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

Kim Svensson
December 11, 2012

Combinatorial optimization of NP-hard problems using dynamic GPU programming

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

A project progress report submitted for the award of
CompSci MEng

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 parallell computing using a GPU have enabled a speedup when it comes to problems which have some data and computation independant parts. The purpose of this project is to develop and evaluate dynamic programming algorithms suitable for parallell 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.

Background

Coalition structure generation is the consept of, given N agents, generate a coalition such that the outcome is optimal in regards to a given property. This however will be an extreme challenge as the number of possible partitions grows exponential, specifically $O(n^n)$ [1].

Literature search

Work

To this point I have implemented an algorithm called the Dynamic Programing (DP) algorithm by Rothkopf, M.H et al [2] which is used to solve combinatorial auctions, but it can also be used for complete coalition formation. It was first implemented as a sequential algorithm to allow firstly for a baseline to compare execution speed by. But also to gain knowledge about the algorithm to allow easier transition towards a parallell version. The algorithm I have derived that run on the GPU consists of generating a disjoint subset from a given set, and then calculated the value the disjoint sets give.

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

λ	How many bids should be evaluated per thread
Υ	A shared array containing threads maximum bid values
Δ	A shared array containing threads maximum bid subsets
υ	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: $\Delta_{tid} := \Upsilon_{tid} := 0$                              Initilize 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** $|C'_i| \leq \frac{|C|}{2} \land \psi \leq \Psi$ **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:     $i := i >> 1$                                 bitshift 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 bitwise AND with C to get a garanteed 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 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.

Result

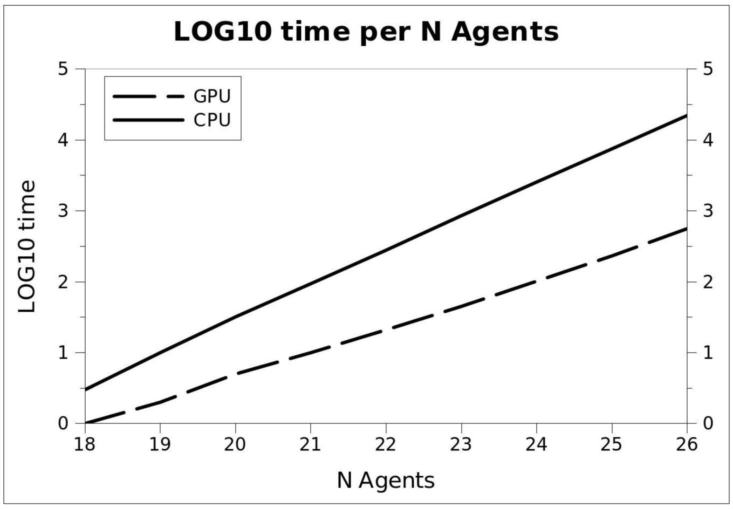

Figure 1: LOG10 runtime per N Agents

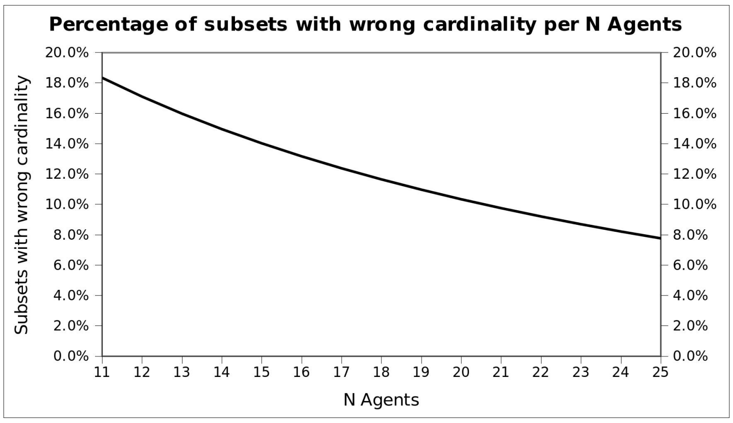

Figure 2: Percentage of generated subset of wrong cardinality

Discussion

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 line CPU and GPU is different, 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 runtime for the cpu increase by three. While the GPU bound algorithm starts its growth at 2 and the growth increases to 2.42 for 25 agents.

Figure 2 is the evaluation of line 4 on how many susbsets 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.

Future work

As there is currently a selected number of algorithms that is regarded faster than the DP algorithm such as the IDP algorithm [1] that have shown a prommising efficiency. The next step would be to implement it both on the CPU and GPU. However as it is fairly simmilar to the DP algorithm, implementation may be trivial and further goals need to be determined. As the current work only works for a small number of agents, the next step would be to implement an algorithm to solve more realistic combinatoral auctions with an relative large number of agents and low number of coalition structures.

Bibliography

[1] 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.

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