MONASH University

Parallel implementation of the Gauss-Jordan inversion algorithm using CUDA

A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA [1]

Presentation by:
Sami Hussein Hamid Saeed
ID:31195261

MONASH University

Introduction & Background

- **Why is matrix inversion important?**
- **Why is matrix inversion slow?**
- **Why use GPUs?**
- **Paper Contribution**

Matrix Inversion Uses

Finding the inverse of a matrix can be important for several different reasons such as:

- Structural analyses using finite element method.
- 3D rendering.
- Digital filtering.
- Image processing.
- In general, solving linear equations.

Example:

Given

$$AX = B$$

we can multiply both sides by the inverse of A, provided this exists, to give

$$A^{-1}AX = A^{-1}B$$

But $A^{-1}A = I$, the identity matrix. Furthermore, $\boxed{IX = X,}$ because multiplying any matrix by an identity matrix of the appropriate size leaves the matrix unaltered. So

$$X = A^{-1}B$$

Simple example of matrix inversion being used [2].

Gauss-Jordan Algorithm for Matrix Inversion

Until the late 1960’s the fastest & simplest method for matrix inversion was the Gauss-Jordan method. It augments a Matrix A and it’s identity I. Then by performing simple row operations we transform A into I and I becomes A^-1

- The method uses a loop to process each column.
- Within each column, the diagonal element (Matrix[k][k]) is checked to see whether it is 0 or not, if it is then a swap/addition operation is performed.
- Another loop is run to divide the row k by matrix[k][k] in order to set matrix[k][k] to 1.
- Finally, we must set all elements within column k to 0 by subtracting the rows to negate the values in column k, except the diagonal element at matrix[k][k].
- **Has a worst case time complexity of O(n^3).**
    - Where n is the matrix size.

Algorithm 1 Gauss-Jordan Algorithm

procedure GAUSSJORDAN(A)
    Let A be the input matrix
    Let I be the identity matrix of the same size as A
    for $k = 1$ to n **do**
        Find the row r such that $A[r][k]$ is the maximum among $A[k][k]$, $A[k+1][k]$, ..., $A[n][k]$
        Swap rows $A[r]$ and $A[k]$
        Divide row $A[k]$ by $A[k][k]$
        Divide row $I[k]$ by $A[k][k]$
        for $i = 1$ to n **do**
            if i is not equal to k **then**
                $factor \leftarrow A[i][k]$
                for $j = 1$ to n **do**
                    $A[i][j] \leftarrow A[i][j] - factor \times A[k][j]$
                    $I[i][j] \leftarrow I[i][j] - factor \times I[k][j]$
                end for
            end if
        end for
    end for
    A^{-1} is the matrix I
end procedure

Why GPUs?

- The Gauss-Jordan method is suitable for massive parallelisation.
    - Lots of independent operations performed sequentially. Each row has N elements where row operations can be done on all elements in parallel.
- GPU architecture supports massively parallel processing by enabling the usage of thousands of threads in parallel.
    - GPUs generally have much more ALUs than a CPU.
- CPUs only allow for 8 to 12 threads.
- Thread creation and memory transfer cost is much lower in GPU.

Comparison of CPU and GPU architecture [3]

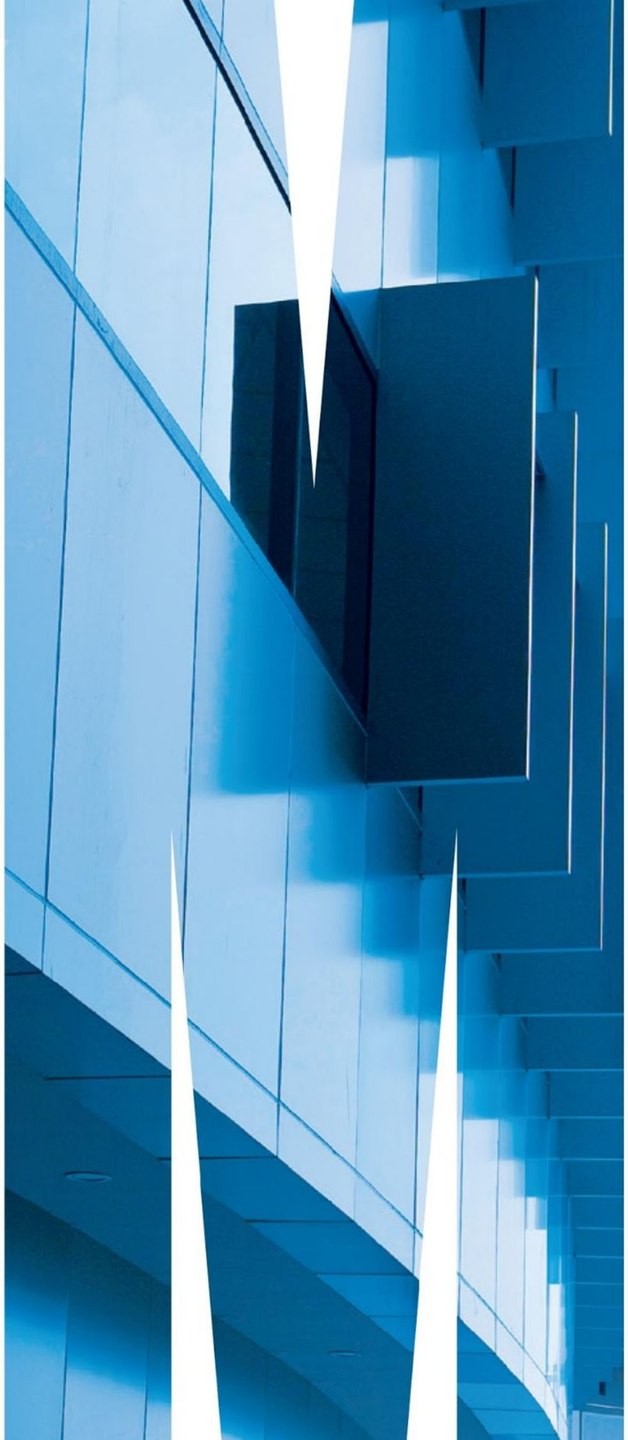
Proposed Contribution

The paper's proposed contributions are:

- **Compare** and **contrast** different matrix inversion methods and algorithms.
- **Redesign** the Gauss-Jordan algorithm in order to exploit massively multithreaded GPUs.
- Perform **testing** on different **types** of matrices of different **sizes**.
- Prove that the time complexity of the algorithm can scale as **O(n)**, **provided n^2 threads** can be created.

Hypothesis & Problem Statement

- Hypothesis
- Problem Statement



Problem Statement

Most of the **basic** matrix inversion methods are usually done by simply performing **repeated independent row operations**. Each row operation requires a loop to iterate through each column in a row.

Problems faced in matrix inversion:

- All of the row operations need to be performed on each element within a row, meaning that at best, a serial implementation of O(n^2) might be possible in the future.
- A large number of independent row and column operations are done in a sequential order.
- CPU parallelism is insufficient for larger matrices as it can only support a limited number of threads.

The hypothesis of the paper is that the proposed parallel Gauss-Jordan inversion algorithm implemented on CUDA can significantly reduce the worst case time complexity for matrix inversion from $O(n^3)$ to a linear $O(n)$ provided n^2 threads.

Related work

Related work

Methods	Pros	Cons
Strassen	- First algorithm to reduce matrix inversion from O(n^3) to O(n^2.808). - This was done by **recursively** breaking down the matrices into smaller submatrices and performing **fewer** arithmetic operations. - Has been **repeated improved** by different researchers using different methods to test the **lowest limit** of exponent value, which is suggested to be 2.	- Cannot be easily parallelised due to its recursive nature. - Requires a large value of N to be parallelised. - Accuracy of the inverse depends on the sub-matrix selected in the initial step.
Strassen-Newton	- A **parallel** variation of **strassen's** method using newton iterations. - Can yield moderate results even on a **single CPU**. - Up to **55% speedup** can be observed even on a single processor.	- Design is coupled with the limitations of the time's architecture (Published in 1988). - Speedup is relatively low. - Does not take into account the usage of GPUs.
Coppersmith & Winograd	- Further improves matrix inversion using strassen's new laser method. - Avoids unnecessary arithmetic operations. - Considerably improved the runtime complexity to a record breaking O(n^2.376).	- Only applicable to specific types of matrices - Extremely complex to implement without a deep understanding of linear algebra.
LUP Decomposition	- Decomposes the matrix into three sub matrices, a lower triangular matrix, upper triangular matrix and a permutation matrix. - More efficient when dealing with a large value of N. - Numerically stable.	- Consumes a lot of RAM to store all 3 matrices. - Time complexity is still non-linear.

Related work

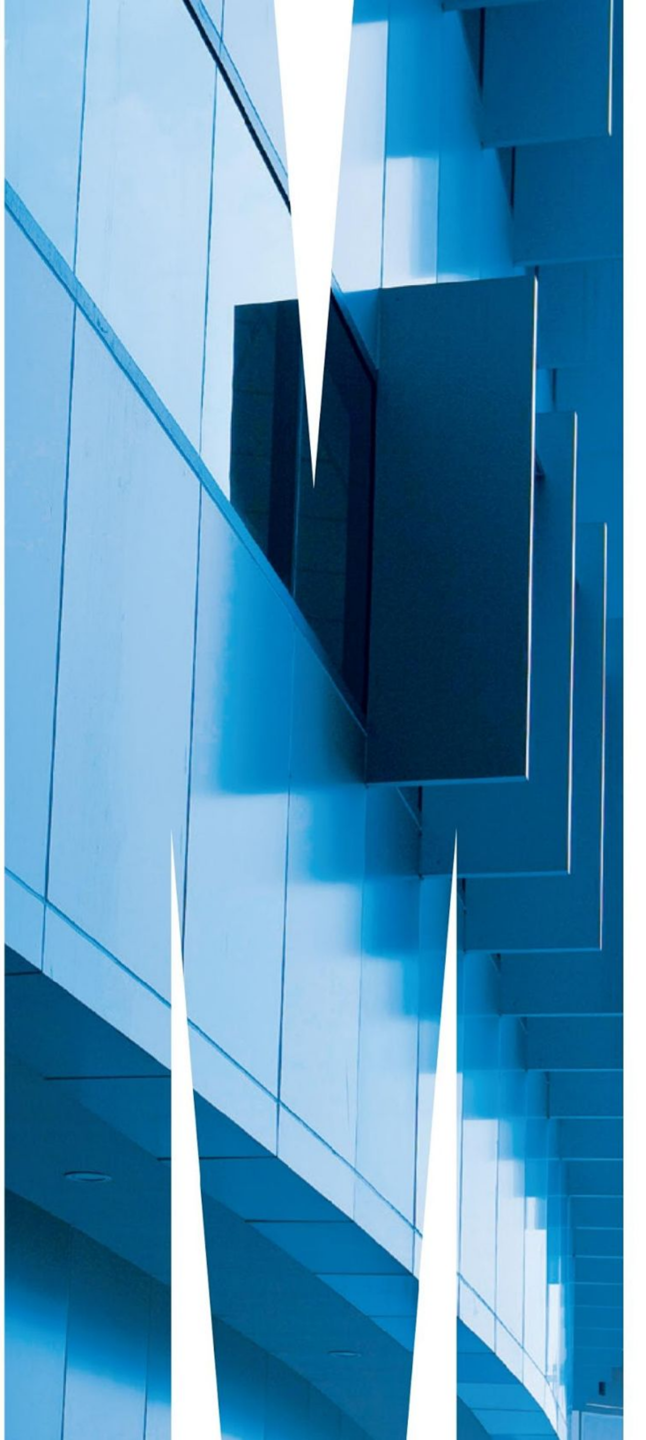
Methods	Pros	Cons
Cholesky Decomposition	- Also performs decomposition of the matrix into submatrices, a lower triangular matrix and its transpose. - Further optimizes the algorithm for specific matrices such as symmetric matrices and matrices with positive definite.	- Consumes a lot of RAM to store 2 matrices. - Only applicable to select matrices. - Tough to parallelise due to dependencies.
QR Decomposition	- Breaks down the matrix into an orthogonal matrix Q and a upper triangular matrix R. - Numerically stable. - Can be applied to rectangular matrices.	- Consumes a lot of RAM to store 2 matrices. - Requires additional pre-processing for some irregular matrices.
RRQR Factorization	- Further improvement of the QR Decomposition method. - Focuses on more important parts of the matrix. Thus allowing for less computations do be done.	- Inherits the same space and pre-processing limitations from QR decomposition.
Monte Carlo Methods	- Provides an approximate solution using random sampling. - No set time complexity. - Used for inverting the Hermitian matrix and positive definite matrix.	- Subject to sampling errors. - Converging to true solution may require large amount of samples. - Only applicable to specific matrices.

Common limitations found in most papers:

- Most papers aim to reduce n’s exponent by a small fraction.
- Most papers do not address the use of massive parallelisation.
  - This is mainly because all of the **advanced** optimizations made in these papers require some form of **dependency** between each iteration, making the code **non parallelizable**.
- Some of the fastest methods are only applicable for specific type of matrices.

MONASH University

Methodology



Methodology - Approach

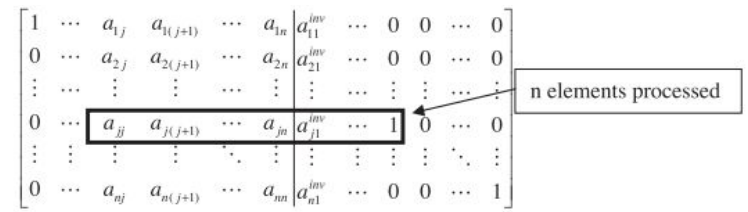
1) **Outermost loop to iterate through each column.**
   a) Cannot be done in parallel due to dependency between iterations.
2) **Ensure that the diagonal element (matrix[j][j]) is non-zero.**
   a) Find row k where column j is not 0.
   b) Add the two rows together in parallel (1 thread for each column).
3) **Convert the diagonal element into 1 by dividing the entire row by matrix[j][j]**
   a) The division of the entire row can be done in parallel using 1 thread to perform the division on each column.
4) **Convert all other rows in the column to 0**
   a) This is done by simply subtracting the value by itself multiplied by the row containing the diagonal.
   b) There are n blocks created containing n threads each. A **block** represents a **column** and each **thread** represents a **row** for each column/block.

```
Read matrix
Initialize n to size of matrix
Initialize j to 0
while j < n, do:

  Find k where matrix[k][j] is not 0
  Spawn n threads in 1 block
  for thread i of n in block 1, do:
    matrix[j][i] = matrix[j][i] + matrix[k][i]
  end for

  Spawn n threads in 1 block
  for thread i of n in block 1, do:
    matrix[j][i] = matrix[j][i]/matrix[j][j]
  end for

  Spawn n threads each in n blocks
  for thread i of n in block r, do:
    matrix[i][r] = matrix[i][r] - matrix[i][j]*matrix[j][r]
  end for
  Increase j by 1

end while
Write matrix
```

Methodology - CUDA Implementation

Row Fixing

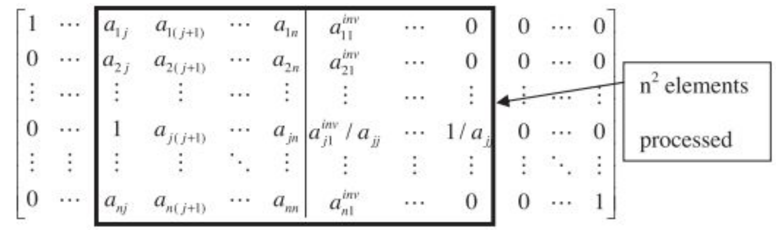
- Each row must be divided by its diagonal element in order to turn the diagonal element itself into 1.
- Here this is done by multithreading, where each thread represents a column and all divisions are done in parallel




```
__global__ void fixRow(float *matrix, int size, int rowId) {
  // The ith row of the matrix
  __shared__ float Ri[512];
  // The diagonal element for ith row
  __shared__ float Aii;
  int colId = threadIdx.x;
  Ri[colId] = matrix[size * rowId + colId];
  Aii = matrix[size * rowId + sharedRowId];
  __syncthreads();
  // Divide the whole row by the diagonal element making sure it is not 0
  Ri[colId] = Ri[colId]/Aii;
  matrix[size * rowId + colId] = Ri[colId];
}
```

Column Fixing

- Each row in each column must be subtracted by itself multiplied by the diagonal element in order to turn the entire column of the diagonal element into 0.
- Here this is done by multithreading, where each block represents a column and each thread within the block represents a row and then all subtractions are done in parallel.




```
__global__ void fixColumn(float *matrix, int size, int colId) {
  int i = threadIdx.x;
  int j = blockIdx.x;
  // The colId column
  __shared__ float col[512];
  // The jth element of the colId row
  __shared__ float AColIdj;
  // The jth column
  __shared__ float colj[512];
  col[i] = matrix[i * size + colId];
  if (col[i] != 0) {
    colj[i] = matrix[i * size + j];
    AColIdj = matrix[colId * size + j];
    if (i != colId) {
      colj[i] = colj[i] - AColIdj * col[i];
    }
    matrix[i * size + j] = colj[i];
  }
}
```

MONASH University

Overall the methodology addresses the two main issues identified earlier since:

1) The proposed approach indicates, that a linear runtime complexity algorithm for all matrix types is possible since the new method **has parallelised all the loops inside the main loop.**
2) By using an architecture that supports **massively parallel** execution such as GPUs we can effortlessly and quickly create a large amount of threads compared to the limited and slow CPU threads.

MONASH University

Analysis of Results & Conclusions

Analysis of Results - Runtime

The testing system uses a Intel Quad Core processor Q8400 @ 2.66 GHz each. The GPU used for executing the CUDA code is a GTX 260.

The authors conduct the runtime tests using different types of matrices:

- Sparse Matrix
  – A matrix where most elements are 0.
- Band Matrix
  – A sparse matrix where it's non-zero elements are located in the diagonal region.
- Hollow Matrix
  – A matrix where the diagonal elements of a matrix are 0. This is the **most computationally expensive type of matrix to inverse**, as it requires row addition or swapping to be done for each column.
- Identity Matrix
  – A matrix where all the elements are set to 0, except the diagonal elements, which are set to 1
- RandomMatrix

Results

- All GPU tests show that for almost all matrices the runtime scales linearly with matrix size.
- There first quadratic curvature can be noticed at around n=100, due to the number of threads requested being more than the thread available in a GTX 260.
- Hollow matrix has a steeper gradient due to it being the matrix with the most number of computations necessary.
- Overall, the code performs almost 10 times as fast in a GPU than a CPU.

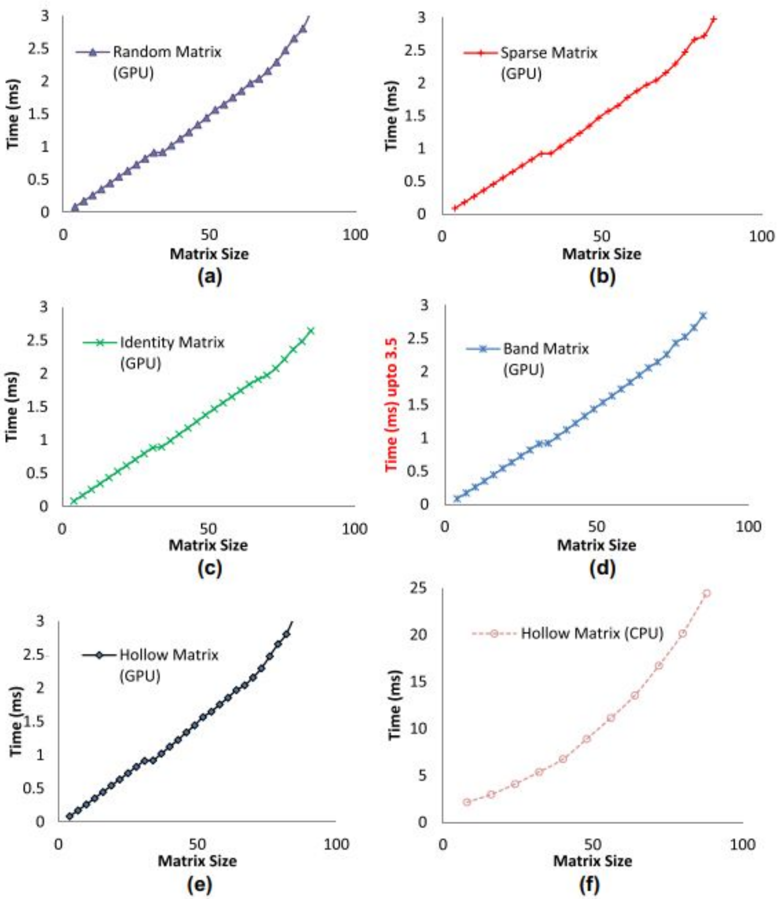


Fig. 4. (a–e): Linear computation time for matrix inversion is observed up to $n \approx 64$ using GPU, (f) computation time for inverting hollow matrix using CPU.

Results (Scaled Up)

- GPU code no longer exhibits a linear increase in runtime as the matrix size increases.
- The change in the time complexity can be attributed to the fact that the GTX 260 GPU can store 9216 threads and 288 dispatched threads concurrently.
- Given the need for 262,144 threads to process a 512-sized matrix, the GPU's thread capacity is exceeded, necessitating some threads to be executed sequentially.
- GPU code is still considerably faster than the CPU code, regardless of the size.

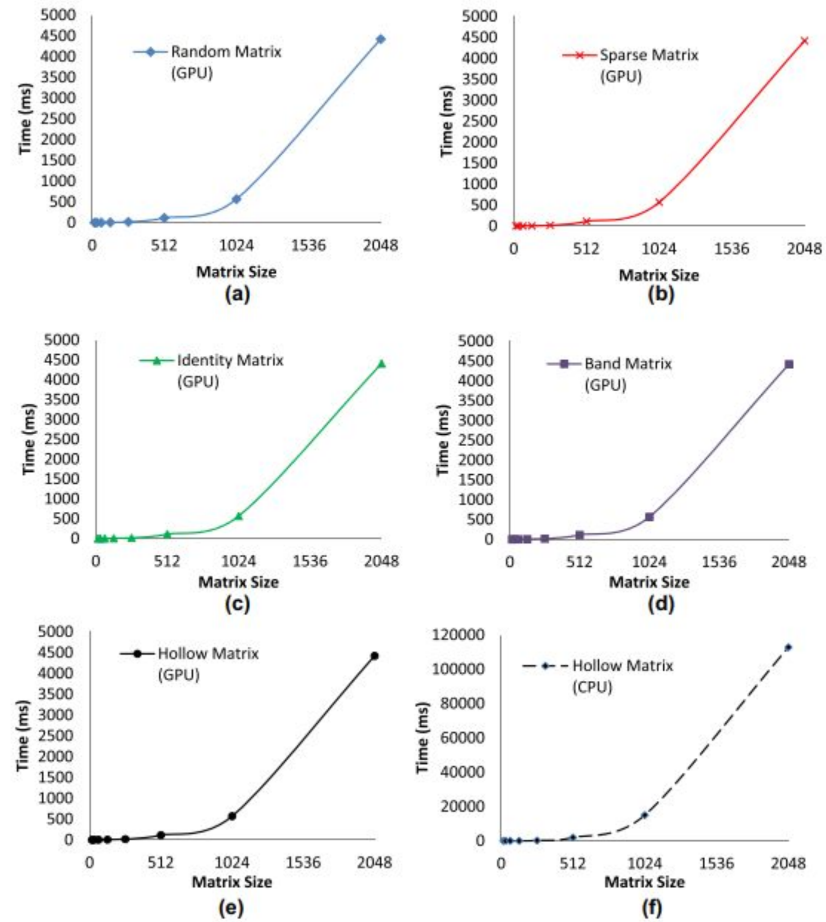


Fig. 5. Computation time for inverting different types of matrices, (a–e): using GPU, (f) using CPU.

Analysis of Results - Conclusion

Overall, the tests performed by the researchers show that the tests do indeed support the hypothesis made, by showing that:

1) The worst case time complexity of the matrix inversion does in fact stay at O(n) when allocated n^2 threads.
2) The algorithm can work for any type of matrix.

References

[1] G. Sharma, A. Agarwala, and B. Bhattacharya, “A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA,” *Computers & Structures*, vol. 128, pp. 31–37, Nov. 2013, doi: 10.1016/j.compstruc.2013.06.015.

[2] Math Centre. "Engineering Math First Aid Kit: Using inverse matrix to solve equations" Math Centre, n.d., https://www.mathcentre.ac.uk/resources/Engineering%20maths%20first%20aid%20kit/latexsource%20and%20diagrams/5_6.pdf.

[3] Thambawita, V., Vajira, R., Ragel, R., & Elkaduwe, D. (2014). "To Use or Not to Use: Graphics Processing Units for Pattern Matching Algorithms." IEEE.