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 applications. 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) [7]. The CSG problem is a hard combinatorial optimisation problem and scales in $\omega(n^n)$ [7]. 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 [4, 3]. 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 Related work

Solving data independant and parallel problems will always have a benefit when run on a GPU, whether it is computing on MRI scans which gave an significantly large speedup factor of 431 using the GPU, or generating hashes and getting a comparably modest speedup of times 11 [6]. It has been shown from time to time, whenever the algorithm have data independant parts which may be run concurrently, the GPU will most likely succeed better.

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 factor speedup of

26 [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, due to being able to fetch more data points at an faster rate. This is one of the most important aspects of GPU programming as you are first limited to the amount of memory the GPU have, but also in order to increase the effective bandwidth of the algorithm.

Moving on to the CSG problem where Michalak, T et. al. uses a distributed variant of the anytime IP algorithm called D-IP to solve the Coalition Structure Generation [2]. Using 14 dualcore workstations to ditribute the workload to speed up their runtime. This study first shows it is possible to distribute the solutions for the CSG problem, secondly the speed benefit they gained from distributing computation, with only a runtime of 11% to 4% compared to the centrilised and serialised IP. However, having a tighter coupling between the computation units such as with the GPU, expensive latencies from ehternet communication and moving essentiall data between computational units will be largely reduced.

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 programming algorithm is entirely 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 2.5 the part of the algorithm that is paralleled is the max function on line 4 which handles the evaluation of all splittings of a given coalition structure.

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 retrieved as outlined in algorithm 2. Essentially, 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 previous steps[5].

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:         $f(C) := max\{f(C\backslash C') + f(C') : C' \subseteq C \wedge 1 \leq |C'| \leq \frac{|C|}{2}\}$
 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 $CS^*$
```

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 multi-threaded, 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 transistors towards many computational units rather than data caching and advanced flow control 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 execute the same instructions concurrently and is proportional to SIMD architectures. This enables their GPUs to be highly advantageous when performing data-independent and non-divergent tasks. To understand this further the grouping of the threads need to be explained and is outlined in figure 2. 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 compatibility of the card, with a maximum grid size of $2^{31} - 1$ blocks subjected to compatibility. When run, the blocks will be distributed onto available multiprocessors, which then independently schedule the run-time of the block. Note that blocks may be executed 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 executing any instructions, which may lead to poor efficiency with worst case of sequential performance. Further, warps are scheduled independently of each other meaning possible concurrent execution 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 synchronization inside the block have to be consider whilst dealing with shared memory. Finally, global memory is the only persistent 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 in between kernels, blocks, and the host. It to should be considered volatile.

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 substantially. 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 techniques were used. Representing a coalition structures members as an array of values, where each value represent a distinct agent may seem intuitive at first. However, if the members are represented as bits set in a fixed sized integer, the memory requirement will be reduced substantially as shown by previous 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 $A = f_0, f_1, f_2, f_3$, the coalition $C = f_0, f_2, f_3$ would be represented as $C = 1101$ in the binary system and 13 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 run-time. That means the only memory constraint on the system is the store all the values for all the coalition structures.

Table 2. Splittings of $C = \{f_0, f_2, f_3\}$ Binary $C = 1101$

Set system	$\{f_0\}\{f_2, f_3\}$	$\{f_2\}, \{f_0, f_3\}$	$\{f_3\}, \{f_0, f_2\}$
Binary system	0001 1100	0100 1001	1000 0101

Algorithm 3 initShift input *Coalition* : *C* *Index* : *n*

```
t := C
count := 0
while t > 0 do
    index := FindFirstSet(t)                Finds first bit set in t
    shift_{n,count} := index
    nullBit(t, index)                       Sets one bit in t to zero
    count++
end while
return shift
```

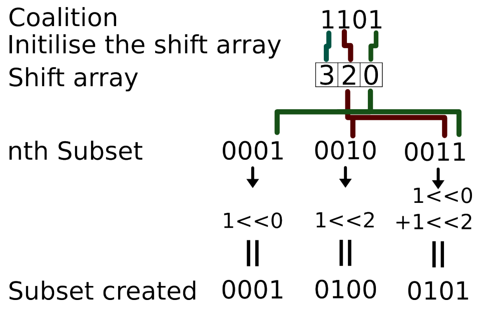

Fig. 1. How initShift and initialSplit works

Coalition Structure Splittings Splittings as mentioned are pairwise disjoint subsets of a coalition structure, given the coalition structure $C = \{f_0, f_3, f_4\}$ the splittings are shown in table 2. In order to generate the splitting there is essentially three methods used, initShift, initialSplit and nextSplit. The function initShift as detailed in algorithm 3 and partly illustraded in figure 1, it 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 the integer coalition input, and sets those indexes as values in the shift array. These numbers will be used by initialSplit in algorithm 5 to distribute the bits of the count to fit the coalition's bits. It does so by taking a count as input representing which nth splitting should be created, finds the index of the counts set bits, and finally left shifts each bit with the value in the shift array its index reference to, this is also illustradet in figure 1. Then to generate the next splitting, a call to nextSplit in algorithm 4 is needed which works through a recurence relation to generate the next splitting.

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

```
$C' := twosComplement(C)$
$S' := bitwiseAND((C' + S), C)$
return $S'$
```

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

```
$t := n$
$S := 0$
while $t > 0$ do
    $index := FindFirstSet(t)$                         Finds the first set bit in t
    $S := S + leftShift(1, shift_{C,index})$ Shifts 1 left with the value in shift and adds
    $nullBit(t, index)$                                Sets the bit of t to 0
end while
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 exist at least one splitting shared between them.

Algorithm 6 Fetch using Collision detection

```
Input: Index:$\psi$   Which nth splitting should be generated Index:$z$   The index in $\upsilon$ for the coalition structure that is evaluated
Variables:Value $\alpha$:                                   Holds temporary values
 1: $C_0 := initialSplit(\psi, CS_0)$          Gets the splitting on $CS_0$ with index $\psi$
 2: $C_1 := initialSplit(\psi, CS_1)$          Gets the splitting on $CS_1$ with index $\psi$
 3: $\upsilon_{0,z} := f(C_0)$                 Fetch the first value
 4: if $C_1 = C_0$ then
 5:     $\upsilon_{0,z+1} := \upsilon_{0,z}$    Sets the other splittings value as its own
 6: else
 7:     $\upsilon_{0,z+1} := f(C_1)$           Else fetches its own value
 8: end if
 9: $C_0 := CS_0 \backslash C_0$              Get the other subset of the pairwise disjoint subset
10: $\alpha := f(C_0)$                         Fetch the second value to a temporary variable
11: $\upsilon_{0,z} := \upsilon_{0,z} + \alpha$   Add it
12: $C_1 := CS_1 \backslash C_1$              Get the other subset of the pairwise disjoint subset
13: if $C_1 = C_0$ then
14:     $\upsilon_{0,z+1} := \upsilon_{0,z+1} + \alpha$     Add the other splittings value to its own
15: else
16:     $\upsilon_{0,z+1} := \upsilon_{0,z+1} + f(C_1)$     Else add its own value
17: end if
18: return $\upsilon$
```

The number of splittings that collide is dependent 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 in common is how many splittings can be done all the members in common. Normally, 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 redundant fetches may be reduced, which may only be possible by evaluating two or more coalitions at the same time.

Lines 1 to 3 generates the initial splittings and fetches the value for the first splitting. It then trough lines 4 to 8 checks whether 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 each other, the less redundant memory fetches will be done.

Reduction As the evaluation of each coalition structure is to find the splitting of the coalition structure which maximizes the value of the coalition structure,

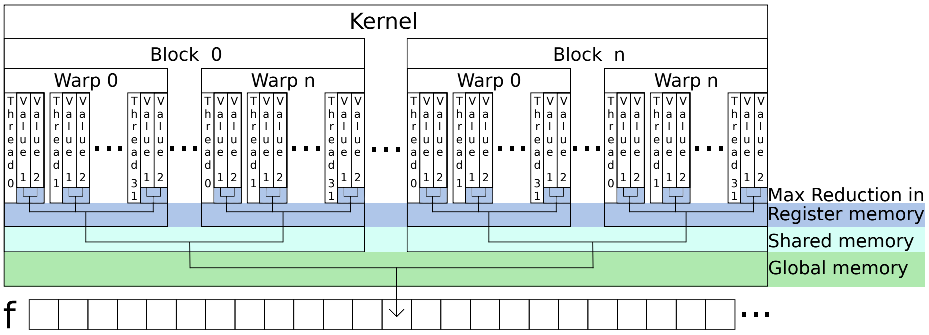

Fig. 2. Outline of reduction across thread, warp, block and kernel

it is simply needed to compare the values of all splittings with each other to find 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.

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 technique allows for a substantial reduction in shared memory use, as all that is needed to be stored is one value per warp in shared memory. 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 respective 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

To parallise the algorithm it evaluates several coaltion structures at the same time, where each thread evaluate a total of two splitting per coaltion structure. As memory is constraint per thread in terms of register and shared memory it will evaluate coalition structures in batches. The total number of coalitions structures is denoted by the constant α and how many coalitions per batch is determined by the constant β. The input it takes is the array which holds the values assigned to f, the initial coalition structure that it will evaluate on and generate the consecutive coalition structures on. The last ones are constants that determine how many splittings that can be done and which is the largest coalition structure which the algorithm is not allowed to exceed.r The algorithm starts by initializing all the variables needed further on, as generating the next coalition structure works through a recurrence relation, only the first thread will update the shared memory. Continuing, next step is to generate the shift array, this can be done in parallel so a number of threads invoke the method initShift. From line 10 it will initilise the loop and start resetting all the values to zero in order to remove any old values in the register. Next line 16 determince whether the thread will continue to the fetching stage or skip it completely. If the thread have a splitting index which denotes which nth splitting should be constructed, is larger than maximum allowed it will jump past the fetch and wait there until all other threads have finished. If it did not jump past it will now generate an amount of splittings and fetching their values to an array to later be evaluated.

Reaching beyond line 25 we reach where all the reduction will happen, on line 26 it evaluates which of the subsets for a coalition structure is the greatest and updates accordingly. Next it will do a reduction in the scope of warp followed by each first thread in each warp will write to the shared memory the maximum value the warp reduction concluded on. Finally, line 35 does the reduction on all the values from all the warps inside the block and when finished the first thread in the block will update the global memory if necessary. Increment the counter and if there is still coalitions to evaluate start from the begining of the loop.

2.6 Experimental setup

The GPU instance of the algorithm 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 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 clock-speed 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 5 and 4 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 previous checkpoint and the current one as outline in algorithm 7. 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 4 showing the result of sharing

Algorithm 7 GPU implementation of the DP algorithm

```
Input
$f$                              The array which holds the values
$C_0$                            The first coalition structure to do evaluation on
$\Psi$                           The maximum number of splittings
$\Omega$                         The full coalition structure
Constants
$\lambda$                        How many splittings should be evaluated per thread
$\alpha$                         The total number of caolition structures that are being evaluated
$\beta$                          The number of coalition structures that are evaluated per batch
Variables
$\Upsilon$                       A shared array containing warps maximum bid values
$\upsilon$                       A local array containing one of the threads bid value
$\Delta$                         Holds all the coalition structures that will be evaluated
$bid = blockIdx.x$               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: if $tid = 0$ then
 2:    $\Delta_0 := C_0$                              Thread 0 sets the first coalition from input
 3:    for $i := 1$ to $\alpha$ do
 4:       $\Delta_i := nextCoalition(\Delta_{i-1})$   Generate the next coalitions to evaluate
 5:    end for                                        Checkpoint 1
 6: end if
 7: if $tid < \alpha$ then
 8:    $initShift(\Delta_{tid}, tid)$                 Initialise the shift array
 9: end if                                            Checkpoint 2
10: for $x := 0$ to $\alpha$ do
11:    Set all values in $\upsilon$ to 0              Reset the values to 0
12:    if $\psi \geq \Psi$ then
13:       goto line 25          The threads index is larger than the number of splittings
14:    end if                                         Checkpoint 3
15:    for $z := 0$ to $\beta$ do
16:       if $\Delta_{z+x} \geq \Omega$ then
17:          goto line 25       The coalition is outside the range of the calculation
18:       end if
19:       $C := initialSplit(\psi, \Delta_{z+x})$     Generate the $\psi$th splitting from a coalition
20:       $\upsilon_{0,z} := f(\Delta_{z+x} \backslash C) + f(C)$   Fetch the splittings value
21:       $C := nextSplit(C)$                         Generate the next splitting of the coalition
22:       $\upsilon_{1,z} := f(\Delta_{z+x} \backslash C) + f(C)$   Fetch the splittings value
23:       $z := z + 1$              Increment counter and move on the next coalition
24:    end for                                        Checkpoint 4
25:    for $z := 0$ to $\beta$ do
26:       if $\upsilon_{1,z} > \upsilon_{0,z}$ then
27:          $\upsilon_{0,z} := \upsilon_{1,z}$         Sets the biggest value
28:       end if
29:       $warpReduction(\upsilon_{0,z})$             Finds maximum value in scope of warp
30:       if $tid\%32 = 0$ then
31:          $i := tid/32$                            Which nth warp does the thread belong to
32:          $\Upsilon_{i,z} := \upsilon_{0,z}$         First thread in each warp writes maximum value
33:       end if
34:    end for                                        Checkpoint 5
35:    $blockReduction()$     Finds maximum value in scope of block     Checkpoint 6
36:    if $tid = 0$ then
37:       $atomicUpdate()$      First thread does an atomic update on global memory
38:    end if                                         Checkpoint 7
39:    $x := x + \beta$
40: end for
41: return $f$
```

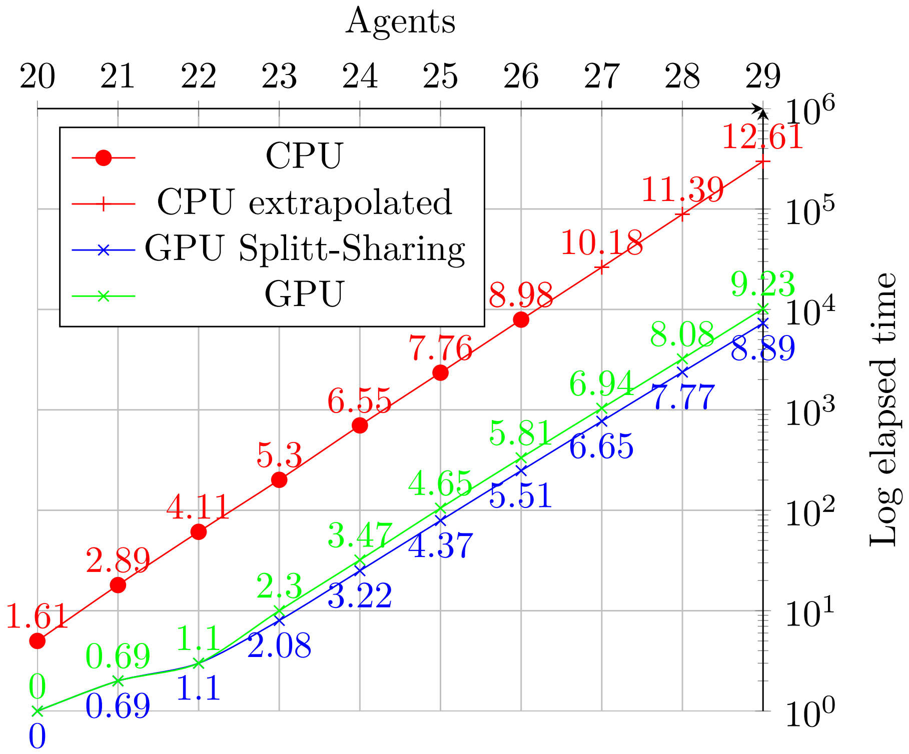

Fig. 3. LOG elapsed time for N agents

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 overall performance.

Figure 3 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 extrapolated 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.

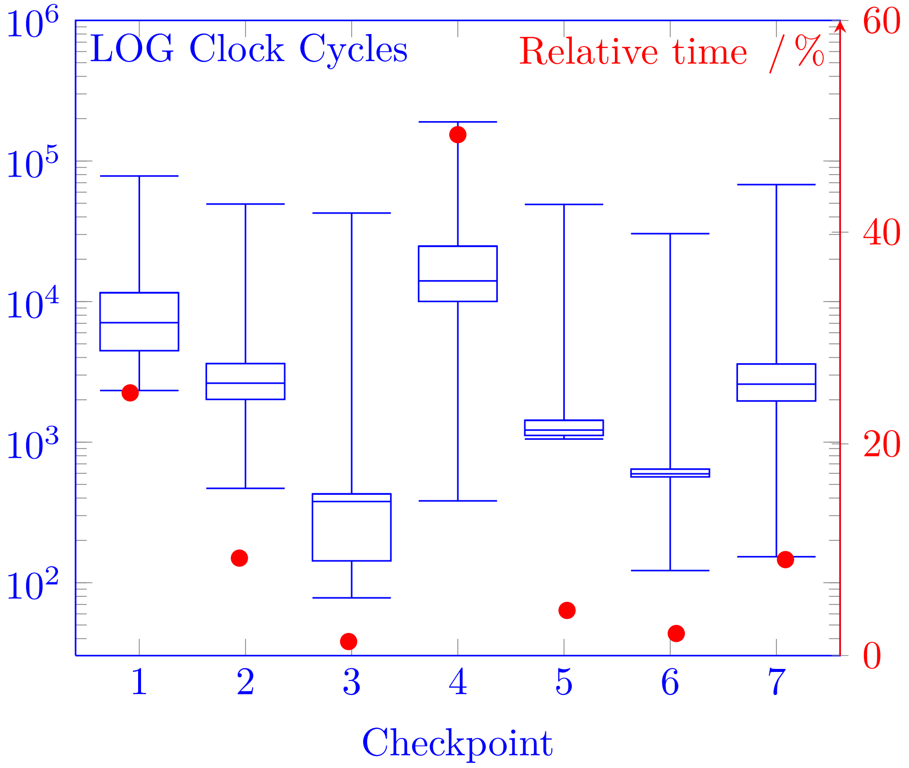

Fig. 4. LOG Clock Cycles between checkpoints and relative time for each code segment, shared splittings

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. Michalak, J. Sroka, T. Rahwan, M. Wooldridge, P. McBurney, and N. 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. Coalition structure generation: Dynamic programming meets anytime optimization. In *AAAI*, pages 156–161, 2008.

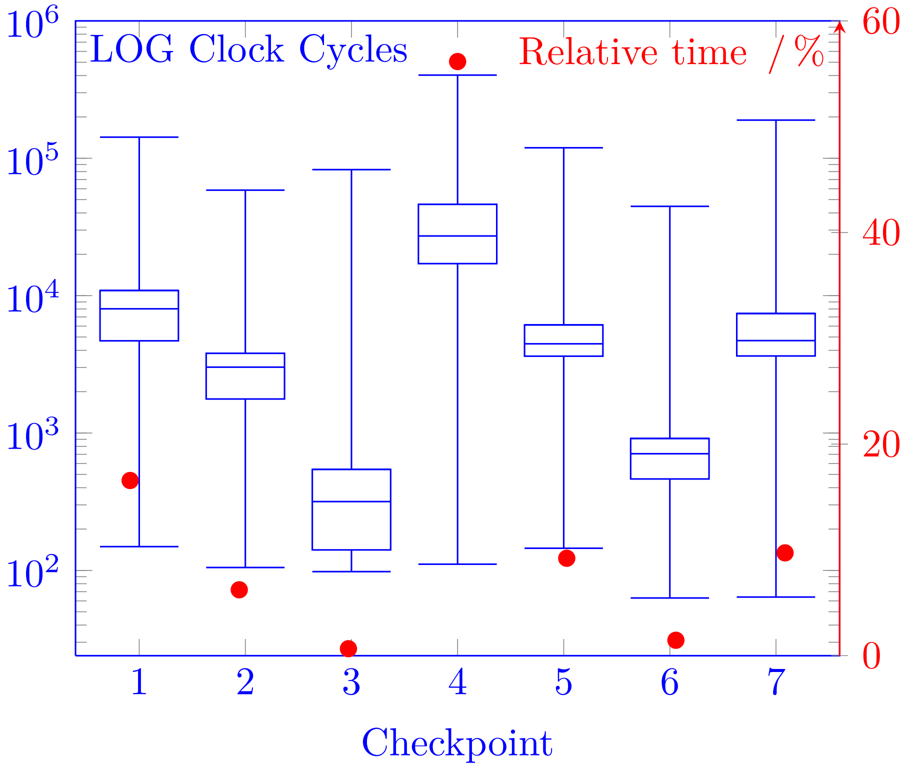

Fig. 5. LOG Clock Cycles between checkpoints and relative time for each code segment

4. T. Rahwan and N. R. Jennings. An improved dynamic programming algorithm for coalition structure generation. In *AAMAS*, pages 1417–1420, 2008.
5. 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.
6. S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008.
7. 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.