Electronics and Computer Science
Faculty of Physical and Applied Sciences
University of Southampton

Kim Svensson
April 28, 2013

Combinatorial optimization of NP-hard problems using dynamic GPU programming

Project supervisor: Dr. Sarvapali D. Ramchurn
Second examiner: Dr Srinandan Dasmahapatra

A project progress report submitted for the award of
MEng Computer Science

Abstract

Solving problems that have an exponential growth in time and space have always been a difficulty as the advancement in computing speed only allows for a small increase of the problem size. The advent of parallel computing using a GPU have enabled a speedup when it comes to problems which have some data and computation independent parts. The purpose of this project is to develop and evaluate dynamic programming algorithms suitable for parallel computation to find an optimal solution in the domain of complete coalition formation and combinatorial auctions. The heterogeneous algorithm is run using the Nvidia CUDA framework and preliminary result shows a substantial difference and is publishable results.

Contents

0.1 Introduction

With NP-hard problems and algorithms with a complexity class of P or greater, a doubling in CPU speed allows only for a small change in sample size due to the nature of such problems, as execution speed and sample size of an algorithm is mutually exclusive. As such, computer scientists have to find other ways of computing difficult problems on large sets of data. The traditional approach has been to increase the CPU speed and the number of CPUs. However, such implementation is often costly and suffers from latency and architectural difficulties. As the GPU function like a single instruction multiple data (SIMD) architecture with up to thousands of programmable cores, researchers have used GPUs to run concurrent algorithms faster than the equivalent sequential algorithms designed for CPUs. Especially since the development of NVIDIA CUDA and OpenCL implementation in both hardware and software. This advance in computer science can allow for large computations that were previously unfeasible by leveraging the parallel nature of GPUs. In combinatorial optimization the problem consist of finding the optimal solution for a problem, which for a large set of data can be unfeasable if the problem is NP-hard.

0.2 Background

0.2.1 Combinatorial Auctions

Sometimes auctioneers may sell out several of their assets at the same time such as the government selling parts of the radio spectrum. The bidder may then make offers on subsets of the assets. The problem occurs when there is a large amount of assets being sold while the amount of bids are non-trivial. As for each bid, it have to made sure that no other bid that only accuires a subset of the original bid is greater. The goal is therefore to calculate which set of pairwise disjoint bids gives the most revenue to the auctioneer.

Consider that the auctioneer is selling the assets $\clubsuit$, $\diamondsuit$ and $\heartsuit$. The bids collected are as following:

Set	Bid
$\clubsuit$	3
$\diamondsuit$	2
$\heartsuit$	6
$\clubsuit, \diamondsuit$	7
$\clubsuit, \heartsuit$	7
$\diamondsuit, \heartsuit$	4
$\clubsuit, \diamondsuit, \heartsuit$	6

Table 1: This table shows some data

The winning bid configuration would be the following set:$\{\{\heartsuit\}, \{\clubsuit, \diamondsuit\}\}$ with a total sum of $7\{\clubsuit, \diamondsuit\} + 6\{\heartsuit\} = 13$ as it is the maximum sum of all combinations of disjoint sets. Where all complete disjoints sets are:

Complete disjoint set	Sum
$\{\{\heartsuit\}, \{\clubsuit, \diamondsuit\}\}$	13
$\{\{\diamondsuit\}, \{\clubsuit, \heartsuit\}\}$	9
$\{\{\clubsuit\}, \{\diamondsuit, \heartsuit\}\}$	7
$\{\{\clubsuit, \diamondsuit, \heartsuit\}\}$	6
$\{\{\clubsuit\}, \{\diamondsuit\}, \{\heartsuit\}\}$	11

Table 2: This table shows some data

The way you calculate the optimal solution for combinatorial auctions is by given any set, find the greates sum for the whole collection of pairwise disjoint subsets of that set and evaluate if it is greater than the set itself. This procedure is done for all sets to finally find a final solution that is the optimal one which give the largest revenue to the auctioneer.

0.2.2 Complete Coalition Formation

Coalition structure generation is a complete analogy to combinatorial auction but from another perspective. It can be translated in such that the assets become agents, an agent can be described as a entity that act on its own, be it a person or a node in a network. A set of assets subjected to a bid is called a coalition formation, in short when two or more agents form a coalition to be regarded as one entity. The auction bid will be translated to coalition value, which means the value gained when two ore more agents form a coalition. Analogous to the previous example, given N agents, generate a set of coalition formations such that the outcome is optimal in regards to the coalition formation value. This however will be an extreme challenge as the number of possible partitions grows exponential with n, specifically $O(n^n)$ [3]. One algorithm to solve this problem in $O(n^3)$ time is called the Dynamic Programming(DP) algorithm [4] described in algorithm 1.

It works with the dynamic programming bottom-up approach where you start at sets with cardinality 2 and see if the sum of the pair-wise disjoint subsets value is greater than the original set. If it is assign the new sum to that set. With this principle, all sets with the greatest sum will be carried forward when the cardinality of the sets increases. This algorithm is deterministic, meaning that even if there is only one bid, the computation will take the same time as 1000 bids for a given set of agents.

Algorithm 1 Dynamic Programming algorithm

INPUT: $b(C)$ the bids for all sets $C \subseteq A$ where A is the set of assets.
VARIABLES: f a function that maps from a subset $C \subseteq A$ to a value
O a function that maps from a subset $C \subseteq A$ to the subset that maximize the value for set C.

```
1: for all $x \in A$, do $f(\{x\}) := b(\{x\}), O\{x\} := \{x\}$ end for
2: for $i := 2$ to $n$ do
3:     for all $C \subseteq A : |C| == i$ do
4:         $f(C) := max\{f(C \setminus C') + f(C') : C' \subseteq C \wedge 1 \leq |C'| \leq \frac{|C|}{2}\}$
5:         if $f(C) \geq b(C)$ then $O(C) := C^*$ Where $C^*$ maximizes right hand side of
           line 4 end if
6:         if $f(C) < b(C)$ then $f(C) := b(C) \wedge O(C) := C$ end if
7:     end for
8: end for
```

0.2.3 NVIDIA CUDA

CUDA (formerly known as Compute Unified Device Architecture) is a framework by NVIDIA equivalent to OpenCL which allows for C,C++ and FORTRAN code to be compiled and run on its Graphics Processing Units (GPUs). As GPUs are many-core processors designed to handle large amount of independent data when running a graphics program, it is advantageous to utilize such architecture to handle general purpose algorithms to solve complex problems with parallel nature. This architecture is described as Single Instruction Multiple Threads (SIMT) which could be seen similar to the Single instruction, multiple data (SIMD) architecture. The fundamental difference is that with SIMT, branch divergence is allowed.

The CUDA architecture consist of a scaled array of Streaming Multiprocessors (SM), where each SM contains from 8 cores to 192 programmable cores depending on the GPU version. Where one GPU can have several SMs. When launching a kernel which can be seen as a function call, the threads assigned are divided up into blocks of threads. Each block gets assigned to an SM to be processed resulting in two blocks from the same kernel can be running concurrently on two different SMs. If there is no SM freely available the block will be put into a queue. The next thing to note is that each block is divided into warps of 32 threads which is scheduled by the warp scheduler. The reason is to hide memory read and write latency as while one warp is blocked waiting for data, another warp can run its instructions. Therefore the size of each threadblock should be a multiple of 32 threads to allow full utilisation of the warp.

0.3 Literature review

With Combinatorial auctions and multi-agent coalition formation being combinatorial optimization problem, an literature review was conducted to find relevant publications in the area.

The DP algorithm have been the best algorithm to solve coalition structur generation problem in deterministic time until the advent of the Improved Dynamic Programming(IDP) algorithm by Rahwan, T. and Jennings, N.R. [3]. It uses a simple, yet novel way to determine which coalition structures should be examined, reducing the edges traveled in the directed graph, which allows for a shorter runtime yet conserves the optimality of the DP algorithm.

Solving the incomplete combinatoral auctions using non-determenistic algorithms trade-offs have to be made. Constructing the bids in a tree structure is common, but how the heuristics is formed to traverse and search the tree for the optimal solution is often quite different. One algorithm called CPLEX is using a heuristic A* search algorithm which explores the most promising branch first. This however suffers from exponential memory requirement, however if the problem can fit into memory it finds a solution faster on hard problems than the purposed CABOB algorithm. CABOB uses a depth-first branch-and-bound heuristic which only require the current branch investigated to be held in the memory. This gives the algorithm a linear memory requirement with regards to the depth of the branch. However as this is an depth-first algorithm, the worst case scinario occurs with a large depth of the tree which is not precent in the CPLEX algorithm. However for a larger problem the CABOB algorithm is most suited as CPLEX require a vast amount of memory to function [5].

One of the few papers regarding a distributed or concurently run algorithm for coalition structure generation used 14 dualcore workstations [2]. Their solution only had a runtime from 11% to 4% of the centralised variant. However as the computers comunicated over the ethernet, latancies may vastly affect the runtime, which could be improved by running it concurrently on a GPU.

Using the GPU together with dynamic programming have been used before to solve similar combinatorial optimization problems. Boyer, V et al successfully implemented and solved the knapsack problem with a great reduction in run-time [1]. They also introduced ways of reducing memory occupancy which enable computation of much larger data sets as well as reducing the bandwidth utilized allowing for a faster processing speed. This is one of the most important aspects of GPU programming as you are first limited to the amount of memmory the GPU have, but also in order to increase the effective bandwidth of the algorithm.

0.4 Work

My work has been written in ANSI C, the reason is large amount of experience in the language but also the fact that one of the native languages supported by the CUDA framework is the C programming language. To this point I have implemented the algorithm called the Dynamic Programming (DP) algorithm by Rothkopf, M.H et al [4] which is used to solve combinatorial auctions, but it can also be used for complete coalition formation as described before. It was first implemented as a sequential algorithm to allow for a baseline to compare execution speed by. But also to gain knowledge about the algorithm to allow easier transition towards a parallel version. The algorithm I have derived that run on the GPU is represented by the max function at line 4 of the DP algorithm, which consists of generating pairwise disjoint subsets from a given set, and then calculated the value the disjoint sets give and compare it to the original set.

0.4.1 Algorithm

Input

f	The array which holds the bids
O	The step array that holds set configurations
Φ	A reference to an maximum value bucket unique for each kernel
C	The set to do subset construction on
Ψ	The maximum subset construction index

Variables

```
$\lambda$                                   How many bids should be evaluated per thread
$\Upsilon$                                  A shared array containing threads maximum bid values
$\Delta$                                    A shared array containing threads maximum bid subsets
$\upsilon$                                  A local array containing one of the threads bid value
$C'$                                        A local array containing one of the threads bid subsets
$bid = blockIdx.x$                          Which block the threads belong to
$bdim = blockdim.x$                         How many threads inside the block
$tid = threadIdx.x$                         The thread index inside the block
$\psi := \lambda * (tid + bdim * bid)$      Initial subset construction index
**Start of algorithm**
1:  $\Delta_{tid} := \Upsilon_{tid} := 0$                        Initialize values to zero
2:  **if** $\psi \leq \Psi$ **then**
3:      **for** $i := 1$ to $\lambda$ **do**
4:          $C'_i := (\neg C + \psi) \bigcap C$                  Integer subset construction
5:          **if** $|C'_i| \leq \frac{|C|}{2} \wedge \psi \leq \Psi$ **then**
6:              $\upsilon_i := f[C/C'_i] + f[C'_i]$            Set the split subset coalition value
7:          **end if**
8:          **if** $\upsilon_i > \Upsilon_{tid}$ **then**
9:              $\Upsilon_{tid} := \upsilon_i$                 Set value if $\upsilon_{tid}$ is greatest so far
10:             $\Delta_{tid} := C'_i$                         Set subset if $\upsilon_{tid}$ is greatest so far
11:         **end if**
12:         $\psi := \psi + 1$                                  Increment by one
13:     **end for**
14: **end if**
15: *syncthreads*                                              make sure all threads have finished
16: **for** $i := bdim >> 1$ to $0$ **do**
17:     **if** $tid < i$ **then**
18:         **if** $\Upsilon_{tid} \leq \Upsilon_{tid+i}$ **then**
19:             $\Delta_{tid} := \Delta_{tid+i}$
20:             $\Upsilon_{tid} := \Upsilon_{tid+i}$
21:         **end if**
22:     **end if**
23:     *syncthreads*                                          make sure all threads have finished
24:     $i := i >> 1$                                           bit-shift i once to the right
25: **end for**
26: **if** $tid == 0$ **then**
27:     **if**$\Upsilon_0 == 0$ **then** *exit* **end if**          exit if the bid is zero
28:     **if**$f_C > \Upsilon_0$ **then** *exit* **end if**         exit if the bid is less than for C
29:     **if** $atomicMax(\Phi, \Upsilon_0) < \Upsilon_0$ **then**
30:         $O_C = \Delta_0$                       Sets the current optimal step in global memory
```

```
31:         $f_C = \Upsilon_0$                    Sets the current optimal value in global memory
32:         $threadfence$                              make sure all threads see the changes
33:     **end if**
34: **end if**
35: **return** $O, f$
```

Line 4 represent how the subset construction is made, it starts by taking the inverse of the integer C, add together the subset construction index ψ, and finally apply a bit-wise AND with C to get a guaranteed subset of C. This implementation suffers though from generating subsets of greater cardinality needed, but testing have shown that the instruction overhead for generating these unwanted subsets only account for 18% to 8% overhead as shown by figure 2. The reason for representing a set as an integer was inspired by [knappsack] due to the substantial lower memmory reqired to store a set as an integer rather than an array of integers.
The code from 16 to 25 details how to do a reduction among all threads inside a block to find the maximum value from all of the subset constructions the threads have previously done. Where for each cycle of the loop, half of the threads will compare its value to a corresponding inactive thread and set its value and subset to whichever value is largest. Then it will reduce the number of active threads by half and continues until there is no more threads to be active. The loop will run $log(n)$ times where n is the number of threads in the block.
Lines 26 to 34 represent how each block of a kernel update their maximum value and subset to the global memory for all the consecutive kernels to see. Beginning at line 26, here only the thread with an id of zero will pass in order to reduce the memory bandwidth but also as there is only need for one thread to update the global memory. Line 29 ensures that only a bigger value can be set by first querying the bucket for this kernel, if it is bigger it can continue to set the global memory. The reason for the atomic function is due to that blocks from the same kernel can and will run concurrently, therefore it will cause race conditions.

This algorithm have been derived through several iterations to find the right balance between amount of registers used, the number of times memory is accesed and how many coalitions should each thread evaluate.

0.5 Result

0.5.1 Experimental Setup

The evaluation of the CUDA DP algorithm is conducted on a NVIDIA GTX 660TI GPU with 7 streaming processors for a total of 1344 cores and memory capacity of 2GB, the maximum blocksize is set to 256 threads to allow maximum occupancy on the GPU. The CPU DP algorithm is run single threaded on a INTEL XEON W3520 with a clockspeed of 2.67GHz with 32KB L1, 256KB L2 cache. The way the data is structured and stored is identical between both implementations and the time is measured more than once. The early test result show an graph of the logarithmic runtime in base 10 for an increasing number of agents.

0.5.2 Evaluation

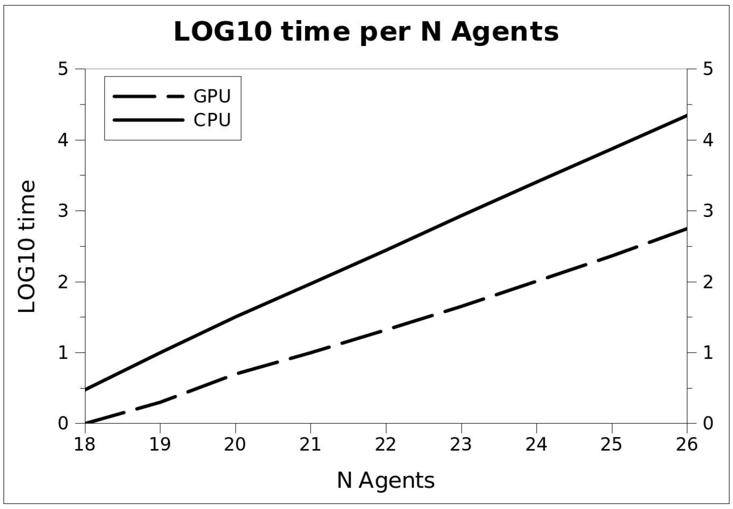


Figure 1: LOG10 run-time per N Agents

As it stands now, the algorithm that is running using the CUDA framework have an substantial performance increase as shown in figure 1. As seen from the figure, the slope of the CPU and GPU lines diverge, indicating an different growth rate for an increase of n, thus having a different time complexity. Empirical measurement show that for each increase in n, the run-time for the CPU increase by three. While the GPU bound algorithm starts its growth at 2 and the growth increases to 2.42 at 25 agents.

Figure 2 is the evaluation of line 4 on how many subsets generated that invalidates the condition $|C'_i| \leq \frac{|C|}{2}$ where $C'_i \subset C$. This is a measure that describe the instruction overhead for that particular part of the algorithm, and as it stands now with the algorithm being limited by memory latency due to random access, this instruction overhead can be considered negligible. This measure however does not take into account any duplicate subsets that are occasionally generated.

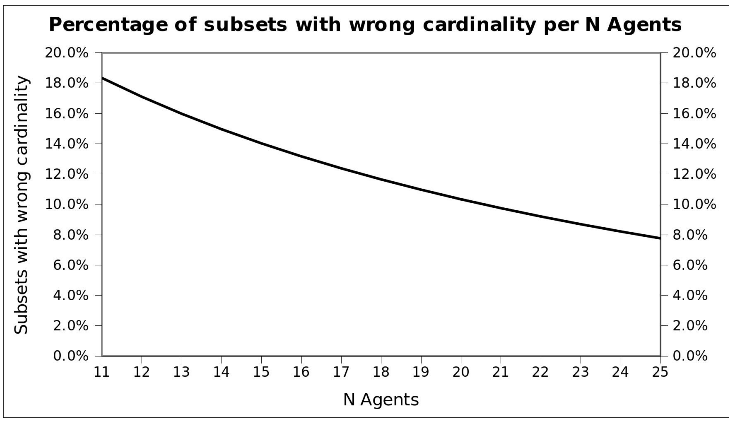


Figure 2: Percentage of generated subset of wrong cardinality

0.6 Future work

As the closest algorithm that is regarded faster than the DP algorithm is the IDP algorithm [3] that have shown a promising reduction in evaluated coalitions. The next step would be to translate my current work to an algorithm that utilizes the IDP technique. This will be both done for the CPU and GPU implementation and later on compared and evaluated to the DP algorithm. However as it is fairly similar to the DP algorithm, implementation should be trivial and further goals need to be determined.
As the current work only works for a small number of agents but the complete set of coalitions, the next step would be to implement an algorithm to solve realistic combinatorial auctions with an relative large number of agents and low number of coalition structures. As the Cabob algorithm uses a branch by branch evaluation heuristic that are independant of each other. Such algorithm would be more than suitable to be implemented to run concurrently.
Due to the promising results gained, a further more robust testing facility will be programmed to ensure the absolute correctness of my algorithm. If the results stays promising a paper will be written to try and publish the results.

Bibliography

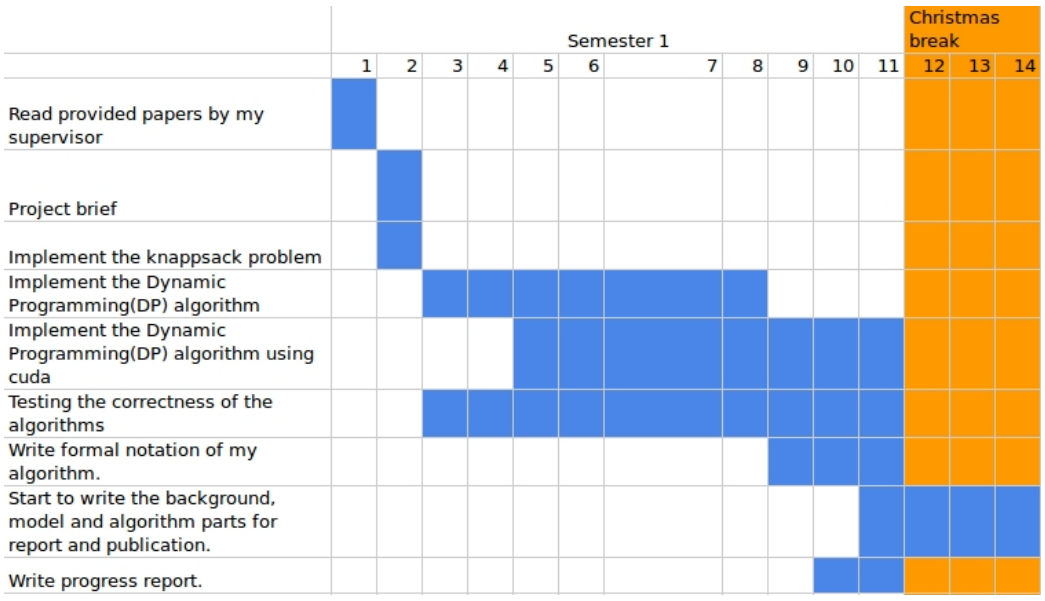
[1] V. Boyer, D. El Baz, and M. Elkihel. Solving knapsack problems on gpu. *Computers & Operations Research*, 39(1):42–47, 2012.

[2] T. Michalak, J. Sroka, T. Rahwan, M. Wooldridge, P. McBurney, and N.R. Jennings. A distributed algorithm for anytime coalition structure generation. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 1007–1014. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[3] T. Rahwan and N.R. Jennings. An improved dynamic programming algorithm for coalition structure generation. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3*, pages 1417–1420. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

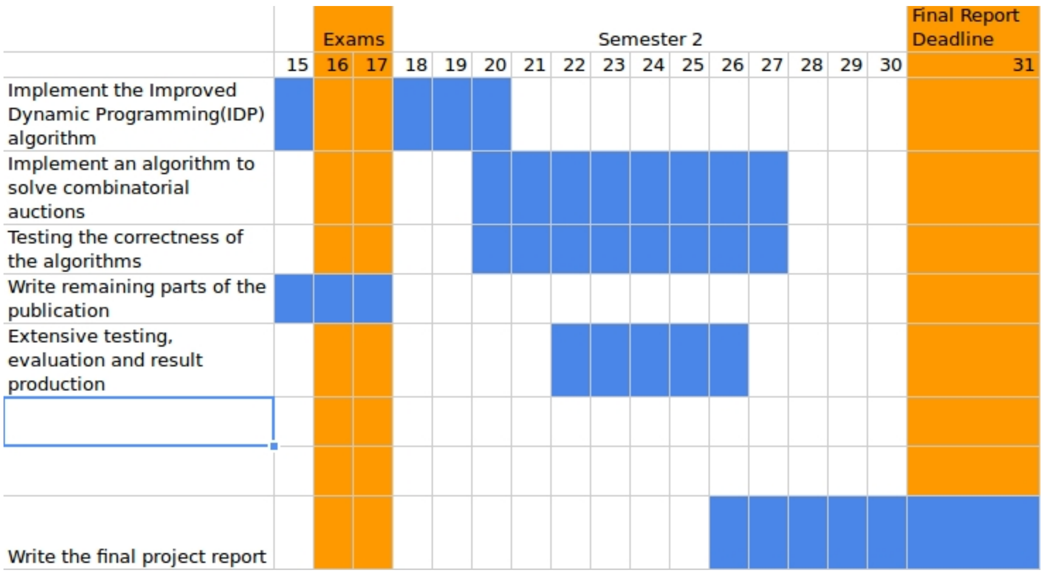
[4] M.H. Rothkopf, A. Pekeč, and R.M. Harstad. Computationally manageable combinational auctions. *Management science*, 44(8):1131–1147, 1998.

[5] T. Sandholm. Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence*, 135(1):1–54, 2002.



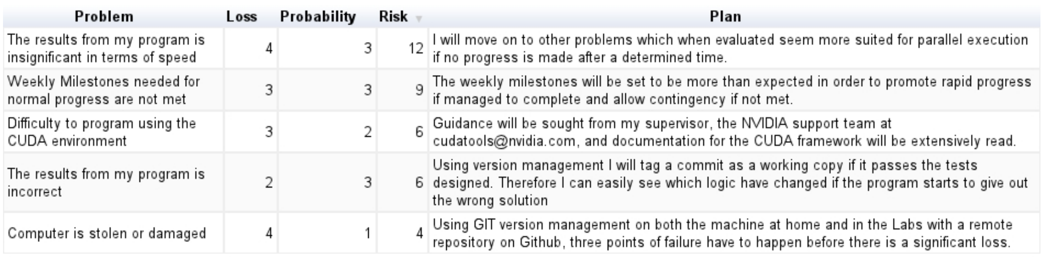
	Semester 1											Christmas break		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Read provided papers by my supervisor	🟦											🟧	🟧	🟧
Project brief		🟦										🟧	🟧	🟧
Implement the knappsack problem		🟦										🟧	🟧	🟧
Implement the Dynamic Programming(DP) algorithm			🟦	🟦	🟦	🟦	🟦	🟦				🟧	🟧	🟧
Implement the Dynamic Programming(DP) algorithm using cuda					🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟧	🟧	🟧
Testing the correctness of the algorithms			🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟧	🟧	🟧
Write formal notation of my algorithm.									🟦	🟦	🟦	🟧	🟧	🟧
Start to write the background, model and algorithm parts for report and publication.											🟦	🟦	🟦	🟦
Write progress report.										🟦	🟦	🟧	🟧	🟧

Figure 3: Semester 1



		Exams		Semester 2													Final Report Deadline
	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Implement the Improved Dynamic Programming(IDP) algorithm	🟦	🟧	🟧	🟦	🟦	🟦											🟧
Implement an algorithm to solve combinatorial auctions		🟧	🟧			🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦				🟧
Testing the correctness of the algorithms		🟧	🟧			🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦				🟧
Write remaining parts of the publication	🟦	🟦	🟦														🟧
Extensive testing, evaluation and result production		🟧	🟧					🟦	🟦	🟦	🟦	🟦					🟧
		🟧	🟧														🟧
		🟧	🟧														🟧
Write the final project report		🟧	🟧									🟦	🟦	🟦	🟦	🟦	🟦

Figure 4: Semester 2



Problem	Loss	Probability	Risk	Plan
The results from my program is insignificant in terms of speed	4	3	12	I will move on to other problems which when evaluated seem more suited for parallel execution if no progress is made after a determined time.
Weekly Milestones needed for normal progress are not met	3	3	9	The weekly milestones will be set to be more than expected in order to promote rapid progress if managed to complete and allow contingency if not met.
Difficulty to program using the CUDA environment	3	2	6	Guidance will be sought from my supervisor, the NVIDIA support team at cudatools@nvidia.com, and documentation for the CUDA framework will be extensively read.
The results from my program is incorrect	2	3	6	Using version management I will tag a commit as a working copy if it passes the tests designed. Therefore I can easily see which logic have changed if the program starts to give out the wrong solution
Computer is stolen or damaged	4	1	4	Using GIT version management on both the machine at home and in the Labs with a remote repository on Github, three points of failure have to happen before there is a significant loss.

Figure 5: Risk assessment