). Again, it is difficult to specify the exact cause. However, the probable reason is the memory latency that has a greater negative effect on the CPU evaluation. The evaluation procedure in the Compute Stack Machine requires a for loop where data must often be fetched from the global memory before a mathematical operation can be performed.

810 When a variable value/derivative is required for calculation, the CPU must wait for the data
811 since the memory location based on the variable index is not known in advance. The memory
812 latency can be hidden on the GPU by using a large number of work-items ("threads"): while
813 work-items in one work-group wait for the data, the scheduler can switch to another work-group
814 to perform mathematical operations. This is particularly true for Benchmark 2 where, based on a
815 high $N_{cs/jacob\_row}$ value, a large amount of data must be fetched from the global memory causing
816 a very poor performance in the sequential case. On the other hand, the speed-ups achieved on the
817 Intel multi-core CPU in Case 2 (OpenCL implementation) are 2.19, 2.44 and 2.01 for evaluation
818 of residuals and 3.39, 3.06 and 2.21 for evaluation of the Jacobian, and in Case 5 (OpenMP
819 implementation) are 2.27, 2.07 and 1.93 for evaluation of residuals and 2.09, 2.05 and 1.89 for
820 evaluation of the Jacobian. The maximum theoretical speed-up in these cases is 8 (Table 6). Case
821 7 is between 24 and 287% faster compared to Case 2 (95% in average) although NVidia GPU
822 can deliver only ~7% more FLOPS. Therefore, the Compute Stack approach appears to be more
823 suitable for GPUs than multi-core CPUs. However, it must be kept in mind that the computation
824 performance also largely depends on the memory performance (since variable values must be
825 fetched from the global GPU memory or from the main CPU memory) and the complexity of
826 model equations (i.e. Benchmark 2 contains a large number of *pow* function evaluations and the
827 performance of mathematical functions may vary between platforms). In addition, in order to
828 reach the maximum performance the Compute Stack Machine and the Evaluator must be further
829 optimised.

830 Regarding the overall simulation speed-up, the best performance is again observed in Case 7
831 (Table 8): (a) the maximum overall speed-ups compared to the sequential Case 1 are 2.84, 3.16
832 and 1.26, and (b) the maximum overall speed-ups compared to Case 4 are 4.66, 3.95 and 1.75 for
833 Benchmarks 1, 2 and 3, respectively. Since the maximum overall simulation speed-ups are 3.80,
834 4.12 and 1.33 for Benchmarks 1, 2 and 3 (Table 6), the achieved performance is between 5.2 and
835 25.3% lower than theoretical. The structure of model equations plays a very important role for
836 determination of the maximum overall speed-up. For instance, as a result of the fact that only
837 28.2% of the integration time is spent on evaluation of model equations, the maximum overall
838 speed-up in Benchmark 3 is only 33% on the hardware used in this work, and 71.3% of the code
839 is still run sequentially.

840 Overall, the performance is generally in the following order: Case 7 (GPU) > Cases 8-10
841 (CPU+GPU) > Cases 11-12 (CPU+GPUx2) > Case 2 (CPU OpenMP) > Case 5 (CPU OpenCL).
842 In the models used in this work, the heterogeneous setups are always somewhat slower than a
843 single discrete GPU. In addition, heterogeneous setups that perform a larger portion of work on a
844 discrete NVidia GPU (Cases 8 and 12) always perform better. Parallel Compute Stack evaluation
845 using OpenMP is typically faster than using OpenCL on the identical hardware: overall, OpenMP
846 simulations perform up to 20% faster. The performance of all heterogeneous configurations are
847 far from the maximum theoretical. For instance, the speed-ups in the fastest heterogeneous Case
848 8 are 2.79, 3.32 and 2.25 for evaluation of residuals and 3.38, 7.59 and 7.59 for evaluation of the
849 Jacobian while the maximum theoretical speed-up is 16.52. Again, it is difficult to attribute the
850 definite causes for the observed trends. The possible reason is that the current implementation
851 of the multi-device Compute Stack Evaluator is not optimised for best performance and more
852 tests are required to make a definite conclusion. In addition, the time required for management of
853 multiple threads, each running a Compute Stack Evaluator, can contribute to the higher execution
854 times than expected.

855 The Stokes equations in Benchmark 3 create a complex system of coupled differential

856 equations. This results in a large number of non-zero items in the Jacobian matrix and affects
857 the matrix factorisation in the linear solver. The consequence is that the numerical solution is
858 dominated by the time spent in the linear solver and the overall speed-ups in parallel cases is fairly
859 modest: up to 26% in Case 7 (NVidia GPU). Therefore, additional runs have been performed
860 using the multi-threaded linear solver (SuperLU_MT) with 8 threads. As it can be seen from
861 Table 17, this leads to much better performance: the maximum overall combined speed-ups is
862 now 2.07 (again in Case 7). As for the first two benchmarks, use of a multi-threaded linear solver
863 produce only a modest improvement since the solution process is dominated by the evaluation
864 functions: less than 10% percents in Benchmark 1 and up to 40% in Benchmark 2.

CONCLUSIONS

866 In this work, the methodology for parallel evaluation of general systems of differential and
867 algebraic equations on shared memory computing devices such as general purpose processors
868 (traditional multi-core CPUs and manycore devices), streaming processors (GPGPU and FPGA)
869 and heterogeneous systems have been presented.

870 The postfix notation expression stacks (Compute Stacks) have been proposed as a platform
871 and programming language independent method to describe, store in computer memory and
872 evaluate general DAE systems of any size. Evaluation of Compute Stacks is performed by
873 Compute Stack Machines implemented in C99 language. Parallel evaluation of systems of
874 equations in DAE Tools modelling software is performed using the Compute Stack Evaluator
875 interface. Two implementations have been developed using: (a) the OpenMP API for general
876 purpose processors, and (b) the OpenCL framework for streaming processors and heterogeneous
877 systems. Both implementations have been verified using the Method of Manufactured Solutions.

878 The performance of the Compute Stack approach has been evaluated by benchmarking three
879 medium-scale models: (1) a viscous fluid flow model described by the transient Burgers equations,
880 (2) a phase separation model described by the transient Cahn-Hilliard equations, and (3) a fluid
881 flow model described by the transient Stokes equations. Twelve different configurations utilising
882 a multi-core CPU, an integrated Intel GPU, a discrete NVidia GPU and heterogeneous computing
883 setups have been analysed.

884 Four phases of the numerical solution have been analysed: (1) time for evaluation of residuals,
885 (2) time spent in a linear solver, (3) time for evaluation of a Jacobian matrix, and (4) time spent
886 in the DAE solver. Execution times of individual phases of the numerical solution and the overall
887 simulation speed-ups have been compared and discussed. Typically, more than 95% of the total
888 integration time is spent on evaluation of equations and the solution of linear systems. The
889 solution process in some examples is dominated by evaluation of equations while in others by
890 matrix factorisation in the linear solver.

891 The maximum theoretical speed-ups for evaluation of model equations and the maximum
892 theoretical overall simulation speed-ups have been estimated and compared to the benchmark
893 results in this work. Evaluation of model equations using the sequential Compute Stack approach
894 has been compared to the direct C++ and the old Evaluation Tree implementations. It has been
895 found that the Compute Stack approach is 45% slower than the direct C++ implementation and
896 more than five times faster than the old one. Regarding the parallel implementations, the highest
897 performance has been achieved using the OpenCL framework running on a discrete NVidia GPU:
898 evaluation of model equations is up to twelve times faster than the sequential version while the
899 maximum overall simulation speed-up is more than three times. It has been discovered that the

Compute Stack approach performs significantly faster on the GPU than on the multi-core CPU with the similar peak power (95% in average). In addition, it has been observed that the overall simulation performance is generally in the following order: single discrete GPU > heterogeneous CPU+GPU > heterogeneous CPU+GPUx2 > multi-core CPU using OpenMP > multi-core CPU using OpenCL.

The future work will be focused on optimisation of the Compute Stack Machine performance and implementations for additional types of computing devices (such as Xeon Phi and FPGA).

REFERENCES

Advanced Micro Devices, Inc. (2018). AMD CPU Libraries.

Akesson, J., Arzen, K.-E., Gafvert, M., Bergdahl, T., and Tummescheit, H. (2010). Modeling and optimization with Optimica and JModelica.org - languages and tools for solving large-scale dynamic optimization problems. *Comput. Chem. Eng.*, 34(11):1737–1749.

Ansys, Inc. (2018). ANSYS Fluent.

Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Rupp, K., Smith, B. F., Zampini, S., and Zhang, H. (2015). PETSc users manual. Technical Report ANL-95/11 - Revision 3.6, Argonne National Laboratory.

Bangerth, W., Hartmann, R., and Kanschat, G. (2007). deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27.

Barton, P. I. and Pantelides, C. C. (1993). gPROMS - A Combined Discrete/Continuous Modelling Environment for Chemical Processing Systems. *Simul. Ser.*, 25:25–34.

Barton, P. I. and Pantelides, C. C. (1994). Modeling of combined discrete/continuous processes. *AIChE J.*, 40(6):966–979.

COMSOL, Inc. (2018). COMSOL Multiphysics.

Dassault Systemes (2018). Abaqus.

Elmqvist, H. (1978). *A Structured Model Language for Large Continuous Systems*. PhD thesis, Department of Automatic Control, Lund University, Sweden.

Fritzson, P., Aronsson, P., Lundvall, H., Nystrom, K., Pop, A., Saldamli, L., and Broman, D. (2005). The OpenModelica modeling, simulation, and development environment. 46th Conference on Simulation and Modelling on the Scandinavian Simulation Society (SIMS2005), Trondheim, Norway.

Fritzson, P. and Engelson, V. (1998). Modelica — A unified object-oriented language for system modeling and simulation. In Jul, E., editor, *ECOOP'98 — Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 67–90. Springer Berlin Heidelberg.

Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M., Williams, A., and Stanley, K. S. (2005). An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423.

Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., and Woodward, C. S. (2005). SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers. *ACM Trans. Math. Softw.*, 31(3):363–396.

Intel Corporation (2018a). Intel Math Kernel Library.

Intel Corporation (2018b). Intel Pardiso and Iterative Sparse Solvers.

Kirk, B. S., Peterson, J. W., Stogner, R. H., and Carey, G. F. (2006). libMesh: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers*, 22(3–4):237–254. https://doi.org/10.1007/s00366-006-0049-3.

Li, X. S. (2005). An Overview of SuperLU: Algorithms, Implementation, and User Interface. *ACM Trans. Math. Softw.*, 31(3):302–325.

Morton, W. (2003). Equation-Oriented Simulation and Optimization. pages 317–357. Indian National Sciences Academy.

Nikolić, D. D. (2016). DAE Tools: Equation-based object-oriented modelling, simulation and optimisation software. *PeerJ Computer Science*, 2:e54.

952 Piela, P. C., Epperly, T. G., Westerberg, K. M., and Westerberg, A. W. (1991). ASCEND: an
953   object-oriented computer environment for modeling and analysis: The modeling language.
954   *Comput. Chem. Eng.*, 15(1):53–72.

955 Salari, K. and Knupp, P. (2000). Code verification by the Method of Manufactured Solutions.
956   Technical Report SAND2000 − 1444, Sandia National Laboratories.

957 Schenk, O., Wächter, A., and Hagemann, M. (2007). Matching-based preprocessing algorithms
958   to the solution of saddle-point problems in large-scale nonconvex interior-point optimization.
959   *Comput. Optim. Appl.*, 36(2-3):321–341.

960 The MathWorks, Inc. (2018). MATLAB.

961 The OpenFOAM Foundation (2018). OpenFOAM.

962 Walther, A. and Griewank, A. (2012). *Getting started with ADOL-C*. Chapman-Hall CRC
963   Computational Science.

964 Waterloo Maple, Inc. (2018). Maple.

965 Wolfram Research, Inc. (2015). Mathematica.