Parallelizing Simulated Annealing-Based Placement using GPGPU

Alexander Choong
Department of Electrical and Computer Engineering
University of Toronto
Toronto, ON, M5S 3G4, Canada
Email: achoong@eecg.toronto.edu

Rami Beidas
Department of Electrical and Computer Engineering
University of Toronto
Toronto, ON, M5S 3G4, Canada
Email: rbeidas@eecg.toronto.edu

Jianwen Zhu
Department of Electrical and Computer Engineering
University of Toronto
Toronto, ON, M5S 3G4, Canada
Email: jzhu@eecg.toronto.edu

***Abstract*—Simulated annealing has became the de facto standard for FPGA placement engines since it provides high quality solutions and is robust under a wide range of objective functions. However, this method will soon become prohibitive due to its sequential nature and since the performance of single-core processor has stagnated.**

General purpose computing on graphics processing units (GPGPU) offers a promising solution to improve runtime with only commodity hardware. In this work, we develop a highly parallel approach to simulated annealing-based placement using GPGPU. We identify the challenges posed by the GPU architecture and describe effective solutions. An average speedup of about 10x was achieved over conventional placement within 3% of wirelength.

I. Introduction

Over the past four decades, the number of transistors in an integrated circuit (IC) has doubled approximately every two years. This trend, known as Moore's Law [1], has been faithfully followed by the field programmable gate array (FPGA) industry, and is increasingly evident as the industry moves toward complete systems on chip (SoC) and high performance computing (HPC). Despite these exciting developments, the exponential growth in device capacity has outpaced computer-aided design (CAD) software, especially as single-processor performance has begun to stagnate.

One of the most computationally intensive stages of FPGA synthesis is placement, where simulated annealing is widely accepted due to superior quality of results and robustness [2], [3]. As with other synthesis stages, the gap is widening between the required computational work and the available processing power. Therefore, there is a pressing need to improve the performance of simulated annealing-based placement [4], [5].

A promising solution to this conundrum is general purpose computing on graphics processing units (GPGPU) since applications from many scientific and computing domains have been successfully accelerated by one or two orders of magnitude[6]. Graphics processing unit (GPU) performance growth has historically followed an exponential trend, and a highly parallel solution could continue to scale with growing FPGA designs.

TABLE I
Classification of Previous Parallel Simulated Annealing Approaches

	Prevent Errors	Allow Errors
Data Parallelism	[4] [8]	[9] [10] [11]
Task Parallelism	[4]	
Both	[5] [7]	OURS

Our contribution is a parallel implementation of simulated annealing-based placement on GPGPU which, to our knowledge, is the first of its kind. We overcome the architectural challenges, and achieve about an order of magnitude speedup without significantly sacrificing quality of results.

The remainder of the paper is organized as follows. Section II reviews previous parallelization efforts. In Section III, we estimate the optimistic speedup for a naïve implementation and discuss how the GPU architecture hampers the realization of this speedup. Section IV discusses our proposed solution. Results are presented and discussed in Section V while Section VI concludes our work.

II. Related Work

The previous methods of parallelizing simulated annealing placement can be classified using two different criteria, namely *error tolerance* and *parallelism domain*:

- Errors arise when parallel computations involve data-dependencies. Error tolerance describes whether errors are prevented (e.g. by using strict synchronization schemes) or are allowed which creates opportunities to improve performance.
- Parallelism domain specifies the type of parallelism exploited. The first type is *task parallel* in which the different stages of simulated annealing are assigned to different processing units, and this is often referred to as *task decomposition* in the literature. The second type is *data parallel*. An example is when multiple moves are made in parallel which is referred to as *parallel moves* in the literature. Since both types of parallelism are independent, so it is possible to utilize both. More details can be found in [7].

Past efforts reviewed below are classified in Table I.

1946-1488/10 $26.00 © 2010 IEEE
DOI 10.1109/FPL.2010.17

One of the earliest efforts to parallelize simulated annealing-based placement was [7], where the authors used task decomposition and parallel moves. To prevent errors, only moves without data dependencies could be evaluated in parallel which restricts the available parallelism. A speedup of 2x was achieved using three processors, but not much additional performance was gained with a forth.

Another approach was implemented on a cluster of machines and used message passing to synchronize the data across all processors. Clearly, this is not scalable: as the number of machines increases, the communication overhead increases quadratically. A speedup of 3.8x on four cores and 5.3x on 8 cores was achieved [9].

A speculative implementation of simulated annealing was reported in [8] and a speedup of 3.25 on 8 processors was achieved. The authors found that their theoretical speedup was $P/\log_2 P$ where P is the number of processors, which is not linearly scalable.

The authors of [4] used commodity multicore processors to accelerate simulated annealing. They implemented two different approaches: i) task decomposition which inherently has limited parallel and they achieve a speedup of 1.3x on two cores, and ii) parallel moves which achieved 2.2x speedup on four cores. In the latter approach, work is speculatively completed, and error is prevented by only committing moves that did not interact or share data. Thus, in the high temperature regime, where many moves interact, a much speculative work will be wasted.

Some of the earlier efforts achieved decent speedups but at the expense of quality of results[11], [5]. In fact, quality worsened as parallelism increased.

III. Ideal Speedup and Challenges

Before we delve into the details of our proposed approach, it is instructive to estimate the "ideal" speedup and then discuss the challenges which prevent the realization of this speedup.

NVIDIA's GTX280 consists of 240 cores divided into 30 groups, called *streaming multiprocessors* (SMPs), each of which consists of 8 processors, called *streaming processors* (SPs). The SMPs operate independently from each other, while each group of eight SPs (in other words those within the same SMP) work in a SIMD fashion.

Optimistically assuming that all 240 cores could function independently, a naive implementation would expect a speedup of 120 times owing to the number of cores and the fact that the GPU's clock frequency is half of the CPU's.

In reality, the GPU has a number of architectural issues which prevent a naïve approach from achieving this ideal speedup, namely the *SIMD architecture* of the SMPs and *constrained memory architecture*. Firstly, the SIMD architecture forces a warp (which is a group of thirty-two threads) to execute the same instruction. If each thread within a warp attempts to execute a different instruction, then they will serialize which can lead a slowdown of up to thirty-two times. The second constraint is memory architecture. The GPU memory controller is optimized to access contiguous regions of memory using a *coalesced access*. However, simulated annealing is memory intensive and involves many random and scattered memory accesses. Consequently, this could degrade performance by an order of magnitude[12].

IV. Proposed Approach

Our approach for parallelizing simulated annealing placement replaces a single annealing move with two new stages, namely *subset generation* and *parallel annealing* (see Algorithm 1).

Algorithm 1 Parallel Simulated Annealing

```
 1: procedure parallelSA(Netlist N)
 2: P = randomInitialPlacement()
 3: Set T = INITIAL_TEMPERATURE
 4: Set W = INITIAL_WINDOW_SIZE
 5: repeat
 6:     for M times do
 7:         {$S_i$} = generateSubsets(W,N,$N_s$,$S_s$,P)
 8:         parallelAnneal({$S_i$},N,P,T,W)
 9:     end for
10:     T = updateT(T)
11:     W = updateW(W)
12: until Termination Condition Met
13: end procedure
```

A. Subset Generation

The objective of subset generation is to create a collection of subsets where each subset is simply a group of nodes from a netlist.

There are three concerns with subset generation.

- *Consistency:* Since this is a parallel approach, race conditions arise when subsets share nodes and then shared nodes are updated in parallel. To resolve this, we prevent subsets from sharing nodes.
- *Available Moves:* Moves only involve nodes which are within a certain distance of each other. If each pair of nodes within a subset is separated by more than this distance, then no moves are available to anneal and this wastes the time spent generating the subset. To avoid this situation, the subset generator is placement aware to ensure some available moves exist.
- *Randomness:* Simulated annealing heavily depends on randomness to ensure high quality solutions. Therefore, subsets are randomly generated with the caveat of being placement aware.

Note that the generated group might not contain all the nodes in a netlist; in other words, it is not a netlist partition.

Our generation solution is given in Algorithm 2. Intuitively, each subset starts with a random initial node and randomly adds nodes which are nearby in terms of placement.

Algorithm 2 Subset Generation Algorithm

```
1: function generateSubsets( Window W, Netlist N, Number of subsets $N_s$, Size of subset $S_s$, Placement P )
2:   Define $\{Q_i\}$ // queues for each subset
3:   Define $\{S_i\}$ // a group of subsets
4:   for $i$ = 1 TO $N_s$ do
5:     $Q_i = \{\}$
6:     n = randomNode() // randomly remove a node from N
7:     $Q_i$.enqueue(n)
8:   end for
9:   for $k$ = 1 TO $S_s$ do
10:    for $i$ = 1 TO $N_s$ do
11:      n = $Q_i$.dequeue()
12:      $S_i$.push(n)
13:      for $j$ = 1 TO 4 do
14:        m = randomWithinWindow(W,n)
15:        $Q_i$.enqueue(m)
16:      end for
17:    end for
18:  end for
19:  return $\{S_i\}$
20: end function
```

Algorithm 3 Parallel Annealing Algorithm

```
1: procedure parallelAnneal(Subsets $\{S_i\}$,Netlist N, Placement P, Temperature T, Window W)
2:   Read subset data into shared memory
3:   Compute pre-bounding box for each net in subset
4:   Generate a pool of valid swapping nodes
5:   for M moves do
6:     Select a swap from pool
7:     Compute changes in cost function per net
8:     Sum changes across all nets using scan operator
9:     Decide and possibly commit
10:  end for
11:  Write data to global memory
12: end procedure
```

B. Parallel Annealing

We have highly optimized the sequential version of simulated annealing for the GPU's manycore architecture.

The same algorithm (Algorithm 3) is executed on different SMPs using different subsets. Given the high memory access latency to the off-chip global memory, we start by following the common practice of prefetching the data associated with one or more subsets into the low latency on-chip shared memory (line 2 of Algorithm 3).

Subsequently, an initial approximation of each net's bounding box, called *pre-bounding box* is computed and stored in shared memory (line 3 of Algorithm 3) for low latency access. Conceptually, the pre-bounding box captures the placement information for nodes outside of the subset. A pre-bounding box for a net is simply the smallest box which encompasses all nodes on the net, but only considers nodes *not* in the subset. The actual bounding box of a net can be computed by combining the pre-bounding box information with the positions of relevant nodes in the current subset.

TABLE II
IWLS Benchmarks Sizes

	Number of Placeable Nodes		
	Cluster Size		
Stitched Benchmark	1	4	10
b17_1	124670	31244	12467
b18_1	157149	39396	15715
b18	156812	39307	15682
b19_1	306719	76907	30672
leon2	273921	68481	27393
leon3mp	216562	54141	21657
netcard	203750	50938	20375
uoft_raytracer	170069	42518	17007

Next, a pool of moves is computed (line 4). Each thread within a warp randomly selects two nodes from the subset. If both nodes are within the window size, they are added to the pool.

Finally, several moves are performed with each consisting of four steps. First, a move from the pool is selected. Second, each net affected by the move is identified and mapped to a different thread which then computes the change in the cost function as a result of the current swap. Thirdly, the changes per net are computed and are summed with a scan primitive as described by [13]. These primitives leverage the SIMD architecture and allow a set of N numbers to be summed in $O(\log N)$ time instead of $O(N)$ time. Lastly, the updated placement information is committed back to global memory.

C. Error Tolerance

Our approach permits transient errors. The pre-bounding box for each net is not updated over the course of a single parallel annealing iteration, and since nodes on those nets may be annealed on other SMPs, it could become stale. We empirically found that this does not significantly impact the quality of results. Furthermore, the error is transient, because all information is synchronized at the end of parallel annealing iteration.

V. Experimental Results

A challenge in evaluating placement scalability and QoR is the lack of large academic circuits for FPGAs. We used the IWLS benchmarks and modified them using a technique first devised at Altera [14]. The resulting benchmark suite is given in Table II after packing with T-VPACK [3] with a cluster size of 1, 4 and 10.

The proposed solution was implemented in C and ran on a 64-bit Linux machine using an Intel Core 2 Quad running at 2.66 GHz with 2GB of memory and an NVIDIA GTX280 GPU running at 1.35GHz and with 1GB of on-chip memory. The sequential implementation is a modified version of VPR 4.3 [3]. We report the QoR degradation which we define as

$$QoR = \frac{S - P}{S}(100\%) \qquad (1)$$

TABLE III
Parameters used for IWLS Benchmarks

Cluster Size	Subset Size	Number of Subsets	Slowdown
1	28	120	1.5
4	20	90	1.0
10	22	30	1.0

TABLE IV
Results for IWLS Benchmarks (Lower QoR is Better)

	Cluster Size=1		Cluster Size=4		Cluster Size=10	
	QoR	Speedup	QoR	Speedup	QoR	Speedup
b17_1	-4.40	10.16	2.06	21.97	-1.53	12.29
b18_1	-9.32	10.69	-1.70	23.60	3.32	12.76
b18	-3.20	10.03	2.44	22.47	3.24	13.13
b19_1	7.13	9.94	4.93	25.31	6.99	14.97
leon2	5.60	6.60	5.21	14.89	-0.07	9.92
leon3mp	-0.79	7.03	4.00	15.28	-3.90	9.04
netcard	-4.00	7.40	1.51	16.82	0.21	9.69
	1.58	8.39	-2.15	12.49	11.58	5.94
Average	**-0.93**	**8.78**	**2.04**	**19.10**	**2.48**	**10.97**

where S is the half-perimeter wirelength of the sequential version and P is the result from the parallel version. These values are reported as percentage. Speedup is the ratio of sequential annealing time to parallel annealing time.

A. Tuning Parameters

The parameters must be carefully selected, since they heavily influence performance and quality of results. The final choices are summarized in Table III.

Number of Subsets and Subset Size: The number of subsets describes how many subsets will be annealed concurrently; while the subset size is the number of nodes in the subset. Ideally, the number of subsets should be large to increase parallelism and each subset should be as large as possible. Unfortunately these values are constrained by the amount of available on-chip memory. In order to select these values, we prioritize number of subsets over the size and ensure that all subsets can fit in on-chip share memory at the same time.

Number of subset groups stored: This is the total number of spaces available for groups to be stored. In other words, when a group is randomly selected it can be selected from one of those stored in these spaces. Empirically, we found that this number has an optimal point value, which is dependent on the benchmark (again we divide cases according to cluster size).

Slowdown: For the cluster size of 4 and 10, we maintain the same annealing schedule as the sequential, but for cluster size of 1 we slow it down by a factor of 1.5.

B. Overall Speedup and Quality of Results

Table IV summarizes our results for the IWLS benchmarks. Note that negative QoR values indicate that the parallel version performed better. On average, we achieve an order of magnitude speedup with minor, if any, degradation in QoR.

VI. Conclusions and Future Work

In this work we have presented a highly parallel solution to the simulated annealing-based placement problem using GPGPU. We find that despite the architectural challenges of SIMD cores and constrained memory interface, an order of magnitude speedup is achievable without sacrificing quality of results.

Future work may consider two extensions: first, more objective functions could be considered, including timing and congestion; and second, determinism, which would assert that multiple runs of the same input netlist would always result in the same outcome.

Acknowledgment

The authors would like to thank Edward Campbell for insightful discourse, Humian Bian for assisting with the software infrastructure, and Andrew Ling for generating the placement benchmarks. Also we would like to acknowledge the funding which NSERC provided for this project.

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, no. 8, pp. 114–117, April 1965.

[2] C. Sechen and A. Sangiovanni-Vincentelli, "The timberwolf placement and routing package," *Solid-State Circuits, IEEE Journal of*, vol. 20, no. 2, pp. 510–522, Apr 1985.

[3] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*. London, UK: Springer-Verlag, 1997, pp. 213–222.

[4] A. Ludwin, V. Betz, and K. Padalia, "High-quality, deterministic parallel placement for FPGAs on commodity hardware," in *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2008, pp. 14–23.

[5] P. Banerjee, M. H. Jones, and J. S. Sargent, "Parallel simulated annealing algorithms for cell placement on hypercube multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 1, pp. 91–106, 1990.

[6] NVIDIA, "NVIDIA CUDA," [online] http://www.nvidia.com/cuda.

[7] S. A. Kravitz and R. A. Rutenbar, "Multiprocessor-based placement by simulated annealing," in *DAC '86: Proceedings of the 23rd ACM/IEEE Design Automation Conference*. Piscataway, NJ, USA: IEEE Press, 1986, pp. 567–573.

[8] E. E. Witte, R. D. Chamberlain, and M. A. Franklin, "Parallel simulated annealing using speculative computation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 4, pp. 483–494, 1991.

[9] W.-J. Sun and C. Sechen, "A loosely coupled parallel algorithm for standard cell placement," in *ICCAD '94: Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 137–144.

[10] A. Casotto, F. Romeo, and A. Sangiovanni-Vincentelli, "A parallel simulated annealing algorithm for the placement of macro-cells," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 6, no. 5, pp. 838–847, September 1987.

[11] J. A. Chandy and P. Banerjee, "A parallel circuit-partitioned algorithm for timing-driven standard cell placement," *J. Parallel Distrib. Comput.*, vol. 57, no. 1, pp. 64–90, 1999.

[12] NVIDIA, "NVIDIA CUDA compute unified device arhcitecture programming guide: Version 2.0," September 2009.

[13] S. Chatterjee, G. E. Blelloch, and M. Zagha, "Scan primitives for vector computers," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 666–675.

[14] Altera, "OpenCore stamping and benchmarking methodology," Altera, Tech. Rep. TB-098-1.1, 2008.