A Massively Parallel 2D Rectangle Placement Method

Michael Moorman and Xinlian Liu
Department of Computer Science
Hood College
Frederick, Maryland 21701, USA
mem19@hood.edu, liu@hood.edu

***Abstract*—Layout design is a frequently occurring process that often combines human and computer reasoning. Because of the combinatorial nature of the problem, solving even a small size input involves searching a prohibitively large state space. An algorithm PEMS (Pseudo-exhaustive Edge Minimizing Search) is proposed for approximating a 2D rectangle packing variant of the problem. The proposed method is inspired by MERA (Minimum Enclosing of Rectangle Area) [1] and MEGA (Minimum Enclosing Under Gravitational Attraction) [2], yet produces higher quality solutions, in terms of final space utilization. To address the performance cost, a CUDA based acceleration algorithm is developed with significant speedup.**

I. Introduction

LAYOUT Design (LD) can be informally described as the assembly of modules in a constrained space. LD problems manifest themselves in a variety of disciplines and industries including materials optimization, container cutting and packing, web design, newspaper layout, and computer circuitry hardware design [3] [4]. These problems have been known for years to be NP-Hard [3] [5]. Finding an optimal solution with brute force is not always feasible, as even a small number of modules corresponds to a massive search space. Various algorithms have been devised to approximate solutions under specific constraints.

2D rectangle packing is one variant of LD, and this variant is explored here. This branch of LD applies to a wide array of domains and the amount of available literature and solution efforts on this subject is immense [6] [7] [8].

In the world of LD there are many different fitness metrics. The goals of one layout designer may differ drastically from the next. For instance, layouts may call for high degrees of homogeneity and symmetry. Others may only be concerned about unused space or total edge distance. In general, all LD problems share the boundary constraint, where no module can be placed outside of a given bounds, and the overlap constraint, where no module can be placed such that it overlaps with another module.

A front runner in 2D rectangular packing is MERA [1] [2]. Genetic algorithm type heuristics are also effective in 2D rectangle packing, but they are viewed as complimentary to heuristics like MERA and the one proposed here, PEMS (Pseudo-exhaustive Edge Minimizing Search). It is suggested that employing genetic algorithm techniques can arrive at good solutions [9]. It is also suggested that packing methods such as simulated annealing, taboo search, and Naive Evolution show promising results as well. The problem with these approaches is the lack of proper analytical techniques that facilitates comparison with other heuristic techniques [10].

The MERA heuristic's overall effectiveness to efficiency ratio as in 2D rectangle packing is exceptional [11]. Moreover, MERA lends itself heavily to implementation on parallelized systems hardware, as is shown in [11]. The proposed method shares many architectural similarities to MERA, most notably in the strategy of exhaustive corner adjacency search.

DOI: 10.5176_2010-2283_1.4.114

II. 2D Module Packing

A 2D module packing problem is defined as given a set of N rectangles, place them on a given board such that the wasted area is minimized. For the comparisons that follow, the problem set used can be seen in Figure I below. It consists of 900 computer generated rectangular modules of varied sizes, with a bin size of 160x160 units.

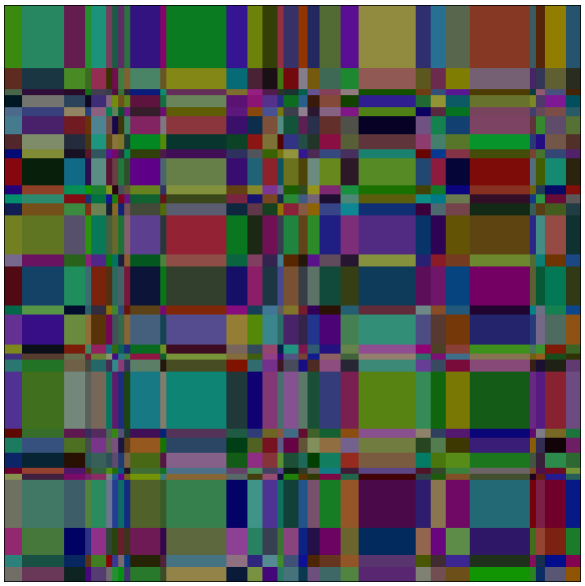


Figure I. An example problem set consisting of 900 modules in a 160x160 bin.

A more formalized list of constraints for this particular problem are shown below:

1) The bin to be packed is rectangular in shape.
2) Modules to be packed may be rotated.
3) Modules are rectangular.
4) A module may not overlap any other module or any border of the bin.

A. MERA

The MERA (Minimum Enclosing of Rectangle Area) placement algorithm provides reasonably fast, pseudo exhaustive search that produces beautiful solutions. As the name implies, the algorithm puts an emphasis on minimizing the total enclosed rectangular area of the solution for each placed module. The method is defined below in Figure II.

```
MERA( M,PM, BIN)
1  Sort M by area descending.
2  Place M[0] in corner of BIN
3  For each unplaced module K in M
4    For each placed module J in PM
5      Try placing K at every
       adjacency of every corner
       of J.
6      If can place, calculate
       the ERA resultant of this
       placement.
7    next
8    Place K at the position with
     minimum ERA
9  next
```

Figure II. MERA pseudo code.

The search strategy used by MERA is what is called a "corner adjacency search", where-by every adjacency of every corner of already placed module is considered in the search. This is shown in Figure III.

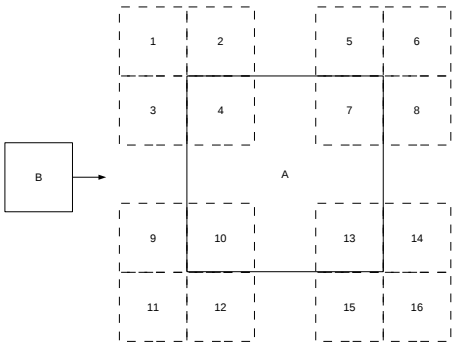


Figure III. Corner adjacency search. In this example, module B can be placed onto A in twelve different configurations, with the inside corner positions of A being invalid.

The search space grows linearly with the number of placed modules. Every placed module yields twelve possible positions for placement of other modules, with the four inside corner positions being easily ignored.

MERA tends to produce aesthetically pleasing, balanced, and symmetrical solutions. An example of a solution produced by MERA is shown in Figure IV.

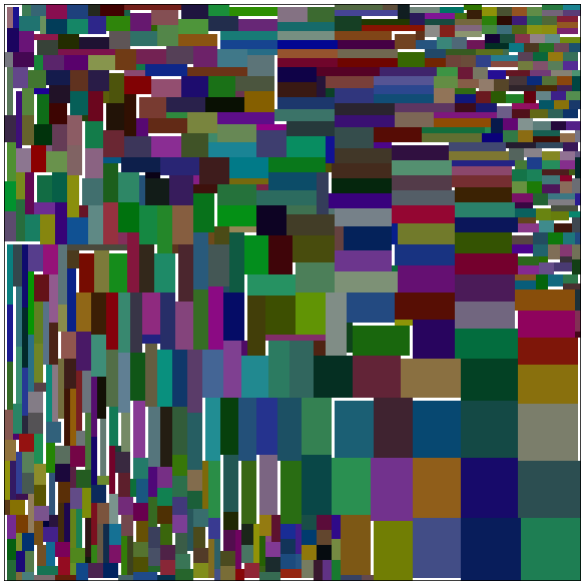


Figure IV. Example of a MERA solution of a 160x160 problem set, with 900 modules.

The source of this aesthetic stems from the criteria used for placement. MERA calculates each candidate module's total distance from every other module on the board. This process is depicted in Figure V.

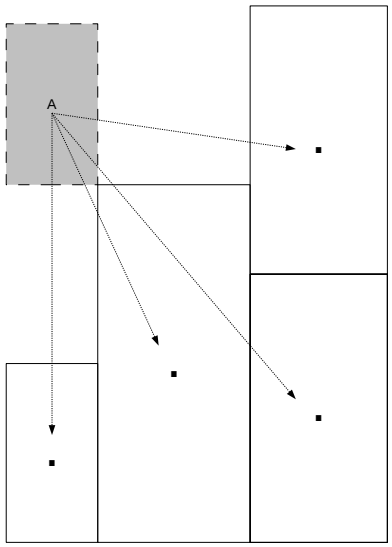


Figure V. Minimum Enclosing Rectangle

B. PEMS

The goal of PEMS (Pseudo-exhaustive Edge Minimizing Search) is to produce a layout that minimizes the total number of edges in the solution. This property has been used similarly by other heuristic methods [12]. Like MERA, PEMS employs a corner adjacency search to enumerate the search space. Unlike MERA, every unplaced module is searched against every placed module at each iteration. With MERA, only one "candidate" module is considered at every placement iteration. In other words, MERA operates with the restriction that modules must be considered for and placed in order

from largest to smallest module. PEMS does not abide by this. At every iteration, every unplaced module is considered as a candidate module. The PEMS method is defined in Figure VI below.

```
PEMS(M, PM, BIN)
1  For 1..Number of Modules:
2    For each unplaced module K in M:
3      For each placed module J in PM:
4        Try placing K at every
         adjacency of every corner
         of J.
5        If can place, calculate
         the number of connected
         edges for this
         module at this placement.
6      next
7    next
8    Place the most connected
     module/position combination.
9  next
```

Figure VI. PEMS pseudo code.

At every iteration, every unplaced module is tested at every possible position in the corner adjacency search. Module position pairings are evaluated on the criteria of maximum connectiveness, as is depicted in Figure VII.

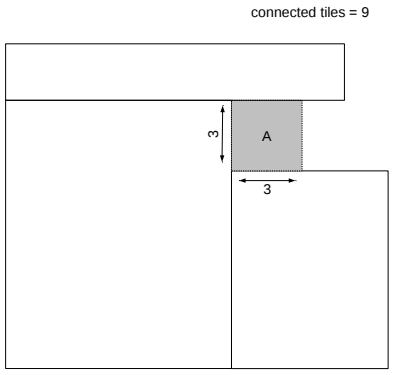


Figure VII. Edge Minimizing strategy.

The exhaustive search of PEMS produces a very high quality solution. Figure VIII shows the resulting placement of the same problem set used earlier for MERA.

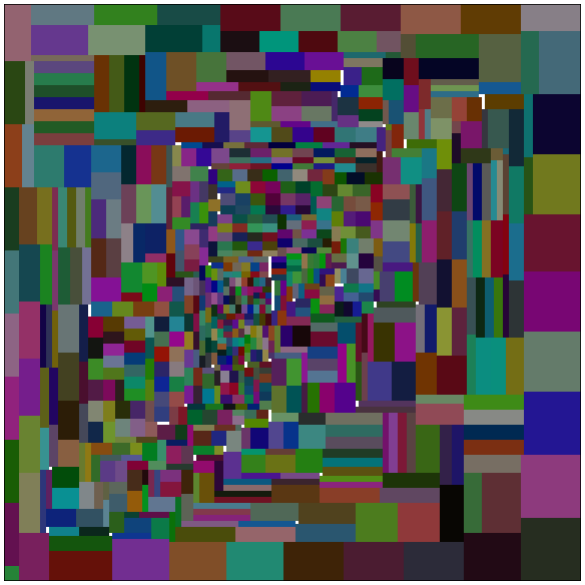


Figure VIII. An example of a PEMS solution to a 160x160 unit problem set with 900 modules. Compared with Figure IV, less white space represents higher utilization of solution area.

Though the end result is pleasing, the amount of computational work needed by PEMS is likely unacceptable for traditional platforms. One way to mitigate this problem is to implement the heuristic on a massively parallel architecture, where the time needed to perform said computations can be drastically reduced. NVIDIA's CUDA (Compute Unified Device Architecture) was selected for this purpose.

NVIDIA's (CUDA) [13] is a software platform for massively parallel high performance computing on their GPUs. CUDA implements an extremely fine grained paradigm of parallelism through use of the single instruction, multiple thread model, or SIMT [14]. In this model, millions of lightweight threads are employed to execute a common function and each thread operates on different data. Although the potential for speedup is great, designing algorithms that achieve optimal hardware performance on the GPU is a difficult task, since memory latency is rather high compared to a traditional CPU.

Parallelizing PEMS into SIMT (Single Instruction Multiple Thread) architecture can be accomplished by dividing the algorithm into two primary kernels; module evaluation and module reduction/placement. The flow of kernel invocations takes the form of:

```
CUDAPEMS(M,BIN)
1  initialize GPU resources
2  place M[0] in a corner
3  for 2 to len(M)
4    invoke module evaluation kernel
5    invoke module placement kernel
6  next
```

Figure IX. Parallel PEMS pseudo code.

The module evaluation phase divides the computational work by allocating N CUDA thread blocks, where N is the total number of modules - 1. Within each thread block, 96 CUDA threads share the responsibility of performing a corner adjacency search against each already placed module. If a block's module has already been placed, it returns immediately. This kernel is shown in Figure X.

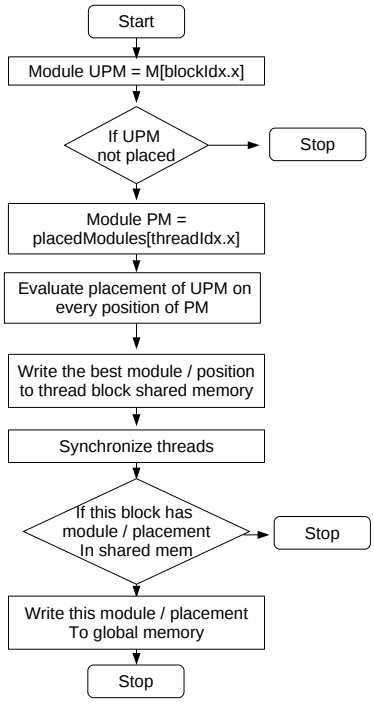


Figure X. Evaluation kernel flow (UPM is an "Unplaced Module", PM is a "Placed Module")

Phase one ultimately produces n evaluation data points. Each evaluation represents the best placement for a given module. In the case of a module already being placed, a sentinel value occupies the evaluation structure.

Phase two, the placement kernel, performs a reduction of the evaluations found in phase one. It simply chooses the evaluation with the highest fitness and places it onto the board. No speed up is actually realized as a result of implementing this placement phase as a GPU kernel; however, it does avoid the cumbersome task of having to copy data back and forth between host and GPU memory [14].

C. Analysis of Algorithmic Complexities

MERA and PEMS share similar asymptotic upper bounds of $O(n^3)$, where n is the total number of modules in the problem set. The ERA calculation step of MERA sums the distances of every already placed module to a candidate module, and it is bound by $O(n)$. This is multiplied by a $O(n^2)$ bound resultant from a doubly nested loop. PEMS is bound by $O(n^3)$ due to a triply nested loop. The calculation step of PEMS is constant with respect to n.

Although PEMS and MERA share asymptotically similar upper bounds, their practical run times differ heavily, with MERA greatly outperforming PEMS by a constant factor. This is the case because MERA forgoes the ERA calculation step ($O(n)$ cost) when a shape placement is not valid for given iteration. Thus, the performance gap between the two methods relates directly $(1/P)$ to the probability that a single MERA iteration will result in a valid shape placement. P was observed to be between .01 and .03 for the problem sets used in this study.

D. Implementation Notes

Results for this study were gathered from C/C++ implementations of MERA, PEMS, and a CUDA implementation of PEMS. The specifications for the hardware used are shown in Table I.

CPU	Intel Core i7 CPU 950 @ 3.06 GHz
Memory	12 GB
GPU 1	GeForce GT 430 w/ 1GB RAM
GPU 2	Tesla C2050 w/ 2.5GB RAM

TABLE I
TEST SYSTEM HARDWARE

The methodology used to compare space utilization between PEMS and MERA was to repeatedly generate solutions for randomly generated problem sets. These problem sets consist of a list of modules that have a total area equal to the bin size. The ideal case is a solution that successfully places all of the modules in a problem, equating to 100 percent utilization. Two additional constraints placed on the generation of random problem sets were the specification of the minimum length for a module dimension and the total number of modules in problem. The total number of modules in problem sets are always configured as a perfect square of some number i.e. 4*4=16 5*5=25 6*6=36. The purpose of the minimum length constraint is to make problem sets more difficult to reassemble. For example, it is trivial to pack rectangles with dimensions of 1x1.

In this study, PEMS CUDAPEMS, and MERA are implemented such that each method shares common routines and data structures. All three implementations use an array of bits (bit matrix) to represent the solution space. The three methods also share a common predicate routine "canPlace", which performs a naive overlap and boundary constraint check on the solution space to determine the validity of a rectangle placement. Each implementation also shares a common method "place", which effectively places a rectangle onto the solution space.

All tests were conducted on an Intel based server running the x86_64 version of Fedora 14 and the NVIDIA CUDA driver and toolkit version 4.0. Only the Tesla card was used for this study. The additional GeForce 430 was used for console display only.

III. RESULTS

Figure XI compares solution space utilization between PEMS and MERA for many instances of automatically generated problem sets.

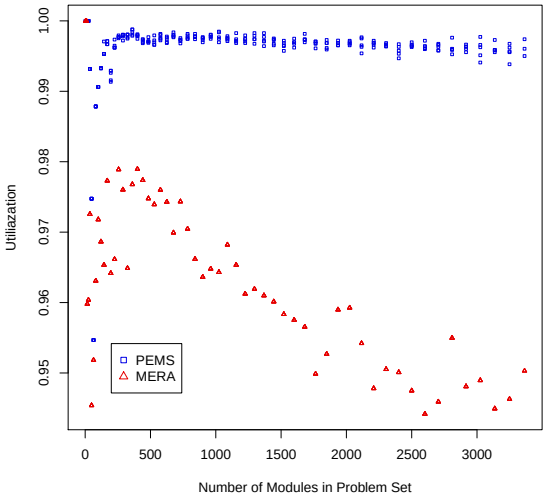


Figure XI. PEMS and MERA solution space utilization (solution quality) using generated problem sets for 160x160 unit bin sizes, with a minimum module dimension constraint of 2.

PEMS yielded a higher quality solution in almost every recorded test. Exceptions to this occurred only in problem sets with less than 81 modules. PEMS' exhaustive evaluation of module placement based on edge minimizing produces more tightly packed solutions as compared to MERA.

Figure XII compares performance between CUDAPEMS and MERA for a varying number of total modules. After accelerating PEMS with CUDA, the total run times are very similar to the serial implementation of MERA. As the problem set size increases , the CUDA PEMS implementation begins to outperform MERA. The cause of this phenomenon is likely attributed to the constant costs associated with using the GPU, e.g the initial time required to allocate and copy memory onto the GPU device.

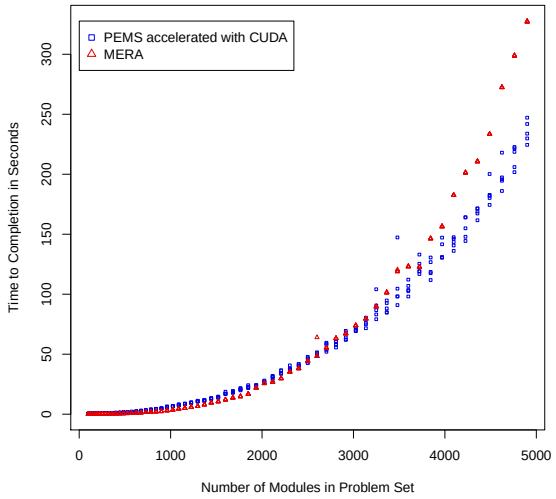


Figure XII. Performance of CUDA accelerated PEMS vs MERA using generated problem sets for 160x160 unit bin sizes, with a minimum module dimension constraint of 2.

Figure XIII compares performance between CUDAPEMS and PEMS for a varying number of modules. The parallel implementation vastly outperforms the serial version, with a maximum recorded speedup of 130x.

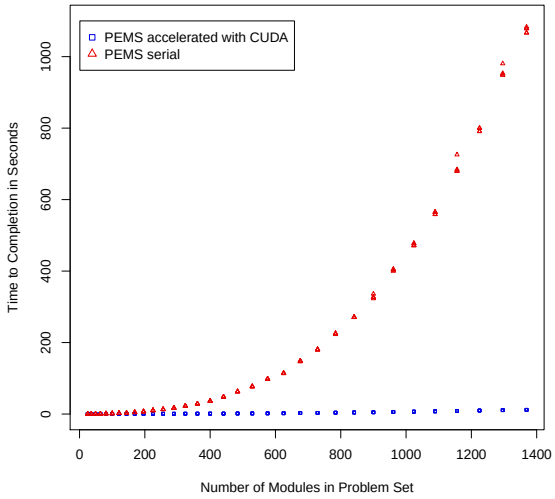


Figure XIII. PEMS parallel vs PEMS serial performance using generated problem sets for 160x160 unit bin sizes, with a minimum module dimension constraint of 2.

The CUDA version of PEMS outperforms its serial predecessor by a wide margin due to massive parallelization. At each iteration of CUDAPEMS, A CUDA thread block is assigned to each unplaced module. Each thread block consists of 1 to 1024 threads; this study found the optimal number of threads per block to be 96. To exemplify this parallelism, consider a scenario where 256 modules have already been placed on the board and 256 modules remain to be placed. In this case, $256 * 96 = 24576$ lightweight CUDA threads are employed to perform an iteration of PEMS. The 448 CUDA cores on the Tesla C2050 GPU [15] dispatch these threads simultaneously.

Optimization played a significant role in accelerating PEMS on CUDA. Early on in the study, a matrix of 32 bit integers was used as the underlying solution data structure for all three implementations. Only a 5-20 times speed up was demonstrated in that rendition. Total global memory required for the rectilinear representation in this case was $160 * 160 * 32 = 819200$ bits, or $819200/8 = 102400$ bytes. Global memory accesses are expensive in CUDA [14], even though best efforts are made by the architecture to coalesce and optimize these accesses among thread blocks [14]. Accesses to this data structure occur often by each thread, which likely bottle necked the flow of execution in this early implementation. Replacing the integer matrix with a bit matrix was a great boon for the acceleration of PEMS. The total memory footprint for the solution representation was reduced by a factor of 32, yielding $160 * 160 = 25600$ bits, or $25600/8 = 3200$ bytes. Total execution time was faster by a factor of at least 4 after implementing the bit matrix.

Another optimization was the removal of all unnecessary registers for the kernels. This allows thread blocks to achieve a higher occupancy level in the GPU, thus expanding the run time's ability to perform thread switching, a critical strategy that the GPU uses to hide high-latency operations.

A third optimization method used was the strategy of using thread block shared memory to reduce memory latency in the evaluation kernel. In CUDA, shared memory is a specialized type of storage that is shared among threads in a thread block [14]. Accesses to shared memory are much faster than accesses to global memory [14]. This strategy involved threads cooperatively copying the entire solution data structure into thread blocks at the beginning of a PEMS evaluation iteration. The idea was to coalesce accesses to global

memory as much as possible. A small speed up was realized as a result of this optimization attempt, likely due to the fact that a large percentage of the thread blocks short circuit their execution in the case of an unplacable module. These blocks tend to have little or no accesses to shared memory, but would still incur the cost of copying in the entire representation. Further investigation into this avenue of optimization could result in an even higher speed up.

All source code for this study is available to the public under the GPLv3 open source license at http://sourceforge.net/p/pemsproject/.

IV. Conclusions

The PEMS method produces high quality solutions with regard to space utilization for the problem sets used in this study. In every case tested, PEMS produced more compacted solutions than MERA. PEMS is likely prohibitively computationally expensive for implementation on traditional systems. PEMS implemented on CUDA is shown to yield up to a 130 times speed up over the serial version. Though not implemented here, it is likely that the parallelized version of MERA outperforms CUDA PEMS. Nevertheless, PEMS produces higher utilization quality solutions than MERA regardless of the implementation. The pseudo-exhaustive search employed by PEMS is computationally expensive, but implementation of PEMS on CUDA mitigates these costs to an acceptable level. Finally, applications that require high utilization quality in 2D rectangular layout design could benefit significantly from the use of PEMS, especially when the number of modules under consideration is numbering in the low thousands or less.

Acknowledgment

The NVIDIA CUDA Center at Hood College is acknowledged for providing us with computer hardware.

References

[1] K. H. M. H. I. Abdul-Rahim Ahmad, Otman A. Basir, "Improved placement algorithm for layout optimization."
[2] K. H. Abdul-Rahim Ahmad, Muhammad Hasan Imam, "A placement algorithm for efficient generation of superior decision alternatives in layout design," 2004.
[3] H. M. Shigetoshi Nakatari, Kunihiro Fujiyoshi and Y. Kajitani, "Module placement on bsg-structure and ic layout applications," 1996.
[4] S. N. Y. K. Hiroshi Murata, Kunihiro Fujiyoshi, "Rectangle-packing-based module placement," 1995.
[5] S. Jakobs, "On genetic algorithms for packing of polygons," 1999.
[6] G. Zhang, "A 3-approximation algorithm for two dimensional bin packing." 2005.
[7] R. V. S. R. Harreni, "Packing rectangles into 2 opt bins using rotations."
[8] R. S.-O. K. Jansen, "New approximability results for 2-dimensional packing problems."
[9] A. L. Ping Chen Zhaohuri Fu and B. Rodrigues, "Two-dimensional packing for irregular shaped objects," 2003.
[10] T. B. Hopper, E., "An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem," 2001.
[11] I. W. D. Amina Y. Maarouf, "High speed hardware implementation of a hueristic 2d rectangle placement algorithm," 2006.
[12] M. S. W. H. Duanbing Chen, Yan Fu, "A quasi-human heuristic algorithm for the 2d rectangular strip packing problem." 2008.
[13] (2011). [Online]. Available: http://www.nvidia.com/object/cuda_home.html
[14] (2011). [Online]. Available: http://developer.download.nvidia.com/compute/cuda/
[15] (2011). [Online]. Available: http://www.nvidia.com/object/cuda_home.html