



HAL
open science

An MDE Approach for Automatic Code Generation from UML/MARTE to OpenCL

de Oliveira Rodrigues Antonio Wendell, Frédéric Guyomarc'H, Jean-Luc
Dekeyser

► **To cite this version:**

de Oliveira Rodrigues Antonio Wendell, Frédéric Guyomarc'H, Jean-Luc Dekeyser. An MDE Approach for Automatic Code Generation from UML/MARTE to OpenCL. *Computing in Science and Engineering*, 2012, 15 (1), pp.46-55. 10.1109/MCSE.2012.35 . hal-01582200

HAL Id: hal-01582200

<https://inria.hal.science/hal-01582200v1>

Submitted on 5 Sep 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An MDE Approach for Automatic Code Generation from UML/MARTE to OpenCL

To reduce the design complexity of OpenCL programming, the approach proposed here generates application code automatically, based on model-driven engineering (MDE) and modeling and analysis of real-time and embedded (MARTE) systems. The aim is to provide application-development resources for nonspecialists in parallel programming, exploiting concepts such as reuse and platform independence.

Advanced engineering and scientific communities have used parallel programming to solve their large-scale complex problems for a long time. Despite their high-level expertise, developers in these communities often find it hard to implement their parallel applications effectively. Some inherent characteristics of parallel programming contribute to this difficulty, including race conditions, memory access bottleneck, granularity decision, scheduling policy, and thread safety. To facilitate the programming of parallel applications, developers have specified several approaches. The most commonly used standards are Open Message Passing (OpenMP) for shared memory and Message Passing Interface (MPI) for distributed memory programming. These approaches let us express and explore the potential parallelism of applications and architectures. OpenMP adds directives having the same syntax level as the target programming language (C, C++, Fortran),

while MPI is implemented as a library to manage communication between nodes. From this viewpoint, these approaches are actually tools to add parallelism resources to a sequential programming model rather than a solution for parallel programming.

In December 2008, the consortium managed by the Khronos Group released the Open Computing Language (OpenCL) 1.0 specification.¹ OpenCL is the first open, royalty-free standard for general-purpose parallel programming of heterogeneous systems. It provides a uniform programming environment for software developers to write efficient and portable code for high-performance computing servers, desktop computer systems, and handheld devices using a diverse mix of multi-core CPUs, GPUs, cell-type architectures, and other parallel processors such as digital signal processors (DSPs).

Here, we present an approach based on model-driven engineering (MDE)² to specify, design, and generate OpenCL applications. This approach relies on the following aspects:

- We propose three new metamodels that satisfy essential concerns of a whole application—scheduling policy, variable definitions, and the OpenCL programming model itself.

1521-9615/13/\$31.00 © 2013 IEEE
COPUBLISHED BY THE IEEE CS AND THE AIP

A. WENDELL O. RODRIGUES

Federal Institute of Education, Science, and Technology of Ceará

FRÉDÉRIC GUYOMARC'H AND JEAN-LUC DEKEYSER

University of Lille

- We propose a transformation chain that includes as many model transformations as we need for generating an efficient code (in other words, we concatenate single, specific model transformations to add or change elements designed in an input model to generate intermediate models toward the target code).
- Because abstract models don't have all of the necessary information for improving performance based on the target hardware, we include some optimization levels to achieve (whenever possible) automatically generated code that's as efficient as manually written code.

To gain a theoretical base for some of the concepts we'll discuss, see the "Background Review" sidebar. Next, we'll discuss our automatic code-generation approach.

Code-Generation Approach

Our approach aims to generate effective code for OpenCL as a new branch of our Gaspard2 development environment.³ During application design, Gaspard2 uses a Unified Modeling Language (UML) profile for modeling and analysis of real-time and embedded (MARTE) systems to define semantics to the application. Then, using transformation chains, it lets us automatically generate code for a chosen target platform. One of the main advantages of MARTE is that it clearly distinguishes the hardware components from the software components. This is done via stereotypes provided in part by the detailed resource modeling (DRM) package, in particular the `HWResource` and its derived stereotypes. Regarding the hybrid (CPU and compute device) conception, this separation is of prime importance. In fact, in a codesign environment, usually different teams simultaneously create these two parts of the system. For instance, this allows for testing the software on different kinds of hardware architecture, or reusing an architecture (with few or no changes) for different applications.

Modeling an Application

The application model concept is a fundamental modeling process. Indeed, developers will define three main aspects of their application: the tasks and their interconnection; the number of repetitions of a task and its hierarchy, as well as whether it will be instantiated as temporal or spatial modes; and how to express the dataflow. Eventually, to optimize the model, a smart procedure can perform *model refactoring*, a change made to the internal structure of a model to make it

easier to understand and cheaper to modify without changing its observable behavior.⁴ This helps us find good trade-offs in the usage of storage and computation resources, as well as in the potential parallelism of tasks and data.

To clearly distinguish a host from a compute device, both defined in an OpenCL platform model, a tagged-value description in the `HWResource` stereotype is assigned either with `Host` or with `Device`. This is a valuable definition at design time, because once we have an allocation model, transformations can promptly identify kernels from allocated tasks.

The allocation phase is defined in allocation modeling (`Alloc`) from the MARTE profile. Allocation of functional application parts onto the available resources is the main concern of system design for specific platforms. This includes both spatial distribution and temporal scheduling aspects, to map certain operations onto available computing and communication resources and services.

Although MARTE is suitable for modeling purposes, it cannot move from high-level modeling specifications to execution platforms. Gaspard2 fills this gap and introduces additional concepts and semantics to attain this requirement for system codesign. Gaspard2 defines a deployment specification level to generate compilable code from an application model. This level is related to the specification of an elementary component (EC): a basic block having atomic functions. The deployment model lets us describe how the intellectual properties (IPs)—very optimized and normally parameterized functions that depend on the target technology—must be associated to ECs.

Transformations

In MDE, a model transformation is a compilation process that transforms a source model into a target model. The source and target models are respectively conformed to the source and the target metamodels. A model transformation relies on a set of rules. Each rule clearly identifies concepts in the source and the target metamodels. Such a decomposition facilitates the extension and the maintainability of a compilation process: new rules extend the compilation process and each rule can be modified independently from others. The rules are specified in programming languages, which can be either imperative (describing how a rule is executed) or declarative (describing what's created by the rules). Declarative languages are often used in MDE because the rules' objectives can be specified independently from the execution.

BACKGROUND REVIEW

Open Computing Language is a standard for parallel computing proposed by Apple and released by the Khronos Group. In the OpenCL platform model, a system is divided into one host (a CPU) and one or more compute devices. The compute devices act as coprocessors (GPUs) to the host. An OpenCL application runs on the host, which sends instructions, defined in special functions called kernels, to the devices. The OpenCL standard defines a data- and a task-parallel programming model. In the data-parallel model, the device runs multiple instances of the kernel in parallel on distinct data. Each instance is called a *work item*. Although all work items follow the same kernel, they might perform different instructions at a time and occasionally change the instruction path (as is the case with the single-program, multiple-data, or SPMD, model). Work items can be arranged in workgroups. OpenCL defines indexing schemes by which a work item can be uniquely identified through either a global ID, or through a workgroup ID along with a local ID. Synchronization of work items is possible within a workgroup only, and takes the form of a barrier.

Model-driven engineering (MDE)¹ aims to raise the level of abstraction in program specification and increases automation in program development. MDE techniques have been widely used in software production. In an MDE approach, we address the use of models at different levels of abstraction for developing systems, thus raising the level of abstraction in program specification. An increase of automation in program development is reached by using executable model transformations. Higher-level models are transformed into lower-level models until the model can be made executable using either code generation or model interpretation.

In an MDE approach, Unified Modeling Language (UML) can be used to specify, visualize, modify, construct, and document software artifacts. To meet specific details of a particular application domain, UML can be extended by profiles. The UML profile for modeling and analysis of real-time and embedded (MARTE)² systems consists of defining foundations for a model-based description of real-time and embedded systems (RTES). Indeed, it extends the possibilities to model the application, architecture, and their relationships. Moreover, MARTE allows extending the performance analysis and task scheduling based on target platform architecture. These core concepts are then refined for both modeling and analysis. MARTE also deals with model-based analysis. From this perspective, the aim isn't to define new techniques for analyzing RTES, but to support them. Hence, it provides facilities to annotate models with information required to perform specific analysis. The main benefits of using this profile are as follows:

- to provide a common way of modeling both hardware and software aspects of a RTES for improving communication between developers;

- to enable interoperability between development tools used for specification, design, verification, and code generation; and
- to foster the construction of models that can be used to make quantitative predictions regarding systems' real-time and embedded features, accounting for both hardware and software characteristics.

Allocation modeling (Alloc) from Foundations; generic resource modeling (GRM) and generic component modeling (GCM) from Design Model; and the repetitive structure modeling (RSM) annex are packages that are part of the MARTE specification. They provide the main resources to model and describe our entire application. In particular, RSM provides concepts to allow expressing the potential parallelism of applications.

With respect to code generation for GPUs, there are a couple of related works, such as the Single-Assignment C (SAC) and CAPS projects (see www.openhmpp.org). SAC³ is a strict, purely functional programming language whose design is focused on the needs of numerical applications. CAPS proposes the Open Hybrid Multicore Parallel Programming (OpenHMPP) toolkit. This toolkit is a set of compiler directives, tools, and software runtime that supports multicore and manycore processors parallel programming in C and Fortran. This approach is similar to the widely available standard, Open Message Passing (OpenMP).

There are several other approaches that generate GPU applications from other existing languages such as C, Fortran, Python, and Java.⁴⁻⁶ However, even if we're inspired by some of their features (such as their optimizations), all of them rely on code-to-code transformations, and don't deal with higher-level and abstract software specification.

References

1. D. Lugato, J.-M. Bruel, and I. Ober, "Model-Driven Engineering for High Performance Computing Applications," *Proc. Modeling Simulation and Optimization Focus on Applications*, Acta Press, 2010, pp. 303–308.
2. Object Management Group, "UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems," version 1.1, 2011; www.omg.org/spec/MARTE/1.1.
3. J. Guo, J. Thiyagalingam, and S.-B. Scholz, "Towards Compiling SAC to CUDA," *Trends in Functional Programming 10*, vol. 10, Gutenberg Press, 2011, pp. 38–48.
4. M. Baskaran, J. Ramanuja, and P. Sadayappan, "Automatic C-to-CUDA Code Generation for Affine Programs," Springer, 2010, pp. 244–263.
5. A. Klöckner, "PyOpenCL: From Python to OpenCL," user guide, 2011; <http://mathematician.de/software/pyopencl>.
6. C. Toepfer, *Using GPU-Enabled Math Libraries with PGI Fortran*, tech. report, The Portland Group, 2011.

Table 1. OpenCL* transformation chain.

Transformation	Module description
1. UML to MARTE metamodel*	This transformation adapts a model conforming to UML to a model conforming to the MARTE metamodel. Having this model, remaining transformations don't need to deal with UML's unnecessary extra complexity.
2. Instances identification	This lets us add instances of ports for each part within the component. Otherwise, we aren't able to identify local variables.
3. Tiler processing	This module transforms every tiler specified in the models into tasks.
4. Local graph generation 5. Global graph generation 6. Static scheduling policy	These transformation modules undertake the definition of task graphs and a scheduling policy.
7. Memory allocation	This is for memory handling, variable definitions, and data communication.
8. Hybrid conception	It summarizes all explicitly modeled or implicitly defined elements by earlier transformations into a single structure.
9. Code generation	This is the only model-to-text transformation in the chain. At this point, we're already able to write out the previously analyzed elements directly to source code according to the OpenCL syntax.

* MARTE = modeling and analysis of real-time and embedded system; OpenCL = Open Computing Language; and UML = Unified Modeling Language.

Table 1 shows the transformation modules present in the OpenCL chain, defined according to our model transformation engine. Two types of transformation are implemented: *model-to-model* and *model-to-text*. The transformation engine is compliant with the Meta-Object Facility Query/View/Transformation (MOF QVT; see www.omg.org/spec/QVT), a language proposed by the Object Management Group (OMG). Gaspard2 uses the QVT-Operational (QVTO) tool to standardize the model transformations, and to render them compatible with future versions of MARTE.

In Table 1, we present the sequence of transformations that compose the UML to OpenCL chain. The input for transformation 1 comprises the whole designed model (application, architecture, deployment, and allocation). Then, we no longer need the UML profile for MARTE. Aiming for simpler transformations, and to add further concepts, we use the MARTE metamodel, whose elements are based on stereotypes from the original profile. Missing notions necessary to our approach, such as memory mapping, are provided by the metamodel. Therefore, the metamodel lets us work with the notions that previously were missing, converting the initial model into a new model that conforms to the MARTE metamodel. Furthermore, some instance concepts, such as port instances (not provided by UML), are added to the model to create variables of instantiated tasks (transformation 2).

MARTE's response-surface model Repetitive Structure Modeling (RSM) package uses definitions

from the Array Oriented Language (ArrayOL).⁵ FlowPort stereotypes are applied to UML ports, and this lets us specify features such as direction, size, and data type. Hence, to interconnect to FlowPorts, RSM defines special connectors tagged with `tilers` stereotypes. The `tiler` specified in ArrayOL lets us declare how the elements from an input array will be distributed into subelements called patterns. Then, these patterns will be consumed by each iteration of a repetitive task. This is the foundation for the model of computation (MoC) adopted in the Gaspard2 environment. This MoC enables the expression of the potential parallelism of data and tasks. `tilers` are analyzed by transformation module number 3 from Table 1. This transformation creates special tasks that will be allocated onto available processors, which usually are the same ones in charge of the respective tasks interconnected by `tilers`.

Transformation modules 4, 5, and 6 deal with static task scheduling. After these transformations, local and global task graphs are generated, as well as an ordered list of tasks. Note that we aren't searching for an optimal scheduling policy at this level of task execution. For our target architecture, GPU, submitted tasks (kernels) execute according to the GPU scheduler for each microprocessor, and we generally can't control this process. The goal is to create a macro call list for tasks globally defined in the model. Therefore, this at least assures coherence for exchanged data by the tasks. For independent tasks, however, OpenCL lets us define asynchronous task scheduling from the host dispatcher. This avoids blocking sequential calls.

To specify future variables in the application, the transformation module 7 handles all `FlowPorts` and their connectors available in the model and creates new elements containing enough information to prepare the memory allocation for each memory bank.

Once we have all necessary elements processed in previous module transformations, we synthesize these elements into a single structure. This produces the Hybrid model, which includes element definitions closer to our target platform: OpenCL. For instance, tasks will become functions and their ports will become parameters.

Now that we have this Hybrid model, a model-to-text step automatically generates code by using template editors. Obeo (see www.acceleo.org) supplies the Acceleo plug-in in the context of MDE. Acceleo scripts, based on the Model-to-Text OMG standard, make it possible to generate files from common format models.

Code Optimization

During the transformation process, we can optimize some aspects to get higher-performance OpenCL code. To do so, we consider three approaches. The first attempts to avoid unnecessary data transfers between the host and device by observing `FlowPort` allocations. The second approach relies on shared memory use. Some applications can exploit memory sharing within workgroups of work items, and thus have high-performance behavior. This procedure involves `tiler` analysis. The third approach involves integrating profiling tools and models. Indeed, the traceability available in Gaspard2 makes the integration of profiling tools and models possible, and thus it can guide designers to improve their application model. These optimization approaches are detailed elsewhere,⁶ and some aspects of them are exemplified in the next section.

Experimental Results

In our approach, we aimed to produce a functional and compilable code for the target platform, taking into account all of the functionalities of the earlier designed model. In addition, we wanted a code that exploits the potential parallelism provided by the runtime architecture. Let's look at three general applications that were designed using our approach. Despite having different goals, all of these applications have tasks that can run in parallel, and they're well-suited to a multithreaded environment. In the following examples, we used a 2.26-GHz Intel Core 2 Duo processor and S1070 unit (with four Tesla T10 Nvidia GPUs; although some benchmarks use only one T10).

Video Downscaling

The modeled feature of the video-processing application deals with scaling. It consists of a classical downscaler, which transforms a video graphics array signal—Common Intermediate Format (CIF)—into a smaller one. Such an operation is necessary to display high-quality live video on a thin-film transistor screen while using low power and real-time previews, such as view mode in the video functionality of a cell phone. The downscaler module has two components: a horizontal filter that reduces the number of pixels from a 352-column frame to a 132-column frame by interpolating patterns of 8 pixels; and a vertical filter that reduces the number of pixels from a 288-line frame to a 128-line frame by interpolating patterns of 8 pixels as well.

Figure 1 gives an overview of this application model. In addition to features such as tiler specifications and task repetitions, a MARTE profile is applied to the OpenCL architecture model to identify the host and device. For this example, we've specified one host (the CPU and its RAM) and one device (the GPU and its global memory). Data and tasks are associated with memories and processors according to the project needs (partially shown in Figure 2). For instance, the six repetitive tasks in `horizontal` and `vertical` filters are placed onto the GPU, and thus they'll become kernels in the runtime environment. Allocated stereotypes provided by MARTE are used to map ports and tasks onto `HwRAMs` and `HwProcessors`. These stereotypes will allow for creating all of the variables and relations between them. Additionally, to distinguish the host from the device, we chose the `description` attribute from the `HwResource` stereotype. Moreover, `tiler` connectors are added to each repetitive task from the `HorizontalFilter` and `VerticalFilter`. These connectors contain structured information based on arrays that allow describing how to distribute an array of {288,352} elements into array patterns of {11} elements. The formalism necessary to implement the `tilers` can be found elsewhere.⁵

Figure 1 shows the association between a software IP (a function's code) and an elementary task's red horizontal filter (RHF). Here, we mean to demonstrate how we include code for single operations in the resulting generated code. These associations are implemented with the artifact component from UML.

Downscaling Results

Four versions of the downscaler were tested. The first is a sequential version using the same

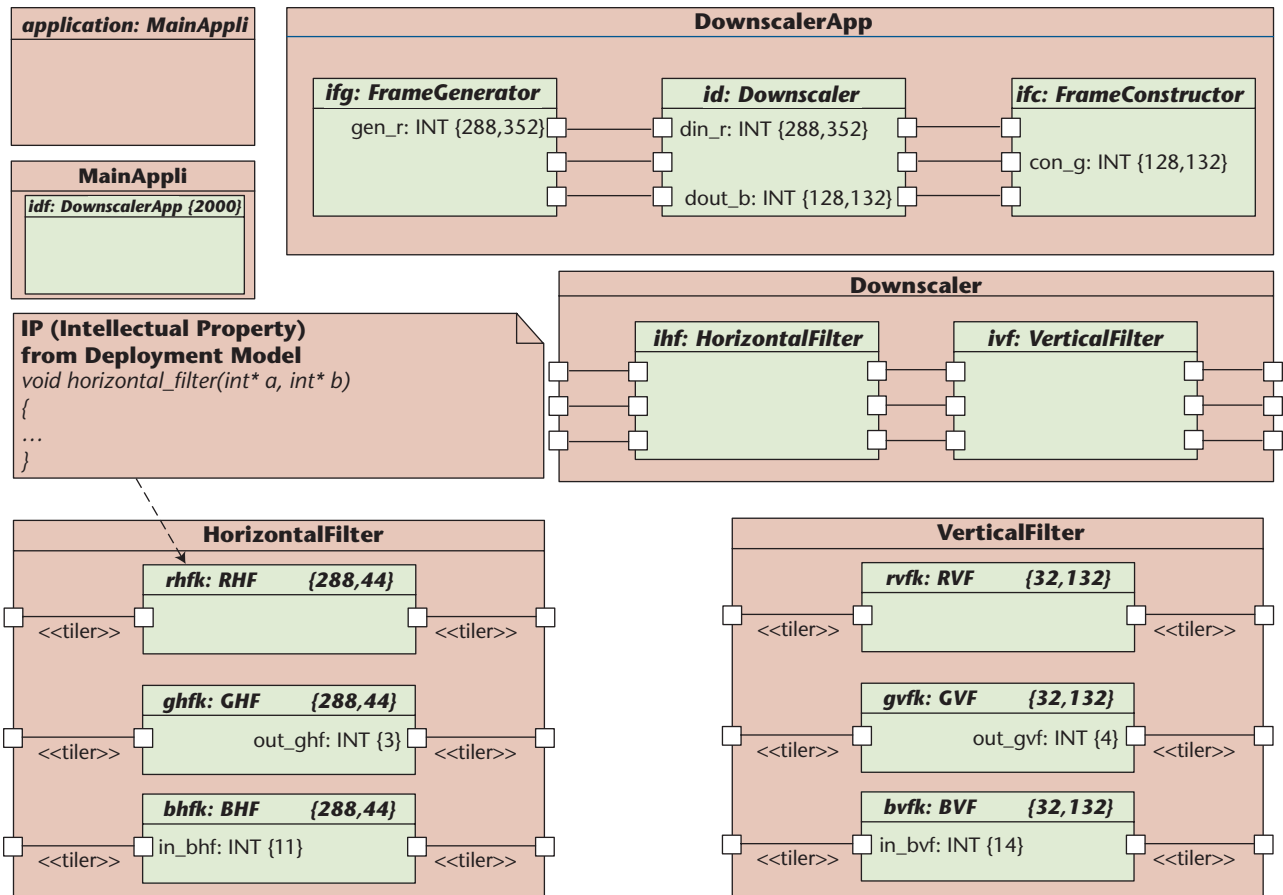


Figure 1. Downscaler application model. This model describes the main tasks of the frame downscaling application.

structure defined in Figure 1. Two are automatically generated OpenCL versions, and the last is a manually written OpenCL version. The first OpenCL-generated code has no optimization or further analysis on memory transfers. The second optimizes the memory transfers between the host and device. Minimizing these transfers notably reduces the total GPU execution time, because data transfers take a lot of time (about 70 percent) in this application. The generated code achieves a speedup of 10 times, as Table 2 shows.

Matrix Multiplication

Most scientific numerical methods apply matrix multiplication. We've designed two models for this application. In the first model, every work item takes one line and one column from matrices A and B and produces one point in matrix C using the scalar product. In the second model, we organize work items in workgroups that perform the multiplication by block. The idea behind this approach is to use shared memories within workgroups.

Figure 3 illustrates the results from three OpenCL codes running on four matrix sizes.

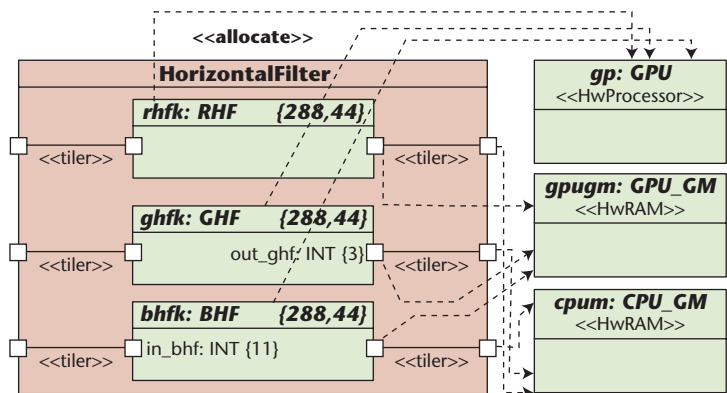


Figure 2. Allocation of application elements on architecture elements. This procedure defines what hardware elements will execute tasks or store variables.

Even though the first and second versions have different codes, they work on same matrix sizes using similar procedures, having the same computational effort. Thus, we observe no performance gains. Indeed, even having different models, all work items do the same computing work. However, the third code has an expressive performance.

Table 2. Results for downscaler applications.

Downscaler version	Time (in seconds)	Speedup	Gflops
Sequential	36.0	1	0.120
OpenCL	4.9	7.35	0.898
OpenCL with transfer optimization	3.6	10	1.2
OpenCL manually coded	3.6	10	1.2

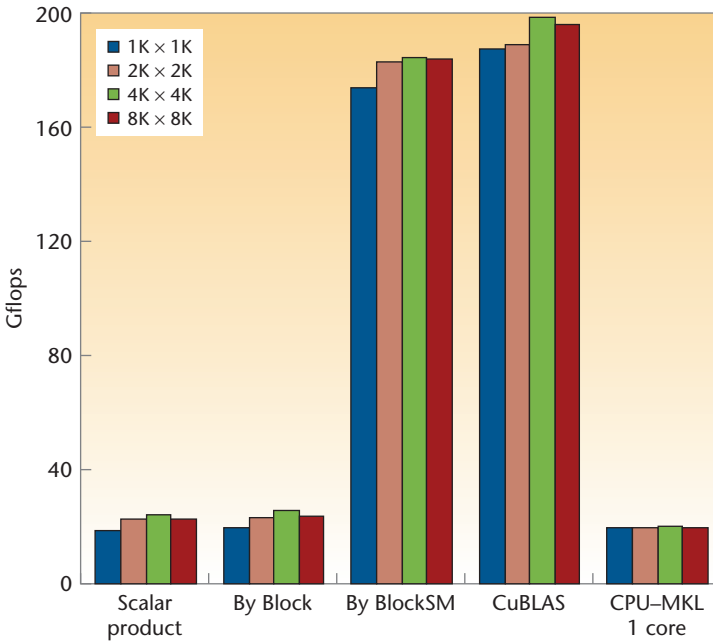


Figure 3. Running results obtained with different matrices. Here you can see the results from three OpenCL codes running on four different matrix sizes. (BlockSM = By Blocks using Shared Memory; CuBLAS = CUDA Basic Linear Algebra Subroutines; and MKL = Math Kernel Library.)

```

1:  $x_0 \leftarrow 0$ 
2:  $r_0 \leftarrow b$ 
3:  $\text{norm}_{r_0} \leftarrow \text{norm}_2(r_0)$ 
4:  $p_0 \leftarrow r_0$ 
5:  $\text{error} \leftarrow 1$ 
6:  $k \leftarrow 0$ 
7: while  $\text{error} > \text{ERROR\_MAX}$  do ▷ We stop if error is sufficiently small
8:  $\alpha \leftarrow \frac{(r_k^T r_k)}{(p_k^T A p_k)}$ 
9:  $x_{k+1} \leftarrow x_k + \alpha_k p_k$ 
10:  $r_{k+1} \leftarrow r_k + \alpha_k A p_k$ 
11:  $\beta \leftarrow \frac{(r_{k+1}^T r_{k+1})}{(r_k^T r_k)}$ 
12:  $p_{k+1} \leftarrow r_{k+1} - \beta_k p_k$ 
13:  $\text{error} \leftarrow \frac{\text{norm}(r)}{\text{norm}_{r_0}}$ 
14:  $k \leftarrow k + 1$ 
15: end while

```

Figure 4. Conjugate gradient (CG) algorithm without the preconditioner. This algorithm is the basic form of the iterative method commonly used on solution of systems of linear equation.

In fact, if we enable optimization analysis in the `tiler` transformation, data copies from the global memory to shared (local) memory in the GPU processors allow for fast data access and reuse by all the work items of the same workgroup. Additionally, as these results indicate, different matrix sizes don't have great influence on performance. To compare our results to classical solutions, we provide performances achieved with the same hardware using the Nvidia CUDA Basic Linear Algebra Subroutines (CuBLAS) library and Intel Math Kernel Library (MKL).

Conjugate Gradient

The conjugate gradient (CG) method⁷ is often used in numerical algorithms. For this example, the input data comes from a finite-element method (FEM) model of an electrical machine. The matrix is stored in a compressed sparse row (CSR) format having $N = 132,651$ and $NNZ = 344,2951$. The CG main loop algorithm (lines 7–15 of the algorithm in Figure 4) is modeled in MARTE (see Figure 5), where data reading and startup configurations are defined by stereotyped blocks allocated on the CPU.

Highlighted blocks represent tasks, which are mapped onto as many devices as we want to distribute the task job. Tasks—such as Sparse Double-precision General Matrix-Vector Multiplication with Sparse Matrices (S-DGEMV) shown in lines 8 and 10—are repetitive, and thus, potentially parallel. The CG is repeated 132,651 times, and some of its input data are replaced by output data between continuous iterations. A *continue-condition* (line 7) is specified by a constraint annotation to the outer CG block, thus the loop stops before running all iterations if it achieves a given tolerance error. Figure 5 is only an internal view of the CG loop model. Here, scalar operations run on the CPU processor, and repetitive operations run on GPU processors according to task allocations. Further information about this case study is detailed elsewhere.⁸

We launched four double-precision versions of the CG. The first (the reference result) is a CPU sequential code that uses Matlab's `pcg` function. The other versions are automatically generated

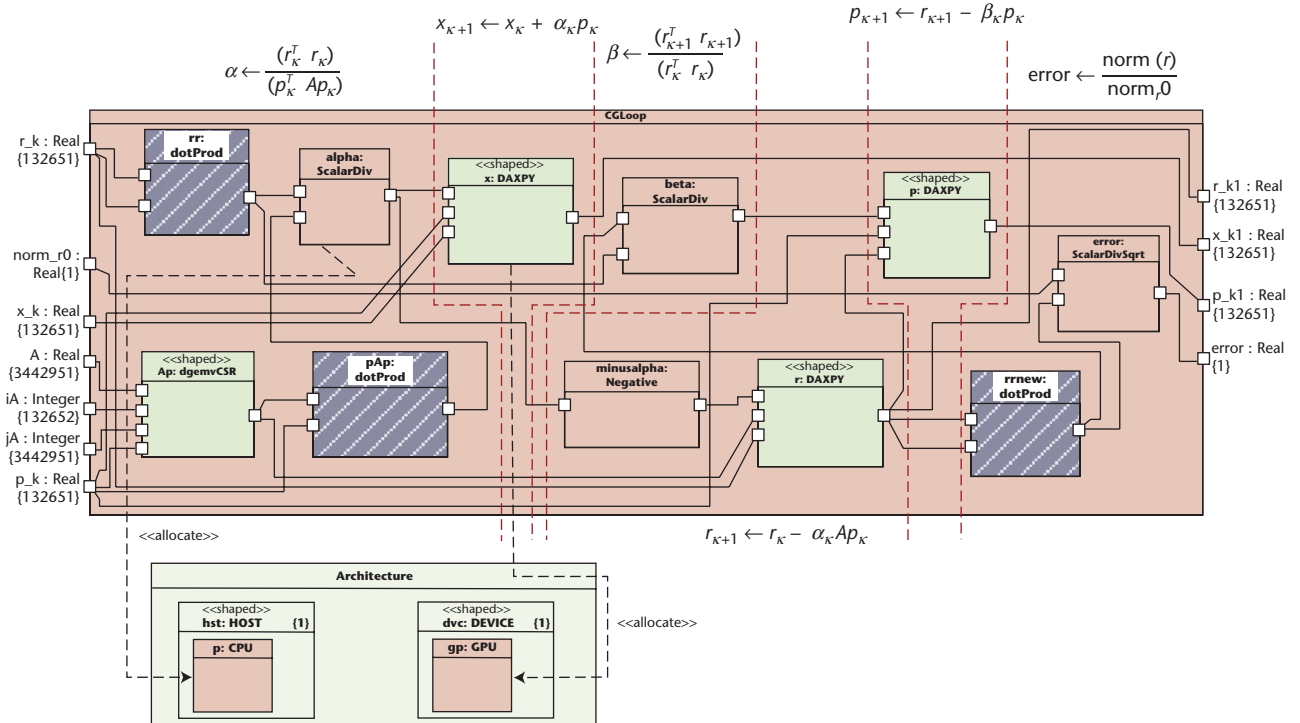


Figure 5. Conjugate gradient application model (based on the CG main loop algorithm shown in Figure 4). This is only an internal view of the CG loop model. Data reading and startup configurations are defined by stereotyped blocks allocated on the CPU.

OpenCL implementations whose kernels are launched onto one, two, and four devices, respectively. The listing in Figure 6 presents a sample of the code generation for the DAXPY kernel. The IP is inserted as a function and the kernel consists of two gather functions (input tilers), one IP call, and one scatter function (output tilers). The number of running devices depends on the task-allocation process. Usually, manually written codes have better performance than automatic codes due to the application-oriented development. However, these generated CG implementations have a more expressive performance (see Table 3) compared to sequential code (benchmarks include just the computing and data transfer times in the CG loop). The multi-GPU aspect is verified in the two other versions. The model compiler decides equally the task (and associated data) partitioning on the multiple devices. Nevertheless, the performance gain isn't linear, due to extra data transfers between the CPU and devices. A detailed analysis of solvers and multi-GPU can be found elsewhere.⁹

From an abstract model defined using UML/MARTE, we generated a compilable OpenCL code and then a functional, executable application. As an

MDE approach, this model is a quick codesign and development tool for nonexpert programmers. We thus consider this approach an effective operational code generator for the newly released OpenCL standard.

This work also provides resources to model applications running on homogeneously configured multidevices, offering two main contributions: it lets the user model simple distributed-memory aspects, such as data transfers and memory allocations, and it lets the user model the platform and execute models of OpenCL. Additionally, its smart transformation capabilities can determine optimization levels in data communication and data access. Studies have shown that these optimizations remarkably increase the application performance.

With respect to language dependence, other target languages can use the Hybrid metamodel that we proposed. This hybrid can match memory, platform, and execution models, such as CUDA. For future work, we'll make model-to-text templates (that we've already created) available on the code-generation engine, to exploit multilanguage capabilities.

Acknowledgments

This work is part of the Gaspard2 project, developed by the Dynamic Adaptivity and Real-Time (DART)

```

1 void daxpyfunc (const double * y, const double * x, double a )
2 {
3   y [0] = a * x[0] + y[0];
4 }
5 __kernel void daxpy_KRN (
6   uint iNumElements ,
7   const __global double * v2_daxpy_KRN,
8   __global double * v1_daxpy_KRN,
9   const __global double ct_daxpy_KRN)
10 {
11   double v1_loc[1]; double v2_loc[1];
12   // get index into global data array
13   int iGID = get_global_id(0) +
14   get_global_size(0) * get_global_id(1) +
15   get_global_size(0) * get_global_size(1) * get_global_id(2);
16   if (iGID < iNumElements) // boundcheck
17   {
18     { // input tiler
19       uint tIter[1];
20       uint t1[1];
21       uint ref[1];
22       uint index[1];
23       tIter[0] = iGID%132651;
24       ref[0] = 0 + 1 * tIter[0];
25       for (t1[0] = 0; t1[0] < 1; t1[0]++) {
26         index[0] = (ref[0] + 0 * t1[0])%132651;
27         v2_loc[t1[0] * 1] = v2_daxpy_KRN[index[0] * 1];
28       }
29     }
30     { // input tiler
31       uint tIter[1];
32       uint t1[1];
33       uint ref[1];
34       uint index[1];
35       tIter[0] = iGID%132651;
36       ref[0] = 0 + 1 * tIter[0];
37       for (t1[0] = 0; t1[0] < 1; t1[0]++) {
38         index[0] = (ref[0] + 0 * t1[0])%132651;
39         v1_loc[t1[0] * 1] = v1_daxpy_KRN[index[0] * 1];
40       }
41     }
42     daxpyfunc (v1_loc, v2_loc, ct_daxpy_KRN); //IP call
43     { // output tiler
44       uint tIter[1];
45       uint t1[1];
46       uint ref[1];
47       uint index[1];
48       tIter[0] = iGID%132651;
49       ref[0] = 0 + 1 * tIter[0];
50       for (t1[0] = 0; t1[0] < 1; t1[0]++) {
51         index[0] = (ref[0] + 1 * t1[0])%132651;
52         v1_daxpy_KRN[index[0] * 1] = v1_loc[t1[0] * 1];
53       }
54     }
55   }
56 }

```

Figure 6. Listing 1. OpenCL code generated for the DAXPY kernel.

Table 3. Performance results (tol = 1e-10).*

Conjugate gradient	No. of iterations	Time (in seconds)	Speedup	Gflops
Matlab PCG	117	3.17	1	0.303
OpenCL (1 GPU)	116	0.659	4.81	1.45
OpenCL (2 GPUs)	116	0.461	6.87	2.07
OpenCL (4 GPUs)	116	0.380	8.34	2.50

* Tol is an indicator for the maximum iterations of tolerance; it's a minimum error used to stop the iterations.

team of Laboratoire d'Informatique Fondamentale de Lille (LIFL)/INRIA Lille. It has been funded by the Conseil Régional Nord-Pas-de-Calais and Valeo.

References

1. Khronos OpenCL Working Group, *The OpenCL Specification*, version 1.1, revision 44, 2011.
2. D. Lugato, J.-M. Bruel, and I. Ober, "Model-Driven Engineering for High Performance Computing Applications," *Proc. Modeling Simulation and Optimization Focus on Applications*, Acta Press, 2010, pp. 303–308.
3. A. Gamatié et al., "A Model Driven Design Framework for Massively Parallel Embedded Systems," *ACM Trans. Embedded Computing Systems*, vol. 10, no. 4, 2011, article no. 39.
4. C. Glitia et al., *Repetitive Model Refactoring for Design Space Exploration of Intensive Signal Processing Applications*, tech. report, INRIA, 2009.
5. P. Boulet, *Array-OL Revisited, Multidimensional Intensive Signal Processing Specification*, tech. report, INRIA, 2007.
6. W. Rodrigues, "A Methodology to Develop High Performance Applications on GPGPU Architectures: Application to Simulation of Electrical Machines," doctoral thesis, Computer Science Dept., Univ. des Sciences et Technologie de Lille, 2012.
7. G.H. Golub and C.F. Van Loan, *Matrix Computations*, 3rd ed., The Johns Hopkins Univ. Press, 1996.
8. W. Rodrigues et al., "Automatic MultiGPU Code Generation Applied to Simulation of Electrical Machines," *IEEE Trans. Magnetics*, vol. 48, no. 2, 2012, pp. 831–834.
9. A. Cevahir, A. Nukada, and S. Matsuoka, "High Performance Conjugate Gradient Solver on Multi-GPU Clusters Using Hypergraph Partitioning," *Computer Science Research and Development*, vol. 25, nos. 1–2, 2010, pp. 83–91.

A. Wendell O. Rodrigues is an associate professor in the Telematics Department at the Federal Institute of Education, Science, and Technology of Ceará. His research interests include parallel architectures and programming, and software engineering, specifically with regard to GPUs and high-level software specification. Rodrigues has a PhD in computer science from the University of Lille, France. Contact him at wendell.rodrigues@inria.fr.


Frédéric Guyomarc'h is an associate professor in the Computer Science Department at the University of Lille. His research interests focus on high-performance computing, from algorithms for numerical computation to compilation techniques to generate optimized code for such algorithms. Guyomarc'h has a PhD in computer science from the University of Rennes, France. Contact him at frederic.guyomarch@inria.fr.

Jean-Luc Dekeyser is a computer science professor at the University of Lille. His research interests include embedded systems, system-on-chip codesign, synthesis and simulation, performance evaluation, high-performance computing, model-driven engineering (MDE), dynamic reconfiguration, and Chip-3D. Dekeyser has a PhD in computer science from the University of Lille. Contact him at jean-luc.dekeyser@inria.fr.

SUBMIT TODAY!
Publishing in 2013

IEEE TRANSACTIONS ON
**EMERGING TOPICS
IN COMPUTING**

IEEE Transactions on Emerging Topics in Computing publishes papers on emerging aspects of computer science, engineering, technology, and applications not currently covered by other IEEE Computer Society Transactions. *TETC* is an open access journal which allows for wider dissemination of information.



Submit your manuscript at: www.computer.org/tetc. *TETC* aggressively seeks proposals for Special Sections and Issues focusing on emerging topics. Prospective Guest Editors should contact the EIC of *TETC* (Dr. Fabrizio Lombardi, lombardi@ece.neu.edu) for further details. Submissions are welcomed on any topic within the scope of *TETC*. Some examples of emerging topics in computing include:

- IT for Green
- Synthetic and organic computing structures and systems
- Advanced analytics
- Social/occupational computing
- Location-based/client computer systems
- Electronic game systems

- Health-care IT
- Computer support for peer tutoring and learning
- Creation and management of learning objects, tools, and techniques
- Advanced paradigms and initiatives in computing and storage with new technologies

