8-2022

A GPU accelerated Genetic Algorithm for the Construction of Hadamard Matrices

Andras Balogh
The University of Texas Rio Grande Valley

Raven Ruiz
The University of Texas Rio Grande Valley

A GPU accelerated Genetic Algorithm for the Construction of Hadamard Matrices

Andras Balogh[1*] and Raven Ruiz[1]

[1]School of Mathematical and Statistical Sciences, The University of Texas Rio Grande Valley, 1201 West University Drive, Edinburg, 78539, TX, USA.

*Corresponding author(s). E-mail(s): andras.balogh@gutrgv.edu;
Contributing authors: ruiziraven@gmail.com;

Abstract

We use a genetic algorithm to construct Hadamard Matrices. The initial population of random matrices is generated to have a balanced number of $+\mathbf{1}$ and $-\mathbf{1}$ entries in each column except the first column with all $+\mathbf{1}$. Several fitness functions are implemented in order to find the most effective one. The crossover process creates offspring matrix population by exchanging columns of the parent matrix population. The mutation process flips $+\mathbf{1}$ and $-\mathbf{1}$ entry pairs in random columns. The use of CuPy library in Python on graphics processing units enables us to handle populations of thousands of matrices and matrix operations in parallel.

Keywords: Genetic algorithm, Hadamard matrix, GPU, Python, CuPy, CUDA

1 Introduction

Hadamard matrices are square matrices with $+1$ and -1 entries where the columns are mutually orthogonal. According to the Hadamard Conjecture [1], Hadamard matrix of size $m \times m$ (order m) exists when $m = 1$, $m = 2$, and when $m = 4k$ for arbitrary k positive integer. Deterministic algorithms are known for creating Hadamard matrices of order 2^k and only a few other special cases of order $4k$. In [2], Goethals and Siedel has shown that there exist skew Hadamard matrices of order $m = 36$ and $m = 52$. The Goethals–Siedel

method was implemented to construct 32 in–equivalent Hadamard matrices of order $m = 404$ in [3]. A classification of Hadamard matrices using types of quadruples of rows with two distinct values were done in [4]. In [5] Seberry and Yamada highlights important Hadamard matrix theorems, use several methods to construct Hadamard matrices, display results, and analyze their findings from each method. More recently Suksmono in [6] applied a Simulated Annealing Algorithm and in [7] a Williamson based quantum computing method to construct Hadamard matrices. The reader is referred to [8] for a recent list of known constructions results for Hadamard matrices of different orders.

In this work we develop a genetic algorithm for the construction of Hadamard matrices. The parallel numerical code uses CUDA GPUs to accelerate the computations.

In Section 2 we briefly describe the difference between traditional CPU computing and our approach of using GPUs to accelerate computations.

In Section 3 we describe how our parallel implementation of a Genetic Algorithm works for finding Hadamard matrices. In Section 4 we discuss the computational results found by the use of our Genetic Algorithm. Finally, the paper concludes with a summary in Section 5, and the numerical code in Section 6.

2 Computing on Graphics Processing Units

Stochastic algorithms and matrix calculations involve large number of calculations. A CPU (Central Processing Unit) does calculations mostly in serial way, which can take a large amount of time if working with many large matrices. GPUs (Graphics Processing Units) were developed for fast graphics rendering by calculating what to do and when with the millions of pixels in a computer screen. Since a GPU can do thousands of calculations simultaneously (in parallel), the time working with large number of matrices can be reduced significantly. The use of GPUs allows us to do parallel calculations more effectively than using CPUs. Most of our calculations have been performed on an EVGA GeForce RTX 3080 XC3 black [9], having 8704 CUDA cores, 10 GB memory, and CUDA capability 8.6. With the use of GPUs our genetic algorithm was able to handle for example a population of $40,000$ matrices of size 20×20.

NumPy [10] is a popular library for the Python programming language. It supports overloaded mathematical functions operation on multidimensional arrays. For example, if A is a 100×100 matrix defined as `A=numpy.random.rand(100,100)`, then `B=numpy.sin(A)` produces a matrix containing the sine of the elements of matrix A. The calculations of the $10,000$ elements are done one-by-one, in serial fashion on the CPU. For this project, we are using the CuPy [11], Python library accelerated with NVIDIA CUDA [12] for GPUs. It was created specifically to be highly compatible with NumPy. For example modifying the previously mentioned NumPy matrix example

to `A=cupy.random.rand(100,100)` and `B=cupy.sin(A)`, the calculations are distributed on the thousands of CUDA threads to be done in parallel.

Using the CuPy library and in general CUDA on the GPU involves the following steps. It is important to note that the CPU controls the GPU too.

1. Data is copied from CPU memory to GPU memory.
2. CPU initiates the calculations on the GPU.
3. GPU executes the calculations.
4. Results are copied back from GPU to CPU.

Each of these steps require time. The speed up of the GPU execution has to be significant in order to balance the extra time required by copying the data back and forth between CPU and GPU.

We include here a simple speed comparison of using Numpy (serial calculations) vs. CuPy (parallel calculations). Since our actual code with Genetic Algorithm includes raw kernels, we never wrote a serial version of it, but due to the complicated nature of the calculations with large multi-dimensional arrays, we expect the parallel computations to be thousands of times faster than the serial code. The example creates a list of $N = 10,000$ random matrices of size 100×100, and then multiplies each with their transform, repeating the calculation M times.

NumPy version:

```
import numpy as np
m=100
N=10**4
Pop = np.random.rand(N, m, m)
for i in range(M):
    F=np.matmul(np.transpose(Pop,axes=(0,2,1)),Pop)
```

CuPy version:

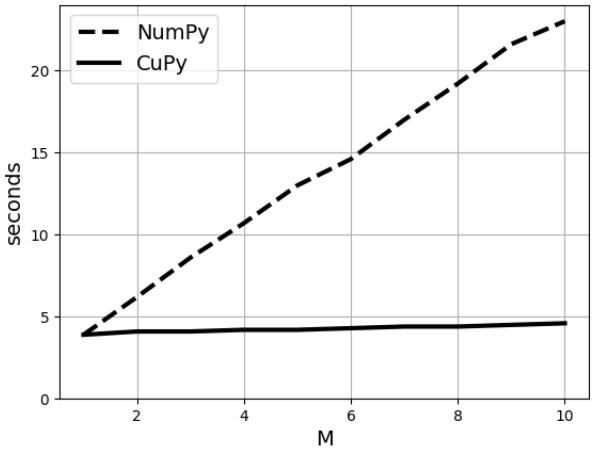
```
import cupy as cp
m=100
N=10**4
Pop=cp.random.rand(N, m, m)
for i in range(M):
    F=cp.matmul(cp.transpose(Pop,axes=(0,2,1)),Pop)
```

Note the identical syntax other than the use of "cp" for CuPy vs. "np" for NumPy. The time required for the calculations are summarized in Table 1 and in Figure 1. For $M = 1$, when the matrix calculations are done only once, the CuPy and NumPy calculations take the same 3.9 seconds. Although the GPU does the calculation faster than the CPU, the time it takes to load the GPU slows the completion of the code down. For repeated calculations of the batched matrix multiplications the Numpy code execution time increases by about 2.12 seconds for each additional loop steps taken. The CuPy calculation

Table 1: CuPy vs. NumPy Execution Times (seconds)

M	NumPy	CuPy
1	3.9	3.862
2	6.2	3.994
3	8.6	4.038
4	10.7	4.095
5	13.0	4.175
6	14.6	4.252
7	17.0	4.287
8	19.2	4.330
9	21.6	4.394
10	23.0	4.454
⋮	⋮	⋮
100	218.6	9.916

Fig. 1: Cupy vs. Numpy Execution Times (seconds)



only increases by about 0.07 seconds for each loop steps. This means that the CuPy operation is 30 times faster than the NumPy operation after ignoring the initial data copy between the CPU and GPU.

It is also possible to combine CuPy code with CUDA kernel function that use C++ syntax. We use several so-called "raw kernels" when we need more complicated operations than simple matrix-vector ones. This requires to know how threads are arranged in grids of blocks, as explained in the next section.

3 Genetic Algorithm and Hadamard Matrices

Genetic algorithms are a search heuristic that was inspired by the Natural Selection process [13]. Natural Selection is a process where individuals adapt and change based on the environment and other variables. Each individual in the population is unique, meaning that each individual has different traits.

Some individuals may have better traits than others, this allows those with better traits to live longer and reproduce. Those superior traits are then passed down to the next generation, with some variation, this is known as evolution. Evolution is a key component of genetic algorithms, it allows the algorithms to search for the optimal solution of several problems in mathematics. We will continue to discuss evolution in the following section.

Genetic algorithms use Darwin's evolution theory of natural selection as a bases to search for optimal solutions. There are three main points behind Darwin's theory of evolution that are given as follows

1. *Inheritance*: Each individual inherits traits that were passed down from their parents. The traits that are most likely to be passed down are those that will improve chances of survival.
2. *Variation*: Each individual in a population will have variation that is unique to them, even those that are related to each other.
3. *Fitness and Selection*: The most fit individuals are more likely to survive, the surviving individuals are then able to reproduce and pass on their genes to future generations.

Ultimately, Darwin's evolution theory suggests that the individuals with the best traits will survive and maintain the population with offspring of their own. The offspring will most likely be better equipped for survival and each generation there after will become more adaptive. This leads to a genetic algorithm that will be used throughout this paper.

1. *Generate Initial Population*: Create set of individuals in a population.
2. *Compute Fitness*: A numerical measure of how close an individual is to becoming 'fit'.
3. *REPEAT*
   (a) *Selection*: Select the most fit individuals which is based on the fitness.
   (b) *Crossover*: The selected individuals become parents, the parents are paired and create a pair of offspring that inherit traits from the parents.
   (c) *Mutation*: Each offspring will have some form of variation to them.
   (d) *Compute Fitness*: A numerical measure of how close an individual is to becoming 'fit'.
4. *UNTIL Population has converged*: An individual has met the required fitness.

The fundamental theorem of genetic algorithms [14] is also know as Holland's schema theorem [15], proposed by John Holland in the 1970's. While the mathematics is quite involved, we will focus on the basic understanding of the theorem. The theorem suggests that an individual will prevail if it has an above average fitness.

In Subsection 3.1, we detail how an initial population of matrices is generated. For this process, two methods were implemented and we discuss their differences. In Subsection 3.2, we use a fitness function to compute the fitness

of each matrix in a population. These fitness values are used as a measurement to see how close a matrix is to becoming an Hadamard matrix. There are four fitness functions that were implemented and we discuss their differences. In Subsection 3.3, we use the fitness values of each matrix to select parent matrices from a population. In Subsection 3.4, we use a process that creates offspring matrices from selected parent matrices. In Subsection 3.5, we discuss how each offspring matrix is given some variation or mutation. Three methods were implemented that achieve this.

3.1 Population

The population consists of $4N$ matrices, each of them of size $m \times m = 4k \times 4k$, where $k, N \in \mathbb{N}$. They are arranged in the form of a 3-dimensional array:

$$\begin{aligned} \text{Population} &= [Q_1, \ldots, Q_{4N}] && (1) \\ &= [P_1, \ldots, P_N, P_{N+1}, \ldots, P_{2N}, O_1, \ldots, O_N, O_{N+1}, \ldots, O_{2N}], && (2) \end{aligned}$$

with $P_1, \ldots, P_{2N}$ denoting the parent matrices and $O_1, \ldots, O_{2N}$ denoting the offspring matrices. During the crossover process matrices $P_1, \ldots, P_N$ are paired up with matrices $P_{N+1}, \ldots, P_{2N}$ to create offspring matrices $O_1, \ldots, O_{2N}$.

At the beginning of the process the population of matrices are initialized to have a balanced number of $+1$ and -1 entries in each column except for the first column that consists of all $+1$ entries. This way columns $2 - -m$ are automatically orthogonal to the first column, and the first columns will not have to be changed during the process. In order to achieve a random order of the $+1$ and -1 entries in columns $2 - m$ of each matrix, first all matrices are initialized to have all $+1$ entries, then the upper half is filled with -1 (see code lines 98–99). Then we implement a raw (CUDA) kernel function called `Shuffle_Column` that uses the Fisher-Yates shuffle algorithm [16, 17] to shuffle the ± 1 entries in columns $2 - m$ of each matrix. While the shuffling in each column is done in serial, the columns are handled simultaneously in parallel. See code lines 7–31 for the `Shuffle_Column` raw kernel, code line 100 for generating a random seed, and code line 101 for calling the raw kernel.

3.2 Fitness Function

In the natural selection process, the most fit members of the population will most likely to survive and reproduce. For our genetic algorithm, the fitness function measures how close a matrix is to being an Hadamard matrix. For an $m \times m$ matrix Q with ± 1 entries the product matrix $Q^T Q$ has diagonal entries m and this product matrix is diagonal if and only if Q is Hadamard matrix. Also, the closer the off-diagonal entries are to zero, the closer the corresponding matrix columns are to orthogonal. This leads to the fitness function

$$F_1 = \sum_{i,j=1}^{m} \left| \left[Q^T Q \right]_{ij} \right| - m^2. \qquad (3)$$

Instead of fitness function (3), we chose the slightly more computationally efficient fitness function

$$f = \text{number of nonzero entries } \left(Q^T Q\right) - m, \tag{4}$$

which is also a nonnegative function with a zero minimum value attained for Hadamard matrices only. The fitness values are calculated for each matrix in the population resulting in an array

$$F = [f_1, \ldots, f_N, f_{N+1}, \ldots, f_{2N}, f_{2N+1}, \ldots, f_{3N}, f_{3N+1}, \ldots, f_{4N}], \tag{5}$$

where $f_1, \ldots, f_{2N}$ and $f_{2N+1}, \ldots, f_{4N}$ refers to the fitness of $P_1, \ldots, P_{2N}$ and $O_1, \ldots, O_{2N}$ in the population respectively. The calculation of fitness values (4) implemented using CuPy can be found on code lines 104 and 126.

3.3 Selection

The purpose of the selection process is to select matrices that are more likely to become Hadamard matrices based on their fitness value in (5). The selected matrices will take the parent position (first half) of the population in (1) in random order for future use while the rest of the matrices are discarded. The compact CuPy implementation of the selection process is on code line 113.

The literature often uses selection process with probabilities proportional to how fit an individual is. In our case we want to minimize the fitness function and hence the probabilities would have to be inverse proportional to the fitness values. We were not able to find probabilistic selection process that would be more effective than the the above deterministic one.

3.4 Crossover

The purpose of the crossover is to create a pair of offspring matrices from a pair of parent matrices. The parent matrices $[P_1, \ldots, P_N, P_{N+1}, \ldots, P_{2N}]$ are paired up as (P_1, P_{N+1}), (P_2, P_{N+2}), $\ldots$, (P_N, P_{2N}). For each pair of matrices a random column index $1 < c_{cop} < m$ is generated as crossover point. This crossover point splits the columns of parent matrices in each pair into two parts: columns $1, \ldots, c_{cop}$ and columns $c_{cop}+1, \ldots, m$. Each part in one matrix is matched with the adjacent part from the other matrix to create a pair of offspring matrices. Figure 2 shows an example with a pair of 4×4 parent matrices (P_1, P_{N+1}), with a crossover point `c_cop = 3` and with resulting offspring matrices (O_1, O_{N+1}).

The calling of the crossover process is from code lines 116-117 with the raw CUDA kernel function `Crossover(grids, blocks, (Pop, c_cop, m, N))` on code lines 36-54. The `Crossover` function creates $2N$ offspring matrices using $2N$ parent matrices and it returns the size $4N$ array `Pop` of matrices such that

$$\texttt{Pop} = [P_1, \ldots, P_N, P_{N+1}, \ldots, P_{2N}, O_1, \ldots, O_N, O_{N+1}, \ldots, O_{2N}]. \tag{6}$$

Fig. 2: Crossover

$$P_1 = \left(\begin{array}{ccc|c} - & + & + & + \\ - & + & - & - \\ + & - & + & - \\ + & - & - & + \end{array}\right), \quad P_{N+1} = \left(\begin{array}{ccc|c} - & + & + & + \\ + & - & + & + \\ - & - & - & - \\ + & + & - & - \end{array}\right)$$

$$O_1 = \left(\begin{array}{ccc|c} - & + & + & + \\ - & + & - & + \\ + & - & + & - \\ + & - & - & - \end{array}\right), \quad O_{N+1} = \left(\begin{array}{ccc|c} - & + & + & + \\ + & - & + & - \\ - & - & - & - \\ + & + & - & + \end{array}\right)$$

The crossover process is parallelized for the pairs of parent matrices indexed by k and for their rows and columns indexed by i and j respectively with the array of matrices flattened. The (i, j) entry of the k^{th} matrix is referenced as `Q[m*m*k+m*i+j]`.

3.5 Mutation

In this next section, we discuss the mutation process in the matrix population. The purpose of the mutation process is for each offspring matrix to have some kind of variation to them. The mutation of the matrices is done by switching one or more pairs of ± 1 entries while maintaining the balance of ± 1 entries in each column. We implemented three methods to achieve variation among offspring matrices:

1. Flipping the same pair of ± 1 entries in each offspring matrix, by using the same random column index and same two random row indices using CuPy code

```
colindx = np.random.randint(1,m)
rowindx1 = np.random.randint(m)
rowindx2 = np.random.randint(m)
```

2. Flipping a different pair of ± 1 entries in different offspring matrices by randomly selecting columns and pairs of rows indices for each matrix using CuPy code

```
colindx = cp.random.random_integers(1,m-1,size=(2*N))
rowindx1 = cp.random.random_integers(0,m-1,size=(2*N))
rowindx2 = cp.random.random_integers(0,m-1,size=(2*N))
```

3. Flipping several pairs of ± 1 entries in several (NC) random columns and several (NR) random pairs of rows in each offspring matrix.

```
NC=3 #NC should be less than m
NR=2 #NR should be less than m/2
colindx = cp.random.random_integers(1,m-1,size=(2*N,NC))
rowindx1 = cp.random.random_integers(0,m-1,size=(2*N,NR))
rowindx2 = cp.random.random_integers(0,m-1,size=(2*N,NR))
```

The third case provides the most variation, and it coincides with case two for $NC = NR = 1$. In all three cases no switch is done if the numbers in the randomly selected pairs have the same sign. The first column has all $+1$ entries, and for that reason the first column is never chosen for mutation. The third form of mutations is coded on lines 120-123 with the actual CUDA raw kernel function on lines 58-83.

To demonstrate how the third mutation process works lets consider an example. Let $m = 8$ and $N = 2$ such that the we have a population of 8 matrices that have sizes 8×8 with a balanced number of ± 1 entries in each column. This implies there are 4 offspring matrices that will be mutated. Let $NC = 3$ be the number of columns mutated and $NR = 2$ be the number of row pairs mutated such that we have the randomly generated column and row indices given by

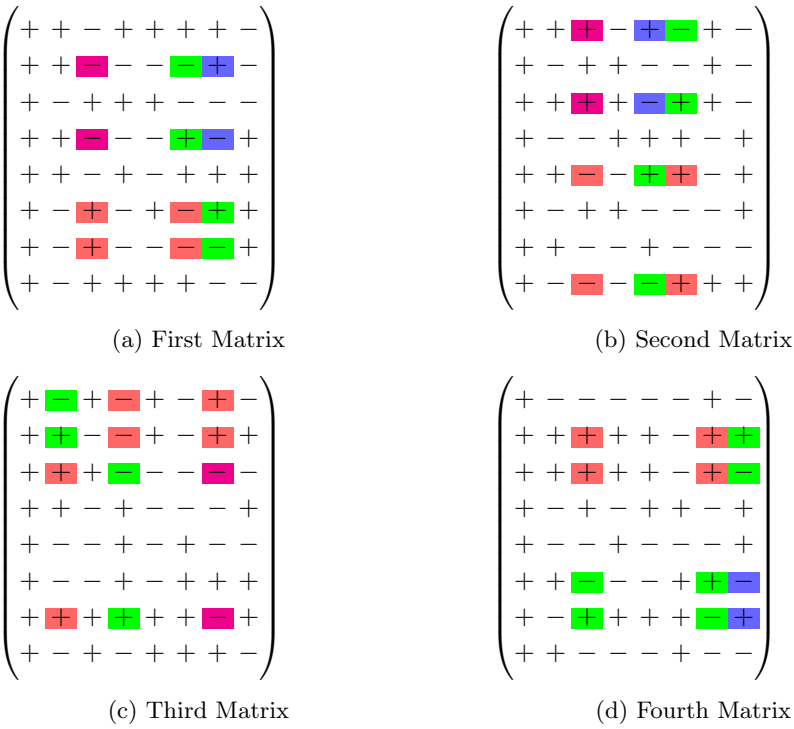
$$\texttt{rowindx1} = \begin{bmatrix} 6 & 5 \\ 4 & 2 \\ 0 & 6 \\ 1 & 6 \end{bmatrix}, \quad \texttt{rowindx2} = \begin{bmatrix} 3 & 1 \\ 7 & 0 \\ 1 & 2 \\ 2 & 5 \end{bmatrix}, \quad \texttt{colindx} = \begin{bmatrix} 6 & 5 & 2 \\ 5 & 2 & 4 \\ 3 & 1 & 6 \\ 2 & 7 & 6 \end{bmatrix} \tag{7}$$

The mutation of four matrices are given in Figure 3 such that

- In part (a), the indices are `rowindx1` $= [6,\ 5]$, `rowindx2` $= [3,\ 1]$, and `colindx` $= [6,\ 5,\ 2]$.
- In part (b), the indicies are `rowindx1` $= [4,\ 2]$, `rowindx2` $= [7,\ 0]$, and `colindx` $= [5,\ 2,\ 4]$.
- In part (c), the indicies are `rowindx1` $= [0,\ 6]$, `rowindx2` $= [1,\ 2]$, and `colindx` $= [3,\ 1,\ 6]$.
- In part (d), the indicies are `rowindx1` $= [1,\ 6]$, `rowindx2` $= [2,\ 5]$, and `colindx` $= [2,\ 7,\ 6]$.

In each column, a successful mutation is represented in green and blue (if there is more than one successful), an unsuccessful mutation is represented in red and magenta (if there is more than one unsuccessful). We can see in Figure 3 that although we do choose the number of columns and pair of rows we want to mutate, it is not guaranteed to mutate all of them. In part (a), we saw the `colindx=2` have no mutation, the `colindx=5` have one mutation, and the `colindx=6` have two mutations. So, the maximum number of mutations for each column is NR.

Fig. 3: Example of Mutation



(a) First Matrix

(b) Second Matrix

(c) Third Matrix

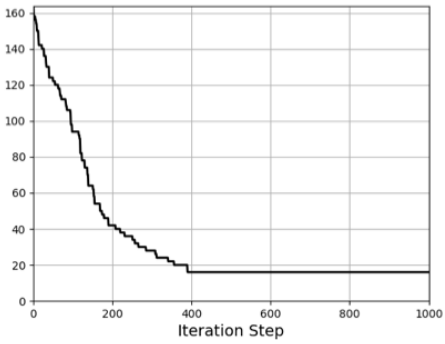
(d) Fourth Matrix

This third mutation method improves upon the issue of working with larger matrices, we have achieved greater variation among offspring matrices and obtained faster convergence to Hadamard matrices. Using this method, the largest Hadamard matrix found so far is of the size 32×32. This is an improvement from the other two mutation methods that produced Hadamard matrices up to size 12×12 only.

4 Computational Results

In this section we further comment on the efficiency of our CUDA accelerated code and discuss the computational results.

Fig. 4: Stalled convergence of the Fit function



4.1 Local Minimum

Genetic Algorithms also suffer from finding local minimums instead of global minimum similarly to the Simulated Annealing Algorithm [6, 13]. Figure 4 shows the minimum of the fit function as a function of the iteration number for matrix size 20×20 with $NC = NR = 4$. It is not just that the minimum of the fit function gets stuck at a positive constant without converging to zero, but the whole parent population becomes homogeneous. One way to try to prevent stalling at a local minimum is to use a selection process with some probability. This approach did not result in improved convergence. We had some success to speed up convergence by selecting low number of columns for the mutations. This is discussed in Subsection 4.3.

4.2 Fitness Function Comparisons

We compare the following two different fitness functions:

$$F_1 = \sum_{i,j} \left| Q^T Q \right| - m^2 \geq 0 \tag{8}$$

$$F_2 = \text{nonzero}\left(Q^T Q\right) - m \geq 0 \tag{9}$$

Table 2 shows the average speed (in seconds) for the two fitness functions while working with $40,000$ matrices at a time and over $10,000$ iterations. The execution time was averaged through 10 runs. The parameter are $N = 10^4$, $T = 10^4$, $NC = 4$, and $NR = 2$ for several matrix sizes. The purpose of this table was to compare the speed of each fitness function while working with large number of matrices. Note that none of these runs resulted in any Hadamard matrices due to the relatively small number of interations.

From Table 2 we can see that F_2 is the superior fitness function. These results were showed for only 10^4 iterations. Typically when finding larger Hadamard matrices, 10^7 iterations and greater is usually needed to find them.

Table 2: Speed of code completion in seconds using 40000 matrices and 10000 iterations

Matrix sizes	F_1	F_2
20×20	60.50	59.38
40×40	103.15	99.84
60×60	176.28	170.20
80×80	275.92	256.59
100×100	416.68	381.38

Table 3: Average Iteration steps for a 12×12 matrix using the Mutation kernel function

	NC			
NR	1	2	3	4
1	34.2	35.8	37.4	35.8
2	32.3	38.8	40.5	43.4
3	35.2	42.6	47.5	50.7
4	36.3	46.0	49.9	65.7

Consequently, for a larger number of iterations, F_2 is expected to perform significantly better. For these reasons, for the results in Section 4.4, we use the fitness function F_2 given in (9).

4.3 Mutation Comparisons

Table 3 shows the average number of iterations required to find a 12×12 Hadamard matrix given different NC and NR values. For each case 10 runs were made. The numbers in the table suggests that mutating two pairs $(NR = 2)$ in one column $(NC = 1)$ takes the least amount of steps to find an Hadamard matrix. In addition, the best results occurred when $NC = 1$ with any amount of row pairs.

Table 4 shows the average number of iterations required to find a 16×16 Hadamard matrix given different NC and NR values. For each case 10 runs were made. We found that the smallest numbers for $NR = 1$ and $NC = 2, 3, 4$ were obtained with the majority runs not finding Hadamard matrix. This suggest that either an Hadamard matrix was found fast or it wasn't found at all. The best results occurred when $NC = 1$ with any amount of row pairs. Although it took slightly more iterations to find the Hadamard matrices, it finding them more consistently. For the case of larger matrices, this table summarizes those results. Therefore when finding Hadamard matrices, we use a low number of columns $(NC = 1)$ or $(NC = 2)$ and a larger number of row pairs depending on the size of the matrix.

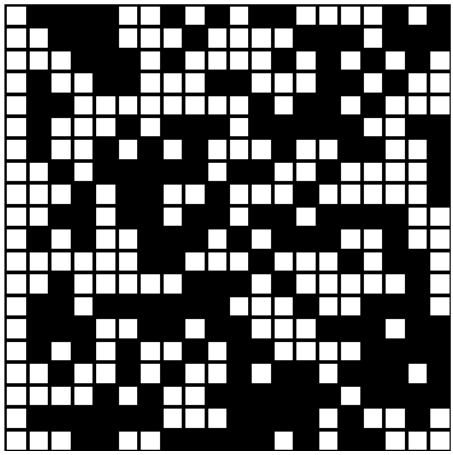
4.4 Computational Results

The following are the Hadamard matrices found using fitness function F_2 and the mutation kernel function `Mutation`.

Table 4: Average Iteration steps for a 16×16 matrix using the Mutation kernel function

	NC			
NR	1	2	3	4
1	130.1	119.4*	116.0*	115.5*
2	121.0	181.3	135.0	168.7
3	120.0	158.1	165.9	202.7
4	120.2	166.2	236.2	301.6

Fig. 5: 20×20 Hadamard matrix



1. In Figure 5, we show a 20×20 Hadamard matrix using the parameters $k = 5$, $N = 10^3$, $T = 10^7$, $NC = 2$, and $NR = 8$. We obtained this result after 517 seconds and 258875 iterations.
2. In Figure 6, we show a 24×24 Hadamard matrix using the parameters $k = 6$, $N = 10^3$, $T = 10^7$, $NC = 2$, and $NR = 10$. We obtained this result after 19997 seconds and 9576814 iterations.
3. In Figure 7, we show a 28×28 Hadamard matrix using the parameters $k = 7$, $N = 10^4$, $T = 10^8$, $NC = 1$, and $NR = 10$. We obtained this result after 3054 seconds and 428082 iterations.
4. In Figure 8, we show a 32×32 Hadamard matrix using the parameters $k = 8$, $N = 2(10^4)$, $T = 10^7$, $NC = 1$, and $NR = 12$. We obtained this result after 94923 seconds and 7472853 iterations.

5 Conclusion

In this paper, a genetic algorithm is proposed for the construction of Hadamard Matrices. The use of CuPy library in Python on graphics processing units enabled us to handle populations of thousands of matrices and matrix operations in parallel. Our algorithm was able to find several Hadamard matrices up

Fig. 6: 24×24 Hadamard matrix

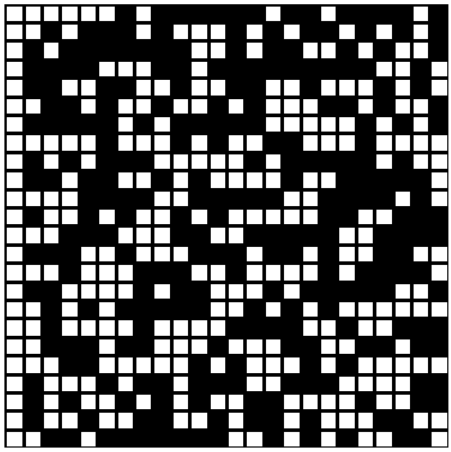
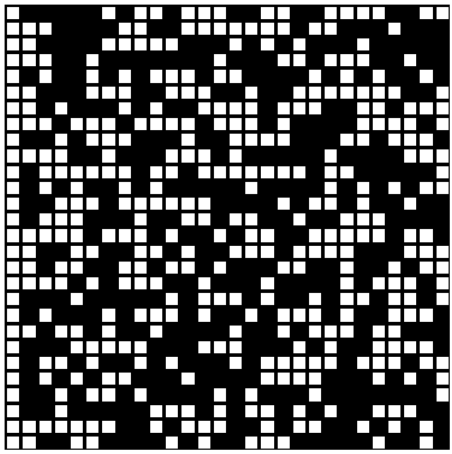


Fig. 7: 28×28 Hadamard matrix

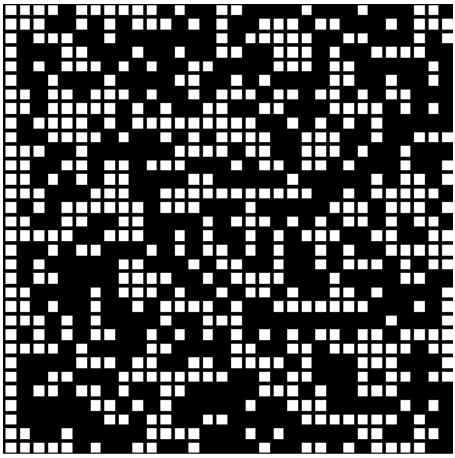


to size 32×32. We did not observe any issues with the crossover part, but the effectiveness was sensitive to the mutation rate as it is typical in genetic algorithms. We also observed premature convergence rate of the fitness function with stalled convergence ultimately for large matrices.

6 Numerical Code

```
import numpy as np
import cupy as cp
import math
import os
```

Fig. 8: 32×32 Hadamard matrix



```
#####
##### Raw kernel function: shuffling of +1/-1 in columns ####
Shuffle_Column = cp.RawKernel(r'''
#include <curand_kernel.h>
extern "C" __global__
void Shuffle_Column(char *Q, int seed, const int m, const int N)
{
    int k = blockDim.x*blockIdx.x+threadIdx.x; // matrix index
    int i = blockDim.y*blockIdx.y+threadIdx.y; // row index
    int j = blockDim.z*blockIdx.z+threadIdx.z; // column index
    unsigned long int seq, offset;
    int i1, i2;
    char ti1, ti2;
    seed=seed+k; seq = j; offset = 0;
    curandState h;
    if ((k<4*N)&&(j>0)&&(j<m)&&(i==0)){
        curand_init(seed,seq,offset,&h);
        for(i1=0; i1<m-1; i1++){
            i2=(int)(curand_uniform(&h)*(m-i1)+i1);
            ti1=Q[m*m*k+m*i1+j];
            ti2=Q[m*m*k+m*i2+j];
            Q[m*m*k+m*i1+j]=ti2;
            Q[m*m*k+m*i2+j]=ti1;
        }
    }
}
''', 'Shuffle_Column', backend='nvcc')
##### End of shuffling kernel function ####
```

```
33
34
35 ##### Raw kernel function: crossover ####
36 Crossover = cp.RawKernel(r'''
37 extern "C" __global__
38 void Crossover(char *Q, const int* c_cop, const int m, const
       int N)
39 {
40     int k = blockDim.x*blockIdx.x+threadIdx.x;
41     int i = blockDim.y*blockIdx.y+threadIdx.y;
42     int j = blockDim.z*blockIdx.z+threadIdx.z;
43     if (k<N){
44         if((j<c_cop[k]) && (i<m)){
45             Q[m*m*(k+2*N)+m*i+j]=Q[m*m*k+m*i+j]; //Copy the
                   first part (columns before c_cop) of P1 into O1
46             Q[m*m*(k+3*N)+m*i+j]=Q[m*m*(k+N)+m*i+j]; //Copy the
                   first part of P2 into O2
47        }
48         if((j>=c_cop[k]) && (j<m) && (i<m)){
49             Q[m*m*(k+3*N)+m*i+j]=Q[m*m*k+m*i+j]; //Copy the
                   second part of P1 into O2
50             Q[m*m*(k+2*N)+m*i+j]=Q[m*m*(k+N)+m*i+j]; //Copy the
                   second part of P2 into O1
51         }
52     }
53 }
54 ''', 'Crossover')
55 ##### End of crossover kernel function ####
56
57 ##### Raw CUDA kernel function for Mutation ####
58 Mutation = cp.RawKernel(r'''
59 extern "C" __global__
60 void Mutation(char *Q, const int* row1, const int* row2, const
       int* col, const int NC, const int NR, const int m, const
       int N)
61 {
62     int k = blockDim.x*blockIdx.x+threadIdx.x;
63     int i = blockDim.y*blockIdx.y+threadIdx.y;
64     int j = blockDim.z*blockIdx.z+threadIdx.z;
65     //k is the index of matrices, i is the index of rows, j is
           the index of columns
66     int jc, ir, coljc, rowir1, rowir2;
67     if ((k>=2*N) && (k<4*N)){ // Only the offspring are mutated
68         if((i==0) && (j==0)){
69             for(jc=0; jc<NC; jc++){
```

```
                coljc=col[(k-2*N)*NC+jc];
                for(ir=0; ir<NR; ir++){
                    rowir1=row1[(k-2*N)*NR+ir];
                    rowir2=row2[(k-2*N)*NR+ir];
                    if (Q[m*m*k+m*rowir1+coljc] !=
                        Q[m*m*k+m*rowir2+coljc]){ /* Only +1 and
                        -1 pairs are flipped to keep the balance*/
                        Q[m*m*k+m*rowir1+coljc] *= -1;
                        Q[m*m*k+m*rowir2+coljc] *= -1;
                    }
                }
            }
        }
    }
}
''', 'Mutation')
##### End of mutation kernel function ####

k=3
m=4*k # size of matrices
N=1000 # 4N is the population size
T=10**3 # maximum number of iterations
NC=10  # columns for swap < m
NR=6  # rows for swap < m/2

blocksx = 8; blocksy = 8; blocksz = 8
blocks = (blocksx, blocksy, blocksz)
grids = (math.ceil(4*N/blocksx), math.ceil(m/blocksy),
    math.ceil(m/blocksz))

#### Initial Population ####
Pop = cp.ones((4*N, m, m), dtype=cp.int8)
Pop[:,0:2*k,1:m] = -1
seed=int.from_bytes(os.urandom(4), 'big')
Shuffle_Column(grids, blocks,(Pop, seed, m, N))

#Fitness Function
F=cp.count_nonzero(cp.matmul(cp.transpose(Pop,axes=(0,2,1)),Pop),
    axis=(1,2))-m
Fmin=cp.amin(F)

t=0
while (t<T) and (Fmin>0):
    #Selection
    #Moves best matrices (with the lowest norm)
```

```
    # to the parent position at the beginning
    # in random order
    Pop[0:2*N,:,:]=Pop[cp.random.permutation(cp.argsort(F)[:2*N]),:,:]


    #Crossover
    c_cop=cp.random.random_integers(1,m-1,N)
    Crossover(grids, blocks, (Pop, c_cop, m, N))

    #Mutation
    rowindx1=cp.random.random_integers(0,m-1,size=(2*N,NR))
    rowindx2=cp.random.random_integers(0,m-1,size=(2*N,NR))
    colindx=cp.random.random_integers(1,m-1,size=(2*N,NC))
    Mutation(grids, blocks, (Pop, rowindx1, rowindx2, colindx,
        NC, NR, m, N))

    #Fitness Function
    F=cp.count_nonzero(cp.matmul(cp.transpose(Pop,axes=(0,2,1)),Pop),
        axis=(1,2))-m
```

Declarations

The work of R. Ruiz was supported by a Dean's Graduate Assistantship Award from the College of Sciences at the University of Texas Rio Grande Valley. The authors have no other relevant financial or non-financial interests or conflict of interests to disclose.

References

[1] Hadamard, J.: Résolution d'une question relative aux déterminants. Bulletin des Sciences Mathématiques **17**, 240–246 (1893)

[2] Goethals, J.M., Seidel, J.J.: A skew Hadamard matrix of order 36. J. Austral. Math. Soc. **11**, 343–344 (1970)

[3] Jayathilake, A.A.C.A., Perera, A.A.I., Chamikara, M.A.P.: A new set of 32 in-equivalent hadamard matrices of order 404 of goethals- seidel type. Elixir **69**, 23266–23272 (2014)

[4] Mohammadian, A., Tayfeh-Rezaie, B.: Hadamard matrices with few distinct types. Linear and Multilinear Algebra **67**(8), 1596–1605 (2019). https://doi.org/10.1080/03081087.2018.1464113

[5] Seberry, J., Yamada, M.: Hadamard matrices, sequences, and block designs. In: Contemporary Design Theory. Wiley-Intersci. Ser. Discrete

Math. Optim., pp. 431–560. Wiley, New York, ??? (1992)

[6] Suksmono, A.: Finding a hadamard matrix by simulated annealing of spin-vectors. Journal of Physics: Conference Series **18(3)**, 66–70 (2016)

[7] Suksmono, A.B., Minato, Y.: Finding hadamard matrices by a quantum annealing machine. Scientific Reports **9**(14380) (2019). https://doi.org/10.1038/s41598-019-50473-w

[8] Browne, P., Egan, R., Hegarty, F., Catháin, P.: A survey of the hadamard maximal determinant problem. The Electronic Journal of Combinatorics **28**(4) (2021). https://doi.org/10.37236/10367

[9] Han, A., Rochford, K.: EVGA GeForce RTX 3080 XC3 BLACK GAMING, 10G-P5-3881-KL, 10GB GDDR6X, iCX3 Cooling, ARGB LED, LHR. https://www.evga.com/products/product.aspx?pn=10G-P5-3881-KL (2020)

[10] Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M.H., Brett, M., Haldane, A., del Río, J.F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T.E.: Array programming with NumPy. Nature **585**(7825), 357–362 (2020). https://doi.org/10.1038/s41586-020-2649-2

[11] Okuta, R., Unno, Y., Nishino, D., Hido, S., Loomis, C.: Cupy: A numpy-compatible library for nvidia gpu calculations. In: Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS) (2017). http://learningsys.org/nips17/assets/papers/paper_16.pdf

[12] NVIDIA, Vingelmann, P., Fitzek, F.H.P.: CUDA, release: 10.2.89 (2020). https://developer.nvidia.com/cuda-toolkit

[13] Wirsansky, E.: Hands-On Genetic Algorithms with Python: Applying Genetic Algorithms to Solve Real-world Deep Learning and Artificial Intelligence Problems. Packt Publishing, ??? (2020). https://books.google.com/books?id=A0vODwAAQBAJ

[14] Bridges, C.L., Goldberg, D.E.: An analysis of reproduction and crossover in a binary-coded genetic algorithm. In: Grefenstette, J.J. (ed.) ICGA, pp. 9–13 (1987). http://dblp.uni-trier.de/db/conf/icga/icga1987.html

[15] Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, MI (1975). second edition, 1992

[16] Fisher, R.A., Yates, F.: Statistical Tables for Biological, Agricultural and Medical Research, pp. 90–1. Oliver and Boyd, Edinburgh, UK; London, UK (1938)

[17] Durstenfeld, R.: Algorithm 235: Random permutation. Commun. ACM **7**(7), 420 (1964). https://doi.org/10.1145/364520.364540