Computing Nash Equilibria in Bimatrix Games: GPU-based Parallel Support Enumeration

Safraz Rampersaud, *Student Member, IEEE,* Lena Mashayekhy, *Student Member, IEEE,* and Daniel Grosu, *Senior Member, IEEE*

Abstract—Computing Nash equilibria is a very important problem in strategic analysis of markets, conflicts, and resource allocation. Unfortunately, computing these equilibria even for moderately sized games is computationally expensive. To obtain lower execution times it is essential to exploit the parallel processing capabilities offered by the currently available massively parallel architectures. To address this issue, we design a GPU-based parallel support enumeration algorithm for computing Nash equilibria in bimatrix games. The algorithm is based on a new parallelization method which achieves high degrees of parallelism suitable for massively parallel GPU architectures. We perform extensive experiments to characterize the performance of the proposed algorithm. The algorithm achieves significant speedups relative to the OpenMP and MPI-based parallel implementations of the support enumeration method running on a cluster of multi-core computers.

Index Terms—Nash Equilibria, Game Theory, Parallel Algorithms, GPU, OpenMP, MPI.

1 Introduction

Game theory studies the interaction between strategic decision-makers [1]. Over the past fifty years, the developments in game theory and computing have provided researchers with methods and tools that allow them to better understand the behavior of agents in strategic situations. Arguably, the most famous equilibrium concept for noncooperative games is the *Nash equilibrium* [2], [3], the solution of a game from which no player can improve her payoff by deviating. This equilibrium can be used as a prediction of the outcome of noncooperative games. The Nash equilibrium concept has been used in diverse fields such as economics, biology, politics, and computer science to understand the behavior of agents in competing situations.

Understanding real world strategic interactions usually requires the modeling of a large number of agents having a large number of choices or actions. For instance, computers worldwide requesting resources over a given time interval on the Internet can be modeled as a large game, where the computation required to solve for equilibrium is intractable. The best approach to compute Nash equilibria for such large games relies on the power of parallelism.

The ability to leverage parallel systems for solving large, complex problems is certainly of interest to any researcher who is investigating large scale games. With access to hundreds of computing cores on a single device, Graphics Processing Units (GPUs) are suitable platforms for massively parallel execution, low cost processing, and fast implementation. In this paper, we design a GPU-based parallel algorithm for computing all Nash equilibria in bimatrix games (i.e., nonzero-sum two-player noncooperative games).

The existing methods and algorithms for solving bimatrix games can be classified into two categories: (i) algorithms for *computing a sample Nash equilibrium*, and (ii) algorithms for *computing all Nash equilibria*.

One of the first algorithms for computing a sample Nash equilibrium in bimatrix games was proposed by Lemke and Howson [4]. The algorithm is a complementary pivoting algorithm that solves the linear complementarity problem corresponding to the bimatrix game. Savani and von Stengel [5] showed that it is possible to create bimatrix games in such a way that in the best case it takes exponential time to find a Nash equilibrium using the Lemke-Howson algorithm [5]. The complexity of computing Nash equilibria has been investigated by Daskalakis [6] who showed that the problem is PPAD-complete, where PPAD stands for Polynomial Parity Arguments on Directed Graphs. It is not known whether a Nash equilibrium can be found in polynomial time in the worst case [7].

The simplest algorithm for finding all Nash equilibria is the *support enumeration algorithm*. This algorithm is searching all the possible pairs of supports of mixed strategies and checks if they satisfy the Nash equilibrium conditions. It takes exponential time since the total number of pairs that need to be explored is exponential in the number of actions. This method is described in [1], [8]. A Mathematica implementation of the support enumeration algorithm is described in [9]. Gambit [10] which is a software tool for solving games also implements this algorithm. Computing all Nash equilibria can also be based on enumerating the vertices of the best response polytopes of the two players. This method was proposed by Mangasarian [11] and implemented in Gambit [10]. Avis [12] designed the lexicographical

• *The authors are with the Department of Computer Science, Wayne State University, 5057 Woodward Avenue, Detroit, MI, 48202.*
E-mail: safraz@wayne.edu, mlena@wayne.edu, dgrosu@wayne.edu.

reverse search library, lrslib, for vertex enumeration of polyhedra, which has recently been used in a sequential algorithm for finding Nash equilibria in bimatrix games. ZRAM [13] was developed by Marzetta as a portable library for parallel search using an old implementation of lrs library. However, it does not appear to have been maintained in the last several years to reflect changes and optimizations in lrslib.

The standard software package for generating games to support the performance analysis of game solving algorithms is GAMUT [14]. Gambit [10] is a collection of software tools created to analyze games. Gambit implements almost all the algorithms described above. However, the implementation of these algorithms is sequential.

Results such as those of Datta [15] promote analytic developments. For instance, it has been shown that any set of Nash equilibria is equivalent to a semi-algebraic set for representing the probability of agents selecting actions. Through these results, existing computer algebra tools may be used to solve for equilibria. In particular, Datta's investigation of using computer algebra [15] in games surveys two methods: *Gröbner* bases method, which uses geometric information to solve a system of polynomial equations; and, polynomial homotopy continuation, which transforms a system representative set of equations into an approximate system set of equations, where the solutions to the approximations are easier to compute. Govindan and Wilson [16] proposed a robust method for finding equilibria in finite games by using topological properties and combining path-following algorithms. von Stengel [17] provided a comprehensive survey of methods for computing Nash equilibria in finite games.

Parallel algorithms for computing Nash equilibria have been investigated by Widger and Grosu [18], [19], [20]. The closest work to ours is by Widger and Grosu [18] who proposed a parallel support enumeration algorithm for finding all equilibria in bimatrix games that was specifically designed for message-passing architectures. Widger and Grosu [19] also proposed a message-passing parallel algorithm for computing all Nash equilibria in bimatrix games based on vertex enumeration. In the vertex enumeration method, all vertices of both players polytopes are enumerated and checked to determine if the vertices are completely labeled and the corresponding mixed Nash equilibrium is produced. Lastly, Widger and Grosu [20] proposed a parallel algorithm for computing Nash equilibria in n-player games based on polynomial continuation methods. There exist very few research papers investigating the use of GPU-based systems for solving game theory problems. Peters et al. [21] leveraged GPU-based platforms to model and investigate behavioral strategies in evolutionary games. Leskinen et al. [22] used GPU processing to find Nash equilibria of a specific multi-objective optimization problem. Bleiweiss [23] exploited the massively parallel GPU architecture to solve zero-sum combinatorial games. None of these works provided a general GPU-based parallel algorithm for finding all Nash equilibria in bimatrix games.

1.1 Our Contributions

We design a GPU-based parallel support enumeration algorithm for computing all Nash equilibria in bimatrix games. The design of the algorithm is based on a new parallelization method which exploits the nature of the problem in order to achieve high degrees of parallelism suitable for massively parallel GPU architectures. The design differs from the existing parallel support enumeration algorithms [18] since it exploits the maximum possible degree of parallelism available. To the best of our knowledge, this is the first parallel algorithm for computing Nash equilibria specifically designed for GPU platforms presented in the literature.

1.2 Organization

The rest of the paper is organized as follows. In Section 2, we introduce the necessary game theoretic concepts and present the support enumeration method for computing Nash equilibria. In Section 3, we describe the GPU platform and the new parallelization method used in the design of our GPU-based parallel support enumeration algorithm. In Section 4, we present the proposed GPU-based parallel support enumeration algorithm. In Section 5, we analyze the proposed algorithm. In Section 6, we investigate the performance of the proposed algorithm by performing extensive experiments. In Section 7, we draw conclusions and present directions for future work.

2 Bimatrix Games and Equilibria Computation

In this section, we present the support enumeration method for solving Nash equilibria in bimatrix games [1], [24]. A bimatrix game [3] is a finite, two-person, non-zero-sum, non-cooperative game.

Definition 1 (Bimatrix game): A *bimatrix game* $\Gamma(A,B)$ consists of:

- A set of two players: {Player 1, Player 2}.
- A finite set of actions for each player:
  $M = (s_1, s_2, \ldots, s_m)$, Player 1's set of actions;
  $N = (t_1, t_2, \ldots, t_n)$, Player 2's set of actions.
- Payoff matrices A, $B \in \mathbb{R}^{m\times n}$ corresponding to Player 1 and Player 2, respectively.

A *mixed strategy* for a player is a probability distribution on the set of player's actions. The mixed strategy of Player 1 is a *m*-vector, $x = (x_1, x_2, \ldots, x_m)$, where x_i is the probability of Player 1 choosing action s_i. The mixed strategy of Player 2 is a *n*-vector, $y = (y_1, y_2, \ldots, y_n)$, where y_j is the probability of Player 2 choosing action t_j. A *pure strategy* is a strategy where a player chooses a single action with probability 1 to use against the other player. We denote by M_x the *support* of mixed strategy x, which is the set of actions having positive probability in x, that is, $M_x = \{s_i | x_i > 0\}$. Similarly, we denote by N_y the support of mixed strategy y, which is the set of actions having positive probability in y, that is, $N_y = \{t_j | y_j > 0\}$.

A *best response* of Player 1 to the mixed strategy y of Player 2 is a mixed strategy x that maximizes Player 1's expected payoff, $x^T Ay$. Similarly, the best response of Player 2 to the mixed strategy x of Player 1 is the mixed strategy y that maximizes Player 2's expected payoff, $x^T By$.

The objective of both players is to choose a strategy resulting in the highest payoff. A common solution for noncooperative games is the Nash Equilibrium, which is guaranteed to exist for any finite game [3]. Nash equilibrium for a bimatrix game is defined as the pair of strategies (x, y), where x and y are the best responses to each other. The following theorem characterizes the Nash equilibria for bimatrix games [17].

Theorem 1 (Nash Equilibrium): The mixed strategy pair (x,y) is a Nash equilibrium of $\Gamma(A,B)$ if and only if the following two conditions are satisfied,

$$\forall s_i \in M_x, (Ay)_i = u = \max_{q \in M} \{(Ay)_q\} \tag{1}$$

$$\forall t_j \in N_y, (x^T B)_j = v = \max_{r \in N} \{(x^T B)_r\} \tag{2}$$

The first condition ensures that a mixed strategy x of Player 1 is a best response to mixed strategy y of Player 2, that is, if all pure strategies s_i in the support of x are best responses to mixed strategy y. The second condition is the best response condition for Player 2.

In this paper, we are considering only non-degenerate games. These are games in which no mixed strategy having the support of size k has more than k pure best responses. A useful property of non-degenerate games is that their Nash equilibria are given by strategies having supports of equal size [17].

The *support enumeration method* consists of enumerating all possible pairs of supports (M_x, N_y) of mixed strategies, where $M_x \subset M$ and $N_y \subset N$, and checking the Nash equilibrium conditions given in Theorem 1 for each pair of supports. For the pair of supports (M_x, N_y) of mixed strategies (x, y), the method involves solving the following equations:

$$\sum_{i \in M_x} x_i B_{ij} = v, \ \forall j \in N_y \tag{3}$$

$$\sum_{i \in M_x} x_i = 1 \tag{4}$$

and

$$\sum_{j \in N_y} y_j A_{ij} = u, \ \forall i \in M_x \tag{5}$$

$$\sum_{j \in N_y} y_j = 1 \tag{6}$$

Algorithm 1 SEQ-SE(A, B)

```
1: Input: Player 1 payoff, Player 2 payoff ($A$, $B$)
2: Output: Set of equilibria ($\mathcal{E}$)
3: $\mathcal{E} = \emptyset$
4: $q = \min(m, n)$
5: for $k = 1, \ldots, q$ do
6:     for each $(M_x, N_y), M_x \subseteq M, N_y \subseteq N, |M_x| = |N_y| = k$ do
7:         Solve:
8:             $\sum_{i \in M_x} x_i B_{ij} = v, \ \forall j \in N_y$
9:             $\sum_{i \in M_x} x_i = 1$
10:            $\sum_{j \in N_y} y_j A_{ij} = u, \ \forall i \in M_x$
11:            $\sum_{j \in N_y} y_j = 1$
12:        if $x_i, y_j \geq 0, \forall\, i, j$ and $x, y$ satisfy Theorem 1 then
13:            $\mathcal{E} = \mathcal{E} \cup (x, y)$
14: output $\mathcal{E}$
```

Formally, the set of equations (3) and (4) determines the strategy x from support set M_x of Player 1 that makes Player 2 indifferent among playing the strategies in N_y. Similarly, equations (5) and (6) determine the strategy y from support set N_y of Player 2 that makes Player 1 indifferent among playing the strategies in M_x. Any solution (x, y) meeting all these conditions is a candidate for the Nash equilibrium. Once the candidate solution for Nash equilibrium is determined the method checks if all the components of x and y are non-negative and if all the pure strategies in the supports yield the same maximum payoff. According to Theorem 1, if these conditions are satisfied the candidate solution (x, y) is a Nash equilibrium.

The sequential support enumeration algorithm, SEQ-SE, given in Algorithm 1, implements the support enumeration method described above. The algorithm generates all possible pairs of supports having the same size q, where $q = 1, \ldots, \min(m, n)$ (Lines 4-6). For each generated pair of supports of equal size the algorithm determines the candidate mixed strategy (x, y) by solving the system of linear equations given by equations (3) to (6) (Lines 8-11). If the system of equations does not have a solution, then no Nash equilibrium is possible for that pair of supports. If the system has (x, y) as a unique solution then, the algorithm checks that x_i, $i = 1, \ldots, m$, and y_j, $j = 1, \ldots, n$, are non-negative and that all pure strategies in the supports yield equal and maximum payoff (Line 12). If a Nash equilibrium is found, it is included in the set of Nash equilibria, denoted by $\mathcal{E}$ (Line 13).

The complexity of SEQ-SE has been shown to be $O((n+m)^3 \binom{m+n}{n})$, where $m > n$ [18]. This results from the complexity of solving the system of linear equations in $O((n+m)^3)$ for each of the $O(\binom{m+n}{n})$ possible strategy pairs of the two players. For square games (where $m = n$), the time complexity of SEQ-SE becomes $O(n^3 4^n)$.

Example. To show how the support enumeration algorithm works, we consider the following bimatrix game as an example:

$$\begin{array}{c|ccc} & \text{L} & \text{M} & \text{R} \\ \hline \text{T} & 5{,}3 & 0,0 & 0{,}1 \\ \text{B} & 0{,}0 & 3,5 & 1{,}3 \end{array} \quad (7)$$

Player 1 is the row player with two available actions, T and B, while Player 2 is the column player with three actions, L, M and R. The entries in the table represent the payoffs of Player 1 and Player 2, respectively. As a result, Player 1 and Player 2 payoff matrices are as follows:

$$A = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 3 & 1 \end{bmatrix}, B = \begin{bmatrix} 3 & 0 & 1 \\ 0 & 5 & 3 \end{bmatrix}. \quad (8)$$

Since $m = 2$ and $n = 3$, the support enumeration algorithm explores the mixed strategies of support size $k = 1$ and $k = 2$. It first explores the supports of size 1 which give the pure strategy Nash equilibria of the game. This game has two pure strategy Nash equilibria given by ((1, 0), (1, 0, 0)) and ((0, 1), (0, 1, 0)).

Next, the support enumeration algorithm explores the mixed strategies with support size 2. There are three possible pairs of mixed strategies with support size 2 that are explored, $((x_1, x_2), (y_1, y_2, 0))$, $((x_1, x_2), (y_1, 0, y_3))$, and $((x_1, x_2), (0, y_2, y_3))$

For the first pair of mixed strategies the algorithm solves the following equations: $3x_1 = 5x_2$; $x_1 + x_2 = 1$ and $5y_1 = 3y_2$; $y_1 + y_2 = 1$. The solution to these equations is: $x_1 = 5/8$, $x_2 = 3/8$ and $y_1 = 3/8$, $y_2 = 5/8$. The vector of expected payoffs to Player 2 is $x^T B = (15/8, 15/8, 14/8)$. The best response conditions from Theorem 1 are satisfied, and thus, the mixed strategy pair ((5/8, 3/8), (3/8, 5/8, 0)) is a Nash equilibrium of the bimatrix game.

For the second pair of mixed strategies, the algorithm solves the following equations: $3x_1 = x_1 + 3x_2$; $x_1 + x_2 = 1$ and $5y_1 = y_3$; $y_1 + y_3 = 1$. The solution to these equations is: $x_1 = 3/5$, $x_2 = 2/5$ and $y_1 = 1/6$, $y_3 = 5/6$. The vector of expected payoffs to Player 2 is $x^T B = (9/5, 10/5, 9/5)$. The best response conditions from Theorem 1 are not satisfied because Player 2's payoffs corresponding to y_1 and y_3 are not maximal. Thus, there is no mixed Nash equilibrium corresponding to the considered pair of strategies.

For the third pair of mixed strategies the algorithm solves the following equations: $5x_2 = x_1 + 3x_2$; $x_1 + x_2 = 1$ and $0 = 3y_2 + y_3$; $y_2 + y_3 = 1$. The solution to these equations is: $x_1 = 2/3, x_2 = 1/3$ and $y_2 = -1/2, y_3 = 3/2$. The vector y is not a vector of probabilities, and thus, there is no mixed Nash equilibrium corresponding to the considered pair of strategies.

The game considered in this example has two pure Nash equilibria given by ((1, 0), (1, 0, 0)) and ((0, 1), (0, 1, 0)), and one mixed strategy Nash equilibrium given by ((5/8, 3/8), (3/8, 5/8, 0)).

3 GPU-Based Parallel Support Enumeration

In this section, we introduce the GPU platform and present the parallelization method used in the design of our proposed GPU-based parallel support enumeration algorithm.

3.1 GPU Platform

The GPU device uses streaming multiprocessors suited for parallel tasks. This streaming architecture follows the Single Instruction Multiple Data (SIMD) model, making it ideal for problems requiring large data sets and/or large number of computations.

The Compute Unified Device Architecture (CUDA$^{\text{TM}}$) is a parallel computing platform that uses the graphics processing unit to increase computing performance [25]. The data parallel computations are performed by calling a method from the CPU that hosts the GPU device known as a *kernel* function. Processing threads are created and grouped together in *blocks*. A block is executed by the GPU scheduler in subsets of parallel threads (known as a *warp*). Each block and thread have a unique index. CUDA$^{\text{TM}}$maintains built-in variables `threadIdx.x` and `blockIdx.x` to identify these indices. The GPU memory consists of three major types, global and shared memory. Information placed in the GPU global memory can be accessed by the GPU and the CPU. Global memory hosts the kernel method and is accessible to the threads. Due to the overhead induced by data transfers, minimizing the accesses to global memory should be considered when designing GPU programs. A special type of global memory is the constant memory, which is used for storing global constants. Shared memory is local to each streaming multiprocessor and can be accessed by all threads within the same block. Shared memory accesses are multiple times faster than global accesses. Thousands of threads can be scheduled efficiently taking advantage of the available parallelism through the device load balancing mechanism. We organize the functions we use in the proposed algorithms presented in Section 4 as either being callable from the host machine and/or the GPU device. These functions have the following type qualifiers: *host* (callable from the host machine), *global* (callable from the host to the device, i.e., kernel function), and *device* (callable from the GPU device).

3.2 Parallelization Method

In order to illustrate our parallelization method we consider a particular bimatrix game in which both players have four actions, that is, a 4-action bimatrix game. Figure 1 shows the 4-action game with actions A, B, C and D for each player. While there are four actions in the game, the support size limits the number of actions available to each player. For support of size 1, both players are limited to choosing only one action

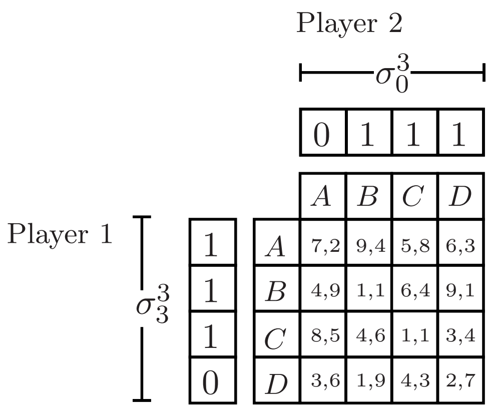

Fig. 1: A 4-action bimatrix game.

with a positive probability (probability 1 in this case). For support of size 2, both players may choose from two of the four actions with positive probabilities. For support of size 3, both players may choose from three of the four actions with positive probabilities, and for support of size 4, both players may choose from all four actions in the game with positive probabilities. As an example, in the case of support of size 3, Player 1 may choose actions A, B, and C with positive probabilities, and Player 2 may choose actions B, C, and D also with positive probabilities. Here, the action combinations (A, B, C) and (B, C, D) make up the supports of the mixed strategies of the two players.

To organize all possible combinations of four actions, we identify the support elements using *support keys*. Support keys are boolean arrays that indicate those actions that are available to the player. The support keys are shown next to the array of actions in Figure 1. Support keys, σ_j^i, are organized by support size i and index j, which refers to the order in which they are created (i.e., the lexicographical order). In Figure 1, Player 1's support is identified by the support key σ_3^3, while Player 2's support is identified by the support key σ_0^3. An entry of 0 in σ_j^i specifies that the corresponding action is not part of the support, while an entry of 1 specifies that the action is part of the support.

There is a finite number of ways to produce supports identified by support keys when considering all arrangements of the boolean value entries for a given support size. Figure 2 shows all support key arrangements for a 4-action bimatrix game grouped by support size. For support of size 1, there are four support keys, $(\sigma_0^1, \ldots, \sigma_3^1)$; for support of size 2, there are six support keys, $(\sigma_0^2, \ldots, \sigma_5^2)$; for support of size 3, there are four support keys, $(\sigma_0^3, \ldots, \sigma_3^3)$, and for support of size 4, there is only one support key, (σ_0^4).

We order the support keys according to the support size and store them in an array Θ^k, where k is the size of the support. For the game given in Figure 1, we have four support arrays, Θ^1 to Θ^4. The proposed algorithm

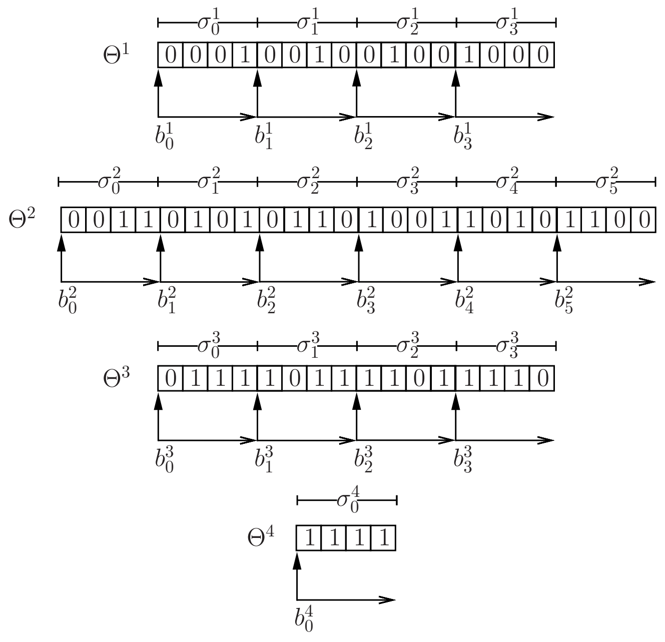
σ_0^1 σ_1^1 σ_2^1 σ_3^1
Θ^1
0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 0
b_0^1 b_1^1 b_2^1 b_3^1
σ_0^2 σ_1^2 σ_2^2 σ_3^2 σ_4^2 σ_5^2
Θ^2
0 0 1 1 0 1 0 1 0 1 1 0 1 0 0 1 1 0 1 0 1 1 0 0
b_0^2 b_1^2 b_2^2 b_3^2 b_4^2 b_5^2
σ_0^3 σ_1^3 σ_2^3 σ_3^3
Θ^3
0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0
b_0^3 b_1^3 b_2^3 b_3^3
σ_0^4
Θ^4
1 1 1 1
b_0^4

Fig. 2: Support keys for a 4-action game.

will access these arrays of support keys Θ^k in parallel to compute Nash equilibria. To do so, the proposed algorithm identifies every possible pair of both player's support elements and processes these pairs to determine the expected payoff solutions in parallel. The resulting solutions are the player strategies; two vectors x and y where their components are the probabilities of selecting an action identified by the support keys.

Using the 4-action bimatrix game as an example, the number of pairs that need to be processed is 69; that is, 16 pairs corresponding to supports of size 1; 36 corresponding to supports of size 2; 16 corresponding to supports of size 3; and one pair corresponding to support of size 4. A serial implementation of the algorithm would have to compute the 69 pairs iteratively.

The existing parallel implementation of the support enumeration method [18] was designed for message-passing systems and involves a coarse-grained decomposition. That is, each processor is assigned a set of support pairs in a round-robin fashion starting with the smallest size supports and ending with the largest size supports. Each processor checks all the pairs of supports in the assigned set sequentially.

Our proposed GPU-based parallel support enumeration algorithm exploits the maximum degree of parallelism available by using a fine-grained decomposition as follows. Each block of threads is assigned a support element (identified by its support key) from the set of Player 1's supports. Blocks, b_j^i, are organized by support size i and index j. When a block is assigned an element from Player 1's support, operations in the pre-kernel execution phase generate the number of threads needed and each thread is assigned a support pair as shown in Figure 3. Threads, $t_{j,k}^i$ are organized by

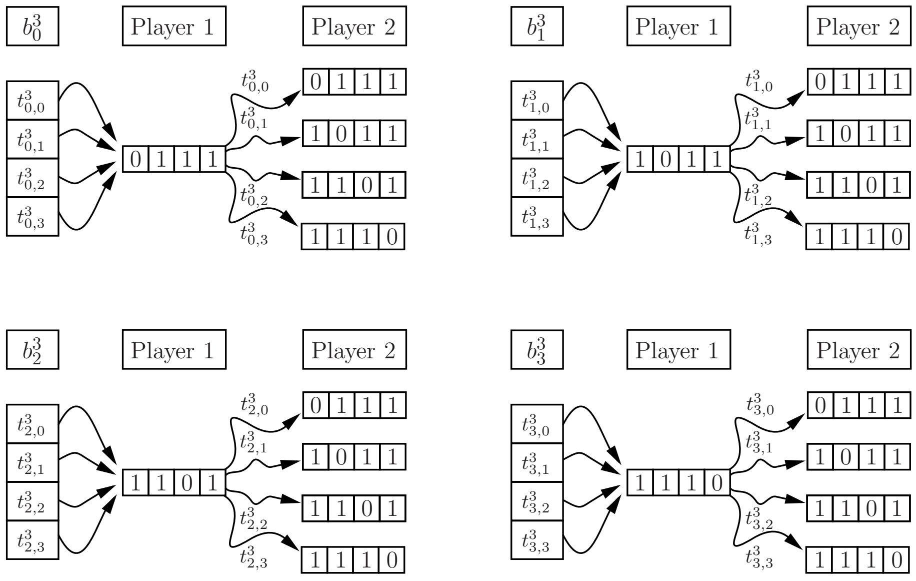

Fig. 3: Block and thread distribution for supports of size 3 (4-action game).

support size i, block j to which the thread belongs, and index k of the thread in the block. Threads are then responsible for computing the strategies for Players 1 and 2, where Player 1 chooses a support element to play against Player 2's choice of support element. Thus, each thread will be responsible for processing a single pair of supports. For instance, thread $t_{2,1}^3$ in Figure 3, calculates the strategy for Player 1 choosing a support element identified by support key σ_2^3 and the strategy for Player 2 choosing a support element identified by support key σ_1^3. The result from the thread execution is a 2-tuple of candidate mixed strategies (x, y) that is checked for the Nash equilibrium conditions given in Theorem 1.

4 GPU-BASED PARALLEL SUPPORT ENUMERATION ALGORITHM

In this section, we design a GPU-based parallel algorithm for computing Nash equilibria in bimatrix games using the support enumeration method. The main algorithm is GPU-SE which utilizes three functions called from the host machine, Generate, Pure, and Mixed. Generate is the function that generates the array Θ^k of support keys for support size k on the host machine. Pure is the function which computes the pure Nash equilibria. It is a kernel function with the global type qualifier. Mixed is the function which computes the mixed Nash equilibria, also a global kernel function.

4.1 GPU-SE: Parallel Support Enumeration

The GPU-SE function, presented in Algorithm 2, is responsible for calling all functions from the host machine. Its input parameters are the two player's payoff arrays A and B. The output $\mathcal{E}$ is a container which stores all Nash equilibria. The container $\mathcal{E}$ is used for transferring solutions from the GPU to the host. GPU-SE finds the minimum number of actions in the game (Line 4). This ensures that both players choose from the same number of actions. In Lines 5 and 8, Θ is the support key array that identifies all possible supports for a given support size. The pure strategy Nash equilibria are computed by calling Pure (Line 6), while the mixed strategy Nash equilibria are computed by calling Mixed for each support size (Lines 7-10).

Algorithm 2 *host* GPU-SE(A, B)

```
1: Input: Player 1 payoff, Player 2 payoff ($A$, $B$)
2: Output: Set of equilibria ($\mathcal{E}$)
3: $\mathcal{E} = \emptyset$
4: $q = \min(m, n)$
5: $\Theta$ = Generate(1, $q$)
6: $\mathcal{E}$ = Pure($A$, $B$, $q$, $\Theta$)
7: for $k = 2, \ldots, q$ do
8:     $\Theta$ = Generate($k$, $q$)
9:     $\mathcal{E} = \mathcal{E} \cup$ Mixed($A$, $B$, $k$, $q$, $\Theta$)
10: output $\mathcal{E}$
```

Algorithm 3 *host* Generate(k, q)

```
1: Input: Support size, Number of actions ($k$, $q$)
2: Output: Array of support keys for support size $k$ ($\Theta^k$)
3: for $i = 0, \ldots, q-1$ do
4:     $aux[i] = 0$
5: for $i = (q-k), \ldots, q-1$ do
6:     $aux[i] = 1$
7: $i = 0$
8: repeat
9:     for $j = 0, \ldots, q-1$ do
10:        $\Theta^k[q \cdot i + j] = aux[j]$
11:    [$flag$, $aux$] = next($q$, $aux$)
12:    $i$++
13: until ($flag$)
14: return $\Theta^k$
```

Algorithm 4 *global* Pure(A, B, q, Θ)

```
1: Input: Player 1 payoff, Player 2 payoff, Number of actions, Array of support keys ($A$, $B$, $q$, $\Theta$)
2: Output: Set of pure equilibria ($\mathcal{E}^p$)
3: $\mathcal{E}^p = \emptyset$
4: for $i = 0, \ldots, q-1$ do
5:     $x[i]$, $y[i]$, $p_1[i]$, $p_2[i]$ = 0
6: $idx$, $idy = 0$
7: for $i = 0, \ldots, q-1$ do
8:     $x[i]$ = $\Theta[q \cdot$ blockIdx.x + $i]$
9: for $i = 0, \ldots, q-1$ do
10:    for $j = 0, \ldots, q-1$ do
11:        $p_1[i]$ += $x[j] \cdot B[q \cdot j + i]$
12: $idx$ = $\arg\max_l\{p_1[l]\}$
13: for $i = 0, \ldots, q-1$ do
14:    $y[i]$ = $\Theta[q \cdot$ threadIdx.x+ $i]$
15: for $i = 0, \ldots, q-1$ do
16:    for $j = 0, \ldots, q-1$ do
17:        $p_2[i]$ += $y[j] \cdot A[q \cdot i + j]$
18: $idy$ = $\arg\max_l\{p_2[l]\}$
19: if ($\Theta[q \cdot$ blockIdx.x + $idy]$ = 1 & $\Theta[q \cdot$ threadIdx.x+ $idx]$ = 1) then
20:    $\mathcal{E}^p = \mathcal{E}^p \cup (x, y)$
21: return $\mathcal{E}^p$
```

4.2 Generate: **Generating Support Keys**

The Generate function, given in Algorithm 3, is responsible for creating the support key array for a given support size. Generate requires as input the support size k, the number of actions q, and outputs the support key array Θ^k. To generate Θ^k, the host function next is called, which implements the lexicographical arrangement algorithm presented in [26]. The algorithm produces all possible ways to arrange the support keys for a given support size. In addition, the next function returns 0 in the *flag* variable when all possible arrangements have been produced. The auxiliary array, *aux*, is used as a temporary container to hold the initial support key for support size k. The function will directly modify *aux* by returning it as the next support key arrangement which is then stored in Θ^k (Line 11). The process continues until the last possible arrangement has been generated using the lexicographical ordering on the support key stored in *aux* and the *flag* variable is set to 0.

4.3 Pure: **Computing Pure Strategy Nash Equilibria**

The Pure function, presented in Algorithm 4, is responsible for calculating the pure strategy Nash Equilibria. Since the pure strategies involve a single action, the probability of using that action is 1 and only a simple search is required to find the maximum expected payoff for Players 1 and 2. The function copies the support key into x, the strategy of Player 1 (Lines 7-8) and then generates all expected payoff values p_1 for Player 1 given the probability of selecting an action (Lines 9-11). It then finds the maximum expected payoff value for Player 1 and the index corresponding to that value, which is stored in idx (Line 12). The same actions are performed for Player 2, the support key is copied into y, the strategy of Player 2 (Lines 13-14), and the expected payoff p_2 is generated (Lines 15-17). Then, the

Algorithm 5 *global* Mixed(A, B, k, q, Θ)

```
1: Input: Player 1 payoff, Player 2 payoff, Support size, Number of actions, Array of support keys ($A$, $B$, $k$, $q$, $\Theta$)
2: Output: Set of mixed equilibria ($\mathcal{E}^m$)
3: $\mathcal{E}^m = \emptyset$
4: for $i = 0, \ldots, q-1$ do
5:     $x[i]$, $y[i]$, $p_1[i]$, $p_2[i]$, $pay$[i], $z[i]$ = 0
6: $idx_1$, $idx_2$, $idy_1$, $idy_2 = 0$
7: $z[(k$ - 1)] = 1
8: $pay$ = Transform($B$, $k$, $q$, $\Theta$)
9: $pay$ = Array-ludcmp($k$, $pay$)
10: $z$ = Array-bcksub($z$, $k$, $pay$)
11: for $i = 0, \ldots, q-1$ do
12:    if $\Theta[q \cdot$ blockIdx.x + $i]$ = 0 then
13:        $x[i] = \Theta[q \cdot$ blockIdx.x + $i] \cdot z[i]$
14:    else
15:        $x[i] = \Theta[q \cdot$ blockIdx.x + $i] \cdot z[idx_1]$
16:        $idx_1$++
17: for $i = 0, \ldots, q-1$ do
18:    for $j = 0, \ldots, q-1$ do
19:        $p_1[i]$ += $x[j] \cdot B[q \cdot j + i]$
20: $idx_2$ = $\arg\max_l\{p_1[l]\}$
21: if $\Theta[q \cdot$ threadIdx.x + $idx_2]$ = 1 then
22:    $proceed$ = TRUE
23: else
24:    $proceed$ = FALSE
25: if ($proceed$) then
26:    $z[(k$ - 1)] = 1;
27:    $pay$ = Transform($A$, $k$, $q$, $\Theta$)
28:    $pay$ = Array-ludcmp($k$, $pay$)
29:    $z$ = Array-bcksub($z$, $k$, $pay$)
30:    for $i = 0, \ldots, q-1$ do
31:        if $\Theta[q \cdot$ threadIdx.x + $i]$ = 0 then
32:            $y[i] = \Theta[q \cdot$ threadIdx.x + $i] \cdot z[i]$
33:        else
34:            $y[i] = \Theta[q \cdot$ threadIdx.x + $i] \cdot z[idy_1]$
35:            $idy_1$++
36:    for $i = 0, \ldots, q-1$ do
37:        for $j = 0, \ldots, q-1$ do
38:            $p_2[i]$ += $y[j] \cdot A[q \cdot i + j]$
39:    $idy_2$ = $\arg\max_l\{p_2[l]\}$
40:    if $\Theta[q \cdot$ blockIdx.x + $idy_2]$ = 1 then
41:        $proceed$ = TRUE
42:    else
43:        $proceed$ = FALSE
44: if ($proceed$) then
45:    $\mathcal{E}^m = \mathcal{E}^m \cup (x, y)$
46: return $\mathcal{E}^m$
```

the maximum expected payoff value p_2 for Player 2 is computed and the index corresponding to that value is stored in idy (Line 18). After this the function determines if both the index of Player 2's highest expected payoff is the response to Player 1's choice of action and the index of Player 1's highest expected payoff is the response to Player 2's choice of action. If these two conditions are satisfied, then the best pure response conditions for both players are satisfied. The function stores the strategies x and y as a 2-tuple in $\mathcal{E}^p$ (the set of pure equilibria) which is then returned to the GPU-SE algorithm (Lines 19-21).

4.4 Mixed: **Computing Mixed Strategy Nash Equilibria**

The Mixed function, presented in Algorithm 5, is responsible for calculating the mixed strategy equilibria. The implementation of this function requires solving a system of equations to determine the probabilities of a player choosing an action against the other player's choice of action. Mixed calls three device functions; Transform, Array-ludcmp, and Array-bcksub. Transform modifies the player's payoff array by placing the coefficient of the constraint equations (4) and (6) on the last entries of the array. The last entries are eliminated from the array using similar operations that would perform row elimination in a matrix. This process transforms the player's payoff array into an array representing a system of equations with constraints. Array-ludcmp is a function implementing the LU decomposition while Array-bcksub implements the back-substitution method. Both Array-ludcmp and Array-bcksub are array-based implementations presented in [27], where the original versions are suited for matrix processing. We omit describing them in this paper but refer the reader to [27] for the description of the general methods.

The function uses a temporary solution array z when solving for a player's probabilities of choosing an action. The last entry in z is set to one, satisfying constraint equations (4) and (6) (Line 7). The Transform function modifies the payoff array such that the last entries in the array are substituted with the coefficients of the constraint equations. After the Transform function returns from the device, the result is stored in the *pay* array (Line 8). The modified payoff array is decomposed into lower and upper triangular partitions, which are stored in *pay* (Line 9). The solution of the system of linear equations is determined by back-substitution and it is stored back in z (Line 10). The solution z does not yet take into account which actions are available according to the support key. For instance, suppose Player 1 chooses a support element identified by σ_1^2 then the probabilities should reflect the actions identified by the support key which are the second and fourth actions (see Figure 2). When the solution z is returned from Array-bcksub (Line 10), the two probabilities are in the first and second entry. The position of the probabilities are reorganized according to the support key, where the first probability will be in the second entry of x and the second probability will be in the fourth entry of x coinciding with σ_1^2 (Lines 11-16). The expected payoff to Player 1 is then determined (Lines 17-20). The function searches for the index of the action that represents the highest expected payoff value in p_1 for Player 1 (Line 21). If the returned index is the response to Player 2's choice of action according to the support key (Line 21), then the boolean variable $proceed$ is updated to TRUE. If $proceed$ is updated to FALSE, then there is no need to further execute the function for this support pair. If $proceed$ is TRUE, then the same sequence of actions as in the case of Player 1 are performed with respect to Player 2 (Lines 36-43). If both sections of the function result in $proceed$ being TRUE, then the strategies x and y are stored as a 2-tuple in $\mathcal{E}^m$ (the set of mixed equilibria) and $\mathcal{E}^m$ is returned to the GPU-SE algorithm (Lines 44-46).

The Transform function manipulates the player payoff array information and adds the constraints (4) and (6) from Section 2 to create a system of equations represented as an array. There are two calls to Transform from Mixed function, each with respect to the original payoff arrays A and B. In addition, the function identifies the actions which are part of the support through the support keys. Modifying the payoff array by taking into consideration which actions are part of the support is necessary to produce the correct system of equations. This is done in the triple-nested *for* loop, in Lines 7 through 14 for Player 1, and in Lines 16 through 23 for Player 2, respectively. To avoid direct manipulation of the original payoff arrays, $auxpay$ is created in order to store the temporary results. The coefficients of the constraint equations are included in the $auxpay$ which is now ready to be passed to the Array-ludcmp and Array-bcksub functions (Lines 24-25).

Algorithm 6 *device* Transform($\{A, B\}$, k, q, Θ)

```
1: Input: {Player 1, Player 2} payoff, Support size, Number of
   actions, Array of support keys ($\{A, B\}$, $k$, $q$, $\Theta$)
2: Output: Modified player payoff ($auxpay$)
3: for $i = 0, \ldots, q-1$ do
4:     for $j = 0, \ldots, q-1$ do
5:         $auxpay[q \cdot i + j] = 0$
6: if $B$ then
7:     for $i = q-1, \ldots, 0$ do
8:         if $\Theta[q \cdot \texttt{threadIdx.x} + i] = 1$ then
9:             for $j = 0, \ldots, i$ do
10:                if $\Theta[q \cdot \texttt{threadIdx.x} + j] = 1$ then
11:                    for $l = 0, \ldots, i$ do
12:                        if $\Theta[q \cdot \texttt{blockIdx.x} + l] = 1$ then
13:                            $auxpay[index] = B[q \cdot l + j] - B[q \cdot l + i]$
14:                            $index{+}{+}$
15: if $A$ then
16:     for $i = q-1, \ldots, 0$ do
17:         if $\Theta[q \cdot \texttt{blockIdx.x} + i] = 1$ then
18:             for $j = 0, \ldots, i$ do
19:                 if $\Theta[q \cdot \texttt{blockIdx.x} + j] = 1$ then
20:                     for $l = 0, \ldots, i$ do
21:                         if $\Theta[q \cdot \texttt{threadIdx.x} + l] = 1$ then
22:                             $auxpay[index] = A[q \cdot j + l] - A[q \cdot i + l]$
23:                             $index{+}{+}$
24: for $i = 0, \ldots, k$ do
25:     $auxpay[k \cdot (k - 1) + i] = 1$
26: return $auxpay$
```

4.5 Memory Access and Allocation

We optimize the execution of GPU-SE for faster memory accesses by identifying which data can be stored on the GPU's specialized memory areas. Since the payoff arrays A, B are not modified during the execution of GPU-SE, we allocate them to constant memory on the GPU hardware. Since constant memory is cached, repetitive thread accesses to the same memory locations when calculating candidate Nash equilibrium strategies will not require additional memory traffic. In Algorithms 2-6, we specified the payoff arrays A, B as inputs to the functions to facilitate the description of these algorithms, but in the actual implementation these two arrays are available from constant memory and do not need to be passed.

The support key array Θ is an input parameter to Pure, Mixed, and Transform functions and is accessed at the thread-level to identify strategies and compare payoffs. After transferring Θ to Pure or Mixed from GPU-SE, each of the kernels restructure Θ as either a two-dimensional shared memory array or a two-dimensional coalesced global memory array. For games with less than 10 actions, Θ can be stored on shared memory and accessed very fast. For games with more than 10 actions, more memory is required for storing Θ than the available shared memory (96 KB); therefore, having to default to global memory access. Restructuring Θ as a two-dimensional global memory array and making sure the threads in a warp access memory sequentially (i.e., coalescing), we were able to reduce the access times substantially.

In addition to optimizing the memory accesses we also optimize the data transfers between CPU and GPU by using the facilities provided within the Thrust library [28]. Thrust is a GPU-specific library for parallel algorithms and data structures accessible in CUDA Toolkit (v5.5) [29]. With respect to our current implementation, we manage transfers of data between CPU and GPU through kernel executions using data structures within Thrust library for fast and efficient memory copy, allocation, and manipulation. In the implementation of our proposed algorithm, we use the data structures `thrust::device_vector` and `thrust::host_vector` for the support key array Θ and the equilibria container $\mathcal{E}$. For games with more than 14 actions, the equilibria containers were too large to store in local memory. Instead, we utilize the print functionality which is available in CUDA compute capability 2.x and higher, to output the results.

5 Analysis of GPU-SE

In this section, we provide a formal description of the thread-block workload allocation strategy implemented in the GPU-SE algorithm and analyze the running time of GPU-SE.

5.1 Thread-Block Workload Allocation Strategy

In Section 3.2, we described the parallelization method and provided an example of how the proposed thread-block workload allocation strategy works for a 4-action game. We now describe the thread-block workload allocation strategy for the general case of n-action bimatrix games. Our proposed strategy is designed to maximize the degree of parallelism by allocating the processing of one strategy support pair to a single thread as follows. We allocate to each block, a single support element (identified by its support key) from the set of Player 1's supports, and then, allocate to each thread within the block a support element from the set of Player 2's supports. Thus, each thread will be responsible for processing a single pair of supports. Using this workload allocation strategy, all threads within a given block check the equilibrium conditions for the candidate mixed strategy tuples using the shared memory; thereby reducing the memory access and the overhead of data manipulation.

In order to formally describe the thread-block workload allocation strategy, we determine the number of blocks and threads required for checking the equilibrium conditions on the pairs of candidate mixed strategies. For each support of size i in an n-action game, the number of actions W_i is given by the number of possible combinations of i actions chosen from a set of n-actions, that is, $W_i = \binom{n}{i}$. Using our proposed workload allocation strategy, we associate W_i blocks with Player 1, where each block corresponds to one strategy out of W_i possible strategies of Player 1. Since we are considering square games, the number of actions for each support of size i of Player 2 is also given by $W_i = \binom{n}{i}$. For each of the W_i strategies of Player 2, we allocate a thread within the block, and thus, the number of threads within a block responsible for determining the Nash equilibria from strategies of support size i is given by W_i. GPU-SE executes W_i blocks, where each block contains W_i threads. Therefore, the total number of threads executed by GPU-SE is $W_i \times W_i = W_i^2 = \binom{n}{i}^2$.

For larger games, executing W_i blocks, each containing W_i threads checking all tuples of candidate strategies for equilibrium conditions, reaches a limitation due to the maximum number of threads that can be created within a block. Since our GPU device has a maximum thread count of 1024 per block, this limit is first reached for a 13-action game of support size 5 (1,287 possible strategies). In order to handle such cases, requiring processing of more candidate strategies than the available threads within a block, we increase the number of blocks to $W_i' = W_i(1 + \lfloor \frac{W_i}{1024} \rfloor)$. Thus, in such cases, some of the blocks will consist of fewer than 1024 threads.

5.2 Run-Time Analysis

In this subsection, we investigate the running time of GPU-SE. In order to determine the running time of GPU-SE, we need to determine the amount of work performed by each of the functions called from GPU-SE. The function Generate is called in line 5 of GPU-SE to generate the support keys corresponding to support size 1 (i.e., pure strategies). The amount of work performed by Generate is $O(n)$ since it generates n support keys of size 1. The function Pure is called in line 6 to determine the pure Nash equilibria for the support keys determined by Generate, performing a total amount of work of $O(n^2)$. Thus, the total amount of work performed by these two functions (lines 5-6, Algorithm 1) is given by $O(n^2)$.

The functions Generate and Pure are then called $n-1$ times within the GPU-SE algorithm to generate the support keys of size 2 to n, and respectively, to determine the Nash equilibria corresponding to supports of sizes ranging from 2 to n (lines 7-9, Algorithm 1). We first determine the amount of work performed by Generate (Algorithm 3) which generates the support keys of size

k of the n-action bimatrix game. The number of support keys (i.e., supports) of size k is $\binom{n}{k}$, which gives the amount of work performed by Generate. Since Generate is called $n-1$ times in GPU-SE, the total amount of work performed by Generate in the main loop of Algorithm 1 is given by $\sum_{k=2}^{n}\binom{n}{k} = 2^n - (n+1)$.

We now determine the total amount of work performed by Mixed (Algorithm 4). The work performed by Mixed is mainly determined by the work performed by the function Transfer (Algorithm 6) and the LU decomposition function Array-ludcmp. The triple for-loop in Transfer, under the provided bounds, results in a total amount of work of $O(n^3)$. The if-statements within Transfer do not reduce the amount of work since each thread will perform the three for-loops according to their thread-block IDs. The array based LU decomposition function Array-ludcmp (Lines 9 and 10, Lines 28) in Mixed performs a total amount of work of $O(n^3)$. Mixed explores all possible pairs of strategies of supports of size k, $k = 2, \ldots, n$, and since the total number of supports of sizes 2 to n for one player is $2^n - (n+1)$, then the total number of pairs checked by Mixed is given by $(2^n - (n+1))^2$, which is $O(4^n)$. Since checking the equilibrium conditions for each pair amounts to $O(n^3)$, then the total amount of work performed by the main loop of GPU-SE is given by $O(4^n n^3)$. Thus, the total amount of work performed by GPU-SE is $O(4^n n^3 + n^2)$ which is $O(4^n n^3)$.

Since each thread is responsible for processing a pair of supports, the total amount of work performed by a single thread is given by $O(n^3)$. To determine the parallel running time, we assume that the number of threads that can be executed in parallel at any given time by the GPU device is T. Therefore, the total parallel execution time of GPU-SE is given by $O(n^3 \frac{4^n}{T})$. If $T = \Omega(4^n)$ the parallel running time becomes $O(n^3)$.

6 Experimental Results

We perform extensive experiments to compare the performance of the proposed GPU-based parallel support enumeration algorithm GPU-SE with that of two other parallel support enumeration algorithms, OMP-SE (for shared-memory parallel machines, based on OpenMP [30]) and MPI-SE (for message-passing systems, based on MPI [31]).

6.1 Experimental Setup

GPU-SE is executed on a 2.6 GHz Intel®Core$^{\text{TM}}$2 Quad CPU Q8400 Dell OptiPlex 780 64-bit system using 4.6 GB RAM. This system supports an NVIDIA$^{\text{TM}}$GeForce GT 440 graphics processing unit with two 48 cores streaming multiprocessors and 2.6 GB RAM.

OMP-SE and MPI-SE are executed on the Wayne State University Grid system [32]. Our experiments use compute nodes on the grid, where each node consists of a 16 core 2.6 GHz Quad processor and 128 GB RAM. The nodes are connected using a 10Gb Ethernet network. The algorithm is executed using five different parallel configurations: 1, 2, 4, 8, and 16 processors.

When comparing execution times obtained on GPUs with those obtained on systems with standard CPUs, the research literature favors the GPUs. Gregg et al. [33] makes the case for determining better ways to compare the run times in order to take into account the memory transfers and other activities managed by the CPU for GPU processing. We have considered this and included timing functions to record every operation for both the GPU and CPU implementations which accounts for all initialization, memory transfers, core computation, and output.

To conduct the experiments, we randomly generate a set of games using GAMUT [14]. We chose as benchmark the *Minimum Effort Games*, where the payoff for an action is dependent on the effort associated with the action minus the minimum effort of the selected player. Player payoffs are calculated using the formula $a + bE_{min} - cE$, where E_{min} is the minimum effort of a player in the game, E is the effort of that player, and a, b, c are constants, where $b > c$. In these games, the players have the same number of actions. The GAMUT arguments used when generating the games are: -int_payoffs -output TwoPlayerOutput -players 2 -actions n -g MinimumEffortGame -random_params. The value of n determines the number of player actions, and thus, the size of the test games. The number of players' actions in a game is the size of the game. We experiment with games of different sizes ranging from 5 to 16 actions. For each of the twelve game sizes, we randomly generate five games and use them as test cases.

6.2 OpenMP-based implementation

In this subsection, we present the OpenMP-based implementation of the support enumeration method for finding Nash equilibria in bimatrix games (called OMP-SE). OpenMP is a shared-memory parallel programming model. OpenMP uses multithreading, where a master thread forks a specified number of slave threads. Tasks are divided among threads and the threads are assigned to different processors in order to run concurrently.

The OMP-SE function is given in Algorithm 7. We consider T threads available in the system, where each thread is responsible for checking the Nash equilibrium condition in Theorem 1 for a subset of support pairs assigned in a round-robin fashion. Each thread has a *counter* variable which selects a subset of supports (Line 6). This implementation processes the support sets M_x and N_y grouped by support size in parallel. Candidate solutions (x, y) are determined for each support size in parallel using LU decomposition and back-substitution (Lines 11-14). If a Nash equilibrium is found by a thread, it is saved in the output variable $\mathcal{E}$ (Line 16).

6.3 MPI-based implementation

In this subsection, we present the MPI-based implementation of the support enumeration method for finding

Algorithm 7 OMP-SE(A, B, T)

```
1: Input: Player 1 payoff, Player 2 payoff, Number of threads ($A$, $B$, $T$)
2: Output: Set of equilibria ($\mathcal{E}$)
3: $q = \min(m, n)$
4: for each thread $t = 1, \ldots, T$ do in parallel
5:     $t.\mathcal{E} = \emptyset$
6:     $t.counter \leftarrow 0$
7:     for $k = 1, \ldots, q$ do
8:         for each $(M_x, N_y), M_x \subseteq M, N_y \subseteq N, |M_x| = |N_y| = k$ do
9:             if $(t.counter \bmod T) + 1 = t$ then
10:                Solve:
11:                    $\sum_{i \in M_x} x_i B_{ij} = v, \ \forall j \in N_y$
12:                    $\sum_{i \in M_x} x_i = 1$
13:                    $\sum_{j \in N_y} y_j A_{ij} = u, \ \forall i \in M_x$
14:                    $\sum_{j \in N_y} y_j = 1$
15:                if $x_i, y_j \geq 0, \forall\ i, j$ and $x, y$ satisfies Theorem 1 then
16:                    $t.\mathcal{E} \leftarrow t.\mathcal{E} \cup (x, y)$
17:            $t.counter$++
18:    output $t.\mathcal{E}$
```

Algorithm 8 MPI-SE(A, B, P)

```
1: Input: Player 1 payoff, Player 2 payoff, Number of processors ($A$, $B$, $P$)
2: Output: Set of equilibria ($\mathcal{E}$)
3: MPI_Bcast($A$, $B$)
4: for $p = 1, \ldots, P - 1$ do in parallel
5:     Processor $p$:
6:     $p.\mathcal{E} = \emptyset$
7:     $q = \min(m, n)$
8:     $counter \leftarrow 0$
9:     for $k = 1, \ldots, q$ do
10:        for each $(M_x, N_y), M_x \subseteq M, N_y \subseteq N, |M_x| = |N_y| = k$ do
11:            if $(counter \bmod P) = p$ then
12:                Solve:
13:                    $\sum_{i \in M_x} x_i B_{ij} = v, \ \forall j \in N_y$
14:                    $\sum_{i \in M_x} x_i = 1$
15:                    $\sum_{j \in N_y} y_j A_{ij} = u, \ \forall i \in M_x$
16:                    $\sum_{j \in N_y} y_j = 1$
17:                if $x_i, y_j \geq 0, \forall\ i, j$ and $x, y$ satisfies Theorem 1 then
18:                    $p.\mathcal{E} = p.\mathcal{E} \cup (x, y)$
19:            $counter$++
20: MPI_Gather($p.\mathcal{E}$)
21: output $\mathcal{E}$
```

Nash equilibria in bimatrix games. MPI is a message-passing interface used for designing programs leveraging multiple processors. MPI defines a standard library of message-passing routines allowing multiple processors to cooperate. We consider P processors available, where each processor p is responsible for checking the Nash equilibrium conditions in Theorem 1 for a subset of support pairs assigned in a round-robin fashion.

The MPI-SE function, shown in Algorithm 8, is the implementation proposed by Widger and Grosu [18]. Like the OpenMP implementation, MPI-SE also processes the support sets M_x and N_y grouped by support size in parallel. Each processor p is sent a copy of the payoff matrices A and B through a broadcast routine, MPI_Bcast (Line 3). This reduces communication overhead, whereby each processor p identifies its partition of support sets to check for the Nash equilibrium conditions in Theorem 1. As a result, excess time does not need to be spent on transferring partial payoff information between processors. Candidate solutions (x, y) are also determined for each support size in parallel using LU decomposition and back-substitution (Lines 13-16). If a Nash equilibrium is found by a processor, it is saved in the output variable $p.\mathcal{E}$ (Line 18). At the end of MPI-SE each processor's equilibria result is aggregated into $\mathcal{E}$ using an all-to-one gather routine, MPI_Gather (Line 20).

6.4 Analysis of Results

We use the execution time and speedup as the metrics for comparing the performance of GPU-SE, OMP-SE, and MPI-SE.

6.4.1 GPU-SE vs. OMP-SE

In Figure 4, we plot the average execution time of GPU-SE and OMP-SE for games of different sizes (given by the number of actions n). The horizontal axis represents the number of actions, while the vertical axis represents the run times in seconds on a log scale.

The OMP-SE running on 16 CPUs performs worse than when it runs on a single CPU for games with the number of actions less than or equal to 6. This is due to the overhead induced by the thread generation and management. By efficiently managing the memory accesses in GPU-SE, its execution times for games with smaller actions ($n < 10$) are almost the same as those of the various configurations of OMP-SE. For games with 10 actions (n = 10), which require the processing of 184,755 support pairs, the GPU-SE algorithm outperforms all OMP-SE CPU configurations, where the lowest average execution time of OMP-SE is 0.148 seconds using the 16 CPU configuration against the GPU-SE average execution time of 0.104 seconds. For games with larger than 10 ($n > 10$) actions, GPU-SE consistently outperforms all OMP-SE CPU configurations within our experiment.

In Figure 5, we plot the average speedup against the number of actions. The horizontal axis represents the number of actions, while the vertical axis represents the speedup on a log scale. Here, the speedup is defined as the ratio of the OMP-SE parallel execution time over the GPU-SE execution time. When the ratio is greater than 1, the GPU-SE algorithm is faster than the corresponding OMP-SE configurations. Each data point represents the average speedup for each parallel configuration in five games having the same number of actions.

For games with 7 through 9 actions, the GPU-SE speedup ratio is consistently greater than 1 against OMP-SE with 1 and 2 CPU configurations. When n = 10, the GPU-SE speedup ratio is greater than 1 for every OMP-SE CPU configuration with a lowest speedup ratio of 1.29 against the 8 CPU configuration and a highest speedup ratio of 5.43 against the single CPU configuration. For n = 16, the GPU-SE algorithm obtains speedups of 108.98, 63.87, 36.95, 22.06, and 15.36 against the 1, 2,

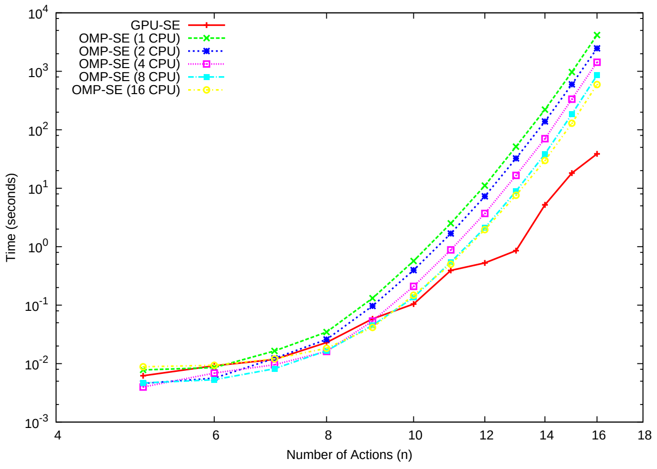

Fig. 4: GPU-SE vs. OMP-SE: Average execution time.

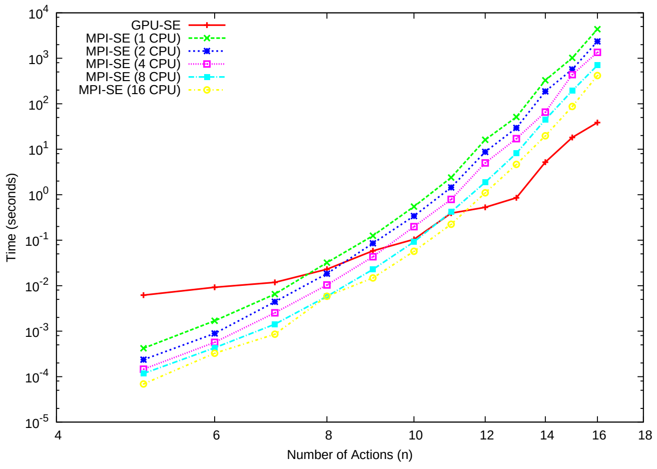

Fig. 6: GPU-SE vs. MPI-SE: Average execution time.

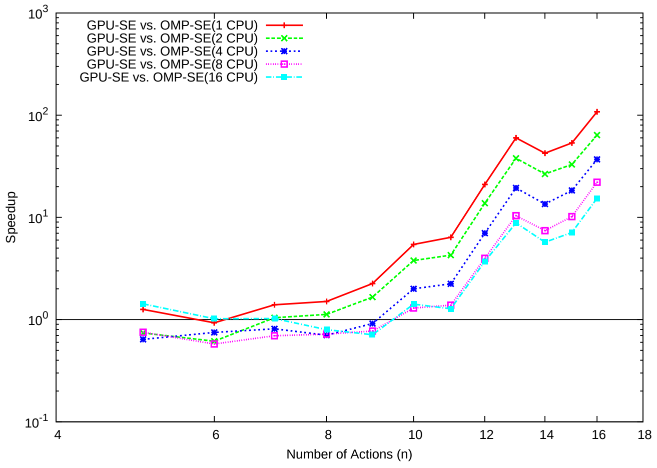

Fig. 5: GPU-SE vs. OMP-SE: Average speedup.

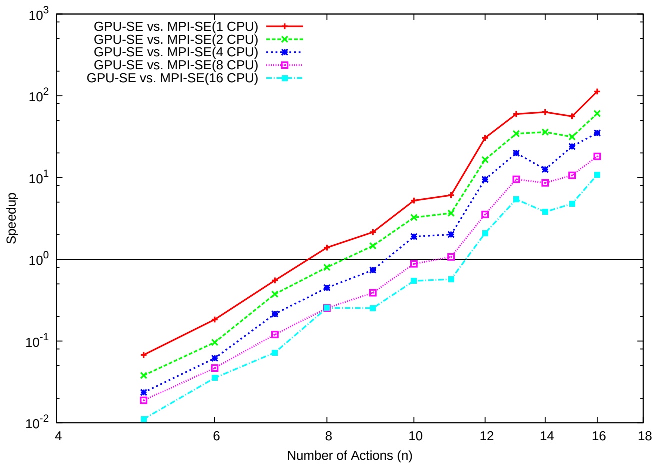

Fig. 7: GPU-SE vs. MPI-SE: Average speedup.

4, 8, and 16 OMP-SE CPU configurations, respectively. Thus, the GPU-SE is able to obtain a significant reduction in the execution time compared to OMP-SE when solving large scale games.

6.4.2 GPU-SE vs. MPI-SE

In Figure 6, we plot the average execution time of GPU-SE and OMP-SE for games of different sizes, where the horizontal axis represents the number of actions and the vertical axis represents the run time in seconds on a log scale. For our experiments, the larger the number of CPUs the better the performance of MPI-SE for games of any size.

The performance of GPU-SE is surpassed by all MPI-SE CPU configurations for games with less than 8 actions ($n < 8$). For games with 8 actions ($n = 8$), the GPU-SE algorithm outperforms only the single MPI-SE CPU configuration. For games with more than 12 actions ($n \geq 12$), GPU-SE is faster than all MPI-SE CPU configurations.

In Figure 7, we plot the average speedup against the number of actions. The horizontal axis again represents the number of actions, while the vertical axis represents the speedup on a log scale. Here, the speedup is the ratio of the MPI-SE parallel execution time over the GPU-SE execution time. GPU-SE shows a speedup greater than 1, approximately 1.39, against the single MPI-SE CPU configuration. For games with 12 actions ($n = 12$), GPU-SE obtains a greater speedup against all MPI-SE CPU configurations. For games with 16 actions ($n = 16$), the GPU-SE algorithm obtains speedups of 112.90, 60.81, 35.04, 18.16 and 10.77 against MPI-SE CPU configurations using 1, 2, 4, 8, and 16 processors, respectively. Thus, the GPU-SE is able to obtain a significant reduction in the execution time compared to MPI-SE when solving large scale games.

7 Conclusion and Future Work

We designed and implemented a GPU-based parallel support enumeration algorithm for calculating Nash equilibria in bimatrix games. We designed a new parallelization method that exploits the available degree of parallelism when computing Nash equilibria. We experimented with multiple games of different sizes and configurations. Our analysis showed that the proposed algorithm, GPU-SE, achieved lower execution times and significant speedups for large games. The GPU-SE algorithm produced 100 times faster executions versus the OpenMP and MPI-based versions by taking advantage of the massively parallel processing platform.

Future work will include the design of GPU-based algorithms implementing other methods for computing Nash equilibria such as polynomial homotopy continuation [15] and global Newton [16].

ACKNOWLEDGMENT

This paper is a revised and extended version of [34] presented at the 31st IEEE International Performance Computing and Communications Conference (IEEE IPCCC'12). The authors wish to express their thanks to the editor and the anonymous referees for their helpful and constructive suggestions, which considerably improved the quality of the paper. This research was supported in part by NSF grants DGE-0654014 and CNS-1116787.

REFERENCES

[1] R. B. Myerson, *Game Theory: Analysis of Conflict*. Harvard University Press, 1991.

[2] J. F. Nash, "Equilibrium points in n-person games," *Proc. Nat'l Academy of Sciences of the United States of Am.*, vol. 36, no. 1, pp. 48–49, Jan. 1950.

[3] J. Nash, "Non-cooperative games," *The Annals of Mathematics*, vol. 54, no. 2, pp. 286–295, 1951.

[4] C. Lemke and J. Howson Jr, "Equilibrium points of bimatrix games," *Journal of the Society for Industrial and Applied Mathematics*, pp. 413–423, 1964.

[5] R. Savani and B. von Stengel, "Hard-to-solve bimatrix games," *Econometrica*, vol. 74, no. 2, pp. 397–429, 2006.

[6] C. Daskalakis, P. Goldberg, and C. Papadimitriou, "The complexity of computing a Nash equilibrium," in *Proc. of the 38th annual ACM Symposium on Theory of Computing*, 2006, pp. 71–78.

[7] C. H. Papadimitriou, *The Complexity of Finding Nash Equilibria*. Cambridge Univ. Press, 2007, ch. 2, Algorithmic Game Theory, (eds. N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani), pp. 29–52.

[8] B. von Stengel, *Equilibrium computation for two-player games in strategic and extensive form*. Cambridge Univ. Press, 2007, ch. 3, Algorithmic Game Theory, (eds. N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani), pp. 53–78.

[9] J. Dickhaut and T. Kaplan, "A program for finding nash equilibria," *Mathematica Journal*, vol. 1, no. 4, pp. 87–93, 1991.

[10] R. D. McKelvey, McLennan, and T. Turocy, "Gambit: Software tools for game theory, Version 0.2007.01.30," Available at http://gambit.sourceforge.net, 2007.

[11] O. L. Mangasarian, "Equilibrium points of bimatrix games," *Journal of the Society for Industrial and Applied Mathematics*, vol. 12, no. 4, pp. 778–780, 1964.

[12] A. Davis, "*lrs*: A revised implementation of the reverse search vertex enumeration algorithm," Available at http://cgm.cs.mcgill.ca/ avis/doc/avis/Av98a.ps, 1999.

[13] A. Marzetta, "Zram: A library of parallel search algorithms and its use in enumeration and combinatorial optimization," Ph.D. dissertation, ETH Zurich, 1998.

[14] K. Leyton-Brown, E. Nudelman, J. Wortman, and Y. Shoham, "Run the gamut: A comprehensive approach to evaluating game-theoretic algorithms," in *Proc. of the 3rd Intl. Joint Conf. on Autonomous Agents and Multiagent Systems*, 2004, pp. 880–887.

[15] R. S. Datta, "Using computer algebra to find Nash equilibria," in *Proc. of the Intl. Symposium on Symbolic and Algebraic Computation*, 2003, pp. 74–79.

[16] S. Govindan and R. Wilson, "A global Newton method to compute Nash equilibria," *Journal of Economic Theory*, vol. 110, no. 1, pp. 65–86, 2003.

[17] B. von Stengel, *Handbook of Game Theory*. North-Holland, 2002, ch. Computing equilibria for two-person games, pp. 1723–1759.

[18] J. Widger and D. Grosu, "Computing equilibria in bimatrix games by parallel support enumeration," in *Proc. of the 7th Intl. Symposium on Parallel and Distributed Computing*, July 2008, pp. 250 –256.

[19] ——, "Computing equilibria in bimatrix games by parallel vertex enumeration," in *Proc. of the Intl. Conf. on Parallel Processing*, Sept. 2009, pp. 116 – 123.

[20] ——, "Parallel computation of Nash equilibria in n-player games," in *Proc. of the 12th IEEE Intl. Conf. on Computational Science and Engineering*, vol. 1, Aug. 2009, pp. 209 – 215.

[21] A. Peters, D. Rand, J. Sircar, G. Morrisett, M. Nowak, and H. Pfister, "Leveraging GPUs for evolutionary game theory," in *Proc. of the GPU Technology Conference*. San Jose, CA, September 2010.

[22] J. Leskinen and J. Priaux, "Distributed evolutionary optimization using Nash games and GPUs. applications to CFD design problems." *Computers and Fluids*, 2012.

[23] A. Bleiweiss, "Producer-consumer model for massively parallel zero-sum games on the GPU," *Intelligent Systems and Control / 742: Computational Bioscience*, 2011.

[24] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani, *Algorithmic Game Theory*. New York, NY, USA: Cambridge University Press, 2007.

[25] NVIDIA, "NVIDIA CUDA C programming guide," April 2012. [Online]. Available: http://developer.download.nvidia.com

[26] J. Arndt, *Matters Computational: Ideas, Algorithms, Source Code*. Springer, 2010.

[27] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in C*, 2nd ed. Cambridge, UK: Cambridge University Press, 1992.

[28] NVIDIA, "Thrust NVIDIA developer zone," October 2013. [Online]. Available: https://developer.nvidia.com/Thrust

[29] ——, "CUDA toolkit," October 2013. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[30] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[31] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI (2nd ed.): portable parallel programming with the message-passing interface*. Cambridge, MA, USA: MIT Press, 1999.

[32] "Wayne State University Grid," http://www.grid.wayne.edu.

[33] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *Proc. of the IEEE Intl. Symposium on Performance Analysis of Systems and Software*, 2011, pp. 134–144.

[34] S. Rampersaud, L. Mashayekhy, and D. Grosu, "Computing nash equilibria in bimatrix games: Gpu-based parallel support enumeration," in *Proc. of the 31st IEEE International Performance Computing and Communications Conference*, December 2012, pp. 332–341.

Safraz Rampersaud received his BSc degree in mathematics and his MSc degree in applied mathematics focusing on optimization from Wayne State University, Detroit, Michigan. He is currently a Ph.D. candidate in computer science at Wayne State University. His research interests include distributed systems, computational finance, and game theory. He is a student member of IEEE and SIAM.

Lena Mashayekhy received her BSc degree in computer engineering-software from Iran University of Science and Technology, and her MSc degree from the University of Isfahan. She is currently a PhD candidate in computer science at Wayne State University, Detroit, Michigan. Her research interests include distributed systems, cloud computing, big data, game theory and optimization. She is a student member of the ACM, the IEEE, and the IEEE Computer Society.

Daniel Grosu received the Diploma in engineering (automatic control and industrial informatics) from the Technical University of Iaşi, Romania, in 1994 and the MSc and PhD degrees in computer science from the University of Texas at San Antonio in 2002 and 2003, respectively. Currently, he is an associate professor in the Department of Computer Science, Wayne State University, Detroit. His research interests include parallel and distributed computing, resource allocation, computer security, and topics at the border of computer science, game theory and economics. He has published more than eighty peer-reviewed papers in the above areas. He has served on the program and steering committees of several international meetings in parallel and distributed computing. He is a senior member of the ACM, the IEEE, and the IEEE Computer Society.