Vietnam National University – HCMC
Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

UNDERGRADUATION PROJECT

KAITHY

Symetric Q-learning Gomoku AI

Project Supervisor's Report:
Dr. Cuong Pham-Quoc

Project by

Name	Student ID
Nguyễn Cao Minh Khánh	
Đặng An Thịnh	1413765

Contents

Acknowledgement

We dearly thank **Dr. Cuong Pham-Quoc** for teaching us all these time, making us work hard, pushing us forward and providing us with countless research advices. His willingness to give his time so generously has been very much appreciated. One simply could not wish for a better or friendlier supervisor.

Monday 15th May, 2017

Abstract

Gomoku is a fascinating game that has yet to be played well by a computer program due to its large board size and exponential time complexity. This report applies the reinforcement learning algorithm Q-learning to this game.The Q-Learning algorithm was proposed as a way to optimize solutions in Markov decision process problems and reduce algotithm developed bu us. With the emergence of CUDA, Nvidia's massively parallel GPU platform, Markov decision process can be parallelized by the GPU to enhance its performance.

Keywords: karthy,Q-learning, CUDA, karthy.

1 Introduction

1.1 Description

Artificial Intelligence (AI) inside modern games is one of the most computationally expensive features, and developers have implemented many optimizations to ensure performance. This project aims to create a GPU-accelerated AI that can take advantage of parallel computation hardware to evaluate high-cost "thinking" operations in real time.

1.2 Purpose

Gomoku is a traditional board game which is normally played by two people at a time. Often one may not have anybody to play with, thus requiring a computer player acting opponent. To make the game enjoyable, the computer player should be appropriately skilled. Using a simple reinforcement learning algorithm, called Q-learning, to create a computer player. And using parallel algorithms in CUDA to improve performance. The aim is to analyse the performance and efficiency of CUDA based parallel programming.

1.3 Terminology

1.3.1 Gomoku

A traditional board game usually played with paper and pencil.

1.3.2 Karthy tree

Karthy tree is a tree data structure in which each node has unboundedly child nodes. Nodes with children are parent nodes, and child nodes may contain references to their parents.

1.3.3 Reinforcement Learning

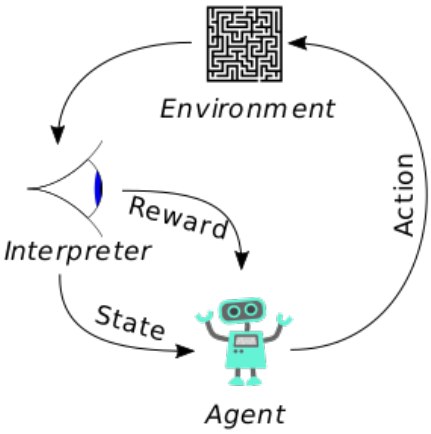

Figure 1: The typical framing of a Reinforcement Learning scenario

Reinforcement learning is an area of machine learning inspired by behaviorist psychology, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. Reinforcement learning differs from standard supervised learning in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected. Further, there is a focus on on-line performance, which involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

1.3.4 CUDA

Figure 2: CUDA Toolkit

GPU-accelerated CUDA(Compute Unified Device Architecture) libraries enable drop-in acceleration across multiple domains such as linear algebra, image and video processing, deep learning and graph analytics. We will talk more about this on CUDA section.

1.3.5 OpenCV

Figure 3: OpenCV logo

OpenCV (Open Source Computer Vision) is a library of programming functions mainly aimed at real-time computer vision.

1.3.6 Symmetric

A simple algorithm reduce the complexity of the board game.

1.3.7 Agent

An agent is a player in the game implementation, either a computer or an interface towards humans.

1.3.8 State

The state of the environment; the current game configuration.

1.3.9 Action

An action is a transition from one state to another, which is what a player does or can do in a specific state.

1.3.10 Feedback

An action generates feedback from the environment. The feedback can be either positive or negative. The feedback is used in the Q-learning algorithm for estimating how good an action is. The term reward is used for positive feedback and negative feedback is called punishment.

1.3.11 Learning rate

The learning rate is a parameter in the Q-learning algorithm, describing the relationship between old and new memories. A high value means that new memories take precedence over old memories and vice versa. A value of zero means that nothing will be learnt.

1.3.12 Discount factor

The discount factor is a parameter in the Q-learning algorithm, affecting the importance of future feedback. A high discount factor increases the importance of future feedback.

1.4 Theory

1.4.1 Karthy

Karthy is a narrow AI, a computer program developed by us to play Gomoku, a abstract strategy board game for two players similar to chess. The algorithm for Karthy is a combination of Karthy tree, Reinforcement Learning, parallel algorithms in CUDA.

1.4.2 Gomoku

Gomoku is an abstract strategy board game. It is known in Vietnamese as "Ca Ro". It is traditionally played with Go pieces (black and white stones) on a go board with 19x19 intersections. However, because pieces are not moved or removed from the board, gomoku may also be played as a paper and pencil game. This game is known in several countries under different names.

Black plays first if white did not win in the previous game, and players alternate in placing a stone of their color on an empty intersection. The winner is the first player to get an unbroken row of five stones horizontally, vertically, or diagonally.

1.5 Thesis Organization

Here's a brief overview of successive chapters.

Chapter 1 Introduction covers the basic concepts.

Chapter 2 Q-learning algorithm

Chapter 3 CUDA Architecture dives deep into Nvidia powered CUDA architecture, and compares it with CPU architecture.

Chapter 4 Symmetric explain how this algorithms work and effect of this project.

Chapter 5 Implementation discusses the actual implementation step-by-step and highlights the differences from the CPU implementation.

Chapter 6 Benchmark shows the testing environment and various benchmarking results from the aforementioned manycore implementation

Chapter 7 Conclusions assesses overall project goals and limitations and delivers future thoughts.

2 Q-learning algorithm

Q-learning is a model-free reinforcement learning technique. Specifically, Q-learning can be used to find an optimal action-selection policy for any given (finite) Markov decision process (MDP). It works by learning an action-value function that ultimately gives the expected utility of taking a given action in a given state and following the optimal policy thereafter.

To understand Q-learning it is crucial to understand how a problem is represented. It is possible to divide almost all problems into several situations, so called states. For example in the Gomoku game there is a state for each possible stone, meaning each different set of stone make up a different state. In each state a action can be taken, an action corresponds to what moves are legal in the game.

The simplest form of Q-learning stores a value for each state-action pair in a matrix or table. The fact that it requires one value for every state-action pair is one of the drawbacks of Q-learning; it requires a huge amount of memory. So, in this report, we store data to hard drive. A possible solution to this problem is to use an artificial neural network, although that approach is not covered in this report.

In every state the algorithm visits it checks the best possible action it can take. It does this by first checking the Q-value of every state it has the possibility to get to in one step. It then takes the maximum of these future values and incorporates them into the current Q-value. When feedback is given, the only value that is updated is the Q-value corresponding to the state action pair that gave the feedback in question. The algorithm for updating Q-values is shown below, where s_t and a_t corresponds to the state and action at a given time.

The transition rule of Q-learning is a very simple formula:

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old Q-value}} + \underbrace{\alpha}_{\text{learning rate}} \times [\overbrace{\underbrace{r_{t+1}}_{\text{feedback}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q(s_{t+1},a)}_{\text{max future Q-value}}}^{\text{expected discounted feedback}} - \overbrace{Q(s_t,a_t)}^{\text{old Q-value}}]$$

Figure 4: Q-learning formula

The fact that only one value is updated when feedback is given, gives Q-learning an interesting property. It takes a while for the feedback to propagate backwards through the matrix. The next time a certain state-action pair is evaluated, the value will be updated with regards to the states it can lead to directly. What this means is that in order for feedback to propagate backwards to the beginning, the same path need to be taken the same number of times the paths are long, each new iteration on the path propagating the effects backwards one step.

If the algorithm always took the best path it knew of it is very likely it will end up in a local maximum where it takes the same path all the time. Even though the path it takes is the best it knows of that does not mean there is no better path. To counter this it is necessary to let the algorithm sometimes take a new path and not the best one, in order for it to hopefully stumble

upon a better solution.

If there is one, and only one, stable solution, that is which action is the best in each state, the Q-learning algorithm will converge towards that solution. Because of the back-propagation property of Q-learning, this however requires a large amount of visits to every possible state action pair.

There are two parameters to the algorithm, the learning rate α and the discount factor γ. These both affect the behavior of the algorithm in different ways.

The learning rate decides how much future actions should be taken into regard. A learning rate of zero would make the agent not learn anything new and a learning rate of one would mean that only the most recent information is considered.

The discount factor determines how much future feedback is taken into account by the algorithm. If the discount factor is zero, no regard is taken to what happens in the future. Conversely, when the discount factor approaches one a long term high reward will be prioritized.

3 CUDA Architecture

3.1 Background

Compute Unified Device Architecture (CUDA) is a recent parallel computing platform developed by Nvidia Corporation. It is neither the first parallel computing platform, nor the first GPU enabled parallel programming construct. However it is arguably the first popularized parallel GPU architecture that sees wide use in both commercial and academic fields. CUDA itself is a massively parallel architecture often containing more than 100 cores; the newer models even have more than 1000 cores. This poses a sharp contrast to traditional CPU architecture, which often contains no more than 4 to 8 cores.

While a CUDA GPU can have hundreds of cores executing instructions in parallel, each core is a lot slower than a CPU core. CUDA emphasizes running many threads concurrently as opposed to running a single thread very quickly. This architecture is also known as Single Instruction Multiple Data (SIMD). The underlying premise is that, if a task can be saturated with high enough volume of input data, the overall performance can improve due to the large scale batch processing. Figure 3.1 demonstrates the peak computing capability in $GFLOPS$[1] of CUDA graphic cards compared to Intel CPUs.

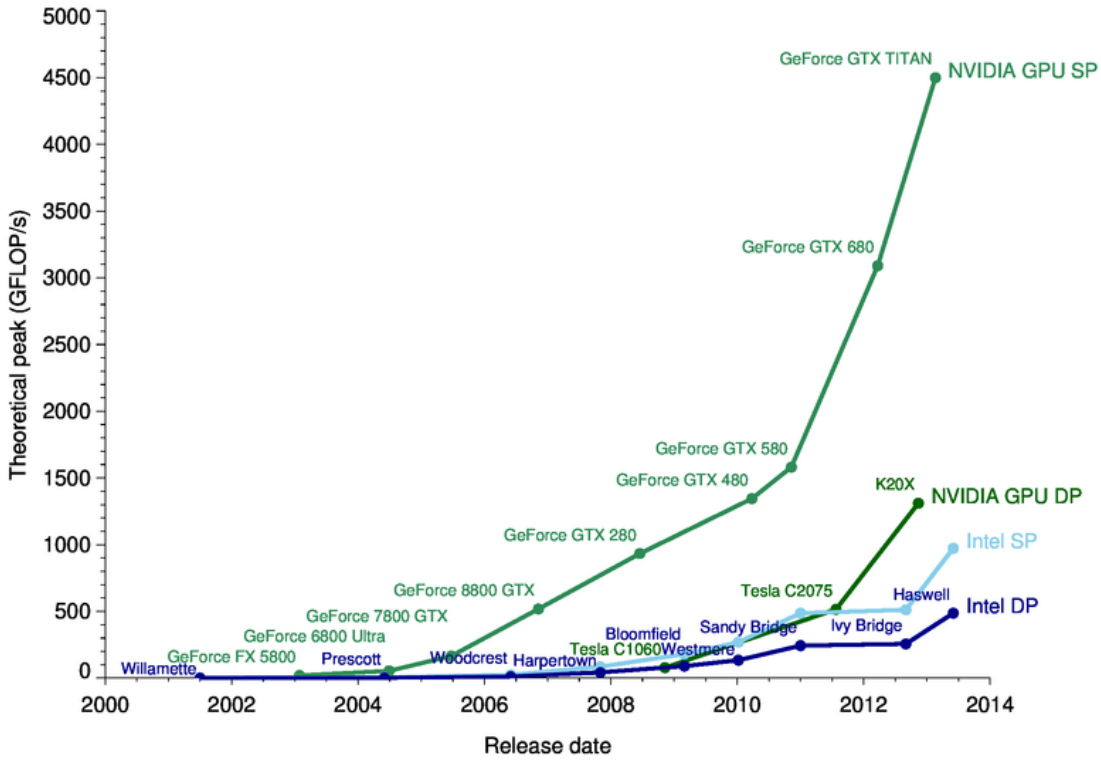

Figure 5: CUDA performance

The programming on the GPU is vastly different from doing so on the CPU, primarily due to the architectural difference and supported instruction set. Historically it is never an easy task to migrate an application running on the CPU to the GPU and also improving its performance. The beauty of the CUDA platform is that it allows the use of C Programming Language to write programs largely the in the same fashion with minimal modifications. Using C also favors portability, as a great number of programs were written in C language to run on the CPU can be relatively easily modified to run on a CUDA platform.

3.1.1 Advantages

Ease of programming: It's using C languages, which is straight-forward and general-purpose, thus more user friendly.

High performance: Harnesses the power of the GPU by using parallel processing; running thousands of simultanious reads instead of single, dual, or quad reads on the CPU.

Shared memory: Each multiprocessor within the device is allocated a region of shared memory, which is extremely fast and have low latency, and can be shared among different threads within

the same block.

3.1.2 Limitations

Small shared memory: Each multiprocessor contains 16-48KB of shared memory, it could pose a challenge to fit large indivisible graph with many nodes.

Thread configuration: Threads need to be cautiously set up and configured to achieve reasonable performance.

Hard debugging: CUDA-GDB does not support to Visual Studio or any Window version.

Complex programing: It require senior developer, who understand about variety, pointer, hardware architecture, ...

3.2 GeForce GTX 970

The GeForce GTX 970 card is used for this project. The $GeForce^{®}$ GTX 970 is a high-performance graphics card designed for serious gaming. Powered by new $NVIDIA^{®}$ $Maxwell^{TM}$ architecture, it features advanced technologies and class-leading graphics for incredible gaming experiences.

Figure 6: GeForce GTX 970

The below specifications represent this GPU as incorporated into NVIDIA's reference graphics card design.

GTX 970 Engine Specs:	
CUDA Cores	1664
Base Clock (MHz)	1050
Boost Clock (MHz)	1178
Texture Fill Rate (GigaTexels/sec)	109
GTX 970 Memory Specs:	
Memory Clock	7.0 Gbps
Standard Memory Config	4 GB
Memory Interface	GDDR5
Memory Interface Width	256-bit
Memory Bandwidth (GB/sec)	224

Figure 7: GeForce GTX 970 specifications

References