A Flexible Execution Framework for High-Performance TCAD Applications

J. Weinbub*, K. Rupp*†, and S. Selberherr*
*Institute for Microelectronics, TU Wien, Gußhausstraße 27–29/E360, A-1040 Wien, Austria
† Institute for Analysis and Scientific Computing, TU Wien, Wiedner Hauptstraße 8-10, A-1040 Wien, Austria
Email: {weinbub|rupp|selberherr}@iue.tuwien.ac.at

***Abstract*—We present our approach for a flexible execution framework, ViennaX, in the field of Technology Computer Aided Design (TCAD). Our modular concept enables not only the decoupling of simulation tools, resulting in an increased level of reusability, but also in a combination of various simulation components for the specification of multiphysics simulation flows. Such combinations allow for intricate simulation setups, as, for example, the investigation of a physical phenomena can be improved by coupling different simulators. We introduce ViennaX to utilize and combine available simulation components by a plugin approach. Plugin data dependencies can be defined, which is the basis for the task execution based on a task graph approach. We depict the versatility and the performance of our approach, by discussing a typical application in the field of TCAD.**

I. Introduction

The field of TCAD is based on modeling various aspects of semiconductor device fabrication and operation. A promising approach to improve the quality of the modeling of physical phenomena is to combine highly specialized simulation tools, as is also common practice in other fields, like computational fluid dynamics [1]. For example, a stochastic simulation might be performed to compute the mobility of the carriers in a complex device and materials arrangement accurately, which is then used in a simpler transport model of a classical, deterministic device simulator. In short, important tasks are to couple simulations modeling relevant phenomena on a different physical level, thus performing multiphysics computations. To minimize the development effort for such application scenarios, a flexible execution framework is required, which enables the combination and utilization of already available tools. In addition to the heterogeneous setting of TCAD tools, the individual tasks become more and more computationally intensive, as the complexity and/or accuracy of the modeling of physical phenomena continually increases. Furthermore, the field of TCAD offers several typical application scenarios with coarse-level parallelism, for instance, current/capacitance-voltage characteristics dependent on random dopant simulations [2]. Such fields of application are of high interest with respect to process variability, which has an increasing influence on device characteristics due to scaling. This fact introduces the need to distribute parallelizable simulation tasks on different computing units, ultimately reducing the overall computation time. Such an approach becomes more and more important, due to the broad availability of multi-core CPUs by simultaneously stagnating clock frequencies. Another important aspect is to introduce a higher level of reusability in the simulation tools. A conventional example is the implementation of a deterministic simulator. Typically, such a tool is composed of core parts, e.g., an initial guess module, a Finite Element assembler, or a linear solver. A common task is to exchange specific modules in order to investigate alternative approaches provided by different tools. For example, a different linear solver might yield improved convergence behavior. This introduces the dire need for orthogonality in the simulator's code base, as switching of a module must not affect other components by, for example, introducing an altered interface which no longer fits to the remaining parts. Finally, the field of TCAD offers a plethora of publicly available simulation tools [3], [4]. However, only a small fraction of the tools is available under a free open source license, also referred to as free software [5]. This impedes the progress of research in academia, as researchers are unable to access and extend the code base of previously implemented software. In such a case, simulation tools must eventually be re-implemented. Clearly, this introduces additional development overhead with a negative impact on the actual net research time. Therefore, the field of semiconductor process and device simulation in academia can greatly benefit from free open source software packages. These facts have already been established by various software developments, such as the Archimedes project [5]. Other fields, such as computational fluid dynamics (CFD), show that this concept works by providing multipurpose simulation frameworks, similar to the COOLFluiD framework [6]. In this work we depict the applicability of parallelized task execution for the field of TCAD, based on our extendible component framework ViennaX [7]. The framework is coded in the C++ programming language and relies on already available functionality, such as the Boost libraries [8]. Additionally, the implementation of the framework utilizes modern programming techniques, like generic programming [9], to realize a maintainable and extendible implementation by simultaneously upholding a small code base. As an example we discuss a simple, yet representative application scenario, being the generation of a set of current-voltage characteristics in dependence on doping levels. We show the applicability of our approach by utilizing an already available implementation of a classical device simulation implementation. The generation of the characteristics is highly accelerated by parallelizing the individual device simulations appropriately.

 ISBN 978-0-615-71756-2

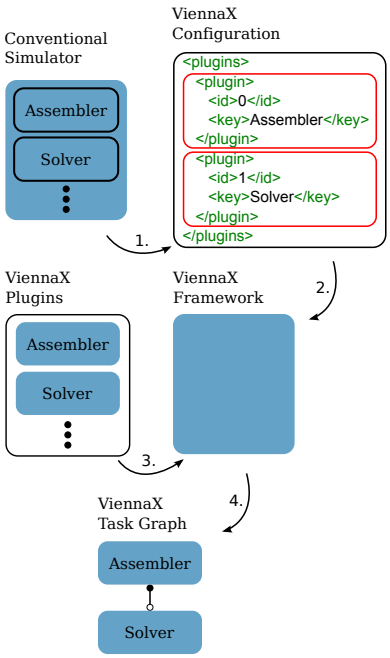

Fig. 1: The proposed decoupling of simulation tools is depicted by utilizing ViennaX. Step 1: A conventional simulation tool is described by a configuration file. Step 2: The configuration file is loaded. Step 3: The required plugins are loaded, according to the configuration data. Step 4: The task graph is generated based on the dependencies, and ultimately executed. The final simulation flow follows the initial one. However, the modularity, hence the flexibility, is increased substantially, as plugins can be easily exchanged.

II. Related Work

The Intel Threading Building Blocks (Intel TBB) library [10] is a free open source library licensed under the GPLv2. The library provides mechanisms to express parallelism based on a shared-memory approach to C++ implementations. One of the core features is the so-called flow graph. A flow graph can be used to send messages, representing arbitrary data, between components. Our approach is similar to the one of Intel's TBB library. The primary difference, though, is the fact, that the TBB library is based solely on a shared-memory approach. Therefore, it does not scale beyond one computing node.

The COOLFluiD project [1], [6] enables CFD-oriented multiphysics simulations based on a component framework. The core is a flexible plugin system, coupled with a data communication layer based on so-called data sockets. Each plugin can set up data sockets which are in turn used to generate a dependence hierarchy. This dependence information is used to drive the overall execution. The source code is available under the LGPLv3 license. The significance of the COOLFluiD framework with respect to our approach is twofold. First, we adopted the plugin system, enabling us to conveniently reuse already available functionality.

Second, the communication layer based on data sockets is the basis for our implementation. However, our approach differs significantly. COOLFluiD performs an automatic partitioning and distribution of the data structures via the Message Passing Interface (MPI), whereas we follow the approach, that distribution should be performed on the user's intent within plugins, not automatically. In contrast to COOLFluiD, our approach enables a more general way to model processes, like the Intel TBB library, which ultimately supports utilization in a much broader field of scientific computing.

III. The Framework

We focus on the decoupling of simulation flows according to their inherent functional blocks, as depicted in Fig. 1. This modularization is implemented by utilizing our ViennaX framework, which provides a plugin system, configuration mechanisms, and execution schedulers. A serial and a parallel MPI-based scheduler are available. While the serial scheduler solely utilizes a single CPU core for executing tasks, the parallel MPI-based scheduler distributes plugins to different available cores, as long as the plugins can be executed concurrently (Fig. 2). The plugin system is powered by a so-called self-registering technique [11]. This plugin approach introduces a high level of reusability by wrapping already available functionality into components with a specific, unified interface. The plugins can contain core parts of simulations, such as a linear solver implementation, but also full-fledged simulators in their own right. This approach is highly flexible; for example, simulation tools may be combined to form multiphysics simulation flows, but they may also be decomposed into smaller components, enabling specific exchanges of functionality by switching the respective plugins.

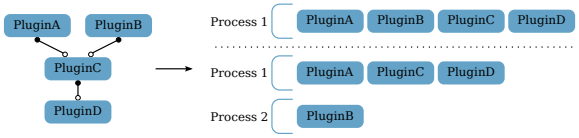

Fig. 2: The plugin execution is handled by the scheduler. If the plugins are parallelizable and there are free processes, the MPI-based scheduler distributes the task executions. Dots refer to outgoing data dependencies, and circles to incoming.

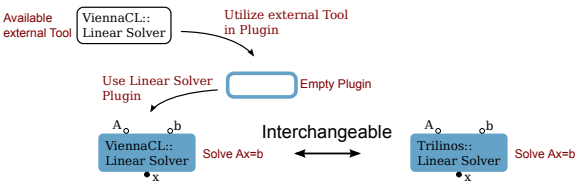

Fig. 3: A plugin can be used to wrap available functionality. Due to the abstraction mechanism provided by the socket input/output dependencies, plugins can be exchanged by other plugins.

Fig. 3 depicts the setup and exchange of a plugin. If the process of interchanging plugins is compared to the one of conventional simulation tools, it is clear that the conventional approach would require actual coding, and as such requires in-depth knowledge of the implementation at hand. For obvious reasons, this fact impedes the implementation of changing functionality. With our plugin-based approach, the exchange can be realized conveniently, by adjusting the input configuration data accordingly.

A core part of a task graph execution framework is the communication layer. As already mentioned, we adopted the data socket approach of the COOLFluiD project [6]. Essentially, each plugin can define input and output data sockets, called sink and source, respectively. These data sockets can be used to send data to and from plugins. Sockets are defined before the execution of the overall task graph, and can be based on parameters provided by the input configuration data. Fig. 4 depicts our approach for defining a plugin's data sockets. Arbitrary data types can be associated with each data socket, which are additionally identified with a unique identification string. The connection of the data sockets, which relates to generating the underlying task graph, is carried out automatically by the framework. The unique identification string as well as the associated data type of each data socket is used to automatically connect the corresponding counterparts. To ascertain the validity of the input data, units of physical quantities can be attached to the identification string of the sockets. As the socket connection algorithm checks against this information, sockets with data of different units cannot be connected. This is an important aspect, as it automatically catches one of the fundamental sources of error in scientific computing.

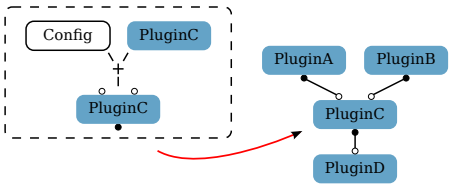

Fig. 4: The data sockets of a plugin are created based on the configuration during run-time. The sockets can be used to exchange data with other plugins.

A typical simulation task is implementing optimization processes. ViennaX enables this type of simulations by supporting loops in the task graphs, as depicted in Fig. 5. This loop mechanism can be used to, for example, implement automatic mesh generation based on convergence to an optimal set of parameters driven by a specific metric, such as the quality of the generated mesh elements.

IV. Reusability

One of the major tasks is to utilize already available functionality in order to reduce the overall implementation time. For simulation components, like linear solvers, this task is typically straightforward, as the implementations are typically

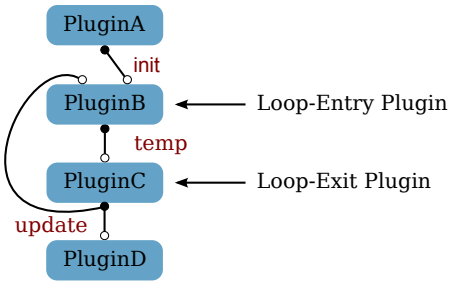

Fig. 5: An archetypal loop execution is depicted. PluginB and PluginC handle the loop logic. The loop is only exited, if the internal logic of the loop-exit plugin, PluginC in this case, decides so.

provided by a library [12], [13], which provides an application programming interface (API) and therefore can be directly utilised from within an implementation, like a ViennaX plugin.

However, in the case of implementing a plugin wrapping standalone simulation tools, the situation changes considerably. The availability of an API destined to work for external access cannot be expected, as the implementation of the simulation application does not require such a convenience layer. In the field of TCAD, standalone simulation applications typically process an input configuration file used for driving the overall simulation, for instance, the Archimedes project [5]. The implementation of the application can be typically directly transferred to a ViennaX plugin. However, the challenge is to handle the configuration of the simulator. A feasible, non-intrusive approach is to implement an additional ViennaX plugin which generates the configuration file based on certain parameters for the external simulation plugin (Fig. 6). A template configuration is stored as a string datatype, where certain parameters are replaced to alter the configuration of the simulation tool. We consider the overhead of generating and processing the simulation file as negligible, as the process has to be performed only once per execution and, above all, the computationally expensive part is the actual simulation. This approach depicts the versatility of our plugin system, as available standalone simulation tools can be utilized with a minimum of effort by simultaneously enabling to actively adjust simulation parameters during the course of the overall ViennaX execution.

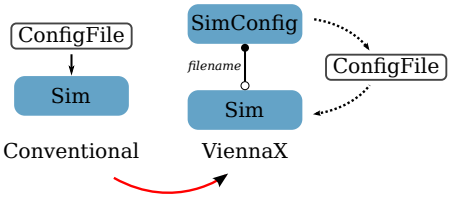

Fig. 6: A standalone simulation application driven by an input configuration file can be utilized by ViennaX by introducing an additional plugin for generating the configuration file.

V. Results

We depict the applicability of our approach for the field of TCAD based on the simulation of current-voltage characteristics of a two-dimensional pn-junction diode, however, any other device can be handled in the same manner. The influence of different doping levels is investigated. An available device simulation implementation is utilized by wrapping it as a plugin. The plugin instances compute the current for a specific voltage and forward it to a dedicated current-voltage characteristics plugin, post-processing the results (Fig. 7).

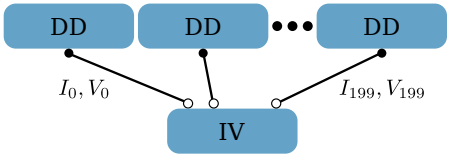

Fig. 7: 200 instances of the Drift-Diffusion (DD) based classical device simulation plugin are used to compute the currents for given contact voltages. The individual results are collected by a current-voltage characteristics plugin (IV).

We evaluate the current for 200 contact voltage levels for one doping setup, thus requiring 200 executions of the device simulation plugin. ViennaX is able to parallelize these plugins efficiently without any user interaction, as there are no input dependencies. The simulations are carried out on three quad-core consumer-level workstations running a 64-Bit Linux and sharing a Gigabit network connection. Fig. 8 depicts the scaling behavior of the simulation setup. As the communication is limited to a single data tuple for each device simulation, there is no significant communication overhead, thus linear scaling is achieved. Fig. 9 shows the current-voltage characteristics for different doping levels. As can be seen from our results, perfect scalability is achieved.

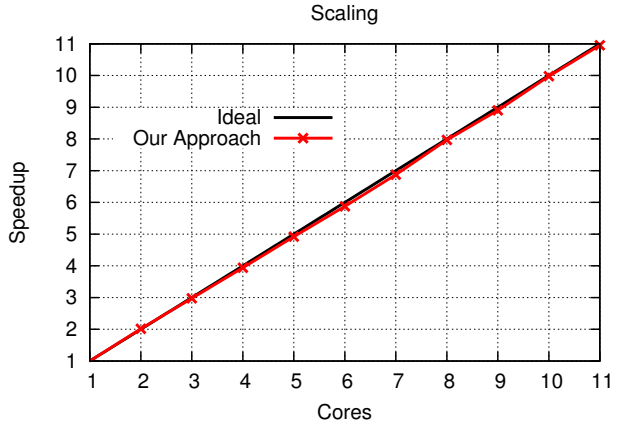

Fig. 8: The scaling behavior of the parallelized execution is shown. Due to negligible communication, the scaling is linear.

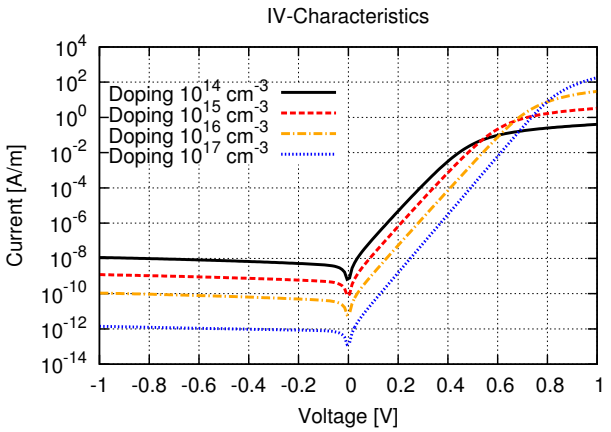

Fig. 9: The current-voltage characteristics of a two-dimensional pn-junction diode for different doping levels are shown.

VI. Summary

We have discussed the challenges of combining high-performance TCAD applications and introduced our approach to tackle them by the means of ViennaX. Our framework is capable of executing arbitrary available simulation tools, with different parameters, in a sequential and loop-wise manner. Our MPI-based execution model offers excellent scalability, as has been shown by a representative simulation result.

Acknowledgments

This work has been supported by the European Research Council through the grant #247056 MOSILSPIN. Karl Rupp acknowledges support by the Austrian Science Fund (FWF), grant P23598.

References

[1] T. Quintino, "A component environment for high-performance scientific computing," Ph.D. thesis, Katholieke Universiteit Leuven, 2008.
[2] D. Reid, C. Millar, G. Roy, S. Roy, and A. Asenov, "Analysis of threshold voltage distribution due to random dopants," *IEEE Transactions on Electron Devices*, vol. 56, no. 10, pp. 2255–2263, 2009.
[3] "nanoHUB." [Online]. Available: http://nanohub.org/
[4] "tiberCAD." [Online]. Available: http://www.tibercad.org/
[5] "Archimedes." [Online]. Available: http://www.gnu.org/software/archimedes/
[6] "COOLFluiD." [Online]. Available: http://coolfluid.github.com/
[7] "ViennaX." [Online]. Available: http://viennax.sourceforge.net/
[8] "Boost." [Online]. Available: http://www.boost.org/
[9] G. D. Reis and J. Järvi, "What is generic programming?" in *Proceedings of the 1st International Workshop on Library-Centric Software Design (LCSD)*, 2005.
[10] "The Intel Threading Building Blocks." [Online]. Available: http://threadingbuildingblocks.org/
[11] D. Kharrat and S. Quadri, "Self-registering plug-ins: An architecture for extensible software," in *Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2005, pp. 1324 –1327.
[12] "ViennaCL." [Online]. Available: http://viennacl.sourceforge.net/
[13] "Trilinos." [Online]. Available: http://trilinos.sandia.gov/