FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A GPU implementation of Counterfactual Regret Minimization

João Reis

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Henrique Lopes Cardoso (PhD)
Co-supervisor: Luís Teófilo (MSc)

June, 2015

A GPU implementation of Counterfactual Regret Minimization

João Reis

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Daniel Augusto Gama de Castro Silva (PhD)

External Examiner: Paulo Jorge Freitas Oliveira Novais (PhD)

Supervisor: Henrique Daniel de Avelar Lopes Cardoso (PhD)

June 26, 2015

Abstract

Notable milestones in the advancement of Artificial Intelligence have been achieved through solving games. Regret minimization is a technique that has seen a lot of use in the context of solving games in the past few years. In particular, Counterfactual Regret Minimization (CFR) is an algorithm that applies this technique and can be used to find equilibria in massive games. Therefore, one can use an algorithm like this to develop an agent with a very solid strategy for such games.

One issue with this algorithm is the amount of execution time it requires, especially when applied to large extensive games. To address this issue, games are usually abstracted which can lead to worse solutions.

This dissertation proposes an implementation of CFR that runs on the GPU, using CUDA, which is able to take advantage of the ability of GPUs to process many parallel streams of data. Using this approach, it is possible to reduce the execution time in some Poker variants, as our results demonstrate. This means that this approach has the potential to allow the computation of Nash Equilibria for games with a larger search space than before.

Resumo

Marcos notáveis no avanço da Inteligência Artificial foram alcançados através da obtenção de soluções para jogos. A técnica da Minimização do Arrependimento tem sido muito usada no contexto da obtenção de soluções para jogos nos últimos anos. Em particular, Counterfactual Regret Minimization (CFR) é um algoritmo que aplica esta técnica e pode ser usado para encontrar equilíbrios em jogos massivos. Portanto, pode-se usar um algoritmo deste género para desenvolver um agente com uma estratégia muito sólida para esses jogos.

Um problema com este algoritmo é a quantidade de tempo de execução que exige, especialmente quando aplicada a jogos com enormes árvores de pesquisa. Para abordar este problema, os jogos são geralmente abstraídos o que pode levar a soluções piores.

Esta dissertação propõe uma implementação do CFR que corre no GPU, usando CUDA, que é capaz de tirar partido da capacidade de GPUs para processar elevadas quantidades de dados de forma paralela. Usando esta abordagem, é possível reduzir o tempo de execução em algumas variantes de Poker, como demonstram os resultados. Isto significa que este método tem o potencial de permitir o cálculo de Equilíbrios de Nash para jogos com um espaço de pesquisa maior do que antes.

Acknowledgements

I would like to express my gratitude to my supervisor, MSc Luís Teófilo, for all the precious help he's given me from the very first day of this entire process, especially during the times I was feeling a little lost. This dissertation would not exist today without his guidance.

I would also like to thank my other supervisor, PhD Henrique Lopes Cardoso, for his important part in the final document review.

I also want to thank my family for the exceptional amount of support they provided. I'm very lucky to have a very patient family, especially my parents who spent the most time with me and always made sure I had everything I needed.

I must also acknowledge my friends, who were there for me when I was feeling exhausted and losing motivation.

João Reis

"A little more persistence, a little more effort,
and what seemed hopeless failure may turn to glorious success."

Elbert Hubbard

Contents

List of Figures

List of Tables

Abbreviations

CFR	Counterfactual Regret Minimization
GPU	Graphics Processing Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
RMA	Regret Minimization Agent
GPGPU	General Purpose Computing on Graphics Processing Units
OpenMP	Open Multi-Processing

Chapter 1

Introduction

1.1 Context

To solve games of incomplete information, a solution concept called Nash equilibrium is typically used. However, it is not feasible to compute or store these solutions for massive games such as Poker though it is possible to do it with approximations of Nash equilibria.

Regret Minimization Agents are agents that are used nowadays to compute such approximations of Nash equilibria for games with a large search space. These agents focus on minimizing the regret, i.e., the difference between the actual payoff and the payoff that would have been obtained if a different course of action had been chosen.

It turns out that there is a well-known connection between regret and the Nash Equilibrium concept. This was proven by having two agents of this type competing against each other and observing that both strategies converged to an approximation of a Nash Equilibrium. In light of this, it is possible to use an algorithm like Counterfactual Regret Minimization (CFR) to develop an agent with a solid strategy, i.e., a conservative strategy that does not focus on maximizing the gains against a specific opponent but shows good results against any type of opponent.

1.2 Motivation

The current state of the art algorithm for regret minimization – CFR – has some issues. The main issue is the fact that it requires a lot of computing resources and execution time. One way of trying to address this issue is to come up with abstractions of complex games. However, this leads to solutions of worse quality. Another way to try to solve this problem is by parallelizing the algorithm so that it runs on several processes or threads. However, as it is commonly known,

CPUs are quite limited on the number of cores they possess. This is not true for GPUs though, which are well-known for their heavy parallel processing and fast arithmetic operations. This leads us to believe that GPUs should definitely be taken into consideration when optimizing an algorithm like CFR.

There is no known CFR implementation that takes advantage of the power that GPUs offer, at the moment. There is a high probability that such implementation would speed up the algorithm to the point where it would be possible to develop strategies that are more solid and closer to a real Nash Equilibrium, as opposed to an approximation of one. This would be a result of requiring less amount of abstraction due to having a faster algorithm.

1.3 Objectives and Hypothesis

The main goal of this dissertation is to find out if a GPU implementation of CFR can be more efficient than other existing implementations. If this turns out to be true, then, using such implementation, it might be possible to solve abstractions of complex games faster than before or even reduce the amount of those abstractions.

The GPU implementation will be tested with different versions of Poker. Poker has a lot of variants, each one associated with different complexity and search tree size. Even a small variant of two player Poker can have almost 10^{18} game states (limit Texas Hold'em). Because of this, Poker has become a common measuring stick for performance in the context of finding solutions for very large extensive games.

1.4 Structure of the Dissertation

Chapter 2 explains fundamental concepts that one must have knowledge of in order to better understand this dissertation. It contains two subchapters: Game Theory and GPGPU. The first subchapter describes in detail several concepts of Game Theory that are used throughout this document and other publications in this area. The purpose of the second subchapter is to provide basic information about what GPGPU is and how it is going to be used in the context of this dissertation.

Related work is included in Chapter 3, which provides a brief review of existing literature in the same area, including existing CFR implementations.

Chapter 4 explains the three implementations that were developed during this dissertation: Recursive CFR, CPU Multithreaded CFR and GPU Multithreaded CFR.

Chapter 5 details the experiments that were made on each implementation and provides some analysis for the results.

Chapter 6 has the conclusions of this dissertation and proposes future work.

Chapter 2

Background

This dissertation requires the knowledge of some game theory concepts. In this chapter, these will be explained briefly with examples.

It also requires basic understanding of parallel computing with focus on CUDA, which will also be briefly introduced in this chapter.

2.1 Game Theory

Game theory is applied in the study of decision problems that can be modelled by games. This theory assumes that decision makers are rational, reason strategically and focus on maximizing their own utility.

Utility is essentially a number that represents the motivation of a player associated with a certain outcome. This means that they make decisions which help them achieve certain objectives and take into account the behavior of other decision makers.

2.1.1 Strategy and Strategy profiles

A *move* is an action that can be made by a player during a game and alters the game state. An example would be a player moving one of his or her pieces in chess from a square to another square during his or her turn.

A *strategy* defines which moves should be made by a player. Therefore, a strategy is an algorithm that tells the player what to do for every possible situation that can happen in a game. A strategy can be *pure* or *mixed*.

In very simple games, a pure strategy can be associated with a single action. In games with a temporal structure, a pure strategy can be a sequence of actions [1].

A mixed strategy defines the probability for each pure strategy that is available to a player. Because probabilities are continuous, there are infinite mixed strategies available to the player, assuming there are always at least two pure strategies available.

Another concept that should be understood is that of a *strategy profile*. A strategy profile is a set of strategies that contains only one strategy per player.

2.1.2 Cooperative and non-cooperative games

A game can be *cooperative* or *non-cooperative*.

In cooperative games, there is usually a system to ensure that commitments between players are kept and decisions or actions are usually made by groups of players. In these types of games, there is competition between groups of players rather than between individual players.

A non-cooperative game is a competition between individual players. It is possible for players to cooperate in these types of games, but there is no system in place to ensure commitments are kept, which means that any cooperation is self-enforced.

There is a simpler way to define these types of games, using the concept of utility. In cooperative games, the utility is shared, i.e., players that cooperate with each other focus on maximizing the utility that is shared between them. In non-cooperative games, utility is not shared between any player and players focus on maximizing their own utility.

For this dissertation, only non-cooperative games are relevant.

2.1.3 Zero-sum and non-zero-sum games

In a *zero-sum* game, a player's gain of utility is exactly balanced by the loss of utility of other players. Following the same logic, a player's loss of utility is also exactly balanced by the gain of utility of other players. Games like these are called zero-sum due to the fact that if the total gains are added up and the total losses are subtracted, the result of this sum will be zero. Naturally, in a *non-zero-sum* game, this sum will be less than or more than zero.

A zero-sum game is also called a *strictly competitive* game.

2.1.4 Sequential and simultaneous games

In *sequential* games, players have information related to previous moves made by other players. It is important to note that this information may be *perfect* or *imperfect*. These two concepts will be explained in the next subsection.

In *simultaneous* games, players either make their moves simultaneously or later players have no knowledge of the moves made by earlier players, therefore making them effectively simultaneous.

2.1.5 Perfect and imperfect information

In a game with *perfect information*, each player, when making a decision, is perfectly informed of all the events that have occurred until that moment. Chess is a good example of a game with perfect information. In a game of chess, both players can see every piece on the board and their positions which means the game state is visible. Furthermore, a player has knowledge of every move that has been made previously when deciding which move to make in his or her turn.

Games with *imperfect information* are the opposite, which means that the information available to each player is not complete. Most card games are of imperfect information. Because of hidden information, some game states are indistinguishable. There is a key concept relevant to this dissertation that is related to hidden information wich is the concept of information set. An information set is basically a set of game states that the player cannot tell apart.

Simultaneous games cannot be of perfect information because when a player is deciding which move to make, the actions of the other players may or may not be known.

2.1.6 Representation of games

There are two main ways to represent non-cooperative games: *extensive form* and *normal form*. Cooperative games are usually represented in the *characteristic function form* but, as these are not relevant for this dissertation, this form of representation will not be explained.

A game in normal form shows the strategies available to each player and the outcomes associated with each possible strategy profile. An outcome is usually represented by its utility. An alternative name for the concept of utility is *payoff*. This form of representation can also be called strategic form. Games in this form are sometimes referred to as strategic games. Normal-form games are simultaneous, so a strategic game can also be defined as a *"model of a situation in which each player chooses his plan of action once and for all"* [2].

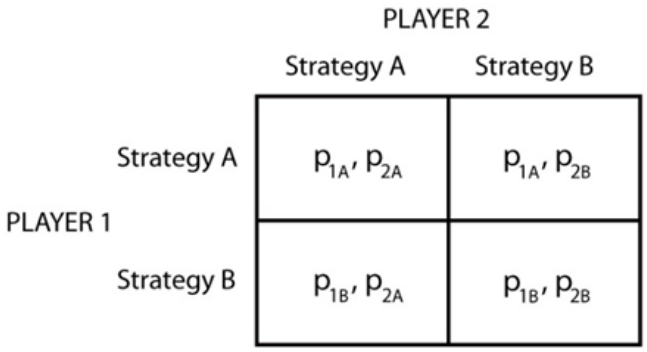

Figure 1: Matrix of a two player strategic game in which each player has two strategies [13]

A normal form game is usually represented by a matrix. Figure 1 shows an example of this type of matrixes. In this example, p_{nx} is the payoff for player n when he or she chooses strategy x.

The extensive form is used for games that have a time sequencing of moves. This form shows the temporal sequence of moves and all the information that is available to each player when he has to make a decision [3]. Thus, in these games, in contrast with normal form games, players can consider their plans of action whenever they reach a decision point [2].

Extensive form games are usually represented by decision trees. In this type of trees, each node represents a state in the game in which a player has to make a decision. The payoffs are specified for each path of the tree in its respective leaf node. Figure 2 shows a portion of the decision tree for a game called Kuhn Poker. This game is a simple 3-card poker game by Harold E. Kuhn. In Kuhn Poker, two players each bet 1 chip before cards are given to them. Three cards are shuffled and each player is given one card. A player does not know which card his opponent has. Play starts with player 1. A player may pass or bet on his turn. A bet is composed by an additional chip. If a player bets and the next player passes, in what is called a terminal pass, the play who made the bet gets all chips in the pot. If two passes or two bets are made successively, the player with the highest card takes all chips in the pot.

2.1.7 Nash Equilibrium

Nash Equilibrium is *"the most commonly used solution concept in game theory"* [2]. It is a strategy profile composed by strategies that the players cannot improve upon unilaterally.

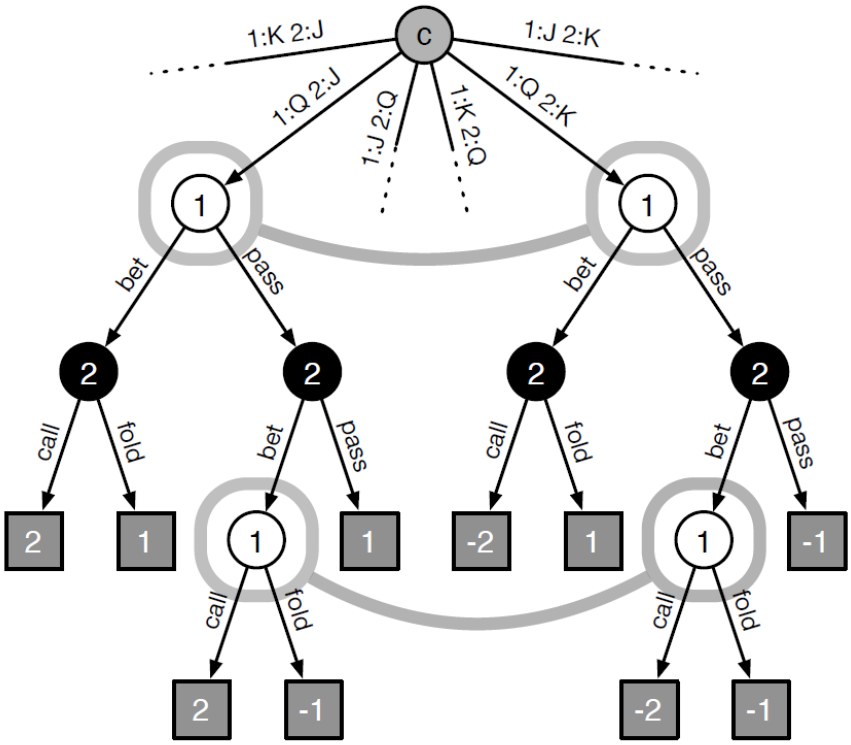

Figure 2: Portion of the extensive-form game representation of three-card Kuhn with jack (J), queen (Q) and king (K) as its three cards [6]

This solution concept is typically used in non-cooperative games with two or more players.

A typical example of a game that can be used to demonstrate the concept of Nash Equilibrium is the Prisoner's Dilemma. In this game, there are two suspects for a crime that are held in two separate cells. Prisoners can choose to confess or not to confess. Based on this, there are three situations that can happen:

- **One prisoner confesses and the other does not** – the one who confessed is freed and used as witness against the other, who will receive a sentence of four years.
- **Both prisoners confess** – they both receive a sentence of three years.
- **Neither prisoner confesses** – both will receive a sentence of one year.

	Don't Confess	*Confess*
Don't confess	$3, 3$	$0, 4$
Confess	$4, 0$	$1, 1$

Figure 3: Matrix for the Prisoner's Dilemma game [2]

The payoff of each strategy is four subtracted by the number of years the prisoner has to spend in prison, if he chooses that strategy. The strategies and their payoffs for this game are represented in Figure 3.

This is a strategic game that is simultaneous and non-cooperative. There is only one strategy profile that fits the definition of a Nash Equilibrium, which is the situation in which both prisoners confess with payoff of 1 to both prisoners. In this situation, neither player can improve the outcome by changing his strategy unilaterally which means this is indeed a Nash Equilibrium.

2.2 Poker

Poker games are used in the experiments of this dissertation, so it is useful to know what they are and some of their basic rules.

Poker is a family of *sequential* card games of *imperfect information* that involve betting. These games are also *zero-sum* and *non-cooperative*. They vary in the number of cards dealt to each player, i.e., *hole cards*, and the number of shared cards, i.e., *board cards*. They can also vary in the number of hidden cards in each player's hands but this is not relevant for this dissertation because in every variant used in the experiments, all cards in the players' hands is hidden.

In Poker, the first *round* of betting may involve forced bets, i.e., *blinds*. The first player bets and then action proceeds clockwise to the next player in the table. Players can match the previous bets of that round (*call*), increase the bet (*raise*) or forfeit (*fold*). The first player can also choose to not bet anything, in what is known as a *check*. The next players can also *check* in these situations. Depending on the specific variant that is being played, a certain amount of *board cards* are drawn from the deck in each round. When every player but one folds the remaining player wins the pot. If the last round finishes and there is still more than one player that has not *folded*, then a *showdown* occurs. In a *showdown*, the players that did not *fold* reveal their *hole cards* and evaluate their hands. The player with the best hand wins the pot. Specific rules about what ranks a hand above another one are not explained here as they are not necessary for this dissertation.

In *limit* Poker variants, the size of bets are fixed and there is a maximum number of *raises* allowed for each player per round. In *no-limit* Poker variants, such limits do not exist and players can even decide to *all in*, by placing the highest bet available to them. This dissertation will only cover *limit* Poker variants, because in *no-limit* variants players have an additional decision to make – choosing the right amount to bet in each situation – which further increases the complexity of the game. Even without this additional decision, *limit* Poker variants can still

be very complex. One example is *limit* Texas Hold'em, which is a two player *limit* Poker variant that has almost 10^{18} game states.

2.3 General Purpose Computing on Graphics Processing Units

GPGPU is an approach that consists of transferring the compute-intensive portion of an application to the GPU while leaving the rest of the application running on the CPU. This process is represented in Figure 4.

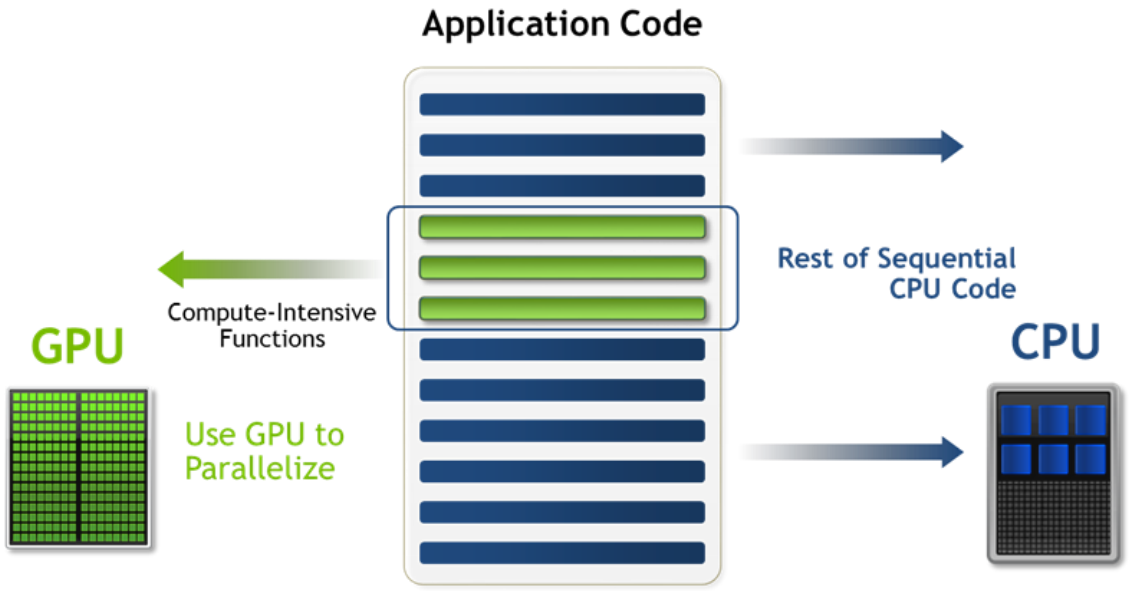

Figure 4: Slide taken from NVIDIA's presentation on CUDA: "Small Changes, Big Speed-up" [14]

CPUs contain a lot of hardware that perform control operations. This is necessary because CPUs were designed to be a generic piece of hardware that accept a wide variety of commands. Since GPUs are not general purpose, they do not have to pay this price of control hardware. Instead, their design focuses on optimizing raw throughput.

2.3.1 Compute Unified Device Architecture (CUDA)

CUDA is a parallel computing platform and programming model invented by NVIDIA. It is the dominant proprietary framework for those who wish to use the GPGPU approach.

In the CUDA programming model, code can be executed either on the CPU, which is the host, or on the GPU, which is the device. Code that is executed on the host can access memory

on both the host and device. Code running on the host can also launch kernels, which are functions executed on the device. Kernels are declared with the __global__ specifier.

```
void saxpy(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
```

Figure 5: Implementation of SAXPY in C [15]

Figure 6 shows a CUDA implementation of SAXPY in C and Figure 5 shows a sequential implementation of SAXPY in C. SAXPY stands for "Single-Precision A•X Plus Y". It is a function that takes as input two vectors of 32-bit floats X and Y with N elements each, and a scalar value A. It multiplies each element X[i] by A and adds the result to Y[i]. SAXPY is a good "hello world" example for parallel computation [4].

```
__global__
void saxpy(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

...
int N = 1<<20;
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);

cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
```

Figure 6: CUDA implementation of SAXPY in C [15]

In the CUDA implementation of SAXPY, the input vectors are copied from host memory to device memory before executing the kernel. Then, the kernel is executed by 4096 blocks of 256 threads, which is a total of $4096 \times 256 = 1048576$ threads (one thread per vector element). After the kernel execution, the output vector is copied back to host memory from device memory. The optimal execution configuration (number of blocks and threads per block) for a specific kernel can be found only by experimentation. In the case of the GPU

implementation of CFR, the optimal number of threads per block was found to be 64 and the number of blocks depend on the total number of threads needed by each Poker variant.

As this dissertation focuses on CUDA for the GPU implementation, it is also necessary to know and understand the different memory spaces that CUDA threads can access:

- **Per-thread local memory** – data is visible only to the thread that wrote it;
- **Per-block shared memory** – data is shared between all threads of the same block;
- **Global memory** – data is visible to all threads of the application, including the host.

Figure 7 illustrates the different types of memory in CUDA.

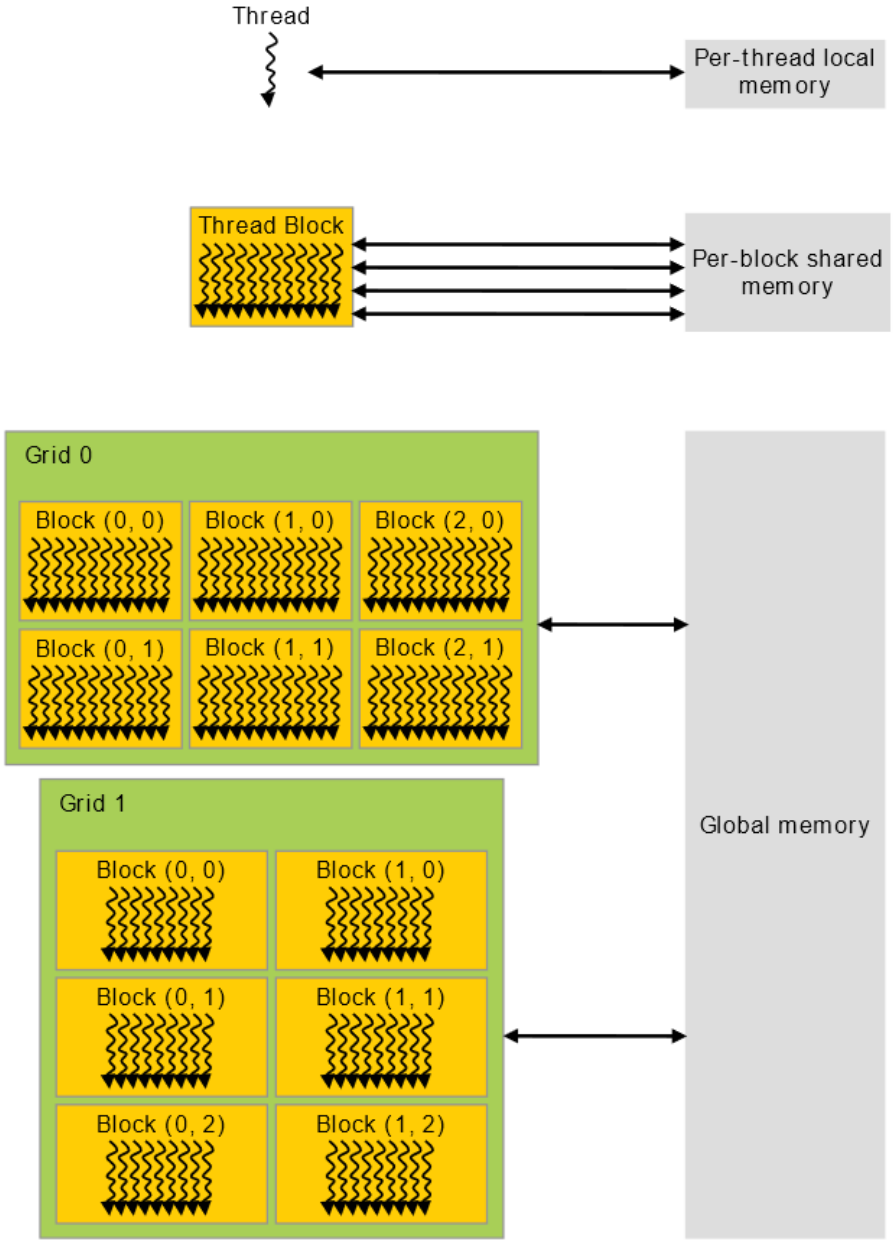

Figure 7: CUDA Memory Hierarchy

There are two types of per-thread local memory: registers and local memory. Registers are used for automatic, i.e. local, variables declared within kernels. Local memory is used when registers are not enough to hold all automatic variables. It is also used for automatic arrays.

Shared memory is the only per-block shared memory type in CUDA. It can be used by declaring a local variable or array with a __shared__ specifier within a kernel.

In the global scope, there are three different memory types: global, constant and texture. Constant memory is used for data that is not changed throughout the execution of a kernel and is, therefore, read only. Texture memory is similar to constant memory as it is read only aswell but is not limited to 64KB like constant memory is. Texture memory is optimized for 2D spatial locality. Both texture and constant memory are cached. Global memory is not cached and can be modified by CUDA threads during kernel execution.

Regarding performance, registers are the fastest out of all memory types. Despite also having thread scope, local memory is slow, because it is an abstraction of global memory. This means that local memory has the same performance as global memory. Shared memory is also quite fast, second only to registers. Constant memory is slower than shared memory but faster than texture memory. Global memory and local memory are the slowest types in the CUDA memory architecture but they are still needed in several situations.

2.3.2 CUDA and CFR

In this dissertation, CUDA is used in the implementation of the CFR algorithm that runs on the GPU. Chapter 3 explains what CFR is and Chapter 4 provides details about the implementations that were developed, one of them being the GPU implementation.

Chapter 3

Related Work

3.1 Counterfactual Regret Minimization (CFR)

Extensive form is often used to represent sequential non-cooperative games, particularly those with imperfect information. The usefulness of this model *"depends on the ability of solution techniques to scale well in the size of the model"* [5]. Poker is one of the most commonly used games when testing the performance of these solution techniques. One of the reasons that may be behind this success is the fact that even its small variants can be quite large. For instance, heads-up limit Texas hold'em has 3.16×10^{17} game states [6].

Zinkevich *et al.* introduce the notion of *counterfactual regret* [5], which is applied in a new technique for finding approximate solutions to large extensive games. An algorithm for minimizing counterfactual regret in poker was developed and used to solve poker abstractions with as many as 10^{12} games states, *"two orders of magnitude larger than previous methods"* [5].

Counterfactual regret minimization extends the techniques of *regret minimization* and *regret matching* to sequential games. Neller and Lanctot use Kuhn Poker to explain the CFR algorithm [7].

3.1.1 Regret matching and minimization

Regret is the difference between the utility of a certain action and the utility of the action that was chosen.

Let´s consider the example given by Neller and Lanctot in order to better explain what regret minimization is: a game of Rock-Paper-Scissors where each player bets one dollar. The

winner takes both dollars and, in case of a draw, players retain their dollars. The utilities of the strategies for both players are represented in Table 1.

Table 1: Matrix for a Rock-Paper-Scissors game where players bet one dollar each

	Rock	*Paper*	*Scissors*
Rock	0, 0	-1, 1	1, -1
Paper	1, -1	0, 0	-1, 1
Scissors	-1, 1	1, -1	0, 0

Let's say we have Player A and Player B who will play a round of this game. Player A plays Rock and Player B plays Paper, with B being therefore the winner. For this round, Player A regrets not having played paper with value of $0-(-1)=1$ and regrets not having played scissors with value of $1-(-1)=2$. In the future, player A prefers to choose the action associated with the highest value of regret, which is to play scissors. Note that this technique leaves player A predictable and thus exploitable which leads to the technique of regret matching. Through regret matching, an agent's actions are selected randomly according to a probability distribution that is proportional to positive regrets [5]. In the last example, Player A has regret value of 0 for having chosen rock, 1 for not having chosen paper and 2 for not having chosen scissors. Now, let's obtain the normalized positive regrets, i.e., positive regrets divided by their sum: 0, $1/3$ and $2/3$ for rock, paper and scissors, respectively. With *regret matching*, Player A chooses his next action using the values obtained previously as probabilities. After the next action is chosen and the round is complete, the next values of regret are known and it is possible to obtain the cumulative regrets which are the sum of the previous regret values. After normalizing these cumulative regrets, Players would use these values as probabilities when choosing the next action.

In any two-player zero-sum game, when both players use regret-matching to update their strategies, their average strategies converge to a Nash equilibrium as the number of iterations tends to infinite [5].

3.1.2 Counterfactual regret

Besides extending regret minimization and regret matching to sequential games, CFR introduces the concept of *counterfactual regret*, which is the regret weighted by the probability of the opponent reaching the information set.

The basic principle behind CFR is the fact that, by minimizing the counterfactual regret at each information set, the average strategy converges to a Nash Equilibrium.

3.1.3 The original CFR algorithm

This algorithm is iterative. In each iteration, CFR plays one game and updates the counterfactual regret values and mixed strategies for each information set according to the outcome of that game. Here is the basic steps that are performed for each information set, in one iteration:

1. Compute expected utility of each action
2. Calculate the counterfactual regret for not taking each action
3. Add up counterfactual regret over all games of past iterations
4. Compute new strategy with probabilities that are proportional to the accumulated positive counterfactual regret values

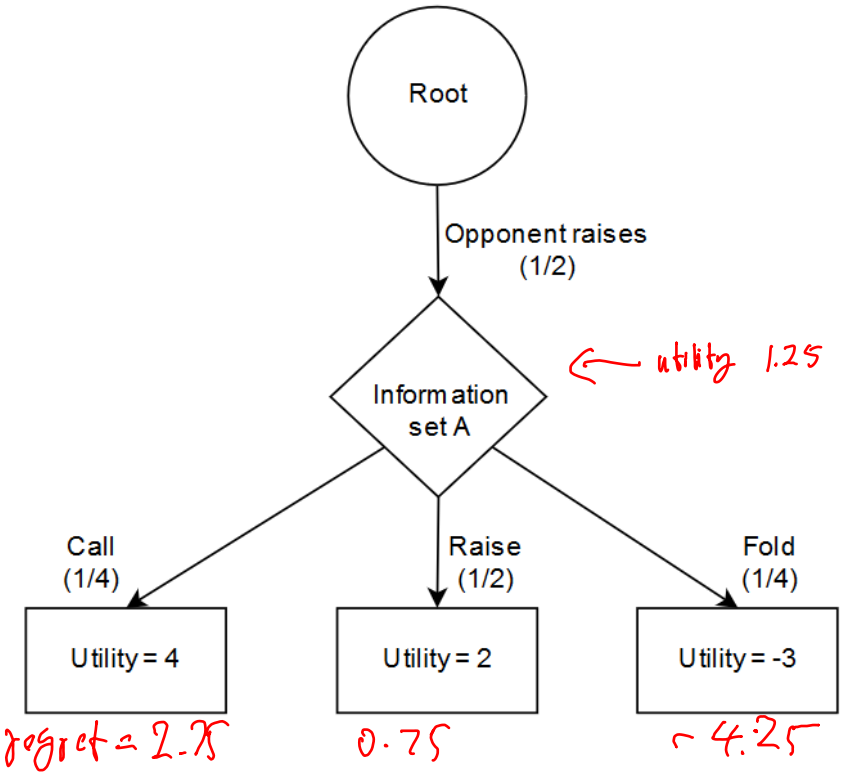

Figure 8: Part of the game tree of a Poker game (example)

To better explain how the algorithm works, consider an exemple of a Poker game. Part of the game tree is illustrated in Figure 8. At information set A, the current strategy is (1/4, 1/2, 1/4), i.e., the probabilities of player calling, raising and folding are 1/4, 1/2 and 1/4, respectively. The probability of the opponent reaching this information set is 1/2. When the game reaches information set A, CFR will first compute the utility of each action: (4, 2, -3). Therefore, the strategy's utility is $4 \times 1/4 + 2 \times 1/2 - 3 \times 1/4 = 1.25$. To obtain the regret values, subtract each action's utility by the strategy's utility: (2.75, 0.75, -4.25). To obtain

the counterfactual regret values, multiply the regret values by the probability of the opponent reaching the information set, i.e., 1/2: (1.375, 0.375, -2.125). Finally, to obtain the new mixed strategy, divide each counterfactual regret value by the sum of the positive counterfactual regret values. The negative counterfactual regret values lead to a probability of zero in the new strategy. So the probabilities of calling, raising and folding in the new strategy are $1.375/(1.375 + 0.375) = 0.786$, $0.375/(1.375 + 0.375) = 0.214$ and 0, respectively.

Note that the strategies at each information set should be initialized with a uniform distribution. In the previous example the initialized strategies would be $(1/3, 1/3, 1/3)$. This means that in the first iteration of the algorithm every information set has the same strategy.

3.2 CFR variants

3.2.1 Monte Carlo CFR

Monte Carlo CFR reduces the time spent traversing the game tree in each iteration by considering only a sampled portion of it [8]. There is a form of Monte Carlo style sampling called *chance-sampling*. Using this form of sampling, the algorithm obtains a single, sampled sequence of actions and only traverses the portion of the game tree that corresponds to that sequence [8].

Different sampling techniques lead to different variations of Monte Carlo CFR algorithms. The original CFR algorithm without sampling is usually called *"vanilla CFR"* [8]. There are three variants of chance sampling: *Opponent-Public Chance Sampling*, *Self-Public Chance Sampling* and *Public Chance Sampling*. Johanson *et al.* demonstrated empirically that an equilibrium approximation algorithm is more efficient on large games using Public Chance Sampling [9].

Since Monte Carlo CFR variants are not particularly relevant for this dissertation, they will not be explained here. Johanson *et al.* provide detailed explanations of them [9].

3.2.2 CFR-BR

Computing Nash Equilibrium strategies in large extensive-form games requires too much memory and time to be tractable [10]. The standard approach to overcome this issue is to use abstractions in order to reduce the size of the game. However, it has been recently found that this type of abstractions leads to the computation of Nash equilibria that can be really far from optimal strategies in the unabstracted games [10]. CFR-BR is an algorithm that finds optimal abstract strategies, i.e., strategies with minimal exploitability in the unabstracted game.

The algorithm leaves one player unabstracted and tries to find the optimal abstract strategy for the other player. Usually, this would not be possible because of the computational requirements, but this algorithm overcomes these issues in two ways:

- The opponent is assumed to employ a best-response strategy on each iteration instead of a no-regret strategy;
- CFR-BR uses sampling techniques when computing the best-response strategy, which means that the resulting strategy is not an exact best-response strategy.

CFR-BR was used to compute the least exploitable strategy ever reported for the game of heads-up limit Texas hold'em [10]. This game is explained in the next subsection.

3.2.3 CFR+ and Cepheus

Several games of perfect information have been solved in the past, like checkers. However, *"no nontrivial imperfect information game played competitively by humans has previously been solved"* [6].

Poker is a good example of a nontrivial imperfect information game that is played competitively by humans. Texas hold'em is the most popular variant of Poker, nowadays. When this variant is played with two players, with fixed bet sizes and fixed number of raises, it is called heads-up limit Texas hold'em.

In 2015, Bowling *et al.* announced that heads-up limit Texas hold'em is now essentially weakly solved. This was enabled by a new algorithm called CFR+. Like CFR, CFR+ is an iterative algorithm that computes an approximation to a Nash equilibrium. Also just like CFR, as the number of iterations grows, the computed solution gets closer and closer to a real Nash equilibrium [6].

Using CFR+, it is possible to solve extensive games with orders of magnitude larger than those that can be solved using other algorithms. While typical CFR implementations traverse only portions of the game tree, CFR+, by contrast, traverses the entire game tree. Also, CFR+ uses a variant of regret matching called regret matching+. With regret matching+, values of regret are constrained to be non-negative [6].

CFR+ requires less computation than state-of-the-art sampling CFR, while keeping its potential of massive parallelization[6].

Cepheus is *"the first computer program to play an essentially perfect game of poker"* [11]. Specifically, this program plays heads-up limit Texas hold'em. Cepheus learns to play poker by playing against itself, using CFR+.

3.3 Summary

CFR is an algorithm commonly used to find approximate solutions to large extensive sequential games. It uses the techniques of regret minimization and regret matching, while introducing a new concept of regret, i.e., counterfactual regret.

The original algorithm uses so much memory and time that it is necessary to use asbtractions in order to apply it to some very large games like some variants of Poker. However these abstractions lead to computed solutions that are very far from being optimal strategies in the unabstracted games. To overcome these issues, several CFR variants have been proposed, such as Monte Carlo CFR, CFR-BR and CFR+. Monte Carlo and CFR-BR were quite successful in improving the quality of solutions computed by the algorithm but these solutions are still not Nash equilibria. CFR+ was a success as it managed to solve heads-up limit Texas hold'em but it took 4800 CPUs running during 68 days to achieve this [12].

This dissertation shows that it is possible to achieve good results with a GPU implementation of CFR. However, the previously mentioned variants were tested with an enormous amount of computational resources and time while the GPU implementation of CFR was tested on a quite small scale. Chapter 4 provides details about the GPU implementation and Chapter 5 describes the details and results of every experiment that was done in the context of this dissertation.

Chapter 4

Proposed CFR Implementations

Three different implementations of CFR for Poker variants were developed in the context of this dissertation. The first one is recursive, singlethreaded and runs on the CPU. The second one is iterative, multithreaded and runs on the CPU. The third one is iterative, multithreaded and runs on the GPU. All three implementations use a C module developed by the Computer Poker Research Group, University of Alberta[1]. This module provides game logic related functions for almost any variant of Poker. The game's rules can be provided to the module by a text file. It was distributed as part of the server software used in the Annual Computer Poker Competition[2].

4.1 Game logic module

The informated related to the game's rules is stored in a struct called *Game* and everything related to the state of the game is stored in a struct called *State*. To create a *Game* structure, the file containing the game's parameters must be provided. Figure 9 shows an example of a text file that defines a specific variant of Poker. It is possible to define several parameters of a Poker game such as:

- ***numPlayers*** – number of players
- ***numRounds*** – number of rounds
- ***blind*** – blinds for each player
- ***raiseSize*** – fixed amount for a raise, only used in limit games
- ***firstPlayer*** – who is the first to perform an action
- ***maxRaises*** – maximum number of raises in a round

[1] http://poker.cs.ualberta.ca/
[2] http://www.computerpokercompetition.org/

- ***numSuits*** – number of suits in the deck
- ***numRanks*** – number of ranks in the deck
- ***numHoleCards*** – number of hole cards for each player
- ***numBoardCards*** – number of public shared cards on the board

By changing each parameter in these files, it is easy to run each CFR implementation with a lot of poker variants, each one with a different level of difficulty and number of information nodes.

```
GAMEDEF
limit
numPlayers = 3
numRounds = 1
blind = 1 1 1
raiseSize = 1
firstPlayer = 1
maxRaises = 1
numSuits = 1
numRanks = 4
numHoleCards = 1
numBoardCards = 0
END GAMEDEF
```

Figure 9: Text file that defines a game of 3 player Kuhn limit poker

4.2 Recursive CFR

4.2.1 Abstractions

Each possible hand is placed on a bucket, which is represented by an integer. The more hands there are per bucket, the more abstracted the game is. Because it is possible to tweak the game parameters in order to obtain Poker variants of different complexity, no abstraction will be used, though each hand is still represented by a bucket number.

4.2.2 Game tree representation

In any Poker variant, there are at most three possible actions at any game state: fold, call and raise. Therefore, each node can have up to three child nodes. It is important to note that a check is considered the same as a call in this implementation. A player will never fold if he can check, since there is a relationship of strategic dominance between the two.

The class that represents a tree node is called Sequence. Every sequence object except the root has a parent Sequence. The root Sequence object represents the state in which no player has taken any action yet. Figure 10 shows a very small example of a game tree that contains only four sequence objects, i.e., nodes. The Sequence class contains several fields that store information about the game state in that node like current player, round number and number of possible actions available to the current player. Furthermore, there is a field that stores the parent's sequence number and a field that maps each possible action to a child Sequence.

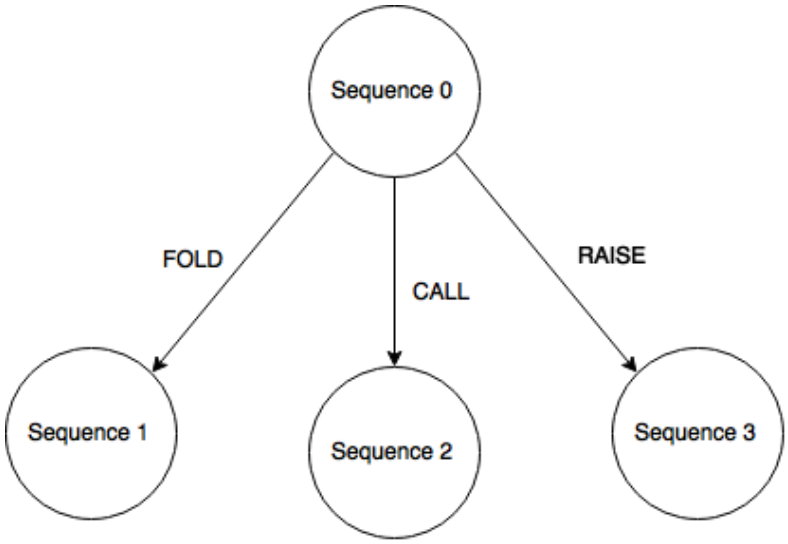

Figure 10: Small example of a game tree in this implementation

It is important to note that this representation does not depend on anything related to the deck that is used in the game. It only depends on the number of players, rounds and maximum number of raises. So if only the parameters related to the cards are changed in the game definition files, then the Sequence objects will be identical. It is also important to note that an information set, in this context, is represented by a pair of one Sequence and one bucket. The number of information sets in a certain game is equal to $N_{SEQUENCES} \times N_{BUCKETS}$.

In the CFR algorithm, the average strategies and counterfactual regret values must be stored and computed for each information set so the previous representation is not enough. At each Sequence, depending on the size of the deck, there are several possible hands that the current player can have and each different hand corresponds to a different information set. Thus, the average strategies and counterfactual regret values are represented by arrays. An example of such array can be seen in Figure 11. Note that there is one array for average strategies and another one for counterfactual regret values but both arrays have the same size and initial state.

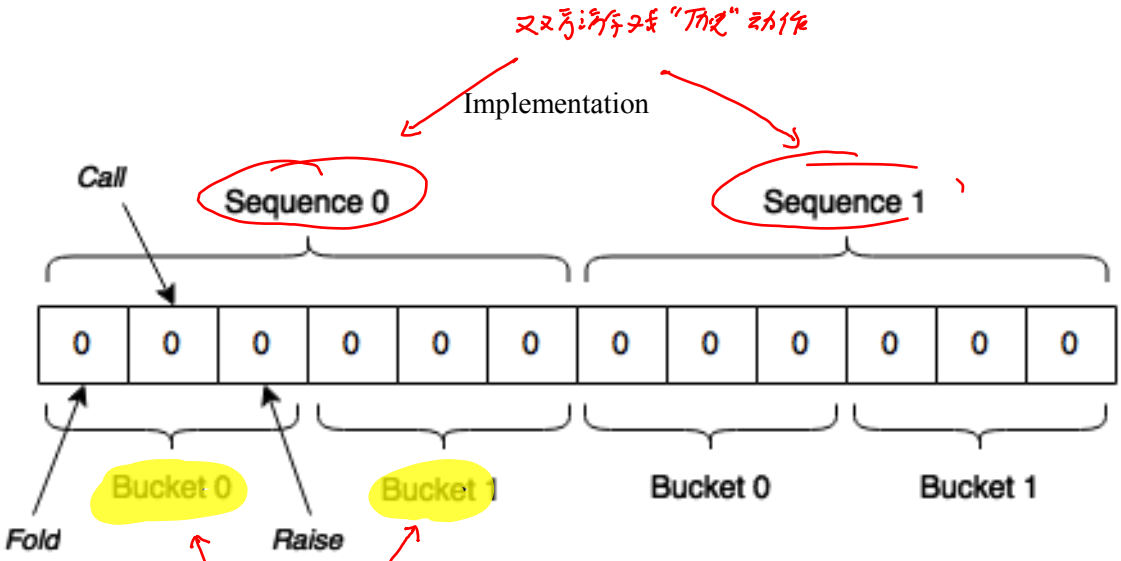

Figure 11: Example of the average strategies array in its initial state for a game with 2 sequences and 2 buckets

4.2.3 Brief explanation of core functions and variables

Figure 12 shows a summarized version of the main training function for this implementation. The arrays `average_strategy` (line 2) and `cfregret` (line 3) are ones that were described in the previous section. The average strategy is basically the output of the algorithm. The counterfactual regret values are stored in one iteration and used in the next so that is why there is a need for this array. The next array, `utilitySum`, is another output of the algorithm (line 4) and it is used to measure the quality of the average strategy. It contains the sum of all utility values per player that were computed in each iteration (lines 13 – 14). By dividing these sums by the number of iterations, the average utility is obtained.

```
1  void train(Game* game, long iterations) {
2      float average_strategy[NUM_SEQUENCES][NUM_BUCKETS][NUM_ACTIONS];
3      float cfregret[NUM_SEQUENCES][NUM_BUCKETS][NUM_ACTIONS];
4      float utilitySum[NUM_PLAYERS] {0.0};
5      Sequence* sequences = createSequences();
6      State state;
7
8      for (long i = 0; i < iterations; i++) {
9          initState(&state);
10         dealCards(game, &state);
11         float utility[NUM_PLAYERS];
12         cfr(sequences, state, average_strategy, cfregret, utility);
13         for (int player = 0; player < NUM_PLAYERS; player++)
14             utilitySum[player] += utility[player];
15     }
16 }
```

Figure 12: CFR training function

The function `createSequences()` creates an array of Sequence objects that holds every Sequence of the tree. The index of a certain Sequence is its sequence number.

The function `initState(...)` initializes a `State` struct to its initial default state while `dealCards(...)` modifies the `State` struct according to the game's rules so that it contains each player's hands and also the board cards. The function that deals the cards determines the buckets for each player.

Finally, `cfr(...)` implements the recursive CFR for an iteration. Figure 13 shows a summarized version of this function. Each call is associated with a specific Sequence which means that each execution of `cfr(...)` can lead to three more calls of the same function, one for each possible action in that Sequence. For each Sequence, the strategy is obtained through the technique of regret matching as it was described in Chapter 3. Using this freshly obtained strategy, the average strategy for the current Sequence and bucket is updated. The strategy will also be used to update the counterfactual regret values but this process also needs the utility values of each possible action. Now comes the recursive part: the utility values are returned by this function which means that, in order to obtain the utility values for each action, a leaf Sequence node must be reached and the utility of this leaf node must be propagated back to every parent node, including the root. In fact, this means that the last Sequence to have its counterfactual regret values updated will actually be the first one, i.e., root while being the first one to have updated average strategies. Figure 13 does not show the part of the algorithm that returns the value of the state when the current Sequence has no actions possible, i.e., when it is a leaf node.

```
1  float currentStrategy[NUM_ACTIONS];
2  getStrategy(cfregret[sequenceNumber][bucket], currentStrategy);
3  updateAvgStrategy(sequenceNumber, bucket, average_strategy,
4      probabilities, currentStrategy);
5
6  int utilities[NUM_ACTIONS];
7  for (int action = 0; action < NUM_ACTIONS; ++action) {
8      if (currentSequence.actionIsPossible(action)) {
9          float nextProbs[NUM_PLAYERS];
10         float nextUtility[NUM_PLAYERS];
11         computeNextProbabilities(currentStrategy[action], probabilities,
12             nextProbs);
13         cfr(sequences, nextState, average_strategy, cfregret,
14             nextProbs, nextUtility);
15         utilities[action] = nextUtility[currentPlayer];
16     }
17 }
18 updateCFRegret(sequenceNumber, bucket, cfregret, probabilities,
19     utilities);
```

Figure 13: Summarized version of `cfr()`

There is a parameter that has not been mentioned yet which is the `probabilities` array. As it was described in Chapter 3, CFR uses counterfactual regret – regret weighted by the probability of the opponent reaching the information set. The `probabilities` array contains the probabilities of each player reaching the current sequence so that the counterfactual regret can be computed.

Figure 14 shows the probabilities array and Strategy for an unspecified bucket in two different sequences. Player 0 is the first to act, choosing to call. According to the strategy at that moment, Player 0 had 1/4 probability of choosing that action, which leads to him having probability $1 \times 1/4 = 1/4$ of reaching the information set represented in Sequence 1.

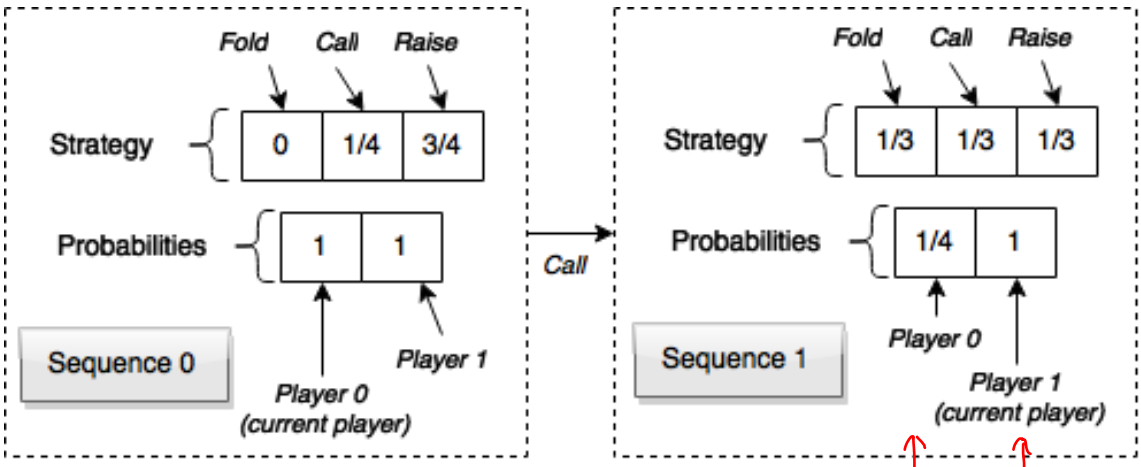

Figure 14: Example that shows the probabilities array and strategy for two sequences.

4.3 Iterative CFR

For the iterative version, it is necessary to split the algorithm into two different phases:

1. Update `average_strategy` and `probabilities` arrays for every sequence in ascending order – first one to be updated is Sequence 0.
2. Update `cfregret` amd `utilities` array for every sequence in descending order – first ones to be updated are the last sequences, i.e., leaf nodes.

This change could not be avoided because the values of counterfactual regret associated with a sequence are dependent on every child sequence's counterfactual regret values while the probabilities associated with a sequence are dependent on its parent sequence's probabilities. In the recursive version, the first phase is done before the recursive call and the second phase is done after the call.

Another change that must be done has to do with the `probabilities` array. This array has $N_{PLAYERS}$ elements in the recursive version because each function call passes a new array to

the next call. In this iterative version, there is only one function call so there can only be one array for every Sequence. The new array has $N_{PLAYERS} \times N_{SEQUENCES}$ elements. Figure 15 shows the probabilities array in the iterative version for the same game state as the one shown in Figure 14.

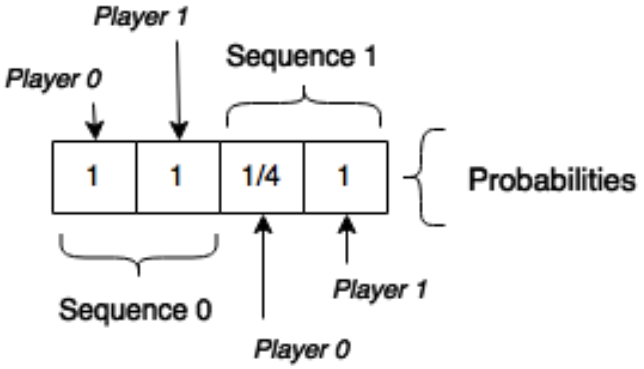

Figure 15: Example of `probabilities` array in the iterative version of CFR

Similar to the situation with the probabilities, the State struct is passed by parameter in the recursive version so a new array must be used in the iterative version to store the State associated with every Sequence. This new array has $N_{SEQUENCES}$ elements.

The utility values are returned by each call in the recursive version so one more array is declared in the iterative version where every player's utility value is stored for every sequence. This new array, `utilities`, has $N_{PLAYERS} \times N_{SEQUENCES}$ elements.

There is still one detail that is missing which is how to make this algorithm parallel. Consider the example shown in Figure 10: it is not possible to have the algorithm process Sequence 0 and Sequence 1 at the same time because they depend on each other. However, once Sequence 0 is processed, it is possible to have all three child Sequences processed at the same time since these do not depend on each other – each Sequence depends only on its parent and child Sequences. The way to make this algorithm parallel is to divide all Sequences in levels so that every Sequence in the same level can be processed concurrently. Therefore we need one more array to make this possible, i.e., an array of arrays where the index is the level and the array associated with the level contains every sequence number that belongs to that level. An example can be seen in Figure 16.

With this last change, the iterative version can be made parallel in both phases. In the first phase, the levels are processed in ascending order and in the second phase they are processed in descending order.

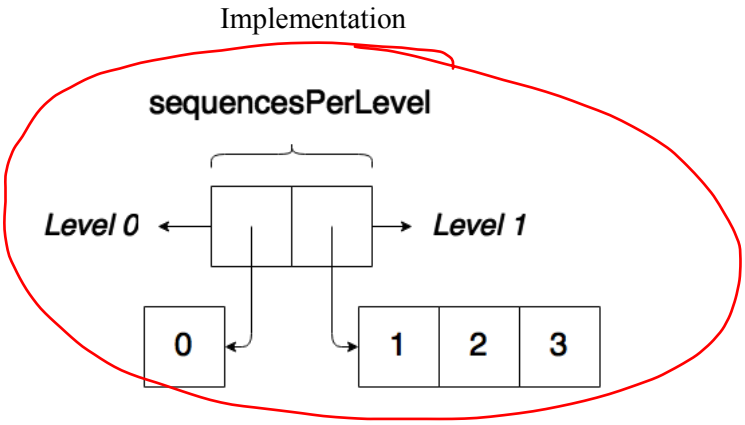

Figure 16: Example of the new array, `sequencesPerLevel`, for the same game that is represented in Figure 10

4.3.1 Multithreaded CPU Implementation

This implementation uses OpenMP, which is an API that supports multi-platform shared memory multiprocessing programming in C++. OpenMP can be used to make a *for* loop parallel by only using a simple compiler directive. The API will be responsible for spawning threads and dividing the iterations of the *for* loop between the available threads. It is also possible to choose how many threads are spawned. This is the only functionality of OpenMP that is needed for this implementation.

The CPU implementation of the iterative version of CFR has many similarities with the recursive implementation as it can be seen in Figure 17 and Figure 18. The inner *for* loop (line 3 in both Figures) is made parallel with the `#pragma omp parallel for` openMP directive (line 2 in both Figures). In the actual implementation, there are some optimizations that were done so that this directive does not spawn and destroy threads in each iteration of the outer *for* loop (line 1). Every important variable and array used in the CPU implementation of iterative CFR has been explained in the previous sections.

```
1  for (int level = 0; level < nrlevels; level++) {
2      #pragma omp parallel for
3      for (int index = 0; index < sequencesPerLevel[level].size(); index++) {
4          int sequenceNumber = sequencesPerLevel[level][index];
5          float currentStrategy[NUM_ACTIONS];
6          getStrategy(cfregret[sequenceNumber][bucket], currentStrategy);
7          updateAvgStrategy(sequenceNumber, bucket, average_strategy,
8              probabilities, currentStrategy);
9
10         for (int action = 0; action < NUM_ACTIONS; ++action) {
11             if (currentSequence.actionIsPossible(action))
12                 updateProbabilities(currentStrategy[action], probabilities);
13         }
14     }
15 }
```

Figure 17: Phase 1 of the multithreaded iterative version of CFR (summarized)

```
1  for (int level = nrlevels-1; level >= 0; level--) {
2      #pragma omp parallel for
3      for (int index = 0; index < sequencesPerLevel[level].size(); index++) {
4          int sequenceNumber = sequencesPerLevel[level][index];
5          for (int action = 0; action < NUM_ACTIONS; ++action) {
6              int childSequence = childSequences[action];
7              for (int player = 0; player < NUM_PLAYERS; player++) {
8                  utilities[sequenceNumber][player] +=
9                      utilities[childSequence][player] * currentStrategy[action];
10             }
11         }
12         updateCFRegret(sequenceNumber, bucket, cfregret, probabilities,
13             utilities);
14     }
15 }
```

Figure 18: Phase 2 of the multithreaded iterative version of CFR (summarized)

There is an important optimization that has not been mentioned yet which considerably improves locality of reference when accessing the large arrays used in this iterative version. This optimization consists of a change in order of Sequences in the game tree. This is illustrated by Figure 19. This optimization changes the structure of every array that is related to sequences since sequences of the same level are close to each other in memory. In practice, the arrays are not actually different and what actually changes are the memory access patterns during the execution of the algorithm.

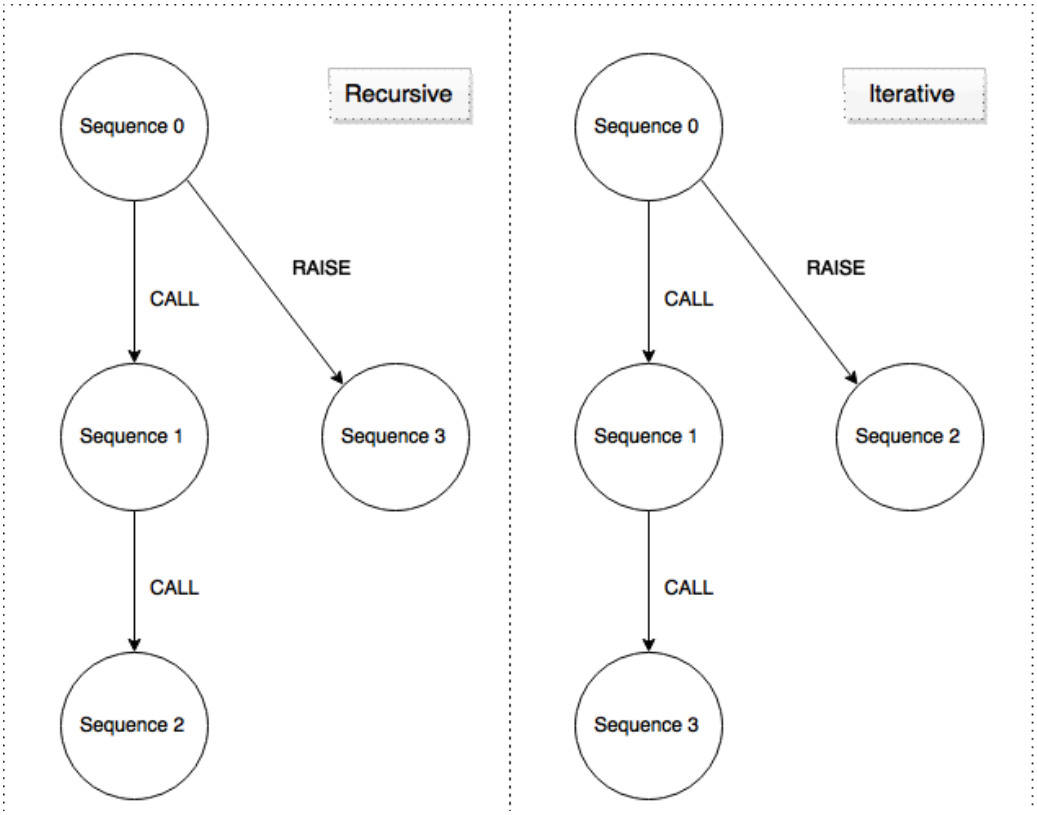

Figure 19: Difference of sequence numbers in each version of CFR

4.3.2 Multithreaded GPU Implementation

Two CUDA kernels were implemented, one for each phase of the iterative CFR. It is very similar to the CPU implementation, except for the arrays. Some considerations about CUDA and memory that were already mentioned in a previous chapter:

- The operation to copy memory between the device and the host is expensive
- Shared memory is very fast but limited
- Global memory available is huge when compared to shared memory and constant memory but is the slowest type
- Constant memory is very fast but read only and limited
- Texture memory is also quite fast and has the same limit as global memory but is also read only

Constant memory should be used whenever possible for read only arrays. If the size is not enough then texture memory should be used. For arrays that need to be modified during kernel execution, global memory must be used. If there are multiple accesses to the same positions in global memory during the same kernel execution, then parts of these arrays should be copied to shared memory in the beginning of the kernel and then copied back to global memory before the kernel returns. Any local array should be allocated on shared memory.

Since copying memory between the device and the host is slow, then the amount of memory copy operations should be reduced to the lowest possible. This means that in an ideal situation, every array would be contiguous and next to each other so that only one copy operation would have to be made.

Because an array of arrays is not contiguous in memory, it can be quite slow to copy it between the device and the host. An example is the `sequencesPerLevel` array. The inner arrays can be located in memory addresses that are far from the memory address of the outer array. The inner arrays have dynamic sizes so it is not possible to turn this array into a contiguous 2 dimensional one. The solution is to flatten this array into an array of only one dimension. To do this, the sizes of each inner array must be stored in a separate one so that it is possible to turn `sequencesPerLevel` into a one dimensional array. Each access does require a computation of the index but the reduction in memory copy operations between the host and device are worth that additional requirement.

There is one array that is read only but is too large to be stored in constant memory: `sequencesPerLevel`. Since constant memory can not be used in this case, this array will be

stored in texture memory. Figure 20 shows how the texture for this array is used in the kernel (device).

```
//inside kernel
int sequenceNumber = tex1Dfetch<int>(sequencesPerLevelTexture,
    blockDim.x * blockIdx.x + threadIdx.x);
```

Figure 20: Fetching data from a CUDA texture

Each CUDA thread has a global unique index that can be obtained with the following expression (Figure 20):

$$blockDim.x \times blockIdx.x + threadIdx.x$$

This index is what enables the CUDA thread to decide which Sequence it is going to process.

The array that contains the Sequence objects is stored in global memory. Sequences are actually never modified during the execution of the algorithm but they cannot be stored in texture memory because CUDA only accepts primitive types in textures. They can't be stored in constant memory either because of the size limit.

The array that stores the State structs for each sequence is also stored in global memory since this array is not read only. Using shared memory in this case does not boost the performance because the cost of the memory copy operations between shared memory and global memory is higher than the time that can be saved in the few memory accesses involving this array. The same reasoning applies to the `probabilities` array which is also stored in global memory.

However, the arrays `average_strategy`, `cfregret` and `utilities` are accessed so many times during each kernel execution that the performance is increased when the following is done:

- Copy the relevant parts of all three arrays from global memory to shared memory in the beginning of the kernel
- Do the computations with accesses to shared memory
- Copy back the results from shared memory to global memory

Figure 21 shows how shared memory is used in the first phase of the algorithm. Shared memory is visible to all threads within the same block so the declaration of shared memory arrays has to take into account how many threads a block has. In this implementation, $BLOCK\_SIZE$ is a constant that represents the number of threads per block.

```
__shared__ float avgStratShd[BLOCK_SIZE*sizeof(float)*NUM_ACTIONS];
__shared__ float currentStrategy[BLOCK_SIZE*NUM_ACTIONS];

memcpy(&avgStratShd[threadIdx.x*sizeof(float)*NUM_ACTIONS],
    &average_strategy[sequenceIndex], sizeof(float)*NUM_ACTIONS);

getStrategy(cfregret[seqIndex], &(currentStrategy[threadIdx.x*NUM_ACTIONS]));
updateAvgStrategy(sequenceNumber, bucket, avgStratShd,
        probabilities, currentStrategy);

memcpy(&average_strategy[sequenceIndex],
    &avgStratShd[threadIdx.x*sizeof(float)*NUM_ACTIONS],
    sizeof(float)*NUM_ACTIONS);
```

Figure 21: Part of the first kernel that shows how the average strategy is updated using shared memory

4.4 Summary

This chapter provides the details about the three implementations that were developed in the context of this dissertation:

- Singlethreaded Recursive CFR
- Multithreaded Iterative CFR (CPU)
- Multithreaded Iterative CFR (GPU)

Chapter 5 describes the experiments that were done with these implementations and their results. It also provides analysis on these results, explaining why a certain implementation performs better than the others in each experiment.

Chapter 5

Experiments and Results

To test the implementations that were described in Chapter 4, four different variants of poker games were made by tweaking the game's parameters in the game definition file. Table 2 shows several details about the game trees of each of these four variants such as number of information sets. Each variant has a very different number of buckets, sequences and levels in order to see how the implementations behave under different circumstances.

The tests compare memory usage and execution time of each implementation for each poker variant. Note that memory usage in the GPU implementation refers to device, i.e. GPU, memory. The execution times are the average values of five different measurements. All measurements, including the average and standard deviation, for each variant can be found in Execution time measurements.

First, the results will be presented and then some remarks will be made about them. The iterative CPU implementation is tested with eight CPU threads while the GPU implementation is tested with as many threads as there are sequences for each game tree level.

Table 2: Some details about the game trees of each variant

	Variant A	*Variant B*	*Variant C*	*Variant D*
Number of sequences	4	4	21285	616592
Number of buckets	4	8347680	2024	8
Number of information sets	16	33390720	43080840	4932736
Number of levels	3	3	21	30
Average number of sequences per level	1.333333	1.333333	1013.571	20553.07

5.1 Poker variant A

This variant is very simple. Games of this variant have two players and there is only one round. There are no board cards and each player has one hole card only. The deck consists of four cards (one suit, four ranks). Figure 22 shows the game definition file for this variant.

```
GAMEDEF
limit
numPlayers = 2
numRounds = 1
blind = 1 1
raiseSize = 1
firstPlayer = 1
maxRaises = 1
numSuits = 1
numRanks = 4
numHoleCards = 1
numBoardCards = 0
END GAMEDEF
```

Figure 22: Game definition file for variant A

5.1.1 Memory usage

Table 3 shows the amount of memory used by each implementation for poker variant A. There are increases of 18.69% and 4660.53% in memory used by the multithreaded CPU and multithreaded GPU implementations respectively, compared to the recursive one.

Table 3: Memory usage of all implementations for variant A

CFR Implementation	*Memory usage in KB*	*Difference relative to Recursive (%)*
Recursive	1348	0
Multithreaded Iterative (CPU)	1600	+18.69
Multithreaded Iterative (GPU)	64172	+4660.53

5.1.2 Execution time

Figure 23 shows the measured execution times (in seconds) of the implementations for variant A. For 10000 iterations, the GPU implementation is approximately 239.06 times slower than the recursive one. For the same number of iterations, the iterative CPU implementation is approximately 2.99 times slower.

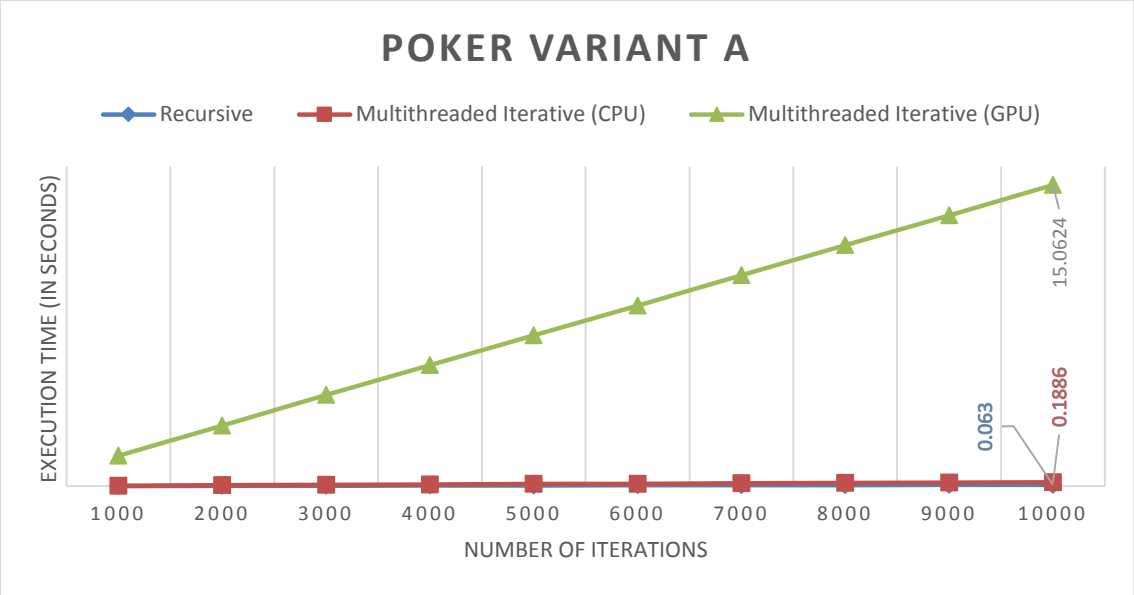

Figure 23: Execution times of the three implementations for variant A

5.2 Poker variant B

The main difference between this variant and the previous one is the size of the deck and player's hands. In this variant, the deck consists of thiry-six cards, there are five board cards and each player has two hole cards. The number of rounds is one and number of players is two, same as the previous variant. Figure 24 shows the game definition file for this variant.

```
GAMEDEF
limit
numPlayers = 2
numRounds = 1
blind = 1 1
raiseSize = 1
firstPlayer = 1
maxRaises = 1
numSuits = 4
numRanks = 9
numHoleCards = 2
numBoardCards = 5
END GAMEDEF
```

Figure 24: Game definition file for variant B

5.2.1 Memory usage

Table 4 shows the amount of memory used by each implementation for poker variant B. There is an increase of 0.01% in memory used by the multithreaded CPU implementation compared to the recursive one. However, there is a reduction of 53.94% in the case of the GPU implementation also compared to the recursive one.

Table 4: Memory usage of all implementations for variant B

CFR Implementation	*Memory usage in KB*	*Increase relative to Recursive (%)*
Recursive	1879876	0
Multithreaded Iterative (CPU)	1880064	+0.01
Multithreaded Iterative (GPU)	865783	-53.94

5.2.2 Execution time

Figure 25 shows the measured execution times (in seconds) of the implementations for variant B. For 10000 iterations, the GPU implementation is approximately 78.85 times slower than the recursive one. For the same number of iterations, the iterative CPU implementation is approximately 1.70 times slower than the recursive implementation.

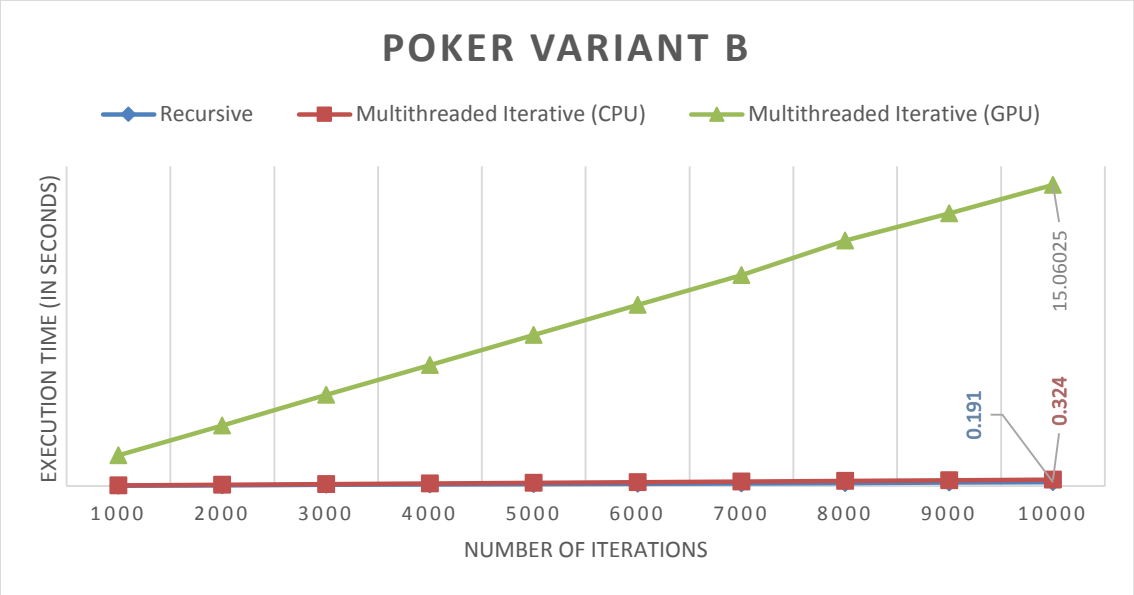

Figure 25: Execution times of the three implementations for variant B

5.3 Poker variant C

```
GAMEDEF
limit
numPlayers = 5
numRounds = 1
blind = 1 1 1 1 1
raiseSize = 1
firstPlayer = 1
maxRaises = 4
numSuits = 4
numRanks = 6
numHoleCards = 2
numBoardCards = 1
END GAMEDEF
```

Figure 26: Game definition file for variant C

Variant C is the one with the largest number of information sets, but is neither the one with the largest number of buckets nor the one with the largest number of sequences. It is, therefore, an intermediate variant with a medium number of sequences and a medium number of buckets. It consists of five players and each player can raise at most five times. There is only one round as well and the deck consists of twenty-four cards (four suits and six ranks). There is only one board card and each player has two hole cards. Figure 26 shows the game definition file for this variant.

5.3.1 Memory usage

Table 5 shows the amount of memory used by each implementation for poker variant C. There are increases of 5.42% and 13.71% in memory used by the multithreaded CPU and multithreaded GPU implementations respectively, compared to the recursive one.

Table 5: Memory usage of all implementations for variant C

CFR Implementation	*Memory usage in KB*	*Increase relative to Recursive (%)*
Recursive	1016252	0
Multithreaded Iterative (CPU)	1071296	+5.42
Multithreaded Iterative (GPU)	1155530	+13.71

5.3.2 Execution time

Figure 27 shows the measured execution times (in seconds) of the implementations for variant C. For 100 iterations, the GPU implementation is approximately 1.06 times faster than the recursive one. For the same number of iterations, the iterative CPU implementation is approximately 2.61 times faster than the recursive implementation.

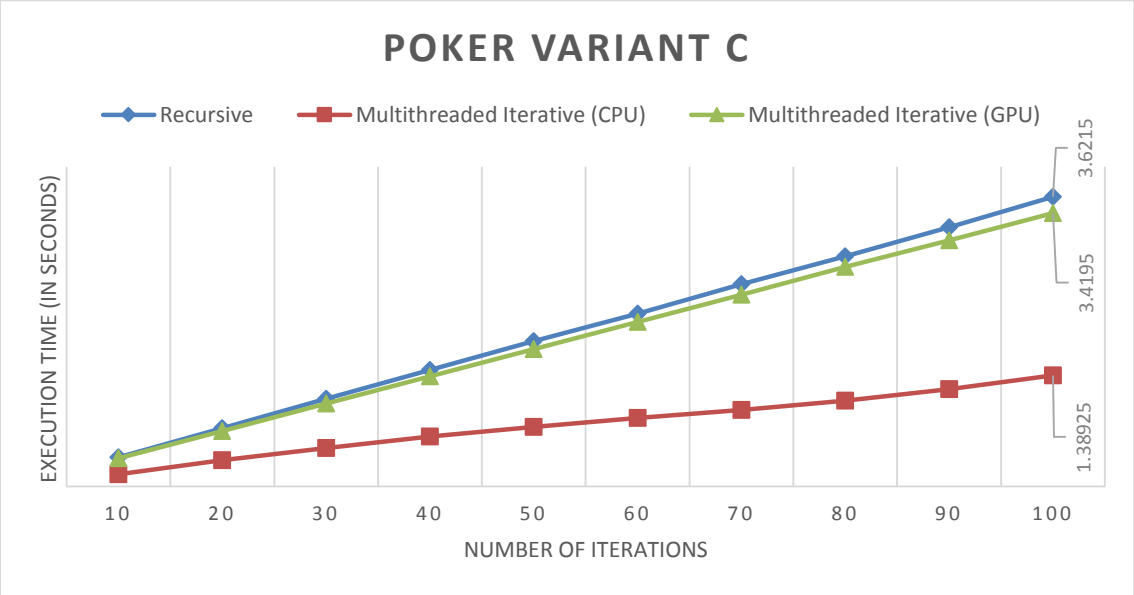

Figure 27: Execution times of the three implementations for variant C

5.4 Poker variant D

```
1  GAMEDEF
2  limit
3  numPlayers = 8
4  numRounds = 1
5  blind = 1 1 1 1 1 1 1 1
6  raiseSize = 1
7  firstPlayer = 1
8  maxRaises = 3
9  numSuits = 1
10 numRanks = 8
11 numHoleCards = 1
12 numBoardCards = 0
13 END GAMEDEF
```

Figure 28: Game definition file for variant D

While variant B has a large number of buckets but a small number of buckets, variant D is the opposite. It has eight players and each can raise at most three times. There are no board cards and each player gets one hole card. The deck has the minimum possible cards which is eight (one suit, eight ranks). Every other variant has only one round and this variant is no different. Figure 28 shows the game definition file for this variant.

5.4.1 Memory usage

Table 6 shows the amount of memory used by each implementation for poker variant D. There are increases of 718.15% and 758.12% in memory used by the multithreaded CPU and multithreaded GPU implementations respectively, compared to the recursive one.

Table 6: Memory usage of all implementations for variant D

CFR Implementation	*Memory usage in KB*	*Increase relative to Recursive (%)*
Recursive	206552	0
Multithreaded Iterative (CPU)	1689908	+718.15
Multithreaded Iterative (GPU)	1772470	+758.12

5.4.2 Execution time

Figure 29 shows the measured execution times (in seconds) of the implementations for variant D. For 100 iterations, the GPU implementation is approximately 6.13 times faster than the recursive one. For the same number of iterations, the iterative CPU implementation is approximately 3.6 times faster than the recursive implementation.

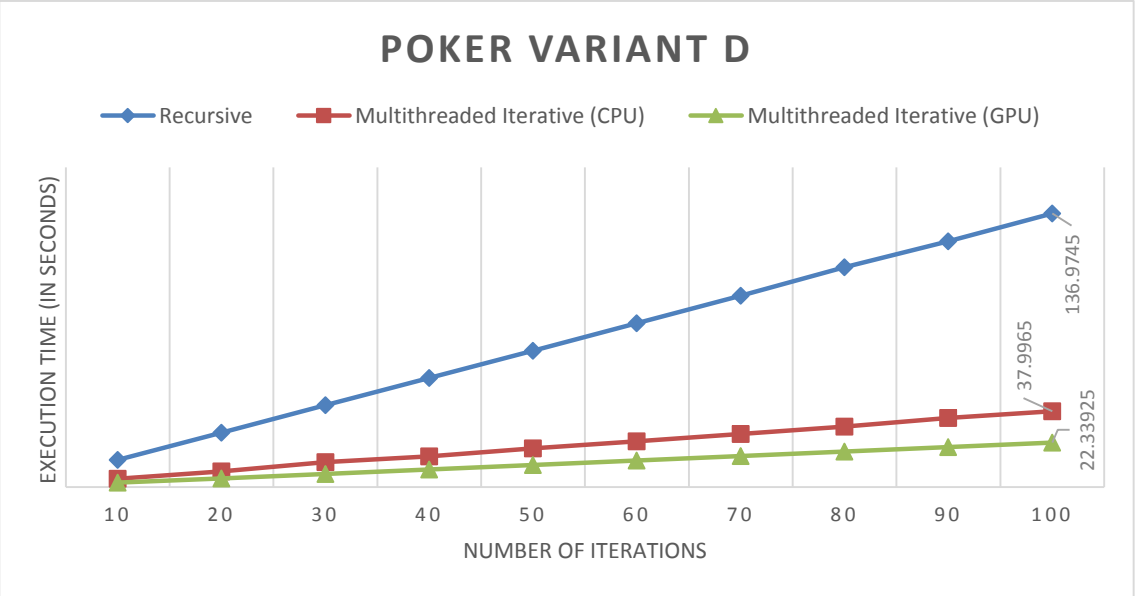

Figure 29: Execution times of the three implementations for variant D

5.5 Analysis of Results

When these implementations were developed, there was not a focus on reducing the memory usage so, although the tests that show how much memory is being used are useful, there is a lot of optimizations that can be done to reduce the impact on the system's memory that the multithreaded implementations have. The focus was definitely on reducing the execution time which is the main objective of this dissertation.

The results for variant A show that a GPU implementation for a very simple game with a small game tree offers very poor performance. This is expected because the cost of setting up the CUDA kernels plus the cost of operations that handle device memory are not worth the time that is actually spent executing the algorithm itself.

In the case of variant B, the results are almost identical to the ones of variant A, because the number of sequences is not changed at all and it has already been estabilished that the number of sequences is the factor that dictates how much speedup is potentially possible for parallel implementations. The number of information sets is quite large, but this is due to the fact that the deck has a large size as well. There are still some differences in the measured times between this variant and variant A in the case of both CPU implementations – in variant A, the GPU implementation is 2.99 times slower while in variant B it is only 1.70 times slower. This happens because the arrays are much larger so the locality of reference is worse, which leads to a larger number of cache misses on the CPU, forcing the system to load memory from RAM to CPU cache more times than what happened with variant A.

The results of variant C show a point in which the GPU implementation starts being faster than the recursive one. Note that the iterative CPU implementation is still faster than the GPU one for this variant.

Variant D is the one that offers the most successful results. In the case of this variant, the GPU implementation was faster than the other two by a reasonable margin. It does come as a cost of higher memory usage but as it was mentioned above, there are several optimizations that can be done to improve this and memory usage was not the focus of this dissertation.

These results show, indeed, that the more sequences the game tree has, the faster the GPU implementation will be when compared to the other two. It is important to keep in mind that what really matters is the number of sequences per level so a high number of sequences alone may not lead to a good performance of the GPU implementation.

They also show that the number of buckets does not seem to have an impact on the performance of the GPU implementation while they certainly affect the performance of both CPU implementations.

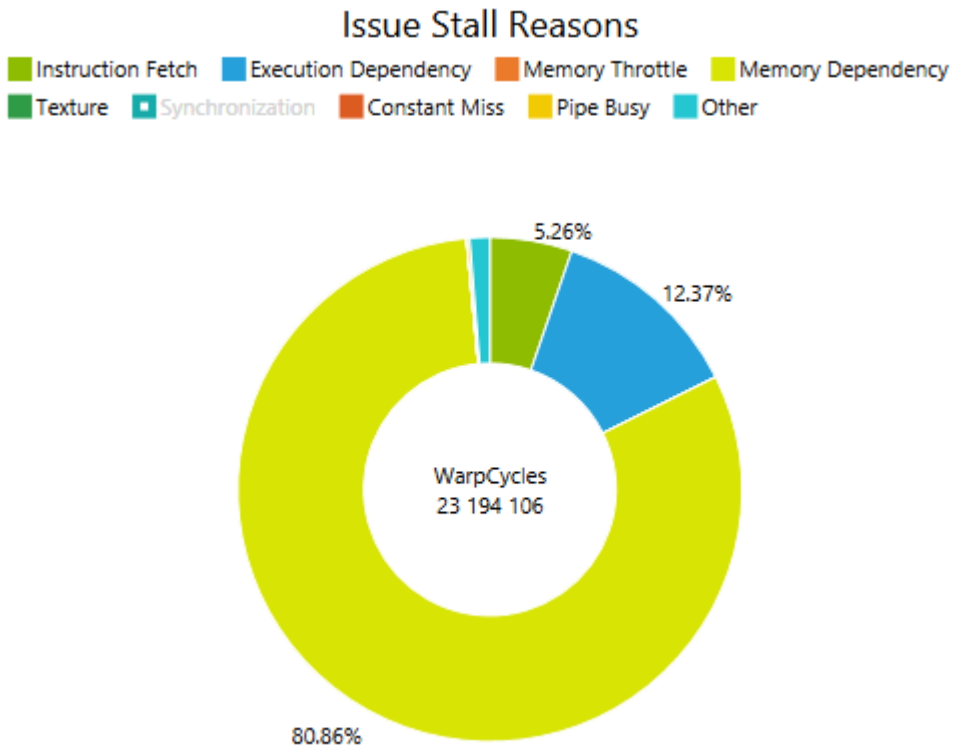

Figure 30: Reasons why the first phase kernel is not faster

Using CUDA Performance Analysis tool, it is possible to know what is actually stalling the GPU for each kernel call. Figure 30 and Figure 31 shows the graphs obtained with this tool for the first phase kernel and second phase kernel, respectively, in the experiments with variant C. Both kernels are heavily dependent on memory accesses but the second phase kernel is also dependent on instruction execution.

Execution dependency is related to how many different execution paths a thread can take. All three implementations contain a lot of *if* statements which leads to divergent control flow. GPUs do not handle divergent control flow very well, since having different execution paths leads to serialized execution of instructions.

The reason why the first phase kernel is more dependent on memory than the second phase kernel is because the second phase kernel does not access the average strategies array while the first phase kernel accesses every array.

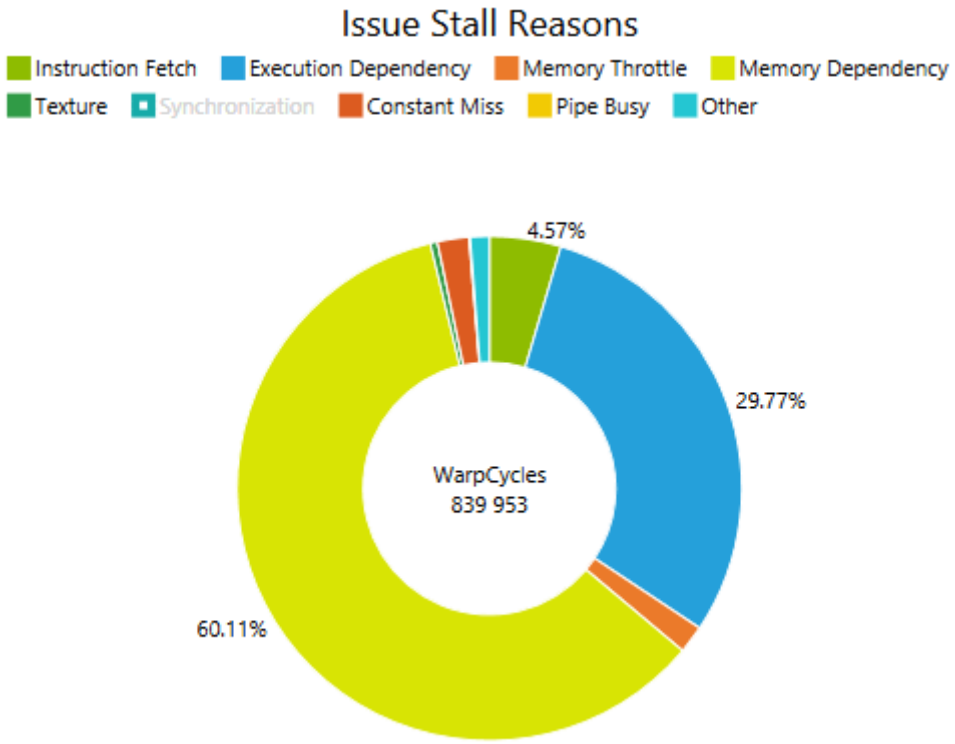

Figure 31: Reasons why the second phase kernel is not faster

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The main objective of this dissertation was to propose a GPU implementation of CFR as it was expected to provide better performance than common CPU implementations.

Three different implementations were developed. The first one is singlethreaded, recursive and runs on the CPU. The second one is multithreaded, iterative and runs on the CPU. Finally, the third one is multithreaded, iterative and runs on the GPU.

These implementations were tested with four different customized Poker variants.

Results show that the GPU implementation can be much faster than the first two, in some cases. In the case of a particular variant, one with a large number of players, the GPU implementation is 6.13 times faster than the singlethreaded recursive implementation while being 3.6 times faster than the multithreaded iterative one. It is still possible for the GPU implementation to be faster than the other two for games with only two players, for instance, but the number of maximum raises per player has to be large, otherwise it will be slower than both CPU implementations. The results also prove the theory that the most relevant factors in a poker game to determine whether a GPU implementation has better performance or not are the number of players and number of maximum raises per player.

The memory usage of all three implementations was also tested and, in general, the recursive version always uses less memory than the multithreaded implementations. However, the focus of this dissertation is not on the memory usage so there are several optimizations that can be made to reduce memory usages of both multithreaded implementations.

6.2 Future Work

As it was mentioned before, memory usage can definitely be improved in both parallel implementations. The game state associated with each sequence is stored on memory until the iteration is over but this is not needed at all. When processing a sequence in a certain level, the only game states that are needed are the ones from the previous level, which means that there is no need for the game states of all other previous levels to remain in memory until the iteration is over. This single optimization can potentially reduce memory usage to numbers close to the recursive implementation. However, even with this optimization both parallel implementations will still use more memory due to the fact that, in the case of the recursive version, memory is freed when calls return while in the case of both parallel versions, memory can only be freed at the end of a tree level.

Also, according to the graphs obtained with the CUDA Performance Analysis tool, there are some potential optimizations to be made on the GPU implementation that might improve execution times further. Future GPU implementations should favor computation of same values instead of storing them on memory for later reuse if possible. They should also try to focus on a specific game variant because one downside of supporting multiple variants of Poker - like this implementation does – is the large number of control flow statements. In the end, supporting multiple game variants leads a reduction in CUDA threads' performance.

References

[1] J. Ratliff, “Jim Ratliff’s Graduate-Level Course in Game Theory,” 1997. [Online]. Available: http://virtualperfection.com/gametheory/index.html. [Accessed: 22-Jan-2015].

[2] M. J. Osborne and A. Rubinstein, *A course in game theory*. MIT Press, 2012.

[3] T. L. Turocy and B. von Stengel, “Game Theory,” *Encyclopedia of Information Systems*, vol. 2, pp. 403–420, 2002.

[4] M. Harris, “An Easy Introduction to CUDA C and C++,” 2012. [Online]. Available: http://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/. [Accessed: 05-Feb-2015].

[5] M. Zinkevich, M. Johanson, M. Bowling, and C. Piccione, “Regret minimization in games with incomplete information,” *Advances in Neural Information Processing Systems*, vol. 20, pp. 1–14, 2008.

[6] M. Bowling, N. Burch, M. Johanson, and O. Tammelin, “Heads-up limit hold’em poker is solved,” *Science*, vol. 347, pp. 145–149, 2015.

[7] T. W. Neller and M. Lanctot, “An Introduction to Counterfactual Regret Minimization,” 2013. [Online]. Available: http://modelai.gettysburg.edu/2013/cfr/.

[8] R. Gibson, N. Burch, M. Lanctot, and D. Szafron, “Efficient monte carlo counterfactual regret minimization in games with many player actions,” *Advances in Neural Information Processing Systems*, pp. 1–9, 2012.

[9] M. Johanson, N. Bard, M. Lanctot, R. Gibson, and M. Bowling, “Efficient Nash equilibrium approximation through Monte Carlo counterfactual regret minimization,” in *AAMAS ’12 Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, 2012, pp. 837–846.

[10] M. Johanson, N. Bard, N. Burch, and M. Bowling, “Finding Optimal Abstract Strategies in Extensive-Form Games.,” *Proceedings of the Twenty-Sixth Conference on Artificial Intelligence (AAAI-12)*, 2012.

[11] Computer Poker Research Group - University of Alberta, “Cepheus Poker Project,” 2015. [Online]. Available: http://poker.srv.ualberta.ca/about. [Accessed: 31-Jan-2015].

References

[12] O. Tammelin, N. Burch, M. Johanson, and M. Bowling, “Solving Heads-up Limit Texas Hold ’ em,” *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI), 2015*, 2015.

[13] “Policonomics.” [Online]. Available: http://www.policonomics.com/lp-game-theory1-extensive-form/. [Accessed: 28-Jan-1BC].

[14] M. Ebersole, “Why GPU Computing,” 2013. [Online]. Available: http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/general/Why_GPU_Computing.pptx.

[15] M. Harris, “Six Ways to SAXPY,” 2012. [Online]. Available: http://devblogs.nvidia.com/parallelforall/six-ways-saxpy/. [Accessed: 05-Feb-2015].

References

Appendix A

Execution time measurements

A.1 Poker variant A

A.1.1 Singlethreaded Recursive CFR

Table 7: Measurements in seconds of singlethreaded recursive CFR with variant A

Iterations	*Measurements*					*Average*	*Standard Deviation*
1000	0.006	0.006	0.006	0.006	0.006	0.006	0
2000	0.013	0.013	0.013	0.013	0.014	0.0132	0.000447
3000	0.02	0.023	0.029	0.019	0.023	0.0228	0.003899
4000	0.026	0.026	0.027	0.025	0.025	0.0258	0.000837
5000	0.031	0.034	0.037	0.033	0.034	0.0338	0.002168
6000	0.038	0.038	0.047	0.049	0.043	0.043	0.00505
7000	0.054	0.057	0.048	0.044	0.044	0.0494	0.005899
8000	0.05	0.051	0.051	0.059	0.051	0.0524	0.003715
9000	0.057	0.061	0.058	0.061	0.058	0.059	0.001871
10000	0.064	0.063	0.063	0.063	0.062	0.063	0.000707

A.1.2 Multithreaded Iterative CFR (CPU)

Table 8: Measurements in seconds of multithreaded iterative CFR (CPU) with variant A

Iterations	*Measurements*					*Average*	*Standard Deviation*
1000	0.023	0.021	0.019	0.023	0.02	0.0212	0.001789
2000	0.039	0.046	0.043	0.042	0.044	0.0428	0.002588
3000	0.051	0.068	0.077	0.065	0.064	0.065	0.009354

4000	0.084	0.09	0.076	0.081	0.086	0.0834	0.005273
5000	0.103	0.101	0.116	0.105	0.099	0.1048	0.006648
6000	0.121	0.103	0.118	0.115	0.118	0.115	0.007036
7000	0.147	0.129	0.131	0.14	0.141	0.1376	0.00747
8000	0.15	0.161	0.156	0.166	0.162	0.159	0.006164
9000	0.131	0.186	0.186	0.196	0.186	0.177	0.026077
10000	0.205	0.182	0.184	0.186	0.186	0.1886	0.009317

A.1.3 Multithreaded Iterative CFR (GPU)

Table 9: Measurements in seconds of multithreaded iterative CFR (GPU) with variant A

Iterations	*Measurements*					*Average*	*Standard Deviation*
1000	1.522	1.558	1.503	1.536	1.504	1.5246	0.023147
2000	3.012	3.032	3.023	3.008	3.067	3.0284	0.023544
3000	4.631	4.527	4.55	4.584	4.557	4.5698	0.039796
4000	6.053	6.017	6.06	6.067	6.052	6.0498	0.019305
5000	7.558	7.512	7.552	7.522	7.528	7.5344	0.019769
6000	9.019	9.035	9.011	9.045	9.047	9.0314	0.0159
7000	10.517	10.559	10.557	10.572	10.53	10.547	0.022683
8000	12.073	12.051	12.039	12.059	12.055	12.0554	0.012361
9000	13.557	13.557	13.529	13.566	13.543	13.5504	0.014519
10000	15.083	15.104	15.044	15.015	15.066	15.0624	0.034472

A.2 Poker variant B

A.2.1 Singlethreaded Recursive CFR

Table 10: Measurements in seconds of singlethreaded recursive CFR with variant B

Iterations	*Measurements*					*Average*	*Standard Deviation*
1000	0.019	0.02	0.019	0.02	0.019	0.0195	0.000577
2000	0.038	0.039	0.038	0.038	0.038	0.03825	0.0005
3000	0.059	0.059	0.057	0.057	0.058	0.05775	0.000957
4000	0.076	0.076	0.076	0.076	0.076	0.076	0
5000	0.096	0.096	0.104	0.097	0.096	0.09825	0.003862
6000	0.115	0.115	0.115	0.115	0.115	0.115	0
7000	0.134	0.134	0.134	0.134	0.133	0.13375	0.0005
8000	0.154	0.152	0.152	0.153	0.152	0.15225	0.0005
9000	0.173	0.178	0.173	0.173	0.173	0.17425	0.0025
10000	0.198	0.191	0.191	0.191	0.191	0.191	0

A.2.2 Multithreaded Iterative CFR (CPU)

Table 11: Measurements in seconds of multithreaded iterative CFR (CPU) with variant B

Iterations	*Measurements*					*Average*	*Standard Deviation*
1000	0.035	0.034	0.033	0.034	0.032	0.03325	0.000957
2000	0.063	0.063	0.064	0.064	0.063	0.0635	0.000577
3000	0.095	0.098	0.094	0.096	0.096	0.096	0.001633
4000	0.127	0.127	0.128	0.126	0.129	0.1275	0.001291
5000	0.16	0.159	0.164	0.156	0.159	0.1595	0.003317
6000	0.19	0.191	0.192	0.192	0.193	0.192	0.000816
7000	0.225	0.222	0.223	0.221	0.225	0.22275	0.001708
8000	0.275	0.254	0.252	0.252	0.254	0.253	0.001155
9000	0.283	0.29	0.283	0.286	0.285	0.286	0.002944
10000	0.315	0.317	0.326	0.33	0.323	0.324	0.005477

A.2.3 Multithreaded Iterative CFR (GPU)

Table 12: Measurements in seconds of multithreaded iterative CFR (GPU) with variant B

Iterations	*Measurements*					*Average*	*Standard Deviation*
1000	1.529	1.526	1.544	1.53	1.533	1.53325	0.007719
2000	3.031	3.03	3.032	3.035	3.024	3.03025	0.004646
3000	4.631	4.608	4.545	4.533	4.594	4.57	0.036579
4000	6.218	6.046	6.087	6.048	6.033	6.0535	0.023302
5000	7.549	7.544	7.594	7.547	7.542	7.55675	0.024918
6000	9.052	9.056	9.059	9.06	9.056	9.05775	0.002062
7000	10.541	10.561	10.551	10.554	10.571	10.55925	0.008884
8000	12.069	12.871	12.06	12.037	12.145	12.27825	0.397887
9000	13.547	13.668	13.543	13.662	13.676	13.63725	0.063095
10000	15.053	15.051	15.076	15.038	15.076	15.06025	0.018945

A.3 Poker variant C

A.3.1 Singlethreaded Recursive CFR

Table 13: Measurements in seconds of singlethreaded recursive CFR with variant C

Iterations	*Measurements*					*Average*	*Standard Deviation*
10	0.354	0.365	0.368	0.365	0.357	0.36375	0.004717
20	0.747	0.724	0.732	0.738	0.716	0.7275	0.009574
30	1.084	1.093	1.089	1.076	1.132	1.0975	0.024118
40	1.456	1.465	1.452	1.447	1.471	1.45875	0.011147

50	1.79	1.807	1.827	1.814	1.818	1.8165	0.008347
60	2.129	2.144	2.157	2.178	2.175	2.1635	0.015969
70	2.504	2.583	2.512	2.533	2.51	2.5284	0.032424
80	2.891	2.893	2.874	2.889	2.857	2.87825	0.016358
90	3.259	3.251	3.24	3.24	3.245	3.244	0.005228
100	3.605	3.604	3.614	3.607	3.661	3.6215	0.026665

A.3.2 Multithreaded Iterative CFR (CPU)

Table 14: Measurements in seconds of multithreaded iterative CFR (CPU) with variant C

Iterations	*Measurements*					*Average*	*Standard Deviation*
10	0.151	0.169	0.123	0.171	0.153	0.154	0.022181
20	0.23	0.333	0.323	0.331	0.335	0.3305	0.00526
30	0.488	0.477	0.481	0.493	0.478	0.48225	0.007365
40	0.635	0.622	0.624	0.625	0.635	0.6265	0.005802
50	0.793	0.749	0.743	0.737	0.747	0.744	0.005292
60	0.94	0.857	0.857	0.843	0.879	0.859	0.014877
70	1.069	0.955	0.954	0.966	0.961	0.959	0.005598
80	1.18	1.087	1.079	1.073	1.061	1.075	0.010954
90	1.284	1.23	1.21	1.219	1.208	1.21675	0.010046
100	1.39	1.395	1.41	1.381	1.371	1.38925	0.016978

A.3.3 Multithreaded Iterative CFR (GPU)

Table 15: Measurements in seconds of multithreaded iterative CFR (GPU) with variant C

Iterations	*Measurements*					*Average*	*Standard Deviation*
10	0.356	0.36	0.351	0.356	0.351	0.3545	0.004359
20	0.704	0.696	0.692	0.694	0.692	0.6935	0.001915
30	1.043	1.041	1.034	1.04	1.039	1.0385	0.003109
40	1.385	1.389	1.377	1.375	1.372	1.37825	0.007455
50	1.717	1.711	1.717	1.718	1.723	1.71725	0.004924
60	2.065	2.067	2.065	2.054	2.052	2.0595	0.007594
70	2.396	2.394	2.4	2.398	2.399	2.39775	0.00263
80	2.735	2.747	2.741	2.745	2.746	2.74475	0.00263
90	3.073	3.079	3.08	3.086	3.073	3.0795	0.005323
100	3.449	3.423	3.412	3.422	3.421	3.4195	0.005066

A.4 Poker variant D

A.4.1 Singlethreaded Recursive CFR

Table 16: Measurements in seconds of singlethreaded recursive CFR with variant D

Iterations	*Measurements*					*Average*	*Standard Deviation*
10	13.73	13.725	13.718	13.665	13.729	13.70925	0.029848
20	27.335	27.316	27.293	27.332	27.298	27.30975	0.017821
30	41.16	41.177	41.14	41.123	41.105	41.13625	0.030696
40	54.82	54.617	54.659	54.537	54.771	54.646	0.097495
50	68.208	68.208	68.322	68.316	68.196	68.2605	0.067772
60	81.89	81.813	81.891	81.943	82.806	82.11325	0.464913
70	95.607	95.51	96.111	95.902	95.811	95.8335	0.249571
80	109.839	110.048	110.356	109.937	109.726	110.0168	0.262655
90	122.674	122.633	123.428	123.136	123.044	123.0603	0.328524
100	136.283	136.656	137.127	137.355	136.76	136.9745	0.324288

A.4.2 Multithreaded Iterative CFR (CPU)

Table 17: Measurements in seconds of multithreaded iterative CFR (CPU) with variant D

Iterations	*Measurements*					*Average*	*Standard Deviation*
10	4.475	4.232	4.172	4.196	4.196	4.199	0.024739
20	9.006	8.003	7.879	7.918	7.872	7.918	0.060172
30	12.632	12.091	12.687	12.261	13.481	12.63	0.62025
40	18.968	15.327	15.348	15.43	15.398	15.37575	0.04685
50	20.057	20.591	18.978	19.344	19.01	19.48075	0.758446
60	23.752	23.48	22.795	22.971	22.734	22.995	0.338586
70	27.368	26.344	26.479	26.547	26.928	26.5745	0.250312
80	31.202	30.234	30.234	30.691	30.275	30.3585	0.222508
90	35.325	34.041	34.921	35.801	33.853	34.654	0.895222
100	38.738	37.613	37.593	38.145	38.635	37.9965	0.496528

A.4.3 Multithreaded Iterative CFR (GPU)

Table 18: Measurements in seconds of multithreaded iterative CFR (GPU) with variant D

Iterations	*Measurements*					*Average*	*Standard Deviation*
10	2.248	2.239	2.242	2.24	2.243	2.241	0.001826
20	4.474	4.471	4.47	4.469	4.475	4.47125	0.00263
30	6.704	6.697	6.698	6.7	6.698	6.69825	0.001258
40	8.928	8.929	8.93	8.926	8.931	8.929	0.00216

50	11.16	11.16	11.157	11.205	11.159	11.17025	0.0232
60	13.392	13.388	13.387	13.388	13.389	13.388	0.000816
70	15.614	15.615	15.614	15.614	15.619	15.6155	0.00238
80	17.851	17.847	17.849	17.843	17.842	17.84525	0.003304
90	20.08	20.077	20.077	20.181	20.099	20.1085	0.049433
100	22.336	22.354	22.333	22.337	22.333	22.33925	0.010012

Appendix A