Automatically Optimized Component Model Computation for Power System Simulation on GPU

Marcel Mittenbühler*, Junjie Zhang*,†,§ and Andrea Benigni*,†,‡
* IEK-10: Energy Systems Engineering
Forschungszentrum Jülich, 52428 Jülich, Germany
{m.mittenbuehler, ju.zhang, a.benigni}@fz-juelich.de
† RWTH Aachen University, 52056 Aachen, Germany
‡ JARA-Energy, Jülich 52425, Germany

***Abstract*—This work provides an approach that automatically optimizes the component computations on graphics processing unit (GPU) devices from different vendors. The approach consists of a two-level optimization, where the first level considers the linear part of the computation for vectorization and applies mixed matrix formats to increase computational throughput further. Then, the second optimization level treats the combination of linear and non-linear parts as a black box and searches for the optimal configuration of parameters such as the degree of vectorization, the combination of matrix formats, and the group (of threads) sizes during parallel execution on GPU. Moreover, we also introduce constraints that reduce the optimization procedure's execution time. Finally, we select three different types of components that could be representative to computational tasks in power system and perform our optimization approach on these kernels. The computational performance is compared with unoptimized baseline and sparse linear algebra library based implementations, result shows that our optimization leads to better performance and more efficient memory utilization.**

***Index Terms*—Automatic Code Generation; Graphics Processing Unit; Parallel Processing; Power System Simulation; Sparse Matrices**

I. Introduction

To meet the need for a global carbon neutrality goal by 2050, the installation of renewable energy resources like photovoltaic and on- and off-shore wind farms has been boosted in recent years. This trend rapidly expands the power system's size and complexity, giving more challenges to power system simulation techniques.

A power system can be described by the differential equations of the components coupled by electrical connections. This results in a large system of differential equations that must be solved, usually numerically. The increasing number of components results in a dramatic growth of the problem size. At the same time, the fast dynamics brought on by the power electronic devices make it increasingly necessary to simulate the power system on a smaller time scale. These reasons have led to the development of different parallel simulation approaches to improve simulation performance.

The optimization technique proposed in this work is by exploiting parallel programming techniques to improve the performance of GPU code for power system component models automatically. We consider the code is already implemented under certain parallel simulation algorithms from the parallel-in-space family [1]–[3]. The main idea of the parallel-in-space based methods is to decouple the solution of the dynamic components and the solution of the network. This enables parallel processing on hardware accelerators such as GPUs and field-programmable gate arrays (FPGAs). In this work, we focused only on the performance optimization on GPUs because different types of hardware accelerators require different optimization techniques. The main contributions of this work are:

- We use automatic vectorization techniques on linear algebraic operations whilst not requiring global synchronization and maintaining good data locality.
- We optimized memory access by considering a variety of matrix formats and storage strategies. Different formats can be combined in a single component kernel allowing more freedom in performance optimization.
- We use automatic benchmarking and automatic code generation technique to allow the performance being tuned automatically for new components and on any compatible platform and hardware. We also introduced a strategy to reduce exploration space so that the benchmarking time is limited.

The paper is organized as follows: Sect. II presents related approaches and positions our work among other methods. Sect. III introduces the background related to heterogeneous computing concepts and execution models on the GPU. Sect. IV describes our approach to process component models on the GPU with vectorization, sparse formats, and automatic optimizations. Sect. V evaluates the achieved speedup based on different component models and Sect. VI summarizes the paper and discusses further work.

II. Related Work

The application of parallel-in-space-based algorithms can be traced back as far as the late 70s [4], where the au-

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Grant 450829162.
M. M. and J. Z. contributed equally to this paper.
§ Corresponding author.

thors split the solution for the ordinary differential equations (ODEs) of synchronous machines and the network equations in the transient stability analysis (TSA) programs. More recent developments and implementations of parallel-in-space type approaches are looking into using hardware accelerators for dynamic simulations, e. g., GPUs [5]–[9] and FPGAs [10]–[13]. In [5], the authors introduce a transformation on different component computations so that a single compute kernel can be used to execute them; moreover, automatic code generation is applied to build the compute kernel. In [6], the overall equations are grouped into linear and nonlinear equation systems; afterwards, the linear and nonlinear systems are decomposed separately and executed using different kernels. Therefore, there are no specific component kernels but general kernels to compute the fine-grained partitioned sub-networks in parallel.

In [8], [9], kernels are implemented for different components and the network separately so that instances of each type of component are calculated in parallel in a single instruction/multiple threads (SIMT) manner. Besides, kernels are implemented for the same type of sub-components inside a component. For instance, in [7], the parallelization is applied on a single component, where the compute kernels are designed to parallelize the computations of sub-modules in a Modular Multi-Level Converter (MMC).

Automatic optimization is achieved by executing performance benchmarks with different presets, and finally finding the best-performing preset for the given program. This is needed since the optimization of scientific computing code is platform-specific, so the optimization for a given hardware architecture is likely to slow-down on other platforms. Traditionally, the code would need to be tuned for each architecture by experienced developers with domain-specific knowledge [14]. Such process is time-consuming, and with the continuous evolution of computer architecture, the process need to be repeated constantly. Therefore, automatic tuners have been adopted in scientific computing community for decades [14], [15], where the ATLAS project could be the best well-known example [14], which uses auto-tuning technique to generate near-optimal code for dense basic linear algebra subprograms (BLAS) routines.

III. Background

Heterogeneous computing frameworks such as OpenCL, CUDA or HIP shares a similar concept in their hardware abstractions, which contains three levels: *device*, *compute unit* (CU), and *processing element* (PE). Illustration of the three levels as well as the memory hierarchy are shown in Fig. 1. To map the hardware abstraction with the actual hardware, take Nvidia GPU as an example, where the Nvidia *streaming multiprocessors* (SMs) are the CUs, which is composed of the *CUDA cores*, being their PEs; and the whole GPU is represented as a device.

During parallel execution, the finest process granularity is a *work-item* (or *thread* in CUDA/HIP), which executes the code written in *kernel* in parallel on the device; a collection of work-items that are scheduled together and executed on the same

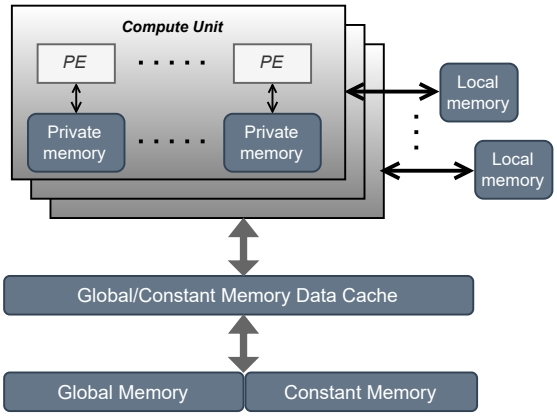

Fig. 1. Hardware abstraction of heterogeneous computing frameworks like OpenCL, CUDA or HIP. Illustration using terminology of OpenCL.

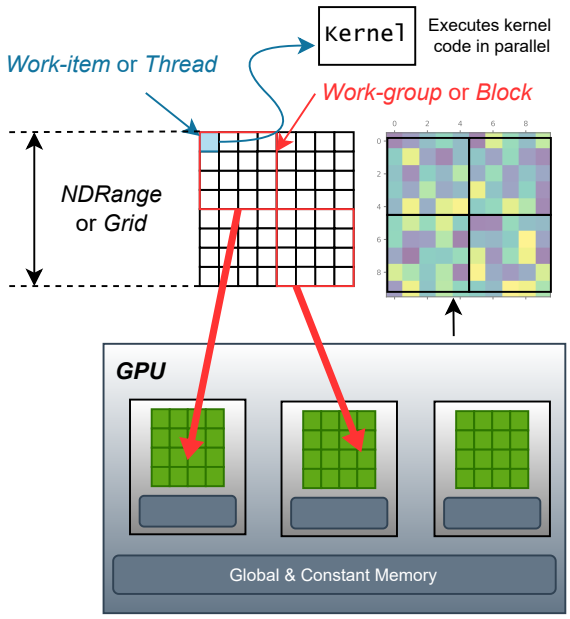

Fig. 2. Illustration of the parallel execution model [16], based on terminology from OpenCL and CUDA/HIP.

CU is a *work-group* (or *block* in CUDA/HIP), work-items in the same work-group can share data via the local memory if needed. Finally, a collection of work-groups forms *NDRange* (or *grid* in CUDA/HIP) which represents all work-items that is spawned during execution of a kernel. An illustration of the execution model is shown in Fig. 2.

IV. Approach

The overall optimization process is illustrated in Fig. 3. By providing the kernel as input to our optimizer, the optimizer executes benchmarks and finally generate code with optimal execution parameters, i. e. NDRange size, work-group layout, etc., automatically. The optimization needs only to be executed once for a given given set of models (kernel optimization is independent from the model parameters and operating points),

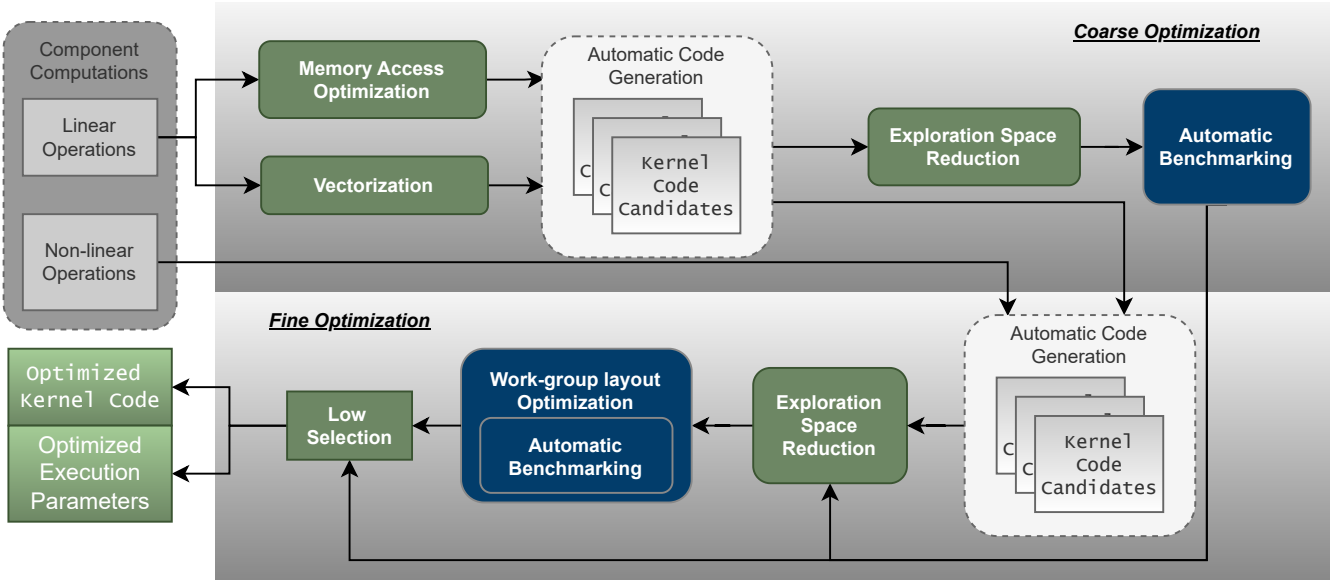

Fig. 3. Illustration of the overall optimization process.

and the outcome, i. e., optimized kernel code and execution parameters, can be stored and reuse for later simulations.

The overall process consists of a two-level optimization process, where the first level considers the possibility of vectorization inside component kernels and applies mixed matrix formats with different matrix storage strategies to increase the computational throughput further. Therefore, the second optimization level treats the combination of linear and nonlinear parts as a black box and searches for the optimal configuration of parameters such as the degree of vectorization, the combination of matrix formats, and the group (of threads) sizes during parallel execution on GPU.

To perform vectorization inside component kernels, we take the discretized ODE of the components and separate them into linear and nonlinear contributions. The computation of the model's linear part is based on matrix-vector multiplication (`mv`) operations. A first optimization level can already be achieved by applying vectorization on the `mv` operations. Moreover, since the `mv` operation is limited by memory transfer speed, applying sparse matrix formats boosts the performance when the associated matrices are sparse [17]. The nonlinear part includes any other operations that is not or difficult to be treated as linear algebraic. By combining the two optimization levels, our implementation provides a framework that automatically benchmarks different parameters and selects the best-performing one. Furthermore, we also introduce an algorithm in Sect. IV-C to reduce the exploration space during automatic benchmarking by constraining the possible parameters.

A. Vectorization

Efficient vectorization requires a regular structure in computations to be efficient. A typical example of such a regular structure is linear algebra operations. In fact, GPUs are mostly designed and optimized for these operations, and automatic vectorization can be easily achieved by processing each dimension with a different thread.

Automatic vectorization is difficult to apply to power system components as they mostly contain nonlinearities. Usually, after discretization, a large portion of the required computations can be formulated into linear algebra operations with a small nonlinear part remaining. Therefore, processing a vast part of the nonlinear component in parallel is possible, whilst a residual part is computed sequentially.

To perform vectorization inside component kernels, the ODE of the components are reformulated similarly to a state-space representation as

$$\dot{x}(t) = A(t)x(t) + B(t)u(t) + \chi_{\dot{x}}(t, x(t), u(t)) \quad (1)$$

$$y(t) = C(t)x(t) + D(t)u(t) + \chi_y(t, x(t), u(t)). \quad (2)$$

with time-dependant input $u \in \mathbb{R}^{N_u}$, state $x \in \mathbb{R}^{N_x}$, output $y \in \mathbb{R}^{N_y}$, component matrices $A \in \mathbb{R}^{N_x \times N_x}$, $B \in \mathbb{R}^{N_x \times N_u}$, $C \in \mathbb{R}^{N_y \times N_x}$, $D \in \mathbb{R}^{N_y \times N_u}$, and nonlinearities $\chi_{\dot{x}}$ and χ_y. Fig. 4 visualizes the interactions of the linear contributions in a block diagram. Effectively, this splits the component computation into linear and nonlinear contributions where the linear contribution can be seen as a state space model.

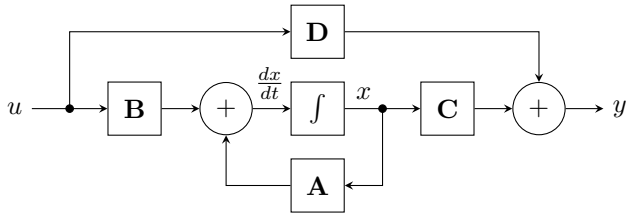

Fig. 4. Block diagram of the model described by (1) and (2) when setting the nonlinearities $\chi_{\dot{x}} = \chi_y = 0$.

The nonlinear computations of these functions are treated as a black box, and are almost impossible to vectorize efficiently. Therefore, each component's nonlinearity is processed by one thread. This also simplifies the implementation as parallelism can be ignored. However, it is still worth to be noted that this sequential execution only applies to a single component instance, the nonlinear parts of different component instances are still computed in parallel. In practice, nonlinearities are computed by callbacks that are invoked between each computation in the state space model as shown in Alg. 1. These callbacks can alter the states, inputs, and outputs for maximal flexibility. Moreover, it is possible to introduce additional arguments to the kernel when needed. For example, a turbine may have a mechanical simulation alongside an electrical one with a shared buffer. With the additional arguments, sharing the turbine frequency with the electrical simulation would be possible.

Algorithm 1 Pseudo-code for linear part of component kernels. Nonlinear part can be added as additional callbacks anywhere.

1: PRECALLBACK$(t, x, y, u, ...args)$
2: $\dot{x} \leftarrow Ax + Bu$
3: DERIVATIVECALLBACK$(\dot{x}, t, x, y, u, ...args)$
4: $x \leftarrow$ INTEGRATE$(x, \dot{x}, t, step)$
5: NEXTSTATECALLBACK$(t, x, y, u, ...args)$
6: $y \leftarrow Cx + Du$
7: OUTPUTCALLBACK$(t, x, y, u, ...args)$

B. Memory Access

The main computation of the linear part of components is matrix-vector multiplication which is largely limited by the bandwidth of the device [18]. By reducing the amount of data that has to be transferred, the overall computation can be accelerated. Moreover, reducing memory requirements allows storing more components in memory, thereby enabling the simulation of larger systems.

A multitude of different sparse formats have been explored in the past with various kernel implementations [17], [19]–[22]. All formats have in common that they try to minimize the number of stored zero elements. The main difference is the method used to store the positions of the remaining non-zero elements. Each format performs differently depending on the sparsity pattern of the matrix and the architecture of the GPU.

Of course, the nonzero structure of the component matrices can vary vastly, although they are for the same component. Thus, each matrix can have its own format in our implemented kernels. Moreover, we also convert matrices that are zero or the identity matrix to a scalar, so that they do not necessarily to be stored as matrices.

When considering matrix storage, except utilizing different sparse formats like ELLPACK format (ELL) [23], compressed sparse row (CSR) [24], Coordinate list (COO), etc., we proposed following strategies to store the matrices of multiple

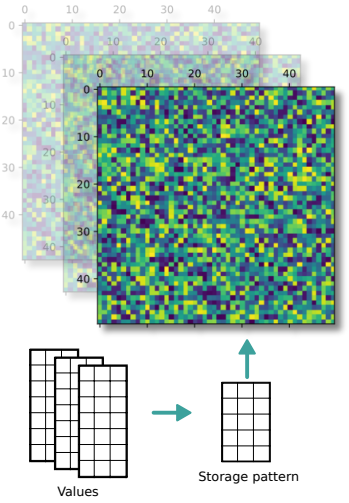

Fig. 5. The pattern storage strategy. Assuming the nonzero pattern for certain component doesn't change during simulation, the nonzero pattern and values are stored separately. Multiple instances of the same component type shares the same storage pattern vector and maps to different value vectors.

component instances. Nevertheless, new sparse formats and strategies can be easily integrated.

1) Block-Diagonal storage: The state space equations are decoupled; therefore, the equations for all component instances can be combined into one. This results in block diagonal coefficient matrices that can be efficiently stored using sparse formats. Each thread can then compute the required rows of the matrix-vector multiplication.

2) Concatenated storage: Not all formats can be stored efficiently in a block diagonal form. For example, storing a block diagonal matrix in a dense format is inefficient. Moreover, some other formats also perform better, or even require (e. g., COO), to store instance individually. In this case, the matrices for each instance are encoded into certain matrix format (dense or sparse format) and then concatenated into one buffer, an additional buffer is used to locate the individual instances.

3) Pattern storage: The nonzero pattern of the component matrices is usually determined by the algebraic description of the component's behavior, implying that it suffices to store this pattern once and reuse it for all instances of that component. By just storing the pattern, and allowing instances to have different values, this reduces memory transfers significantly as only the values of the component matrices are transferred, not the pattern. The downside of this approach is that the component must have a known and fixed nonzero pattern that can be determined during optimization and remains fixed in simulation, but it is usually the case.

C. Exploration space reduction and optimization

The linear part is vectorized, allowing multiple threads executing the same `mv` operation. Normally, one thread is assigned to process each row, however, when there are still resource available, i. e., if there are more threads available than

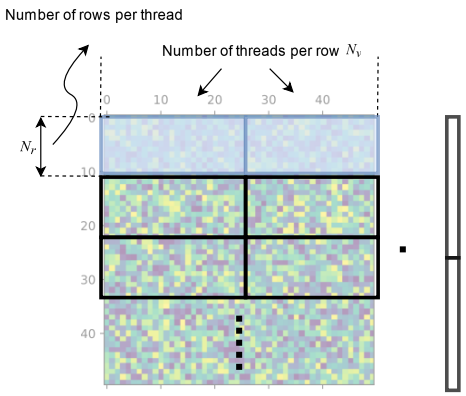

Fig. 6. Illustration of two of the vectorization parameters N_v and N_r on the matrix-vector multiplication (mv). When more than one thread work on the same row, a parallel binary reduction is used to collect the sum for each row.

the number of rows, then multiple threads process each row simultaneously. In this case, the element-wise products in each row are summed using a binary reduction.

To quantify the constraint for exploration space reduction, we first limit the number of threads per row to be a power of two. Then we introduce the number of rows per thread N_r and the number of threads per row during vectorized mv execution N_v, as shown in Fig. 6 as

$$N_r = \left\lceil \frac{N_m N_j}{N_g} \right\rceil,$$
$$N_v = 2^{\left\lfloor \log_2 \frac{N_g}{N_m N_j} \right\rfloor},$$

with N_m rows per matrix, a group size of N_g, and N_j components per group. Furthermore, the maximal number of threads per row $\hat{N}_v$ for a matrix with N_n columns is

$$\hat{N}_v = 2^{\lceil N_n \rceil}.$$

The optimization for component kernels have six degrees of freedom. The matrix format choice for the component matrices introduces four parameters. In addition, a kernel should be built for a certain group size N_g on the GPU, i. e., the number of SIMT threads working in sync. Finally, the number of instances processed by one group N_j must be selected.

By design, the device limits the group size N_g to $1 \leq N_g \leq \hat{N}_g$ with the upper limit group size $\hat{N}_g$. Furthermore, the group size should be a multiple of two for efficient scheduling. For example, the NVIDIA A100 supports group sizes of up to 1024 threads, resulting in just 10 efficiently usable group sizes. Also, we assume that $1 \leq N_j \leq N_g$ – i.e., each thread computes at most one component – to stay within the regime of thin threads for better work distribution and less memory consumption [25].

Our goal is to minimize idle time among threads, as this can increase computation times. Thus, we maximize the number of components per group N_j so that the work performed per thread stays constant. Effectively, this should remove inefficient parameter sets. In practice, we loop over all N_j for a given group size N_g and only benchmark those where $N_j + 1$ results in a different N_r or N_v.

With these prerequisites, we can define a function f

$$f : (N_m, N_n, \hat{N}_g) \to \{(N_g, N_j)\} \tag{3}$$

with N_m rows and N_n columns of the considered matrix that maps the external parameters given by the matrix dimensions and the GPU to a set of parameters fulfilling the above-mentioned constraints. As four matrices describe each component, the set of suitable parameters $\mathcal{P}$ is given by

$$\mathcal{P} = \bigcup_{x \in \{A,B,C,D\}} f(N_m^{(x)}, N_n^{(x)}, \hat{N}_g). \tag{4}$$

For the parameter selection, we split the optimization into a coarse optimization that benchmarks one part at a time and a fine-grained optimization to select the best combination of parameters as shown in Fig. 7. This two-step process significantly lowers the required optimization time.

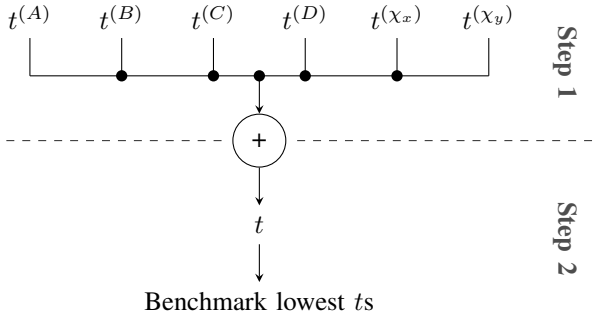

Fig. 7. Visualization of the optimization flow. The first phase benchmarks each part isolated. The second part then combines the results to predict runtimes for combinations and benchmarks the ones with the lowest predicted runtimes.

1) Coarse Optimization: For the coarse optimization, we benchmark each component of Alg. 1 individually to try out each format for each matrix. Assume F is the matrix format, and x identifies the matrix, then the runtime for the matrix-vector multiplication is $t_F^{(x)}$ and depends on the total number of instances N_i, as well as the group size N_g, and the number of components per group N_j. We determine this by running a benchmark, as runtime prediction is generally a hard problem [26]–[28]. The same benchmark is performed only with the nonlinearities to get $t^{(\chi_x)}$ and $t^{(\chi_y)}$.

2) Fine Optimization: In general, the optimization goal is to minimize the kernel's runtime. This can be estimated by assuming the runtime t of the combined kernel consists of the runtimes of the parts benchmarked in the coarse optimization as

$$t \equiv t^{(A)} + t^{(B)} + t^{(C)} + t^{(D)} + t^{(\chi_x)} + t^{(\chi_y)}. \tag{5}$$

Using this, we find the parameter configurations expected to perform best. Nevertheless, this is only an approximation, as some other influences from the combination may change the runtime slightly. Moreover, predicting runtimes on GPU is generally difficult, imprecise, or computationally expensive.

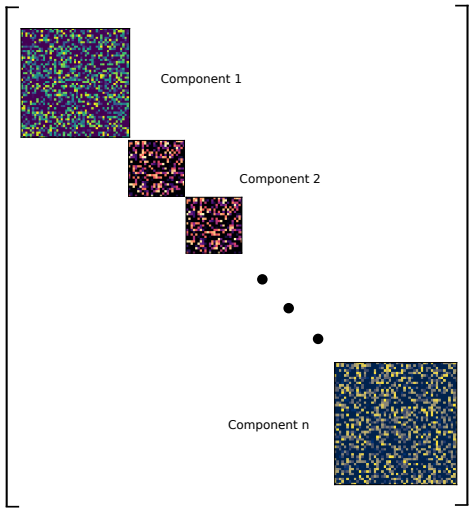

Fig. 8. Illustration of the library implementation to represent computations in generic solvers, e. g. in [29]

Therefore, we benchmark some of the predicted configurations and select the one with the lowest runtime.

V. Evaluation

We evaluate our approach using three representative components with different models – in particular, we consider a distributed generation (DG) inverter, an electrolyzer, and a synchronous machine. The time integration of (1) uses the explicit Euler method; however, an implementation for RK-4 also exists. The correctness of our kernels for all matrix formats, group sizes, and components per group was verified with negligible computation errors.

Simulations in this section are compiled and executed on a server with two AMD EPYC 7H12 CPUs (2.6 GHz base clock frequency, 64 cores each, hyper-threading disabled); 256 GB DDR4 main memory; one Nvidia A100-40 GB GPU with 40 GB HBM2 global memory, and one AMD MI100 GPU with 32 GB global memory.

To evaluate the performance of the optimized kernels, we prepare two alternative implementations to compare:

1) the library implementation: Previous works on power system simulation with GPU show that component computations can be transformed and aggregated into a unified kernel. Moreover, general purpose numerical solver libraries [29], [30] will usually also aggregate all differential equations and solve simultaneously, e. g., by calculating the Jacobian, and computed with the help of a BLAS library. An example to this is the SUNDIALS library with GPU acceleration [29]. The benefit of using general solvers is that the user, or modeller, needs less concern regarding computational performance, since the computations are relying on other linear algebra libraries. However, this poses difficulty in implementing new models or power system specific models that are essentially an algorithms, e. g., specific controllers, automation systems, etc. Therefore, to represent the computation resulted from this approach, the linear part of the considered component models are aggregated into a single state-space representation, where the coefficient matrices of each component are placed along the diagonal of the aggregated coefficient matrix, as shown in Fig. 8. The numerical integration will then be processed by a vendor-supplied sparse BLAS library – so that the many off-diagonal zeros do not affect the computation – such as cuSparse or hipSparse. These libraries are highly optimized as they are utilized in many simulation environments. It needs to be pointed out that for simplicity, the nonlinear part of each model is ignored for this implementation.

In addition, to ensure a fair comparison, since the tested sparse libraries only support fixed sparse matrix format throughout the computation, prior to each benchmark, we tested among different matrix formats to find the most performant format for each library and use it in that benchmark.

2) the baseline implementation: We take the kernel implementations with the formulation in Sect. IV-A, i. e., the reformulation into a linear and nonlinear part, without applying any optimization. The group size and components per group are set to $N_g = 32$ and $N_j = 32$, respectively. Such group size matches the recommended default group size on the considered devices as it matches the SIMT width of one compute unit. This is a versatile configuration that should work well in many cases.

The kernels for the baseline implementation as well as the optimized code uses OpenCL C; library implementations only need to invoke the related Application Programming Interface (API) calls from the host, which is implemented in C++ in our case.

A. Component Models

1) Distributed generation inverter: We take the DG inverter introduced in [31]; it is modeled by an averaged inverter model with a grid following control and neglects the switching dynamics. The kernel is also implemented in our previous work [32]. Such model depth is already sufficient for a majority of test cases [33] involving power electronics. The controller and circuit representation of the inverter model is shown in Fig. 9. The input three-phase alternating current (AC) signal to the controllers is first transformed into the dq domain via a Park transform. The controller can be divided into three main parts: a phase-locked loop (PLL) tracks the system's angular frequency; an average power calculation block that calculates the current output power, and two PI controllers that track reference power set points by controlling the output voltage of the converter.

The converter controllers, excluding the Park transform block, can be formulated via the following state-space formulation:

$$\dot{x} = Ax + B(x)u, \tag{6}$$

$$y = Cx + Du, \tag{7}$$

where the B matrix is dependent on the state x. This is due to the average power calculation block that performs multiplications over states (v_{c-dq} and i_{g-dq}) to calculate

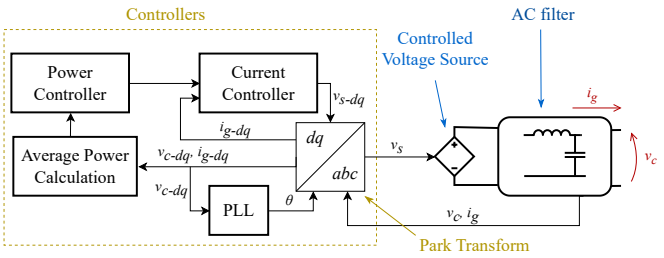

Fig. 9. Distributed generation inverter model [31], [32]

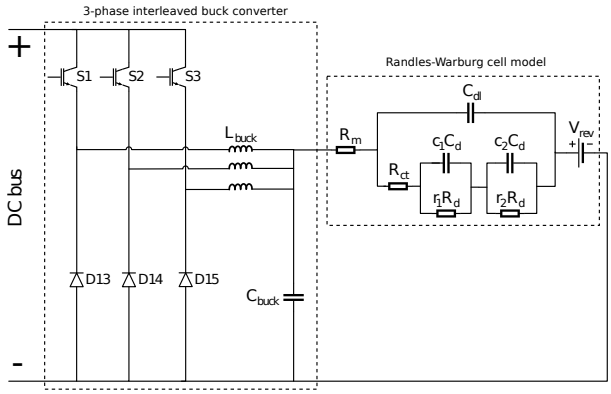

Fig. 10. Electrolyzer with three-phase interleaved buck converter [34], [35]

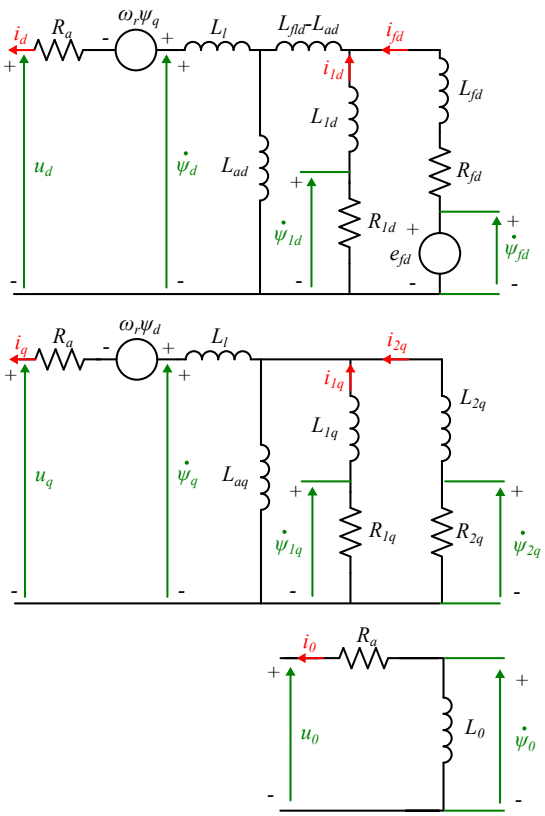

Fig. 11. The dq0-axis equivalent circuit of the synchronous machine [36]

the power, thus introducing non-linearity in the state-space representation.

2) Electrolyzer: We use the electrolyzer model in [34], where the electrolyzer is considered to be connected with a three-phase interleaved buck converter. The electrolyzer is modeled with Randles-Warburg (RW) cell model, and the buck converter uses the generalized state-space average model introduced in [35]. The reason for selecting an electrolyzer model for the benchmark is twofold: first, electrolyzers are gaining more attention due to the increasing interest in hydrogen; second, the main computational load in this model we selected is the interleaved buck converter, therefore, it could be used to partly represent the computational tasks when simulating a power-electronics-based system with switching dynamics.

3) Synchronous Machine: We take the machine model in [36] with saturation ignored, and its dq0-axis equivalent circuits are shown in Fig. 11. Similar to the previous inverter model, the machine is modeled in the dq frame as well. The machine model also introduces non-linearity due to the coupling of $d-$ and $q-$axis and between electromagnetic and mechanical equations. The overall equation set can be represented by:

$$\dot{\Psi} = f(\Psi, \omega_r, U), \tag{8}$$

$$\dot{\delta_r} = f(\omega_r), \tag{9}$$

$$\dot{\omega_r} = f(\Psi, I, \omega_r), \tag{10}$$

$$0 = g(\Psi, I). \tag{11}$$

Where Ψ is a 7×1 vector of flux linkage; U, I are vectors of stator and rotor voltages and currents with the same length, respectively. ω_r is the mechanical angular frequency of the rotor.

B. Performance Evaluation

We benchmarked our optimized component kernels against the library and the baseline implementation by performing numerical integration with a simple Euler forward method. The measured execution time for different component types and different component counts are shown in Fig. 12, including a benchmark of simulating a combination with three types of components together with an equal number of each type. The speedup of our optimized kernels to the library implementation is between 1.3 and 6.7 times and to the baselines by up to 10.2 times, which shows that the optimized kernels outperform the compared implementations by some margin.

Compared with the library implementation, the customized component kernels, i. e. the baseline and optimized, have a reduced number of kernel launches, as our kernels perform the whole computation in a single kernel launch although with part of computations being computed sequentially, i. e. nonlinear contributions, whereas the library implementation requires separated launches e. g. for updating the states and outputs. This results in better cache coherence and less overhead in our implementations. Nevertheless, the library implementation still outperforms the baselines when the problem size is large enough.

In all test cases of the library implementation, the NVIDIA A100 outperformed the AMD MI100. It can be attributed

to the better optimization with the cuSparse library than the hipSparse library. However, it needs to be noted that the A100 GPU has higher memory bandwidth, therefore, it gains more advantage in the linear algebra related benchmarks which are mainly memory-bounded operations.

Finally, we performed a roofline analysis [37] with the optimized kernels. This relates the computational intensity, given by FLOP per byte transferred, to the achievable performance. The plot is shown in Fig. 13 and shows that especially high component counts lead to efficient utilization of the hardware and close to the peak performance. The devices are likely not fully utilized for lower component counts. This is because runtime overheads, such as scheduling time, become a significant factor compared to light computational load, leading to a large gap between peak and actual performance.

C. Memory consumption

Reformulation of the components inevitably leads to increased memory consumption, which could be a limiting factor for very large systems. We tracked the required buffer sizes needed for each component to perform numerical integration and compared them between the optimized and library implementation, as shown in Fig. 14. The electrolyzer model consumes the most memory per component since it has a more detailed converter model considering switching dynamics; the DG inverter considers only controller dynamics and therefore needs less memory but is still larger than the synchronous machine. Results show that our optimizer finds different matrix format combinations for different problem sizes: when the component count is small, the GPU memory bandwidth is usually not saturated, hence leading our optimization to choose more performant formats, including dense, regardless of memory usage, and the increasing memory usage clearly shows that dense format was used in some cases. With the growing component count, the GPU memory bandwidth is eventually saturated, therefore, the optimizer tends to find format combinations that provide better compression on the matrices.

VI. Conclusion

This work provides an approach to automatically accelerate the numeric integration of component models for power system simulations on GPU. Our approach automatically exploits the data parallelism of the GPU by introducing vectorization and different matrix storage strategy with mixed matrix formats into the component computation. The approach can be flexibly applied to kernels for any new components or be applied to existing compatible implementations to improve performance.

We demonstrated that our approach outperforms the aggregated model approach based on sparse linear algebra libraries with a speedup between 1.3 and 6.7 times, and up to 10 times compared to the unoptimized baseline implementation, demonstrating the effectiveness of our optimizations.

With the growing size and complexity of the power system, or when specific study cases require detail modelling, component models can be more complex. Therefore, memory bandwidth and, in some cases, memory volume will become a limiting factor. The memory footprint shown in Sect. V-C suggests that our approach could increase the efficiency in memory utilization for component computations.

Nevertheless, it needs to be pointed out that the optimization procedure takes a few minutes per component since it depends on many automatically executed benchmarks, and need to be re-executed for different problem sizes, indicating further improvements needed, e. g., on the optimization algorithm.

We plan to consider different explicit and implicit integration schemes other than the explicit Euler and RK-4 schemes. Moreover, one may consider the vectorization potential of the nonlinearities to achieve even better performance. Including more specialized matrix formats for the component matrices may increase performance further.

References

[1] A. Benigni and A. Monti, "A parallel approach to real-time simulation of power electronics systems," *IEEE Transactions on Power Electronics*, vol. 30, no. 9, pp. 5192–5206, sep 2015.

[2] C. Dufour, J. Mahseredjian, and J. Bélanger, "A combined state-space nodal method for the simulation of power system transients," *IEEE Transactions on Power Delivery*, vol. 26, no. 2, pp. 928–935, apr 2011.

[3] V. Jalili-Marandi and V. Dinavahi, "SIMD-Based Large-Scale Transient Stability Simulation on the Graphics Processing Unit," *IEEE Transactions on Power Systems*, vol. 25, no. 3, pp. 1589–1599, Aug. 2010.

[4] V. Brandwajn, "Synchronous generator models for the simulation of electromagnetic transients," Ph.D. dissertation, University of British Columbia, 1977.

[5] Y. Song, Y. Chen, S. Huang, Y. Xu, Z. Yu, and W. Xue, "Efficient gpu-based electromagnetic transient simulation for power systems with thread-oriented transformation and automatic code generation," *IEEE Access*, vol. 6, pp. 25 724–25 736, 2018.

[6] Z. Zhou and V. Dinavahi, "Fine-grained network decomposition for massively parallel electromagnetic transient simulation of large power systems," *IEEE Power and Energy Technology Systems Journal*, vol. 4, no. 3, pp. 51–64, 2017.

[7] N. Lin and V. Dinavahi, "Exact Nonlinear Micromodeling for Fine-Grained Parallel EMT Simulation of MTDC Grid Interaction With Wind Farm," *IEEE Transactions on Industrial Electronics*, vol. 66, no. 8, pp. 6427–6436, Aug. 2019.

[8] Z. Zhou and V. Dinavahi, "Parallel Massive-Thread Electromagnetic Transient Simulation on GPU," *IEEE Transactions on Power Delivery*, vol. 29, no. 3, pp. 1045–1053, 2014.

[9] V. Jalili-Marandi, Z. Zhou, and V. Dinavahi, "Large-Scale Transient Stability Simulation of Electrical Power Systems on Parallel GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 7, pp. 1255–1266, 2012.

[10] M. Milton, A. Benigni, and J. Bakos, "System-Level, FPGA-Based, Real-Time Simulation of Ship Power Systems," *IEEE Transactions on Energy Conversion*, vol. 32, no. 2, pp. 737–747, 2017.

[11] M. Milton and A. Benigni, "Latency insertion method based real-time simulation of power electronic systems," *IEEE Transactions on Power Electronics*, vol. 33, no. 8, pp. 7166–7177, 2018.

[12] M. Milton, A. Benigni, and A. Monti, "Real-Time Multi-FPGA Simulation of Energy Conversion Systems," *IEEE Transactions on Energy Conversion*, pp. 1–1, sep 2019.

[13] L. Zhang, J. Liu, W. Qi, Q. Chen, R. Long, and S. Quan, "A parallel modular computing approach to real-time simulation of multiple fuel cells hybrid power system," *International Journal of Energy Research*, vol. 43, no. 10, pp. 5266–5283, 2019.

[14] R. Clint Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1, pp. 3–35, Jan. 2001.

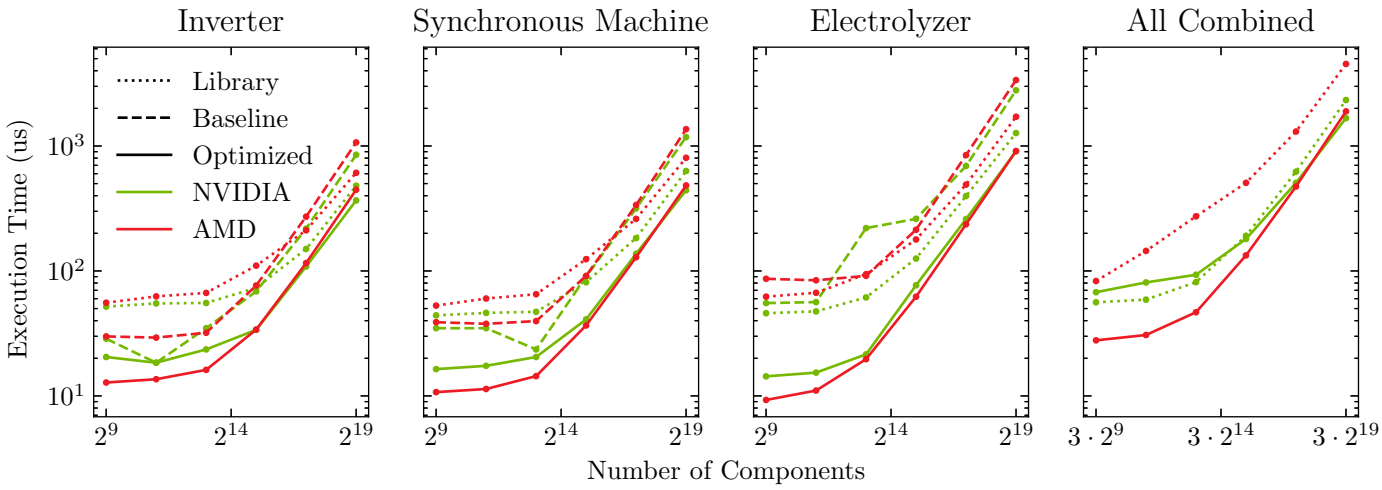

Fig. 12. Execution time for performing numerical integration with different component models and component counts, with simulating each type of component along or with all types combined.

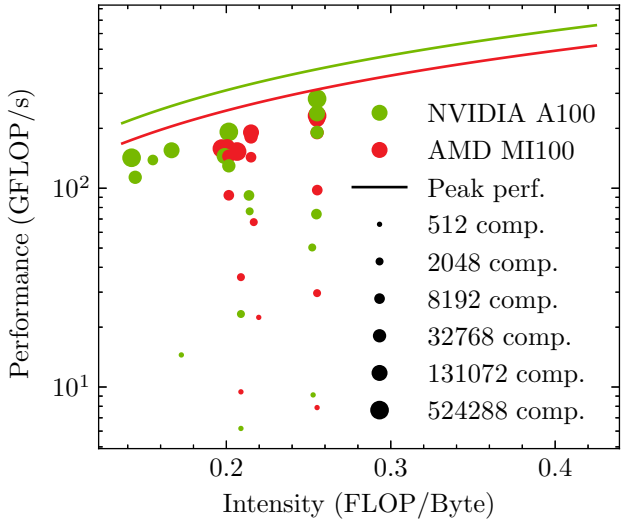

Fig. 13. Performance of the optimized kernels in a roofline plot to relate kernel performance to peak performance. The size of the dots relates to the number of components.

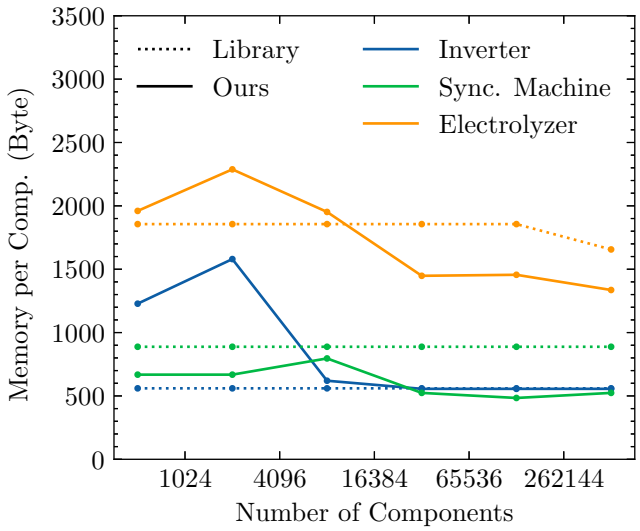

Fig. 14. Memory consumption per component on the NVIDIA A100 GPU of different component models, for our and the library implementation. Lower limit is calculated by considering minimum number of parameters required for each component.

[15] K. Sato, H. Takizawa, K. Komatsu, and H. Kobayashi, "Automatic Tuning of CUDA Execution Parameters for Stencil Processing," in *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, K. Naono, K. Teranishi, J. Cavazos, and R. Suda, Eds. New York, NY: Springer, 2010, pp. 209–228.

[16] D. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 2nd ed. Amsterdam: Elsevier, Morgan Kaufmann, 2013.

[17] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.

[18] G. I. Goumas, K. Kourtis, N. Anastopoulos, V. P. Karakasis, and N. Koziris, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," *The Journal of Supercomputing*, vol. 50, pp. 36–77, 2009.

[19] B.-Y. Su and K. Keutzer, "clspmv: A cross-platform opencl spmv framework on gpus," in *Proceedings of the 26th ACM international conference on Supercomputing*, 2012, pp. 353–364.

[20] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills, "Vectorized sparse matrix multiply for compressed row storage format," in *International Conference on Computational Science*. Springer, 2005, pp. 99–106.

[21] J. L. Greathouse, K. Knox, J. Poła, K. Varaganti, and M. Daga, "clsparse: A vendor-optimized open-source sparse blas library," in *Proceedings of the 4th International Workshop on OpenCL*, 2016, pp. 1–4.

[22] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *GPU Technology Conference*, 2010.

[23] D. R. Kincaid, T. C. Oppe, and D. M. Young, "Itpackv 2d user's guide," Texas Univ., Austin, TX (USA). Center for Numerical Analysis, Tech. Rep., 1989.

[24] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 339–350.

[25] G. Klingbeil, R. Erban, M. Giles, and P. K. Maini, "Fat versus thin threading approach on gpus: Application to stochastic simulation of chemical reactions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 280–287, 2011.

[26] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," *ACM sigplan notices*, vol. 45, no. 5, pp. 115–126, 2010.
[27] P. Guo, L. Wang, and P. Chen, "A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1112–1123, 2013.
[28] B. C. Lee, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply," in *International Conference on Parallel Processing, 2004. ICPP 2004.* IEEE, 2004, pp. 169–176.
[29] C. J. Balos, D. J. Gardner, C. S. Woodward, and D. R. Reynolds, "Enabling GPU accelerated computing in the SUNDIALS time integration library," *Parallel Computing*, vol. 108, p. 102836, Dec. 2021.
[30] S. Abhyankar, J. Brown, E. M. Constantinescu, D. Ghosh, B. F. Smith, and H. Zhang. PETSc/TS: A Modern Scalable ODE/DAE Solver Library.
[31] N. Pogaku, M. Prodanovic, and T. C. Green, "Modeling, Analysis and Testing of Autonomous Operation of an Inverter-Based Microgrid," *IEEE Transactions on Power Electronics*, vol. 22, no. 2, pp. 613–625, 2007.
[32] J. Zhang, M. Mittenbuehler, L. Razik, and A. Benigni, "Parallel Simulation of Power Systems with High Penetration of Distributed Generation Using GPUs and OpenCL," in *2022 IEEE 13th International Symposium on Power Electronics for Distributed Generation Systems (PEDG)*, 2022, pp. 1–6.
[33] G. De Carne, G. Lauss, M. H. Syed, A. Monti, A. Benigni, S. Karrari, P. Kotsampopoulos, and M. O. Faruque, "On modeling depths of power electronic circuits for real-time simulation–a comparative analysis for power systems," *IEEE Open Access Journal of Power and Energy*, vol. 9, pp. 76–87, 2022.
[34] H. Zhang, Y. Lu, J. Zhang, and A. Benigni, "Real-Time Simulation of an Electrolyzer with a Diode Rectifier and a Three-Phase Interleaved Buck Converter," in *2022 IEEE 13th International Symposium on Power Electronics for Distributed Generation Systems (PEDG)*, 2022, pp. 1–6.
[35] P. Azer and A. Emadi, "Generalized State Space Average Model for Multi-Phase Interleaved Buck, Boost and Buck-Boost DC-DC Converters: Transient, Steady-State and Switching Dynamics," *IEEE Access*, vol. 8, pp. 77 735–77 745, 2020.
[36] P. Kundur, N. J. Balu, and M. G. Lauby, *Power system stability and control*, ser. EPRI power system engineering series. New York: McGraw-Hill, 1994.
[37] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, Apr. 2009.

APPENDIX

TABLE I
OPTIMIZATION RESULT FOR NVIDIA A100-40GB GPU

Component	#Components	Group size	#Components per group	Matrix format with storage strategy A	B	C	D
Inverter	512	16	2	CDia*	CDia*	Dense-cat	ELL*
	2048	32	32	Dense-cat	Dense-cat	Dense-cat	Dense-cat
	8192	128	32	ELL*	ELL*	CDia*	ELL*
	32768	128	64	ELL*	ELL*	ELL*	ELL*
	131072	64	40	ELL*	ELL*	ELL*	ELL*
	524288	32	20	ELL*	ELL*	ELL*	ELL*
SyncMachine	512	128	14	CDia*	Dia*	ELL*	-
	2048	256	28	CDia*	CDia*	ELL*	-
	8192	256	28	CDia*	Dia*	CDia*	-
	32768	128	64	ELL*	ELL*	ELL*	-
	131072	64	32	ELL*	Dia-BD	CSR*	-
	524288	128	64	ELL*	CDia*	ELL*	-
Electrolyzer	512	256	12	ELL-BD	Dense-cat	CSR*	-
	2048	128	6	Dia*	ELL-BD	CSR-BD	-
	8192	128	12	ELL-BD	ELL-BD	CSR-BD	-
	32768	256	48	CSR-BD	ELL-BD	CSR-BD	-
	131072	64	6	CSR-BD	CDia*	CSR*	-
	524288	256	48	CSR-BD	CSR-BD	CSR*	-

*: pattern storage
-cat: concatenated storage
-BD: block diagonal storage

TABLE II
OPTIMIZATION RESULT FOR AMD MI100 GPU

Component	#Components	Group size	#Components per group	Matrix format with storage strategy A	B	C	D
Inverter	512	32	4	ELL*	ELL*	ELL*	ELL*
	2048	256	32	ELL*	ELL*	ELL*	ELL*
	8192	256	32	ELL*	ELL*	CSR*	ELL*
	32768	256	96	ELL*	ELL*	ELL*	ELL*
	131072	128	48	ELL*	ELL*	ELL*	ELL*
	524288	128	32	ELL*	ELL*	ELL*	CSR*
SyncMachine	512	64	7	CDia*	Dia*	ELL*	-
	2048	64	7	CDia*	ELL*	ELL*	-
	8192	256	28	ELL*	Dia*	ELL*	-
	32768	256	96	ELL*	Dia*	ELL*	-
	131072	128	28	ELL*	CSR*	ELL*	-
	524288	256	28	CSR*	CDia*	ELL*	-
Electrolyzer	512	64	3	ELL*	Dense-cat	CSR*	-
	2048	128	6	ELL*	ELL*	CSR*	-
	8192	256	12	ELL*	CSR*	CSR*	-
	32768	256	12	ELL*	CSR*	CSR*	-
	131072	256	12	ELL*	CSR*	CSR*	-
	524288	128	3	CSR*	ELL*	CSR*	-

*: pattern storage
-cat: concatenated storage
-BD: block diagonal storage