Binary GPU-Planning for Thousands of NPCs

Cardon Stéphane* and Jacopin Éric†
CREC
Écoles de Saint-Cyr Coëtquidan
Guer, France
Emails:
*stephane.cardon@st-cyr.terre-net.defense.gouv.fr
†eric.jacopin@st-cyr.terre-net.defense.gouv.fr

***Abstract*—Game Artificial Intelligence Engines of commercial games run on CPUs and not on GPUs. With more and more powerful GPUs and Cloud gaming, our vision is that GPUs will become the dedicated Game AI hardware, just as it now provides computing power for Game Physics (e.g. nVidia PhysX). In particular, we believe GPUs can run Game AI Planning, which computes plans in order to control the behaviors of Non-Player Characters (NPCs) in video-games. Our objective is an efficient online GPU-based Game AI Planning component with Cloud gaming as a target. We here report on our most recent implementation which can control thousands of NPCs each frame with only one GTX 1080 on the AI server, pushing thousands of plans to the client (PC or console).**

***Index Terms*—Cloud gaming, artificial intelligence, goal-oriented action planning (GOAP), graphics processing unit (GPU), real-time.**

I. Introduction

Modern video games immerse players in ever larger worlds in which non-player characters (NPCs), controlled by Game Artificial Intelligence (Game AI), are increasingly numerous and need to be more realistic, more "human". How can we offer players larger cities, teeming with NPCs living their "own" life, while CPU computing power is stabilizing? We believe that in the near future, Game AI will be boosted by Graphics Processing Units (GPUs) [2], [6], [16] and, on a larger scale, will be deported to GPU-accelerated clouds.

Our work focuses Game AI action planning which was first introduced in the game F.E.A.R. [13] as Goal-Oriented Action Planning (GOAP) [15]. In its first version, it made it possible to control teams of about ten NPCs and the average plan length was one or two actions (the numerical value calculated from our in-game data is 1.48 [11]). We observe that the average plan length has little or not increased since 2005 while the number of NPCs has increased from 10 to 50, as is the case in the game Shadow of Mordor, for instance [10]. 15 years later, the problem is the scaling up of GOAP to manage several hundred, even several thousand, NPCs online, with, if possible, longer plans; in this paper we here report our first results on a benchmark using a GPU to speed-up GOAP to the desired performance.

We first developed a GPU-based planner [4], [5] which proved to be about 100 times faster than its CPU version on the well-known example of the blocksworld [7]; for this new work, we started from the premise that a predicate (e.g. is the top of a block clear? is block-a above block-b? etc) merely returns a binary value, to encode a blocksworld state over the 32 bits of an integer. We here reuse this binary encoding for (game) predicates: is the position of the point of interest reached? Is the NPC facing the point of interest? Is it possible to move forward to reach the point of interest? As a result, the state of an NPC is a conjunction of boolean value, that is, an integer. An action transforms a state into another, and thus can be encoded as a pair of states; we decided otherwise, however. Indeed, actions have specific preconditions and effects and using a whole state would be a waste of memory space. We thus encode actions as functions checking specific bits of an integer, i.e. its preconditions, and modifying only some bits of an integer, its effects: when the preconditions are met, the new state is computed from the logical effects of the action encoded into the function. Algorithm 1 illustrates the function for action *facing the point of interest*: if a noise has been spotted and the NPC is not yet facing the noise then he turns to the noise direction. Note that actions as C++ classes were introduced in F.E.A.R. [14] and actions in Dana Nau's PyHop [8] uses Python functions to implement methods and operators, for example. Consequently, we encode the situation of each NPC with an integer and then send the vector of these integers to the planner which computes, on the GPU, the plans according to the goal of each NPC. These plans are then sent to the game engine for the execution of the NPC behaviors. We will describe our GPU-Planning in the next section. The experiments we carried out have allowed to control more than a thousand NPCs in 1.6 milli-second (10% of a frame when the frame rate is 60 frames per second); average plan length was above 2 actions.

The rest of this paper is organized as follows. In the next section, we detail how we encode a planning problem so as to solve it with a GPU using breadth-first search. We then present our results on a first benchmark. We conclude with a brief discussion and highlight perspectives of our work.

II. Game AI Binary GPU-Planning

The main features of GOAP are [14]: states as array of values[1], actions as C++ classes, action costs, procedural

[1]This feature alone makes GOAP much closer to the Sequential Action Structure (SAS Planning) [1] than to STRIPS [9] as reported in [14].

978-1-7281-4533-4/20/$31.00 ©2020 IEEE

action	pre	pos
MoveToInterest	[F,T,F,?,F,?,?,?]	[F,F,F,?,T,?,?,?]
FaceInterest	[T,F,?,?,?,?,?,?]	[F,T,?,?,?,?,?,?]
BypassRight	[F,T,T,F,?,?,?,?]	[F,T,F,F,F,?,?,?]
BypassLeft	[F,T,T,T,?,?,?,?]	[F,T,F,F,F,?,?,?]
Attack	[?,?,?,?,T,T,F,F]	[?,?,?,?,F,F,T,F]
Eat	[?,?,?,?,?,F,T,F]	[?,?,?,?,F,F,F,T]
Wander	[?,?,?,?,T,F,F,F]	[?,?,?,?,F,F,F,T]

TABLE I: Actions for the Zombies benchmark; T stands for `true`, F for `false` and ? for unknown or useless; bits $[b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8]$ in a state correspond to the following predicates: [InterestSpotted, InterestFaced, BlockedForward, BlockedRight, InterestReached, MealsPresent, MealsReady, Glut].

preconditions and the use of A* to build plans from the goal state to the initial state. To begin with, we consider states as 32-bit integers, actions as functions over these 32 bits, and leave out both costs and context preconditions. Search thus becomes breadth-first: completeness (i.e if a solution exists then it shall be build) is at the cost of $O(b^d)$ where b is the branching factor (the number of actions applied to a state) and d is the depth (the solution length) of the search from the initial search node. We eventually decided to search from the initial state to the final state to ease debugging. In this section, we detail how we reduced space complexity so that our planning algorithm can run on a GPU; throughout this section, we use a set of zombies looking for food to illustrate our purpose.

A. Networks of action segments

We begin with an informal presentation. A planning problem is made of an initial state, a final state and a set of actions. As we represent predicates as bits and states as integers, we encode each action a as a function $a : \mathbb{N} \to \mathbb{N}$. Algorithm 1 details the function which encodes the action allowing an NPC to face a point of interest (&, |, and $\sim$ respectively correspond to the bitwise operators **and**, **or** and **neg**): both the preconditions and the postconditions of this action concern only a subset of the bits of a state. Table I gives the bit values which need to be checked as preconditions and the bit values after the execution of each action; '?' marks bits that are irrelevant for each action.

We furthermore gather actions into disjointed sets so that all the actions in a set concern only certain relevant states; we call these sets *action segments*. Our objective is to reduce the branching factor: in a given state, we only consider the actions of the action segments relevant for that state. Then, we order action segments in a directed acyclic graph [3] (see figure 1 for an illustration) which we call a *network of action segments* (NAS); the longest path of an NAS gives us an estimate of the depth of our planning search space and we can then allocate memory on the GPU.

We now turn to formal definitions. Let P be a set of predicates encoded over $|B|$ bits of an integer and let S be the set of states with $|S| = 2^{|B|}$. Let $\mathsf{pre}(a)$ be the preconditions of action a; it is a set of predicates to be checked so that a can be executed. Let $\mathsf{pos}(a)$ be the postconditions of action a; it is a set of predicates which shall be modified by the execution of action a. Formally, we can think of an action a as a function taking two sets of integers $\subset [-|B|..|B|]$, say pre and pos, and one state, say s, as parameters. The semantics of executing a are the following: a returns a state, says r, such that $r = a(\mathsf{pre}, \mathsf{pos}, s)$. If there exists $i \in \mathsf{pre}$ such that $s[b_i] = 1$ when $i < 0$ or else $s[b_i] = 0$ when $i > 0$ then a fails and returns $r = s$; otherwise a succeeds and returns r such that: let $r = s$ and for all $i \in \mathsf{pos}$, let $r[b_i] = 0$ when $i < 0$ and $r[b_i] = 1$ when $i > 0$.

Algorithm 1 $FaceInterest(e)$

Require: e a state
Ensure: state built from e where interest point is faced, if possible
  if $(e \wedge InterestSpotted) \wedge \neg(e \wedge InterestFaced)$ **then**
    return $(e \mid InterestFaced)$ & $(\sim InterestSpotted)$
  end if
  return e

To make sure that search shall not exceed the available memory on the GPU, our central idea is to pre-compute a directed acyclic graph which we call Network of Action Segments (NAS):

Definition 1: Let A be a set of actions; a *network of action segments* is a directed acyclic graph $\mathsf{NAS} = (V, E)$ such that:

1) $V \subseteq \mathcal{P}(A)$ is the set of vertices of NAS where each vertex v is a set of actions called an action segment such that:
   a) $\forall a \in v, \exists a' \in v, a \neq a'$ s.t. $\mathsf{pos}(a) \cap \mathsf{pre}(a') \neq \emptyset \veebar \mathsf{pre}(a) \cap \mathsf{pos}(a') \neq \emptyset$,
   b) $\exists a \in v$ s.t. $\forall a' \in v, a \neq a', \mathsf{pre}(a) \cap \mathsf{pos}(a') = \emptyset$,
   c) $\exists a \in v$ s.t. $\forall a' \in v, a \neq a', \mathsf{pos}(a) \cap \mathsf{pre}(a') = \emptyset$,
   d) $\forall n \geq 2, \forall (a_1, a_2, ..., a_n) \in \{v^n | \forall l < n, \mathsf{pos}(a_l) \cap \mathsf{pre}(a_{l+1}) \neq \emptyset\}, \mathsf{pos}(a_n) \cap \mathsf{pre}(a_1) = \emptyset$,
   e) $\mathsf{pre}(v) = \{p \in B | \forall a \in v, p \notin \mathsf{pos}(a)\}$,
   f) $\mathsf{pos}(v) = \{p \in B | \forall a \in v, p \notin \mathsf{pre}(a)\}$.
2) E is the set of directed edges of NAS such that $E = \{(i, j) \in V \times V | \mathsf{pre}(j) \cap \mathsf{pos}(i) \neq \emptyset\}$.
3) $\forall n \geq 2, \forall ((i_1, j_1), (i_2, j_2), ..., (i_n, j_n)) \in \{E^n | \forall l < n, j_l = i_{l+1}\}, i_1 \neq j_n$.

1a) to 1d) define a lattice structure over the actions of an action segment, while 2) asserts that a NAS is a directed graph and 3) that it is acyclic. Starting from the complete graph of $\underline{a}$ctions, it is possible to build the corresponding NAS in $O(\underline{a}^2)$ time, checking the above definition against each edge of the graph.

Figure 2 illustrates a NAS for our Zombies. Let's assume the goal of a Zombie is to eat. To achieve his goal, his behaviour can be segmented into three parts: moving, possibly obtaining food with his consumption, and wandering. More precisely, to define a point of interest as the place that may contain food. Obviously, if the point of interest does not contain food, the Zombie's behavior at that point will be limited to randomly generating another point of interest, in which there may be food. We thus define three joining predicates $InterestSpotted$, $InterestReached$, and $MealsPresent$ which respectively correspond to the three action segments $MoveToInterest$, $HaveMeals$ and $Wander$.

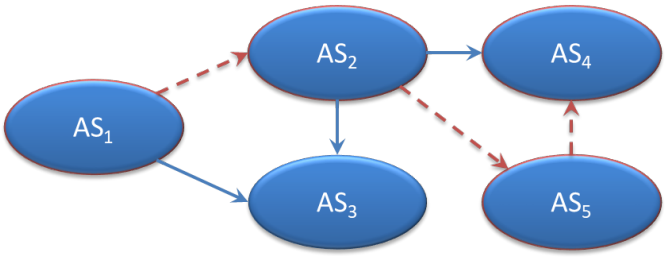

Fig. 1: NAS_a: an abstract NAS; red edges mark the longest path of 4 action segments starting from action segment AS_1.

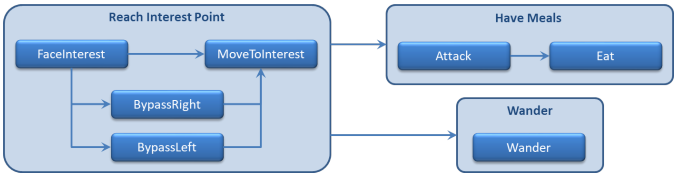

Fig. 2: NAS_z: our Zombies benchmark NAS.

For the *MoveToInterest* action segment, we have the predicates *FaceInterest*, *BlockedForward*, and *BlockedRight*. For the *HaveMeals* action segment, we have *MealsReady* indicating whether there is a dead body or a living entity.

In order to find an upper-bound of the size of our binary GPU-Planning search, we need to define a *predicate segment* in a NAS:

Definition 2: Let (V, E) be a NAS and as a vertex; a *predicate segment* of as is the set of predicates involved in any action of as: $\mathsf{PS}_{\mathsf{as}} = \{p \in B | \exists a \in \mathsf{as}, p \in \mathsf{pre}(a) \vee p \in \mathsf{pos}(a)\}$.

As a corollary to this definition, we have the following properties:

$$\forall (i,j) \in E, \mathsf{PS}_i \cap \mathsf{PS}_j = \mathsf{pos}(i) \cap \mathsf{pre}(j) \quad (1)$$

$$\forall (i,j) \in V \times V \text{ s.t. } (i,j) \notin E, \mathsf{PS}_i \cap \mathsf{PS}_j = \emptyset \quad (2)$$

Property (2) says that two disjoint action segments compute disjoint predicate segments. Predicates satisfying property (1) are sufficient to build the plan when considering the actions of a given action segment; we call them *joining predicates*.

Let $\Delta(\mathsf{NAS})$ be the maximal degree of a NAS (for instance, $\Delta(\mathsf{NAS}_a) = 3$ and $\Delta(\mathsf{NAS}_z) = 2$). The number of reachable states from the predicate segment PS of any action segment as of NAS is at most $2^{|\mathsf{PS}_{\mathsf{as}}|+\Delta(\mathsf{NAS})}$. Thus, an upper bound for the size of our binary GPU-Planning search space along a path of NAS is $\sum_{\mathsf{as}\in\mathsf{path}} 2^{|\mathsf{PS}_{\mathsf{as}}|+\Delta(\mathsf{NAS})}$. Therefore, an upper-bound for the size of our binary GPU-Planning search is $\max_{\forall \mathsf{path}\in\mathsf{paths}_{\mathsf{NAS}}} \sum_{\mathsf{as}\in\mathsf{path}} 2^{|\mathsf{PS}_{\mathsf{as}}|+\Delta(\mathsf{NAS})}$[2].

We use 5 bits to encode an action, thus allowing at most 32 actions. We would need $5 \times 32 = 160$ bits to encode a 32-action plan but we decided to limit ourselves to 64-bit integers, which allows plans of up to 12 actions; we use the last 4 bits to encode the plan length.

[2]This maximum amounts to the computation of the longest path in terms of predicate segments. The longest path in a directed acyclic graph (V, E) can be found in $O(|V|+|E|)$ [12]; it thus is possible to compute the longest path in terms of predicate segments in linear time as well, and therefore to compute this maximum in linear time. The description of such an algorithm is outside the scope of this paper.

B. Solving the Binary GPU-Planning Problem

Solving a planning problem means finding a totally ordered set of actions which, once applied in order from the initial state, will reach the goal state. In the case of NASs on a CUDA architecture, this solving is equivalent to a breadth-first search over a set of tuples (NPC number, NPC state, associated action plan to reach this state) as described in figure 3.

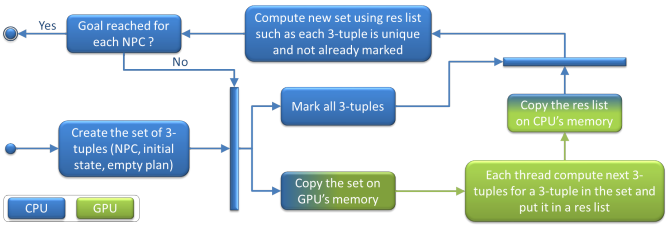

Fig. 3: GPU-Planning activity diagram.

The initial set of tuples is made of the (NPC number, current status of the NPC, empty plan) tuples for all NPCs. Then, as long as the goal of each NPC is not reached, two parallel computations will run: (1) on the CPU, we make an asynchronous copy of the current set of tuples to the GPU memory and then mark these tuples as already visited; (2) on the GPU, we execute a 256 threads block grid (i.e. $|T| = 256$ in Algorithm 2) computing the next tuple by applying the available actions, for each (a NPC, state, plan leading to this state) tuple (cf. Algorithm 2). We eventually copy the list of tuples to the CPU which takes care of updating the new set so that the tuples are unique and are not already marked.

Algorithm 2 $ThreadComputeNext(actions, current)$

Require: *actions* the set of possible actions
Require: *current* the current set of 3-tuples $(NPC, state, plan)$
Ensure: *next* contains the result set of 3-tuples
  $i \leftarrow threadIdx.x + blockIdx.x \times |T|$
  if $(i >= |current|)$ **then**
    This thread do nothing
  else
    $(npc, e, plan) \leftarrow current[i]$
    for all $a \in actions$ **do**
      $next[i \times |actions| + a] \leftarrow (npc, actions[a](e), plan \cup a)$
    end for
  end if

III. Benchmarking

We implemented a demonstrator involving thousands of Zombies in search of food. This benchmark follows a client-server architecture, described in figure 4, and uses the Unreal 4 game engine. It is split into several parts. The game, implementing Zombie actions, delegates the AI to a dedicated controller. The latter queries the game engine to obtain the state of each Zombie and then sends to the plugin a vector

of initial states. Then, it sends a scheduling request. Finally, at each frame, it queries the plugin to obtain the plans and in particular the action to execute. During this time, the controller stays on the previously obtained plans and does not send other requests. When the plugin receives the request, it sends a schedule request, with the initial states of the NPCs still alive, to the server via the network. Then it waits for the response. As it is a DLL-based plugin, it does not block the execution of the engine. The server application then receives a scheduling request and calculates the result as described in II-B. When the plans are computed, they are returned to the client through the network which makes them available to the AI controller.

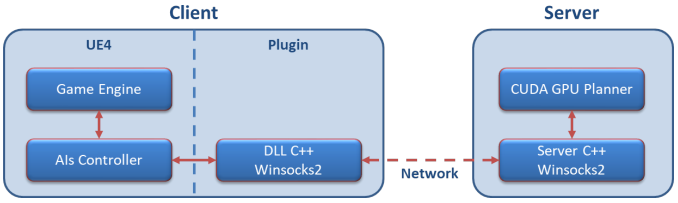

Fig. 4: Architecture of the Zombies benchmark.

This benchmark was implemented on a client and a server which are roughly equivalent in their configuration: Intel Core i7 + GetForce GTX 1080. We ran our benchmark in a local network over 3600 frames and measured: (a) the total time taken by the AI controller from sending the request to receiving the plans; (b) the time taken by the server (CPU+GPU) to compute the plans when receiving a request and (c) the time taken by the GPU to compute all the plans only. Figure 5 shows the average of the 3 measured runtimes, in milli-seconds : (a) red, (b) blue, and (c) green.

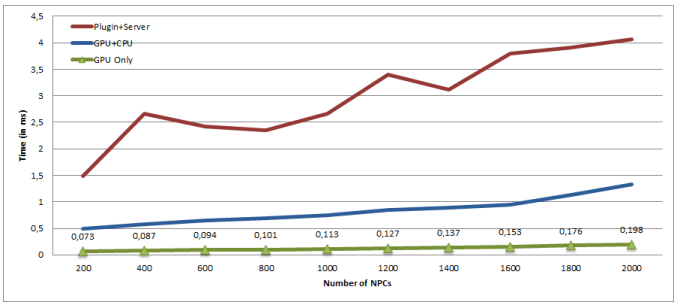

Fig. 5: Average runtimes for the Zombies benchmark.

IV. Conclusion

As far as we are aware, the benchmark presented in this paper is the first proof of concept for binary GPU-Planning. It shows that it is possible to control more than one thousand of NPCs in a frame, thanks to GPU computations driven by one CPU thread which we use to mark the new states of our binary planning problem.

If we can dream of multiple GPUs, then our numbers show that 8 GPUs coupled to 2 CPUs with 64Mb of RAM would allow to control more than 30.000 NPCs in one frame. Each GPU could focus on one or more types of NPCs and thus allow an enrichment of the diversity of NPCs. Moreover, the computing time on CPU+GPU being 10 times less important than the latency time of the network (a good ping to play is around 50 ms), the latter will prevail. The online control of tens of thousands of NPCs is therefore possible on a dedicated data server within a few frames between the planning request and its response.

With regard to our future work, we would like to list two of the possibilities open to us: (1) benchmarking with several GPUs, and (2) enrich our state representation from boolean values to integer values. For this second option, SAS planning can certainly be a source of inspiration.

References

[1] C. Bäckström, "Equivalence and tractability results for SAS$^+$ planning," in *Proceedings of 3rd on the Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, 1992, pp. 126–137.

[2] W. Blewitt, G. Ushaw, and G. Morgan, "Applicability of GPGPU computing to real-time ai solutions in games," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 5, no. 3, pp. 265–275, 2013.

[3] B. Bollobás, *Modern Graph Theory*, ser. Graduate Texts in Mathematics. Springer, 1998, vol. 184.

[4] S. Cardon and É. Jacopin, "CUDA constraint programming for AI gaming in the cloud," Poster at the nVidia GPU Technology Conference, March 2015. [Online]. Available: http://on-demand.gputechconf.com/gtc/2015/posters/GTC_2015_Game_Development_01_P5147_WEB.pdf

[5] ——, "Game AI planning: Which GPU for how many npcs?" Poster at the nVidia GPU Technology Conference, March 2016. [Online]. Available: https://on-demand.gputechconf.com/gtc/2016/posters/GTC_2016_Game_Development_GD_03_P6268_WEB.pdf

[6] A. Champandard and A. Richards, "Massively parallel AI on GPGPUs with opencl or C++," GDC AI Summit, March 2014.

[7] S. Cook and Y. Liu, "A complete axiomatization for blocks world," *Journal of Logic and Computation*, vol. 13, no. 4, pp. 581–594, 2003, Oxford University Press.

[8] N. Dana, "Game applications of HTN planning with state variables," Keynote talk at the ICAPS 2013 Workshop on Planning in Games, p. 21, 2013. [Online]. Available: https://www.cs.umd.edu/~nau/papers/nau2013game.pdf

[9] R. Fikes and N. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, no. 5, pp. 189–208, 1971.

[10] P. Higley, "GOAP at monolith productions," GDC AI Summit, March 2015.

[11] É. Jacopin, "Game AI planning analytics: The case of three first-person shooters," in *Proceedings of the 10th AIIDE*. AAAI Press, 2014, pp. 119–124.

[12] E. Lawler, *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.

[13] Monolith Productions, "F.E.A.R." October 2005.

[14] J. Orkin, "Applying goal-oriented action planning to games," in *AI Game Programming Wisdom 2*, S. Rabin, Ed. Charles River Media, 2003, ch. 3.4, pp. 217–227.

[15] ——, "Three States and a Plan: The A.I. of F.E.A.R." in *Proceedings of the Game Developer Conference*, 2006, p. 17 pages.

[16] D. Sulewski, S. Edelkamp, and P. Kissmann, "Exploiting the computational power of the graphics card: Optimal state space planning on the GPU," in *Proceedings of the 21st Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 2011, pp. 242–249.