DG-RePlAce: A Dataflow-Driven GPU-Accelerated Analytical Global Placement Framework for Machine Learning Accelerators

Andrew B. Kahng, *Fellow, IEEE*, and Zhiang Wang, *Member, IEEE*

***Abstract*—Global placement (GP) is a fundamental step in VLSI physical design. The wide use of 2-D processing element (PE) arrays in machine learning accelerators poses new challenges of scalability and quality of results (QoR) for state-of-the-art academic global placers. In this work, we develop *DG-RePlAce*, a new and fast GPU-accelerated GP framework built on top of the OpenROAD infrastructure , which exploits the inherent dataflow and datapath structures of machine learning accelerators. Experimental results with a variety of machine learning accelerators using a commercial 12-nm enablement show that, compared with *RePlAce* (*DREAMPlace*), our approach achieves an average reduction in routed wirelength by 10% (7%) and total negative slack (TNS) by 31% (34%), with faster GP and on-par total runtimes relative to *DREAMPlace*. Empirical studies on the *TILOS MacroPlacement Benchmarks* further demonstrate that post-route improvements over *RePlAce* and *DREAMPlace* may reach beyond the motivating application to machine learning accelerators.**

***Index Terms*—Dataflow, GPU acceleration, physical design (EDA), VLSI placement.**

I. Introduction

GLOBAL placement is a fundamental step in VLSI physical design that determines the locations of standard cells and macros in a layout. The backend design closure flow requires a fast placement engine for rapid design prototyping, feeding back to synthesis, and guiding optimization. However, emerging machine learning accelerators have introduced new challenges for global placement (GP). On the one hand, machine learning accelerators with millions of standard cells and macros raise runtime concerns for the design closure process. On the other hand, machine learning accelerators featuring 2-D processing element (PE) arrays, such as systolic arrays [26], have gained prominence because of their efficiency in convolutional neural network computations [9]. The dataflow and datapath architectures of modern machine learning accelerators exhibit substantial differences compared to those of traditional datapath designs, requiring dedicated treatment during GP to achieve decent quality of results (QoR).

To address aspects of the aforementioned challenges, several global placers have been proposed over the past decades. To improve runtime, researchers have focused on parallelizing GP algorithms to leverage the computational substrates provided by multicore CPUs and GPUs. [8] introduces a multithreaded shared-memory implementation of RePlAce [3] using off-the-shelf multicore CPU hardware. Cong and Zou [7] and Lin and Wong [15] proposed GPU-accelerated analytical placers by parallelizing the computation of the logarithm-sum-exponential (LSE) wirelength function as well as the density function. Recently, *DREAMPlace* [16], [20] and *Xplace* [23] have implemented the approach of *RePlAce* on GPU by casting the placement problem as a neural network training problem; these works demonstrate the superiority of GPU-accelerated global placers. While *DREAMPlace* has already achieved significant runtime improvement relative to *RePlAce*, our aim is to push these boundaries even further by leveraging optimized data structures and a new parallel wirelength gradient computation algorithm.

Netlist and register clustering are widely used in placement to achieve better QoR. Lu et al. [17], [22] used clustering information to induce soft placement constraints for commercial placers. Chang et al. [4] proposed a register clustering approach to balance clock power reduction and timing degradation. Liu et al. [21] adopted c-spectral clustering to reduce the problem size and ensure that standard-cell clusters and macros are of comparable size. Kahng et al. [12] proposed a power, performance and area (PPA)-aware clustering approach that takes into account timing, power and logical hierarchy during netlist clustering, effectively accelerating GP runtime and improving post-route QoR. Other pioneering works adopt the clustering idea to exploit the dataflow and datapath structures during macro placement and GP. Kahng et al. [10] and Vidal-Obiols et al. [27] exploited RTL information and dataflow to guide macro placement. Lin et al. [19] integrated the dataflow information into the mixed analytical GP framework through virtual objects. Furthermore, Fang et al. [6] introduced the first GP framework that exploits the datapath regularity of 2-D PE arrays. In our present work, we propose a new, fast GPU-accelerated GP framework, exploiting both dataflow information and datapath regularity. Our approach ultimately guides GP toward better QoR. The main contributions of this article are as follows.

Manuscript received 23 March 2024; revised 19 June 2024; accepted 19 July 2024. Date of publication 1 August 2024; date of current version 22 January 2025. This article was recommended by Associate Editor E. Young. *(Corresponding author: Zhiang Wang.)*

Andrew B. Kahng is with the Electrical and Computer Engineering Department and the Computer Science and Engineering Department, University of California at San Diego, La Jolla, CA 92093 USA (e-mail: abk@ucsd.edu).

Zhiang Wang is with the Electrical and Computer Engineering Department, University of California at San Diego, La Jolla, CA 92093 USA (e-mail: zhw033@ucsd.edu).

Digital Object Identifier 10.1109/TCAD.2024.3436521

1) We propose *DG-RePlAce*, a new and fast global placer that leverages the intrinsic dataflow and datapath structures of machine learning accelerators, to achieve high-quality GP.
2) *DG-RePlAce* is built on top of the OpenROAD infrastructure with a permissive open-source license, enabling other researchers to readily adapt it for other enhancements.[1]
3) We propose efficient data structures and algorithms to further speed up the GP. Experimental results on a variety of machine learning accelerators show that, our approach is, respectively, on average 22.49*X* and 1.75*X* faster than *RePlAce* and *DREAMPlace* in terms of GP runtime. Overall turnaround time (TAT) is on par with that of *DREAMPlace*, despite (one-time) file IO runtime overheads that are due to OpenDB/OpenROAD integration.
4) Experimental results on a variety of machine learning accelerators also show that in comparison with *RePlAce* and *DREAMPlace*, our approach achieves an average reduction of routed wirelength by 10% and 7%, and of total negative slack (TNS) by 31% and 34%, respectively.
5) Experimental results on the two largest *TILOS MacroPlacement Benchmarks* [32] testcases show that compared with *RePlAce* and *DREAMPlace*, *DG-RePlAce* achieves much better-timing metrics (worst-negative slack (WNS) and TNS) measured post-route optimization. This suggests that the proposed dataflow-driven methodology is not limited to machine learning accelerators.

The remaining sections are organized as follows. Section II introduces the terminology and background. Section III discusses our approach. Section IV shows experimental results, and Section V concludes this article and outlines future research directions.

II. Preliminaries

In this section, we begin by discussing the fundamentals of the systolic array structure in Section II-A. Following this, we delve into the electrostatics-based placement formulation, which is incorporated into DG-RePlAce, in Section II-B. Finally, we examine previous works on dataflow-driven placement in Section II-C. Table I summarizes important terms and their meanings; for clarity, we give 1-D (x component) notation.

A. Systolic Array Structure

A systolic array [5] is a 2-D array of $M \times N$ PEs, which performs massively parallel convolution and matrix multiplication operations. A PE is often composed of a multiply-accumulate (MAC) unit and registers, and PEs that are aligned in the same row collectively form a *processing unit* (PU). Fig. 1 shows an example execution flow of a systolic array-based machine learning accelerator. Here, the input data horizontally propagate through the PEs, are multiplied by the weights in each PE, and then are accumulated vertically along the columns of the systolic array. This structure restricts data transfers (multiple bitwidth) to neighboring PEs, thus achieving better performance and efficiency.

TABLE I
Terminology and Notation

Notation	Description
v	instance (standard cell or macro)
p	instance pin or input-output pin
e	net $e = \{p\}$
V	Set of all instances $\{v\}$
E	Set of all nets (hyperedges) $\{e\}$
P	Set of all pins $\{p\}$
$WL_{grad_x}(p)$	Wirelength gradient on pin p
x_p	x coordinate of pin p
x_e^+	$max_{i \in e}\ x_i,\ \forall e \in E$
x_e^-	$min_{i \in e}\ x_i,\ \forall e \in E$
a_i^+	$exp(\frac{x_i - x_e^+}{\gamma}),\ \forall i \in e,\ e \in E$
a_i^-	$exp(-\frac{x_i - x_e^-}{\gamma}),\ \forall i \in e,\ e \in E$
b_e^+	$\sum_{i \in e} a_i^+,\ e \in E$
b_e^-	$\sum_{i \in e} a_i^-,\ e \in E$
c_e^+	$\sum_{i \in e} x_i a_i^+,\ e \in E$
c_e^-	$\sum_{i \in e} x_i a_i^-,\ e \in E$
X_v	Instance location, $\forall v \in V$
$F_{WL_x}(v)$	Wirelength force on instance v
PU	Processing unit
PE	Processing element

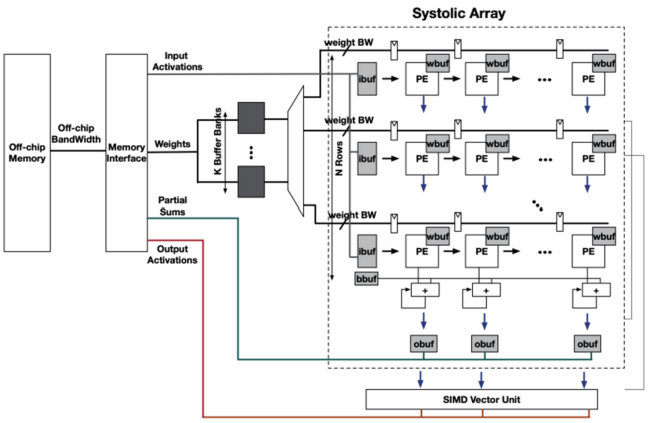

Fig. 1. Illustrative execution flow of a systolic array-based machine learning accelerator (figure reproduced from [5]).

B. Electrostatics-Based Placement

State-of-the-art academic global placers, such as *RePlAce* [3] and *DREAMPlace* [16], [20], usually adopt the electrostatics-based placement approach [14]. Let $(X_v, Y_v)^T$ denote the vectors of x-y coordinates of movable instances. The electrostatics-based placers formulate the GP problem as follows:

$$\min_{X_v, Y_v} \sum_{e \in E} \mathrm{WL}(e; X_v, Y_v) + \lambda \times D(X_v, Y_v) \tag{1}$$

where $\mathrm{WL}(\cdot;\cdot)$ is the wirelength cost function, $D(\cdot)$ is the instance density cost function, and λ is the weighting factor. In this work, we use the weighted-average (WA) wirelength as

[1]The source code is available in the *DG-RePlAce* GitHub repository [40].

the wirelength cost function, where the x-component of WA for net e is given by

$$\mathrm{WL}_e = \frac{\sum_{i\in e} x_i \cdot \exp\left(\frac{x_i}{\gamma}\right)}{\sum_{i\in e} \exp\left(\frac{x_i}{\gamma}\right)} - \frac{\sum_{i\in e} x_i \cdot \exp\left(-\frac{x_i}{\gamma}\right)}{\sum_{i\in e} \exp\left(-\frac{x_i}{\gamma}\right)} \quad (2)$$

where γ is a parameter that controls the smoothness and accuracy of the approximation to the half-perimeter wirelength (HPWL). With the notations in Table I, the gradient of WA wirelength to a pin location x_i is given as follows:

$$\frac{\partial \mathrm{WL}_e}{\partial x_i} = \frac{\left(1+\frac{x_i}{\gamma}\right)b_e^+ - \frac{1}{\gamma}c_e^+}{\left(b_e^+\right)^2} \cdot a_i^+ - \frac{\left(1-\frac{x_i}{\gamma}\right)b_e^- + \frac{1}{\gamma}c_e^-}{\left(b_e^-\right)^2} \cdot a_i^-. \quad (3)$$

C. Dataflow-Driven Placement

To obtain high-quality macro placement, human designers usually rely on their understanding of the dataflow of a design to determine the relative locations of macros. However, this manual process is very time-consuming, often taking several days to weeks to complete. To automate this process, Vidal-Obiols et al. [27] and *Hier-RTLMP* [11] introduced dataflow-driven multilevel macro placement approaches. However, both approaches apply the Simulated Annealing [13] algorithm to determine the locations of macros, resulting in poor runtime scalability [1]. Lin et al. [18] presented an analytical-based placement algorithm to handle dataflow constraints in mixed-size circuits. Their approach initially assigns larger weights to nets connecting to datapath-oriented objects, and then gradually shrinks the weights according to the status of placement utilization.[2] However, their method requires dataflow constraints from designers and cannot handle the unique datapath regularity in machine learning accelerators (see Section III-C). In this work, we incorporate the physical hierarchy extraction approach in *Hier-RTLMP* [11] into the GP framework to capture the dataflow information during GP. Besides, we pay special attention to the datapath regularity in machine learning accelerators during placement.

III. Our Approach

The architecture of our *DG-RePlAce* framework is shown in Fig. 2. The input is a synthesized hierarchical gate-level netlist and a floorplan.def file that contains the block outline and fixed IO pin or pad locations. The output is a.def file with placed macros and standard cells. *DG-RePlAce* is built on top of the open-source OpenROAD infrastructure [30], and consists of four major steps.

1) *Physical Hierarchy Extraction (Section III-A):* During this step, we convert the structural netlist representation of the RTL design into a clustered netlist. The instances within the same cluster are expected to remain close to each other during GP.
2) *Dataflow-Driven Initial Global Distribution (Section III-B):* During this step, we insert the dataflow information into the clustered netlist, and determine the location for each cluster through our new *GPU-accelerated parallel analytical placement* method. Then, every instance within a cluster is positioned at the cluster's center. Furthermore, we incorporate pseudo net constraints into the original netlist, ensuring that instances belonging to the same cluster are placed in close proximity to each other.
3) *Datapath Constraints Construction (Section III-C):* This step extracts datapath information from the original netlist. Following this extraction, we transform the datapath information into pseudo net constraints.
4) *Parallel Analytical Placement (Section III-D):* At this step, we execute the GPU-accelerated mixed-size GP on the original gate-level netlist, integrating the pseudo net constraints derived from the Dataflow-Driven Initial GP and the Datapath Constraints Construction steps. Here, we leverage parallel processing capabilities of GPU to accelerate the Nesterov's method in *RePlAce*. In this work, we use the *GPU-accelerated parallel analytical placement* for both cluster-level and gate-level netlist placement.

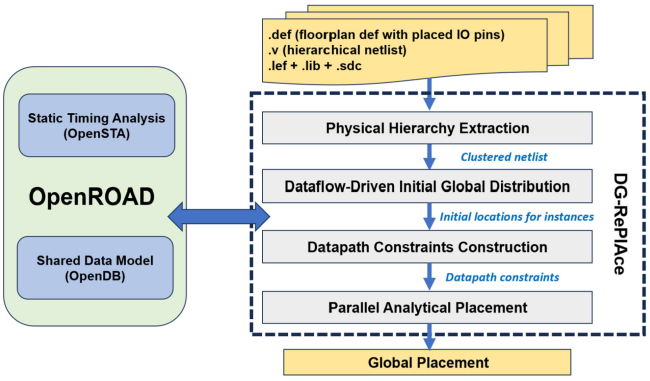

Fig. 2. Overview of the proposed *DG-RePlAce* flow.

In the remainder of this section, we will explain each step in detail. The runtime analysis for each step is presented in Fig. 8 (see Section IV-B).

A. Physical Hierarchy Extraction

During this step, we transform the original logical hierarchy into a physical hierarchy. Much like the logical hierarchy, which is composed of logical modules, the physical hierarchy consists of physical clusters. In contrast to logical modules, a physical cluster consists of instances that are expected to remain close to each other during GP. Specifically, we employ the *Multilevel Autoclustering* component of the open-source *Hier-RTLMP* [11] to perform physical hierarchy extraction.[3] Upon establishing the physical hierarchy, we convert the original gate-level netlist into a clustered netlist.

B. Dataflow-Driven Initial Global Distribution

In this section, we first describe how to insert dataflow information into the clustered netlist. Then, we explain how

[2]Lin et al. [18] has reported an excellent dataflow-driven analytical mixed-size placer. Unfortunately, no testcases or executables can be released by their group.

[3]The detailed algorithm is presented as [11, Algorithm 2]. The source code is available in [40].

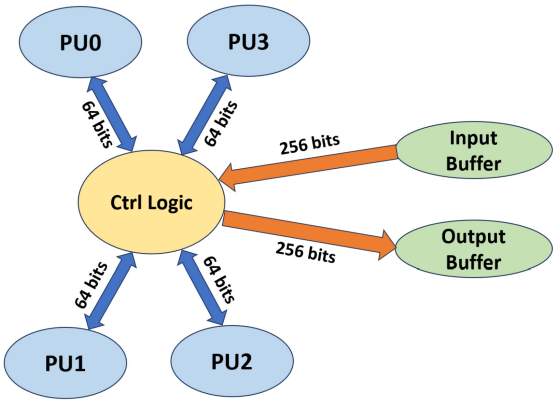

Fig. 3. Dataflow visualization of Tabla01 design [5].

we use the *GPU-accelerated parallel analytical placement* framework to distribute the clustered netlist evenly, and how we solve the divergence issue by applying a bloat factor to each cluster. Finally, we discuss how we use the placed clustered netlist to guide the GP process.

After physical hierarchy extraction, we have a clustered netlist in which the nodes are clusters and the nets are bundled connections. We then insert the dataflow information into the clustered netlist through virtual connections. We use the term *dataflow* to refer to the way in which data moves between different functional units of a netlist. The dataflow can be visualized as the high-level conceptual movements of data and how they are processed step by step. Fig. 3 shows the dataflow visualization of the Tabla01 design (see Section IV for details of this and other testcases). When backend engineers perform the place-and-route (P&R) flow for a netlist, understanding the dataflow is critical for optimizing PPA, as the dataflow determines how the netlist is pipelined and how the parallel processing is implemented. We adopt the same idea as [10], [11], and [27] and transform the dataflow information into *virtual connections* between clusters. The virtual connections (*virtual_conn*(A, B)) between clusters A and B are defined as

$$\text{virtual\_conn}(A, B) = \frac{\text{info\_flow}(A, B)}{2^{\text{num\_hops}}}. \tag{4}$$

Here, *info_flow* corresponds to connection bitwidth and *num_hops* is the length of the shortest path of registers between clusters. As pointed out in [27], the virtual connections between clusters capture the pipelined signal flow and the implementation of parallel processing. This helps to ensure that the cluster placement aligns with the design's dataflow structure. When calculating the virtual connections between clusters, we follow the same convention as [10] and [11]. If the register distance (*num_hops*) between clusters is greater than 4, then no virtual connection is added. In the example shown in Fig. 3, if the register distance between *PU0* and *Output Buffer* is 2, then the calculated virtual connections are 16, given that the connection bitwidth is 64 bits.

Upon incorporating the dataflow information into the clustered netlist, we call the *GPU-accelerated parallel analytical placement* framework, for which details are given in Section III-D, to evenly distribute the clustered netlist.

Fig. 4. Illustration of the bloat-shrink approach for reducing *cluster_overflow*. Left: Density overflow caused by overlap between clusters A and B; and Right: Removal of overlap by shrinking clusters A and B.

However, directly working on the clustered netlist could lead to divergence issues, particularly if the layout has a large amount of whitespace. To solve the divergence issues, we introduce a *bloat-shrink* methodology guided by the final density overflow of the cluster placement (refer to [14, eq. (37)] for the definition). In the bloat-shrink methodology, we initially bloat each cluster before cluster placement to achieve total cluster area that matches the area of the placement region. If the cluster placement diverges, we then shrink each cluster to solve the divergence issue. More specifically, this methodology includes two steps:

1) *Bloat:* We first bloat each cluster by applying a bloat factor (*bloat_factor*), defined as

$$\text{bloat\_factor} = \frac{\text{Area of placement region}}{\text{Total area of clusters}}. \tag{5}$$

2) *Shrink:* If the cluster placement ends with a density overflow *cluster_overflow* that exceeds the target density overflow *target_overflow*, we then shrink each cluster using a shrink factor (*shrink_factor*). *shrink_factor* is determined by dividing the target density overflow (*target_overflow*) by the actual density overflow (*cluster_overflow*)

$$\text{shrink\_factor} = \frac{\text{target\_overflow}}{\text{cluster\_overflow}}. \tag{6}$$

Here, *target_overflow* (*target_overflow* $= 0.2$ by default for cluster placement) is the convergence criterion for the Nesterov's approach.

Global placers like *RePlAce* usually fill whitespace through filler insertion. The fillers are equally sized rectangles, movable and disconnected (with zero pins). The additional density force created by the insertion of fillers helps squeeze the standard cells closer to their connected neighbors while still satisfying the density constraints [14]. In contrast to filler insertion used in *RePlAce* for filling whitespace, the bloat-shrink methodology ensures a feasible cluster placement solution that satisfies the density overflow constraint. The shrinking step is necessary because all clusters are square-shaped, which may lead to infeasible solutions. Fig. 4 demonstrates how the bloat-shrink approach reduces *cluster_overflow*.

After completing the placement of clusters, we place the instances within each cluster at the cluster's center to obtain a good initial placement. Furthermore, for each cluster, we add one pseudo net that connects all of the instances within

the cluster. This ensures that instances belonging to the same cluster are placed in close proximity to each other. However, we notice that these high-fanout pseudo-nets could cause convergence problems. To address this issue, we transform the pseudo nets into multiple two-pin nets by the star model (i.e., by adding a pseudo vertex as the star's center, per cluster). To ensure that the global placer follows the pseudo net constraints imposed by the clustering constraints, we initially assign a high-penalty factor $penalty\_factor_p$ to the pseudo nets, starting at the value of $penalty\_factor_{p0}$. With each successive iteration, the penalty factor is progressively decreased to allow for a more even distribution of instances across the placement region. The adjustment of the penalty factor $penalty\_factor_p$ is determined by the following equation:

$$\text{penalty\_factor}_{p0} = \exp(\text{iter}_0) \tag{7}$$

$$\text{penalty\_factor}_p = \frac{\text{penalty\_factor}_{p0}}{\exp(\text{iter})} \tag{8}$$

where *iter* is the current iteration number, and $iter_0$ ($iter_0 = 4$ by default) is used to determine the initial value.[4] To determine the default value of $iter_0$, we study $iter_0$ values ranging from 0 to 9, utilizing Tabla01 and Tabla02 (see Table II) as testcases. The score for our evaluation is the routed wirelength, which we normalize against the baseline results obtained from *RePlAce*. Based on this experiment, we use $iter_0 = 4$ as the default.

C. Datapath Constraints Construction

After capturing the dataflow between clusters, we examine the detailed data movement within each cluster, i.e., datapath information. In contrast to dataflow, the *datapath* refers to the actual hardware components and interconnections that implement the dataflow, representing the paths that data traverse in a digital design. More specifically, the datapath is the circuit performing bit-wise data operations in parallel on multiple bits [2]. Each operation corresponds to a dedicated functional block, such as adder, register, buffer, multiplexer, multiplier, etc. Fang et al. [6] further pointed out that there is a significant difference between the datapath within a systolic array and that of traditional datapath designs, as shown in Fig. 5.

Fig. 5(a) shows the datapath in traditional datapath designs, characterized by a continuous bit-sliced structure for operations across different bits [2]. In such scenarios, a pseudo net can be applied to each alignment group (i.e., A[1:12], B[1:12], and C[1:12]), ensuring that the instances in each alignment group are placed in proximity. In contrast, as depicted in Fig. 5(b), the datapath in a systolic array is not continuous, with operations for different bits across multiple PEs. This may lead to overlaps of PEs when a pseudo net is directly applied to each alignment group, as shown in Fig. 5(c). To address this issue, we propose to assign pseudo nets to local alignment groups within each cluster. For example, in Fig. 5(d), pseudo nets are independently applied to A[1:3] in PE1, A[4:6] in PE2, A[7:9] in PE3 and A[10:12] in PE4. Here, we also transform the pseudo nets into multiple two-pin nets according to the star model. To maintain the integrity of local connectivity, we set the initial penalty factor $penalty\_factor_{p0}$ to 1 for the pseudo nets induced by datapath constraints.

[4]In our implementation, we set $penalty\_factor_p$ to 0 if $\exp(iter)$ is NaN [45].

D. Parallel Analytical Placement

Our GPU-accelerated mixed-size parallel analytical placement framework uses the same Nesterov's method as *RePlAce*, and is developed on top of the OpenROAD infrastructure. To minimize memory overhead, we integrate a data structure inspired by Gessler et al. [8], which optimizes data locality for the frequently accessed components during GP. As pointed out by [15], the fast computation of wirelength gradient and bin density is crucial for the efficiency of the global placer. We adopt the parallel bin density computation algorithm from Gessler et al. [8] [8, Algorithm 2]. For the fast computation of wirelength gradient, we introduce a novel parallel algorithm, presented in Algorithm 1. Our algorithm distinguishes itself from Algorithm 1 in *DREAMPlace* [16] primarily in lines 1–6 and 12–18, where we leverage net-level parallelization rather than pin-level parallelization to eliminate the need for atomic additions. Furthermore, it differs from Algorithm 2 in *DREAMPlace* [16] primarily in lines 7–11 and 19–22, where we implement pin-level computation parallelization with multiple threads rather than the sequential computation within a single thread. This approach is more efficient for managing high-fanout nets while maintaining comparable efficiency in handling low-fanout nets (see Section IV-B for details). Empirical results demonstrate that our algorithm is approximately 3.25X faster than the one implemented in *DREAMPlace* [16, Algorithm 2].

IV. Experimental Results

DG-RePlAce is implemented with approximately 14K lines of C++ (and CUDA) with a Tcl command line interface on top of the OpenROAD infrastructure [30]. We run all experiments on a Linux server with an Intel Xeon E5-2690 CPU (48 threads) with 256-GB RAM and an NVIDIA TITAN V GPU.

To show the effectiveness of our global placer, the following three scenarios are evaluated and compared.

1) *RePlAce:* GP is done by *RePlAce*, which is the default global placer in the OpenROAD project [30].
2) *DREAMPlace:* GP is performed by the latest version of *DREAMPlace* [31], which is the state-of-the-art GPU-accelerated global placer. The default hyperparameter settings that we use for *DREAMPlace* are from [33].
3) *DG-RePlAce:* Results are obtained using our global placer.

Our experiments use the following flow.

1) We first synthesize the design using a state-of-the-art commercial synthesis tool, preserving the logical hierarchy.
2) Next, we determine the core size of the testcase and place all the IO pins using a manually developed script (see [43]).
3) Then, the GP is performed using different methods (*RePlAce*, *DREAMPlace* and *DG-RePlAce*).

Fig. 5. Datapath constraints construction on the 2-D PE array. (a) Pseudo nets for bit stacks of a traditional datapath. (b) Example of a 2-D PE array. (c) Applying pseudo nets on the 2-D PE array. (d) Datapath constraints on the 2-D PE array.

4) Finally, we use a state-of-the-art commercial P&R tool, Cadence Innovus 21.1, to finish the legalization of macros, detailed placement of standard cells and routing. We follow the SP&R scripts in the public *MacroPlacement repository* [32]. All metrics are collected after post-route optimization. All studies use a commercial foundry 12-nm technology (13 metal layers) with cell library and memory generators from a leading IP provider.

In this section, we first present the results for two types of machine learning accelerators: non-DNN machine learning accelerators (Tabla designs) and DNN machine learning accelerators (GeneSys designs), detailed in Section IV-A. Then, we discuss the runtime comparison between *DG-RePlAce* and *DREAMPlace* in Section IV-B. Following this, Section IV-C compares *DG-RePlAce* with the dataflow-driven macro placer *Hier-RTLMP*, which uses the same method to perform physical hierarchy extraction. Next, we study the respective effects of dataflow and datapath constraints by conducting an ablation study of *DG-RePlAce*, in Section IV-D. Finally, we apply *DG-RePlAce* to large nonmachine learning testcases in Section IV-E, demonstrating the versatility and potential benefit of the proposed dataflow-driven approach beyond our motivating application context of large-scale machine learning accelerators.

A. Results on Machine Learning Accelerators

We have validated our global placer using two types of machine learning accelerators (Tabla and GeneSys) from the VeriGOOD-ML platform [5]. The Tabla accelerators are designed for training and inference for non-DNN machine learning algorithms, and the GeneSys accelerators are for DNN machine learning algorithms. Both Tabla and Genesys adopt the systolic array structure, thus each design has an $m \times n$ PE array. The major characteristics of the testcases are summarized in Table II.

Table III shows the experimental results after completion of post-route optimization. Rows represent testcases and GP flows, and columns give information on total routed wirelength, power, WNS, TNS, runtime of GP and TAT.[5] The metrics are normalized to protect foundry IP: 1) wirelength and power are normalized to the *RePlAce* results and 2) timing metrics (WNS and TNS) are normalized to the clock period which we leave unspecified.

We can observe the following conclusions.

1) Our approach outperforms both *RePlAce* and *DREAMPlace* in terms of routed wirelength, achieving average reductions of 10% and 7%, respectively.
2) Our approach outperforms both *RePlAce* and *DREAMPlace* in terms of TNS, achieving average reductions of 31% and 34%, respectively.
3) Our approach achieves similar speedup as *DREAMPlace* in terms of total TAT, but our approach is about 1.75*X* faster than *DREAMPlace* in terms of GP runtime. The detailed runtime analysis is presented in Section IV-B.
4) For the Tabla03 design, our approach significantly outperforms both *RePlAce* and *DREAMPlace* in all the

[5]The runtime of GP refers to the runtime required to distribute the original netlist across the placement region using the Nesterov's method. For *DREAMPlace*, we extract the relevant information from the following log file entry: "[INFO] DREAMPlace - nonlinear placement takes xx seconds."

Algorithm 1: Parallel Wirelength Gradient Computation

Input: Instances V, Nets E, Pins P and Instance locations X_v
Output: Wirelength force for each instance $F_{WL_x}(v)$

for *each thread* $0 \le t < |E|$ **do**
  Define e as the net corresponding to thread t;
  $x_e^+ \leftarrow max_{p \in e} x_p$; ▶ x_e^+ is in the global memory
  $x_e^- \leftarrow min_{p \in e} x_p$; ▶ x_e^- is in the global memory
  $b_e^\pm \leftarrow 0$; $c_e^\pm \leftarrow 0$; ▶ $b_e^\pm$, $c_e^\pm$ are in the global memory
end
for *each thread* $0 \le t < |P|$ **do**
  Define p as the pin corresponding to thread t;
  Define e as the net that pin p belongs to;
  $a_p^\pm \leftarrow e^{\pm(x_p - x_e^\pm/\gamma)}$; ▶ $a_p^\pm$ is in the global memory
end
for *each thread* $0 \le t < |E|$ **do**
  Define e as the net corresponding to thread t;
  for *pin* $p \in e$ **do**
    $b_e^\pm \leftarrow b_e^\pm + a_p^\pm$;
    $c_e^\pm \leftarrow c_e^\pm + x_p a_p^\pm$;
  end
end
for *each thread* $0 \le t < |P|$ **do**
  Define p as the pin corresponding to thread t;
  Compute the wirelength gradient of pin $WL_{grad_x}(p)$ using Equation (3); ▶ $WL_{grad_x}(p)$ is in the global memory
end
for *each thread* $0 \le t < |V|$ **do**
  Define v as the instance corresponding to thread t;
  $F_{WL_x}(v) \leftarrow 0.0$ ▶ $F_{WL_x}(v)$ is in the global memory
  for *pin* p *of* v **do**
    $F_{WL_x}(v) -= WL_{grad_x}(p)$;
  end
end
return $F_{WL_x}(v)$

TABLE II
BENCHMARKS. "MACRO UTIL" STANDS FOR MACRO UTILIZATION, WHICH IS DEFINED AS THE TOTAL AREA OF MACROS DIVIDED BY THE CORE AREA. "UTIL" STANDS FOR UTILIZATION, WHICH IS DEFINED AS THE TOTAL AREA OF STANDARD CELLS AND MACROS WITH A 2μM HALO WIDTH DIVIDED BY THE CORE AREA

Designs	PE Array	# Macros	# Std Cells	# Nets	Macro Util	Util
Tabla01	4 × 8	368	232K	252K	0.60	0.75
Tabla02	4 × 16	1232	441K	486K	0.59	0.79
Tabla03	8 × 8	760	372K	408K	0.58	0.78
Tabla04	8 × 16	2488	741K	830K	0.54	0.76
GeneSys01	16 × 16	368	986K	1056K	0.46	0.72
GeneSys02	16 × 16	368	1055K	1135K	0.52	0.71

metrics (wirelength, power and timing). We attribute this to *DG-RePlAce*'s ability to identify the dataflow and datapath of the design, which enables it to generate the placement in accordance with dataflow and datapath constraints. Fig. 6(c) shows the post-route layout of the Tabla03 design for the GP generated by *DG-RePlAce*. We can see that it perfectly matches the dataflow pattern illustrated in Fig. 3. By contrast, in the layout from *DREAMPlace* [Fig. 6(b)], we can see that PU2 (in orange) gets mixed up with other PUs, leading to a significant degradation in wirelength, power and performance. *DREAMPlace*'s lack of awareness of the design's dataflow and datapath information means that it has more difficulty in generating placements that align with the dataflow and datapath structure of the design.

TABLE III
EXPERIMENTAL RESULTS. WE HIGHLIGHT BEST VALUES OF METRICS IN BLUE BOLD FONT. DATA POINTS FOR WL, POWER, WNS, AND TNS ARE NORMALIZED

Design	Global Placer	WL	Power	WNS	TNS	GP (s)	TAT (s)
Tabla01	*RePlAce*	1.00	1.00	-0.180	-81.349	150	204
	DREAMPlace	0.98	1.01	**-0.151**	**-55.844**	14	**27**
	Hier-RTLMP	1.10	1.00	-0.156	-74.929	-	2351
	DG-RePlAce	**0.93**	**0.98**	-0.180	-62.622	**7**	31
Tabla02	*RePlAce*	1.00	1.00	-0.197	-22.695	374	476
	DREAMPlace	0.95	**0.98**	-0.188	-24.188	24	**47**
	Hier-RTLMP	1.12	1.02	**0.160**	**-17.807**	-	3613
	DG-RePlAce	**0.91**	**0.98**	-0.187	-19.642	**13**	50
Tabla03	*RePlAce*	1.00	1.00	-0.092	-43.136	279	364
	DREAMPlace	1.21	1.05	-0.154	-84.152	21	**41**
	Hier-RTLMP	0.98	0.99	-0.136	-88.578	-	3872
	DG-RePlAce	**0.88**	**0.97**	**-0.084**	**-14.910**	**16**	47
Tabla04	*RePlAce*	1.00	1.02	**-0.174**	**-48.756**	689	883
	DREAMPlace	**0.83**	**0.96**	-0.177	-62.990	47	**81**
	Hier-RTLMP	1.10	1.06	-0.281	-222.119	-	8418
	DG-RePlAce	0.85	0.97	-0.219	-54.755	**20**	82
GeneSys01	*RePlAce*	1.00	1.00	-0.191	-94.176	630	850
	DREAMPlace	**0.89**	**0.98**	-0.213	-101.598	61	101
	Hier-RTLMP	0.97	1.00	**-0.110**	**-23.151**	-	3134
	DG-RePlAce	**0.89**	**0.98**	-0.162	-45.939	**40**	**100**
GeneSys02	*RePlAce*	1.00	1.00	-0.132	-14.937	752	972
	DREAMPlace	**0.89**	**0.97**	-0.108	-14.979	64	112
	Hier-RTLMP	N/A					
	DG-RePlAce	0.94	0.99	**-0.062**	**-7.78**	**44**	**111**

5) For the GeneSys02 design, our approach delivers significantly better timing compared to both *RePlAce* and *DREAMPlace*. This is because *DG-RePlAce* follows the dataflow pattern inherent in the GeneSys02 design, as shown in Fig. 6(f). We also observe that the *Input Buffer* module (highlighted in purple) becomes mixed up with other modules when placed by *RePlAce* [Fig. 6(d)] and *DREAMPlace* [Fig. 6(e)]. While *DREAMPlace*'s solution generates significantly better wirelength, its power improvement is very limited. We attribute this to the GeneSys02 design's extreme macro dominance, where the leakage power and internal power (i.e., the power consumed by the CMOS circuit during the brief period when both pMOS and nMOS transistors are simultaneously switching as the logic changes its state [29]) constitute 67% of total power consumption. We leave how to achieve a better power and timing tradeoff as a direction for future work.

B. Runtime Comparison Against DREAMPlace

We now compare the runtime of *DG-RePlAce* against that of the leading GPU-accelerated global placer, *DREAMPlace*. As shown in Table III, the GP runtime of *DG-RePlAce* is less than that of *DREAMPlace*, while its overall TAT is similar. We will first discuss the GP runtime, and then examine the overall TAT.

The GP runtime efficiency of *DG-RePlAce* can be attributed to the following two factors.

1) *DG-RePlAce* achieves convergence with fewer iterations, due to an improved initial placement generated by our *Dataflow-Driven Initial GP*. The iterations

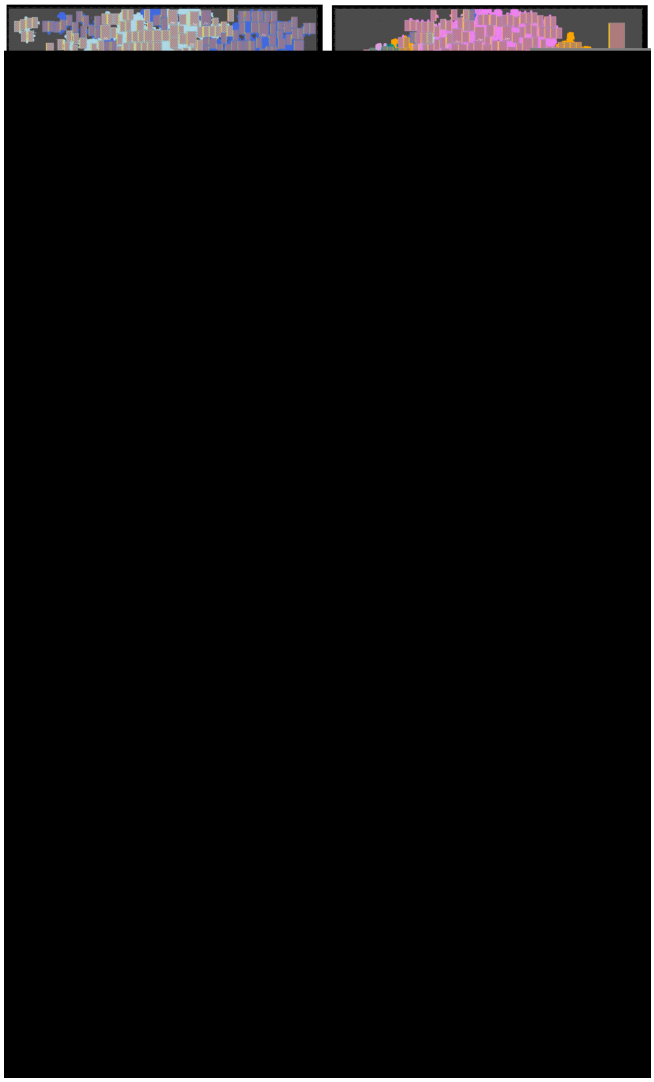

Fig. 6. Post-route layouts of Tabla3 and GeneSys02 designs with different flows. Design (Flow): (a) Tabla03 (*RePlAce*); (b) Tabla03 (*DREAMPlace*); (c) Tabla03 (*DG-RePlAce*); (d) GeneSys02 (*RePlAce*); (e) GeneSys02 (*DREAMPlace*); and (f) GeneSys02 (*DG-RePlAce*). For the same design, each module maintains consistent coloring across different layouts.

required for convergence of *RePlAce*, *DREAMPlace* and *DG-RePlAce* are presented in Table IV.[6] On average, *DG-RePlAce* achieves convergence in 24% fewer iterations compared to *DREAMPlace*.

2) Our parallel wirelength gradient algorithm (Algorithm 1) outperforms the one used by *DREAMPlace*, denoted as *DREAMPlace-Alg2*. For a fair comparison, we implement the wirelength gradient algorithm used by DREAMPlace [16, Algorithm 2]. The result is shown in Fig. 7, which suggests that our algorithm is on average 3.25*X* faster. To further confirm that the increased runtime overhead of *DREAMPlace-Alg2* is due to high-fanout nets, we remove all nets connecting over 100 instances.[7] After removing all of these high-fanout nets, *DREAMPlace-Alg2* achieves the same runtime as our Algorithm 1. This suggests that our parallel wirelength gradient algorithm is effective on high-fanout nets.

[6]The *stop_overflow* hyperparameter is 0.1 for *RePlAce* [34], *DREAMPlace* [33] and *DG-RePlAce*.

[7]*ignore_net_degree* is 100 by default in *DREAMPlace* [33].

TABLE IV
ITERATIONS REQUIRED FOR CONVERGENCE OF *RePlAce*, *DREAMPlace*, AND *DG-RePlAce*. WE HIGHLIGHT BEST VALUES IN BLUE BOLD FONT

Design	*RePlAce*	*DREAMPlace*	*DG-RePlAce*
Tabla01	410	450	**380**
Tabla02	460	546	**390**
Tabla03	460	507	**380**
Tabla04	520	687	**490**
GeneSys01	520	593	**470**
GeneSys02	510	598	**450**

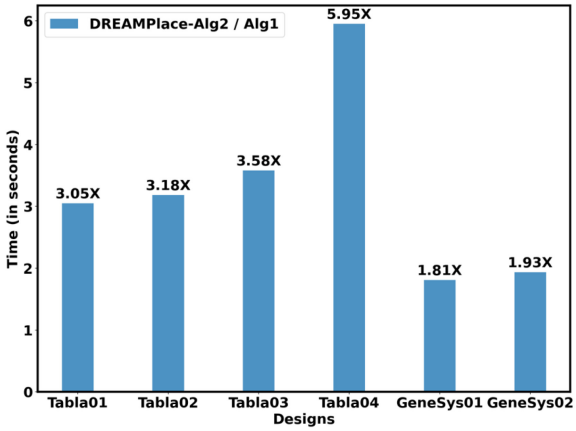

Fig. 7. Runtime comparison for different implementations of wirelength gradient computation.

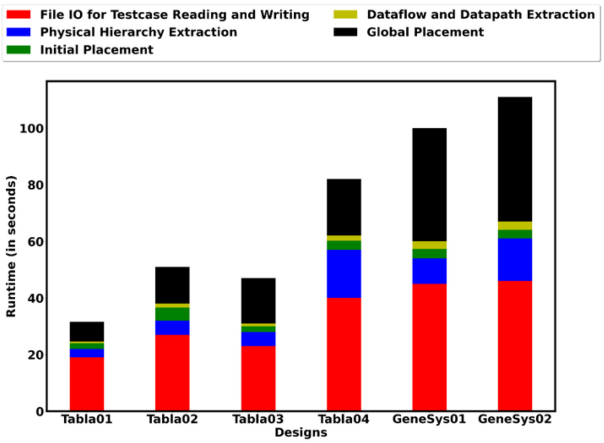

Fig. 8. Runtime breakdown of *DG-RePlAce*.

The longer overall TAT for *DG-RePlAce* results from the *file IO for testcase reading and writing* and *physical hierarchy extraction*. Fig. 8 shows the detailed runtime breakdown of *DG-RePlAce*. We see that *file IO for testcase reading and writing* accounts for 50% of the overall TAT. This is due to complexity of the industry-strength database (OpenDB [38] in OpenROAD [30]) that we use, which brings increased loading times for designs. On the other hand, considering that the design is typically loaded just once, substituting *DG-RePlAce* for *DREAMPlace* in scenarios where GP is executed many times (e.g., 1000 placement samples per design are obtained

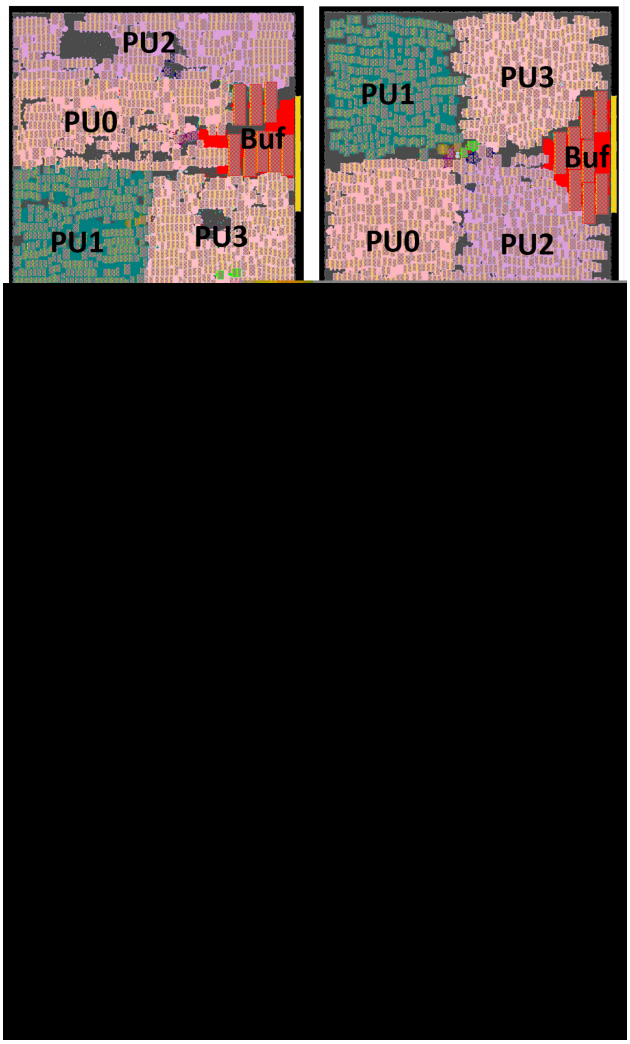

Fig. 9. Post-route layouts of Tabla02, Tabla04, and GeneSys01 designs with different flows: (a) Tabla02 (*Hier-RTLMP*); (b) Tabla02 (*DG-RePlAce*); (c) Tabla04 (*Hier-RTLMP*); (d) Tabla04 (*DG-RePlAce*); (e) GeneSys01 (*Hier-RTLMP*); and (f) GeneSys01 (*DG-RePlAce*). For the same design, each module maintains consistent coloring across different layouts. In this figure, "Buf" stands for input and output buffer.

by *AutoDMP* [1]) will significantly improve runtimes for such scenarios.

C. Comparison With Hier-RTLMP

In this section, we compare *DG-RePlAce* with the dataflow-driven multilevel macro placer *Hier-RTLMP* [11]. *Hier-RTLMP* uses the same physical hierarchy extraction approach and also considers dataflow information when determining locations of macros.[8] The results are presented in Table III, and Fig. 9 shows the post-route layouts. According to Table III, *DG-RePlAce* achieves 15% wirelength reduction compared to *Hier-RTLMP*. *Hier-RTLMP* has worse wirelength because it models each cluster as a rectangular shape, as shown in Fig. 9, potentially leading to unnecessary signal net detours.

Moreover, *Hier-RTLMP* fails to generate macro placement for the GeneSys02 design. *Hier-RTLMP* uses the Sequence Pair [24] representation and Simulated Annealing [13] algorithm to determine shapes and locations for clusters level by level. Therefore, it may not be able to obtain a feasible solution when it tries to place macros within a cluster whose location and shape have been determined in the previous step. Additionally, as has been pointed in [1], the use of Simulated Annealing algorithm in *Hier-RTLMP* makes it suffer from poor runtime scalability. Fig. 10 shows how the speedup achieved by *DG-RePlAce* over *Hier-RTLMP* changes with the number of PUs (#PU) and the number of PEs per PU (#PE per PU) for Tabla designs. We see that when the total number of PEs increases from 32 (Tabla01) to 128 (Tabla04), the speedup provided by *DG-RePlAce* over *Hier-RTLMP* increases from 76*X* to 103*X*. Such speedups are enabling for architects or front-end designers who seek to identify the optimal #PU and #PE per PU during the initial stages of machine learning accelerator development.

We also observe that for the GeneSys01 design, *Hier-RTLMP* generates the best-timing metrics in terms of both WNS and TNS. We attribute this to the relatively low-macro utilization of GeneSys01 (see Table II). In such contexts, *Hier-RTLMP* is able to generate reasonable macro tilings that are aligned with the dataflow structure, as shown in Fig. 9(e).

[8]We use the latest version of *Hier-RTLMP* from the OpenROAD repository [39].

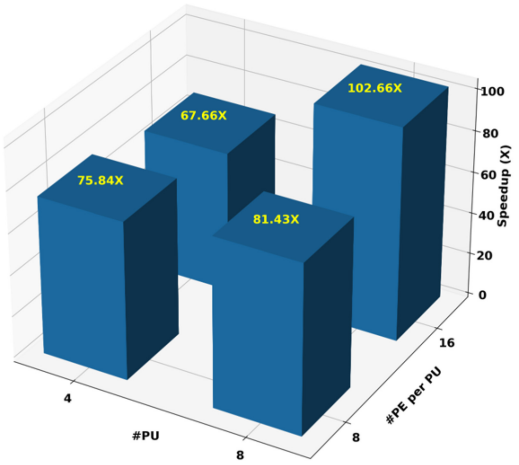

Fig. 10. Speedup of *DG-RePlAce* over *Hier-RTLMP* for Tabla designs.

TABLE V
EFFECT OF DATAFLOW AND DATAPATH CONSTRAINTS (AVERAGES OVER ALL TESTCASES). WE HIGHLIGHT BEST VALUES OF METRICS IN BLUE BOLD FONT. DATA POINTS ARE NORMALIZED

Metrics	*RePlAce*	*DG-RePlAce*$_{nf}$	*DG-RePlAce*$_{np}$	*DG-RePlAce*
WL$_{avg}$	1.00	0.92	0.91	**0.90**
TNS$_{avg}$	1.00	0.61	0.80	**0.61**

D. Ablation Study

To demonstrate the effect of dataflow and datapath constraints, we run an ablation study [37] by removing dataflow or datapath constraints to understand their respective contributions to the overall performance of *DG-RePlAce*. We conduct two separate experiments using variants of *DG-RePlAce*. The first variant, referred to as *DG-RePlAce*$_{nf}$, is executed without the dataflow constraint. The second variant, designated as *DG-RePlAce*$_{np}$, is executed without the datapath constraint. These

TABLE VI
EXPERIMENTAL RESULTS ON *TILOS MacroPlacement* BENCHMARKS. WE HIGHLIGHT BEST VALUES OF METRICS IN BLUE BOLD FONT. DATA POINTS FOR WL, POWER, WNS, AND TNS ARE NORMALIZED. *DREAMPlace** REPRESENTS RUNNING *DREAMPlace* WITH UPDATED HYPERPARAMETERS: *ignore_net_threshold* = 1E9 AND *iterations* = 5000

Design	Global Placer	WL	Power	WNS	TNS	GP (s)	TAT (s)
BlackParrot	*RePlAce*	1.00	1.00	-0.123	-108.15	387	653
	DREAMPlace	0.92	0.98	-0.023	-2.623	61	**88**
	DG-RePlAce	**0.90**	**0.97**	**-0.014**	**-0.078**	**32**	200
MemPool Group	*RePlAce*	1.00	1.00	-0.073	-99.989	1896	2712
	DREAMPlace	**0.92**	**0.97**	-0.086	-134.421	**72**	**167**
	*DREAMPlace**	**0.92**	**0.97**	-0.069	-108.193	178	284
	DG-RePlAce	0.95	0.98	**-0.067**	**-38.71**	122	591

modifications allow us to assess the individual contributions of each constraint. The experimental results are presented in Table V. In this table, WL_{avg} and TNS_{avg}, respectively, represent the average normalized routed wirelength and average TNS over all the testcases in Table II, compared with those from *RePlAce*. We observe that both *DG-RePlAce*$_{nf}$ and *DG-RePlAce*$_{np}$ generate better results than *RePlAce* in terms of TNS, but *DG-RePlAce* always generates the best results. This suggests that both dataflow and datapath constraints are important components of *DG-RePlAce*.

E. Results on TILOS MacroPlacement Benchmarks

The dataflow-driven approach proposed in this work is inspired by the unique demands of dataflow and datapath structures in modern, highly scaled machine learning accelerators. However, benefits of our proposed method may reach beyond machine learning accelerators. To demonstrate the generality and effectiveness of our *DG-RePlAce*, we conduct evaluations on the two largest benchmarks – BlackParrot (Quad-Core) [35] (827K instances, 196 macros) and MemPool Group [36] (2529K instances, 326 macros) – from the *TILOS MacroPlacement Benchmarks* [32]. The experimental results are summarized in Table VI, and the post-route layouts are presented in Fig. 11.

For the BlackParrot design, *DG-RePlAce* dominates *RePlAce* and *DREAMPlace* across all metrics, including wirelength, power, and timing. Fig. 11(a)–(c) show the post-route layouts from *RePlAce*, *DREAMPlace* and *DG-RePlAce*, respectively. It is clear that one of the CPU cores (marked in yellow) gets mixed up when placed by *RePlAce* and *DREAMPlace*, resulting in worse wirelength, power, and timing. Additionally, we notice that *DG-RePlAce* is 2*X* faster than *DREAMPlace* in terms of GP runtime, but has total TAT larger than *DREAMPlace*. This is because it takes 134 s to load the design into OpenROAD. Subtracting the loading time, the TAT for *DG-RePlAce* drops to 66 s. These findings are consistent with the runtime analysis presented in Section IV-B.

For the MemPool Group design, *DG-RePlAce* achieves significantly better-timing (TNS) compared to both *RePlAce* and *DREAMPlace*. However, *DG-RePlAce* suffers from 3% wirelength degradation over *DREAMPlace*. To understand this wirelength increase, we use early global route (eGR) in a commercial place-and-route tool (Cadence Innovus 21.1) to examine the congestion maps for placements generated

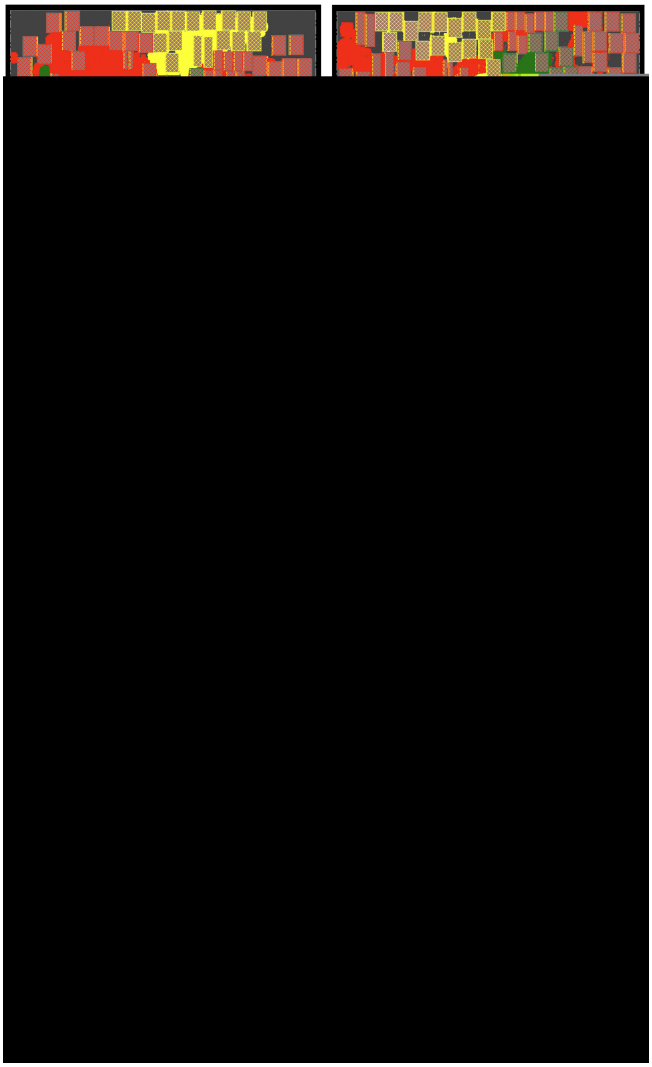

Fig. 11. Post-route layouts of BlackParrot and MemPool Group designs with different flows. Design (Flow): (a) BlackParrot (*RePlAce*); (b) BlackParrot (*DREAMPlace*); (c) BlackParrot (*DG-RePlAce*); (d) MemPool Group (*RePlAce*); (e) MemPool Group (*DREAMPlace*); and (f) MemPool Group (*DG-RePlAce*). The layouts from *DREAMPlace* are generated with the default hyperparameter settings [33]. For the same design, each module maintains consistent coloring across different layouts.

by *DG-RePlAce* and *RePlAce*. We observe that the placement from *DG-RePlAce* is free from congestion, while there are 0.05% horizontal and 0.02% vertical congestion in the placement from *DREAMPlace*. This accounts for the increased wirelength with *DG-RePlAce*, as *DG-RePlAce* tries to distribute instances more evenly due to the bloat-shrink methodology (see Section III-B)—but at the cost of wirelength degradation. We leave how to achieve better wirelength and congestion tradeoffs as a direction for future work.

We also notice that *DREAMPlace*, using its default parameter settings [33], terminates because it reaches its maximum number of iterations (*iteration* = 1000 by default). To prevent such early termination, we rerun *DREAMPlace* with updated hyperparameters that leave ample margin: *ignore_net_threshold* = 1e9 and *iterations* = 5000; we denote these runs as *DREAMPlace**. These hyperparameters are also set to be the default limits on net filtering and iteration count for *RePlAce* and *DG-RePlAce*. As shown in Table VI,

TABLE VII
RUNTIME BREAKDOWN FOR THE MEMPOOL GROUP AND MEGABOOM_X4 BENCHMARKS. EFFECTIVE TAT IS THE NET TAT, CALCULATED BY SUBTRACTING THE TIME SPENT ON HANDLING INPUT AND OUTPUT FILES (IO) FROM THE TOTAL TAT. *DREAMPlace** REPRESENTS RUNNING *DREAMPlace* WITH THE UPDATED HYPERPARAMETERS *ignore_net_threshold* = 1E9 AND *iterations* = 5000

Design	Global Placer	Convergence Iterations	GP (s)	IO (s)	TAT (s)	Effective TAT (s)
MemPool Group	*RePlAce*	690	1896	332	2712	2380
	DREAMPlace	1001	72	95	167	72
	*DREAMPlace**	1091	178	95	284	189
	DG-RePlAce	530	122	332	591	259
MegaBoom_X4	*RePlAce*	870	7433	370	8546	8176
	DREAMPlace	993	319	230	550	319
	*DREAMPlace**	1036	881	230	1113	883
	DG-RePlAce	770	418	370	937	567

TABLE VIII
EXPERIMENTAL RESULTS ON THE MEGABOOM_X4 DESIGN. DATA POINTS FOR WL ARE NORMALIZED

Design	Std Cells	Nets	Global Placer	WL	Horizontal Congestion	Vertical Congestion
MegaBoom_X4	5807K	5831K	*RePlAce*	1.00	0.01%	0.07%
			DREAMPlace	1.00	0.02%	0.08%
			*DREAMPlace**	1.00	0.01%	0.08%
			DG-RePlAce	1.00	0.00%	0.08%

*DREAMPlace** delivers better results compared to the default configuration of *DREAMPlace*, but at the cost of increased GP runtime and total TAT. Even with the updated hyperparameters, *DG-RePlAce* continues to outperform *DREAMPlace** in terms of timing metrics (WNS and TNS).

Additionally, for the MemPool Group design, the GP runtime of *DG-RePlAce* exceeds that of *DREAMPlace*. To delve into the reasons behind the GP runtime degradation, we examine the detailed runtime breakdown for both *DG-RePlAce* and *DREAMPlace*. The results are presented in Table VII. We can observe the following conclusions.

1) The GP runtime of *DREAMPlace* increases by 2.47× when considering all the signal nets during placement.
2) With the same hyperparameter settings, *DG-RePlAce* is 1.46× faster than *DREAMPlace** in terms of GP runtime, while its total TAT is larger than that of *DREAMPlace**.
3) *DG-RePlAce* converges in fewer iterations compared to *DREAMPlace**, which accounts for its 1.46*X* speedup of GP runtime. The runtime per iteration of *DG-RePlAce* is longer than that of *DREAMPlace**, due to the additional pseudo nets introduced by the dataflow constraints (Section III-B) and datapath constraints (Section III-C). By removing these pseudo nets, the GP runtime decreases from 122 s to 68 s. To further confirm this speedup, we run the same experiments on another large design, MegaBoom_X4 (four-core RISC-V MegaBoom [42]), which has more than 5.8M instances. Experimental results on MegaBoom_X4 (shown in Table VII) give support to our analysis. Detailed metrics for MegaBoom_X4 are presented in Table VIII.[9]
4) We also study the effect of *ignore_net_threshold* on *DG-RePlAce*. We sweep the *ignore_net_threshold* values across 50, 100, 200, 500, 1000. The results are shown in Fig. 12. We can see that increasing *ignore_net_threshold* from 50 to 1000 results in approximately 1% improvement in wirelength. However, the improvement in TNS is significantly more substantial.
5) After subtracting the time spent on handling file input and output (IO) from the total TAT, the net TAT ("Effective TAT") of *DG-RePlAce* is smaller than that of *DREAMPlace**. As pointed out in Section IV-B, we attribute this to the *physical hierarchy extraction* process. Enhancing the efficiency of the *physical hierarchy*

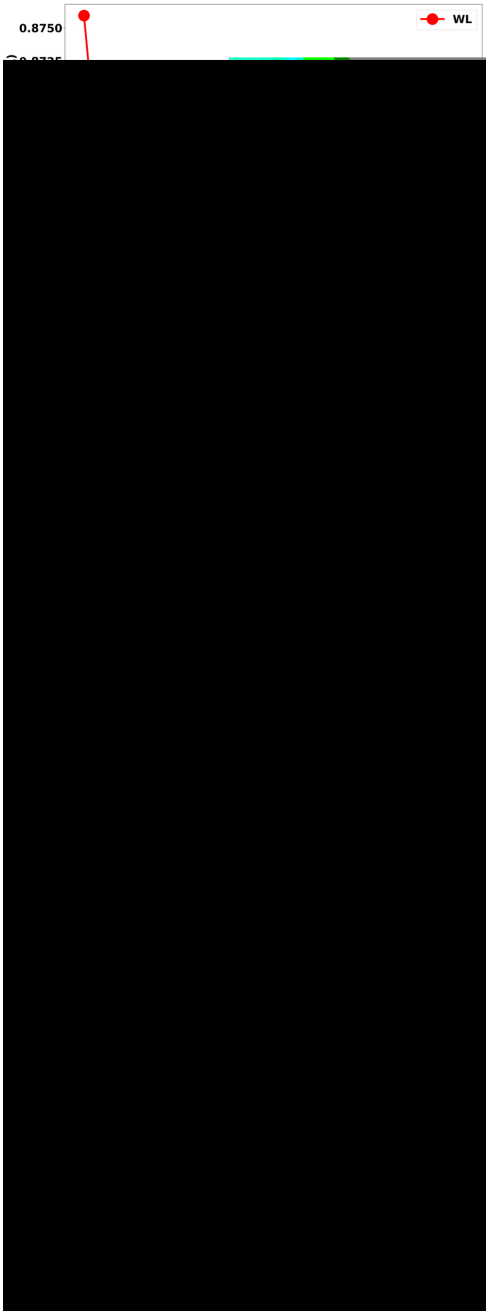

Fig. 12. Effect of *ignore_net_threshold* on *DG-RePlAce*. All the metrics are collected after completion of post-route optimization and normalized as described in Section IV-A. (a) BlackParrot (normalized wirelength). (b) BlackParrot (normalized TNS). (c) MemPool Group (normalized wirelength). (d) MemPool group (normalized TNS).

[9]Since we cannot finish the place-and-route flow for the MegaBoom_X4 design, we report metrics (wirelength, horizontal and vertical congestion) after early global routing in the commercial tool.

extraction process is a key objective for our future research efforts.

V. Conclusion and Future Work

In this work, we develop *DG-RePlAce*, a new and fast GPU-accelerated GP framework which is built on top of the OpenROAD infrastructure [30], and which exploits the inherent dataflow and datapath structures of machine learning accelerators to achieve superior results. Experimental results show that *DG-RePlAce* outperforms both *RePlAce* and *DREAMPlace* in terms of routed wirelength and TNS metrics. Extensions to *DG-RePlAce* that we are currently exploring include: 1) incorporation of density screens for routability and virtual resizing for timing optimization; 2) application of ML-based multiobjective optimization methods to autotune the hyperparameters of *DG-RePlAce*, potentially achieving better-tradeoffs across wirelength, congestion, power and timing; and 3) improving the runtime of the *physical hierarchy extraction* process. In combination with open-sourcing and OpenROAD integration, we believe that this work will add to the foundations for new research on fast and high-quality global placers.

References

[1] A. Agnesina et al., "AutoDMP: automated DREAMPlace-based macro placement," in *Proc. ISPD*, 2023, pp. 149–157.

[2] S. Chou, M.-K. Hsu, and Y.-W. Chang, "Structure-aware placement for datapath-intensive circuit designs," in *Proc. DAC*, 2012, pp. 762–767.

[3] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang, "RePlAce: Advancing solution quality and routability validation in global placement," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* vol. 38, no. 9, pp. 1717–1730, Sep. 2019.

[4] Y.-C. Chang, T.-W. Lin, I. H.-R. Jiang, and G.-J. Nam, "Graceful register clustering by effective mean shift algorithm for power and timing balancing," in *Proc. ISPD*, 2019, pp. 11–18.

[5] H. Esmaeilzadeh et al., "VeriGOOD-ML: An open-source flow for automated ML hardware synthesis," in *Proc. ICCAD*, 2021, pp. 1–7.

[6] D. Fang, B. Zhang, H. Hu, W. Li, B. Yuan, and J. Hu, "Global placement exploiting soft 2D regularity," in *Proc. ISPD*, 2022, pp. 203–210.

[7] J. Cong and Y. Zou, "Parallel multi-level analytical global placement on graphics processing units," in *Proc. ICCAD*, 2009, pp. 681–688.

[8] F. Gessler, P. Brisk, and M. Stojilovič, "A shared-memory parallel implementation of the RePlAce global cell placer," in *Proc. VLSID*, 2020, pp. 78–83.

[9] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. ISCA*, 2017, pp. 1–12.

[10] A. B. Kahng, R. Varadarajan, and Z. Wang, "RTL-MP: Toward practical, human-quality chip planning and macro placement," in *Proc. ISPD*, 2022, pp. 3–11.

[11] A. B. Kahng, R. Varadarajan, and Z. Wang, "Hier-RTLMP: A hierarchical automatic macro placer for large-scale complex IP blocks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 43, no. 5, pp. 1552–1565, May 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10372220

[12] A. B. Kahng, S. Kang, S. Kundu, K. Min, S. Park, and B. Pramanik, "PPA-relevant clustering-driven placement for large-scale VLSI designs," in *Proc. DAC*, 2024, pp. 1–6.

[13] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science* vol. 220, no. 4598, pp. 671–680, 1983.

[14] J. Lu et al., "ePlace: Electrostatics-based placement using fast fourier transform and Nesterov's method," *ACM Trans. Design Autom. Electron. Syst.* vol. 20, no. 2 p. 17, 2015.

[15] C.-X. Lin and M. D. F. Wong, "Accelerate analytical placement with GPU: A generic approach," in *Proc. DATE*, 2018, pp. 1345–1350.

[16] Y. Lin et al., "DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* vol. 40, no. 4, pp. 748–761, Apr. 2021.

[17] Y.-C. Lu, S. Pentapati, and S. K. Lim, "The law of attraction: Affinity-aware placement optimization using graph neural networks," in *Proc. ISPD*, 2021, pp. 7–14.

[18] J.-M. Lin, W.-F. Huang, Y.-C. Chen, Y.-T. Wang, and P.-W. Wang, "DAPA: A dataflow-aware analytical placement algorithm for modern mixed-size circuit designs," in *Proc. ICCAD*, 2021, pp. 1–8.

[19] J.-M. Lin, Y.-L. Deng, Y.-C. Yang, J.-J. Chen, and P.-C. Lu, "Dataflow-aware macro placement based on simulated evolution algorithm for mixed-size designs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 5, pp. 973–984, May 2021.

[20] P. Liao, S. Liu, Z. Chen, W. Lv, Y. Lin, and B. Yu, "DREAMPlace 4.0: Timing-driven global placement with momentum-based net weighting," in *Proc. DATE*, 2022, pp. 939–944.

[21] Y. Liu et al., "GraphPlanner: Floorplanning with graph neural network," *ACM Trans. Des. Autom. Electron. Syst.* vol. 28, no. 2, p. 21, 2022.

[22] Y.-C. Lu, T. Yang, S. K. Lim, and H. Ren, "Placement optimization via PPA-directed graph clustering," in *Proc. MLCAD*, 2022, pp. 1–6.

[23] L. Liu, B. Fu, M. D. F. Wong, and E. F. Y. Young, "Xplace: An extremely fast and extensible global placement framework," in *Proc. DAC*, 2022, pp. 1309–1314.

[24] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "VLSI module placement based on rectangle-packing by the sequence-pair," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* vol. 15, no. 12, pp. 1518–1524, Dec. 1996.

[25] R. X.T. Nijssen and J. A.G. Jess, "Two-dimensional datapath regularity extraction," in *Proc. ACM/SIGDA Physical Design Workshop*, 1996, pp. 111–117.

[26] P. Quinton, "An introduction to systolic architectures," in *Future Parallel Computers: An Advanced Course*, P. Treleaven and M. Vanneschi, Eds., Heidelberg, Germany: Springer, 1987, pp. 387–400.

[27] A. Vidal-Obiols, J. Cortadella, J. Petit, M. Galceran-Oms, and F. Martorell, "Multilevel dataflow-driven macro placement guided by RTL structure and analytical methods," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* vol. 40, no. 12, pp. 2542–2555, Dec. 2021.

[28] S. Ward, D. Ding, and D. Z. Pan, "PADE: A high-performance placer with automatic datapath extraction and evaluation through high-dimensional data learning," in *Proc. DAC*, 2012, pp. 756–761.

[29] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. Boston, MA, USA: Addison-Wesley Publ. Co., 2010.

[30] "OpenROAD." Aug. 2024. [Online]. Available: https://github.com/The-OpenROAD-Project/OpenROAD

[31] "DREAMPlace 4.0.0." Jun. 2024. [Online]. Available: https://github.com/limbo018/DREAMPlace

[32] "TILOS AI Institute, MacroPlacement repository." Jun. 2023. [Online]. Available: https://github.com/TILOS-AI-Institute/MacroPlacement

[33] "Default hyperparameter settings for DREAMPlace." 2021. [Online]. Available: https://github.com/limbo018/DREAMPlace/blob/master/test/ispd2019/lefdef/ispd19_test1.json

[34] "Default hyperparameter settings for RePlAce." 2020. [Online]. Available: https://github.com/The-OpenROAD-Project/RePlAce/blob/cf289bb141a995d8304656cd994ba3f2f95b2f8a/src/nesterovPlace.cpp#L24

[35] "BlackParrot repository." 2024. [Online]. Available: https://github.com/black-parrot/black-parrot

[36] "MemPool repository." Jun. 2024. [Online]. Available: https://github.com/pulp-platform/mempool

[37] "Ablation (artificial intelligence)." 2024. [Online]. Available: https://en.wikipedia.org/wiki/Ablation_(artificial_intelligence)#::text=An

[38] "OpenDB." 2024. [Online]. Available: https://github.com/The-OpenROAD-Project/OpenROAD/tree/master/src/odb

[39] "Hier-RTLMP." 2024. [Online]. Available: https://github.com/The-OpenROAD-Project/OpenROAD/tree/master/src/mpl2

[40] "The DG-RePlAce repository." 2024. [Online]. Available: https://github.com/ABKGroup/DG-RePlAce

[41] "Himax Technologies." 2024. [Online]. Available: https://www.himax.com.tw/company/about-himax

[42] "BOOM: The Berkeley out-of-order RISC-V processor." 2024. [Online]. Available: https://github.com/riscvboom/

[43] "VeriGOOD-ML: Verilog generator, optimized for designs for machine learning." 2024. [Online]. Available: https://github.com/VeriGOOD-ML/public/blob/main/backend_flow/tabla/design01/script/floorplan.tcl

[44] "Dynamic and internal power." 2008. [Online]. Available: https://asic-soc.blogspot.com/2008/04/dynamic-and-internal-power.html

[45] "Code for penalty factor calculation." 2024. [Online]. Available: https://github.com/ABKGroup/DG-RePlAce/blob/579378a4441c1ce57c4e89365dbdd97490364f9c/src/gpl2/src/placerBase.cu#L1714

Andrew B. Kahng (Fellow, IEEE) received the Ph.D. degree in computer science from the University of California at San Diego, La Jolla, CA, USA, in 1989.

He is a Distinguished Professor with the CSE and ECE Departments, University of California at San Diego. His interests include IC physical design, the design-manufacturing interface, large-scale combinatorial optimization, AI/ML for EDA and IC design, and technology roadmapping.

Zhiang Wang (Member, IEEE) received the M.S. degree in electrical and computer engineering from the University of California at San Diego, La Jolla, CA, USA, in 2022, where he is currently pursuing the Ph.D. degree.

His current research interests include partitioning, placement methodology, and optimization.