A Parallel Algorithm for Coalition Structure Generation

Kim Svensson,[1] Sarvapali D. Ramchurn,[1] Francisco Cruz,[2] Juan-Antonio Rodriguez,[2] Jesus Cerquides[2]

[1] Electronics and Computer Science, University of Southampton, SO17 1BJ, UK
{ks,sdr}@ecs.soton.ac.uk
[2] IIIA, CSIC, Spain {tito,jar,jesus}@iiia.csic.es

Abstract. We develop the first parallel algorithm for combinatorial optimisation problem of Coalition Structure Generation (CSG) that is central to many multi-agent systems applicatoins. Our approach involves distributing the key steps of a dynamic programming approach to CSG across computational nodes on a Graphics Processing Unit (GPU) such that each of the thousands of threads of computation can be used to perform small computations that speed up the overall process. In so doing, we solve important challenges that arise in solving combinatorial optimisation problems on GPUs such as the efficient allocation of memory and computational threads to every step of the algorithm. In our empirical evaluations on a standard GPU, our results show an improvement of orders of magnitude over current dynamic programming approaches with an ever increasing divergence between the CPU and GPU-based algorithms in terms of growth. Thus, our algorithm is able to solve the CSG problem for 29 agents in one hour as opposed to two days for the current state of the art dynamic programming algorithms.

1 Introduction

Coalition formation is a one of the key coordination mechanisms in multi-agent systems. It involves the coming together of a number of agents to achieve some individual or group objective. An important step in the coalition formation process involves partitioning the set of agents into coalitions that, on aggregate (as a coalition structure), maximise the efficiency in the system. This problem is termed the *Coalition Structure Generation* problem (CSG) [5]. The CSG problem is a hard combinatorial optimisation problem and scales in $\omega(n^n)$ [5]. To date, many algorithms have been designed to solve the CSG. These range from branch-and-bound approaches such as [**?**], to dynamic programming approaches such as [3, 2]. The latter are particularly attractive given their lower complexity but most of these algorithms scale to around 30 agents given the exponential growth in computation involved. We also note that these algorithms were mainly designed for single threaded architectures, and even if distributed variants exist [**?**], these require storing the input data redundantly across multiple computers and sharing information over network links that may be liable to delays and losses.

In contrast, in the last few years, a surge in the development of frameworks for parallel programming on a single machine has been noted with the development of general purpose Graphics Processing Units (GPUs). Indeed, these processors, originally developed for only processing high end graphics, can now be programmed to perform simple mathematical operations that, when performed in thousands in parallel, can significantly outperform single-threaded machines with higher frequencies and memory speeds. There are multiple challenges, however, that need to be overcome before complex algorithms as for CSG can be implemented onto the GPU (we elaborate on these in Section **XX**) including the need to limit random memory access, the limits on the number of threads that can be launched to share the same memory blocks, and the need to avoid loading data frequently from main memory.

Against this background, in this paper, we present a parallel algorithm for CSG for highly multi-threaded GPUs that meets the challenges above and outperforms the best algorithms for CSG. Our algorithm, GPU-CSG, parallelises the computation of individual steps of a dynamic program to solve the CSG. It does so by partitioning the computation across thousands of threads where each solves a small sub-problem of the larger optimisation problem. The solution to each sub-problem is then used to find the best partition of agents. In more detail, this paper advances the state of the art in the following ways. First, we develop the first parallel algorithm for CSG that avoids redundant memory storage and inter-processor communication. Second, we prove that GPU-CSG is correct and complete and demonstrate through empirical evaluation that its growth rate is significantly lower than that of the dynamic programming approach it builds upon. Third, we empirically show that GPU-CSG outperforms the state of the art by orders of magnitude, solving the CSG problem for 30 agents in **XX** hours as opposed to **XX days or months** for the previous state of the art.

The rest of this paper is structured as follows. Section 2 presents the background to this work and discusses related work while Section 3 introduces the GPU architecture. Section 4 then details the DP algorithm upon which we build GPU-CSG while section 5 while Section 7 concludes.

2 Background

2.1 Coalition formation

2.2 The DP Algorithm

The DP algorithm as shown in algorithm 1 works by producing two output tables, O and f, where each table have one entry per coalition structure. An entry in f represent a value a certain coalition structure is given, while O represent which splitting, if any, maximised the coalition structure for the entry in f which it represent. More elaborated, given all coalitions of agents $C \subseteq A$, for each coalition in C, evaluate all pairwise disjoint subsets here named splittings on their pairwise collective sum against the coalitions original value. Given one splitting is greater, update the value of the coalition $f(C) := f(C') + f(C \setminus C')$ and

assign O on C to represent the new splitting, $O(C) := \{C', C \setminus C'\}$. These steps are first carried out on all coalition structures with two agents, continuing until N agents. This means, given a coalition structure S with cardinality $|S| = n$, then all coalition structures for the sizes 1,2,...,n-1 have already been evaluated. The dynamic programing algorithm is entierly deterministic meaning that even if there was only one or two valuations, the algorithm will evaluate all splittings before it reaches a conclusion. However this algorithm does not work well with an large amount of agents as it grow exponential and have an time complexity of $O(3^n)$. As described later in section ?? the part of the algorithm that is parallilsed is the max function on line 4 which handles the evaluation of all splittings of a given coalition structure.

Algorithm 1 Dynamic Programming algorithm

INPUT: b: collection of the bids for all coalitions
VARIABLES: f: collection holding the maximum value for all coalitions
O: collection holding the most beneficial splitting for all coalitions.

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**
9: Set $CS^* := \{A\}$
10: **for** all $C \in CS^*$ **do**
11: **if** $O(C) \neq C$ **then**
12: Set $CS^* := (CS^* \setminus \{C\}) \cup \{O(C), C \setminus O(C)\}$
13: Goto 10 and start with a new CS^*
14: **end if**
15: **end for**
16: **return** CS^*

The table O may be discarded and not calculated to reduce the memory requirement by half removing instant access to the final splittings. These final splittings are easily retrived as outlined in algorithm 2. Essentialy, all coalitions in $C \in CS^*$ which value in f is not equal to the initial bid in b, find the first splitting that is equal to the value in f. The overhead of this is insignificant as it needs to evaluate at most $n - 1$ coalitions compared to the exponential number of evaluations carried out in the previoius steps[4].

Algorithm 2 Enumeration of the optimal splittings through re-evaluation of small amount of coalitions

INPUT: b : array of the initial bids for all coalitions $C \subseteq A$. f : the final evaluated values gathered from evaluating splittings.

1: Set $CS^* := \{A\}$
2: **for** all $C \in CS^*$ **do**
3: **if** $f(C) \neq b(C)$ **then**
4: find first C^* where $f(C) = f(C \backslash C^*) + f(C^*) : C^* \subseteq C \wedge 1 \leq |C^*| \leq \frac{|C|}{2}$
5: Set $CS^* := (CS^* \setminus \{C\}) \cup \{C^*, C \setminus C^*\}$
6: Goto 2 and start with a new CS^*
7: **end if**
8: **end for**
9: **return** CS^*

2.3 The CUDA Architecture

Graphics Processing Units(GPU) from Nvidia and AMD is highly multithreaded, many-core architectures primarily aimed at highly parallel image processing and rendering, however it have in the recent years moved towards supporting general purpose computing through the OpenCL and Nvidia CUDA framework. It does so by devoting a larger amount of transitors towards many computational units rather than data caching and advanced flow controll more often seen in CPU architectures. Nvidia describes their general purpose GPU CUDA architecture as a Single Instruction Multiple Threads (SIMT) architecture, meaning groups of multiple threads excecute the same instructions concurrently and is proportional to SIMD architectures. This enables their GPUs to be highly advantageous when performing data-independant and non-divergent tasks.

To understand this further the grouping of the threads need to be explained and is outlined in figure 1. A kernel which is a device specific CUDA function that is called by the sequential host code, will request a specified number of blocks in a grid of blocks. Each block may to this date consist of up to 1024 threads depending on the compatability of the card, with a maximum grid size of $2^{31} - 1$ blocks subjected to compatability. When run, the blocks will be distributed onto available multiprocessors, which then independantly schedule the runtime of the block. Note that blocks may be excecuted concurrently or sequential depending on the current workload and the number of available multiprocessors. The block is split into smaller units of 32 threads called warps, all threads within the same warp are always scheduled the same instruction to be run and this is what embodies the SIMT paradigm. Therefore, branching threads causing inter-warp divergence means a warp will have inactive threads not excecuting any instructions, which may lead to poor efficiency with worst case of sequential performance. Further, warps are scheduled independantly of eachother meaning possible concurent excecution of warps.

The threads communicate with each other through writes to various types of memory outlined in table 1. There are three types of thread writable memory in

the architecture; registers and local memory are each threads coupled memory which is not volatile and may be shared with other threads inside the same warp as described in section 2.4. Shared memory is as its name tells shared between all threads within the same block, as it may be written to by any thread within the block it should be treated as volatile, thus syncronization inside the block have to be consider whilts dealing with shared memory. Finally, global memory is the only persistant memory which will persist between each kernel call, it may be manipulated by the host, but also by any thread, and is the only means of communication inbetween kernels, blocks, and the host. It to should be considered voaltile.

Regarding access to the global memory, certain access patterns must be followed in order maximise performance. For global memory, it is important to note that each load request from memory will fetch in cachelines of size $32 * wordsize$, meaning cachelines of 32, 64, and 128 bytes each when pulling the primitives char, short and int respectively. The reason for this is the fact mentioned earlier, all 32 threads within the same warp issue the same instruction, thus each warp fetches 32 entities of a specific word type. As a result, if the memory reads within a warp is not coalesced(grouped within the same cacheline) within consecutive words, the effective bandwidth will drop immediately.

Table 1. Memory scope, lifetime, and speed

Type	Scope	Lifetime	Relative Speed
Register	Thread & Warp	Thread	Fastest
Shared	Block	Block	Fast
Global	Kernel & Host	Program	Slow

2.4 Data structure

How the data is represented and structured is important, especially in bandwidth bound algorithms where the majority of time is spent fetching data from memory and the arithmetic overhead is low. Selecting the right composition will reduce the memory requirements substantialy. Given the two entities of data that is needed to be represented for each coalition structure, the coalition structure itself and its value in f. Memory constraints will be imposed given a large amount of agents as a result of DP's exponential growth. In order to minimize memory usage several technices were used. Representing a coalition structures members as an array of values, where each value represent a distinct agent may seem intinuative at first. However, if the members are represented as bits set in a fixed sized integer, the memory requirement will be reduced substantialy as shown by previoius studies.[1] If solving for the complete coalition structure formation problem with n agents representing members as an array of values. There are $\binom{n}{i}$ coalition structures of size i, where i entries have to be stored per

coalition structure. The total number of values needed to store just to represent the coalition structures is there for equal to:

$$\sum_{i=i}^{n} \binom{n}{i} * i$$

Given the same constraints, representing the coalition structure as an fixed sized integer, it is only needed to store one entry per coalition structure, totaling to $2^n - 1$ data points.

To give an example, with four agents $A = f_1, f_2, f_3, f_4$, the coalition $C = f_1, f_3, f_4$ would be represented as $C = 1101$ in the binary system and 11 in the decimal system. Therefore, if the coalition structure is represented as an integer it can implicitly be stored as an index to its coalition value, by enumerating it on runtime. That means the only memory constraint on the system is the store all the values for all the coalition structures.

Table 2. Splittings of $C = \{f_1, f_3, f_4\}$ Binary $C = 1101$

Set system	$\{f_1\}\{f_3, f_4\}$		$\{f_3\}, \{f_1, f_4\}$		$\{f_4\}, \{f_1, f_3\}$	
Binary system	0001	1010	0010	1001	1000	0011

Algorithm 3 initShift input $Coalition : C$

```
1: t := C
2: count := 0
3: while t > 0 do
4:     index := FindFirstSet(t)
5:     shift_count := index
6:     nullBit(t, index)
7:     count++
8: end while
9: return shift
```

Coalition Structure Splittings Splittings as mentioned are pairwise disjoint subsets of a coalition structure, given the coalition structure $C = \{f_1, f_3, f_4\}$ the splittings are shown in table 2. In order to generate the splitting there is essentially two methods used, the, initShift, initialSplit and nextSplit methods. The function initShift as detailed in algorithm 3 is necessary to setup the environment for all calls to initialSplit, what it does is using the bit operation

$findfirstset$ to find the indexes of all bits set in from the integer coalition input. This will give each entry in the $shift$ array an unique number. This unique numbers will be used by initialSplit to distribute the bits of the count to fit the configurations bits. It does so by taking a count as input representing which n'th splitting should be created, finds the index of its set bits, and finaly left shifts each bit with the value in the shift array its index reference to.

nextSplit works through a recurence relation which means in order to have concurent threads independant of eachother, an initial splitting for each thread have to be calculated using initialSplit. initialSplit works by first genereting an packed index array of which bits are set in the coalition structure using initShift. Given which n'th splitting it should generate, it distributes the bits of n to the corresponding bits of coalition C. Thereafter nextSplit will be used to generate the next splitting.

Algorithm 4 nextSplit input $Coalition : C$ $Splitting : S$

1: $C' := twosComplement(C)$
2: $S' := bitwiseAND((C' + S), C)$
3: **return** S'

Algorithm 5 initialSplit input $Count : n, Coalition : C$

1: $t := n$
2: $S := 0$
3: **while** $t > 0$ **do**
4: $\quad index := FindFirstSet(t)$
5: $\quad S := S + leftShift(1, shift_{index,C})$
6: $\quad nullBit(t, index)$??
7: **end while**
8: **return** S

Collisions between Splittings of Coalitions Given that each thread evaluate several splittings over numerous coalition structures in parallel, it is bound that splittings on two coalition structures that overlap will have splittings that collide. A collision means that splittings of coalition structures contain at least one identical subset shared between them.

$$\forall S \subset C \wedge S' \subset C' : C \cap C' \neq \emptyset \Rightarrow \exists S \wedge S' : S = S'$$

For all splittings of C and C' where the coalition structures intersect, there must excist at least one splitting shared between them.

Algorithm 6 Fetch using Collision detection

Input: Index:ψ Which n'th splitting should be generated Index:z The index in υ for the coalition structure that is evaluated
Variables:Value α: Holds temporary values

1: $C_0 := initialSplit(\psi, CS_0)$
2: $C_1 := initialSplit(\psi, CS_1)$
3: $\upsilon_{0,z} := f(C_0)$
4: **if** $C_1 = C_0$ **then**
5: $\quad \upsilon_{0,z+1} := \upsilon_{0,z}$
6: **else**
7: $\quad \upsilon_{0,z+1} := f(C_1)$
8: **end if**
9: $C_0 := CS_0 \backslash C_0$
10: $\alpha := f(C_0)$
11: $C_1 := CS_1 \backslash C_1$
12: **if** $C_1 = C_0$ **then**
13: $\quad \upsilon_{0,z+1} := \upsilon_{0,z+1} + \alpha$
14: **else**
15: $\quad \upsilon_{0,z+1} := \upsilon_{0,z+1} + f(C_1)$
16: **end if**
17: $\upsilon_{0,z} := \upsilon_{0,z} + \alpha$
18: **return** υ

The number of splittings that collide is dependant on how many common members the coalitions have in common.

$$2^{n-1} - 1 : C \neq C' \wedge |C \cap C'| = n \wedge n > 1$$

Specifically, the number of splittings incommon is how many splittings can be done all the members in common. Normaly, each splitting would be fetched from global memory resulting in it being fetched several times, if it have not been evicted from the cache memory.

By using collision detection as described in algorithm 6 the number of reduntant fetches may be reduced. It works by evaluating two or more coalitions at the same time, lines 1 to 3 generates the intial splittings and fetches the value for the first splitting. It then trough lines 4 to 8 checks wheater the splitting of the second splitting is equal to the first. If so assign it the first splittings value, else fetch its own value from global memory. Finally, the last lines does the same thing as above just that it does that on the pairwise disjoint subset, and the first splittings next value is stored in an temporary variable. The more splittings that are evaluated at the same time and checked against eachother, the less reduntant memory fetches are done.

Reduction As the evaluation of each coalition structure is to find the splitting of the coalition structure which maximises the value of the coalition structure, it is simply needed to compare the values of all splittings with eachother to find

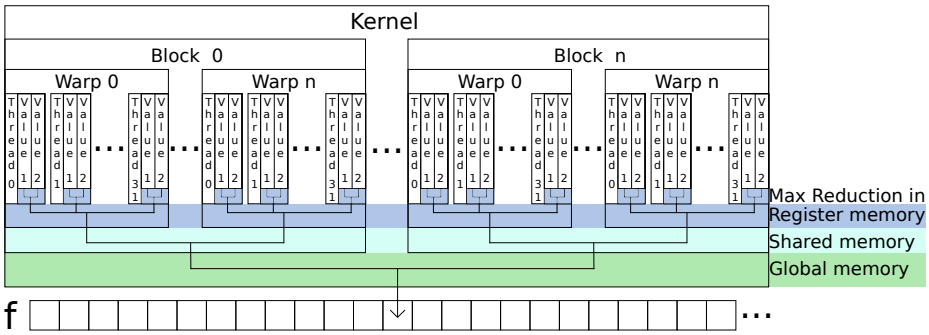

Fig. 1. Outline of reduction across thread, warp, block and kernel

the most valued one. The reduction is done on four levels of scope as seen on lines 25 to 37 in algorithm 7, as well as outlined in figure 2.4.

On thread level, each thread evaluate a number of splittings to determine their most valued splitting. On warp level, all threads inside the same warp concurrently exchange their largest register values to find the most valued splitting among the warp. This is done by utilizing a function called $\_shfl\_xor$ which allows for an exchange of register values between any thread within the same warp. Using this technices allows for a substantial reduction in shared memory use, as all that is needed to be stored is one value per warp. With each single value from each warp moved into shared memory, initially the total number of active threads will be equal to the number of warps inside the block divided by two. These thread will compare one warp value each with the respectiv value that is not being compared by another thread, half the number of threads and iterate over again. This is done until only one thread remains which will compare the last two values. Finally, the single thread that compared the last values, will try to update the value in global memory if it is larger using atomic functions.

2.5 Algorithm

The algorithm starts by initilising all the variables needed further on, as generating the next coalition structure works through a recurrance relation, only the first thread will update the shared memory. Continuing, next step is to generete the shift array, this can be done in parallel so a number of threads invoke the method initShift.

2.6 Experimental setup

The GPU instance of the algoritm was run on a linux desktop computer using CUDA version 5.0 containing 12GiB DDR3 RAM, 3.2GHz AMD Phenom II X4 CPU and a consumer grade NVIDIA GeForce GTX 660 Ti with a GPU clock of 915MHz and 6008MHz effective clock on the memory. It ran 256 threads per

Algorithm 7 GPU implementation of the DP algorithm

Input

f	The array which holds the bids
C_0	The first coalition struction to do evaluation on
Ψ	The maximum number of splittings

Constants

λ	How many bids should be evaluated per thread
$confpkernel$	
$nparallelconf$	

Variables

Υ	A shared array containing warps maximum bid values
υ	A local array containing one of the threads bid value
$bid = blockIdx.x$	Which block the threads belong to
$conf$	
$bdim = blockdim.x$	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: **if** $tid = 0$ **then**
2: $conf_0 := C_0$
3: **for** $i := 1$ to $confpkernel$ **do**
4: $conf_i := nextCoalition(conf_{i-1})$
5: **end for** Checkpoint 1
6: **end if**
7: **if** $tid < confpkernel$ **then**
8: $initShift(conf_{tid})$
9: **end if** Checkpoint 2
10: **for** $x := 0$ to $confpkernel$ **do**
11: Set all values in υ to 0
12: **if** $\psi \geq \Psi$ **then**
13: goto postfetch
14: **end if** Checkpoint 3
15: **for** $z := 0$ to $nparallelconf$ **do**
16: **if** $conf_{z+x} \geq maxval$ **then**
17: break
18: **end if**
19: $C_z := initialSplit(\psi, conf_{z+x})$
20: $\upsilon_{0,z} := f(conf_{z+x} \backslash C_z) + f(C_z)$
21: $C_z := nextSplit(C_z)$
22: $\upsilon_{1,z} := f(conf_{z+x} \backslash C_z) + f(C_z)$
23: $z := z + 2$
24: **end for** Checkpoint 4
25: **for** $z := 0 tonparallelconf$ **do**
26: **if** $\upsilon_{1,z} > \upsilon_{0,z}$ **then**
27: $\upsilon_{0,z} := \upsilon_{1,z}$
28: **end if**
29: $warpReduction(\upsilon_{0,z})$
30: **if** $tid \% 32 = 0$ **then**
31: $i := tid/32$
32: $\Upsilon_{i,z} := \upsilon_{0,z}$
33: **end if** Checkpoint 5
34: **end for**
35: $blockReduction()$ Checkpoint 6
36: **if** $tid = 0$ **then**
37: $atomicUpdate()$
38: **end if** Checkpoint 7
39: $x := x + nparallelconf$
40: **end for**

block, with each thread evaluating two splittings per coalition structure, where 8 coalition structures was evaluated in parallel for a total of 32 coalition structures visited. 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.

3 Results

4 Discussion

Figures **??** and **??** shows the difference between employing shared splittings discussed in section 2.4. What it show are the number of logarithmic clock cycles elapsed between the previoius checkpoint and the current one as outline in algorithm **??**. What can first be noted is that checkpoint 4 is where the algorithm spends most of the time. Checkpoint 4 denote the generation of splittings and fetching them. As mentioned earlier the algorithm is bandwidth bound and spending almost 60% of the time fetching memory shows that. Another reason of the result is due to the not aligned random access making the effective bandwidth low. Knowing that and looking at figure **??** showing the result of sharing splittings it can be concluded that whilst it still spends most of the time fetching memory, henceforth improving the sharing of splittings make a significant impact on the overal performance.

Figure **??** shows the difference between the CPU and GPU algorithm. For every N agents the y axis shows the logarithmic elapsed time. The difference shown here between the implementations is substantial showing that solving the problem on the GPU is highly beneficial. For 29 agents it solves it in 2 hours while the CPU bound algorithm was etrapolated to show that it would take up to 83 hours to complete. This is an speed factor of over 40 which would grow over the amount of agents. The reason it would grow is due to the implementations not being linearly parallel in logarithmic time where the GPU implementation having a smaller growth. The difference between the GPU implementations where one utilizes internal sharing of splittings may also be noted to being 40% faster than the variant without.

5 Conclusion

This paper show using the GPU to solve to Complete coalition structure formation problem is much viable option. Being so much faster it shows that

References

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

2. T. Rahwan and N. R. Jennings. Coalition structure generation: Dynamic programming meets anytime optimization. In *AAAI*, pages 156–161, 2008.
3. T. Rahwan and N. R. Jennings. An improved dynamic programming algorithm for coalition structure generation. In *AAMAS*, pages 1417–1420, 2008.
4. T. Rahwan and N. R. Jennings. An improved dynamic programming algorithm for coalition structure generation. In *Proc 7th Int Conf on Autonomous Agents and Multi-Agent Systems*, pages 1417–1420, 2008.
5. T. Sandholm, K. Larson, M. Andersson, O. Shehory, and F. Tohmé. Coalition structure generation with worst case guarantees. *Artif. Intell.*, 111(1-2):209–238, 1999.