# Parallel Simulated Annealing for FPGA Placement Using VTR Software

Tan Nguyen and Ting Cao

May 2018

## 1 Introduction

With the end of the Dennard scaling and Moore's law, the computing industry is gradually shifting gear to favor domain-specific architecture for application acceleration instead of relying on general-purpose processors. Among the accelerator solutions, FPGA offers a competitive solution for speeding up a wide variety of workloads due to its reconfigurability with attractive energy efficiency in comparison to multi-core CPU or GPU. FPGA is a massively parallelism hardware device that contains an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects that wires the blocks together. We are witnessing a huge adoption of FPGAs in the industry lately (Intel, Microsoft, Amazon, Baidu, etc.). Nonetheless, FPGA still remains exclusive to hardware engineers with expertise in hardware design and programming, since a productive programing language for general users has been lacking so far.

In addition to the programming issue, the obstacle that makes FPGAs less appealing to software engineers is the time-consuming FPGA CAD compilation process. The ever growing size of FPGAs has made it demanding for users to map huge and complex applications onto the platforms. As a result, the compilation issue could get worse. This is a stark contrast with the software compilation flow, and thus might impede the productivity and massive adoption of FPGAs. The CAD flow typically consists of three steps: logic synthesis, placement, and routing, among which placement usually requires the largest amount of runtime. In this work, we propose an implementation of a parallel placement algorithm in the CAD flow to accelerate the FPGA placement itself and in turn, the whole compilation process.

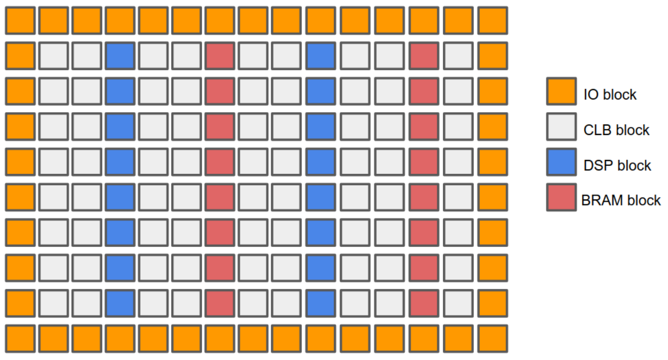


Figure 1: An example of a FPGA architecture

Figure 1 shows one example of how an FPGA looks like. In general, an FPGA is composed of many physical blocks of types: IO, CLB (look-up tables and registers), DSP (multipliers), and BRAM (memories) organized in a rectangular 2D grid. The IO blocks form a boundary of an FPGA. The DSP and BRAM blocks arrange into columns. The rest is filled with CLB blocks.

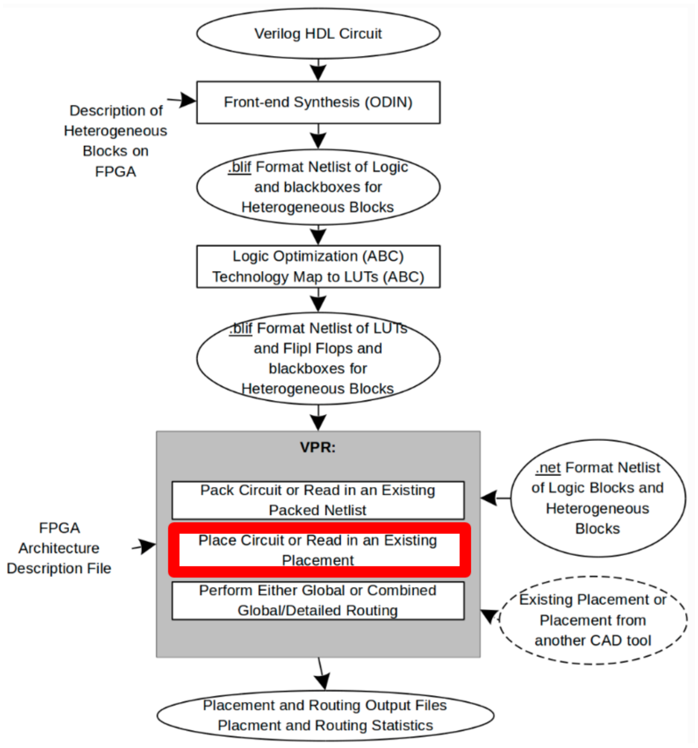

Verilog HDL Circuit
Front-end Synthesis (ODIN)
Description of Heterogeneous Blocks on FPGA
.blif Format Netlist of Logic and blackboxes for Heterogeneous Blocks
Logic Optimization (ABC) Technology Map to LUTs (ABC)
.blif Format Netlist of LUTs and Flipl Flops and blackboxes for Heterogeneous Blocks
VPR:
Pack Circuit or Read in an Existing Packed Netlist
Place Circuit or Read in an Existing Placement
Perform Either Global or Combined Global/Detailed Routing
FPGA Architecture Description File
.net Format Netlist of Logic Blocks and Heterogeneous Blocks
Existing Placement or Placement from another CAD tool
Placement and Routing Output Files Placment and Routing Statistics

Figure 2: VTR CAD workflow, the placement step relevant to the present work is highlighted by a red box

We pick the VTR (Verilog-To-Routing) tool [Luu+14] as a codebase for our project. VTR is a state-of-the-art academic CAD flow which source code is freely available. In this project, we focus on parallelizing the placement component of the flow, which belongs to a sub-tool called VPR (Versatile Placement & Routing). Fundamentally, the placement maps FPGA netlists (composed of many logic blocks implementing user's hardware circuit) onto a grid of FPGA physical blocks such that the total wirelength or critical path delay is minimized to which we regard as a cost function,

$$Cost = \sum_{n=1}^{N_{nets}} q(n)\left[\frac{bb_x(n)}{C_{av,x}(n)} + \frac{bb_y(n)}{C_{av,y}(n)}\right] + Cost_{criticalpath},$$

where $q(n)$ is compensation factor depending on the number of terminals of the net $n$, $C_{av}$ the channel capacity in $x$ and $y$ directions over the bounding box of net $n$, $bb_x(n)$ and $bb_y(n)$ the length of the bounding box of net $n$ along the $x$ and $y$ directions, respectively. The bounding box of the net $n$ is the smallest rectangle that encloses all the logic blocks belonging to the net $n$. $Cost_{criticalpath}$ refers to the cost of the longest path that connects two related nets. The global minimum of the cost function therefore can be qualitatively perceived as a configuration that places each net's logic blocks as close as possible, but meanwhile avoids long-distance connections between two related nets.

For the placement, VPR uses a simulated annealing algorithm which employs a probabilistic technique to find a approximate global optimum of the cost function. For this purpose, the VPR placer randomly generates an initial mapping, and then tries to move blocks heuristically while evaluating the cost function to determine if a move is legal and cost-effective. A move or swap of a block is randomly initiated, and is accepted if

$$e^{-\frac{\Delta_{Cost}}{T}} > R(0,1),$$

where $\Delta_{Cost}$ is the change of the cost function following each move, and $T$ the temperature which equals to a large value at the beginning of the annealing and gradually decreases in the ensuing steps. $R(0,1)$ generates a random number between 0 and 1. The key of the simulated annealing is that, due to this acceptance criterion (involving temperature and random number), a moves has a certain probability of being accepted even though the cost becomes higher. This guarantees that the algorithm could traverse a large fraction of the configurable space without being trapped by local minima. However, the process of random number generation and the nature of moving blocks in the simulated annealing are prone to racing conditions, a issue that we will solve in this project.

The simulated annealing for the placement problem usually involves millions of moves on the logic blocks of a modern FPGA. As a result, an efficient parallelization scheme must be able to execute multiple moves simultaneously by multiple threads. A natural way to achieve this goal is to partition the FPGA logic blocks into domains (spatial decomposition), with one thread holding one domain. In principle, since the cost function is linear with the perimeter of the bounding box and additive between different domains, this method can ideally give a result of the same quality as the serial algorithm.

## 2 Previous work

The acceleration of FPGA placement is a well-explored topic with numerous publications. For example, [WD03] proposed a specialized hardware solution to perform placement rather than using the software flow. This resulted in an orders-of-magnitude speedup but with a 35% wirelength degradation. [ASB14] parallelized FPGA placement using OpenMP; they avoided move conflicts by letting a master thread doing a move proposal, thus became the bottleneck of the implementation. There were also works that used GPUs to accelerate the FPGA placement [CBZ10], [Fob+12], or proposed different algorithms for placement other than the simulated annealing such as analytic annealing or a genetic algorithm [Yan+05].

[WL11] used domain decomposition to parallel the placement process. They achieved 125x speedup with around 11% and 8% degradation in term of bounding box and critical path, respectively on VTR 5.0. The follow-up work [GLW11] improved upon the existing results with a different domain decomposition technique. As we also chose the domain decomposition approach, these works serve as inspiration and reference sources for our implementation. However, our strategy of domain decomposition is unique from the previous works and will be discussed in the subsequent section.

## 3 Implementation

### 3.1 Original implementation

The original implementation of the VPR placement can be briefly described as follows.

```
temp = initial_temperature();
random_placement();
while (exit(temp, cost) == 0) {
  timing_analysis();
  for i = 0 until move_limits {
    result = try_swap();
    update_statistics(result);
    if timing_updated == true
      timing_analysis();
  }
  update_statistics();
  update_cost(&cost);
  update_temperature(&temp);
}
```

A legal placement is randomly generated initially. A placement is considered “legal” when a logic block is mapped to a correct FPGA resource type (CLB, memory, multiplier, or IO); a type of a logic block has been determined through the packing stage. The simulated annealing progresses until the temperature reaches an exit condition. For each annealing iteration, VPR placement performs multiple swap attempts. A swap trial is evaluated based on its impact on the bounding box cost and timing-delay cost. The cost is computed incrementally per swap instead of recomputing for the whole placement for performance reason. VPR placement updates the statistics for the swapping process to calculate the success rate: the number of accepted swaps over the total number of attempted swaps. The success rate is used to update the temperature for the simulated annealing. When the success rate is high (i.e. many blocks are being swapped to improve the overall cost), the temperature decreases significantly. The temperature cools down if the success rate plummets to which it signifies that further moves might not improve the cost anymore.

The VPR placement also performs a timing analysis before the swapping loop and once for a while inside the swapping loop. This is important as to improve the timing-delay cost, which has an impact on the final critical path result. Nonetheless, timing analysis is a lengthy process as we will see in the result section – this is a reason why it is not regularly being executed inside the swapping loop.

## 3.2 Domain decomposition

Our approach is to spatially decompose the FPGA grid into multiple domains where each thread will perform swaps on each domain. To avoid move conflict, the domains do not overlap. However, to improve the QoR and the likelihood of discovering the global minima, we must ensure that a block has a chance to move out of its originally assigned domain. Thus, our placement process consists of two phases. The first phase decomposes the FPGA horizontally, while the second phase decomposes it vertically. This decomposition strategy enables blocks to move across the FPGA. In addition, the strategy also works well with heterogeneous physical blocks on the FPGA: memory and multiplier blocks. Unlike CLBs, those blocks do not span across the FPGA, but rather they are organized into columns. With the proposed decomposition scheme, a block will be able to move within one column (vertical phase) or “jump” to a different column (horizontal phase), thus it effectively enriches move exploration. Figure 3 demonstrates the placement phases and how we assign threads for each phase. The red logical block is able to move diagonally over the FPGA as the phases progress.

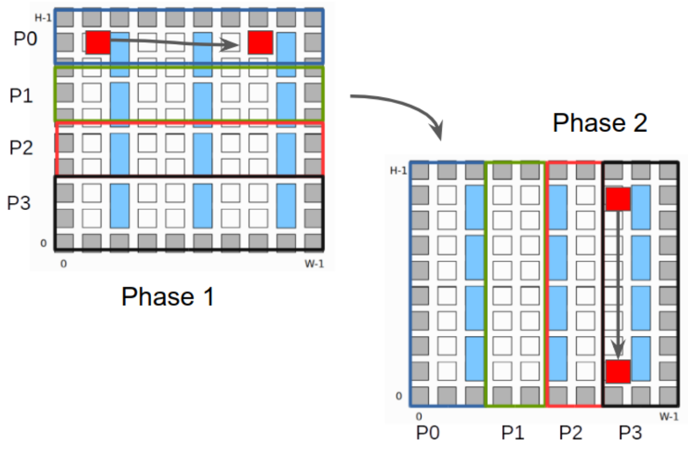


Figure 3: Placement phases and the domain decomposition for each phase

One could also imagine a different decomposition strategy that meets the criterion above. Yet due to time constraint, we only experiment with the aforementioned scheme. We think that a reasonable decomposition strategy is likely to improve the Quality-of-Result.

## 3.3 Data structure privatization & Algorithmic description

The original VPR placement implementation is a complex piece of software that heavily makes use of global variables. This poses a challenge for us at first when we try to parallel the code since we must ensure that our parallel implementation does not introduce any race condition bug. We make some modification to the code to localize several global variables. In particular, each thread will have a separate random seed to initiate moves so that a move of one thread does not affect the others. The pseudo thread-level code for our parallel algorithm is described as follows. Note that the variable **num_decompose_phases** has a value of 2 since we perform horizontal decomposition followed by vertical decomposition as mentioned in the previous subsection.

```
while (exit_crit(temp, cost) == 0) {
  barrier();
  if (master_thread) {
    timing_analysis();
  }
  barrier();
  for i = 0 until num_moves {
    for j = 0 until num_decompose_phases {
      global_to_local_update();
      for k = 0 until num_trials {
        for m = 0 until (nx + 1) * (ny + 1) / (num_divs * OMP_NUM_THREADS) {
          try_swap();
          update_statistics();
        }
        barrier();
        compute_bb_cost();
        compute_td_cost();
        barrier();
      }
    }
  }
  if (master_thread) {
    update_statistics();
    update_temperature(&temp);
  }
  barrier();
}
```

Although blocks are moved within a current assigned domain, race bugs could occur when there is an attempt to move blocks that are close to the domain boundary. This is because blocks are connected via nets. A net may connect two or more blocks – one source block and multiple sink blocks. Nets are important for computing the costs for the evaluation of a swap attempt, since we need to identify the bounding box of a net. For each move, the VPR placement will examine all affected nets regarding the moved blocks, and update the cost associated with it. Therefore, there might be a case when a source block of a net is moved by one thread, and one of the sink blocks is moved by another thread. This likely leads to a race condition when computing the cost of moving for that net.

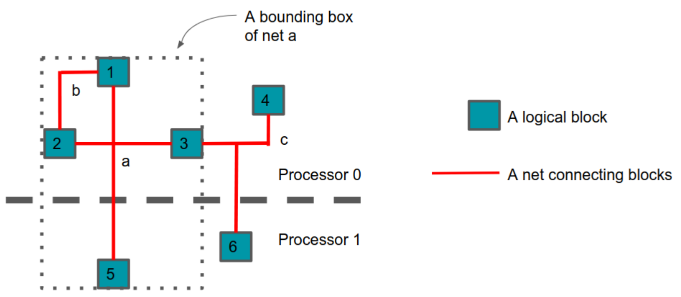


Figure 4: Blocks and nets in FPGA placement problem.

Figure 4 illustrates the relationship between blocks and nets. As an example, net a wires block 1, 2, 3, and 5 together; block 1, 2, 3, and 4 are managed by processor 0, while block 5 and 6 are managed by processor 1. There is a global array of nets which keep track of the current cost of each net.

To solve this problem, we privatize the cost arrays for each thread. A thread will have a separate cost arrays to keep track of the moves within its domain (each element of the array keeps track of the current cost of a net). For nets and blocks outside of a domain, a thread will consider as if they were not moved. In other words, threads use the old information of blocks outside of their domains. This alleviates the race condition problem with the downside of increasing memory footprint. However, there is a risk of excessive use of stale information when computing the costs. To fix this problem, we break the swapping procedure into shorter processes. After each process is done, we recompute the bounding cost as well as the timing-delay cost. The objective is to not let a thread relying on stale information for too long.

One might also notice that we did not do timing analysis within the swapping procedure as it is a very expensive operation. For this work, we do not focus on parallelizing the timing analysis process, hence the master thread is in charged of doing this work solely. Thus, instead of doing timing analysis, we recompute the cost functions more frequently in hope that it can compensate for the lack of timing analysis. The master thread also updates the temperature based on the success rate of swapping blocks collected from all the threads. The temperature and the overall cost are the global variables shared by all the threads. To ensure that there is no race condition occurred when modifying these and other global variables, as well as preventing threads from modifying the same blocks when switching phases, we make use of barriers. At the beginning of each phase, we broadcast the information from global data structures to the local copies of each thread.

We made our change directly to the VPR placement source code. We use OpenMP for our parallel implementation. Our implementation does not require any sophisticated technique other than workload distribution among threads and thread synchronization. Hence, we believe that our implementation does not tie to any specific thread-level parallelism API and other techniques, such as Pthread, MPI, or PGAS, can also be used.

## 3.4 Annealing parameters tuning

Annealing parameters concern the rate of which the annealing process updates the temperature according to the swapping success rate. Generally, the temperature decreases dramatically if the success rate is high, and slowly cools down when the success rate is low. The annealing parameters have a substantial impact on the performance of our parallel implementation as it determines when the annealing process terminates. We found that the original annealing parameters did not work well for us as the temperature went down very slowly, thus led to more annealing iterations. We modified the annealing parameters to adapt to our implementation. As we are able to explore a large combination of moves using multiple threads in parallel, we think that it makes sense to reduce the temperature faster than the original implementation did.

Table 1: The table demonstrates the value of the temperature coefficient based on the success rate for the original VPR placement and our placement. Note that new_temperature = current_temperature × temp_coeff

| Success rate (range) | Original VPR placement (temp_coeff) | Our parallel placement (temp_coeff) |
|---|---|---|
| 0.96 - 1 | 0.5 | 0.5 |
| 0.8 - 0.96 | 0.9 | 0.7 |
| 0.15 - 0.8 | 0.95 | 0.8 |
| 0 - 0.15 | 0.8 | 0.7 |

Furthermore, to improve the quality of result, we introduce the parameter **num_moves**. This parameter determines the amount of repetition of the swapping procedure. Small **num_moves** tends to produce low QoR, whereas big **num_moves** leads to longer annealing process while modestly improves the QoR. Therefore, choosing the right value for **num_moves** for tradeoff between runtime and QoR is very essential. We adapt **num_moves** based on the success rate. At a high success rate (above 0.7), we set **num_moves** to 2 and increases it to 6 when the success rate is low (below 0.7). The rationale behind this decision is that we are able to explore lots of moves at a high success rate, hence **num_moves** should be small to minimize the impact on the overall runtime, whereas **num_moves** becomes bigger at a low success rate to increase the opportunity of discovering new moves that improve the QoR.

Apart from **num_moves**, we also have other tuning parameters such as **num_trials** and **num_divs**. These parameters control how frequent should we recompute the cost values or update block information so that threads do not rely on stale information for a long time. They also have impact on the runtime and QoR. Our experiments have shown that a value of 8 for **num_trials** and 2 for **num_divs** work the best for our choice of the annealing parameters and **num_moves**.

## 4 Result & Discussion

To evaluate the performance of our parallel implementation, we pick 5 benchmarks from the existing VTR software package (**vtr_benchmarks_blif** suite). Those benchmarks are chosen because of their sizes and the original placement runtimes. We did not conduct experiment on small benchmarks as we think they might not be worthwhile parallelizing or provide interesting data points when scaling to a large number of threads, yet it is still very important to have a comprehensive evaluation. This is subject to future work. Table 2 shows the size and placement runtime of each benchmark that we chose.

Table 2: VTR Benchmarks.

| Benchmark | Size (CLB x CLB) | Placement runtime (second) |
|---|---|---|
| LU32PEEng | 98 x 98 | 357.759 |
| LU8PEEng | 53 x 53 | 60.1726 |
| bgm | 63 x 63 | 91.3206 |
| mcml | 95 x 95 | 296.216 |
| stereovision2 | 86 x 86 | 67.7757 |

Here the size (in term of CLB x CLB) refers to the minimum array of FPGA physical blocks that can fit a corresponding benchmark. The target FPGA device is **k6_frac_N10_mem32K_40nm**. It is a synthetic device developed for FPGA architectural and CAD research.

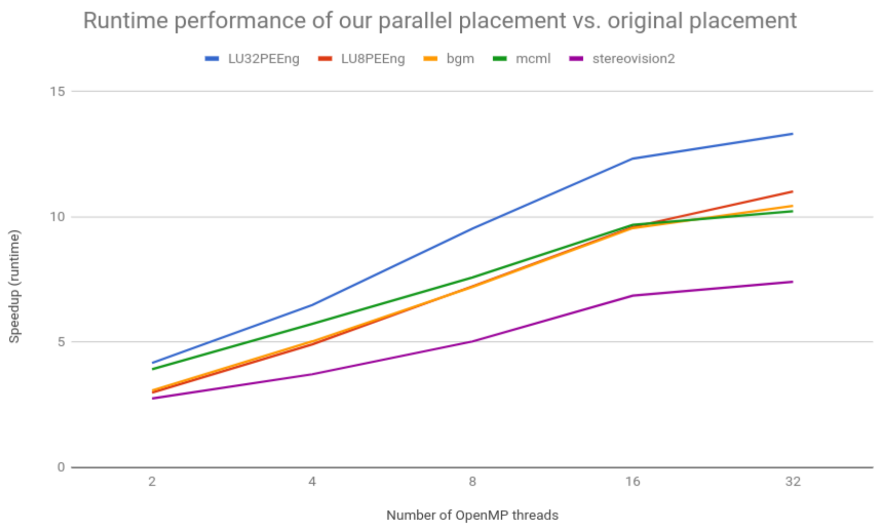
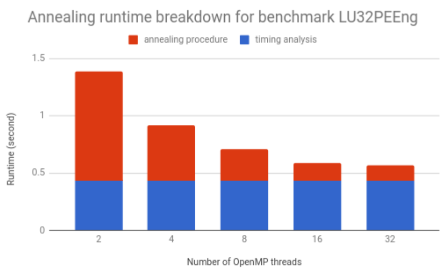


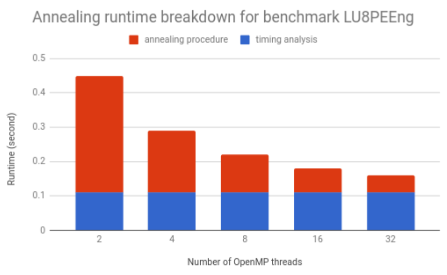
Figure 5: Performance comparison between the original VPR placement and our parallel implementation.

Figure 5 demonstrates the speedup that our parallel implementation achieves over the original VPR placement. With 2 threads, our implementation is 2.7-4.2x faster than the original version. This speedup can be explained as our annealing parameters are different from the original one, thus our annealing process may terminate faster (and also the swapping procedure is not equivalent). Our performance starts to flatten with more than 16 threads. The best speedup is 13.3x with 32 threads for benchmark LU32PEEng. Among the benchmarks, stereovision2 achieves the lowest speedup curve. As we break down the performance of all the components of our implementation per annealing iteration as seen from Fig 6, we found that the timing analysis is the actual bottleneck in our design. From the figures, we can observe that the runtime of the timing analysis actually does not change regardless to the number of threads being used. This is valid as we do not attempt to parallelize this procedure for our work. The overall speedup is obtained when the compute time is more dominant than the timing analysis portion. Here the compute time refers to the time doing the whole swapping procedure per thread for each annealing iteration. However, as more threads are utilized, the compute time goes down, and thus the timing analysis becomes the main factor for the poor performance gain. This explains why we could not achieve a linear speedup for our tested benchmarks. When closely evaluating the compute time, it tends to flatten at higher number of threads. We suspect that this is due to the synchronization that we introduce to prevent race condition bugs from happening, or the imbalance of workload distribution which leads to some threads having to doing swaps on a (slightly) larger regions than the others.

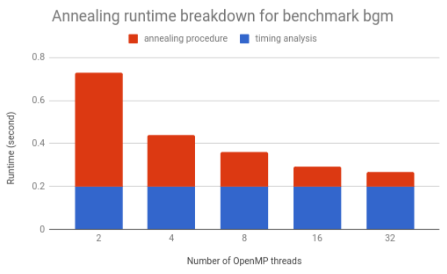
Another equally important metric for evaluation is the Quality-of Result in term of bounding box cost (wirelength) and timing-delay cost (critical path delay). Figure 7 and 8 demonstrate the degradation of bounding box cost and critical path delay caused by our parallel implementation. As can be seen from the figures, our implementation does not introduce any substantial degradation. We believe that our degradation is within the acceptable range and comparable to previous publications. Furthermore, our implementation even produces better hardware results in some cases (referring to the ones which have negative degradation percentage). We attribute this to the capability of exploring a large possible number of moves in parallel. Note that the original VPR placement does not exhaustively explore all possible moves; it heuristically sets a move limiter for exploration per annealing iteration. Thus, it is likely that the original placement can miss better solutions.



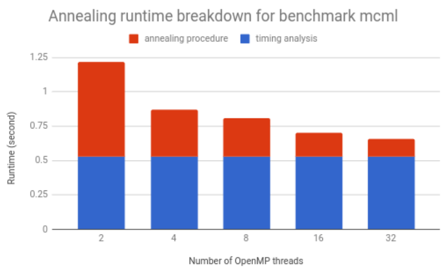
(a) LU32PEEng



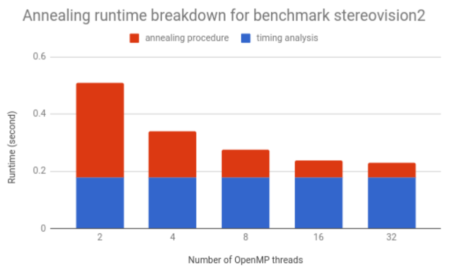
(b) LU8PEEng



(c) bgm



(d) mcml



(e) stereovision2

Figure 6: Annealing runtime breakdown for VTR benchmarks

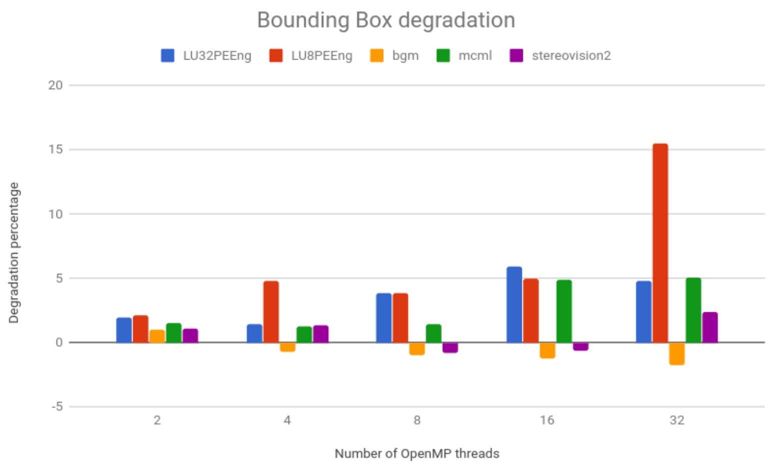


Figure 7: Bounding box Degradation.

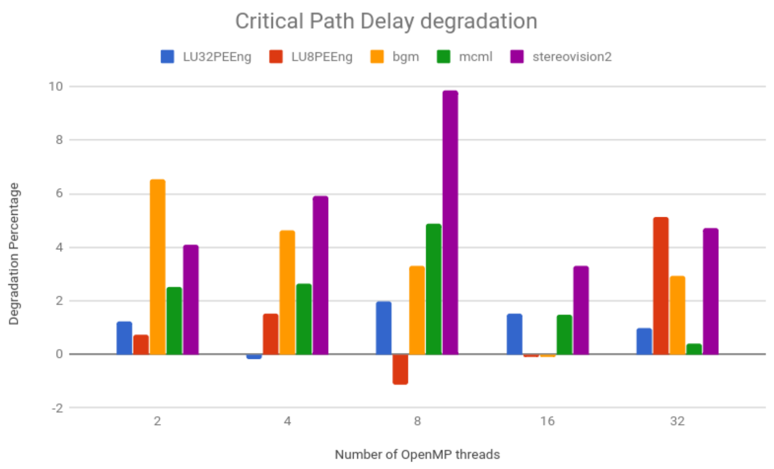


Figure 8: Critical path delay Degradation.

By looking at the QoR figures, we speculate that there is hardly a recognizable correlation between the number of threads and the QoR degradation. For example, some data points perform better at small number of threads, yet become worse as we increase the thread count, and the behaviour might be different between bounding box case and critical path delay case. When collecting data, we actually tried to tune our annealing parameters to achieve decent critical path results in the case of 16 threads, and used the same set of parameters to gather data for other thread counts. Therefore, we think that each thread count requires a different tuning of annealing parameters to achieve good QoR. Further study needs to be conducted to better understand the relationship between the level of parallelism, the overall runtime, and the achieved QoR.

## 5 Conclusion & Future work

In this work, we focus on parallelizing the FPGA simulated annealing placement problem. We achieve our goal by using a two-phase domain decomposition where each thread is assigned a non-overlapped region to perform block movements in parallel. We also tune the annealing parameters to reduce the runtime as well as improve the Quality-of-Result. The results have shown that our parallel version is around 7.4-13.4x faster the the original version at 32 threads while introduces minimal degradation in wirelength (within 10% on average) and critical path delay (within 5% on average).

As dissected in the previous section, the bottleneck of our implementation is the timing analysis part which we currently use only the master thread to run it. It is yet another complex function which we did not have time for exploration. One could also try to run this function less frequently in the annealing process with the trade-off of QoR. This is also a possible direction to investigate.

A different direction is to use hardware accelerator to speed up the placement problem. Initially, we tried using the KNL processors for our parallel code. However, the performance was disappointing. It is challenging to achieve a good performance on KNL unless the code is fine-grained parallelized. For our implementation, since the workload of each thread is very bulky (proposing and evaluating numerous moves), KNL architecture would not be a good fit to this problem. We speculate the same conclusion for GPU.

To really make an innovative advancement to the placement and also CAD problem in general, we think that it is equally important to understand how various cost functions and related parameters are derived. For our implementation, we just use them as-is, and identify (and modify) the parallelizable regions. However, the behaviour of certain cost functions may change when the problem is parallelized. Having a thorough understanding of all the aspects in the placement algorithm is necessary to achieve good speedup with low to zero QoR degradation.

## References

[WD03] Michael G. Wrighton and André M. DeHon. “Hardware-assisted Simulated Annealing with Application for Fast FPGA Placement”. In: *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays.* FPGA ’03. Monterey, California, USA: ACM, 2003, pp. 33–42. ISBN: 1-58113-651-X. DOI: 10.1145/611817.611824. URL: http://doi.acm.org/10.1145/611817.611824.

[Yan+05] M. Yang et al. “FPGA placement using genetic algorithm with simulated annealing”. In: *2005 6th International Conference on ASIC.* Vol. 2. Oct. 2005, pp. 806–810. DOI: 10.1109/ICASIC.2005.1611450.

[CBZ10] A. Choong, R. Beidas, and J. Zhu. “Parallelizing Simulated Annealing-Based Placement Using GPGPU”. In: *2010 International Conference on Field Programmable Logic and Applications.* Aug. 2010, pp. 31–34. DOI: 10.1109/FPL.2010.17.

[GLW11] J. B. Goeders, G. G. F. Lemieux, and S. J. E. Wilton. “Deterministic Timing-Driven Parallel Placement by Simulated Annealing Using Half-Box Window Decomposition”. In: *2011 International Conference on Reconfigurable Computing and FPGAs.* Nov. 2011, pp. 41–48. DOI: 10.1109/ReConFig.2011.27.

[WL11] Chris C. Wang and Guy 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. Monterey, CA, USA: ACM, 2011, pp. 153–162. ISBN: 978-1-4503-0554-9. DOI: 10.1145/1950413.1950445. URL: http://doi.acm.org/10.1145/1950413.1950445.

[Fob+12] Christian Fobel et al. “GPU Approach to FPGA placement based on star+”. In: *10th IEEE International NEWCAS Conference* (2012), pp. 229–232.

[ASB14] M. An, J. G. Steffan, and V. Betz. “Speeding Up FPGA Placement: Parallel Algorithms and Methods”. In: *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines.* May 2014, pp. 178–185. DOI: 10.1109/FCCM.2014.60.

[Luu+14] Jason Luu et al. “VTR 7.0: Next Generation Architecture and CAD System for FPGAs”. In: *ACM Trans. Reconfigurable Technol. Syst.* 7.2 (July 2014), 6:1–6:30. ISSN: 1936-7406. DOI: 10.1145/2617593. URL: http://doi.acm.org/10.1145/2617593.