Simulation and Benchmarking of Modelica Models on Multi-core Architectures with Explicit Parallel Algorithmic Language Extensions

Afshin Hemmati Moghadam
Mahder Gebremedhin
Kristian Stavåker
Peter Fritzson

PELAB – Department of Computer and Information Science
Linköping University

Introduction

Goal: Make it easier for the non-expert programmer to get performance on multi-core architectures.

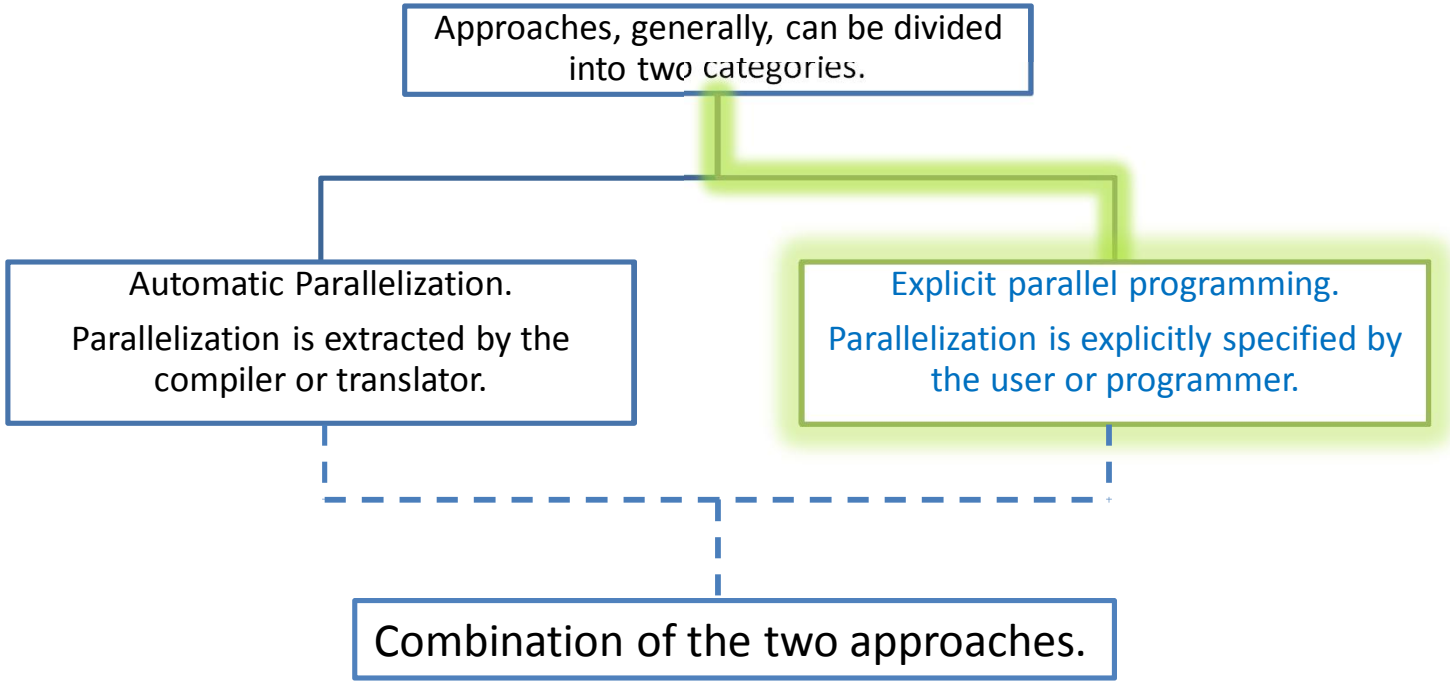
- **The Modelica language is extended with additional parallel language constructs**, implemented in OpenModelica.
- Enabling explicitly parallel algorithms (OpenCL-style) in addition to the currently available sequential constructs.

- **Primarily focused on generating optimized OpenCL code for models.**
- At the same time providing the necessary framework for generating CUDA code.

- A **benchmark suite** has been provided to **evaluate the performance** of the new extensions.
- Measurements are done using algorithms from the benchmark suite.

Multi-core Parallelism in High-Level Programming Languages

How to achieve parallelism?

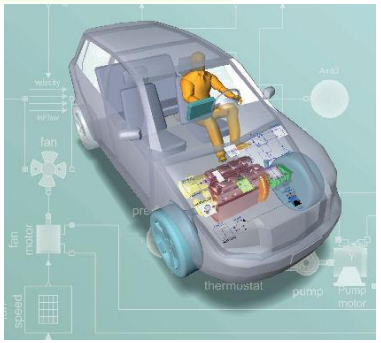


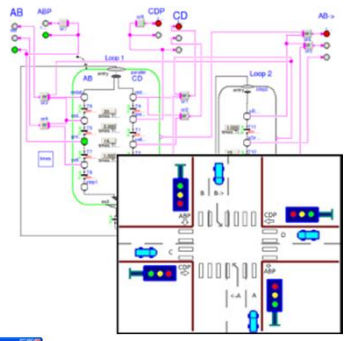
Presentation Outline

- Background
- ParModelica
- MPAR Benchmark Test Suite
- Conclusion
- Future Work

Modelica

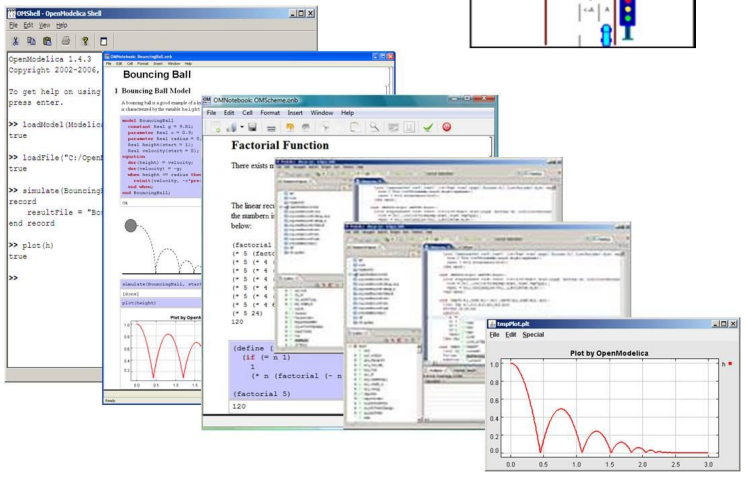
- Object-Oriented Modeling language
- Equation based
  - Models symbolically manipulated by the compiler.
- Algorithms
  - Similar to conventional programming languages.
- Conveniently models complex physical systems containing, e.g.,
  - mechanical, electrical, electronic, hydraulic, thermal. . .





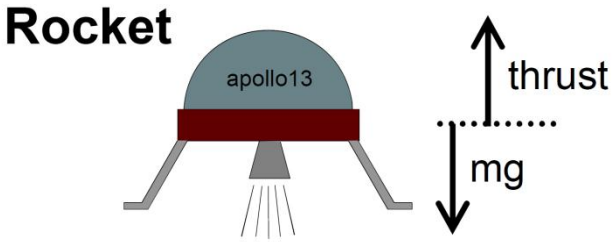
OpenModelica Environment

- Open-source Modelica-based modeling and simulation environment.
  - OMC – model compiler
  - OMEdit – graphical design editor
  - OMShell – command shell
  - OMNotebook - nteractive electronic book
  - MDT – Eclipse plug-in



Modelica Background: Example – A Simple Rocket Model

```
class Rocket "rocket class"
  parameter String name;
  Real mass(start=1038.358);
  Real altitude(start= 59404);
  Real velocity(start= -2003);
  Real acceleration;
  Real thrust;    // Thrust force on rocket
  Real gravity;   // Gravity forcefield
  parameter Real massLossRate=0.000277;
equation
  (thrust-mass*gravity)/mass = acceleration;
  der(mass) = -massLossRate * abs(thrust);
  der(altitude) = velocity;
  der(velocity) = acceleration;
end Rocket;
```



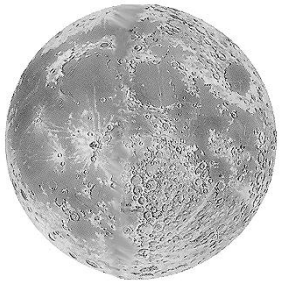
$$\text{acceleration} = \frac{\text{thrust} - g \cdot \text{mass} \cdot \text{gravity}}{\text{mass}}$$

$$\frac{d\,\text{mass}}{dt} = -\,\text{massLossRate} \cdot |\text{thrust}|$$

$$\frac{d\,\text{altitude}}{dt} = \text{velocity}$$

$$\frac{d\,\text{velocity}}{dt} = \text{acceleration}$$

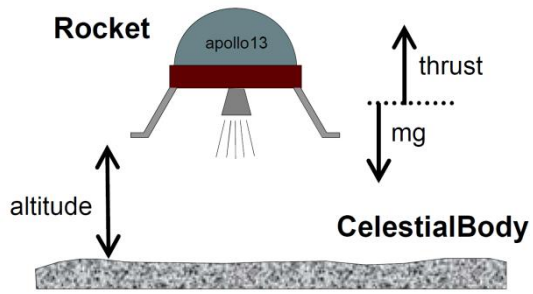
```
class CelestialBody
 constant Real g = 6.672e-11;
 parameter Real radius;
 parameter String name;
 parameter Real mass;
end CelestialBody;
```



From: Peter Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, 1st ed.: Wiley-IEEE Press, 2004

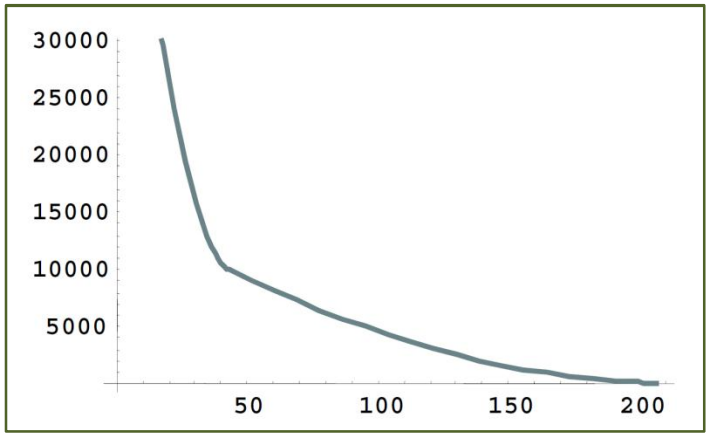
Modelica Background: Landing Simulation

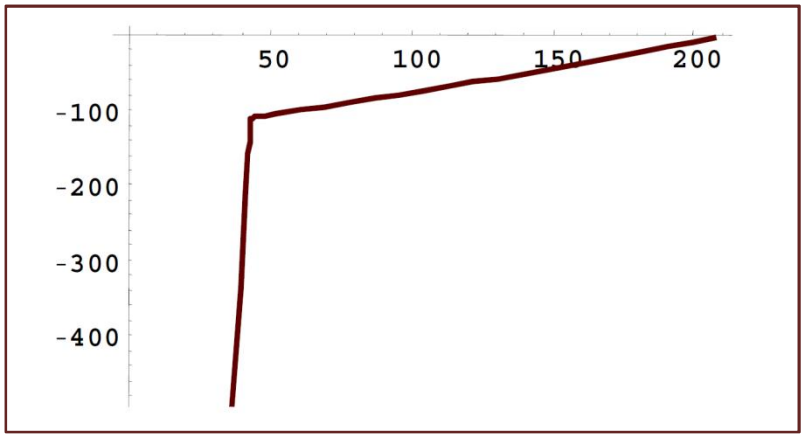
```
class MoonLanding
 parameter Real force1 = 36350;
 parameter Real force2 = 1308;
protected
 parameter Real thrustEndTime = 210;
 parameter Real thrustDecreaseTime = 43.2;
public
 Rocket apollo(name="apollo13");
 CelestialBody moon(name="moon",mass=7.382e22,radius=1.738e6);
equation
  apollo.thrust = if (time < thrustDecreaseTime) then force1
                  else if (time < thrustEndTime) then force2
                  else 0;
apollo.gravity=moon.g*moon.mass/(apollo.altitude+moon.radius)^2;
end MoonLanding;
```



$$\text{apollo.gravity} = \frac{\text{moon.g} * \text{moon.mass}}{(\text{apollo.altitude} + \text{moon.radius})^2}$$

simulate(MoonLanding, stopTime=230)
plot(apollo.altitude, xrange={0,208})
plot(apollo.velocity, xrange={0,208})





ParModelica Language Extension

- Goal – easy-to-use efficient parallel Modelica programming for multi-core execution
- Handwritten code in OpenCL – error prone and needs expert knowledge
- Instead: automatically generating OpenCL code from Modelica with minimal extensions

Why Need ParModelica Language Extensions?

GPUs use their own (different from host) memory for data.

Variables should be explicitly specified for allocation on GPU memory.

OpenCL and CUDA provide multiple memory spaces with different characteristics.

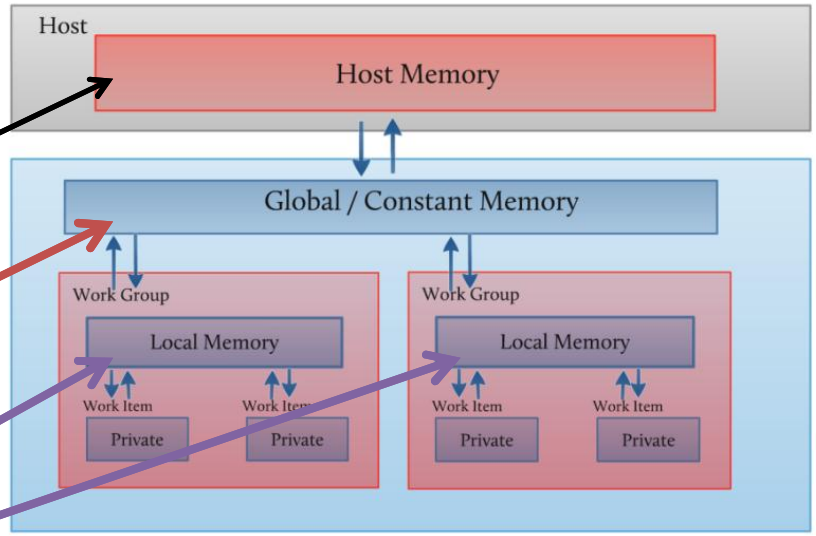
- Global, shared/local, private.

Different variable attributes corresponding to memory space.

Variables in OpenCL Global shared and Local shared memory

ParModelica parglobal and parlocal Variables

Modelica + OpenCL = ParModelica




```
function parvar
  Integer m = 1024;
  Integer A[m];
  Integer B[m];
  parglobal Integer pm;
  parglobal Integer pn;
  parglobal Integer pA[m];
  parglobal Integer pB[m];
  parlocal  Integer ps;
  parlocal  Integer pSS[10];
algorithm
  B := A;                    //copy to device
  pA := A;                   //copy from device
  B := pA;                   //copy device to device
  pB := pA;
  pm := m;
  n := pm;
  pn := pm;
end parvar;
```

Memory Regions	Accessible by
Global Memory	All work-items in all work-groups
Constant Memory	All work-items in all work-groups
Local Memory	All work-items in a work-group
Private Memory	Priavte to a work-item

ParModelica Parallel For-loop: *parfor*

What can be provided now?

- Using only parglobal and parlocal variables

Parallel for-loops

- Parallel for-loops in other languages
  - MATLAB parfor,
  - Visual C++ parallel_for,
  - Mathematica parallelDo,
  - OpenMP omp for (~dynamic scheduling) . . . .

ParModelica

Loop → Kernel

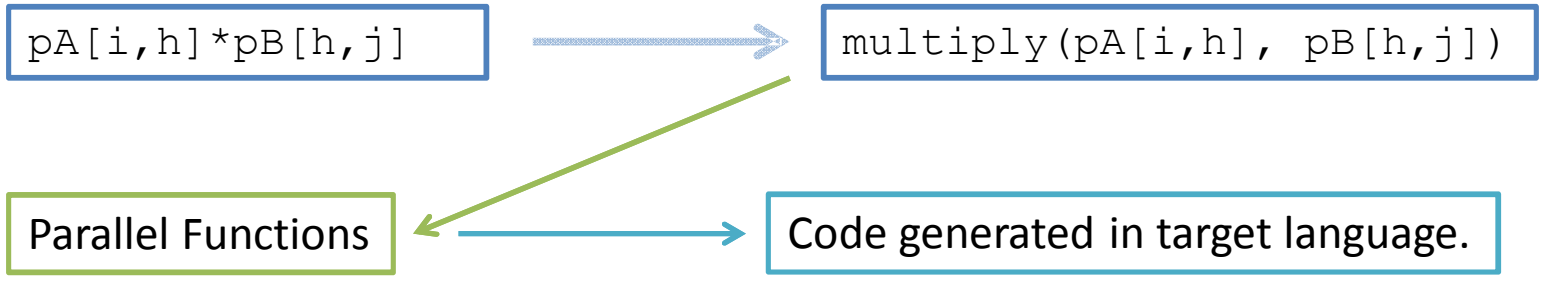
Body → Body

Iterations → Threads

ParModelica Parallel For-loop: *parfor*

```
pA := A;
pB := B;
parfor i in 1:m loop
  for j in 1:pm loop
    ptemp := 0;
    for h in 1:pm loop
      ptemp := pA[i,h]*pB[h,j] + ptemp;
    end for;
    pC[i,j] := ptemp;
  end for;
end parfor;
C := pC;
```

- All variable references in the loop body must be to parallel variables.
- Iterations should not be dependent on other iterations – no loop-carried dependencies.
- All function calls in the body should be to parallel functions or supported Modelica built-in functions only.
- The iterator of a parallel for-loop must be of integer type.
- The start, step and end values of a parallel for-loop iterator should be of integer type.



ParModelica *Parallel Function*

- OpenCL kernel file functions or CUDA __device__ functions.

```
parallel function multiply
  parglobal input Integer a;
  parlocal  input Integer b;
  output Integer c;
algorithm
   c := a * b;
end multiply;
```

OpenCL Work-item functions,

OpenCL Synchronization functions

- They cannot have parallel for-loops in their algorithm.
- They can only call other parallel functions or supported built-in functions.
- Recursion is not allowed.
- They are not directly accessible to serial parts of the algorithm.

ParModelica Parallel For-loops + Parallel Functions

Simple and easy to write.

- No direct control over arrangement and mapping of threads/work-items and blocks/work-groups
  - Suitable only for limited algorithms.
  - Not suitable for thread management.
  - Not suitable for synchronizations.

Kernel Functions

Can be called **directly** from sequential Modelica code.

ParModelica *Kernel Function*

```
oclSetNumThreads(globalSizes,localSizes);
pC := arrayElemWiseMultiply(pm,pA,pB);
```

```
parkernel function arrayElemWiseMultiply
  parglobal input Integer m;
  parglobal input Integer A[:];
  parglobal input Integer B[:];
  parglobal output Integer C[m];
  Integer id;
  parlocal Integer portionId;
algorithm
   id  = oclGetGlobalId(1);
   if(oclGetLocalId(1) == 1) then
     portionId = oclGetGroupId(1);
   end if;
   oclLocalBarrier();
   C[id] := multiply(A[id],B[id], portionId);
end arrayElemWiseMultiply;
```

```
oclSetNumThreads(0);
```

- OpenCL __kernel functions or CUDA __global__ functions.

- Full (up to 3d), work-group and work-item arrangment.
- OpenCL work-item functions supported.
- OpenCL synchronizations are supported.

ParModelica

ParModelica Kernel Functions

ParModelica Kernel functions (vs OpenCL-C):

- Are called the same way as normal functions.

```
pC := arrayElemWiseMultiply(pm,pA,pB);
```

- Can have one or more return or output variables.

```
parglobal output Integer C[m];
```

- Can allocate memory in global memory space (in addition to private and local memory spaces).

```
Integer s;                  //private memory space
parlocal Integer s[m];              //local/shared memory space
Integer s[m] ~ parglobal Integer s[m]; //global memory space
```

  - Allocating small arrays in private memory results in more overhead and information being stored than the necessary.

ParModelica Synchronization and Thread Management

All OpenCL work-item functions supported.

OpenCL		ParModelica
get_work_dim	->	`oclGetWorkDim`
get_local_id	->	`oclGetLocalId`
get_group_id	->	`oclGetGroupId`

. . .

OpenCL work-item functions

Function	Description
get_work_dim	Number of dimensions in use
get_global_size	Number of global work items
get_global_id	Global work item ID
get_local_size	Number of local work items
get_local_id	Local work item ID
get_num_groups	Number of work groups
get_group_id	Work group ID

Vs. OpenCL-C

- Å ids (e.g. oclGetGlobalId) start from 1 instead of from 0:
  - Å To fit Modelica arrays. Modelica arrays start from 1.
- Å Work-group and work-item dimensions start from 1

e.g for N work-items, with one dimensional arrangement

C get_global_id(0) returns 0 to N-1

ParModelica `oclGetGlobalId`(1) returns 0 to N

Modelica PARallel benchmark test suite (MPAR)

Why do we need to have a suitable benchmark test suite?

To evaluate the feasibility and performance of the new language extensions.

Benchmarking and Performance Measurments.

Linear Algebra:
- Matrix Multiplication
- Computation of Eigenvalues

Heat Conduction:
- Stationary Heat Conduction

- Intel(R) Xeon(R) CPU E5520 @ 2.27GHz (16 cores).
- NVIDIA Fermi-Tesla M2050 GPU @ 1.14 GHz (448 cores).

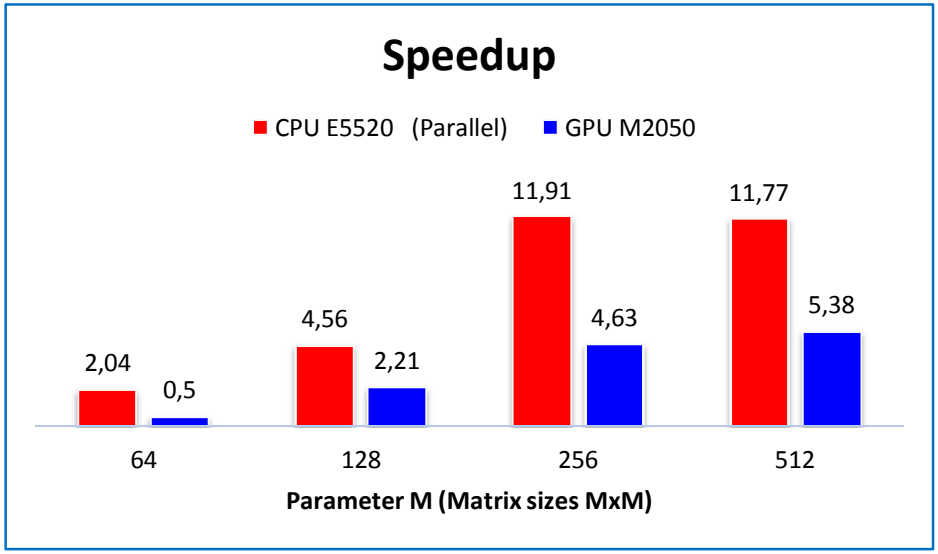
Matrix Multiplication using *parfor*

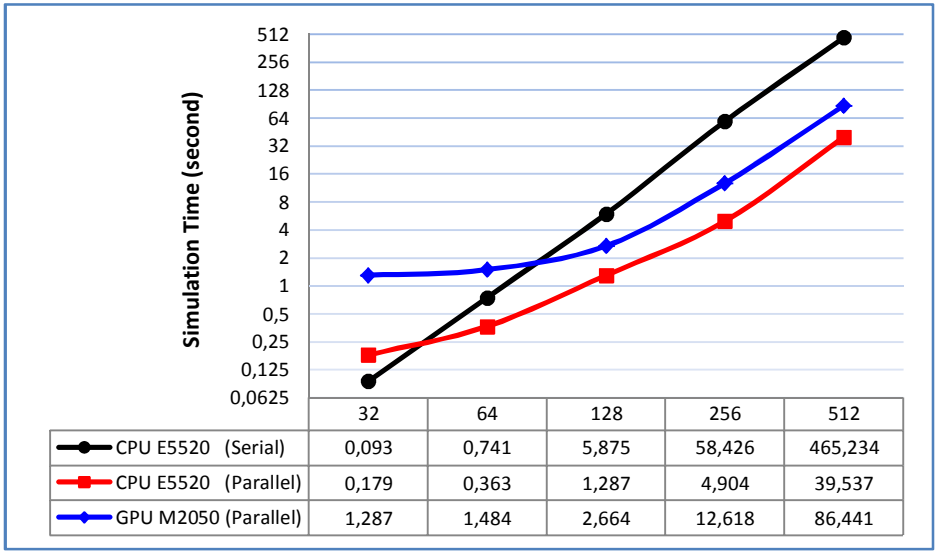
Gained speedup

- Intel Xeon E5520 CPU (16 cores) 12
- NVIDIA Fermi-Tesla M2050 GPU (448 cores) 6

Speedup comparison to sequential algorithm on Intel Xeon E5520 CPU

$$c_{ij} = \sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{p} a_{ik} \cdot b_{kj}$$

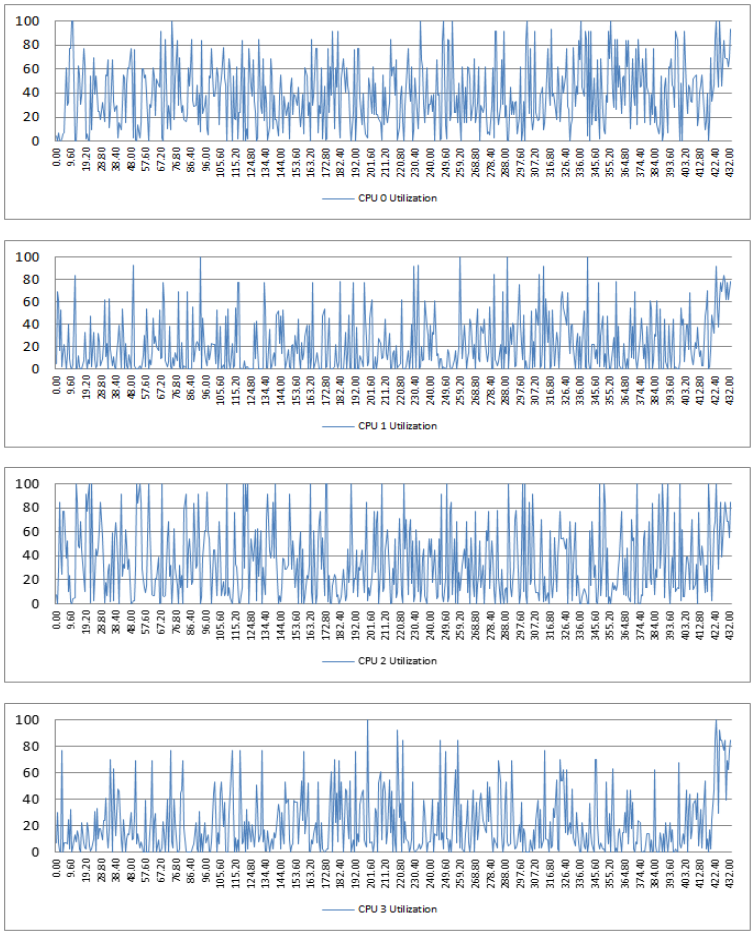




	32	64	128	256	512
CPU E5520 (Serial)	0,093	0,741	5,875	58,426	465,234
CPU E5520 (Parallel)	0,179	0,363	1,287	4,904	39,537
GPU M2050 (Parallel)	1,287	1,484	2,664	12,618	86,441

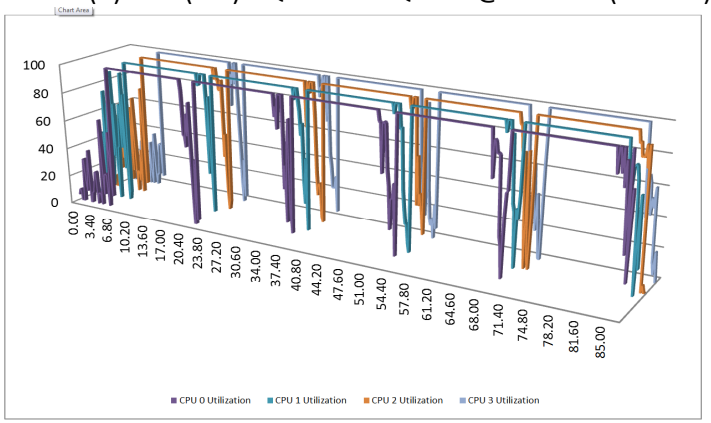
Matrix Multiplication using *parfor*: Core Usages for CPU

Sequential Matrix multiplication

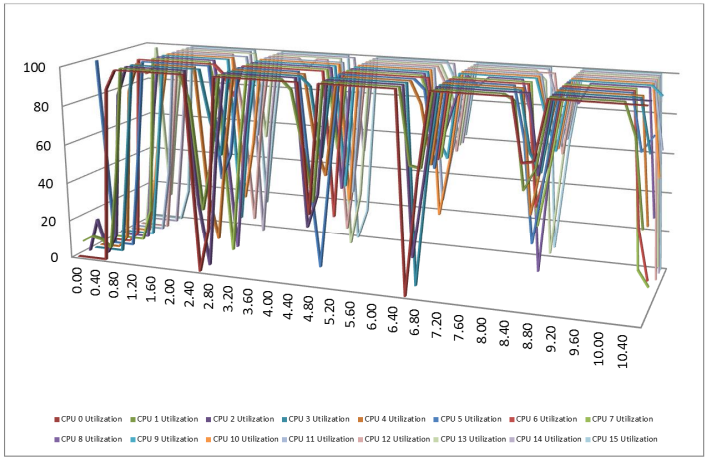


Parallel Matrix multiplication

Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.4 GHz (4 cores)



Intel(R) Xeon(R) CPU E5520 @ 2.27GHz (16 cores)

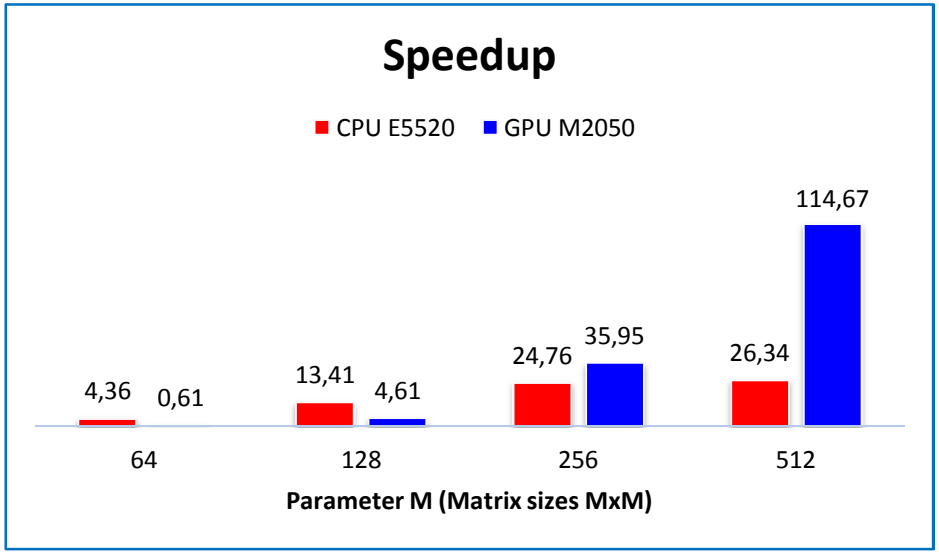


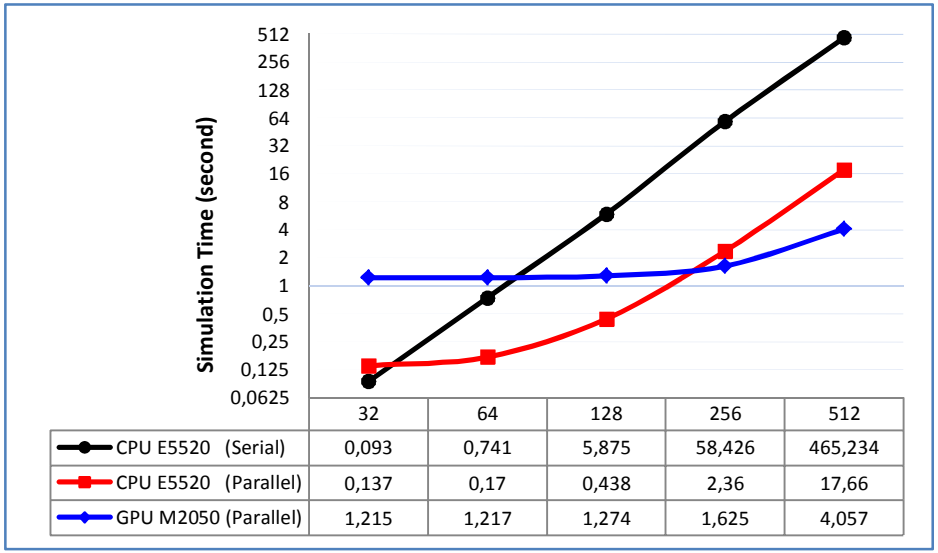
Matrix Multiplication using *Kernel function*

Gained speedup

- Intel Xeon E5520 CPU (16 cores) 26
- NVIDIA Fermi-Tesla M2050 GPU (448 cores) 115

Speedup comparison to sequential algorithm on Intel Xeon E5520 CPU





	32	64	128	256	512
CPU E5520 (Serial)	0,093	0,741	5,875	58,426	465,234
CPU E5520 (Parallel)	0,137	0,17	0,438	2,36	17,66
GPU M2050 (Parallel)	1,215	1,217	1,274	1,625	4,057

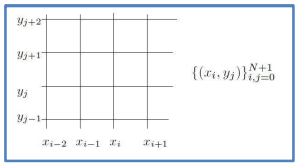
Stationary Heat Conduction

Gained speedup

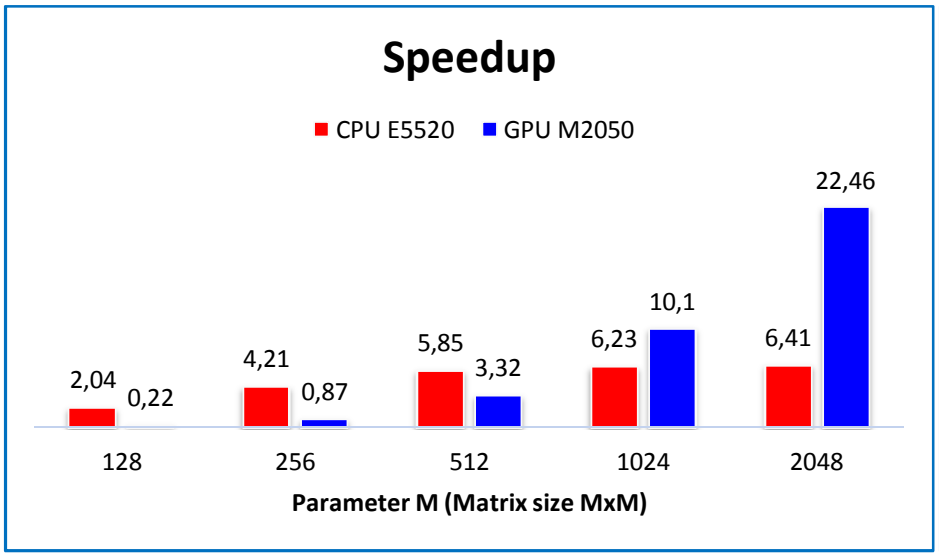
- Intel Xeon E5520 CPU (16 cores) 7
- NVIDIA Fermi-Tesla M2050 GPU (448 cores) 22

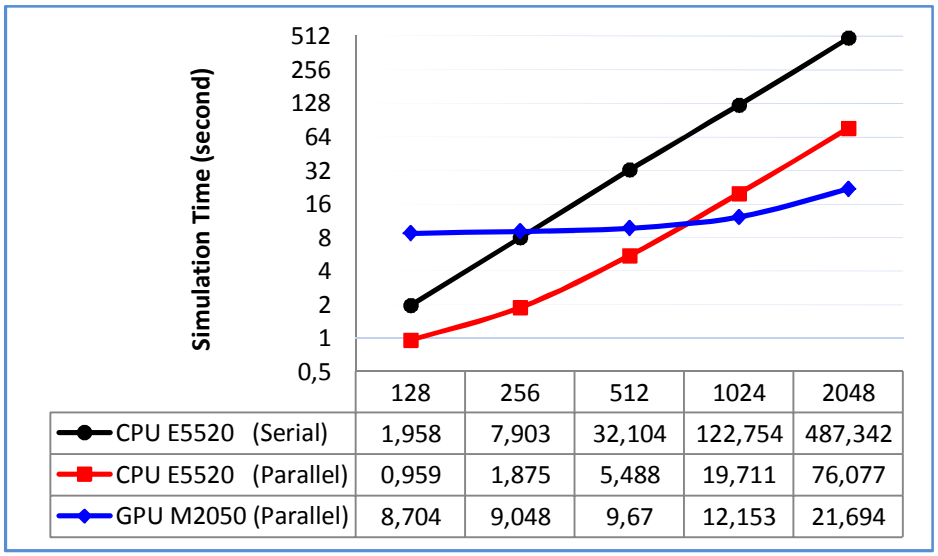
Speedup comparison to sequential algorithm on Intel Xeon E5520 CPU

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$



$$T_{i,j}^{k+1} = (T_{i+1,j}^k + T_{i-1,j}^k + T_{i,j-1}^k + T_{i,j+1}^k)/4, \qquad 1 \le i, j \le N-1$$





	128	256	512	1024	2048
CPU E5520 (Serial)	1,958	7,903	32,104	122,754	487,342
CPU E5520 (Parallel)	0,959	1,875	5,488	19,711	76,077
GPU M2050 (Parallel)	8,704	9,048	9,67	12,153	21,694

Computation of Eigenvalues

Gained speedup

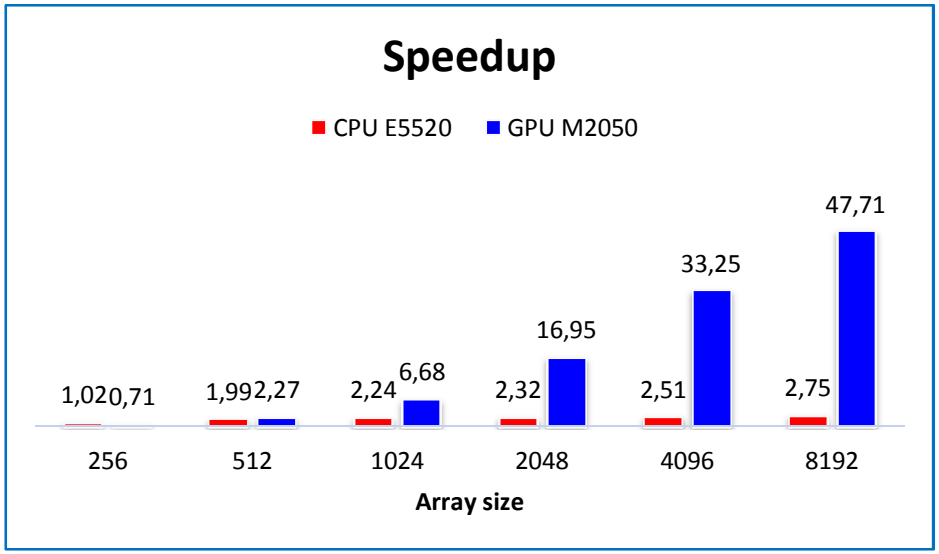
- Intel Xeon E5520 CPU (16 cores) 3
- NVIDIA Fermi-Tesla M2050 GPU (448 cores) 48

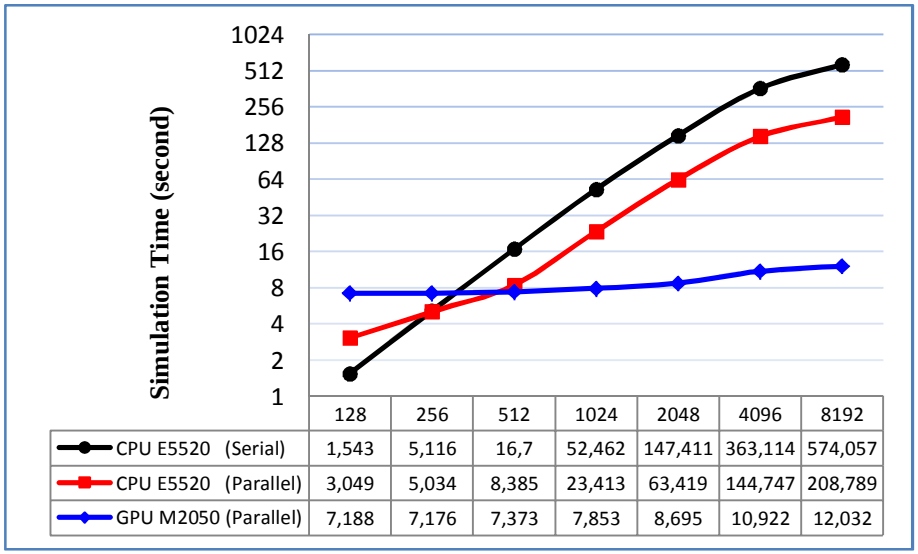
Speedup comparison to sequential algorithm on Intel Xeon E5520 CPU

$$\lambda(A) \subseteq \bigcup_{i=1}^{n} [a_i - r_i, a_i + r_i]$$
$$r_1 = b_1$$
$$r_i = b_i + b_{i-1} \quad 2 \le i \le (n-1)$$
$$r_n = b_{n-1}$$

$$A = \begin{bmatrix} a_{11} & a_{21} & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{32} & \ddots & \vdots \\ 0 & a_{32} & a_{33} & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & a_{nn} \end{bmatrix}$$

Gerschgorin Circle Theorem for Symmetric, Tridiagonal Matrices.





	128	256	512	1024	2048	4096	8192
CPU E5520 (Serial)	1,543	5,116	16,7	52,462	147,411	363,114	574,057
CPU E5520 (Parallel)	3,049	5,034	8,385	23,413	63,419	144,747	208,789
GPU M2050 (Parallel)	7,188	7,176	7,373	7,853	8,695	10,922	12,032

Conclusion

- Easy-to-use high-level parallel programming provided by ParModelica.
- Parallel programming integrated with advanced equation system and object orientation features of Modelica.
- Considerable speedup with the current implementation.
- A benchmark suite for measuring the performance of computationally intensive Modelica models.
- Example algorithms in the benchmark suite help to get started with ParModelica.

Future Work

- CUDA code generation will be supported.
- The current parallel for-loop implementation should be enhanced to provide better control over parallel operations.
- Parallel for-loops can be extended to support OpenMP.
- GPU BLAS routines from AMD and NVIDIA can be incorporated as an option to the current sequential Fortran library routines.
    - APPML and CUBLAS.
- The Benchmark suite will be extended with more parallel algorithms.

Thank you

Questions?