# Solving the Coalition Structure Generation Problem on a GPU

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

[1] Electronics and Computer Science, University of Southampton, SO17 1BJ, UK
{ks6g10,sdr}@ecs.soton.ac.uk
[2] IIIA, CSIC, Spain {tito,jar,jesus}@iiia.csic.es

**Abstract.** We develop the first parallel algorithm for Coalition Structure Generation (CSG), which 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 and thirty minutes as opposed to three days for the current state of the art dynamic programming algorithms.

## 1 Introduction

Coalition formation is 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) [8]. The CSG problem is a hard combinatorial optimisation problem and scales in $\omega(n^n)$ [8]. 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 [5, 4]. 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 [3], 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 3) 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 29 agents in less than 1.5 hours as opposed to 83 hours for the CPU-based algorithm.

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

## 2 Background

Solving data-independent and parallel problems will always have a benefit when run on a GPU, whether it is computing on MRI scans, which gave a significantly large speedup factor of 431 using the GPU, or generating hashes and getting a comparably modest speedup of times 11 [7]. This is particularly the case whenever the algorithm has data-independent sub-routines which may be run concurrently. If so the GPU will most likely perform better. Combining the GPU together with dynamic programming has been used before to solve similar combinatorial optimisation problems. Boyer, et al. successfully implemented and solved the knapsack problem with a factor speedup of 26 [1]. They also introduced ways to reduce memory usage and thus enable computation of much larger data sets, as well as reduce the bandwidth utilised. In so doing, they also

reduce computation time by being able to fetch more data points at an faster rate. Thus, they fully exploit the key features of GPU programming whereby, with limited amount of memory available on a GPU, they are able to maximise the effective bandwidth of the algorithm.

Now, turning to the CSG problem, in terms of parallel programming approaches, we note the work of Michalak et. al. that uses a distributed variant of the anytime IP algorithm called D-IP to solve the Coalition Structure Generation [3]. They use 14 dual-core workstations to distribute the workload to speed up their run-time. Their algorithm shows it is possible to distribute the solutions for the CSG problem and that there may be significant from distributing computation (taking only 11% to 4% of the time, compared to the centrilised and serialised IP). However, while their algorithm use traditional programming strategies effective on powerful single-threaded systems, it does require data to be shared between computational nodes over potentially slow Ethernet links and also sharing redundant copies of the input across each. Finally, their algorithm also suffers from the same weaknesses as IP, that is, no deterministic completion time (in the worst case ($O(n^n)$) as it is dependent on the coalition value function. In contrast, GPU-CSG has a deterministic completion time ($O(3^n)$) and relies on much faster GPU memory access.

In what follows, we first describe the CUDA architecture (the GPU architecture provided by Nevada) to clarify what are the key features and issues of using GPUs to solve large combinatorial problems. We then describe the DP algorithm and the data structures we use to implement it on the CUDA architecture.

### 2.1 The CUDA Architecture

Graphics Processing Units (GPUs) from NVIDIA and AMD are highly multi-threaded, many-core architectures primarily aimed at highly parallel image processing and rendering. In recent years, however, there has been a move to use these to support more general-purpose computing through the OpenCL and NVIDIA CUDA framework. This was achieved 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 1. The kernel is a device-specific CUDA function that is called by the sequential host code, which 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

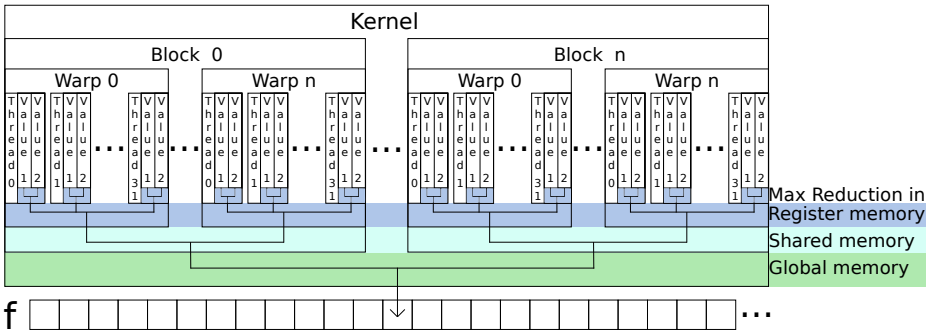


**Fig. 1.** Outline of reduction across thread, warp, block and kernel . The kernel is a function called by the CPU.

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 [3] paradigm. Therefore, branching threads causing intra-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 3.1. Shared memory is as its name tells is 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.

Now, an important element of GPU programming is to manage the access to global memory effectively as it is very slow compared to memory access of the on-chip memory on the GPU. In more detail, each load request from memory will fetch in cache-lines of size $32 * wordsize$, meaning cache-lines of 32, 64, and 128 bytes each when pulling the primitives char, short and int respectively. This is because, as mentioned earlier, all 32 threads within the same warp issue the same instruction. Thus each warp fetches 32 entities of a specific word type, if the memory reads within a warp is not coalesced (grouped within the same cache-line) within consecutive words, the effective bandwidth will drop immediately.

[3] Single Instruction Multiple Threads (SIMT) architecture, All threads inside a warp execute the same instruction

Table 1 summarises the key features of GPU memory access speeds and shows why it is important to avoid loading data from the global memory.

**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 |

In the next subsection, we describe the DP algorithm, which we build upon.

### 2.2 The DP Algorithm

The DP algorithm [2] was originally designed to solve the winner determination problem for combinatorial auctions. However, in recent years, it has been applied as the de facto algorithm (complete and with the lowest worst case complexity) to solve the CSG problem and variants of it have improved and adapted it to different settings [5, 9]. However, the core of the algorithm is the same in all these variants and relies on dynamic programming. Hence, in this paper we use this as the basis for our GPU version as the dynamic programming approach has shown potential for parallelisation as evidenced by previous work (as for the Knapsack problem as mentioned earlier).

To show how the algorithm works, we first formalise the CSG problem. Let $A = \{1, \ldots, |A|\}$ be a set of agents. A subset $C \subseteq A$ is termed a coalition. Then, a $CSG$ problem is completely defined by its characteristic function $v : 2^A \rightarrow \Re$ (with $v(\emptyset) = 0$), which assigns a real value representing utility to every feasible coalition. The CSG problem is to identify the exhaustive disjoint partition of the space of agents into coalitions (or, *coalition structure*) $CS = \{C_1, \ldots, C_k\}$ so that the total sum of values, $\sum_{i=1}^{k} v(C_k)$, is maximised.

Now, DP (see Algorithm 1) works by producing two output tables, $O$ and $f$, where each table has one entry per coalition structure. An entry in $f$ represents a value a certain coalition structure is given, while $O$ represent which splitting, if any, maximised the coalition structure for the entry in $f$ that it represents. 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 $|A|$ 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 a large number of agents as it grows exponentially and has time and memory complexity $O(3^n)$. As described later in Section 4, the part of the algorithm that is parallelized 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: $v$: 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**$f(\{x\}) := v(\{x\}), O\{x\} := \{x\}$ **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**$f(C) \geq v(C)$ **then** $O(C) := C^*$ Where $C^*$ maximizes right hand side of line 4 **end if**
6: **if**$f(C) < v(C)$ **then** $f(C) := v(C) \wedge O(C) := C$ **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 memory requirements by half removing instant access to the final splittings. These final splittings are easily retrieved as outlined in algorithm 2. Essentially, all coalitions in $C \in CS^*$ whose value in $f$ is not equal to the initial value in $v$, 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[6].

Having described the DP algorithm, we next elaborate on the data structure we use to extend DP into GPU-CSG and then go on to detail the algorithm.

## 3 GPU-CSG

GPU-CSG parallelises the key steps of the DP algorithm. This is a non-trivial process (as we will see) as it requires us to be efficient in memory access and in sharing the computation among the threads on the GPU so that access to global

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

INPUT: $v$ : 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** $f(C) \neq v(C)$ **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** $CS^*$

memory is reduced while minimising the need for synchronising the threads. To this end, we first propose a memory efficient technique to store coalitions and their values in memory. This is important because (as discussed in Section 2), the GPU typically have relatively smaller amounts of memory. Moreover, we discuss different ways to navigate through the search space of coalitions as DP requires splitting coalitions into their components to evaluate each sub-problem separately.

### 3.1 The Data Structure

How 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 are needed to be represented for each coalition structure, the coalition structure itself and its value in $f$, we propose data structures that aim to allow a large number of agents to be represented in spite of the exponential growth of the input (i.e., $2^n$).

In order to minimize memory usage we applied several techniques as follows. First, we represent each coalition as a fixed sized array of values, where each value represents a distinct agent. While this may seem intuitive at first, 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]. When solving the CSG 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 therefore equal to:

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

Given the same constraints, representing the coalition structure as an fixed sized integer, it is only needed to store one entry per coalition, which is all together $2^n - 1$ 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 at run-time. That means the only memory constraint on the system is the storage for all coalition values.

Given that the index to the value of a coalition structure is as a binary representation, the distribution of, coalition structures that are lexicographically adjacent with the same cardinality will be evenly distributed over the fixed sized array. This rise two constraints. First, the whole fixed sized array have to fit into the memory of the GPU, limiting the number of agents that can be represented. Second, fetching values will be in non-coalesced manner causing waste of bandwidth, which to some degree is resolved by detecting values that can be shared between coalition structures, as described in Section 3.1.

Now, having defined coalitions as binary arrays, we next move on to explain how we create splittings of such coalitions efficiently for our algorithm to go through sub-solutions of the CSG problem.

**Coalition Structure Splittings** Splittings as mentioned are pairwise disjoint subsets of a coalition structure, given the coalition structure $C = \{f_3, f_2, f_0\}$ the splittings are shown in table 2. In order to generate the splitting there are essentially three methods used: $initShift$, $initialSplit$, and $nextSplit$.

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

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

The function $initShift$, as detailed in Algorithm 3 and partly illustrated in Figure 2, is necessary to setup the environment for all calls to $initialSplit$. It takes as input the coalition structure that should be evaluated and the index $n$ which represent the index for the coalition structure in the two dimensional array $shift$. The general idea is to put into the shared $shift$ array which members the

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

```
1: t := C
2: count := 0
3: while t > 0 do
4:     index := FindFirstSet(t)                    Finds first bit set in t
5:     shift_{n,count} := index
6:     nullBit(t, index)                           Sets one bit in t to zero
7:     count++
8: end while
9: return shift
```

coalition structure have in an ordered fashion. As illustrated in Figure 2, the coalition $C = \{f_3, f_2, f_0\}$ will have the values 0, 2 and 3 put in the array, these numbers represent which unique members it contains. It does so by using the bit operation $FindFirstSet$ which return the index of the first set bit, where the index represent which unique member in the coalition structure. It uses $FindFirstSet$ to find the first set bit in the coalition structure, and sets the bits index in the $shift$ array. Remove that member from the coalition structure by setting that bit to zero and scan for next set bit again, do so until all members have been identified and the coalition structure is empty. The two-dimensionality of the $shift$ array is to have one row in the array for each coalition structure that is being evaluated by the kernel.

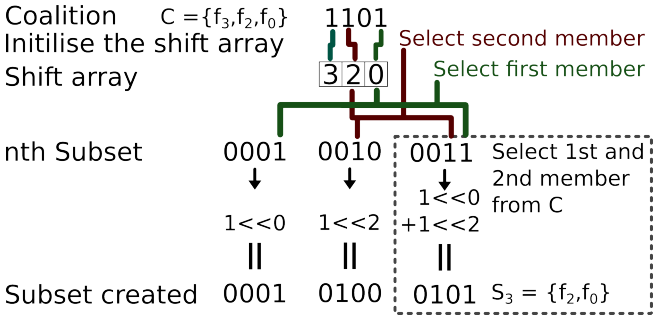


**Fig. 2.** How initShift and initialSplit works

Further, using these values with the function *initialSplit* described in Algorithm 5 to generate an initial splitting works as follows: Given the index $n$ as input that is an ascending integer value, which represent the $nth$ subset we want to create, and $C^*$ which represent the row in $shift$ for coalition structure $C$. The idea is to map the bits set in $n$ to bits in the coalition structure $C$, effectively transforming members of $n$ to members in $C$. It does so by using $FindFirstSet$ to find the indexes of set bits in $n$, where the index will eventually represent the $n$th ordered member of coalition structure $C$. First, take the indexes of bits

set in $n$ and use them to reference a column in row $C^*$ in the shift array, this will yield one member value $v$ from coalition structure $C$. To add that member to the subset to be created, take a single bit and binary left shift it $v$ places, and add it to the subset. Remove the set bit in $n$ and continue with previous operations until $n$ is empty.

To illustrate as shown in Figure 2, the third subset represented as the binary value of 0011, have its bits set in the indexes zero and one. In the shift array for coalition structure $C$, the indexes zero and one represent the first and secondly ordered members, which is agents $f_0$ and $f_2$ with values $v$ of 0 and 2. To form the subset, as described earlier, add together a single bit left shifted 0 places with a single bit left shifted 2 places. This will finally yield the subset 0101 which have the members $f_0$ and $f_2$ in it. Then to generate the next splitting, a call to *nextSplit* in Algorithm 4 is needed that works through a recurrence relation 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 $index : n$ $index : C^*$

1: $t := n$
2: $S := 0$
3: **while** $t > 0$ **do**
4: $index := FindFirstSet(t)$ Finds the first set bit in t
5: $S := S + leftShift(1, shift_{C^*,index})$ Shifts 1 left with the value in shift and adds
6: $nullBit(t, index)$ Sets the bit of t to 0
7: **end while**
8: **return** $S$

**Collisions between Splittings of Coalitions** Given that each thread evaluates several splittings over numerous coalition structures in parallel, it is bound that splittings on two coalitions that overlap will have splittings that collide. A collision means that splittings of coalition 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 coalitions intersect, there exist at least one subset of a splitting shared between them.

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

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

Specifically, the number of splittings in common is how many splittings can be done on all the members of the intersection. 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.

For example, given two coalition structures $C_1 = \{f_0, f_1, f_2\}$ and $C_2 = \{f_0, f_1, f_3\}$ that intersect on $f_0$ and $f_1$, all splittings value where one subset only contain one or both members may be shared in-between them. I.e. the values for $S_1 = \{f_0\}$, $S_2 = \{f_1\}$, and $S_3 = \{f_0, f_1\}$ are interchangeable.

**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: $\quad \upsilon_{0,z+1} := \upsilon_{0,z}$ Sets the other splittings value as its own
6: **else**
7: $\quad \upsilon_{0,z+1} := f(C_1)$ 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: $\quad \upsilon_{0,z+1} := \upsilon_{0,z+1} + \alpha$ Add the other splittings value to its own
15: **else**
16: $\quad \upsilon_{0,z+1} := \upsilon_{0,z+1} + f(C_1)$ Else add its own value
17: **end if**
18: **return** $\upsilon$

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 generate the initial splittings and fetch the value for the first coalition structure. It then trough lines 4 to 8 checks whether the splitting of the second coalition structure is equal to the first. If so it assigns to it the first coalition structure value, else it fetches its own value from global memory. Finally, the last lines do the same thing as above just that it does that on the pairwise disjoint subset, and the first splitting next value is stored in a 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. This can easily be extended throughout each warp by intra-warp communication as long as the arithmetic overhead is less than the time it takes to fetch from memory.

**Reduction** As the evaluation of each coalition means finding the splitting of the coalition which maximizes the value of the coalition structure, 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 between lines 28 to 40 in algorithm 7, as well as outlined in figure 1, where the lines detail the propagation and reduction of values throughout the scopes.

Now, turning to how the above computations are spread on the GPU, at a 'thread level' (see Section 2.1), each thread evaluates a number of splittings to determine their most-valued splitting. Further, at warp level, all threads inside the same warp concurrently exchange their largest value 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 in shared memory, is one value per warp. With each most-valued value from each warp moved into shared memory, a number of threads corresponding to half of the number of warps will be active, these active threads will for each thread evaluate two values in shared memory to again determine the most-valued value. Half the number of active threads, these will now evaluate the most-valued values by previous iteration. Iterate until the most-valued value is determined. Finally, a single thread using the most-valued value, will try to update the value in global memory if it is greater using atomic functions. We next detail how we bring all of the above techniques together in our algorithm.

## 4 The GPU-CSG Algorithm

To parallelise the steps of the DP algorithm, GPU-CSG (depicted in Algorithm 7) evaluates several coalition structures at each kernel invocation, where each thread evaluates a total of two splitting per coalition structure. As memory is constrained per thread in terms of registers and shared memory, the algorithm will evaluate coalition structures in batches. The total number of coalitions structures evaluated for each kernel is denoted by the constant $\alpha$, while how many coalitions per batch is determined by the constant $\beta$. The input it takes is the array which holds the values for each coalition structure assigned to $f$, the initial coalition structure $C_0$ that it will evaluate and generate the consecutive coalition structures with. $\Omega$ and $\Psi$ are invariants which guards against out of bound calculations, where $\Omega$ represent the largest coalition structure available and $\Psi$ is the number of splittings per coalition structure.

As each kernel evaluates several coalition structures yet only one is supplied as input, all other coalition structures need to be generated. As the function

to generate coalition structures works through a recurrence relation only one thread is allowed to generate those. Line 1 checks whether the thread has an index of 0, meaning the first thread in each block. Line 2 follows by setting the first index of the coalition structure array in shared memory to be the coalition structure received from input. Lines 3 till 5 generates the remaining $\alpha - 1$ coalition structures using the function *nextCoalition*, which takes as input a coalition structure and will output the next lexicographically adjacent coalition structure. We synchronise the threads at line 7 to make sure next step is not executed before each coalition structure is in shared memory. Next, it will now generate the $shift$ array using the function $initShift$ as described in Algorithm 3, as this can be done in parallel, line 8 evaluates if the thread-id is less than the number total number of coalition structures evaluated by the kernel denoted by $\alpha$. Then $\alpha$ threads will run the function $initShift$ concurrently, to later converge at the synchronisation step at line 11.

Generating and fetching the splittings is the next step of the algorithm. It starts by entering the loop at line 12 where it directly sets all values in array $\upsilon$ to zero. The evaluation at line 14 determine if the thread is eligible to evaluate splittings, $\psi$ represents the which $n$th splitting the thread should create, if it is greater than the total number of splittings $\Psi$, it will directly goto the reduction part of the algorithm. Else, the thread will start to fetch splittings of the first batch of coalition structures at line 17. It does so by first evaluating at line 18 if the coalition structure it is currently evaluating is greater than $\Omega$, meaning a coalition structure generated from *nextCoalition* is out of scope for this problem size, if so it goes to the reduction part of the algorithm. Line 21 sets $S$ to one of the pairwise disjoint subsets of the first splitting the thread should fetch, by using the function *intialSplit* inputting $\psi$ and the coalition structure evaluated. Line 22 then sets $\upsilon$ in row 0 to the addition of two values from the value array $f$, they are the values of the two subsets that make up one splitting, one is $S$, given by the *initialSplit* function, the other is generated by taking the set difference between the coalition structure in $\Delta$ and subset $S$. Next, generate one subset of the next splitting by calling *nextSplit*, fetch the values and add them together again into row 1 in $\upsilon$. Final, increment the counter $z$ and loop until the whole batch have been fetched.

Reduction is used to find the largest value among all threads in an efficient manner, starting at line 27, all threads have converged due to synchronisation. The reduction is also made in batches of $\beta$ coalitions and the for loop starts by at line 29 comparing if the second splittings value is greater than the first splitting, if so, write the second splittings value to the first splittings value index in $\upsilon$. The next step at line 32 is the warp reduction, its input is the threads maximum value, and returns the warps maximum value. The CUDA function $shfl\_xor$ allows threads inside the same warp to exchange values with each other, by using that function, *warpReduction* will exchange and compare values with other threads in an orderly fashion, converging at the maximum value for the warp. Next, from line 33 to 36, the first thread of each warp will assign the maximum warp value to $\Upsilon$ in shared memory in row $i$, where $i$ represent

the $i$th warp the thread belongs to. When all warps have put their value to shared memory, find the maximum value using the function *blockReduction*, which simply goes over all values to find the greatest one. Finally, thread 0 will try update the value in global memory using atomic functions as there may be more than one block that evaluates the same coalition structure. Increment $x$ with $\beta$ and loop again.

We next empirically evaluate GPU-CSG and compare it against the DP algorithm. The aim is to test whether our memory and computation efficient techniques do permit significant speedups, particularly given that the allocation of threads on the GPU cannot be exactly controlled.

## 5 Empirical Evaluation

In this section we detail two experiments we evaluate the performance of GPU-CSG. First, we evaluate the two techniques to share the splittings among the threads we presented in Section 3.1 to identify whether there are any significant differences in performance. Second, we compare the run-time of GPU-CSG against an optimised version of the DP algorithm (implemented in the C-programming language).

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 are evaluated in parallel for a total of 32 coalition structures visited. The CPU DP algorithm is run single-threaded on an 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. Final, each data-point were an average of ten samples excluding the extrapolated values.

### 5.1 Experiment 1: Splittings Sharing and general bottlenecks

Figures 3 and 4 show the number of clock cycles in logarithmic scale each code section consumes, which was measured on a complete run on 22 agents. Each code section is denoted as Checkpoint $n$ referred to in algorithm 7, the amount of cycles is measured from the end of the previous checkpoint $n-1$ to the end of the current checkpoint $n$. The general bottleneck in both figures show that the stage of generating splittings and fetching the values (Checkpoint 4) consumes more than half of the runtime, while generating the coalition structures (Checkpoint 1) only consumes around 20% of the runtime. The difference between the two figures is that figure 4 uses shared splittings as discussed in Section 3.1, and the relative time spent generating splittings and fetching (Checkpoint 4) its values is slightly under 50%, versus almost 60% of the time without sharing splittings. It can be concluded that whilst the method that share splittings still spends most

of the time fetching memory, sharing splittings is beneficial in order to reduce memory transactions, henceforth improving the sharing of splittings makes a significant impact on the overall performance by means of algorithm 6. It can also be noted from figure 5 that the GPU implementations where one utilizes internal sharing of splittings is twice as fast than the one without at 29 agents.

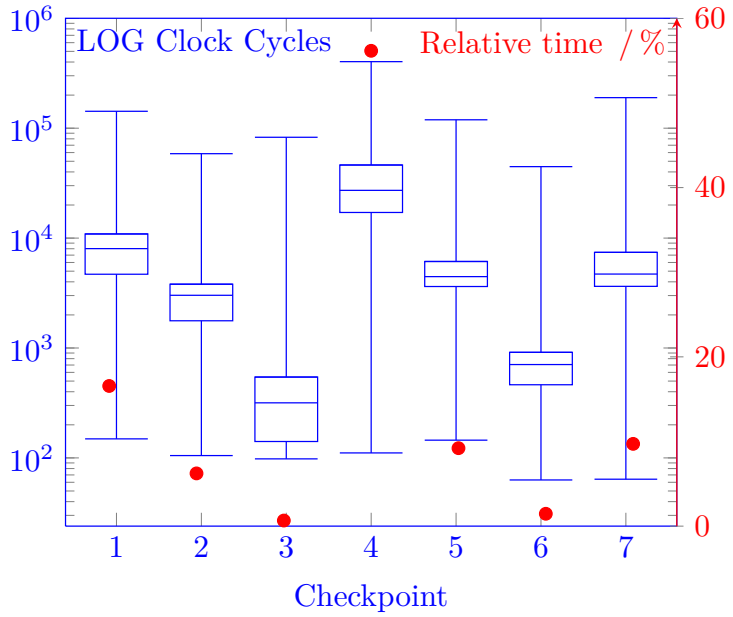


**Fig. 3.** LOG Clock Cycles between checkpoints and relative time for each code segment

### 5.2 Experiment 2: GPU-CSG v/s DP

Figure 5 shows the difference between the CPU and GPU algorithms. 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 GPU-CSG employs 1.5 hours, while the CPU bound algorithm was extrapolated to show that it would take up to 83 hours to complete. This is a speedup factor of over 55, which is expected to grow with the number 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.

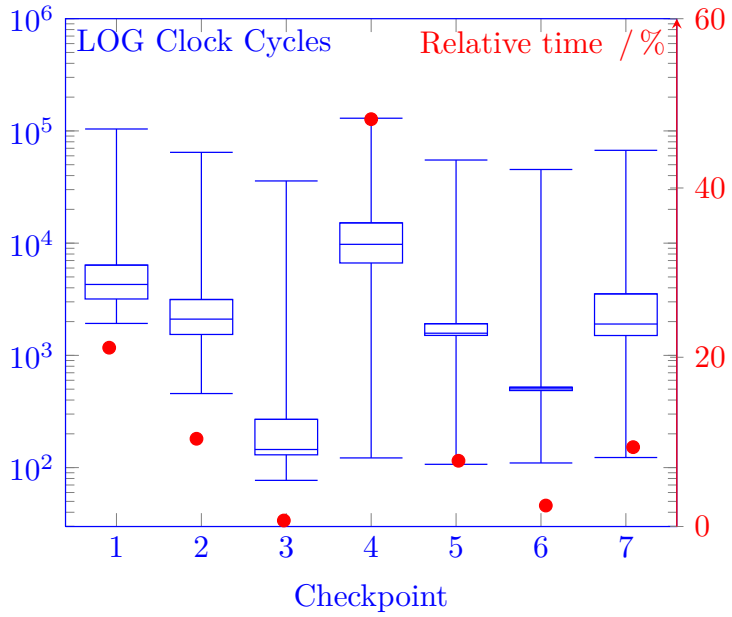


**Fig. 4.** LOG Clock Cycles between checkpoints and relative time for each code segment, (shared splittings)

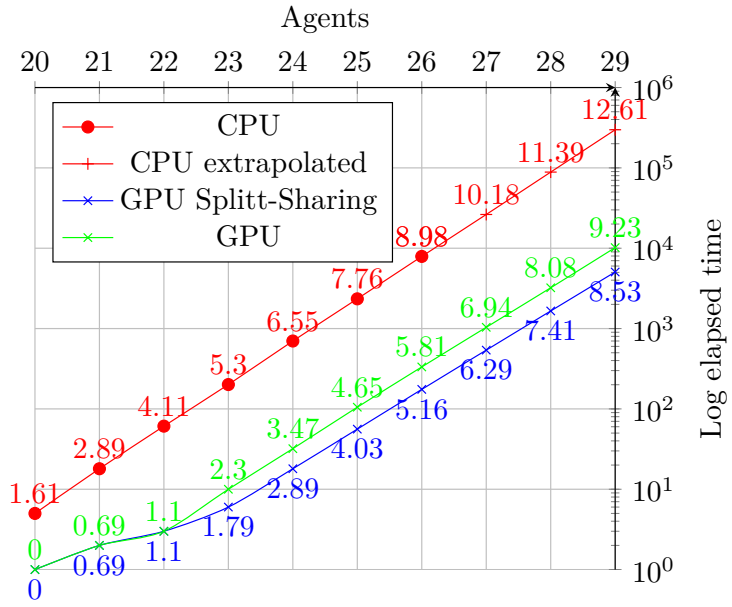


**Fig. 5.** LOG elapsed time for N agents

## 6 Conclusion

This paper presented the first GPU-based solution for the combinatorial optimisation problem of Coalition Structure Generation. Our algorithm, GPU-CSG, is shown to efficiently use memory access and thread allocation on the GPU in order to speed up the computations performed by a dynamic program for the CSG problem. In so doing, it is able to outperform the DP algorithm by up to 55 times, reducing the time taken to solve the problem for 29 agents to 1.5 hours as compared to 83 hours previously. Given the promising results of this initial work, we aim to develop new solutions for other similar optimisation problems in the future.

Further work would include incorporating the IDP technique and exploring further options to reduce memory bandwidth such as a more extensive collision detection. At the moment of writing an algorithm implementing IDP and extensive collision detection have a speedup factor of 314 times at 28 agents.

## References


1. V. Boyer, D. El Baz, and M. Elkihel. Solving knapsack problems on gpu. *Computers & Operations Research*, 39(1):42–47, 2012.
2. A. M. H. Rothkopf, A. Pekec, and R. M. Harstad. Computationally manageable combinatorial auctions. *Management Science*, 44(8):1131–1147, 1995.
3. 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.
4. T. Rahwan and N. R. Jennings. Coalition structure generation: Dynamic programming meets anytime optimization. In *AAAI*, pages 156–161, 2008.
5. T. Rahwan and N. R. Jennings. An improved dynamic programming algorithm for coalition structure generation. In *AAMAS*, pages 1417–1420, 2008.
6. 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.
7. 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.
8. 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.
9. T. Voice, S. D. Ramchurn, and N. R. Jennings. On coalition formation with sparse synergies. In *AAMAS*, pages 223–230, 2012.

**Algorithm 7** The GPU-CSG 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 coalition structures that are being evaluated |
| $\beta$ | The number of coalition structures that are evaluated per batch |

**Variables**

| | |
|---|---|
| $\Upsilon$ | A shared array containing warps maximum values |
| $\upsilon$ | A local array containing one of the threads value |
| $\Delta$ | Holds all the coalition structures that will be evaluated |
| $tid = threadIdx.x$ | The thread index inside the block |
| $\psi := \lambda * (tid + blockdim.x * blockIdx.x)$ | Initial subset construction index |

**Start of algorithm**

1: **if** $tid = 0$ **then**
2: $\quad\Delta_0 := C_0$ Thread 0 sets the first coalition from input
3: $\quad$**for** $i := 1$ to $\alpha$ **do**
4: $\quad\quad\Delta_i := nextCoalition(\Delta_{i-1})$ Generate the next coalitions to evaluate
5: $\quad$**end for** Checkpoint 1
6: **end if**
7: $syncthreads()$
8: **if** $tid < \alpha$ **then**
9: $\quad initShift(\Delta_{tid}, tid)$ Initialise the shift array
10: **end if**
11: $syncthreads()$ Checkpoint 2
12: **for** $x := 0$ to $\alpha$ **do**
13: $\quad$Set all values in $\upsilon$ to 0 Reset the values to 0
14: $\quad$**if** $\psi \geq \Psi$ **then**
15: $\quad\quad$goto line 28 The threads subset index is larger than the number of splittings
16: $\quad$**end if** Checkpoint 3
17: $\quad$**for** $z := 0$ to $\beta$ **do**
18: $\quad\quad$**if** $\Delta_{z+x} \geq \Omega$ **then**
19: $\quad\quad\quad$goto line 28 The coalition is outside the range of the calculation
20: $\quad\quad$**end if**
21: $\quad\quad S := initialSplit(\psi, \Delta_{z+x})$ Generate the $\psi$th splitting from a coalition
22: $\quad\quad\upsilon_{0,z} := f(\Delta_{z+x} \backslash S) + f(S)$ Fetch the splittings value
23: $\quad\quad S := nextSplit(S)$ Generate the next splitting of the coalition
24: $\quad\quad\upsilon_{1,z} := f(\Delta_{z+x} \backslash S) + f(S)$ Fetch the splittings value
25: $\quad\quad z := z + 1$ Increment counter and move on the next coalition
26: $\quad$**end for** Checkpoint 4
27: $\quad syncthread()$
28: $\quad$**for** $z := 0$ to $\beta$ **do**
29: $\quad\quad$**if** $\upsilon_{1,z} > \upsilon_{0,z}$ **then**
30: $\quad\quad\quad\upsilon_{0,z} := \upsilon_{1,z}$ Sets the biggest value
31: $\quad\quad$**end if**
32: $\quad\quad\upsilon_{0,z} := warpReduction(\upsilon_{0,z})$ Finds maximum value in scope of warp
33: $\quad\quad$**if** $tid\%32 = 0$ **then**
34: $\quad\quad\quad i := tid/32$ Which nth warp does the thread belong to
35: $\quad\quad\quad\Upsilon_{i,z} := \upsilon_{0,z}$ First thread in each warp writes maximum value
36: $\quad\quad$**end if**
37: $\quad$**end for** Checkpoint 5
38: $\quad blockReduction()$ Finds maximum value in scope of block Checkpoint 6
39: $\quad$**if** $tid = 0$ **then**
40: $\quad\quad atomicUpdate()$ First thread does an atomic update on global memory
41: $\quad$**end if** Checkpoint 7
42: $\quad x := x + \beta$
43: **end for**
44: **return** $f$