Parallelizing an Analytical Placer

David Goldman*
david.goldman@rogers.com

Bryce Leung*
bryce.leung@rogers.com

Jungmoo Oh*
moo8181@gmail.com

*Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada

ABSTRACT

Analytical Placement (AP) is a CAD algorithm for ASIC placement. This algorithm generates a system of linear equations, which we solve using Gaussian elimination. We demonstrate how the specific sub-problem of Gaussian elimination on AP input matrices has the unique property where the pivot row is always in place, and show how this greatly enhances our ability to parallelize the algorithm. We demonstrate why a sparse-matrix implementation of the elimination algorithm would be difficult if not impossible to implement on a GPU. We then parallelize a full-matrix implementation of the placer on a GPU using CUDA, and show good speedups versus an optimized CPU implementation.

1. INTRODUCTION

GPUs are a useful tool for parallelization of non-graphical programs that would traditionally be executed on a CPU. They contain an abundance of parallelization hardware that can be exploited to provide fine-grained parallelization that would require too much overhead in a thread-based CPU approach. This, combined with the fact that GPU manufacturers have been adding APIs to access this hardware from pre-processed C code, makes this approach particularly attractive. We focus our research exclusively on the NVIDIA API called CUDA.

CAD algorithms are generally known to be computationally expensive, so we chose to parallelize a classic algorithm in this space called Analytical Placement (AP). AP is a placement technique, often used commercially in an ASIC CAD flow. It is known in literature for its good quality and fast runtimes. A typical AP flow computes a placement in a single run of the algorithm. It does this by forming a set of simultaneous linear equations which represent a fixed IO placement, a set of blocks to be placed, the nets that connect them, and a goal to minimize. These equations are then solved, producing a real-valued coordinate for each block. The block coordinates are then discretized using various heuristics. For simplicity purposes, this paper will focus purely on minimization of wirelength, although other goals, such as timing, power, and signal integrity are also commonly used. As well, this paper will not focus on the heuristics and techniques used for discretizing the placement.

The sections of this paper are as follows. In section 2, we discuss the problem definition as well as the theoretical background for AP. In section 3, we discuss a unique property of the problem space that allows us to implement a much faster parallelization technique, as well as the pseudo-code for the algorithm that we chose. In section 4 we show our results. In section 5, we give a step-by-step study of how one might re-design the algorithm in order to fully maximize use of the hardware. In section 6 we present our conclusion.

2. PROBLEM DEFINITION

The AP problem inputs are:

A netlist of blocks and connectivity
Coordinates for fixed blocks (I/Os)
A net model

The AP output is a real-valued (x, y) coordinate for every non-fixed block in the netlist.

One example of a net model is the clique model. A clique in an undirected graph G is a set of vertices V such that for every two vertices in V there exists an edge connecting the two. In this model, each of the blocks in the netlist is a vertex, and the pins of a net form a clique. A net with |V| pins is thus modeled as a clique, and assigned edge weights 2/|V|. The total weight of all edges in the clique is |V|-1, the number of connections in a binary tree.

Another popular model for a net is the star model, where a dummy block is introduced to connect the blocks of a net. We will not focus on this model, although [1] demonstrates that these two techniques are actually equivalent.

With the netlist represented as a graph made of cliques, the AP cost function to minimize is defined as shown in Equation 1. In

this function, the square of the wirelength is minimized, which approximates the goal of minimizing wirelength.

This cost function is separable for x and y components. To minimize this cost function, we set the derivative with respect to each unknown variable to zero. This produces a set of linear equations, which can be expressed in the form shown in Equation 2. The system of equations to be solved is shown more succinctly in equations 3 to 5.

$$\varphi = \sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}\left(x_i - x_j\right)^2 + w_{ij}\left(y_i - y_j\right)^2 \quad (1)$$

Equation 1:
i and j iterate over all n blocks in the netlist
w_{ij} is the sum of the weights between blocks i and j or 0 for no edge
x_* and y_* are x and y coordinates for the fixed IOs, or unknowns for blocks to place.

$$[Q]x = c \quad (2)$$

Equation 2:
Q is an m × m "connectivity matrix," where m is the number of objects in the system whose placements are unknown
x is an m × 1 variable matrix
c is an m × 1 "anchoring" vector, which results from the fixed blocks

$$Q_{ii} = \sum_{k} w_{ik} \quad (3)$$

$$Q_{ij} = Q_{ji} = -w_{ij} \quad (4)$$

$$c_i = \sum_{z} w_{iz} \times x_z \quad (5)$$

Equations 3 – 5:
k is the set of all blocks
z is the set of all fixed blocks

This system of equations is solved using standard techniques. This paper will focus on Gaussian elimination, since this technique has the smallest computational complexity. See Table 1 for details.

Technique	Additions	Multiplications/ Divisions
Gauss-Jordan	$n^3/2$	$n^3/2$
Gaussian Elimination	$n^3/3$	$n^3/3$
$x = Q^{-1} c$	$2n^3$	$2n^3$
Cramer's Rule	$n^4/3$	$n^4/3$

Table 1: Computational Complexity of Various Solving Techniques (adapted from [2])

3. GAUSSIAN ELIMINATION USING CUDA

In the context of the Q matrix for AP, we note that pivot elements are always in place, and row swaps are never necessary during Gaussian Elimination. This is because the pivot variable essentially acts as a summation of all forces acting on a given block. In practice, pivots can only be zero if:

1. Summation of connected weights is zero
2. Zero is generated during row subtraction

Situation 1 only occurs if a block is completely unconnected. Situation 2 only occurs if some blocks form a disjoint subgraph. Since neither of these degenerate netlists is relevant, we can eliminate the row swapping portion of the Gaussian elimination operation.

Using this knowledge, we can condense the traditional Gaussian Elimination algorithm to the algorithm shown in Algorithm 1.

```
Algorithm Gaussian Elimination (Q,,x, c) {
  For row = 1 to m of Q {
    Normalize the (pivot) row so that the
     leftmost entry is one
    CUDA:  one entry per thread
    Cache the pivot column so that the value of
the subtractor for each row will be known
    CUDA:  One cached value per thread
    For each row/column combination below the
     pivot row, multiply the pivot row value by
     the cached pivot column entry and subtract
     from the entry – thus eliminating the pivot
     value
    CUDA:  One entry per thread
  }
  // Q is now upper triangular
  Perform back-substitution to find each value of
   x matrix
}
```

Algorithm 1: Gauss-Elimination

Note that the algorithm shown in Algorithm 1 relies on caching the pivot column. In order to eliminate the leftmost value of some row, we must use the correct multiplier. The multiplier is the value of the leftmost entry, however, since we cannot rely on the fact that this entry will remain in place during the processing of

subsequent row entries, we must cache this value before launching any elimination threads.

A key observation is that the Q matrix is constant for solving both the x and the y dimensions. Using this observation, we can perform the Gaussian elimination on the Q matrix, and simultaneously augment the constraint matrix for both dimensions. This effectively halves the runtime of our solution, and lends support to our choice of Gaussian elimination as our solving technique.

We implemented an AP using the C++ programming language using more primitive CUDA code wherever necessary. A screenshot of a placed netlist is shown in Figure 1

Figure 1, below. Note that we used EasyGL [3] for graphical debugging purposes.

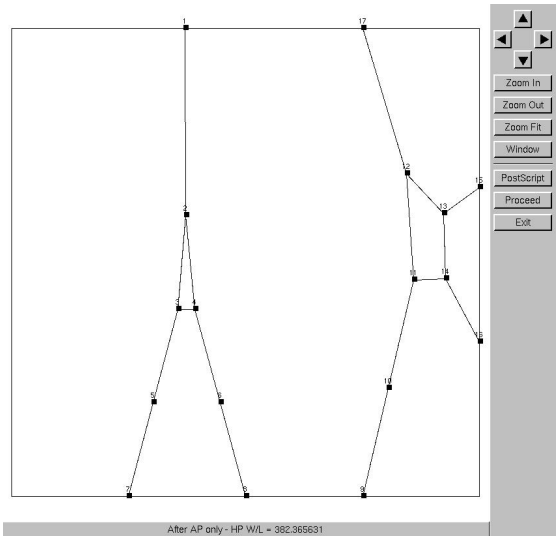

Figure 1: ***A screenshot of a netlist placement***

We had intended to create a sparse-matrix implementation and parallelize it in order to achieve further gains. However, it became apparent that non-zero entries would become zero, and, more importantly, zero-entries would become non-zero during the course of Gaussian Elimination. It was not obvious how to achieve this in the CUDA implementation since it would rely on re-allocating arrays at times.

4. RESULTS AND ANALYSIS

In order to test our placer, we needed netlists of varying sizes. We chose to implement a random circuit generator, and use it to create our test circuits. The circuit generator takes as parameters the number of blocks, nets, and IOs, and the likelihood of large multi-fanout nets. Using this generator, we created benchmarks with one, two, and four-thousand blocks, and the number of nets equal to one third the number of blocks. 35% of the nets in all netlists had high fanout.

All our testing was done on an Intel Core 2 Quad Q9550 computers at 2.83GHz, using the GTX280 NVIDIA card.

Figures 2 and 3 show the results of our initial implementation in terms of absolute runtime and percentage speedups. Note that the speedups improve dramatically as the netlists get bigger, as we see a nearly 8X speedup on the GPU implementation compared to the serial algorithm on the largest netlist containing 4000 blocks.

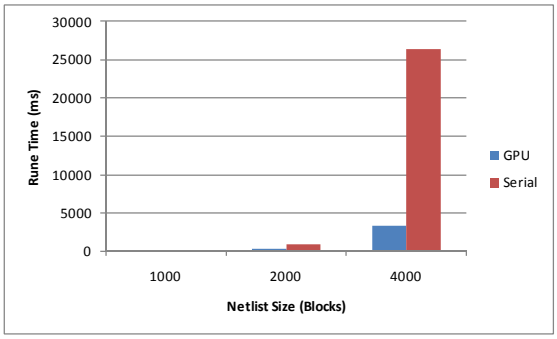

Figure 2: First-implementation GPU versus CPU

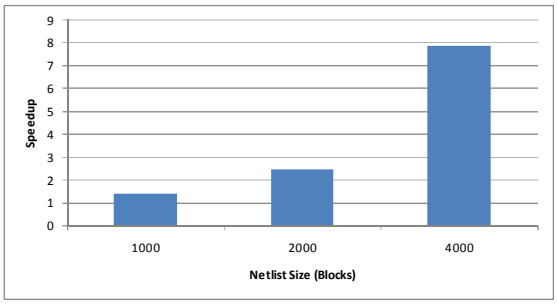

Figure 3. First-implementation GPU versus CPU (speedup)

In practice, it is difficult to calculate the optimal runtime of the algorithm on a GPU. For a rough upper bound, we calculate as follows. We know that we need precisely the following number of global memory accesses, where x is the number of blocks in the netlist.

$$\sum_{0}^{n} 2k^2 = \frac{x(x+1)(2x+1)}{3}$$

We could then take this number, divide it by 16, the number of elements retrieved in a

coalesced memory access. Furthermore, we take the number of cycles of latency for a memory access, and, assuming that one read/write request can be dispatched in each of these cycles, divide by this number to get the number of memory accesses that are not "hidden" by other accesses. This assumes a "perfect" (no synchronization) algorithm. We did not perform this calculation in our case.

We note, however, that while we do achieve good speedups, we are not approaching anything near the maximum throughput of the device. This is because we must sync after the normalization step, which is quite small relative to the main subtraction component. Furthermore, the steps themselves result in a non-optimal (un-coalesced) memory access pattern.

Using the domain-specific property, we implemented the optimization of skipping the row swapping operation. The results are shown in figures 4 and 5. These graphs show a more dramatic performance increase in the 1000 block netlist, but the performance difference diminishes as netlist size increases as gains are amortized over the longer absolute run-time. This demonstrates the fact that no real swappings were necessary in the base case, so the overhead of doing the swaps was primarily of constant time.

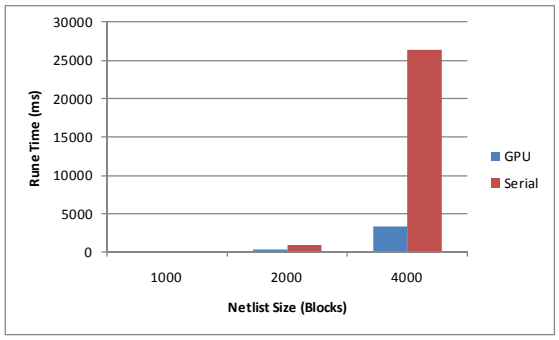

Figure 4: No row-swapping

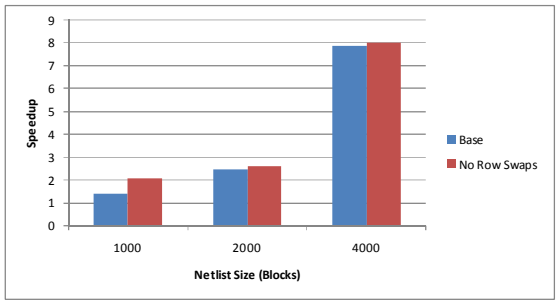

Figure 5. No row-swapping (speedup)

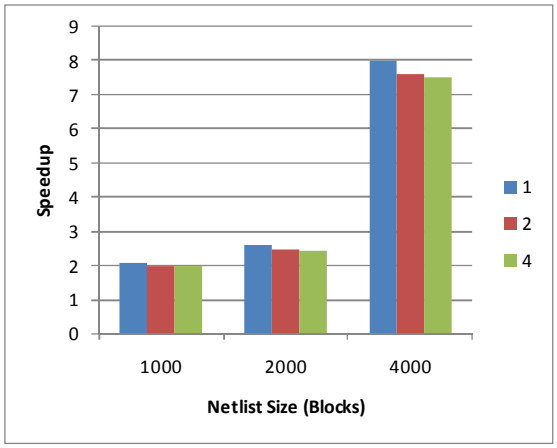

Figure 6: Number of items per thread

We also measured the effect of increasing the per-thread workload, as show in figure 6. Not surprisingly, increasing the number of items each thread processes slowed the implementation down rather than sped it up. This demonstrates the fine-grained breakdown of the problem combined with some amortized amount of memory coalescing in retrieving array elements from global memory.

5.ALGORITHM REDESIGN STUDY

The nature of the algorithm lends itself well to coalescing, since we are running inherently linear memory accesses. However, there may still be sub-optimal global memory access patterns due to misaligned starting addresses when operations are performed on an individual row. Therefore, a promising avenue of future investigation is to optimize the memory accesses in order to achieve optimal coalescing, as shown in figure 7.

0	0	x	y	z

Figure 7: Half-warp begin address for coalescing.

Generally, we want every thread to work on one array element. However, we also want a half-warp to access locations in a contiguous block of memory beginning at a "region-sized" boundary. We would therefore choose that the first half-warp should actually work on fewer than the maximum number of elements. This is so that the next half-warp can begin accessing on a region-sized boundary. In the case shown, the two zero elements would mean that the algorithm should first process node x. If the first element is on a region boundary, for example, and we are

working with floats, the region size is 128 bytes, so the first warp would access 30 values instead of 32.

Since every thread acting on an individual row must retrieve the pivot value, this is a prime candidate for use of shared memory. If we partition the problem so that all threads in a warp are working on the same row, another promising avenue of future investigation is to exploit the shared memory to retrieve the pivot value of each row. We would achieve this partitioning by padding the input array so that dummy values (e.g. zeros) are processed by the extraneous threads within the last warp working on every row. We would effectively use the shared memory broadcast mode, rather than our current use of global memory.

Using the knowledge that swaps will never be necessary, we propose an additional extension to our parallelization algorithm: Pipelined Elimination.

In this extension, we use synchronization primitives to ensure that the maximum number of processing units operate at once. We make the observation that, after the elimination has been done on the row below the current pivot, the new pivot row is ready to use. That is, the algorithm need not wait for the current pivot to pass down the matrix. We can assign some threads to do normalization while other threads are doing elimination. Futhermore, we do not need to synchronize after an elimination stage ends before the next can begin. This optimization, combined with optimal memory coalescing, should allow us to achieve near optimal results.

In a similar manner, we can parallelize the back-substitution algorithm, which is currently done serially due to the relatively small amount of time the algorithm spends performing this operation. However, the exact formulation for this is not obvious.

6. CONCLUSION

We have successfully implemented and parallelized an analytical placer on a GPU. We used domain-specific knowledge of the input problem in order to greatly optimize our algorithm. We then also give a step-by-step study of how one might create a near optimal implementation of the algorithm to work on a GPU, in order to fully exploit the graphics card capabilities.

7. REFERENCES

[1] N. Viswanathan, C. Chu. "FastPlace: Efficient Analytical Placement using Cell Shifting, Iterative Local Refinement and a Hybrid Net Model," In *Proc. of 2004 ICCAD.*

[2] H. Anton, C. Rorres. "Elementary Linear Algebra: Applications Version," 8th edition, John Wiley and Sons, 2000, pg. 468.

[3] "An Easy-to-Use Graphics Interface" http://www.eecg.toronto.edu/~vaughn/easygl/easygl.html