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 clock-speed 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, utilization of redundant operation and further optimization's 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, maximized 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 ?? 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.

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 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[2].

*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 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 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 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 2.4.3. 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.

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 access to the global memory, certain access patterns must be followed in order maximize performance. For global memory, it is important to note that each load request from memory will fetch in cache-lines of size $32 * word - size$, meaning cache-lines 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 cache-line) within consecutive words, the effective bandwidth will drop immediately.

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

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

Table 3: Initialize shift on $C = \{f_0, f_2, f_3\}$

Binary	$1_3 1_2 0_1 1_0$	
Shift	0 3 2 0	Result
Splitting 1	$0001 : 1 << shift[0]$	0001
Splitting 2	$0010 : 1 << shift[1]$	0100
Splitting 3	$0011 : 1 << shift[0]$	
	$+1 << shift[1]$	0101

constraint on the system is the store all the values for all the coalition structures.

2.4.1 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 3. 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 nth splitting should be created, finds the index of its set bits, and finally left shifts each bit with the value in the shift array its index reference to.

nextSplit works through a recurrence relation which means in order to have concurrent threads independent of each other, an initial splitting for each thread have to be calculated using initialSplit. initialSplit works by first generating an packed index array of which bits are set in the coalition structure using initShift. Given which nth 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.

2.4.2 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 split-

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

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

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

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

Algorithm 6 Fetch using Collision detection

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

```
1: C_0 := initialSplit(ψ, CS_0)
2: C_1 := initialSplit(ψ, CS_1)
3: υ_{0,z} := f(C_0)
4: if C_1 = C_0 then
5:     υ_{0,z+1} := υ_{0,z}
6: else
7:     υ_{0,z+1} := f(C_1)
8: end if
9: C_0 := CS_0 \ C_0
10: α := f(C_0)
11: C_1 := CS_1 \ C_1
12: if C_1 = C_0 then
13:     υ_{0,z+1} := υ_{0,z+1} + α
14: else
15:     υ_{0,z+1} := υ_{0,z+1} + f(C_1)
16: end if
17: υ_{0,z} := υ_{0,z} + α
18: return υ
```

tings 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.

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. It works 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 are done.

2.4.3 *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, 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 1.

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

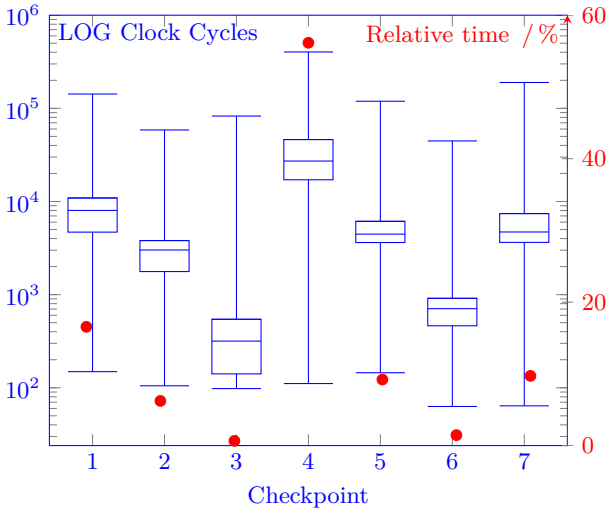
The algorithm starts by initializing all the variables needed further on, as generating the next coalition structure works through a recurrence relation, only the first thread will update the shared memory. Continuing, next step is to generate the shift array, this can be done in parallel so a number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

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

Figure 2: LOG Clock Cycles between checkpoints and relative time for each code segment, not sharing splittings



4. DISCUSSION

Figures 2 and 3 shows the difference between employing shared splittings discussed in section 2.4.2. 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 3 showing the result of sharing splittings it can be concluded that whilst it still spends most of the time fetching memory, henceforth improving the

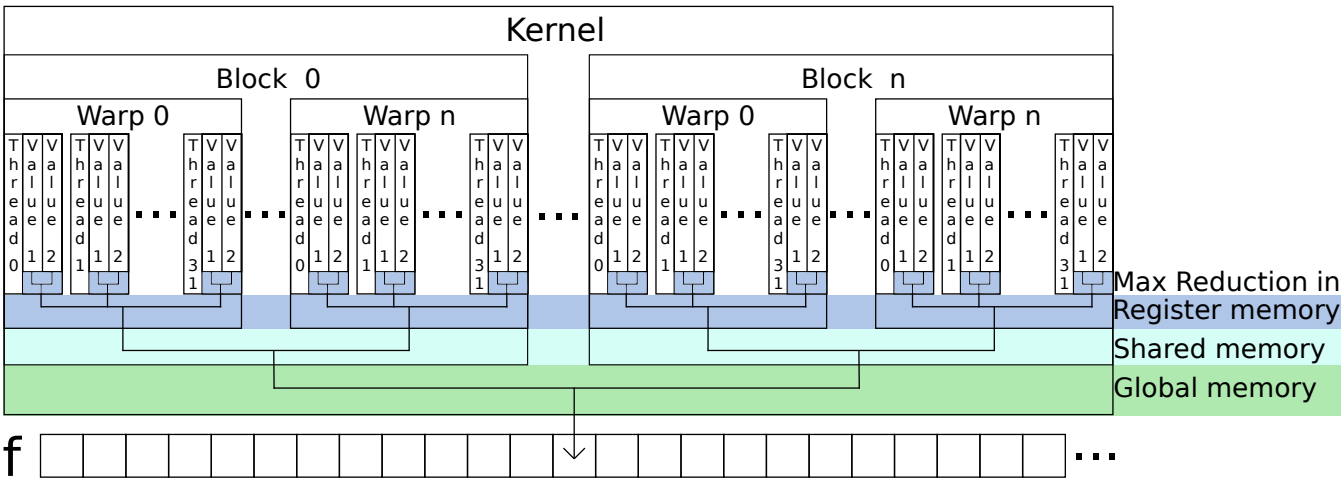
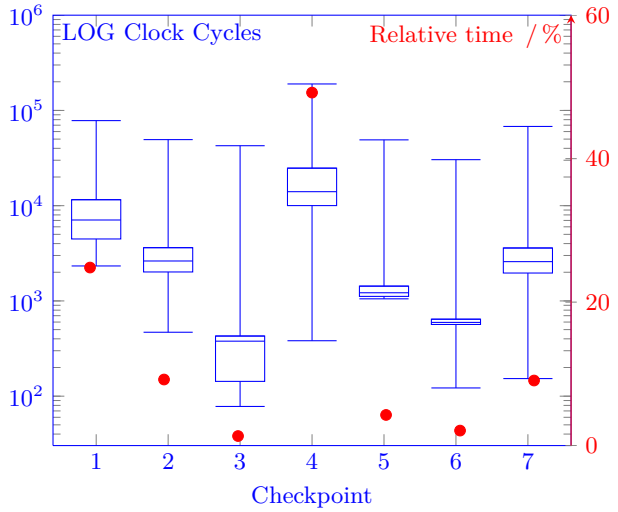


Figure 1: Outline of reduction across thread, warp, block and kernel

Figure 3: LOG Clock Cycles between checkpoints and relative time for each code segment, shared splittings

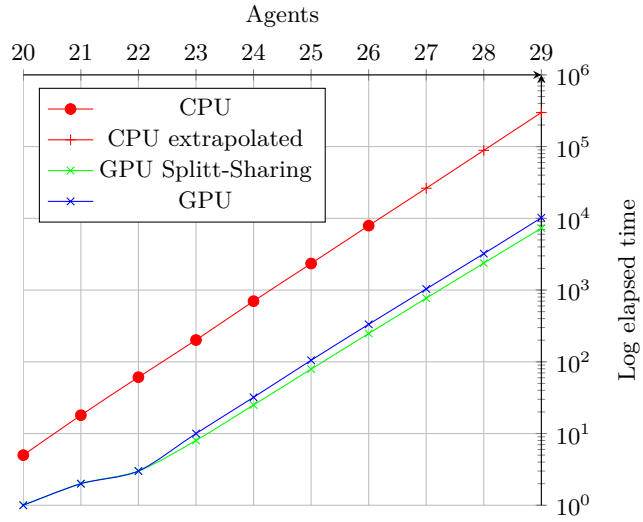


sharing of splittings make a significant impact on the overall performance.

Figure 4 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.

5. CONCLUSION

Figure 4: LOG elapsed time for N agents



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

6. FURTHER WORK

The notion of removing redundant fetches from memory can be expanded into fully constructing the coalitions bottom up,

7. REFERENCES

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

[2] T. Rahwan and N. R. Jennings. 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.

Algorithm 7 GPU implementation of the DP algorithm

Input

f The array which holds the bids

C_0 The first coalition structure 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**
    $conf_i := nextCoalition(conf_{i-1})$
  end for Checkpoint 1
end if
if $tid < confpkernel$ **then**
  $initShift(conf_{tid})$
end if Checkpoint 2
for $x := 0$ to $confpkernel$ **do**
  Set all values in υ to 0
  if $\psi \geq \Psi$ **then**
    goto postfetch
  end if Checkpoint 3
  for $z := 0$ to $nparallelconf$ **do**
    if $conf_{z+x} \geq maxval$ **then**
      break
    end if
    $C_z := initialSplit(\psi, conf_{z+x})$
    $\upsilon_{0,z} := f(conf_{z+x} \backslash C_z) + f(C_z)$
    $C_z := nextSplit(C_z)$
    $\upsilon_{1,z} := f(conf_{z+x} \backslash C_z) + f(C_z)$
    $z := z + 2$
  end for Checkpoint 4
  for $z := 0 tonparallelconf$ **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 Checkpoint 5
  $blockReduction()$ Checkpoint 6
  if $tid = 0$ **then**
    $atomicUpdate()$
  end if Checkpoint 7
  $x := x + nparallelconf$
end for