

Parallel FPGA Placement with Symmetric Multiprocessors (SMPs) and Vector Functional Units

Pak K. Chan and Martine D.F. Schlag
Department of Computer Engineering
University of California, Santa Cruz, California 95064 USA *

Abstract

Placement and routing are the most time-consuming processes in automatically synthesizing and configuring circuits for field-programmable gate arrays (FPGAs). Over the last decade researchers have developed methods for accelerating these processes. These approaches either require non-standard computing environments, or affect the quality and/or reproducibility of the results. In this paper, we accelerate placement using a readily available computing platform without sacrificing the quality of the results or introducing non-determinism. We obtain speedups of 1.34 using the multi-core architecture and the SIMD units commonly available on contemporary microprocessors. Our experiences in parallelizing the analytic placer NAP (Negotiated Analytic Placement) show that the grain of parallelism, memory access patterns and synchronization overhead must be carefully considered to achieve speedup. These results can be extended to other placement methods.

1 Introduction

Given a circuit represented as a connection of logic blocks, the problem of placement for FPGAs can be stated as that of assigning each logic block to a unique physical resource while achieving a given overall performance. As the capacity of FPGA devices as well as the size of FPGA users' circuits grow, there is a great demand to place designs rapidly. There are several approaches to accelerating placement.

- One can sacrifice/trade the quality of the placement for the amount of time it takes to place the circuit[1, 2]. This approach seems to imply a user will know the proper quality/time trade-off of the placer *a priori*.
- Techniques for parallelizing simulated annealing have been used to accelerate VPR[4] on expensive shared-memory machines (SGI Origin) or specialized distributed memory multiprocessors (IBM-SP2)[3].
- FPGA based computing platforms to accelerate placement and routing have been proposed [5, 6].
- A placer that is easily parallelizable in a ubiquitous network environment, NAP [7], produces speedups of 2-3 using a small number of machines connected on a local network. While more

*We acknowledge Xilinx and UC MICRO for their support.

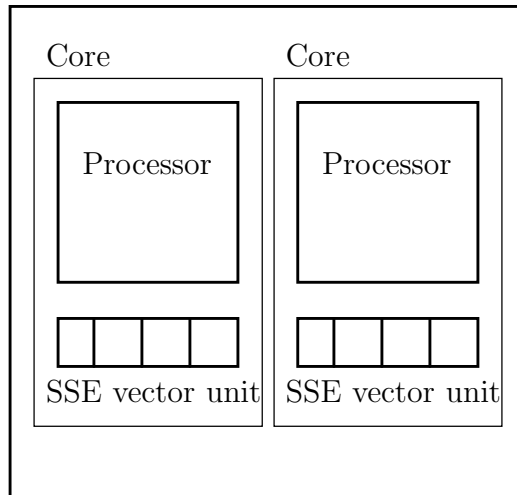


Figure 1: SMP architecture with SIMD/SSE vector unit.

readily available, this environment still requires the user to have access to a network and multiple machines. The variability in message delays introduces non-determinism into the method.

Although these methods show potential for achieving significant speed up in placement, FPGA designers are still using serial placement algorithms, while the modest resources for parallel computation available in their machines remain untapped. This paper describes the parallelization of the Negotiated Analytical Placer (NAP) using both multi-core and SIMD units. Although the speedups are not in the same range as some of the previously reported methods, they are immediately and more widely applicable.

Contemporary Symmetric Multiple Processor (SMPs) have a small number of processor cores and each core has a Single Instruction Multiple Data (SIMD) vector unit as illustrated in Figure 1. Examples of multi-cores are AMD Opteron cores, Intel Xeons, and IBM cell architecture. The Single Instruction Multiple Data (SIMD) unit in each processor core has been designed for vector arithmetic, logical, and permutation operations common in multimedia applications. Examples of SIMD vector units are **SSE** (Streaming SIMD Extension) in Intel and AltiVec in PowerPC.

We begin by briefly reviewing the NAP placement algorithm in Section 2. In Section 3, we examine different options for partitioning the work among the threads running on different cores. Experiments show that the grain of parallelism and memory traffic patterns are critical to achieving speedup. In Section 4 we show that additional speedup can be obtained by using the SIMD unit available within each processor. We conclude by providing the overall speedups using both features.

2 The NAP placement algorithm

In this section we briefly describe the NAP algorithm [7] which is a hybrid of the well known analytical placement technique and the negotiation method combined in an iterative framework. Analytical placement is used to minimize the weighted quadratic displacement between blocks while the negotiation paradigm is used to resolve the competition of blocks for locations. The negotiation method has been shown to converge for bipartite matching problems such as the problem of assigning blocks to locations[8]. Figure 2 gives a high-level description of the algorithm. The input is the netlist of I/O

```

NAP(netlist, FPGA architecture)

  Cover the netlist with trees
  while the placement is not feasible
    for each tree  $T_i$ 
      Calculate analytical placement  $\vec{V}_i$  of  $T_i$ 
    endfor
    for each tree  $T_i$ 
      for each node  $n_j$  in  $T_i$ 
        find position for  $n_j$  in window centered at  $V_{i,j}$  minimizing
           $clone\_cost + history\_cost + shared\_cost + Iteration(\text{mod}2)\Delta wirelength$ 
      endfor
      Update delays and criticalities
    endfor
    Update history costs
  endwhile

end

```

Figure 2: Placement algorithm.

and logic blocks; the output is the assignment of logic blocks to locations.

The first step is to cover the netlist with trees. Analytical placement of a tree structure is extremely efficient hence suitable for the inner-loop of an iterative algorithm. To represent every circuit connection as an edge in at least one tree, blocks will appear multiple times, either as nodes in the same tree or separate trees. The tree nodes corresponding to a particular logic block are referred to as the *clones* of that block. In a feasible placement all of the clones of a block must be placed at the same location.

Fine placement of each clone is achieved by searching the neighborhood of the position calculated in the analytic placement and selecting a location within the neighborhood which minimizes the cost function:

$$clone_cost + history_cost + shared_cost + Iteration(\text{mod}2)\Delta wirelength. \quad (1)$$

The *clone_cost* is the distance to the logic block's center of gravity (the geometrical center of its clones' positions). The *history_cost* of a location is increased at the end of every iteration after which that location is over-committed. The *shared_cost* is the number of other logic blocks that currently occupy that location (have at least one clone present). The weights of the shared and clone components as well as the size of the neighborhoods are adjusted as the algorithm proceeds to achieve feasibility.

The spans of each net in both dimensions are maintained so that the change in wirelength of the nets can be quickly evaluated. The centers of gravity of the blocks are used as the coordinates of the logic blocks for the wirelength and delay calculations.

The delays of all the edges (source-to-sink connections) are updated after all trees have been placed,

and their criticalities are updated for use in the analytical placement in the next iteration. The delay of a source-to-sink connection is the sum of the delays associated with a shortest path through the routing architecture; in this case, the sum of the delays for the pips, switches and wire segments.

The algorithm is applied hierarchically by clustering the grid locations into which the clones must be placed. For example, a 60×60 grid using a cluster size of 3 becomes a 20×20 grid with each grid position (slot) capable of holding 9 logic blocks. This improves the efficiency by allowing placement constraints to be resolved at a coarser level. All delay and wirelength calculations use the full coordinate system; the coordinates of the grid position at the center of the slot are used as the position for each clone in that slot. Once feasibility or near feasibility is attained at the coarser level, the current placement is expanded with each grid position inheriting an interpolated version of the history of its slot.

Table 1 gives the running times and results of NAP on all 20 sequential and combinational benchmarks from FPGA Place and Route Challenge [9]. These times are elapsed time on a dual-core 2.0 GHz Opteron machine (Sun Fire V20z). For comparison with other placement methods, VPR 4.3 [4] with the option `-place_algorithm path_timing_driven` has x wirelength 71.18, y wirelength 69.67 estimated delay 2043.4367 ns, and running time 4262.96 seconds, summed over all 20 benchmarks on the same machine.

Design	x span	y span	delay	time
alu4	3.45	2.77	70.60	27.04
misex3	3.88	3.48	68.80	39.32
seq	3.75	3.85	71.80	43.14
des	4.55	4.15	83.60	46.67
apex4	4.68	3.84	74.80	49.87
ex5p	5.03	4.94	70.40	55.73
dsip	3.98	4.03	68.60	67.44
tseng	2.43	2.40	65.20	78.78
apex2	4.01	3.59	80.20	86.25
bigkey	2.92	3.22	55.60	187.66
spla	4.37	4.02	107.60	251.24
frisc	4.04	3.94	132.00	267.27
ex1010	3.61	4.12	142.20	275.48
pdc	4.93	5.08	122.40	334.26
diffeq	2.39	3.01	71.60	354.95
s298	1.67	1.92	117.00	374.82
s38584	2.65	2.76	87.80	444.85
elliptic	3.55	4.10	113.60	458.20
s38417	2.85	2.96	104.80	644.85
clma	3.90	3.85	150.00	858.61
Totals	72.64	72.03	1858.60	4946.43

Table 1: Results of NAP on the 20 benchmarks from the FPGA Place and Route Challenge. Wirelengths in the x and y dimensions are in grid units. Delays (in nanoseconds) are estimated using the same delay model as VPR. Elapsed running times are given in seconds.

3 Thread-Based Parallelism

Our new SMP parallel placer implementation is based on a global shared memory model with thread synchronization. Multiple threads can be executed on processors simultaneously. Each thread has a local stack and shares the global address space with other threads. Synchronization primitives are signals and semaphores.

Our SMP implementation of the NAP algorithm is based on GCC 3.4 with the posix thread library. The posix thread library includes functions to support simultaneous multiple thread creation and synchronization. GCC also has primitives to support the SSE instruction set whose use we will explore in the next section. Our hardware platform is two dual-core 2.0 GHz Opterons (Sun Fire V20z) running Linux 2.6 kernel, essentially two of the structures (four cores) shown in Figure 1.

Key considerations in multi-thread programming are memory interference, thread synchronization overhead, and result determinism. Creation and synchronization of threads are expensive, so care must be taken in choosing the right level of granularity to thread a process. Semaphores must be introduced carefully to avoid non-determinism. It is also important to minimize the number of semaphores. Threading a small function is counter-productive in terms of speedup, while conservative threading will produce little speedup. Since the threads share a global address space it is advantageous to thread functions whose memory access patterns don't interfere with each other; they don't require the same data and their results will not be combined immediately.

After some experimentation, our final thread implementation uses a group of live "workers" with signals to enable or disable them. The worker threads are created at the beginning of the program and they only expire when the program exits. Each worker is awakened by a broadcast conditional signal issued from the main program and each is given a subroutine to evaluate. Upon completion, a worker hibernates and conditional waits for work again. This scheme minimizes the creation and synchronization overhead of threads. The number of semaphore synchronizations is limited to one per worker activation. Work is partitioned among disjoint data sets to avoid the use of semaphores on shared objects. There are **no** semaphores on the data sets!

Figure 3 contains the profile of NAP running on the largest benchmark (*clma*). It shows that the bulk of the computation is spent in the analytic tree placement routines and the wirelength change calculations. The routine `mgetChange` calculates the weighted change in wirelength in one dimension for all locations within a range. The routine `update_spans` modifies the data structures when the position of a clone is moved. The routines `calcGdown`, `calcVup`, `calcVdown`, `calcGup` which together account for 29.7% of the total time, calculate the analytic placement of the trees.

A seemingly obvious division of labor in the wirelength calculation is to have the x and y dimensions calculated by two different threads. But in our experiments this resulted in a slowdown of the program by almost a factor of two. While the x and y calculations are independent, they both require access to their common signal's data, and the results of these independent calculations need to be combined immediately after their completion.

Instead we threaded the analytical placement and the timing analysis routines. NAP's tree decomposition of the netlist provides the necessary flexibility for experimenting with variable granularity of the threads. Since the partitioning of the trees among the workers is fixed, no parameters must be passed to the worker threads other than a worker tag. Each worker works on a predetermined subset of trees according to his/her tag. The results of the tree placement are not required until the next loop through the trees (the fine placement).

% time	cumulative seconds	self seconds	self calls	self Ks/call	total Ks/call	name
18.04	1595.05	1595.05	147382840	0.00	0.00	Placement::mgetChange(entry*, int, int, float*)
11.63	2623.27	1028.22	168742993	0.00	0.00	update_spans(entry*, float)
10.87	3584.86	961.59	164846	0.00	0.00	Placement::bigplace(Tree*)
9.62	4435.50	850.64	183490	0.00	0.00	calcGdown(Tree*)
7.36	5086.49	650.99	183490	0.00	0.00	calcVup(Tree*, int)
6.69	5678.50	592.01	183490	0.00	0.00	calcVdown(Tree*)
6.47	6250.47	571.97	3228407695	0.00	0.00	Placement::newinslot(Module*, int, Occupancy**)
5.42	6729.45	478.98	183490	0.00	0.00	calcGup(Tree*, float)
3.63	7050.36	320.91	28650978	0.00	0.00	Placement::mgetChangeNof3(entry*, int, int, float*)
2.92	7308.68	258.32	98402574	0.00	0.00	calcArcDelay(Arc*)
2.18	7501.21	192.53	37170	0.00	0.00	Placement::place(Tree*, int)
2.05	7682.83	181.62	16772	0.00	0.00	LoopcalcClones(void*)
1.85	7846.80	163.97	54938098	0.00	0.00	Placement::remove_Occ(Tree*)
1.57	7985.73	138.93	3121	0.00	0.00	DAGPaths::findPaths(Vertex*)
1.50	8118.52	132.79	3121	0.00	0.00	DAGPaths::findRevPaths(Vertex*)
1.36	8238.68	120.16	95028960	0.00	0.00	ArcEdge::Heapify()
0.92	8319.76	81.08				Array::nextElement(LoopData*)
0.80	8390.51	70.75	54983363	0.00	0.00	Placement::AddOcc(int, Tree*)
0.79	8460.39	69.88	190711826	0.00	0.00	DAGPaths::relax(Vertex*, Vertex*, Edge*, int)
0.58	8511.77	51.38	766164	0.00	0.00	getChangeSwap(float, int, entry*, Module*)
0.58	8563.09	51.32	54892609	0.00	0.00	update_xy(Tree*, int, int)
.						
.						
.						

Figure 3: Profile of NAP running on the largest benchmark (*clma*).

The delay calculation requires both a forward and backward traversal of the signal dependency graph to obtain the criticality of each connection (routines `findPaths` and `findRevPaths`). Though this calculation accounts for only 3% of the time for *clma*, it ranges upto 8% for other benchmarks. A thread is created for the backward traversal so that both may be calculated simultaneously.

In our experiments 4 threads are used for analytical placement, and two threads are used for parallel forward and backward path tracing. Table 2 presents the elapsed times obtained using these multiple threads. The placements are the same as those produced by the unthreaded version (reported in Table 1). The average of the speedups of all 20 benchmarks is 1.196, while the speedup of the total running time of all 20 is 1.227.

To put these speedup values into perspective: since the placer spends only 29.7% of the time in the analytic placement calculation and 3.07% of the time in the path traversals for *clma*, the maximum speedup achievable for this benchmark by using multiple threads is about 1.32. Note that the speedup tends to grow with running time. In the tables, the benchmarks are ordered by running time. The average speedup observed for the first 10 benchmarks is 1.156 while the last 10 have an average speedup 1.24. The largest benchmark, *clma*, has 8527 blocks which is small by today's standards.

4 Multiple Instruction Parallelism

Each processor core has a 128-bit (SIMD) vector unit which supports both integer and floating-point data formats[10]. The vector unit has the same clock frequency as the main processor. Our implementation uses 32-bit floating-point representations and hence 4 data items can be operated on simultaneously in a single Streaming SIMD Extensions (SSE) instruction. SSE instructions are limited to either arithmetic, logical, load/store and comparison; there are no branch instructions.

Design	Serial	SMP	Speedup
alu4	27.04	21.41	1.26
misex3	39.32	36.98	1.06
seq	43.14	35.48	1.22
des	46.67	40.82	1.14
apex4	49.87	41.71	1.20
ex5p	55.73	49.55	1.12
dsip	67.44	61.08	1.10
tseng	78.78	70.28	1.12
apex2	86.25	72.59	1.19
bigkey	187.66	164.61	1.14
spla	251.24	211.33	1.19
frisc	267.27	247.14	1.08
ex1010	275.48	219.55	1.25
pdcc	334.26	258.24	1.29
diffeq	354.95	263.92	1.34
s298	374.82	328.42	1.14
s38584	444.85	341.57	1.30
elliptic	458.20	376.93	1.22
s38417	644.85	510.74	1.26
clma	858.61	678.03	1.27
Totals	4946.43	4030.38	

Table 2: Comparison of running times of unthreaded and threaded versions of NAP on the 20 benchmarks from the FPGA Place and Route Challenge. Elapsed running times are given in seconds.

```
// Sample C code
void delta_left(float result[4], float pos[4], float loc, float wgt)
{
    for(int i=0;i<4;i++)
        result[i] += (pos[i] - loc)*wgt;
}
//-----
// Re-written using SSE instructions
typedef float v4sf __attribute__((mode(V4SF))); // vector of 4 floats
void delta_left(float result[4], float pos[4], float loc, float wgt)
{
    v4sf vresult, vpos, vloc, vwgt;
    vresult = _mm_loadu_ps(result);
    vpos = _mm_loadu_ps(pos);
    vloc = _mm_set1_ps(loc); // fills vector with loc
    vwgt = _mm_set1_ps(wgt); // fills vector with wgt
    vresult = _mm_add_ps(vresult, _mm_mul_ps(_mm_sub_ps(vpos, vloc),vwgt));
    _mm_store_ps(&result[0], vresult);
}
```

Figure 4: Example of C code rewritten for SSE vector unit.

The analytical placement phase of NAP computes “coarse” coordinates for the clones. The fine placement phase searches for the best location within the neighborhood of the “coarse” coordinate according to the objective function. The score of each location within this search window is computed in vector form in floating point arithmetic. Because it is such a fine grain operation, threading is counter productive for fine placement. Instead we rewrote this calculation using the the SSE arithmetic and selection operators. Figure 4 shows how a segment of C code can be rewritten to use the SSE instruction set.¹ To obtain speedup it is important to reduce the number of conditional branches, replacing them with selection wherever possible.

Design	x span	y span	delay	time	speedup
alu4	3.45	2.87	72.00	27.38	0.988
misex3	3.78	3.46	68.20	34.26	1.148
seq	3.78	3.89	70.60	39.85	1.083
des	4.66	4.37	84.80	40.91	1.141
apex4	4.58	3.92	72.20	45.81	1.089
ex5p	5.06	4.94	70.20	48.43	1.151
dsip	3.48	3.35	70.40	62.87	1.073
tseng	2.42	2.42	64.60	66.19	1.190
apex2	4.03	3.59	81.20	77.97	1.106
bigkey	3.08	3.14	65.40	145.15	1.293
spla	4.18	4.09	108.00	231.53	1.085
frisc	4.12	3.92	130.20	278.68	0.959
ex1010	3.76	3.77	143.00	257.26	1.071
cdc	4.86	5.10	124.00	345.56	0.967
diffeq	2.43	2.90	70.60	118.46	2.996
s298	1.73	1.85	113.80	332.40	1.128
s38584	2.84	2.87	88.80	414.56	1.073
elliptic	3.64	3.98	115.00	385.73	1.188
s38417	3.00	2.96	111.20	766.69	0.841
clma	3.79	4.14	154.20	805.38	1.066
Totals	72.67	71.53	1878.40	4525.07	

Table 3: Results of NAP using SSE instructions. Wirelengths in the x and y dimensions and estimate delays are reported. Elapsed running times are given in seconds. Speedups with respect to running times without SSE instructions are reported although numerical variations also affect the running time.

Table 3 contains the results on the 20 benchmarks. The wirelengths and delays are slightly different as is expected when moving a numerical algorithm from one platform to another. The elapsed running times and speedup are reported, but it is more difficult to quantitatively assess the speedup of the SIMD unit due to changes in the numbers of iterations and operations. An extreme example is the benchmark `diffeq` which appears to enjoy a speedup of 3.0, but in fact this is primarily due to the number of iterations dropping by a factor of 0.42. We estimate that the contribution of the SIMD unit is somewhere between 5% and 10% based on the benchmarks for which the number of iterations varied by less than 10%. In some cases, we observed a decrease in running time despite an increase in the number of iterations. Though the reorganization of the operations to take advantage the SIMD unit results in numerical variations, the algorithm is still deterministic. The results are reproducible are long as the SSE unit is used.

¹This code is a simple example to illustrate the process. The actual code from the fine placement routine is more complicated.

5 Conclusions

We have described a parallel implementation of negotiated analytical placement using SMP and SSE hardware platforms. The combined results are given in Table 4. This implementation yields a speedup of 1.34 for running all 20 benchmarks.

Design	Elapsed Times				Speedups		
	Serial	SMP	SIMD	SMP+SIMD	SMP	SIMD	SMP+SIMD
alu4	27.04	21.41	27.38	21.43	1.263	0.988	1.262
misex3	39.32	36.98	34.26	28.54	1.063	1.148	1.378
seq	43.14	35.48	39.85	32.36	1.216	1.083	1.333
des	46.67	40.82	40.91	35.13	1.143	1.141	1.328
apex4	49.87	41.71	45.81	37.47	1.196	1.089	1.331
ex5p	55.73	49.55	48.43	42.25	1.125	1.151	1.319
dsip	67.44	61.08	62.87	52.15	1.104	1.073	1.293
tseng	78.78	70.28	66.19	57.27	1.121	1.190	1.376
apex2	86.25	72.59	77.97	67.00	1.188	1.106	1.287
bigkey	187.66	164.61	145.15	124.36	1.140	1.293	1.509
spla	251.24	211.33	231.53	190.26	1.189	1.085	1.321
frisc	267.27	247.14	278.68	229.11	1.081	0.959	1.167
ex1010	275.48	219.55	257.26	224.45	1.255	1.071	1.227
pdc	334.26	258.24	345.56	258.50	1.294	0.967	1.293
diffeq	354.95	263.92	118.46	101.11	1.345	2.996	3.511
s298	374.82	328.42	332.40	281.73	1.141	1.128	1.330
s38584	444.85	341.57	414.56	359.53	1.302	1.073	1.237
elliptic	458.20	376.93	385.73	319.92	1.216	1.188	1.432
s38417	644.85	510.74	766.69	595.29	1.263	0.841	1.083
clma	858.61	678.03	805.38	627.32	1.266	1.066	1.369
Totals	4946.43	4030.38	4525.07	3685.18			

Table 4: Elapsed running times and speedups NAP versions: Serial, threaded SMP, SIMD using SSE instructions, and threaded SMP plus SIMD using SSE instructions.

High performance microprocessors with many processor cores are on the horizon: Opteron Quad Core, Intel Quad Core, Sun Niagara II, as well as IBM Cells. We believe that acceleration of placement and routing for FPGAs can be made available to the “common” FPGA designer by taking advantage of these multiple core microprocessors.

Another possible source of speedup is to exploit the natural partitions resulting from FPGA designs consisting of multiple logic blocks or IP cores. For example, an FPGA-based PCI express ethernet card might consist of a DDR IP core, a PCI express core, and an ethernet MAC core. Since each logic block has a well defined boundary, these core-based designs provide natural partitions for parallel placement.

References

- [1] Y. Sankar and J. Rose, "Trading quality for compile time: Ultra-fast placement for FPGAs," in *Proceedings of International ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, (Monterey, California, USA), pp. 157–166, Feb. 1999.
- [2] C. Mulpuri and S. Hauck, "Runtime and quality tradeoffs in FPGA placement and routing," in *9th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, (Monterey, California), February 2001.
- [3] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "Parallel algorithms for FPGA placement," in *The Great Lakes Symposium on VLSI*, (Chicago), March 2000.
- [4] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," in *Seventh International Workshop on Field Programmable Logic and Applications*, (London, England), pp. 213–222, 1997.
- [5] M. G. Wrighton and A. M. DeHon, "Hardware-assisted simulated annealing with application for fast FPGA placement," in *9th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, (Monterey, California), February 2001.
- [6] P. K. Chan and M. Schlag, "Acceleration of an FPGA router," in *1997 IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 175–181, 1997.
- [7] P. K. Chan and M. D. Schlag, "Parallel placement for field-programmable gate arrays," in *11th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, (Monterey, California), February 2003.
- [8] P. K. Chan, M. D. F. Schlag, C. Ebeling, and L. McMurchie, "Distributed-Memory Parallel Routing for Field-Programmable Gate Arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-19, pp. 850–862, August 2000.
- [9] V. Betz, "FPGA Place-and-Route Challenge," tech. rep. <http://www.eecg.toronto.edu/vaughn/challenge/challenge.html>.
- [10] "Processors, Define SSE2 and SSE3," tech. rep. <http://www.intel.com/support/processors/sb/cs-001650.htm>.