A Scalable, Serially-Equivalent, High-Quality Parallel Placement Methodology Suitable for Modern Multicore and GPU Architectures

Christian Fobel
School of Computer Science
University of Guelph
Guelph, Ontario, Canada
Email: cfobel@uoguelph.ca

Gary Grewal
School of Computer Science
University of Guelph
Guelph, Ontario, Canada
Email: gwg@uoguelph.ca

Deborah Stacey
School of Computer Science
University of Guelph
Guelph, Ontario, Canada
Email: dastacey@uoguelph.ca

Abstract—**Placement and routing run-times continue to dominate the automated FPGA design flow. As the size of FPGA architectures continue to grow exponentially, it remains critical to develop parallel tools for FPGA design where the amount of exposed concurrent work scales with the size of the designs to be synthesized. In this paper, we propose a novel algorithm for parallel placement, based on simulated annealing, where the amount of parallel work directly scales with the size of the net-list to be placed. Our approach concurrently evaluates and conditionally applies very large sets of non-conflicting swaps using common parallel computing primitives, including stream compaction, category reduction, and sort. While our design is suitable for targeting all modern parallel computing platforms, we present results from our implementation which targets NVIDIA's CUDA platform, where we achieve a mean speed-up of 19x over VPR with post-routing critical-path-delay and wire-length quality that matches or exceeds VPR. We believe that this work is an important step towards the development of a scalable, high-quality placement tool.**

I. Introduction

In *Field-Programmable Gate Array (FPGA)* design, *"placement"* is an NP-complete problem[5], which assigns each block in a netlist to a position on the target architecture, while minimizing one or more objective costs. Heuristic and analytic[7] methods have been developed to provide approximate solutions. However, with the size of target architectures growing exponentially, placement can already take hours or even days to perform for large designs. Thus, it is essential to develop parallel methods for placement that will scale along with the growth of FPGA architectures.

Despite recent work developing parallel techniques, placement times continue to dominate the runtime of the FPGA CAD flow. Several attempts have been made to improve the performance of simulated annealing for placement, which is the most common heuristic applied to placement, but with limited success. The most successful parallel annealing approaches to-date suffer from two main issues. The first issue is that of run-time scalability, where the run-time speed-ups of existing parallel annealers[1], [6], [2] do not scale beyond a small number of threads or processing cores. For example, in [6], the maximum speed-up obtained over a single thread was 21x, with a maximum speed-up of 17.5x over VPR[3]'s annealer. Also, as reported in [2], the speed-ups observed in [6] do not hold on newer architectures, where the maximum speed-up over a single thread was reduced by a factor of approximately 4x compared to the results originally presented in [6]. While the algorithmic alterations made in [2] improved run-time by approximately 46% compared to [6], the maximum speed-up failed to scale for more than 16 threads. The main reason existing parallel placement annealers fail to scale is that the amount of sequential work in the approaches, including synchronization and communication, scales along with the problem size[1], [6], [2], causing the sequential run-time to quickly dominate the overall algorithm run-time as the number of threads increases due to Amdahl's Law. The second key issue with existing parallel annealing techniques is a significant reduction in quality-of-result as the amount of parallelism increases. This is mainly due to the mechanism used to divide up work for parallel execution. In the most promising approaches in literature, the parallel work is generated in a way that varies with the number of processing threads or cores being used. The work is divided up by assigning a subset of the placement area to each processing thread. However, a side effect of dividing up work in this manner is that restrictions are placed on the possible swaps that can be considered within each execution group. For example, in [2], as the number of threads increases, the maximum possible swap distance decreases. The effect of restricted maximum swap distance is equivalent to starting VPR with an `rlim` value that is *significantly* smaller than the extent of each dimension of the placement grid *(i.e., the default starting value in VPR)*, which, as reported in [2], results in significant reductions in quality-of-result. The quality-of-result in the approaches proposed in [6], [2] degrades as the number of threads increases and is between 3-5% worse than VPR in the results presented.

In this paper, we present a novel parallel placement methodology, which directly addresses the issues of 1) run-time scalability, and 2) quality-of-result. Our approach is designed to only use structured parallel primitives[4] to ensure strong run-time scalability. The run-time complexity of the sequential portion of our approach is $O(1)$, such that its run-time does not increase with the size of the net-list to be placed, ensuring that we do not expect our approach to suffer from the negative effects of Amdahl's Law. The parallel portion of our approach accounts for over 99% of total run-time for the experiments discussed in this paper, where all parallel work is conducted using structured parallel primitives, providing a well-balanced load across computing threads through operations including, *map, pack, sort, category reduce,* and *permutation scatter*[4]. There exist highly optimized implementations of all parallel primitives used in our approach in parallel frameworks such as Cilk Plus[4] and NVIDIA's Thrust, which can target all modern, commodity, parallel hardware, including multi-core CPUs and GPUs. Another benefit of using structured parallel primitives is that serial-equivalency is preserved in our approach, assuming that a serially-equivalent category reduction implementation is used[4]. A serially-equivalent parallel algorithm implementation guarantees the same results as a serial implementation of the same algorithm, regardless of the number of parallel threads or cores, enabling consistent solution quality and easier application debugging. The amount of concurrent work in our approach, which scales with the net-list size, is also far higher than the number of cores typically available, providing forward-scaling parallelism to seemlessly benefit from future advances in parallel hardware architectures without any code changes required. Furthermore, our method of decoupling the creation of potential parallelism from the number of execution threads ensures consistent algorithm behaviour, independent of the number of processing cores, eliminating restrictions imposed by the division of parallel work, such as limits on maximum swap distance as imposed in [6], [2].

The remainder of this paper is laid out as follows. Section II describes our hyper-graph cost model that serves as a basis for maintaining and updating net-costs in our parallel annealer, while Section III discusses the design of our structured parallel annealer. In Section IV, we introduce our method for generating large sets of non-overlapping swaps, which is the key to enabling high concurrency in our parallel annealer. In Section V we present our experimental results and in Section VI we conclude our discussion and present future work.

II. Hyper-graph cost model

The parallel FPGA placement methodology we propose in this paper, represents the input net-list as a *hyper-graph* and a placement of the blocks from the net-list onto a target architecture as a *permutation*. In this section, we present a general hyper-graph cost model, which defines a mechanism for recursively computing the cost of a full hyper-graph *(e.g., net-list)*, by defining functions that compute the cost contribution of *each connection* between a *node* and a *hyper-edge*. By recursively aggregating the cost contribution from each connection, our model provides costs on a per-hyper-edge basis, which can be further aggregated to produce a total cost for the entire hyper-graph. Furthermore, we show that, provided an appropriate set of functions, the costs in our model 1) may be computed using structured parallel primitives, and 2) may be updated due to a change in state of a node *(e.g., changing the assigned position of a net-list block in a placement)*. In this section, we discuss our hyper-graph cost model in detail, and illustrate our model using Star+ wire-length cost[7], while Section III introduces our parallel simulated-annealing variant which operates on the cost model described here to concurrently evaluate and conditionally apply very large sets of non-overlapping moves. Note that we selected Star+ as our wire-length cost model because of its effectiveness in reducing critical path length[7].

A. Permutation-based placement encoding

The parallel simulated-annealing variant we propose in Section III uses a permutation-based encoding to represent the current state of a placement. Here, we demonstrate how a placement may be encoded as a one-dimensional permutation, which will allow us to demonstrate our general hyper-graph cost model applied to FPGA placement in the following sections. Using a one-dimensional permutation encoding not only simplifies our implementation, but also generalizes our approach, such that it may be applied other permutation-based problems, by defining problem-specific cost functions.

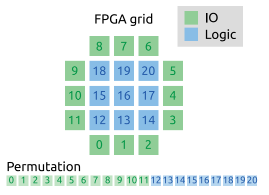
Consider a FPGA architecture, which has several types of target locations where various types of corresponding net-list blocks may be mapped. For example, a typical modern FPGA may contain *Input/Output (I/O)* pads, *Configurable Logic Blocks (CLBs)*, dedicated multipliers, etc., where each such component is associated with a particular (x, y) position on the architecture. Each type of component, *e.g., I/O pad*, typically has an associated capacity, defining the number of net-list blocks which may be mapped to a common (x, y) position, corresponding with a single component instance. In our permutation-encoding, we define the length of a placement permutation to be $\sum_{\forall t \in T} t_c t_T$, where T is the set of component types, t_c denotes the capacity of component type t, and t_T denotes the number of components of type t available on the target architecture. We define each location in the permutation as a *slot*, where we refer to the index of the slot in the permutation as the "slot-key". Each *slot-key* provides a means of referring to a particular permutation position in the calculations performed in our parallel placement methodology.

Figure 1a shows provides an example permutation encoding for an island-style FPGA, consisting of I/O pads and CLBs. Here, each configurable location on the FPGA grid is assigned a corresponding slot-key in the permutation. Note that our methodology places no restriction on the ordering of the mapping of slot-keys to positions on the architecture. The only requirement imposed by our approach is that there exists a function with $O(1)$ time-complexity that returns the (x, y) position assigned to each *slot-key*. Note that since the architecture is fixed for a given placement, even complex architectures may be mapped by using a simple pre-computed look-up table, which defines the slot-key for each position on the architecture. Figure 1a shows one potential slot-key mapping, where all slots corresponding to I/O positions are grouped together, followed by all slots associated with CLB grid positions. We define a *slot-key-to-position* function, $p(s_k)$, which maps each permutation slot-key, s_k, to a (x, y) position on the target architecture. In the example in Fig. 1a, $p(8)$ corresponds to a position of $(1, 4)$ and $p(15)$ corresponds to a position of $(2, 1)$. We denote a permutation representing a placement using the notation P.

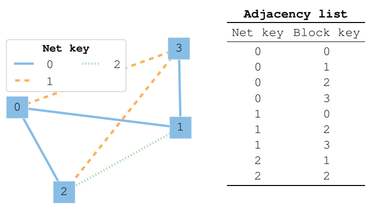
B. Hyper-graph cost functions

Our parallel placement methodology operates on a hyper-graph problem representation. Recall that a hyper-graph is defined as a graph where each edge defines a *set of two or more nodes*, rather than a *pair of nodes*. Note that a net-list may be modeled as a hyper-graph, where each node corresponds to a block in the net-list and each edge corresponds to a net in the net-list, where each net connects two or more blocks together.

1) Adjacency-list: In our approach, we encode a hyper-graph using an *adjacency-list*, which is the set containing every connection that exists between any node *(i.e, block)* and a respective edge *(i.e., net)*. Figure 1b illustrates a simple net-list consisting of four blocks, where the blocks are connected by three nets. As shown in the figure, an adjacency-list can be used to represent the hyper-graph encoding of the net-list, where each net is assigned a key and each block is assigned a key. Each connection in A represents the membership of the respective block to the corresponding net in the net-list. For example, in Fig. 1b, since net 2 is connected to two blocks, there are two corresponding connection pairs in the adjacency-list, one for the connection to block 1, and the other for the connection to block 2. The adjacency-list representation presented here serves as the basis for all operations in our parallel placement approach, described in Section III.



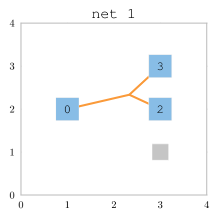
(a) Example slot-key-to-position mapping.



Adjacency list

Net key	Block key
0	0
0	1
0	2
0	3
1	0
1	2
1	3
2	1
2	2

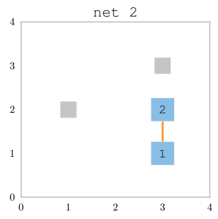
(b) Connections adjacency list.



$$e_{vx}(1) = (7,19) \leftarrow r_{vx} \begin{cases} r_{vx} & \leftarrow \begin{cases} c_{vx}(1,0) & = (1,1^2) = (1,1) \\ c_{vx}(1,2) & = (3,3^2) = (3,9) \end{cases} \\ & \leftarrow \begin{cases} c_{vx}(1,3) & = (3,3^2) = (3,9) \end{cases} \end{cases}$$

$$e_{cx}(1) = \alpha\sqrt{\sum x^2 - \frac{\{\sum x\}^2}{|e|}} + \beta = 1.59\sqrt{19 - \frac{7^2}{3}} + 1.00$$
$$= 3.04$$

(c) Net 1 *(x-dimension only)*.



$$e_{vy}(2) = (3,5) \leftarrow r_{vy} \begin{cases} r_{vy} & \leftarrow \begin{cases} c_{vy}(2,1) & = (1,1^2) = (1,1) \\ c_{vy}(2,2) & = (2,2^2) = (2,4) \end{cases} \end{cases}$$

$$e_{cy}(2) = \alpha\sqrt{\sum y^2 - \frac{\{\sum y\}^2}{|e|}} + \beta = 1.59\sqrt{5 - \frac{3^2}{2}} + 1.00$$
$$= 1.95$$

(d) Net 2 *(y-dimension only)*.

Fig. 1: Hyper-graph cost model a) net-list adjacency list, and Star+ functions applied to to b) hyper-edge *(i.e., net)* 1, and c) hyper-edge 2.

2) Connection-values: Given an adjacency-list, A, we can assign a representative value to each connection in the adjacency-list, based on characteristics of the associated node *(i.e., block)* and edge *(i.e., net)*. Formally, we can define a function $c_v(n, e)$, which maps each connection in the adjacency-list to what we define as a *"connection-value"*, where n denotes the key of the node *(i.e., block)* and e denotes the key of the edge *(i.e., net)*. In the case of Star+, we define $c_v(n, e)$ as shown in Eq. 1, where $s(n)$ corresponds to the permutation slot-key in P currently assigned to the node n. Recall that the function $p(s)$ maps each slot-key to an (x, y) position on the target FPGA architecture.

$$c_v(n, e) = (p(s(n)), p(s(n))^2) \tag{1}$$

In Fig. 1c, we demonstrate the computation of the *connection-value* for each connection corresponding to net 1 *(e.g., $c_{vx}(1, 2)$)*. Note that only the x-dimension position components are shown due to space restrictions. Fig. 1d demonstrates similar calculations for the y-dimension of connections associated with net 2. By applying the function c_v to each connection in the adjacency-list, we obtain a list of *connection-values* for the entire hyper-graph, with one *connection-value* per node-edge pair. We define the list of *connection-values* for all connections in the adjacency-list as C_v.

3) Hyper-edge values: As described above, by applying the function c_v to permutation P, we can compute the list of connection-values, C_v for all connections in the adjacency-list of the hyper-graph H. Given C_v, we define a *connection-reduction function* as a recursive function $r_v(a, b)$, where a and b each represent either a connection-value, or a result from another application of the r_v. Provided with function c_v, and a recursive *reduction* function, r_v, to recursively combine connection-values together, we can form a function $e_v(e)$, such that $e_v(e)$ corresponds to the reduction of connection-values for all connections in the adjacency-list, A, that are connected to the hyper-edge e. We define each mapping of e_v applied to a hyper-edge as an "edge-value". We use E_v to denote the set of *edge-values* for all hyper-edges in H.

In the case of Star+, we define r_v as element-wise addition. As shown in Fig. 1c and Fig. 1d, this definition of r_v can be used to recursively reduce all connection-values corresponding to an edge *(i.e., net)* to a single *edge-value*. For example, $e_{vx}(1)$ corresponds to the reduced x-dimensional edge-value corresponding to net 1. As shown in Fig. 1d, the y-dimensional component is computed in a similar manner.

4) Hyper-edge costs: Given, E_v, containing the *edge-value* of each hyper-edge in H, we can define a function $e_c(e_v(e))$, which maps each *edge-value* to a corresponding cost. By applying the function e_c to each edge-value in E_v, we obtain the set E_c, which corresponds to the cost of each hyper-edge in H, i.e., the cost of each net in the net-list. Aggregating all costs in E_c by adding them together, we obtain a summary cost for the entire hyper-graph H, which we denote as H_c.

In the case of Star+, we define e_c as shown in Fig. 1c and Fig. 1d, based on a form of the Star+ cost function[7]. In the context of FPGA placement, H_c corresponds to the total net-list cost for a particular placement. As shown above, given a placement encoded as a permutation, we can apply the Star+ definitions of the c_v, r_v, e_v, and e_c functions to compute the cost of each net in the net-list, and the total cost of the net-list, based on the placement encoded by the permutation. Note that while we use Star+ in the implementation proposed in this paper, hyper-graph cost model functions may be defined to integrate other net-cost models, such as *bounding-box*. Section III describes how the hyper-graph cost model presented above may be used as the basis for a structured, highly-parallel method for concurrently evaluating very large sets of permutation swaps.

III. Structured Parallel Annealer (SPA)

In this section we describe our placement approach, which we call *Structured Parallel Annealer (SPA)*. Our placement method is based on the annealing algorithm used in VPR, where a set number of random swaps, which we denote as S_t, is evaluated at each annealing temperature state. As in VPR, any swap resulting in a cost improvement is accepted, while non-improving swaps are conditionally accepted based on the current temperature and a stochastic variable [3]. The temperature and radius-limit parameters[3] are updated after every set of S_t swaps is evaluated. The anneal continues until the temperature is very low compared to the average cost of a net, at which point the placement state is taken as the final solution.

The key difference between our method and the VPR annealer is that, rather than evaluating S_t swaps, one after another, at each temperature state, we concurrently evaluate very large sets of swaps until S_t swaps have been evaluated. Note that although this sounds straight-forward, it is only possible because of our novel technique *(described in Section IV)* for generating very large sets of swaps in a scalable way, which guarantees no *hard-conflict*[6] will occur between swaps in the same set. A hard-conflict occurs when a block is involved in more than one concurrent move, resulting in an inconsistent placement.

Apart from evaluating swaps in parallel, the remainder of our algorithm mimics the behaviour of the annealer in VPR, including the radius-limit and temperature update schedule. Section III-A describes how we evaluate *and conditionally apply* swaps concurrently, using only structured parallel primitives applied to the Star+ hyper-graph cost model discussed in Section II.

A. Concurrent evaluation and application of move-pairs

1) Inverse connection value reduction function: Recall from Section II that we can use the e_v function to compute a cost for the corresponding edge using the e_c function. To evaluate swaps efficiently, it is critical that we are able to quickly compute the difference in cost to a net due to a connected block undergoing a displacement from its current position. To enable this within the context of our hyper-graph cost model from Section II, we define an inverse reduction function $r_v^{-1}(e_v(e), c_v(n, e))$ which computes the edge-value associated with edge e, but not including the contribution of the connection between node n and edge e. In terms of Star+, this corresponds to subtracting the connection-value associated with a particular block from the corresponding net-value. Using this inverse reduction operator, it is possible to take a pre-computed net value, $e_v(e)$, and subtract the contribution from a block that we would like to move, after which we can reduce again using r_v with the updated connection-value based on the proposed displaced position for the block being moved. The r_v^{-1} function is used by our parallel annealer, described below.

2) Iteration of concurrent swaps: In our approach, during each round of concurrent swaps, each block is assigned a displacement *(may be zero, if the block should not move)*. All swaps are evaluated by evaluating the difference in cost resulting from applying the displacement assigned to each block based on the moves defined by the current set of block displacements. Figure 2 depicts an example iteration of applying concurrent swaps using our methodology. The *displacements* structure, D, lists the displacement assigned to each block in the current iteration, both in terms of (x, y) position *(labelled as d_x and d_y in the figure)*, and in terms of slot-index *(labelled as d_s)*. The *permutation* structure, P, indicates the slot-key, and the corresponding (x, y) position, currently assigned to each block. In other words, P defines the current placement state. Note that all block displacements in D are matched such that where any block is assigned a displacement that targets a position currently occupied by another block, the displacement assigned to the block occupying the target slot is of equal magnitude, but in the opposite direction. While this is discussed in greater detail in Section IV, it is this matching of block moves into what we call, "move-pairs", that avoids hard-conflicts from occurring.

Each iteration of swaps starts by pre-computing the edge-value associated with each net in the net-list. This computation is done by applying a parallel *map* to compute the connection-value for each connection in A using the function c_v, followed by a parallel *category reduce* operation to recursively compute a reduced edge-value for each net using the function r_v. Next, a parallel *map* operation is applied to the connection-value for each connection in A using the inverse reduction function, r_v^{-1}, and the displacement assigned to the corresponding block to compute the difference in cost to the connection due to moving the block by the specified displacement. Note that the parallel *map* operation is fused with a parallel *pack* to only compute and gather the difference in cost of each connection where the respective block is assigned a *non-zero* displacement. In Fig. 2, the calculation of the difference in cost in the x dimension is shown for connection number 4, where block 0 is moved with an x-displacement of $+2$, resulting in a difference in cost for net 1. The result of the fused map-and-pack is the *connection deltas* structure. Note that, in addition to storing the estimated difference in cost for each connection based on the proposed block displacements, the *connection deltas* structure also holds what we refer to as the *positive slot-key*. Each swap between two blocks consists of two equal magnitude displacements, but in opposite directions. We refer to the lowest index of the permutation slot in P containing one of the blocks in each swap as the "positive slot-key", since the block occupying the slot is assigned a positive slot-displacement, d_s. For example, in Fig. 2, blocks 0 and 1 are involved in a common swap, where block 0 occupies the slot with lowest index, resulting in a positive slot-key of 1. Using the positive slot-key of each connection delta to group delta costs based on the respective pair of moves, a parallel *category reduction* is applied to recursively add together all connection deltas that correspond to each swap. A parallel *map* which applies an assessment to the total difference in cost due to each swap is fused with a parallel *permutation scatter* to update the placement based on any swaps that were accepted. The set of parallel operations described in this section is repeated multiple times at each temperature state until S_t swaps have been evaluated.

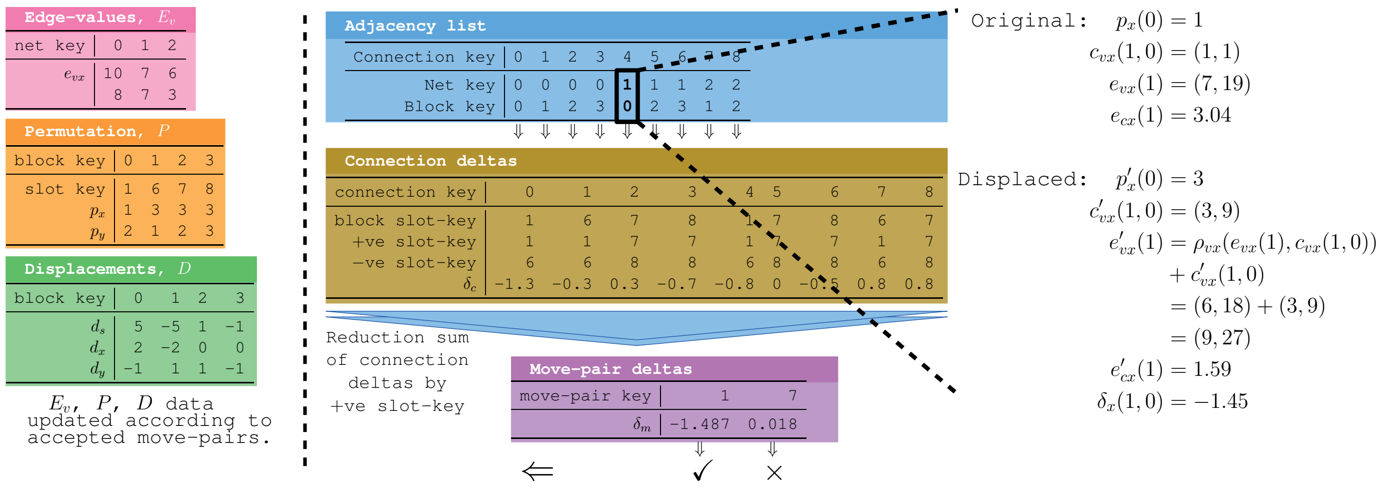


Fig. 2: Example hyper-graph anneal iteration.

IV. Parallel move-pairs

In this section we describe our method for generating very large sets of non-conflicting swaps, where the number of swaps per set scales *directly* with the *size of the net-list*. We have developed a novel method to quickly generate many *(i.e., hundreds to thousands of)* pairs of permutation slots, where each pair may be considered for swapping contents, similar to a swap operation in VPR, but where the definition of our swaps guarantee that each block belongs to *at most* one pair. This ensures that multiple swaps may be applied from the set concurrently without the risk of a *hard-conflict*[6]. Recall that a hard-conflict occurs when a block is involved in more than one concurrent move, resulting in an inconsistent placement. In addition to *hard-conflicts*, *soft-conflicts* may arise when applying concurrent moves to blocks which are connected to the same net[6]. However, as shown in our results in Section V, we find, as in [6], these soft-conflicts have little to no appreciable impact on solution quality. In fact, our experiments show a marked improvement in critical-path-delay and wire-length compared to VPR's annealer in several cases.

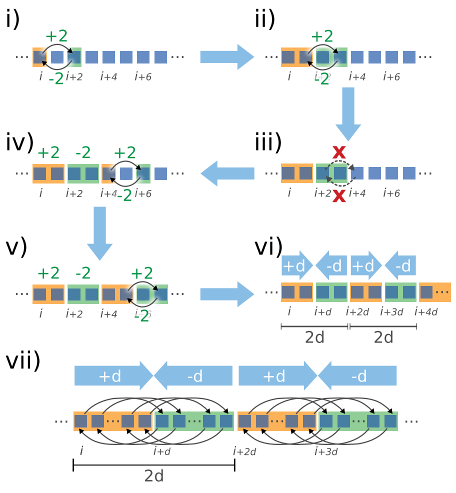
We begin by describing our approach applied to a one-dimensional permutation, that could be applied directly to the permutation, P. However, since, when performing FPGA placement, it is useful to generate moves based on constraints involving the two-dimensional position associated with each permutation slot, we will extend the core concepts of our approach to two-dimensional displacements assigned to positions on the FPGA grid, which can easily be *flattened* into one-dimensional slot displacement relative to the permutation P.

As described in Section III, since each swap may be viewed as the result of moving two blocks with equal, *but opposite*, displacements, we redefine each swap as a *"move-pair"*, which consists of two ordered *permutation slot-keys* and a *displacement-magnitude*. Recall that within a *move-pair*, we call the key of the slot with the lowest index the *"positive"*-slot-key, and the key corresponding to the slot with the highest index the *"negative"*-slot-key. Note that the *negative-slot-key* can be reached by *adding* the displacement-magnitude to the *positive-slot-key*.

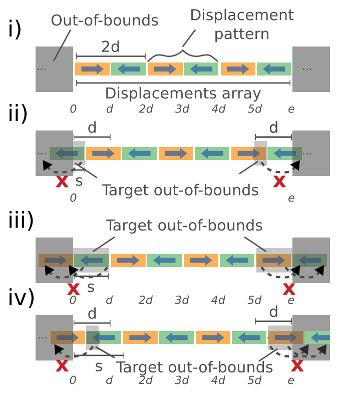
By mapping move-pairs with the same displacement-magnitude to consecutive permutation slots in a specific pattern, we can create non-overlapping pairs of slots, completely avoiding hard-conflicts. For example, let us consider a move-pair with a *displacement-magnitude* of 2, as shown in Fig. 3ai, between slots with keys `i` and `i+2`. As shown in Fig. 3aii, a second move-pair can be defined using index `i+1` as the positive-slot-key. Note that the second move-pair shares the same displacement-magnitude used for the first move-pair. As shown in Fig. 3aiii, the next slot in the permutation, with slot-key `i+3`, already belongs to a move-pair. Note that assigning a second move-pair to slot `i + 3` would result in a hard-conflict. In general, we skip slots that already belong to a move-pair and seek the next available slot to create a new move-pair with the same displacement-magnitude, as shown in Fig. 3aiv. This process can be repeated to create a series of move-pairs which are grouped in contiguous sets where the slots in the first half of each set have a positive displacement *(i.e., $+d$)*, and the slots in the second half of the set have an equal displacement, but in the opposite direction *(i.e., $-d$)*. As shown in Fig. 3avi-vii, the displacements of contiguous slots follow a cycle, where the period is twice the displacement length. We refer to each period of alterating displacements *(half positive, then half negative)* as a *"displacement-pattern"*. These *displacement-patterns* have a length of twice the corresponding displacement-magnitude *(i.e., $2d$)*. The patterns continue in a repeated sequence to fill the permutation. Fig. 3avii illustrates how this concept can be extended to any selected displacement-magnitude. In Section IV-A, we discuss displacement-patterns in more detail, including how to handle sections of a displacement-pattern that result in a move with a target that is out of bounds, i.e., beyond the edge of the FPGA.

A. Series of displacement-patterns

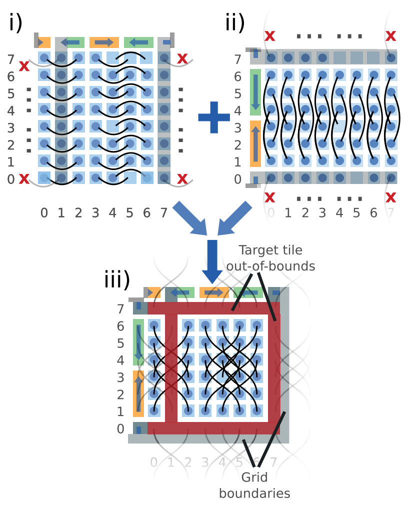
In the previous section, we introduced the concept of a *displacement-pattern*, which consists of a set of displacements, with the first half in the positive direction and the second half in the negative direction. While the displacement-patterns in Fig. 3a are relative to the entire permutation P, Fig. 3b illustrates a scenario where a series of displacement-patterns with the same magnitude are applied to the permutation slots



(a) Displacement relative to P



(b) Displacements w.r.t. row on FPGA



(c) Displacements w.r.t. 2-dimensional FPGA grid.

Fig. 3: Parallel move-pair generation.

within a single row of the FPGA grid. Note that the series of displacement-patterns begins in Fig. 3bi at first slot in the row. However, we may apply a *shift* to the series, such that the first *complete* displacement-pattern starts at the position corresponding to the specified *shift* value, as shown in Fig. 3bii-Fig. 3biv. The pattern produced by increasing the shift repeats at every multiple of $2d$. Therefore, each effective shift value, which we denote as s, falls within a range such that $0 \leq s < 2d$. Notice that in some cases, a section of one or more displacement-patterns that overlap with either the upper- or lower-boundary of the FPGA row cause some moves to target positions that are outside the grid boundaries. The largest number of moves that can be out-of-bounds at each edge of the row is equal to d. To ensure moves remain within the FPGA, d is set to zero for each slot originally assigned a target that is out-of-bounds. This includes any *negative-slot* with a slot-key less than d, or any *positive-slot* with an slot-key that is within d positions *(i.e., half the period of the displacement-pattern)* from the end of the row. Note that the same rules apply when the extent, *i.e.,* e, is not a multiple of $2d$. In general, any slot assigned a move with a target slot-key relative to the FPGA row, s_{rk}, where $s_{rk} < 0$ or $s_{rk} \geq e$ has its displacement set to zero.

B. *Extending to two-dimensions*

The examples considered so far only target a single row of a placement grid. However, the concepts described can easily be extended to multiple rows, by generating displacement patterns in two *(or more)* dimensions. As shown in Fig. 3c, we can generate a *series of displacement-patterns* along *each dimension* of the FPGA, creating an *array* of displacements, defining the displacement to be applied to each location on the FPGA. Note that every curved line within the grid represents a *move-pair*, that is, a pair of locations on the FPGA corresponding to slots in P, and the magnitude of the displacement that must be applied to the locations to exchange contents between them. We call this array of displacements the *"displacement-array"* for each dimension. Each *displacement-array* defines the displacement to be applied to each FPGA location along the corresponding dimension. In the case of two-dimensions, this results in each FPGA location being assigned two *orthogonal* displacement components, which we call d_x and d_y, that combine to form a two-dimensional *displacement-vector*, which we denote as d_T. We denote the extent of each row as e_x and the extent of each column as e_y.

For example, consider the FPGA location at grid row 6 and column 0, shown in Fig. 3ciii. According to the orthogonal displacement-patterns, this location is assigned displacements $d_x = +2$ *(see Fig. 3ci)* and $d_y = -3$ *(see Fig. 3cii)*. However, if we consider the location at row 1 and column 7, the lateral displacement assigned in Fig. 3ci would result in a target that is outside of the FPGA (i.e., $x_i > e_x - 1$). Similarly, the location at row 0 and column 2 has a vertical displacement of $d_y = -3$ assigned, as shown in Fig. 3cii, that would result in a target position outside the FPGA (i.e., $y_i < 0$).

To deal with displacements resulting in target positions that are out of bounds, we assign no displacement at all to any FPGA location with a displacement that targets either a row or column that is out-of-bounds. The resulting slot displacements from applying this method are depicted in Fig. 3ciii. Notice that two rows and two columns of locations are assigned *no displacement at all* (i.e., $d_T = (0, 0)$), since the corresponding contents would be moved out-of-bounds in at least one dimension based on the original displacements.

C. *Generating sets of move-pairs*

As shown in the previous section, a two-dimensional displacement can be generated for a location in the FPGA grid by combining two completely independent, orthogonal displacement components. Moreover, provided with a displacement-magnitude, we can create consecutive, non-overlapping move-pairs to cover an entire row or column of an FPGA. Figure 4 illustrates how to generate a displacements-array for the x dimension, given the current annealing radius-limit parameter (i.e., $rlim$) and the extent of the x-dimension of the FPGA architecture, e_x. First, a displacement-magnitude, d_x is randomly selected, bounded by both e_x and $rlim$. Based on the selected value of d_x, a shift value, s, is randomly chosen, bounded by the period of the corresponding displacement-pattern, $2d_x$. Using the randomly selected d_x and s values, we can construct the displacement-array for the x-dimension, as described in Section IV, which we denote as D_x. The same procedure is applied to the y-dimension to generate D_y. Each slot with key, s_k, corresponding to position $p(s_k) = (x_k, y_k)$ on the FPGA is then assigned flattened, slot-displacement $(e_x d_{xk}, d_{yk})$, where e_x is the extent of the x-dimension of the FPGA, $d_{xk} = D_x[x_k]$ and $d_{yk} = D_y[y_k]$.

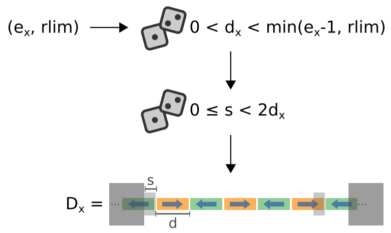


Fig. 4: Generate displacements-array for x-dimension

V. Experiments

To evaluate the efficacy of our proposed *SPA* design with respect to run-time scalability and quality-of-result, we developed a CUDA implementation of SPA using the Thrust parallel library. We used our CUDA SPA implementation along with VPR to conduct several sets of experiments. First, we ran VPR using the bounding-box cost function with `inner_num=1`, across a set of 7 large benchmark net-lists originally reported in [1], ranging from 575395 to 1428437 adjacency connections. Note that we selected this set of benchmark net-lists due to their large size to better evaluate the run-time scalability of our design. For each netlist, we ran ten trials, each with a different seed for the random-number generator. Next, we ran ten trials using our *SPA* for each net-list in the same set of benchmarks, on each of the following NVIDIA GPU cards: a) GeForce8800GT, b) GT430, and c) GTX480. Note that we ran all experiments using on an Intel Core i7-930 2.8GHz processor running 64-bit Ubuntu 12.04.3. All host code was compiled using `g++` with the flags `-O3 -msse2`. All CUDA code was compiled using `nvcc` version 5.5.0.

A. Run-time

Table I shows the mean placement run-times in seconds, using VPR with `inner_num=1`, and our SPA running on three different GPU cards (with `inner_num=1`) , across the ten trials for each of the 7 IWLS-based net-lists. By observing the summed total run-time for each placer configuration in the last row, we can see that the SPA is significantly faster than VPR when run on any of the GPU cards. Specifically, on the highest performing card tested, the GTX480, speed-ups range from 16.3× to 27.8×, with a mean speed-up of 19×. However, it is important to note that, as shown in the plot in Table Ib, there is a strong positive correlation between the observed run-time speed-ups and the adjacency-list connection count. Note that the connection-count scales directly with the size of the net-list to be placed. This indicates that as the problem size grows, our approach exhibits a larger speed-up, suggesting that none of the benchmarks used in our experiments were large enough to fully utilize any of the GPUs we tested with. Further more, the difference in slope between the speed-ups observed for the different GPU cards correlates with the peak floating-point performance of each card. This implies that even greater speed-ups should be attainable by simply running our code on a more capable GPU, such as the recently announced NVIDIA TITAN-Z, which promises nearly four times the peak floating-point performance of the GTX480.

B. Post-routing wire-length and critical-path-delay

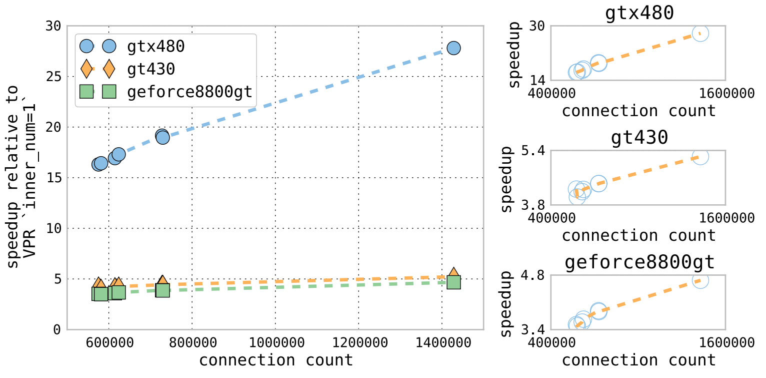
After performing placement under each of the configurations described above, we routed each placement using VPR's router. The resulting routed critical-path-delays and wire-lengths for VPR and our proposed SPA are listed in Table IIa, where μ_{wl} represents the mean wire-length (in segments), and μ_{cpd} represents the mean critical-path-delay (in nanoseconds), for each placer configuration across 10 trials for each net-list. Note that the quality-of-result for SPA does not depend on the number of processing cores, and therefore does not change based on the GPU card used. The last two columns in Table IIa show the percentage difference in the resulting wire-length and critical-path-delay from VPR versus our parallel annealer, denoted as Δ_{wl} and Δ_{cpd}, respectively. Note that a negative value in either the Δ_{wl} column or the Δ_{cpd} column indicates that SPA improved upon the corresponding metric from VPR. The box-plots in Table IIb, show head-to-head comparisons between the wire-length and critical-path-delay results from SPA and VPR. From the plots, we see that our SPA improves the critical-path-delay and wire-length compared to VPR with statistical significance *(determined using Wilcoxon signed-rank test with $\alpha = 0.05$)* for 3 out of 7 net-lists, with an overall improvement of 5.1% in wire-length and 4.8% in critical-path-delay. These results clearly show that our SPA is able to effectively explore the placement solution space, despite the soft-conflicts introduced by evaluating swaps in concurrent sets and any bias introduced by the regularity of the applied displacement-patterns. Furthermore, it is evident that despite the highly parallel nature of our approach, quality is maintained, and even improved in several cases compared to VPR.

VI. Conclusions and Future Work

In this paper, we proposed a novel methodology for performing parallel FPGA placement utilizing structured parallel primitives. Our approach utilizes a general hyper-graph cost model that may be easily extended to include other objectives and even to other problem domains. Moreover, we have presented a method of generating very large sets of non-overlapping swaps where the number of swaps scales with the size of the target FPGA architecture. Using our proposed hyper-graph cost model and parallel swap generation method, we have designed a simulated annealing variant, which we call *SPA*, that evaluates and applies the generated sets of swaps using only parallel programming primitives. The SPA design presented in this paper can be implemented using any parallel framework (e.g., Cilk Plus, Thrust, etc.) which provides implementations of the structured parallel patterns

Net-list	**SPA** GeForce8800GT	GT430	GTX480	**VPR** CPU
uoft_raytracer	1003	829	217	3538
b17_1_new	820	711	174	2869
leon3mp	1243	1066	263	4468
netcard	1172	1010	248	4300
b18_new	1070	939	217	4157
b18_1_new	1072	935	218	4142
b19_1_new	2468	2206	414	11517
Sum	8851	7699	1753	34994

(a) Placement run-time (seconds)

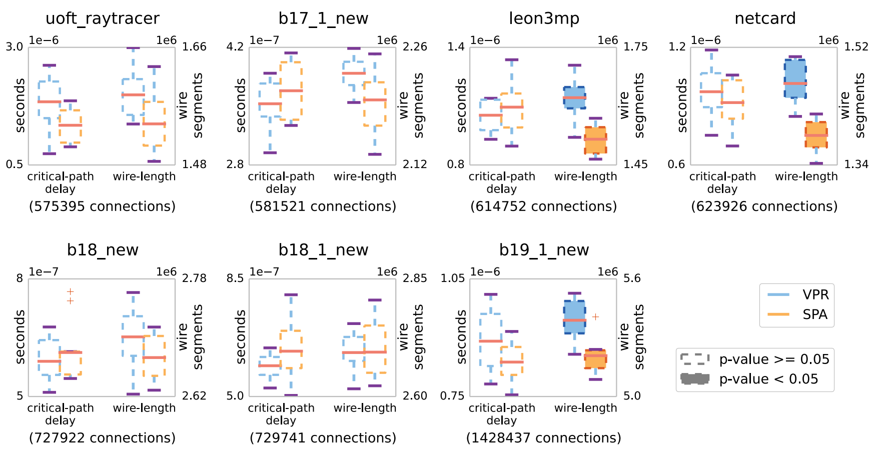


(b) Speedup of SPA running on NVIDIA GeForce8800GT, GT430, and GTX480, relative to VPR (`inner_num=1`).

TABLE I: Placement run-time comparison between *Structured Parallel Annealer (SPA)* and VPR (`inner_num=1`).

Netlist	**VPR** u_{wl}	u_{cpd}	**SPA** u_{wl}	u_{cpd}	**% diff** Δ_{wl}	Δ_{cpd}
uoft_raytracer	1.6M	1.8u	1.5M	1.3u	-2.8	-27.1
b17_1_new	2.2M	352.9n	2.2M	368.3n	-1.4	4.4
leon3mp	1.6M	1.1u	1.5M	1.1u	-6.5	3.9
netcard	1.5M	973.6n	1.4M	918.2n	-5.4	-5.7
b18_new	2.7M	589.8n	2.7M	612.3n	-1.0	3.8
b18_1_new	2.7M	591.6n	2.7M	634.9n	33.5m	7.3
b19_1_new	5.4M	891.1n	5.2M	837.8n	-3.4	-6.0
Sum	17.7M	6.3u	17.2M	5.8u	-	-
Mean	-	-	-	-	-5.1	-4.8

(a)



(b)

TABLE II: Post-routing critical-path and wire-length: *Structured Parallel Annealer (SPA)* versus VPR (`inner_num=1`).

used. To demonstrate this, we developed a Thrust-based implementation of SPA and compared the placement run-time and solution quality of SPA running on three GPU cards of varying performance with the results from running VPR with *`inner_num=1`*. Our experimental results show that not only does our speed-up increase with core count, *(e.g., when moving from the GT430 to the GTX480)*, but *also* increases along with problem size. This indicates that our design exhibits strong run-time scalability, unlike [1], [6], [2]. Moreover, our annealer achieves greatly reduced run-times without sacrificing quality, unlike in [6], [2], where the quality-of-result degrades as the number of processing cores increases. In fact, as our results show, our annealer is capable of out-performing VPR in terms of post-routing wire-length and critical-path-delay on average.

In future work, we plan to extend the base SPA design with several additions. Since our current implementation is not yet saturating the peak computational throughput of the GPU cards we tested with, we plan to develop population-based approaches where we process several placements simultaneously in an attempt to either improve quality or arrive at a superior result in fewer iterations. Moreover, we plan to incorporate timing by adding per-edge and, potentially, per-connection weights to the connection cost calculations.

References

[1] A. Choong, R. Beidas, and J. Zhu. Parallelizing simulated annealing-based placement using gpgpu. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 31–34. IEEE, 2010.

[2] J. B. Goeders, G. G. Lemieux, and S. J. Wilton. Deterministic timing-driven parallel placement by simulated annealing using half-box window decomposition. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 41–48. IEEE, 2011.

[3] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, and J. Rose. VPR 5.0: FPGA cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 133–142, Monterey, California, USA, 2009. ACM.

[4] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.

[5] K. Shahookar and P. Mazumder. VLSI cell placement techniques. *ACM Comput. Surv.*, 23(2):143–220, 1991.

[6] C. C. Wang and G. G. F. Lemieux. Scalable and deterministic timing-driven parallel placement for FPGAs. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 153–162, New York, NY, USA, 2011. ACM. ACM ID: 1950445.

[7] M. Xu, G. Gréwal, and S. Areibi. Starplace: A new analytic method for FPGA placement. *Integration, the VLSI Journal*, 44(3):192–204, 2011.