

# A Data-Parallel Algorithmic Modelica Extension for Efficient Execution on Multi-Core Platforms

Mahder Gebremedhin, Afshin Hemmati Moghadam, Peter Fritzson, Kristian Stavåker  
Department of Computer and Information Science  
Linköping University, SE-581 83 Linköping, Sweden  
{mahder.gebremedin, peter.fritzson, Kristian.stavaker}@liu.se, afshe586@student.liu.se

## Abstract

New multi-core CPU and GPU architectures promise high computational power at a low cost if suitable computational algorithms can be developed. However, parallel programming for such architectures is usually non-portable, low-level and error-prone. To make the computational power of new multi-core architectures more easily available to Modelica modelers, we have developed the ParModelica algorithmic language extension to the high-level Modelica modeling language, together with a prototype implementation in the OpenModelica framework. This enables the Modelica modeler to express parallel algorithms directly at the Modelica language level. The generated code is portable between several multi-core architectures since it is based on the OpenCL programming model. The implementation has been evaluated on a benchmark suite containing models with matrix multiplication, Eigen value computation, and stationary heat conduction. Good speedups were obtained for large problem sizes on both multi-core CPUs and GPUs. To our knowledge, this is the first high-performing portable explicit parallel programming extension to Modelica.

*Keywords:* Parallel, Simulation, Benchmarking, Modelica, Compiler, GPU, OpenCL, Multi-Core

## 1 Introduction

Models of large industrial systems are becoming increasingly complex, causing long computation time for simulation. This makes it attractive to investigate methods to use modern multi-core architectures to speedup computations.

Efficient parallel execution of Modelica models has been a research goal of our group for a long time [4], [5], [6], [7], involving improvements both in the compilation process and in the run-time system for parallel execution. Our previous work on compilation of data-parallel models, [7] and [8], has primarily addressed

compilation of purely equation-based Modelica models for simulation on NVIDIA Graphic Processing Units (GPUs). Several parallel architectures have been targeted, such as standard Intel multi-core CPUs, IBM Cell B.E, and NVIDIA GPUs. All the implementation work has been done in the OpenModelica compiler framework [2], which is an open-source implementation of a Modelica compiler, simulator, and development environment. Related research on parallel numeric solvers can for example be found in [9].

The work presented in this paper presents an algorithmic Modelica language extension called ParModelica for efficient portable explicit parallel Modelica programming. Portability is achieved based on the OpenCL [14] standard which is available on several multi-core architectures. ParModelica is evaluated using a benchmark test suite called Modelica PARallel benchmark suite (MPAR) which makes use of these language extensions and includes models which represent heavy computations.

This paper is organized as follows. Section 2 gives a general introduction to Modelica simulation on parallel architectures. Section 3 gives an overview of GPUs, CUDA and OpenCL, whereas the new parallel Modelica language extensions are presented in Section 4. Section 5 briefly describes measurements using the parallel benchmark test suite. Finally, Section 6 gives programming guidelines to use ParModelica, and Section 7 presents conclusions and future work.

## 2 Parallel Simulation of Modelica Models on Multi-Core Computers

The process of compiling and simulating Modelica models to sequential code is described e.g. in [3] and [12]. The handling of equations is rather complex and involves symbolic index reduction, topological sorting according to the causal dependencies between the equations, conversion into assignment statement form, etc. Simulation corresponds to "solving" the compiled

equation system with respect to time using a numerical integration method.

Compiling Modelica models for efficient parallel simulation on multi-core architectures requires additional methods compared to the typical approaches described in [3] and [12]. The parallel methods can be roughly divided into the following three groups:

- *Automatic parallelization of Modelica models.* Several approaches have been investigated: centralized solver approach, distributed solver approach and compilation of unexpanded array equations. With the first approach the solver is run on one core and in each time-step the computation of the equation system is done in parallel over several cores [4]. In the second approach the solver and the equation system are distributed across several cores [5]. With the third approach Modelica models with array equations are compiled unexpanded and simulated on multi-core architectures.
- *Coarse-grained explicit parallelization using components.* Components of the model are simulated in parallel partly de-coupled using time delays between the different components, see [11] for a summary. A different solver, with different time step, etc., can be used for each component. A related approach has been used in the xMOD tool [26].
- *Explicit parallel programming language constructs.* This approach is explored in the NestStepModelica prototype [10] and in this paper with the ParModelica language extension. Parallel extensions have been developed for other languages, e.g. parfor loop and gpu arrays in Matlab, Visual C++ parallel\_for, Mathematica parallelDo, etc.

### 3 GPU Architectures, CUDA, and OpenCL

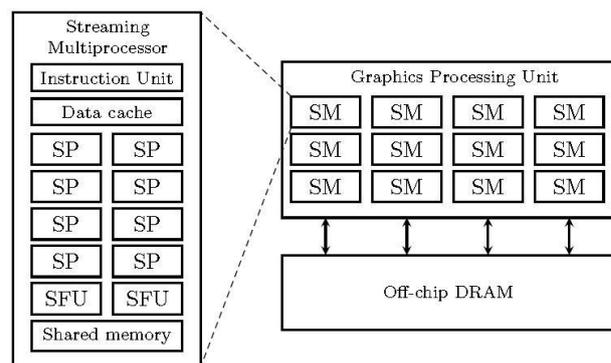
Graphics Processing Units (GPUs) have recently become increasingly programmable and applicable to general purpose numeric computing. The theoretical processing power of GPUs has in recent years far surpassed that of CPUs due to the highly parallel computing approach of GPUs.

However, to get good performance, GPU architectures should be used for simulation of models of a regular structure with large numbers of similar data objects. The computations related to each data object can then be executed in parallel, one or more data objects on each core, so-called data-parallel computing. It is also very important to use the GPU memory hierarchy effectively in order to get good performance.

In Section 3.1 the NVIDIA GPU with its CUDA programming model is presented as an influential example of GPU architecture, followed by the portable OpenCL parallel programming model in Section 3.2.

#### 3.1 NVIDIA GPU CUDA – Compute Unified Device Architecture

An important concept in NVIDIA CUDA (Computer Unified Device Architecture) for GPU programming is the distinction between host and device. The host is what executes normal programs, and the device works as a coprocessor to the host which runs CUDA threads by instruction from the host. This typically means that a CPU is the host and a GPU is the device, but it is also possible to debug CUDA programs by using the CPU as both host and device. The host and the device are assumed to have their own separate address spaces, the host memory and the device memory. The host can use the CUDA runtime API to control the device, for example to allocate memory on the device and to transfer memory to and from the device.



**Figure 1.** Simplified schematic of NVIDIA GPU architecture, consisting of a set of Streaming Multiprocessors (SM), each containing a number of Scalar Processors (SP) with fast private memory and on-chip local shared memory. The GPU also has off-chip DRAM.

The building block of the NVIDIA CUDA hardware architecture is the *Streaming Multiprocessor* (SM). In the NVIDIA Fermi-Tesla M2050 GPU, each SM contains 32 *Scalar Processors* (SPs). The entire GPU has 14 such SMs totaling to 448 SPs, as well as some off-chip DRAM memory, see Figure 1. This gives a scalable architecture where the performance of the GPU can be varied by having more or fewer SMs.

To be able to take advantage of this architecture a program meant to run on the GPU, known as a *kernel*, needs to be massively multi-threaded. A kernel is just a C-function meant to execute on the GPU. When a kernel is executed on the GPU it is divided into *thread blocks*, where each thread block contains an equal number of threads. These thread blocks are automatically distributed among the SMs, so a programmer

need not consider the number of SMs a certain GPU has. All threads execute one common instruction at a time. If any threads take divergent execution paths, then each of these paths will be executed separately, and the threads will then converge again when all paths have been executed. This means that some SPs will be idle if the thread executions diverge. It is thus important that all threads agree on an execution path for optimal performance.

This architecture is similar to the *Single Instruction, Multiple Data* (SIMD) architecture that vector processors use, and that most modern general-purpose CPUs have limited capabilities for too. NVIDIA call this architecture *Single Instruction, Multiple Thread* (SIMT) instead, the difference being that each thread can execute independently, although at the cost of reduced performance. It is also possible to regard each SM as a separate processor, which enables Multiple Instructions, Multiple Data (MIMD) parallelism. Using only MIMD parallelism will not make it possible to take full advantage of a GPU's power, since each SM is a SIMD processor. To summarize:

- Streaming Multiprocessors (SM) can work with different code, performing different operations with entirely different data (MIMD execution, Multiple Instruction Multiple Data).
- All Scalar processors (SP) in one streaming multiprocessor execute the same instruction at the same time but work on different data (SIMT/SIMD execution, Single Instruction Multiple Data).

### 3.1.1 NVIDIA GPU Memory Hierarchy

As can be seen in Figure 1 there are several different types of memory in the CUDA hardware architecture. At the lowest level each SP has a set of registers, the number depending on the GPU's capabilities. These registers are shared between all threads allocated to a SM, so the number of thread blocks that a SM can have active at the same time is limited by the register usage of each thread. Accessing a register typically requires no extra clock cycles per instruction, except for some special cases where delays may occur.

Besides the registers there is also the shared (local) memory, which is shared by all SPs in a SM. The shared memory is implemented as fast on-chip memory, and accessing the shared memory is generally as fast as accessing a register. Since the shared memory is accessible to all threads in a block it allows the threads to cooperate efficiently by giving them fast access to the same data.

Most of the GPU memory is off-chip Dynamic Random Access Memory (DRAM). The amount of off-

chip memory on modern graphics cards range from several hundred megabytes to few gigabytes. The DRAM memory is much slower than the on-chip memories, and is also the only memory that is accessible to the host CPU, e.g. through DMA transfers. To summarize:

- Each scalar processor (SP) has a set of fast registers. (private memory)
- Each streaming multiprocessor (SM) has a small local shared memory (48KB on Tesla M2050 ) with relatively fast access.
- Each GPU device has a slower off-chip DRAM (2GB on Tesla M2050) which is accessible from all streaming multiprocessors and externally e.g. from the CPU with DMA transfers.

### 3.2 OpenCL – the Open Computing Language

OpenCL [14] is the first open, free parallel computing standard for cross-platform parallel programming of modern processors including GPUs. The OpenCL programming language is based on C99 with some extensions for parallel execution management. By using OpenCL it is possible to write parallel algorithms that can be easily ported between multiple devices with minimal or no changes to the source code.

The OpenCL framework consists of the OpenCL programming language, API, libraries, and a runtime system to support software development. The framework can be divided into a hierarchy of models: Platform Model, Memory model, Execution model, and Programming model.

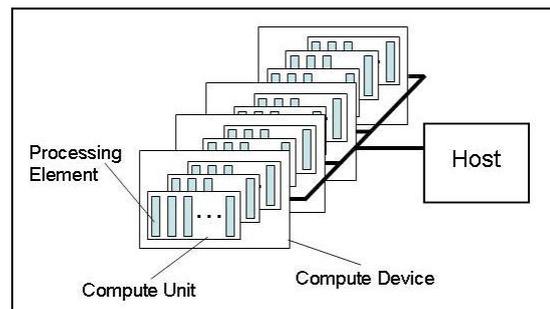


Figure 2. OpenCL platform architecture.

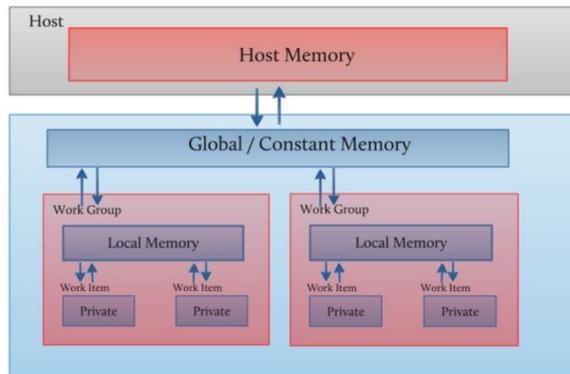
The OpenCL platform architecture in Figure 2 is similar to the NVIDIA CUDA architecture in Figure 1:

- Compute device – Graphics Processing Unit (GPU)
- Compute unit – Streaming Multiprocessor (SM)
- Processing element – Scalar Processor (SP)
- Work-item – thread
- Work-group – thread block

The memory hierarchy (Figure 3) is also very similar:

- Global memory – GPU off-chip DRAM memory

- Constant memory – read-only cache of off-chip memory
- Local memory – on-chip shared memory that can be accessed by threads in the same SM
- Private memory – on-chip registers in the same



**Figure 3.** Memory hierarchy in the OpenCL memory model, closely related to typical GPU architectures such as NVIDIA.

The memory regions can be accessed in the following way:

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

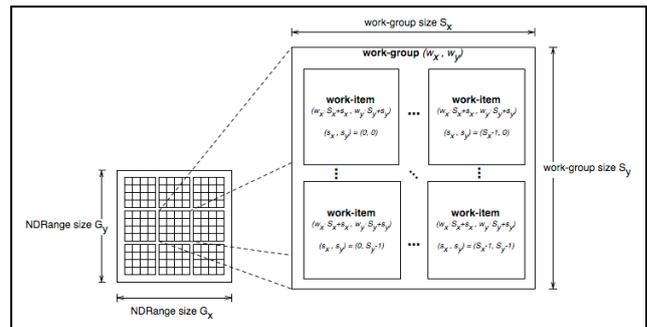
### 3.2.1 OpenCL Execution Model

The execution of an OpenCL program consists of two parts, the *host program* which executes on the host and the parallel *OpenCL program*, i.e., a collection of *kernels* (also called kernel functions), which execute on the OpenCL device. The host program manages the execution of the OpenCL program.

*Kernels* are executed simultaneously by all threads specified for the kernel execution. The number and mapping of threads to Computing Units of the OpenCL device is handled by the host program.

Each *thread* executing an instance of a kernel is called a *work-item*. Each thread or work item has *unique id* to help identify it. Work items can have additional id fields depending on the arrangement specified by the host program.

Work-items can be arranged into *work-groups*. Each work-group has a unique ID. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID.



**Figure 4.** OpenCL execution model, work-groups depicted as groups of squares corresponding to work-items. Each work-group can be referred to by a unique ID, and each work-item by a unique local ID.

The work-items in a given work-group execute concurrently on the processing elements of a single compute unit as depicted in Figure 4.

Several programming models can be mapped onto this execution model. OpenCL explicitly supports two of these models: primarily the data parallel programming model, but also the task parallel programming model

## 4 ParModelica: Extending Modelica for Explicit Algorithmic Parallel Programming

As mentioned in the introduction, the focus of the current work is an extension (ParModelica) of the algorithmic subset of Modelica for efficient explicit parallel programming on highly data-parallel SPMD (Single Program Multiple Data) architectures. The current ParModelica implementation generates OpenCL [14] code for parallel algorithms. OpenCL was selected instead of CUDA [15] because of its portability between several multi-core platforms. Generating OpenCL code ensures that simulations can be run with parallel support on OpenCL enabled Graphics and Central Processor Units (GPUs and CPUs). This includes many multi-core CPUs from [19] and Advanced Micro Devices (AMD) [18] as well as a range of GPUs from NVIDIA [17] and AMD [18].

As mentioned earlier most previous work regarding parallel execution support in the OpenModelica compiler has been focused on automatic parallelization where the burden of finding and analyzing parallelism has been put on the compiler. In this work, however, we have decided to leave this responsibility to the end user programmer. The compiler provides additional high level language constructs needed for explicitly stating parallelism in the algorithmic part of the modeling language. These, among others, include parallel variables, parallel functions, kernel functions and paral-

lel for loops indicated by the parfor keyword. There are also some target language specific constructs and functions (in this case related to OpenCL).

#### 4.1 Parallel Variables

OpenCL code can be executed on a host CPU as well as on GPUs whereas CUDA code executes only on GPUs. Since the OpenCL and CUDA enabled GPUs use their own local (different from CPU) memory for execution, all necessary data should be copied to the specific device's memory. Parallel variables are allocated on the specific device memory instead of the host CPU. An example is shown below:

```
function parvar
protected
  Integer m = 1000;          // Host Scalar
  Integer A[m,m];          // Host Matrix
  Integer B[m,m];          // Host Matrix
// global and local device memories
  parglobal Integer pm;    // Global Scalar
  parglobal Integer pA[m,m]; // Glob Matrix
  parglobal Integer pB[m,m]; // Glob Matrix
  parlocal Integer pn;    // Local Scalar
  parlocal Integer pS[m]; // Local Array
end parvar;
```

The first two matrices A and B are allocated in normal host memory. The next two matrices pA and pB are allocated on the global memory space of the OpenCL device to be used for execution. These global variables can be initialized from normal or host variables. The last array pS is allocated in the local memory space of each processor on the OpenCL device. These variables are shared between threads in a single work-group and cannot be initialized from host variables.

*Copying of data* between the host memory and the device memory used for parallel execution is as simple as *assigning the variables* to each other. The compiler and the runtime system handle the details of the operation. The assignments below are all valid in the function given above

- Normal assignment -  $A := B$
- Copy from host memory to parallel execution device memory -  $pA := A$
- Copy from parallel execution device memory to host memory -  $B := pB$
- Copy from device memory to other device memory -  $pA := pB$

Modelica parallel arrays are passed to functions only by reference. This is done to reduce the rather expensive copy operations.

#### 4.2 Parallel Functions

ParModelica *parallel functions* correspond to OpenCL functions *defined in kernel files* or to CUDA device

functions. These are functions available for distributed (parallel) independent execution in each thread executing on the parallel device. For example, if a parallel array has been distributed with one element in each thread, a parallel function may operate locally in parallel on each element. However, unlike kernel functions, *parallel functions cannot be called from serial code* in normal Modelica functions on the host computer just as parallel OpenCL functions are not allowed to be called from serial C code on the host. Parallel functions have the following constraints, primarily since they are assumed to be called within a parallel context in work-items:

- Parallel function bodies may *not* contain parfor-loops. The reason is that the kernel containing the parallel functions is already distributed on each thread.
- Explicitly declared parallel variables are not allowed since execution is already taking place on the parallel device.
- All memory allocation will be on the parallel device's memory.
- Nested parallelism as in NestStepModelica [10] is not supported by this implementation.
- Called functions must be parallel functions or supported built-in functions since execution is on the parallel device.
- Parallel functions can only be called from the body of a parfor-loop, from parallel functions, or from kernel functions.

Parallel functions in ParModelica are defined in the same way as normal Modelica functions, except that they are preceded by the `parallel` keyword as in the multiply function below:

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

#### 4.3 Kernel Functions

ParModelica *kernel functions* correspond to OpenCL kernel functions [14] or CUDA global functions [16]. They are simply functions compiled to execute on an OpenCL parallel device, typically a GPU. ParModelica kernel functions are allowed to have several return- or output variables unlike their OpenCL or CUDA counterparts. They can also allocate memory in the global address space. Kernel functions can be called from serial host code, and are executed by each thread in the

launch of the kernel. Kernel functions share the first three constraints stated above for parallel functions.

However, unlike parallel functions, kernel functions *cannot* be called from the body of a parfor-loop or from other kernel functions.

Kernel functions in ParModelica are defined in the same way as normal Modelica functions, except that they are preceded by the `kernel` keyword. An example usage of kernel functions is shown by the kernel function `arrayElemWiseMult`. The thread id function `oclGetGlobalId()` (see Section 4.5) returns the integer id of a work-item in the first dimension of a work group.

```
kernel function arrayElemWiseMultiply
input Integer m;
input Integer A[m];
input Integer B[m];
output Integer C[m];
protected
Integer id;
algorithm
id := oclGetGlobalId(1);
// calling the parallel function
multiply is OK from kernel functions
C[id] := multiply(A[id],B[id]); //
multiply can be replaced by A[id]*B[id]
end arrayElemWiseMultiply;
```

#### 4.4 Parallel For Loop: parfor

The iterations of a ParModelica parfor-loop are executed without any specific order in *parallel* and *independently* by multiple threads. The iterations of a parfor-loop are equally distributed among available processing units. If the range of the iteration is smaller than or equal to the number of threads the parallel device supports, each iteration will be done by a separate thread. If the number of iterations is larger than the number of threads available, some threads might perform more than one iteration. In future enhancements parfor will be given the extra feature of specifying the desired number of threads explicitly instead of automatically launching threads as described above. An example of using the parfor-loop is shown below:

```
// Matrix multiplication using parfor loop
parfor i in 1:m loop
for j in 1:pm loop
ptemp := 0;
for h in 1:pm loop // calling the
// parallel function multiply is OK
// from parfor-loops
ptemp := multiply(pA[i,h], pB[h,j])
+ ptemp;
end for;
pC[i,j] := ptemp;
end for;
end parfor;
```

ParModelica parallel for loops, compared to normal Modelica for loops, have some additional constraints:

- All variable references in the loop body must be to parallel variables.
- Iterations should not be dependent on other iterations i.e. no loop-carried dependencies.
- All function calls in the body should be to parallel functions or supported built-in functions only.

#### 4.5 Executing User-written OpenCL Code from ParModelica.

There are also some additional ParModelica features available for directly compiling and executing user-written OpenCL code:

- `oclbuild(String)` takes a name of an OpenCL source file and builds it. It returns an OpenCL program object which can be used later.
- `oclkernel(oclprogram, String)` takes a previously built OpenCL program and create the kernel specified by the second argument. It returns an OpenCL kernel object which can be used later.
- `oclsetargs(oclkernel,...)` takes a previously created kernel object variable and a variable number of arguments and sets each argument to its corresponding one in the kernel definition.
- `oclexecute(oclkernel)` executes the specified kernel.

All of the above operations are synchronous in the OpenCL jargon. They will return only when the specified operation is completed. Further functionality is planned to be added to these functions to provide better control over execution.

#### 4.6 Synchronization and Thread Management

All OpenCL work-item functions [20] are available in ParModelica. They perform the same operations and have the “same” types and number of arguments. However, there are two main differences:

- Thread/work-item index ids start from 1 in ParModelica, whereas the OpenCL C implementation counts from 0.
- Array dimensions start from 1 in Modelica and from 0 in OpenCL and C.

For example `oclGetGlobalId(1)` call in the above `arrayElemWiseMultiply` will return the integer ID of a work-item or thread in the first dimension of a work group. The first thread gets an ID of 1. The OpenCL C call for the same operation would be `ocl_get_global_id(0)` with the first thread obtaining an ID of 0.

In addition to the above features, special built-in functions for building user written OpenCL code directly from source code, creating a kernel, setting arguments to kernel and execution of kernels are also available. In addition parallel versions of some built-in algorithm functions are also available.

## 5 Benchmarking and Evaluation

To be able to evaluate the relative performance and behavior of the new language extensions described in Section 4, performing systematic benchmarking on a set of appropriate Modelica models is required. For this purpose we have constructed a benchmark test suite containing some models that represent heavy and high-performance computation, relevant for simulation on parallel architectures.

### 5.1 The MPAR Benchmark Suite

The MPAR benchmark test suite contains seven different algorithms from well-known benchmark applications such as the LINear equations software PACKage (LINPACK) [21], and Heat Conduction [23]. These benchmarks have been collected and implemented as algorithmic time-independent Modelica models.

The algorithms implemented in this suite involve rather large computations and impose well defined workloads on the OpenModelica compiler and the run-time system. Moreover, they include different kinds of for-loops and function calls which provide parallelism for domain and task decomposition. For space reasons we have provided results for only three models here.

Time measurements have been performed of both sequential and parallel implementations of three models: Matrix Multiplication, Eigen value computation, and Stationary Heat Conduction, on both CPU and GPU architectures. For executing sequential codes generated by the standard sequential OpenModelica compiler we have used the Intel Xeon E5520 CPU [24] which has 16 cores, each with 2.27 GHz clock frequency. For executing generated code by our new OpenCL based parallel code generator, we have used the same CPU as well as the NVIDIA Fermi-Tesla M2050 GPU [25].

### 5.2 Measurements

In this section we present the result of measurements for simulating three models from the implemented benchmark suite. On each hardware configuration all simulations are performed five times with start time 0.0, stop time of 0.2 seconds and 0.2 seconds time step, measuring the average simulation time using the `clock_gettime()` function from the C standard li-

brary. This function is called once when the simulation loop starts and once when the simulation loop finishes. The difference between the returned values gives the simulation time.

All benchmarks have been simulated on both the Intel Xeon E5520 CPU (16 cores) and the NVIDIA Fermi-Tesla M2050 GPU (448 cores).

### 5.3 Simulation Results

The Matrix Multiplication model (Appendix A) produces an  $M \times K$  matrix  $C$  from multiplying an  $M \times N$  matrix  $A$  by an  $N \times K$  matrix  $B$ . This model presents a very large level of data-parallelism for which a considerable speedup has been achieved as a result of parallel simulation of this model on parallel platforms. The simulation results are illustrated in Figure 5 and Figure 6. The obtained speedup of matrix multiplication using kernel functions is as follows compared to the sequential algorithm on Intel Xeon E5520 CPU:

- Intel 16-core CPU – speedup 26
- NVIDIA 448-core GPU – speedup 115

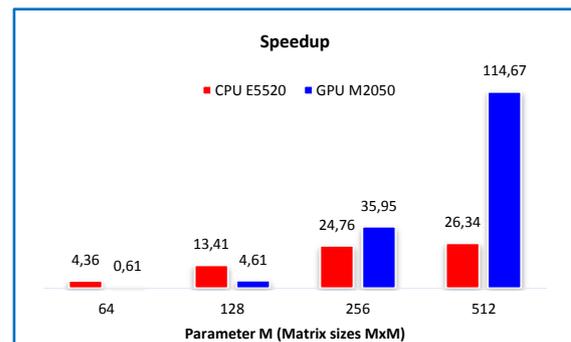


Figure 5. Speedup for matrix multiplication, Intel 16-core CPU and Nvidia 448 core GPU.

The measured matrix multiplication model simulation times can be found in Figure 6.

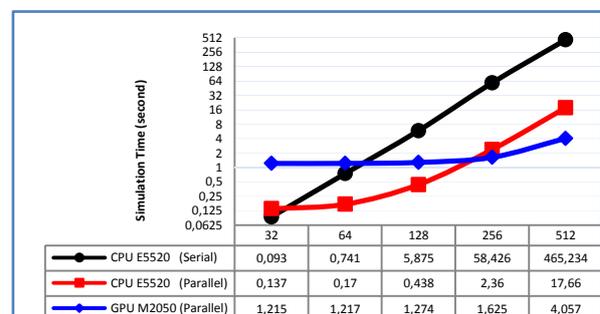
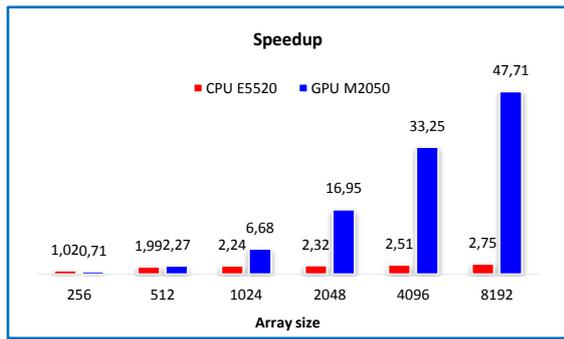


Figure 6. Simulation time for matrix multiplication, Intel 1-core, 16-core CPU, Nvidia 448 core GPU.

The second benchmark model performs Eigen-value computation, with the following speedups:

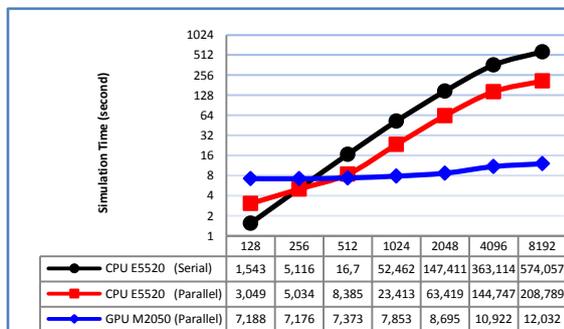
- Intel 16-core CPU – speedup 3

- NVIDIA 448-core GPU – speedup 48



**Figure 7.** Speedup for Eigen value computation as a function of model array size, for Intel 16-core CPU and NVIDIA 448 core GPU, compared to the sequential algorithm on Intel Xeon E5520 CPU.

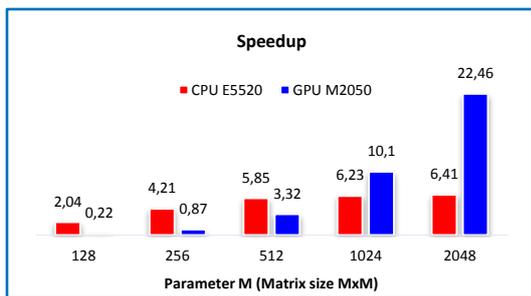
The measured simulation times for the Eigen-value model are shown in Figure 8.



**Figure 8.** Simulation time for Eigen-value computation as a function of model array size, for Intel 1-core CPU, 16-core CPU, and NVIDIA 448 core GPU.

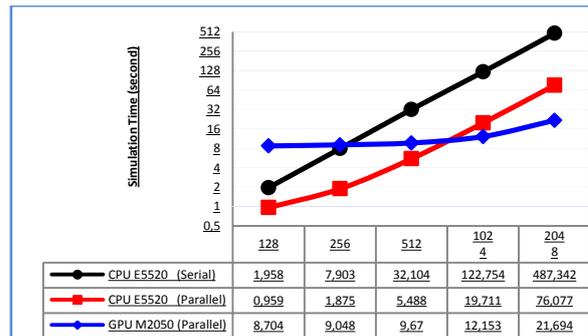
The third benchmark model computes stationary heat conduction, with the following speedups:

- Intel 16-core CPU – speedup 7
- NVIDIA 448-core GPU – speedup 22



**Figure 9.** Speedup for the heat conduction model as a function of model size parameter M, Intel 16-core CPU and Nvidia 448 core GPU, compared to sequential algorithm on Intel Xeon E5520 CPU.

The measured simulation times for the stationary heat conduction model are shown in Figure 10.



**Figure 10.** Simulation time (seconds) for heat conduction model as a function of model size parameter M, for 1-core CPU, 16-core CPU, and 448 core GPU.

According to the results of our measurements illustrated in Figure 5, Figure 7, and Figure 9, absolute speedups of 114, 48, and 22 respectively were achieved when running generated ParModelica OpenCL code on the Fermi-Tesla M2050 GPU compared to serial code on the Intel Xeon E5520 CPU with the largest data sizes.

It should be noted that when the problem size is not very large the sequential execution has better performance than the parallel execution. This is not surprising since for executing even a simple code on OpenCL devices it is required to create an OpenCL context within those devices, allocate OpenCL memory objects, transfer input data from host to those memory objects, perform computations, and finally transfer back the result to the host. Consequently, performing all these operations normally takes more time compared to the sequential execution when the problem size is small.

It can also be seen that, as the sizes of the models increase, the simulations get better relative performance on the GPU compared to multi-core CPU. Thus, to fully utilize the power of parallelism using GPUs it is required to have large regular data structures which can be operated on simultaneously by being decomposed to all blocks and threads available on GPU. Otherwise, executing parallel codes on a multi-core CPU would be a better choice than a GPU to achieve more efficiency and speedup.

## 6 Guidelines for Using the New Parallel Language Constructs

The most important task in all approaches regarding parallel code generation is to provide an appropriate way for analyzing and finding parallelism in sequential codes. In automatic parallelization approaches, the whole burden of this task is on the compiler and tool developer. However, in explicit parallelization approaches as in this paper, it is the responsibility of the modeler to analyze the source code and define which

parts of the code are more appropriate to be explicitly parallelized. This requires a good understanding of the concepts of parallelism to avoid inefficient and incorrect generated code. In addition, it is necessary to know the constraints and limitations involved with using explicit parallel language constructs to avoid compile time errors. Therefore we give some advice on how to use the ParModelica language extensions to parallelize Modelica models efficiently:

- Try to declare parallel variables as well as copy assignments among normal and parallel variables as less as possible since the costs of data transfers from host to devices and vice versa are very expensive.
- In order to minimize the number of parallel variables as well as data transfers between host and devices, it is better not to convert for-loops with few iterations over simple operations to parallel for-loops (parfor-loops).
- It is not always useful to have parallel variables and parfor-loops in the body of a normal for-loop which has many iterations. Especially in cases where there are many copy assignments among normal and parallel variables.
- Although it is possible to declare parallel variables and also parfor-loops in a function, there are no advantages when there are many calls to the function (especially in the body of a big for-loop). This will increase the number of memory allocations for parallel variables as well as the number of expensive copies required to transfer data between host and devices.
- Do not directly convert a for-loop to a parfor-loop when the result of each iteration depends on other iterations. In this case, although the compiler will correctly generate parallel code for the loop, the result of the computation may be incorrect.
- Use a parfor-loop in situations where the loop has many independent iterations and each iteration takes a long time to be completed.
- Try to parallelize models using kernel functions as much as possible rather than using parfor-loops. This will enable you to explicitly specify the desired number of threads and work-groups to get the best performance.
- If the global work size (total number of threads to be run in parallel) and the local work size (total number of threads in each work-group) need to be specified explicitly, then the following points should be considered. First, the work-group size (local size) should not be zero, and also it should not exceed the maximum work-group size supported by the parallel device. Second, the local size should be less or equal than the global-size. Third, the

global size should be evenly divisible by the local size.

- The current implementation of OpenCL does not support recursive functions; therefore it is not possible to declare a recursive function as a parallel function.

## 7 Conclusions

New multi-core CPU and GPU architectures promise high computational power at a low cost if suitable computational algorithms can be developed. The OpenCL C-based parallel programming model provides a way of writing portable parallel algorithms that perform well on a number of multi-core architectures. However, the OpenCL programming model is rather low-level and error-prone to use and intended for parallel programming specialists.

This paper presents the ParModelica algorithmic language extension to the high-level Modelica modeling language together with a prototype implementation in the OpenModelica compiler. This makes it possible for the Modelica modeler to directly write efficient parallel algorithms in Modelica which are automatically compiled to efficient low-level OpenCL code. A benchmark suite called MPAR has been developed to evaluate the prototype. Good speedups have been obtained for large problem sizes of matrix multiplication, Eigen value computation, and stationary heat condition.

Future work includes integration of the ParModelica explicit parallel programming approach with automatic and semi-automatic approaches for compilation of equation-based Modelica models to parallel code. Autotuning could be applied to further increase the performance and automatically adapt it to varying problem configurations. Some of the ParModelica code needed to specify kernel functions could be automatically generated.

## 8 Acknowledgements

This work has been supported by Serc, by Ellit, by the Swedish Strategic Research Foundation in the EDOp and HIPO projects and by Vinnova in the RTSIM and ITEA2 OPENPROD projects. The Open Source Modelica Consortium supports the OpenModelica work. Thanks to Per Östlund for contributions to Section 3.1.

## References

- [1] Modelica Association. *The Modelica Language Specification Version 3.2*, March 24th 2010. <http://www.modelica.org>. Modelica Association.

- Modelica Standard Library 3.1. Aug. 2009. <http://www.modelica.org/>
- [2] Open Source Modelica Consortium. OpenModelica System Documentation Version 1.8.1, April 2012. <http://www.openmodelica.org/>
- [3] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2004.
- [4] Peter Aronsson. *Automatic Parallelization of Equation-Based Simulation Programs*, PhD thesis, Dissertation No. 1022, Linköping University, 2006.
- [5] Håkan Lundvall. *Automatic Parallelization using Pipelining for Equation-Based Simulation Languages*, Licentiate thesis No. 1381, Linköping University, 2008.
- [6] Håkan Lundvall, Kristian Stavåker, Peter Fritzson, Christoph Kessler: Automatic Parallelization of Simulation Code for Equation-based Models with Software Pipelining and Measurements on Three Platforms. MCC'08 Workshop, Ronneby, Sweden, November 27-28, 2008.
- [7] Per Östlund. *Simulation of Modelica Models on the CUDA Architecture*. Master Thesis. LIU-IDA/LITH-EX-A{09/062}{SE}. Linköping University, 2009.
- [8] Kristian Stavåker, Peter Fritzson. Generation of Simulation Code from Equation-Based Models for Execution on CUDA-Enabled GPUs. MCC'10 Workshop, Gothenburg, Sweden, November 18-19, 2010.
- [9] Matthias Korch and Thomas Rauber. Scalable parallel rk solvers for odes derived by the method of lines. In Harald Kosch, Laszlo Böszörményi, and Hermann Hellwagner, editors, Euro-Par, volume 2790 of Lecture Notes in Computer Science, pages 830-839. Springer, 2003.
- [10] Christoph Kessler and Peter Fritzson. NestStep-Modelica – Mathematical Modeling and Bulk-Synchronous Parallel Simulation. In Proc. of PARA'06, Umeå, June 19-20, 2006. In Lecture Notes of Computer Science (LNCS) Vol 4699, pp 1006-1015, Springer Verlag, 2006.
- [11] Martin Sjölund, Robert Braun, Peter Fritzson and Petter Krus. Towards Efficient Distributed Simulation in Modelica using Transmission Line Modeling. In *Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, (EOOLT'2010), Published by Linköping University Electronic Press, [www.ep.liu.se](http://www.ep.liu.se), In conjunction with MOD-ELS'2010, Oslo, Norway, Oct 3, 2010.
- [12] Francois Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer, 2006.
- [13] Khronos Group, Open Standards for Media Authoring and Acceleration, OpenCL 1.1, accessed Sept 15, 2011. <http://www.khronos.org/opencl/>
- [14] The OpenCL Specication, Version: 1.1, Document Revision: 44, accessed June 30 2011. <http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf>
- [15] NVIDIA CUDA, accessed September 15 2011. [http://www.nvidia.com/object/cuda\\_home\\_new.html](http://www.nvidia.com/object/cuda_home_new.html)
- [16] NVIDIA CUDA programming guide, accessed 30 June 2011. [http://developer.download.nvidia.com/compute/cuda/4\\_0\\_rc2/toolkit/docs/CUDA\\_C\\_Programming\\_Guide.pdf](http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUDA_C_Programming_Guide.pdf)
- [17] OpenCL Programming Guide for the CUDA Architecture, Appendix A, accessed June 30 2011. [http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL\\_Programming\\_Guide.pdf](http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf)
- [18] AMD OpenCL, System Requirements & Driver Compatibility, accessed June 30 2011. <http://developer.amd.com/sdks/AMDAPPSDK/packages/DriverCompatibility.aspx>
- [19] INTEL OpenCL, Technical Requirements, accessed June 30 2011. <http://software.intel.com/enus/articles/opencl-release-notes/>
- [20] OpenCL Work-Item Built-In Functions, accessed June 30 2011. <http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/workItemFunctions.html>
- [21] Jack J. Dongarra, J. Bunch, Cleve Moler, and G. W. Stewart. LINPACK User's Guide. SIAM, Philadelphia, PA, 1979.
- [22] Ian N. Sneddon. *Fourier Transforms*. Dover Publications, 2010. ISBN-13: 978-0486685229.
- [23] John H. Lienhard IV and John H. Lienhard V. *A Heat Transfer Textbook*. Phlogiston Press Cambridge, Massachusetts, U.S.A, 4th edition, 2011.
- [24] Intel Xeon E5520 CPU Specifications, accessed October 28 2011. [http://ark.intel.com/products/40200/Intel-Xeon-Processor-E5520-\(8M-Cache-2\\_26-GHz-5\\_86-GTs-Intel-QPI\)](http://ark.intel.com/products/40200/Intel-Xeon-Processor-E5520-(8M-Cache-2_26-GHz-5_86-GTs-Intel-QPI))
- [25] NVIDIA Tesla M2050 GPU Specifications, accessed June 30 2011. [http://www.nvidia.com/docs/IO/43395/BD-05238-001\\_v03.pdf](http://www.nvidia.com/docs/IO/43395/BD-05238-001_v03.pdf)
- [26] Cyril Faure. *Real-time simulation of physical models toward hardware-in-the-loop validation*. PhD Thesis. University of Paris East, October 2011.

## Appendix A. Serial Matrix Multiply

```

model MatrixMultiplication
  parameter Integer m=256 ,n=256 ,k =256;
  Real result ;
algorithm
  result := mainF (m,n,k);
end MatrixMultiplication ;

function mainF
  input Integer m;
  input Integer n;
  input Integer k;
  output Real result ;
protected
  Real A[m,n];
  Real B[n,k];
  Real C[m,k];
algorithm
  // initialize matrix A, and B
  (A,B) := initialize (m,n,k);
  // multiply matrices A and B
  C := matrixMultiply (m,n,k,A,B);
  // only one item is returned to speed up
  // computation
  result := C[m,k];
end mainF;

function initialize
  input Integer m;
  input Integer n;
  input Integer k;
  output Real A[m,n];
  output Real B[n,k];
algorithm
  for i in 1:m loop
    for j in 1:n loop
      A[i,j] := j;
    end for;
  end for;
  for j in 1:n loop
    for h in 1:k loop
      B[j,h] := h;
    end for;
  end for;
end initialize ;

function matrixMultiply
  input Integer m;
  input Integer p;
  input Integer n;
  input Real A[m,p];
  input Real B[p,n];
  output Real C[m,n];
  Real localtmp ;
algorithm
  for i in 1:m loop
    for j in 1:n loop
      localtmp := 0;
      for k in 1:p loop
        localtmp := localtmp + (A[i,k]*
          B[k,j]);
      end for;
      C[i,j] := localtmp ;
    end for;
  end for;
end matrixMultiply;

```

## Appendix B. Parallel Matrix-Matrix Multiplication with parfor and Kernel functions

```

model MatrixMultiplicationP
  parameter Integer m=32,n=32,k=32;
  Real result;
algorithm
  result := mainF(m,n,k);
end MatrixMultiplicationP ;

function mainF
  input Integer m;
  input Integer n;
  input Integer k;
  output Real result ;
protected
  Real C[m,k];
  parglobal Real pA[m,n];
  parglobal Real pB[n,k];
  parglobal Real pC[m,k];
  parglobal Integer pm;
  parglobal Integer pn;
  parglobal Integer pk;
  // the total number of global threads
  // executing in parallel in the kernel
  Integer globalSize [2] = {m,k};
  // the total number of local threads
  // in parallel in each workgroup
  Integer localSize [2] = {16 ,16};
algorithm
  // copy from host to device
  pm := m;
  pn := n;
  pk := k;
  (pA ,pB) := initialize(m,n,k,pn ,pk);

  // specify the number of threads and
  // workgroups
  // to be used for a kernel function
  // execution
  oclSetNumThreads(globalSize, localSize);
  pC := matrixMultiply(pn ,pA ,pB );

  // copy matrix from device to host
  // and return result
  C := pC;
  result := C[m,k];

  // set the number of threads to
  // the available number
  // supported by device
  oclSetNumThreads(0);
end mainF ;

function initialize
  input Integer m;
  input Integer n;
  input Integer k;
  input parglobal Integer pn;
  input parglobal Integer pk;
  output parglobal Real pA[m,n];
  output parglobal Real pB[n,k];
algorithm

```

```
parfor i in 1:m loop
  for j in 1: pn loop
    pA[i,j] := j;
  end for;
end parfor;
parfor j in 1:n loop
  for h in 1: pk loop
    pB[j,h] := h;
  end for;
end parfor ;
end initialize ;

parkernel function matrixmultiply
input parglobal Integer pn;
input parglobal Real pA [: ,:];
input parglobal Real pB [: ,:];
```

```
output parglobal Real pC[size(pA,1),
size(pB,2)];
protected
Real plocaltmp ;
Integer i,j;
algorithm
// Returns unique global thread Id value
// for first and second dimension
i := oclGetGlobalId (1);
j := oclGetGlobalId (2);
plocaltmp := 0;
for h in 1: pn loop
  plocaltmp := plocaltmp + (pA[i,h] *
pB[h,j]);
end for;
pC[i,j] := plocaltmp;
end matrixmultiply;
```