AAMAS 2012 Submission in LaTeX Format*

Paper XXX Kim Svensson†
University of Southampton
Southampton SO17 1BJ
United Kingdom
ks6g10@ecs.soton.ac.uk

ABSTRACT

A parallel algorithm and implementation using the Nvidia Cuda framework of the dynamic programming algorithm, due To Rothkopf et al. in order to solve the combinatorial auction problem and the complete coalition structure formation problem. Using a consumer grade Nvidia GTX 660 Ti for computational test, processing and implementation in order to compare against an sequential version running on a Intel Xeon W3520 with a clockspeed of 2.67GHz. Test results show a speedup factor of 40 with 29 agents with an ever increasing divergence between the CPU and GPU algorithm in terms of growth. Techniques of memory compression, utilizastion of reduntant operation and further optimisations to implement in order to raise performance.

1. INTRODUCTION

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 numbers of evaluations carried out [1].

*For use with aamas2012 .cls
†Something

Appears in: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, Conitzer, Winikoff, Padgham, and van der Hoek (eds.), June, 4–8, 2012, Valencia, Spain.

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 2.4.4. Shared memory is by its name 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.

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

Regarding 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 cach lines of size $32 * wordsize$, meaning cachelines of 32, 64, and 128 bytes each when pulling the primitives char, short and int respectively. As a result, if the memory reads within a warp is not coalesced within consecutive words, the effective bandwidth will drop immediately.

2.4 Model and Cuda Implementation

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 ??. 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.

The coalition structure itself may be represented as an integer where the n'th agent of the coalition structure is represented by setting the n'th bit in an binary integer. Given four agents $A = f_1, f_2, f_3, f_4$, 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 and most beneficial splitting.

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

2.4.1 Coalition Structure Splitting Collisions

2.4.2 Coalition Structure Splittings

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
```

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'
```

2.4.3 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.

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

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

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

Normaly, each splitting would be fetched from memory resulting in it being fetched several times. Instead, before fetching the values of the splitting, determin if any previoius coalitions contained that splitting and if so evaluate the coalition by re-using the previoiusly fetched value. This will enable a cost effective way to reduce the relative random access to memory making the effective bandwidth higher.

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

```
1: t := n
2: S := 0
3: while t > 0 do
4:     index := FindFirstSet(t)
5:     S := S + leftShift(1, shift_index,C)
6:     nullBit(t, index)??
7: end while
8: return S
```

2.4.4 Reduction

figure* to enclose the figure and its caption. 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 the most valued one.

The reduction is done on four levels of scope as seen on lines ?? to ?? in algorithm ??. On thread level, each thread evaluate a number of splittings to determine their most valued splitting, then 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 threads within the same warp, allowing for a substantial reduction in shared memory use. On thread block level, the threads are split up into two groups, active and inactive threads. The active threads will compare its value against a corresponding inactive thread, then half the number of active thread. Iterate until the maximum value have converged into one single thread holding the maximum value for the whole thread block. Finally, the single thread will attempt to update the global memory using atomic functions given that the value of the coalition in table f is less.

2.5 Algorithm

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 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

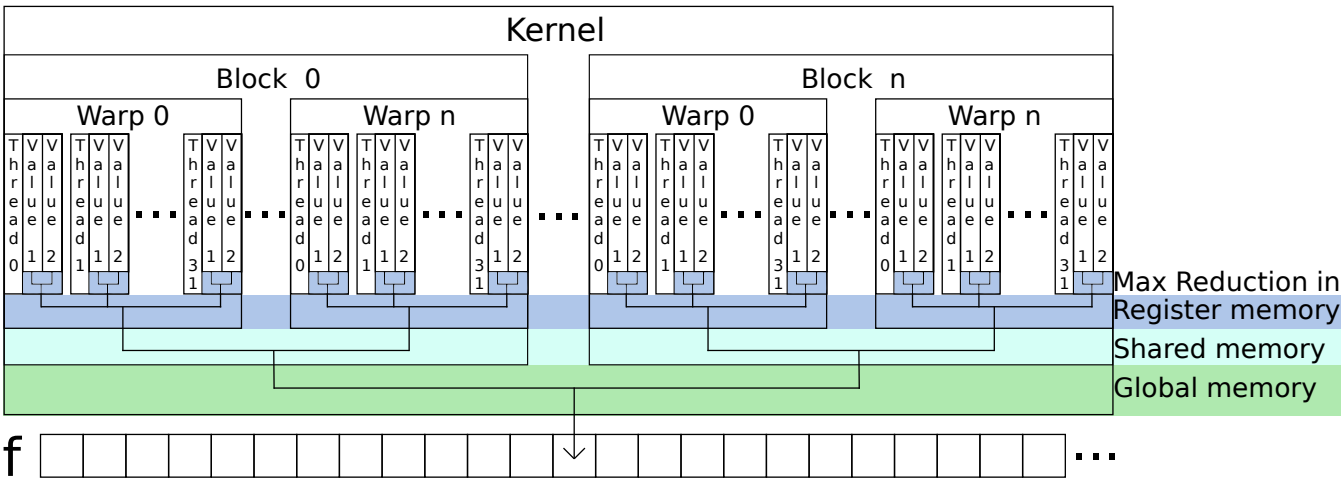
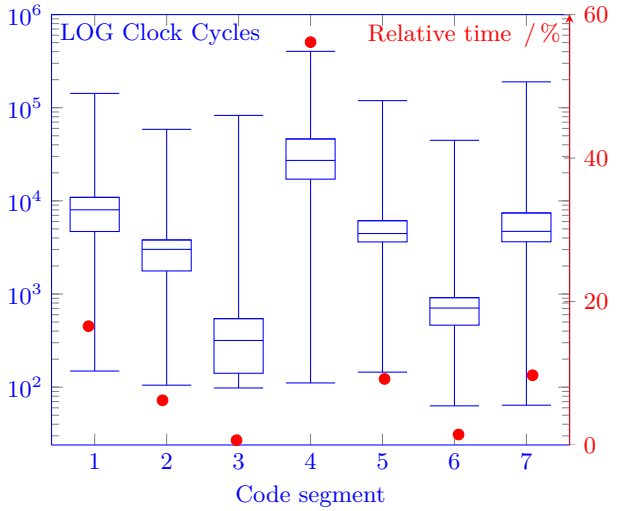


Figure 1: Outline of reduction across thread, warp, block and kernel

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

LOG Clock Cycles and relative time for each code segment



4. DISCUSSION

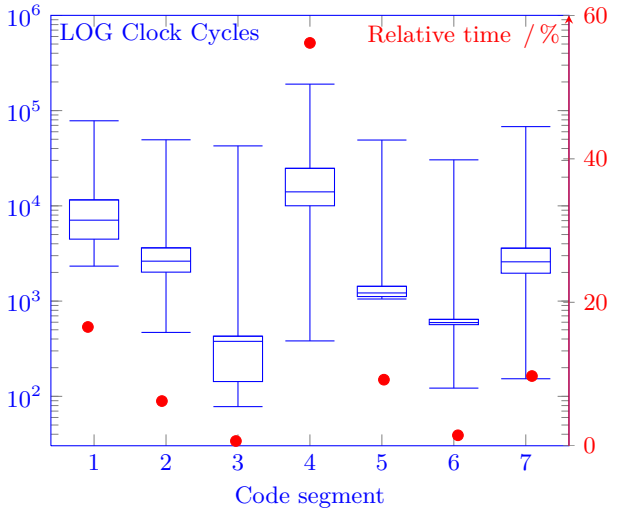
5. CONCLUSION

This paper show using the GPU to solve to Complete coalition structure formation problem is a viable option. However due to limitations of the hardware solving a problem

6. REFERENCES

[1] 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.

LOG Clock Cycles and relative time for each code segment



APPENDIX

A. HEADINGS IN APPENDICES

The rules about hierarchical headings discussed above for the body of the article are different in the appendices. In the **appendix** environment, the command **section** is used to indicate the start of each Appendix, with alphabetic order designation (i.e. the first is A, the second B, etc.) and a title (if you include one). So, if you need hierarchical structure *within* an Appendix, start with **subsection** as the highest level. Here is an outline of the body of this document in Appendix-appropriate form:

A.1 Introduction

A.2 The Body of the Paper

A.2.1 Type Changes and Special Characters

A.2.2 Math Equations

Algorithm 6 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
$confpkernel$
$nparallelconf$

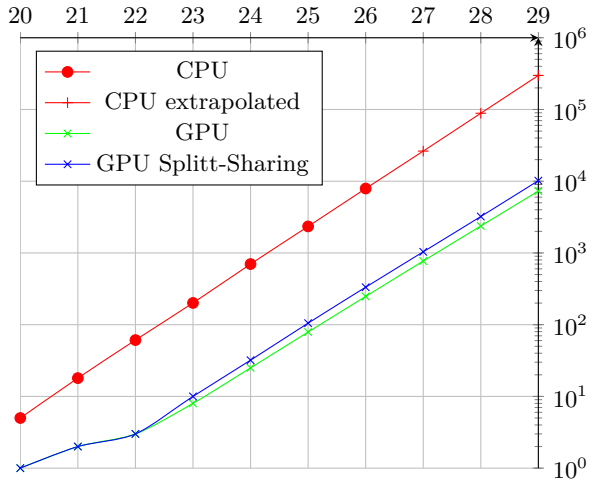
Variables

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

if $tid = 0$ **then**
  $conf_0 := C_0$
  for $i := 1$ to $confpkernel$ **do**
    $conf_i := nextCoalitoin(conf_{i-1})$
  end for
end if
if $tid < confpkernel$ **then**
  $initShift(conf_{tid})$
end if
for $x := 0$ to $confpkernel$ **do**
  Set all values in υ to 0
  if $\psi \geq \Psi$ **then**
    goto postfetch
  end if
  for $z := 0$ to $nparallelconf$ **do**
    if $conf_{z+x} \geq maxval$ **then**
      break
    end if
    $C := initialSplit(\psi, conf_{z+x}$
    $\upsilon_{0,z} := f(conf_{z+x} \backslash C) + f(C)$
    $C := nextSplit(C)$
    $\upsilon_{1,z} := f(conf_{z+x} \backslash C) + f(C)$
  end for
  for $z := 0tonparallelconf$ **do**
    if $\upsilon_{1,z} > \upsilon_{0,z}$ **then**
      $\upsilon_{0,z} := \upsilon_{1,z}$
    end if
    $warpReduction(\upsilon_{0,z})$
    if $tid\%32 = 0$ **then**
      $i := tid/32$
      $\Upsilon_{i,z} := \upsilon_{0,z}$
    end if
  end for
  $blockReduction()$
  if $tid = 0$ **then**
    $atomicUpdate()$
  end if
  $x := x + nparallelconf$
end for

Figure 2: hello



Inline (In-text) Equations.

Display Equations.

A.2.3 Citations

A.2.4 Tables

A.2.5 Figures

A.2.6 Theorem-like Constructs

A Caveat for the TEX Expert

A.3 Conclusions

A.4 Acknowledgments

A.5 Additional Authors

This section is inserted by LaTeX; you do not insert it. You just add the names and information in the `\additionalauthors` command at the start of the document.

A.6 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references) to create the .bbl file. Insert that .bbl file into the .tex source file and comment out the command `\thebibliography`.

B. MORE HELP FOR THE HARDY

The aamas2012 .cls file is based on the sig-alternate.cls file that is itself chock-full of succinct and helpful comments. If you consider yourself a moderately experienced to expert user of LaTeX, you may find reading it useful but please remember not to change it.