Parallelization of Support Enumeration Methods for Computing Nash Equilibria

Peter Nguyen
Advisor: Andrew Sherman
Yale University
30 April 2015

Abstract

The computation of Nash equilibria has practical applications in economics, political science, biology and numerous other fields. However, determining the set of all Nash equilibria for a given strategic game is a computationally expensive process. In order to scale back the duration of computation by at the very least a constant factor, we utilize a few platforms including multicore and GPU architectures to parallelize the computation. We present a few algorithms that take advantage of the properties of each platform and evaluate the extent to which the support enumeration method of computing Nash equilibria fits into each platform's parallelization models.

I. Introduction

Nash equilibrium is a concept drawn from the study of game theory. It is a solution concept for strategic, or normal form, games. Each player in a strategic game has a set of actions, and each player chooses a subset of these actions as a strategy. The combination of the players' strategies determines each player's payoff. A normal form is a n-dimensional matrix that defines the payoffs for the players. The indices of the matrix represent the actions the players may take in a strategy, and each player can determine her payoff based on her own strategy in combination with everyone else's strategies. It is assumed that all players are aware of the payoffs associated with each combination of strategies. A Nash equilibrium describes a state in which all players have chosen their strategies, and no players have any incentive to deviate from their own strategy, given the strategies of all other players. This solution concept can be applied in many fields outside of game theory and economics, including international relations, where one can model conflict bargaining and resolution between states, as well as political science, in order to analyze how voters interact with a given set of candidates.

Although there is much utility in identifying the set of Nash equilibria in a game, the complexity of the problem makes the process of doing so computationally expensive in both time and resources. Existing algorithms that search for and compute Nash equilibria are known to take exponential time in the worst case [1]. However, given that *every game has a Nash equilibrium*, we cannot simply classify the process of computing Nash equilibria as NP-complete, because the guarantee of a solution distinguishes this problem from standard NP-complete problems. What is known is that current methods of computing Nash equilibria are expensive; our goal is to reduce the runtime of one particular method, support enumeration, in order to make the computation of Nash equilibria more feasible in smaller cases.

OpenMP is a platform that allows one to tap into the parallel capability of multicore and multiprocessor machines. Through the annotation of C compiler directives, one is able to launch teams of threads that execute structured blocks of code in parallel. OpenMP allows the programmer to focus on the higher level aspects of a problem rather than spent excessive quantities to time deal with the lower level logic of thread initialization and deallocation. The threads in OpenMP are general purpose and support the same set of instructions, including conditional branching, at the same level of efficiency as single threaded applications.

CUDA is a platform by Nvidia that enables programmers to use Nvidia GPUs for general purpose computing. By writing specialized C code that is compiled with Nvidia's `nvcc` compiler, the programmer has the ability to utilize the massively parallel nature of GPUs to launch a number of threads that is orders of magnitude greater than most typical CPUs. CUDA programming emphasizes extreme parallelism and a single instruction, multiple thread (SIMT) model as a means to outweigh the more limited capability of GPU threads, which do not support branching at the same level of efficiency as CPU threads. With judicious thread-block allocation, gpu memory usage and avoidance of branching logic, CUDA applications have the potential to provide orders of magnitude speedup over equivalent sequential serial applications.

II. Background

How do we quantitatively formalize the concept of Nash equilibria? We begin by formally defining a strategic game between two players, call them 1 and 2. Players 1 and 2 may pick their strategies out of m and n actions, respectively. The strategies of player 1 and player 2 may consist either of single actions or probability distributions between multiple actions. The former case is referred to as a *pure strategy* and the latter case as a *mixed strategy*. Note that a pure strategy may be treated as a special case of mixed strategy, where the entirety of the probability distribution is placed on one action. We have two $m \times n$ matrices A and B whose entries represent the payoffs of player 1 and 2, respectively: when player 1 plays action i and player 2 plays action j, player 1's payoff is A_{ij} and player 2's payoff is B_{ij}. We may also refer to player 1 as the row player and player 2 as the column player, since player 1 chooses from her "row" actions and player 2 from her "column" actions.

The *support* of a strategy is defined as the set of all actions that have been assigned positive probability in a player's strategy. The size of a support is the cardinality of this set.

We next define the best response condition. We formally describe the strategies of player 1 and 2 as vectors *x* and *y* of length *m* and *n*, respectively, where the magnitude of each component x_i or y_j represents the amount of probability weight allocated to action *i* or *j* in each player's strategy. A *best response* is the strategy that maximizes a player's payoff, given the strategy of the other player. Using our earlier definitions for the payoff matrices and strategy vectors, we say that the best response for player 1, given player 2's strategy *y* is the vector *x* that maximizes x^TAy. Likewise, the best response for player 2, given player 1's strategy *x* is the vector *y* that maximizes x^TBy.

A result derived from this concept, as described in [2], which we will use in our algorithm for computing nash equilibria is as follows:

A strategy vector x is a best response to the strategy vector y if and only if for all i in player 1's set of actions M,

$$x_i > 0 \Rightarrow (Ay)_i = \max \{(Ay)_k \mid k \in M\}$$

In layman's terms, strategy vector *x* is a best response to strategy vector *y* if and only if each individual action x_i of positive probability can be played as a pure strategy that is also a best response to the vector *y*. The converse of this result, for vector y as a best response to x using the payoff matrix B, is also true.

An implication of this result is that each player must be indifferent to the actions in her mixed strategy's support, meaning all probability distributions across the support yield the same payoff, given the other player's strategy. In order to compute mixed strategy Nash equilibria, we will enforce this condition by choosing probability distributions that make other players indifferent between the actions in the other supports. Why make the other player indifferent between their strategy choices?

In a pair of mixed strategies, each action in the support of a strategy is a best response to the other mixed strategy. In order for this pair of strategies to be a NE, the other player's mixed strategy must make the first player indifferent between the actions of the support of the mixed strategy, otherwise the elements that yield less payoff would be excluded from the support and thus we would not have a NE. Given this point, one can deduce the probability distribution of the adversary's mixed strategy by solving for the distribution that leads to indifference. The adversary will do the same for the first player for the same reasoning. Afterwards, we check to see if there are any actions outside of the mixed strategies that gives strictly higher payoff than the payoff each player receives for the mixed strategies.

To conclude, the following four conditions must be satisfied in order to have a Nash equilibrium:

1. The sum of the probability distribution must be 1
2. All probabilities in the distribution must be nonnegative
3. Each player must be indifferent between the actions within their mixed strategy (meaning they each unilaterally yield the same payoff given the other player's strategy)
4. No action outside of a player's mixed strategy can give strictly better payoff than any action within the mixed strategy

III. Methodology

For the scope of this project, we consider only square, 2-player, non-degenerate strategic games. A square strategic game means that the size of set of all actions for player 1 is equal to that of player 2. A degenerate game is one where there exists a mixed strategy of size k that has a best response of more than k components. Why is this a point of concern?

Consider the following degenerate normal form game in Figure 1. Suppose the column player plays the second column as a strategy. In response, the row player may choose either the top row, bottom row, or any probability distribution of the two. There are an infinite number of Nash equilibria in this game because player 1 may distribute as much probability weight to the top row or bottom row as she wishes as long as the weights sum to 1 and are nonnegative. In degenerate games such as these, some Nash equilibria cannot be uniquely determined and enumerated.

0, 0	10, 10
0, 0	10, 10

Figure 1: An example of a degenerate strategic game

Therefore, for our purposes we deal with strategic games with a finite, enumerable set of nash equilibria. We now present the support enumeration algorithm, which we will use for the remainder of the paper.

This algorithm computes all nash equilibria (pure and mixed) for non-degenerate matrices (meaning that the size of the supports for each player in any equilibrium will be equal). The algorithm computes the equilibria by satisfying the 4 points mentioned in the background section. It is implemented as follows:

```
Def payoff matrices A and B for players 1 and 2, respectively
Def strategy vectors x and y for players 1 and 2, respectively
Def M := set of all player 1 actions
Def N := set of all player 2 actions
Def n := max size support of player 1 and player 2
For k in range [1, n]:
  For all $I \in M,\ J \in N,\ |I| = |J| = k$ :
```

Solve this system of linear equations:

$$\sum_{i \in I} x_i B_{ij} = u \text{ for } j \in J$$

$$\sum_{i \in I} x_i = 1$$

The purpose of the first set of sums is to ensure that player 2 is indifferent between the actions in her strategy's support. The last sum is to guarantee that the probability distribution of x_i sums to 1.

Next, solve an equivalent system of linear equations for the other player:

$$\sum_{j \in J} A_{ij} y_j = w \text{ for } i \in I$$

$$\sum_{j \in J} y_j = 1$$

We then check that no action in *x* or *y* has been allocated negative probability.

$$x \geq 0, y \geq 0$$

Lastly, we check that no other action outside our support gives strictly better payoff than any action in our support, as explained in the result from [2] in the background section.

If the linear systems are not singular and all conditions pass, return *I, J* as a Nash equilibrium.

All further references to the variables n, M, N, I, or J in this paper refer to this pseudocode.

In order to generate sample strategic games, we used the GAMUT java suite[3]. We tested our algorithms for square games from size n = 10 to n = 16. Initially, we used the Gambit[4] application to check the correctness of our initial algorithms for computing all NE. We wrote a variety of python scripts in order to quickly estimate maximum memory usage relative to the resource limits of our hardware, to prototype the serial implementation of the algorithm and to massage the GAMUT output to enable easy parsing for the C code. The codes that were benchmarked were all written in C, with compiler directives and annotations for OpenMP and with platform specific functions and syntax for CUDA.

In order to check for correctness of the NE, a function "printSolution" is called that takes both players' solution vectors and prints them to stdout. No NE is stored in memory.

IV. Implementation

In all implementations of our codes, we use a function called "kSubsetsIt" that iterates through all combinations of a a given support size in lexicographic order.

For our serial implementation, we used various libraries to handle the linear algebra components of the algorithm. We utilized the Intel Math Kernel Library[5] (MKL) for the linear system solver and the matrix multiplication routines, which are based off of the LAPACK and BLAS collection of utility libraries.

For our multicore approach, we utilized the OpenMP platform in order to simplify the logic associated with initializing and deallocating threads. Our parallelization strategy involved having all threads enumerate through all pairs of supports in the same order, keeping a counter in the process. However, each thread only processes a support pair whenever the counter mod the total number of threads is equal to the thread ID, which is assigned upon initialization. We utilized the same libraries for the linear algebra utilities, as the MKL library functions are thread safe. We specify the number of threads that is run at command line invocation.

We implement our CUDA version of the algorithm as a variation on the OpenMP implementation. For each support size, our CUDA kernel allocates a number of blocks equal to the number of supports at that size. Each block is associated with one of player 1's supports, identified by blockIdx.x. We then optionally set the number of threads per block through the command line, or else the program defaults to 512 threads per block. Each thread within a block is responsible for the same support for player 1, but for multiple, nonoverlapping supports for player 2. In order to fully cover all support pairs, each thread enumerates a number of cycles in order to process all supports of player 2 as a block. Since the array of all supports is passed in as an array, we may use threadIdx.x as an offset to assign player 2 supports to threads. Once a cycle is done, all threads in the block jump a distance equal to the number of threads in the block, and process the supports again based on the offset based on threadIdx.x, until there are no more supports to process. In the two earlier implementations, the threads generated and enumerated through support pairs one at a time, as they were needed. Therefore, the number of support pairs that were allocated into arrays was never more than the number of threads that were executing. We take the opposite approach in the CUDA implementation by allocating the space to record *all* of the supports of a given size, a priori, on the host, and then transfer this array over to the GPU. Then, we process as many support as possible in parallel, taking as many cycles as necessary to completely process all pairs.

When implementing the CUDA approach to the algorithm, the following 5 considerations were at play:

i) Excessive allocation of blocks

In the Grosu scheme[6] for computing all NE, each block is responsible for a support from player 1's strategy space, and then each thread within the block is responsible for handling the pair consisting of player 1's support and 1 support from player 2's strategy space. Thus, each thread in the grid is responsible for one pair of strategies. However, if the number of supports exceeds 1024, the Grosu scheme suggests allocating additional blocks to generate additional threads that will handle the remaining supports. For instance, consider the case when we are processing a square matrix of size 16 x 16 and we are now handling supports of size 8. The number of supports for each player would be ${}_{16}C_{8}$, equal to 12870. Since the thread limit per block is 1024, we cannot process all of the supports of player 2 within one block using this scheme, given one support of player 1.

Our approach avoids allocating yet even more blocks, because there are only so many blocks that can run concurrently on the a GPU at a time. The GPU used in this paper has a Compute Capability of 2.0, so it can support no more than 48 concurrent blocks[7], 8 per streaming multiprocessor(SM) and 16 SMs. We instead rely on the existing threads in the blocks to handle a set of player 2's supports, rather than a single one, processing them over multiple cycles. We avoid allocating blocks too much in order to reduce the frequency of data transfers between blocks moving on and off the SMs.

ii) Memory limits & excessive thread allocation

The Grosu scheme does not seem to take into account the limitations of the GPU, particularly in terms of the limited memory allocatable. There exists only 6GB for the M2090 GPU, and only 48KB allocatable shared memory per block. Given that each thread must at the very least compute the solution of a linear system before determining whether or not to continue execution, there would be insufficient memory to handle the aggregate size of the stack frames of all threads.

Recall the example from i) with the 16 x 16 matrix handling supports of size 8. If we generated a one thread for each support pair, and process one linear system of floating point entries as specified in the algorithm, the memory overhead of running all threads would be 12870^2 * (9 x 9 matrix linear system) * sizeof(float) > 40GB of storage required > 6GB available memory. In order to conserve memory, we instead make each thread responsible for a set of player 2's supports and process them over multiple cycles.

iii) Warp divergence

The GPU issues the same instruction to groups of 32 threads called "warps" that execute the instruction in parallel. Problems still arise when conditional branches are encountered. If a warp enters a conditional branch and some subset of the warp takes a different branch from the other subset, the GPU handles this by executing the branches sequentially. As a result, branches usually tend to degrade GPU performance if the branch separates the branching paths of threads within a warp. If we recall the 4 conditions necessary to find a NE, we will find that we cannot avoid using conditional statements in the GPU implementation of the algorithm.

iv) Occupancy

1. Recalling that the GPU issues instructions at warp level granularity, we are cognizant of this fact by allocating 512 threads per block (divisible by 32) by default in order to fully utilize the issuing of instructions by the GPU. If we chose a value not divisible by 32, some warps would contain inactive threads and would not be affected by the instruction, wasting an issuance on "dead" threads.
2. Although choosing a block size that is divisible by 32 may improve performance, we also consider the overall size of each block as a factor in performance. Recall that each SM can run up to 8 blocks, and each SM can handle up to 1536 threads. We wish to have as many threads in flight as possible. The SMs bounds us both in terms of number of blocks and in number of threads, therefore we must have a large enough block size such that the max number of blocks is not saturated before saturating the max number of threads.

v) Linear System Solver on the GPU

For the CUDA implementation, there were no obviously available libraries that implemented general linear system solvers or matrix vector multiplication on a *per thread basis*: rather than parallelizing the action of solving a linear system or performing a matrix multiplication using the GPU, we have each thread in flight solve its own linear system. We implement our general linear system solver using Crout's method of in-place LU decomposition and back substitution, as described in [8]. There are two competing interests at play in this approach.

Excessive conditional branching and warp divergence

On the one hand, we are given no guarantees regarding any properties of the linear system that we solve. For this reason, we must solve the linear system as a general matrix. That being said, in order to ensure stability of the system and solution, it is inevitable that we use pivoting. In addition, during the process of our LU decomposition, we have to check whether the matrix we are decomposing is singular, and exit execution if it is so. This imposes costs in terms of delayed and sequential execution between threads within warps, and thus unfulfilled occupancy within the GPU.

Lack of effectiveness in using a parallel linear system solver

1. The complexity and size of computing Nash equilibria is not due to an extremely large single problem, but due to very many small problems that all need to be handled.
2. Each of these small problems involves solving a linear system; to use a parallel linear system solver would be ineffective, because the overhead associated with data transfer from CPU to GPU and back would far outweigh the speedup in solving the small linear systems. We could also assign a small set of threads for each linear system, but as the number of actions increases, the number of support pairs increases at an exponential rate.
3. Given that we are using Compute Capability 2.x, we have no access to dynamic parallelism, so we would not be able to call a nested kernel to parallelize solving a single system at a time. Even if we had the capability, this disruption in parallel computation flow may outweigh the benefits of such a tool.

V. Findings

All of the codes were executed on the nodes in the Yale High Performance Computing cluster. The CPUs used for the codes were Intel Xeon E5-2650 running at 2.00 GHz. The GPU used was the Nvidia Tesla M2090 GPU, with up to 6GB device memory. In all cases, the codes executed with 46GB of main memory.

All GPU timing used blocks that contained 512 threads each, and the number of blocks initialized was equal to the number of supports of a given size. All timing information was gathered using either C timing function implemented for the Yale CPSC424 course or the unix "time" command, using the real time result from that call.

As one can see, the efficiency of the OMP-8 and OMP-16 implementations is not initially very good, but as the problem size grows sufficiently large, we get a good efficiency, from less than 50% at n = 10 for OMP-16 to over 90% by the time we reach n = 16.

There seems to be a geometric correlation between the increase in problem size and the resulting increase in computation time; there seems to be a change of a factor approximately 4 as we continue to increase the size of n. This exponential increase makes sense, given that we are inspecting the entire search space (all subsets) of n on every trial. Given that the number of subsets in a given set of size n is 2^n, it does not come as a surprise that the increase in computation time as n increases is geometric in nature.

Recall that for our algorithm, we evaluate every support pair of equal size, for support pairs of size 1 up to n. We may therefore represent the total number of pairs evaluated as a summation that resolves as follows:

$$\sum_{k=1}^{n}\binom{n}{k}^2 = \frac{4^n\,\Gamma\left(n+\frac{1}{2}\right)}{\sqrt{\pi}\;\Gamma(n+1)} - 1$$

The exponential term helps to explain the approximate factor of 4 increase in our computation timing.

There are some interesting points to observe involving the respective speedups between the OpenMP implementations and the GPU implementation. We can observe that the relative speedup between the GPU and OMP-16 implementations goes as far as approximately 4 : 1. The speedup from serial to GPU goes up to approximately 60. While an improvement in time as an absolute may be considered good, a speedup of merely 60 may be a testament to performance degradation. As expressed in the concerns in the implementation section, there were numerous aspects regarding the nature of the support enumeration algorithm that did not seem to fit the GPU model. Particularly, the extensive use of conditional branching in an iteration of the algorithm was a significant area of concern, as it would be a major culprit in slowdowns associated with warp divergence. Sure enough, the speedup for GPU was not able to exceed or even reach reach 2 orders of magnitude.

Upon comparing this paper's results with that of Grosu, our GPU speedups were more modest. While Grosu was able to attain a speedup of slightly over 100 by n = 16, in our case, we reached a speedup of slightly less than 56. Likewise, while Grosu had a speedup of 15.36 between the GPU execution and the OMP-16 execution, for our experimental data we reached slightly less than 4.

There still seems to be inefficiencies in our GPU code, at least for those inherent with the support enumeration algorithm. From our discussion on implementation, there seems to be fundamental differences between the programming approach of the GPU programming model and the support enumeration algorithm.

VI. Future Work and Conclusion

The support enumeration, while straightforward to implement, is not the ideal algorithm to parallelize on GPU architectures, but scales rather well on multiCPU thread platforms. Although GPUs did indeed provide absolute speedup over the OMP implementation, the efficiency of the GPU implementation comes into question, especially when putting into consideration the number of SMs and cores available to the hardware. Further investigation and research can be done involving accelerating the computation of linear system solutions. A paper by Baboulin, Dongarra, Herrmann and Tomov[9] presents a mechanism for solving linear systems without pivoting by first performing some transformations on the input matrix. These Random Butterfly Transformations can ameliorate the shortcomings of GPUs by removing some conditional branching, at least in the linear system solving phase of our algorithm.

TABLE 1
AVERAGE COMPUTATION TIME (seconds)

n	Serial	OMP-2	OMP-4	OMP-8	OMP-16	GPU
10	0.465	0.236	0.120	0.069	0.062	0.243
11	1.579	0.879	0.453	0.237	0.159	0.187
12	6.688	3.564	1.816	0.938	0.539	0.219
13	28.079	14.668	7.682	3.826	2.089	0.531
14	111.240	58.954	29.117	15.312	8.162	1.874
15	462.019	235.660	119.718	60.481	32.532	7.953
16	1908.015	944.333	491.653	247.198	128.771	34.246

TABLE 2
SPEEDUP RELATIVE TO SERIAL IMPLEMENTATION

n	OMP-2	OMP-4	OMP-8	OMP-16	GPU
10	1.97	3.87	6.74	7.55	1.91
11	1.80	3.49	6.66	9.93	8.46
12	1.88	3.68	7.13	12.40	30.54
13	1.91	3.65	7.34	13.44	52.90
14	1.89	3.82	7.26	13.63	59.37
15	1.96	3.86	7.64	14.20	58.09
16	2.02	3.88	7.72	14.82	55.72

References
[1] C. Daskalakis, P. Goldberg and C. Papadimitriou, 'The complexity of computing a Nash equilibrium', Commun. ACM, vol. 52, no. 2, p. 89, 2009.
[2] Nisan, Algorithmic game theory. Cambridge: Cambridge University Press, 2007.
[3] Gamut.stanford.edu, 'GAMUT: Game-Theoretic Algorithms Evaluation Suite'. [Online]. Available: http://gamut.stanford.edu/. [Accessed: 26- Feb- 2015].
[4] Gambit.sourceforge.net, 'Gambit: Software Tools for Game Theory', 2015. [Online]. Available: http://gambit.sourceforge.net/. [Accessed: 30- Mar- 2015].

[5] Software.intel.com, 'Intel® Math Kernel Library Reference Manual', 2015. [Online]. Available: https://software.intel.com/sites/products/documentation/doclib/iss/2013/mkl/mklman/index.htm. [Accessed: 02- May- 2015].
[6] S. Rampersaud, L. Mashayekhy and D. Grosu, "Computing Nash Equilibria in Bimatrix Games: GPU-based Parallel Support Enumeration" in IEEE Transactions on Parallel and Distributed Systems, Vol. 25, No. 12, December 2014, pp. 3111-3123.
[7] Docs.nvidia.com, 'Programming Guide :: CUDA Toolkit Documentation', 2015. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz3YvL3BusX. [Accessed: 19- Apr- 2015].
[8] W. Press, *Numerical recipes in C*. Cambridge [u.a.]: Cambridge Univ. Press, 1992.
[9] Baboulin, Dongarra, Herrmann and Tomov, 'Accelerating Linear System Solutions Using Randomization Techniques', 2011.