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

Safraz Rampersaud
Department of Computer Science
Wayne State University
Detroit, MI 48202, USA
Email: safraz@wayne.edu

Lena Mashayekhy
Department of Computer Science
Wayne State University
Detroit, MI 48202, USA
Email: mlena@wayne.edu

Daniel Grosu
Department of Computer Science
Wayne State University
Detroit, MI 48202, USA
Email: dgrosu@wayne.edu

***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 faster execution times it is essential to exploit the available parallelism 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-based parallel implementation of the support enumeration method running on conventional multicore machines.**

I. Introduction

Game theory studies the interaction between strategic decision-makers [1]. Over the past fifty years, simulating systems by "gamifying" its agents interactions allowed researchers to better understand agent's behavior in strategic situations. Arguably, the most famous equilibrium concept from game theory is the Nash equilibrium [2], 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 the game. The Nash equilibrium concept has been used in diverse fields such as economics, biology, politics, and computer science to understand agent behavior in competing situations.

Real world strategic interactions usually require the modeling of large number of agents having a large number of choices or actions. For instance, computers worldwide requesting resources over a large time scale 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 wants to model large scale games, i.e., system interactions, large number of actions per player, etc. 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 parallel algorithm for computing Nash equilibria that exploits the power of the existing GPU systems.

One of the first algorithms for computing Nash equilibria in bimatrix games was proposed by Lemke and Howson [3]. This algorithm finds a single Nash equilibrium. Since this method finds only one Nash equilibrium in a game, even if there are multiple equilibria available, the obvious limitation is in not identifying all possible equilibria. On the other hand the algorithm is expected to complete in a reasonable time. The complexity of computing Nash equilibria has been investigated by Daskalakis [4] who showed that Nash is PPAD-complete, where PPAD stands for Polynomial Parity Arguments on Directed Graphs.

Results such as those of Datta [5] promote analytic development. 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 simulate game interactions and solve for equilibrium. In particular, Datta's investigation of using computer algebra [5] in games surveys two methods; *Gröbner* bases using 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 [6] proposed a robust method for finding equilibria in finite games by using topological properties and combining path-following algorithms. von Stengel [7] provided a comprehensive survey of methods for computing Nash equilibria in finite games.

A. Related Work

Parallel algorithms for computing the Nash equilibria have been investigated by Widger and Grosu [8], [9], [10]. The closest work to ours is Widger and Grosu [8] that proposed a

978-1-4673-4883-6/12/$31.00 ©2012 IEEE

parallel support enumeration algorithm specifically designed for message-passing architectures. Widger and Grosu [9], also proposed a message-passing parallel algorithm for computing 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 [10] proposed a parallel algorithm for computing Nash equilibria in n-player games based on polynomial continuation methods. The research literature at the intersection of game theory and GPU processing is very limited. Peters et al. [11] leveraged the GPU platform to model behavioral strategies using equilibrium concepts from evolutionary game theory. Leskinen et al. [12] used GPU processing on Nash games for evolutionary optimization. Bleiweiss [13] exploited the massively parallel GPU architecture to model producer-consumer zero-sum games.

B. Our Contribution

We design a GPU-based parallel support enumeration algorithm for computing 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 such as [8] 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.

C. Organization

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

II. Bimatrix Games & Equilibria Computation

In this section, we present the support enumeration method for solving Nash equilibria in bimatrix games [1], [14]. A bimatrix game [2] 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.
  - Player 1's set of actions: $M = (s_1, s_2, \ldots, s_m)$
  - Player 2's set of actions: $N = (t_1, t_2, \ldots, t_n)$
- Payoff matrices A, $B \in \mathbb{R}^{m \times n}$ corresponding to Player 1 and Player 2.

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 [2]. 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 [7].

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

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

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

where 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 [7].

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:

$$\sum_{i \in M_x} x_i B_{ij} = v, \ \forall j \in N_y \tag{1}$$

$$\sum_{i \in M_x} x_i = 1 \tag{2}$$

and

$$\sum_{j \in N_y} y_j A_{ij} = u, \ \forall i \in M_x \tag{3}$$

$$\sum_{j \in N_y} y_j = 1 \tag{4}$$

Formally, the set of equations (1) and (2) 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 (3) and (4) 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.

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 to 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 (1) to (4) (Lines 8 to 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 Φ (Line 13).

The complexity of SEQ-SE has been shown to be $O((n+m)^3 \binom{m+n}{n})$, where $m > n$ [8]. This results from the complexity of solving the system of linear equations in $O((n+m)^3)$ for all possible strategy pairs $O(\binom{m+n}{n})$ of the two players.

Example. To show how the support enumeration algorithm works, we consider the following game as an example where both players have two actions:

	L	R
T	5,5	5,2
B	4,7	10,10

(5)

Algorithm 1 SEQ-SE(A, B)

1: **Input:** Player 1 payoff, Player 2 payoff (A, B)
2: **Output:** Set of equilibria (Φ)
3: $\Phi = 0$
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 satisfies Theorem 1 **then**
13: $\Phi = \Phi \cup (x, y)$
14: **output** Φ

Player 1 is the row player with two actions, T and B, and Player 2 is a column player with two actions, L 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:

$$A = \begin{bmatrix} 5 & 5 \\ 4 & 10 \end{bmatrix}, B = \begin{bmatrix} 5 & 2 \\ 7 & 10 \end{bmatrix}. \tag{6}$$

The support enumeration algorithm explores the mixed strategies of support size $k = 1$ and $k = 2$. Note that the support of size 1 gives the pure strategy Nash equilibria. This game has two pure strategy Nash equilibria given by ((1, 0), (1, 0)) and ((0, 1), (0, 1)). For the mixed strategies with support of size 2, a pair of mixed strategies $((x_1, x_2), (y_1, y_2))$ is possible. To find a candidate mixed strategy, we need to solve: $5x_1 + 7x_2 = 2x_1 + 10x_2$; $x_1 + x_2 = 1$ and $5y_1 + 5y_2 = 4y_1 + 10y_2$; $y_1 + y_2 = 1$. The solution to these equations is $x_1 = 1/2$, $x_2 = 1/2$ and $y_1 = 5/6$, $y_2 = 1/6$. The vector of expected payoffs to Player 1 is $Ay = (5, 5)$, and the vector of expected payoffs to Player 2 is $x^T B = (6, 6)$, thus (x, y) satisfies the best response conditions from Theorem 1. As a result, the pair of mixed strategies ((1/2, 1/2), (5/6, 1/6)) is a Nash equilibrium for the bimatrix game.

III. GPU-Based 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.

A. 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™) is a parallel computing platform that uses the graphics processing unit to increase computing performance [15].

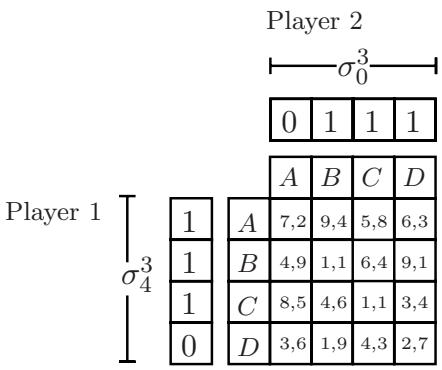

Figure 1: A 4-action bimatrix game.

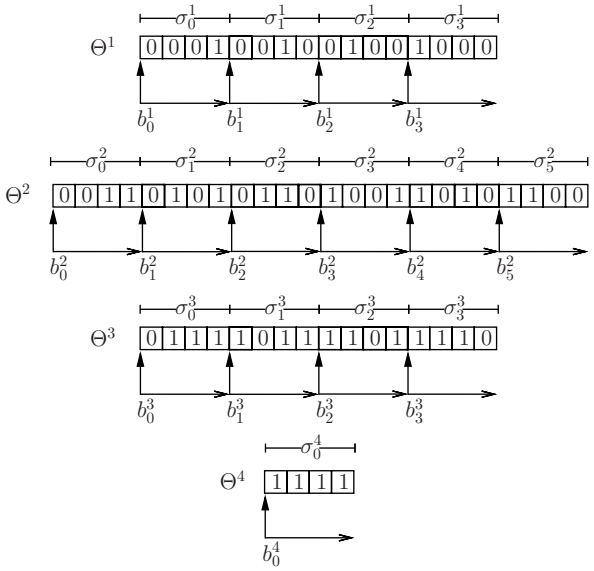

Figure 2: Support keys for a 4-action game.

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™ maintains built-in variables `threadIdx.x` and `blockIdx.x` to identify these indices. The GPU memory consists of two types, global and shared memory. Information placed on 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. Shared cache accesses are multiple times faster than global accesses. The shared cache is local to each streaming multiprocessor where thread accesses are restricted to a unique block. 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 IV as either being callable from the host machine and/or the GPU device. These function type qualifiers are *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).

B. Parallelization Method

In order to illustrate our parallelization method we will 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 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 from 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 of the 3 action combinations possible, we identify the support elements using *support keys*. Support keys are the boolean arrays represented next to the array of actions in Figure 1 which indicate those actions that are available to the player. Support keys, σ_j^i, are organized by support size i and index j which refers to the order they are created using *co-lexicographic ordering* [16]. In Figure 1, Player 1's support is identified by the support key σ_4^3, 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

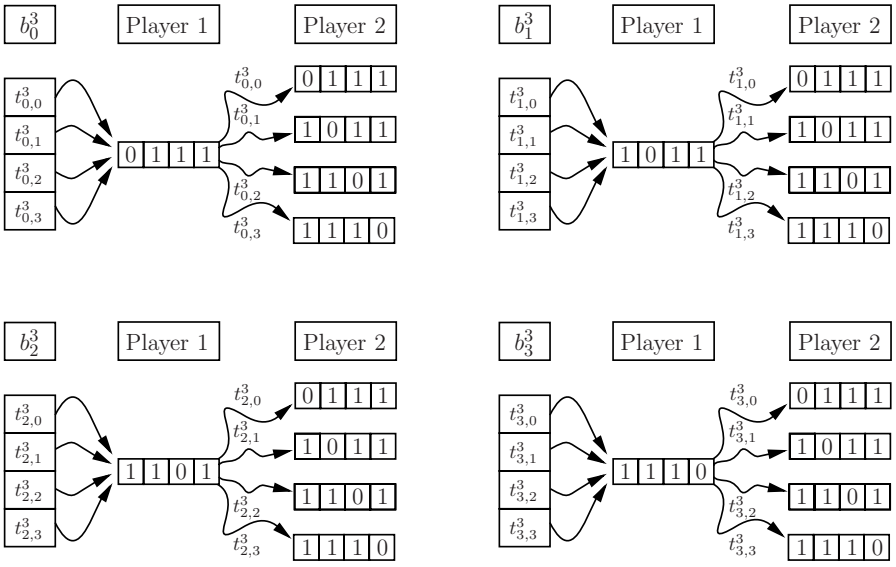

Figure 3: 4-action game with support size 3 block and thread distribution.

support. For example in Figure 1, we have four support arrays Θ^1 to Θ^4. The proposed algorithm 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 parallel methodology explored in previous implementations [8] explores these support pairs according to their support size in parallel. For example, in the 4-action bimatrix game with support of size 2 (see Figure 2), Player 1 chooses a support element identified by its support key from the six available and compares the expected payoff of using this support against Player 2 choosing every support element available. This process is repeated for Player 1 using every support element which requires 36 iterative executions. Computing the solutions in parallel by support size leads to a execution time determined by the exploration of the 36 pairs of supports (the highest number of supports that need to be explored iteratively).

Our proposed algorithm uses a different parallelization method which increases the degree of parallelism. Instead of parallelizing by support size and computing pairs of supports, our technique parallelizes the computation of the support pairs and serially computes solutions grouped by support size. Referring back to the 4-action bimatrix game example, our proposed methodology iteratively explores the 16 support pairs in parallel for support of size 1; 36 support pairs in parallel for support size of 2; 16 support pairs in parallel for support of size 3, and only a single support pair of size 4. This parallel methodology has shown significant reductions in execution time which are highlighted in Section V-C. Larger games will certainly see the benefit of this parallelization technique. For example, a 10-action bimatrix game with support of size 5 has 252 unique supports which results in 63,504 support pairs that require processing.

The proposed parallelization method is designed specifically for the GPU architecture. In order to facilitate the implementation of this method, each block of threads identifies a support element using a support key. Blocks, b_j^i, are also organized by support size i and index j. When a block identifies an element from the support, operations in the pre-kernel execution phase generate the number of threads needed and each thread then computes a support pair as shown in Figure 3. Threads, $t_{j,k}^i$ are organized by support size i, which block owns the thread j, and the index of the thread in the block k. 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. 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 will be a 2-tuple of candidate mixed strategies (x, y) that will be checked for the Nash equilibrium conditions given in Theorem 1.

IV. 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.

A. GPU-SE Function

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 payoff arrays A, B. The output variable Φ is a container which stores all Nash equilibria. The container Φ 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 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 determined by calling Mixed for each support size (Lines 7 through 10).

B. Generate Function

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 co-lexicographical arrangement algorithm detailed in [16]. 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 12). The process continues until the last possible arrangement has been generated using the co-lexicographical ordering on the support key stored in *aux* and the *flag* variable is set to 0.

C. Pure Function

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.

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

```
1: Input: Player 1 payoff, Player 2 payoff (A, B)
2: Output: Set of equilibria (Φ)
3: Φ = ∅
4: q = min(m, n)
5: Θ = Generate(1, q)
6: Φ = Pure(A, B, q, Θ)
7: for k = 2, ..., q do
8:     Θ = Generate(k, q)
9:     Φ = Φ ∪ Mixed(A, B, k, q, Θ)
10: output Φ
```

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 (Θ^k)
3: for i = 0, ..., q do
4:     aux[i] = 0
5: for i = (q − k), ..., q do
6:     aux[i] = 1
7: i = 0
8: repeat
9:     for j = 0, ..., q do
10:        Θ^k[q · i + j] = aux[j]
11:    [flag, aux] = next(q, aux)
12:    i++
13: until (flag)
14: return Θ^k
```

In Lines 8 through 10, the function copies the support key to strategy x for Player 1. In Lines 11 through 15 it generates all expected payoff values p_1 for Player 1 given the probability of selecting an action. In Line 16, the maximum expected payoff value is found and the index of that value is stored in idx for Player 1. In Lines 17 through 19, the function copies the support key to strategy y for Player 2. In Lines 20 through 24 it generates all expected payoff values p_2 for Player 2 given the probability of selecting an action. In Line 25, the maximum expected payoff value is found and the index of that value is stored in idy for Player 2. After this the function determines if both the index of Player 2's highest expected value is the response to Player 1's choice of action and the index of Player 1's highest expected value 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 Φ^p (the set of pure equilibria) which is then returned to the GPU-SE algorithm (Lines 26 through 29).

D. Mixed Function

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 substituting the constraint equations (2) and (4) into the

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, Θ)
2: **Output:** Set of pure equilibria (Φ^p)
3: $\Phi^p = \emptyset$
4: **for** $i = 0, \ldots, q$ **do**
5: $x[i]$, $y[i]$, $p_1[i]$, $p_2[i]$ = 0
6: idx, $idy = 0$
7: **for** $i = 0, \ldots, q$ **do**
8: $x[i]$ = $\Theta[q \cdot$ `blockIdx.x` + $i]$
9: **for** $i = 0, \ldots, q$ **do**
10: **for** $j = 0, \ldots, q$ **do**
11: $p_1[i]$ += $x[j] \cdot B[q \cdot j + i]$
12: idx = $\arg\max_l\{p_1[l]\}$
13: **for** $i = 0, \ldots, q$ **do**
14: $y[i]$ = $\Theta[q \cdot$ `threadIdx.x` + $i]$
15: **for** $i = 0, \ldots, q$ **do**
16: **for** $j = 0, \ldots, q$ **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: $\Phi^p = \Phi^p \cup (x, y)$
21: **return** Φ^p

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, Θ)
2: **Output:** Set of mixed equilibria (Φ^m)
3: $\Phi^m = \emptyset$
4: **for** $i = 0, \ldots, q$ **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, Θ)
9: pay = Array-ludcmp(k, pay)
10: z = Array-bcksub(z, k, pay)
11: **for** $i = 0, \ldots, q$ **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** $i = 0, \ldots, q$ **do**
18: **for** $j = 0, \ldots, q$ **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, Θ)
28: pay = Array-ludcmp(k, pay)
29: z = Array-bcksub(z, k, pay)
30: **for** $i = 0, \ldots, q$ **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** $i = 0, \ldots, q$ **do**
37: **for** $j = 0, \ldots, q$ **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: $\Phi^m = \Phi^m \cup (x, y)$
46: **return** Φ^m

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 referenced in [17], where the original versions are suited for matrix processing. We omit describing them in this paper but refer the reader to [17] for the description of the general methods.

In Line 8, z is created as a temporary solution array when solving for a player's probabilities of choosing an action. The last entry in z is set to one, satisfying constraint equations (2) and (4). 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 9). The modified payoff array is decomposed into lower and upper triangular partitions, which are stored in *pay* (Line 10). The solution is determined by back-substitution and it is stored back in z (Line 11). In Line 11, 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 11), 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 12 through 19). The expected payoff to Player 1 is then determined (Lines 20 through 24). In Line 25, the function searches for the index of the action that represents the highest expected payoff value in p_1 for Player 1. If the returned index is the response to Player 2's choice of action according to the support key (Line 25), 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 (Lines 8 through 30) are performed with respect to Player 2 (Lines 32 through 55). If both sections of the function result in $proceed$ being TRUE, then the strategies

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, Θ)
2: Output: Modified player payoff (auxpay)
3: for i = 0, ..., q do
4:     for j = 0, ..., q do
5:         auxpay[q · i + j] = 0
6: if B then
7:     for i = q − 1, ..., 0 do
8:         if Θ[q · threadIdx.x + i] = 1 then
9:             for j = 0, ..., i do
10:                if Θ[q · threadIdx.x + j] = 1 then
11:                    for l = 0, ..., i do
12:                        if Θ[q · blockIdx.x + l] = 1 then
13:                            auxpay[index] = B[q · l + j] - B[q · l + i]
14:                            index++
15: if A then
16:    for i = q − 1, ..., 0 do
17:        if Θ[q · blockIdx.x + i] = 1 then
18:            for j = 0, ..., i do
19:                if Θ[q · blockIdx.x + j] = 1 then
20:                    for l = 0, ..., i do
21:                        if Θ[q · threadIdx.x + l] = 1 then
22:                            auxpay[index] = A[q · j + l] - A[q · i + l]
23:                            index++
24: for i = 0, ..., k do
25:     auxpay[k · (k - 1) + i] = 1
26: return auxpay
```

x and y are stored as a 2-tuple in Φ^m (the set of mixed equilibria) and Φ^m is returned to the GPU-SE algorithm (Lines 56 through 59).

E. Transform Function

The Transform function manipulates the player payoff array information and adds the constraints (2) and (4) from Section II to create a system of equations represented as an array. There are two calls to Transform from the 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 in 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-if* patterns, in Lines 9 through 22 for Player 1, and in Lines 25 through 38 for Player 2, respectively. To avoid direct manipulation of the original payoff arrays, $auxpay$ is created in order to store the results from the operations in Lines 15 and 31. In Lines 40 through 42, 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.

V. Experimental Results

In this section, we perform extensive experiments to compare the GPU-based parallel support enumeration algorithm GPU-SE against the OpenMP support enumeration algorithm OMP-SE (given in Algorithm 7).

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 (Φ)
3: q = min(m, n)
4: for each thread t = 1, ..., T do in parallel
5:     t.Φ = ∅
6:     t.counter ← 0
7:     for k = 1, ..., q do
8:         for each (M_x, N_y), M_x ⊆ M, N_y ⊆ N, |M_x| = |N_y| = k
           do
9:             if (t.counter mod T) + 1 = t then
10:                Solve:
11:                    Σ_{i∈M_x} x_i B_ij = v,  ∀j ∈ N_y
12:                    Σ_{i∈M_x} x_i = 1
13:                    Σ_{j∈N_y} y_j A_ij = u,  ∀i ∈ M_x
14:                    Σ_{j∈N_y} y_j = 1
15:                if x_i, y_j ∉ ℝ⁻, ∀ i, j and x, y satisfies Theorem 1 then
16:                    t.Φ ← t.Φ ∪ (x, y)
17:            t.counter++
18:    output t.Φ
```

A. Experimental Setup

GPU-SE is executed on a 2.6 GHz Intel® Core™ 2 Quad CPU Q8400 Dell OptiPlex 780 64-bit system using 4.6 GB RAM. This system supports an NVIDIA™ GeForce GT 440 graphics processing unit with two streaming multiprocessors, each of which contains 48 cores, and uses 2.6 GB RAM.

The OpenMP implementation of the parallel support enumeration algorithm, OMP-SE, is executed on the Wayne State University Grid system [18]. Our experiment uses compute nodes on the grid, where each node has a 16 core 2.6 GHz Quad processor using 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 GPU vs CPU run times, the research literature favors the GPU. Gregg et al. [19], makes the case for determining better ways to compare the run times so as to account for memory transfers and other activities managed by the CPU for GPU processing. We have taken every opportunity to include 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 [20]. We chose 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 arguments to generate the games are: -int_payoffs -

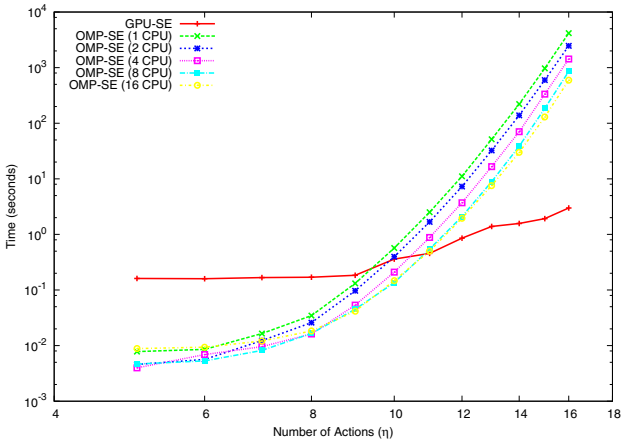

Figure 4: Average execution time vs. Number of actions

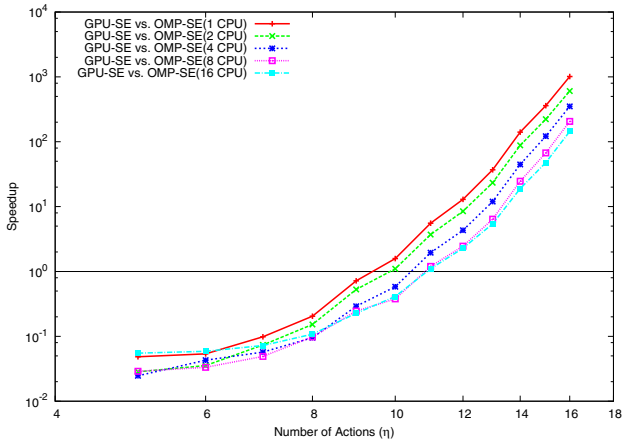

Figure 5: Average speedup vs. Number of actions

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.

B. OpenMP-based implementation

In this subsection, we present the OpenMP implementation of the support enumeration method for finding Nash equilibria in bimatrix games. 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. The threads are assigned to different processors in order to run concurrently.

We consider T threads available in the system where each thread is responsible for checking the Nash equilibrium condition in Theorem 1 for each possible support size. The OMP-SE function is given in Algorithm 7. Each thread t is responsible for generating all the supports and checking Nash equilibria conditions only for a subset of supports. In Line 6, each thread has a *counter* variable which selects a subset of supports. 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 through 14). If a Nash equilibrium is found by a thread, it is saved in the output variable Φ (Line 16).

C. Analysis of Results

We compare the performance of GPU-SE and OMP-SE. In Figure 4, we plot the average execution time for games of different sizes (given by the number of actions q). The horizontal axis represents the number of actions while the vertical axis represents run times in seconds on a log scale. For games of size $q \leq 6$, the parallel configuration for 16 processors performs worse than for a single processor. This is due to the overhead induced by the thread generation and management. The performance of GPU-SE is surpassed by all OMP-SE parallel configurations for small games. Taking into consideration the number of memory transfers, we expect a significant amount of GPU overhead to increase the run time of GPU-SE for small games. For q = 10, the GPU-SE algorithm outperforms the OMP-SE 1 and 2 CPU configurations. For q = 10, there are 184,755 pairs that require processing. For q = 11, the GPU-SE algorithm has the fastest average execution time of 0.45 (sec), where the fastest OMP-SE configurations require 0.50 (sec) with 16 processors and 0.54 (sec) with 8 processors. For games with eleven actions, q = 11, the algorithms require the processing of 705,431 pairs when solving for Nash equilibria. The GPU-based algorithm described in this paper makes it possible to execute 19,404 blocks containing various thread counts processing these pairs in parallel. The execution of this large number of blocks is possible since our GPU device supports a maximum of 65,000 blocks. The execution of threads and blocks for games with $q > 11$ shows that the GPU-SE algorithm outperforms all OMP-SE parallel configurations.

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 OMP-SE parallel execution time over the GPU-SE execution time. When the ratio is greater than 1, the GPU-SE algorithm attains a faster speedup than a corresponding OMP-SE configuration. The plot points represent the average speedup for each OMP-SE parallel configuration and GPU-SE in five games per number of actions. In line with the results from Figure 4, when q = 10, the GPU-SE speedup is approximately 1.58 and 1.10 when compared to the 1

and 2 processor OMP-SE configurations, respectively. For q = 11, the GPU-SE speedup is greater than every OMP-SE parallel configuration with the lowest speedup of 1.10 when compared against the 16 processor configuration and the highest speedup is 5.54 compared against the single processor configuration. For q = 16, the GPU-SE algorithm obtains speedups of 1013.53, 604.79, 351.46, 205.96 and 144.07 against OMP-SE parallel configurations using 1, 2, 4, 8 and 16 processors, respectively.

VI. Conclusion and Future Work

We designed and implemented a GPU-based parallel support enumeration algorithm for calculating Nash equilibria in bimatrix games. We experimented with multiple games of different sizes and configurations. Our analysis shows that our algorithm, GPU-SE, achieves lower execution times and significant speedups for large games. We designed a new parallelization method that increases the degree of parallelism when computing Nash equilibria. The GPU-SE algorithm produced 100 to 1000 faster executions versus the OpenMP 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 [5] and global Newton [6].

Acknowledgment

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

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] ——, "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.

[11] 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.

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

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

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

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

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

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

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

[19] 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.

[20] 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.