Fall 2022 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

Optimizing the LU Factorization Algorithm for CPU-GPU Heterogenous Systems

Vishnu Ramadas

December 14, 2022

Abstract

LU Factorization is a very important part of computational mathematics today. It is a generic and simple way to solve a system of linear equations, invert a matrix, or calculate its determinant and is faster than the straightforward reduced row echelon method. LU Factorization has many parallelizable elements since it involves a series of matrix multiplications involving elementary row operation matrices. This project attempts to study several possible parallel implementations of the algorithm and try to reason about the performances they exhibit

Link to Final Project `git` repo: https://git.doit.wisc.edu/VRAMADAS/repo759.git

Contents

1. General information

1. Home Department - ECE
2. Current status: MS student
3. I release the ME759 Final Project code as open source and under a BSD3 license for unfettered use of it by any interested party.

2. Problem statement

The project I chose to work on is implementing LU factorization using different parallelization schemes to compare relative performance. LU factorization is a method commonly used by computers to solves systems of linear equations, invert matrices, or compute determinants. Various efforts to distribute the computation on heterogenous systems exist. Tan et al optimize LU Factorization in LINPACK to introduce a pipeline consisting of various stages on customized AMD GPU clusters [1]. Wu et al propose a two-stage column block parallelization using BLAS that is distributed over multiple CPU nodes and GPUs [2]. This project, inspired by these, attempts to find the most optimal solution for an implementation written from scratch for a system containing a single CPU node connected to a GPU.

The LU factorization (or decomposition) algorithm factors a matrix into a product of a lower triangular and an upper triangular matrix. It makes use of Elementary Row Operations (EROs) to represent a matrix, say A, as follows [3]

$$A = LU \tag{1}$$

If a system of linear equations exists such that,

$$Ax = b \tag{2}$$

then, the LU factorization is a far more efficient way to calculate its solutions when compared with the Reduced Row Echelon (RRE) form method. This is because the factorization avoids the extra computation required to reduce a matrix to RRE form. This algorithm can also be used to calculate the inverses and determinants of matrices. If A is of the form

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \tag{3}$$

Then multiplying by the following ERO E_{12} [3]

$$E_{12} = \begin{bmatrix} 1 & \dots & 0 \\ -a_{21}/a_{11} & \dots & 0 \\ \dots & \dots & 1 \end{bmatrix} \tag{4}$$

results in the first element of the second row of A reducing to a 0. The non-zero non-diagonal entry ($-a_{21}/a_{11}$ in this equation 4) is called the pivot for the row reduction corresponding to its row. The report will refer to the set of pivots in a particular column as the pivots as a pivot family. In a similar fashion, all lower triangular elements of A can be reduced to give the U factor [3]

$$U = E_{n-1,n}E_{n-2,n-1}E_{n-2,n} \dots E_{12}A \tag{5}$$

Inverting the EROs gives us L such that equation 1 is satisfied [3].

$$L = E_{12}^{-1}E_{12}^{-1} \dots E_{1n}^{-1}E_{23}^{-1} \dots E_{n-1,n}^{-1} \quad (6)$$

Equations 3 to 6 assume A is a square matrix of size n but the factorization can be extended to a matrix of any dimension. Since the sequence of operations are the same, the matrices will be assumed to be square for simpler representation. Having decomposed a matrix 'A' into its LU form, a two-step process can be employed to find its solution.

$$Let\ y = Ux \quad (7)$$

This reduces the system of linear equations to [3]

$$Ly = b \quad (8)$$

On solving for y via forward substitution, the system reduces to

$$Ux = y \quad (9)$$

Using substitution once again leads to the solution x. Since L and U are triangular, computing their inverses is also computationally simpler. Once their inverses are found, calculating the inverse of A is straightforward [3]

$$A^{-1} = U^{-1}L^{-1} \quad (10)$$

Since LU decomposition is a very popular algorithm in computational numerical analysis [4], it is critical to improve its computational performance. Amdahl's law suggests that the regions of code that are independent of each other can be parallelized to achieve speedup [5]. However, analytical performance models, such as LogCA, suggests that there are other overheads to consider when expecting acceleration speedups [6]. Speedup is not seen when the parallelizable gains cost of transferring the data are less than the costs involved in transferring data and/or scheduling the computation. Having established that smaller data sizes are better off using a serial algorithm, a straightforward acceleration for larger sizes is to parallelize the ERO operations for a particular pivot family. This can be achieved on a multicore CPU system by launching each row reduction as a separate OpenMP thread. The same can be achieved on a CPU-GPU heterogenous systems by offloading the entire factorization onto a GPU. Array sizes that are large enough to offset the scheduling overheads and data transfer speeds should produce a performance boost. Lastly, a hybrid approach involving both OpenMP and CUDA can also be tried.

Additional optimizations, such as the two-column block parallel factorization proposed by Wu et al [2], are also options. The authors present a two-stage solution where a matrix is first partitioned into blocks based on the number of processors in a cluster node. These blocks are arranged in a column. The algorithm iterates over each block assigned to a processor and splits compute across CPU and GPU. This relies heavily on inter-process communication and very large data sets to achieve speedup. For smaller datasets and systems, a simplified approach can be evaluated to study how the algorithm scales down to less bulky systems.

GPU computations can also be sped up by introducing concurrency between kernels using CUDA streams. Since GPUs have finite resources, there must be an optimal number of streams beyond which there are no additional performance benefits or are performance degradations. One of the objectives of the project is to identify the optimal number of CUDA streams for the GPU device used.

3. Solution description

The project focused on five different implementations of the LU factorization code as outlined in the previous section. The first is a serial implementation that used the naïve, straightforward approach. The second uses OpenMP to parallelize part of the reduction while the third implements it entirely in CUDA for GPU devices. The fourth implementation mixes aspects of OpenMP and CUDA to distribute the computation across a CPU-GPU Heterogenous system. The final implementation deals with a modified column block solution that also uses a heterogenous system to accelerate computation. All matrices in the implementation are stored in row-major form and are assumed to be square. The L matrix is initialized to all 0s, and the U matrix initialized to the input matrix in all implementations.

3.1. Serial Implementation

The first implementation of the LU factorization is a function called `luFactorizationSerial` and decomposes the matrix brutely. A "for" loop set all diagonal entries in L to 1 before the LU factorization began. The factorization itself is a straightforward version of the standard algorithm. It uses three loops iterating over the number of columns of the matrix. The outer loop calculates the pivot used for that column which will then be used to reduce subsequent rows. The rows that are reduced are in the lower triangular region of the matrix. The middle and inner-most loops track the row and column entries of the lower triangular region. The reductions for L, U are performed here. This, however, is not the most efficient way to implement LU factorization. The non-zero entries of L are essentially the pivot values used for each row reduction. The L matrix computations can be moved into the same loop that calculates the pivot values using this observation.

3.2. OpenMP Implementation

The OpenMP implementation, a function called `luFactorizationOMP`, uses the same base code as the serial implementation and parallelizes loop iteration across various cores. The two candidates for parallelization are the loop that initializes the diagonal elements in L and the loop that calculates the pivot for each set of row reductions (i.e., the middle loop). Both these loops are parallelized and scheduled statically [7].

3.3. CUDA Implementation

The CUDA implementation uses two kernels to factor a matrix into its L and U components. These kernels are iteratively called as many times as the number of columns. This is so that each kernel is called once for each pivot family. The first kernel, called `luFactorizationCUDA_pivotKernel`, sets the diagonal elements of L to 1 and off-diagonal elements in the column number ("for" loop iterator index) to the pivot values used during that iteration. The second kernel, called `luFactorizationCUDA_kernel`, calculates the elements of the upper triangle using the pivot values calculated and stored in L. The number of threads in each block for both kernel is the minimum of either the number of columns in the matrix or 1024. The number of blocks used are calculated differently for each kernel. For `luFactorizationCUDA_pivotKernel`, an entire column is processed parallelly, and the number of blocks is such that the elements equal to the number of rows is passed in one go. For a square matrix with n rows and columns,

$$numThreads = \min(n, 1024);\ \ numBlocks = \ (n + numThreads - 1)/numThreads \quad \textbf{(11)}$$

The second kernel uses the configuration, assuming the same matrix in equation 11 [8].

$$numThreads = \min(n, 1024);\ \ numBlocks = \ (n^2 + numThreads - 1)/numThreads \quad \textbf{(12)}$$

3.4. Hybrid OpenMP-CUDA Implementation

The Hybrid implementation uses both OpenMP and CUDA to parallelize the LU factorization algorithm. This approach reduces the number of "for" loops used from three in the serial/OpenMP implementations to two. The outer loop iterates over the number of columns of the matrix. The inner loop iterates over each of the remaining rows and calculates the pivot to be used to reduce that row. It then offloads the reduction to the GPU. The kernel configuration is the same as equation (12). To maximize concurrency, different CUDA streams are used to simultaneously reduce different rows.

3.5. Column Block Implementation

The algorithm proposed by Wu et al. proposed to first split the input matrix into a set of column blocks based on the number of processors in a node. It then decomposes the first column block before casting it to the others using MPI. The algorithm then swaps columns after running the LU algorithm on the processor's column block and before updating the blocks to the left and right of the said block [2]. The algorithm makes extensive use of BLAS and LAPACK routines. The 759 semester project, on the other hand, does not use pre-existing routines as the goal is to develop proficiency in parallel computing. Wu et al's algorithm is also aimed at accelerating LU factorization for huge data sets over multiple processes in a resource rich system. Evaluating something similar for smaller systems that run on a single node is the focus of this section. The algorithm was therefore simplified to compensate for the absence of inter-node synchronization. The simplified algorithm builds on the Hybrid implementation while doing two things differently. First, the calculation of the pivot (and L matrix, since it is the inverse of the ERO matrices, and therefore contains the pivots) is done in the outer loop. This is achieved by offloading computation to the GPU and calculating all the pivots needed to reduce rows across that column. Second, the inner loop processes blocks that span all rows but are only a few columns wide and applies EROs to these blocks by offloading computation onto the GPU on different streams. The number of columns in each block is referred to as the column width.

The intuitively expected outcome for the various implementations is that CUDA outperforms OpenMP, which in turn outperforms the serial implementation. The full benefits of this will be visible only for matrix sizes above a threshold that depends on scheduling and offloading overheads [6]. The Hybrid schemes will suffer from additional overheads at lower sizes while their behavior for larger matrices would be interesting. Similarly, increasing the number of CPU threads should initially improve performance until a certain point where it hits maximum parallelizability and overheads start dominating. Increasing the number of CUDA streams should also follow a similar pattern with the maxima depending more on hardware capabilities than the nature of the algorithm. Increasing block size should also have a similar effect on performance until the point where the matrices are too large for one CPU node to handle. It is possible that this point is not observed from the size of the experiments conducted for the project.

4. Results

The various implementations were run on an Euler node with one CPU and one GPU. The serial implementation was verified against the Python LU decomposition function from the SciPy package [9].

The output of the serial algorithm was treated as the golden reference output once it was verified to be correct. The rest of the implementations were invoked one after another and their runs were timed. Their L and U matrices were compared with that of the serial algorithm to verify correctness. The input square matrices, whose L and U factors were computed, were generated randomly with each entry being a float value in the range $(-n, n)$, where n is the size of each dimension of the matrix.

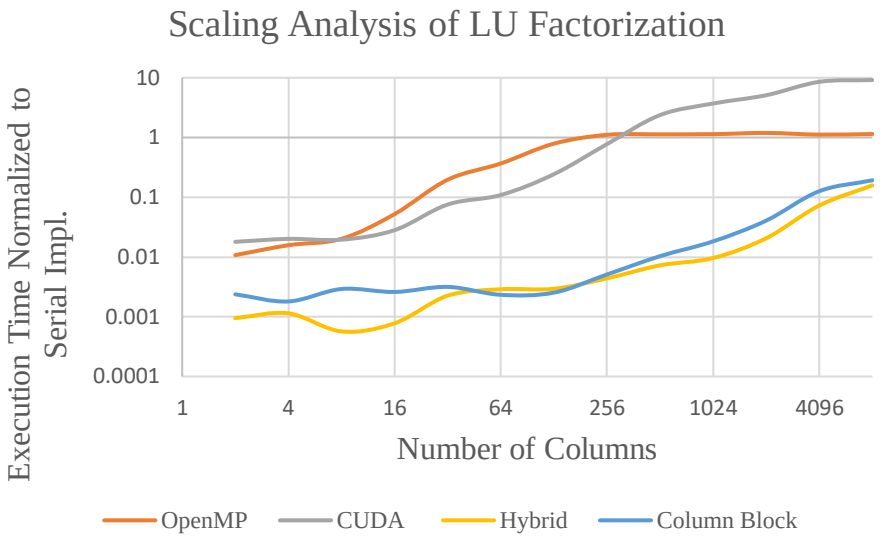

Figure 1

Figure 1 compares the performance of each of the parallel implementations normalized to the performance of the serial code for square matrices of lengths in range 2 to 8192. It is evident from the figure that CUDA performs the best. This, however, happens only when the matrix size is larger than 256x256 which aligns with the LogCA analytical model. OpenMP too performs poorly till the matrix size is 256x256, after which it is marginally better. This can be explained by pointing out that the amount of work parallelized by OpenMP is not much considering that it is not in the outer loop and thus, the parallelizable region is reduced. Amdahl's law predicts that the gains shouldn't be very high [5]. Moreover, the extra overheads needed to create n^2 parallel threads are a burden on the scheduler [6]. The surprising cases are the performances of the Hybrid and Column Block methods. They perform significantly worse than the serial for all sizes. The hybrid nature of scheduling in both methods creates significant scheduling overheads commensurate with the analytical model. One promising sign, however, is that they seem to get closer to the serial implementation for larger sizes. This might tend to the performance benefits seen in the two-stage column block factorization algorithm for extremely large data sets.

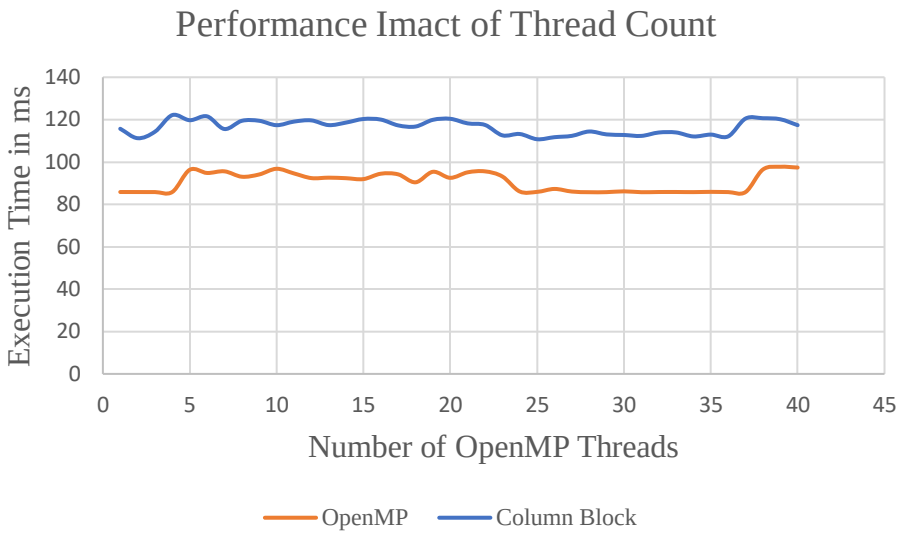

Figure 2

Figure 2. shows the performance variance of the OpenMP codes when the number of CPU threads used are changed. The limited “for” loop parallelization makes the code performance less sensitive to the number of threads used.

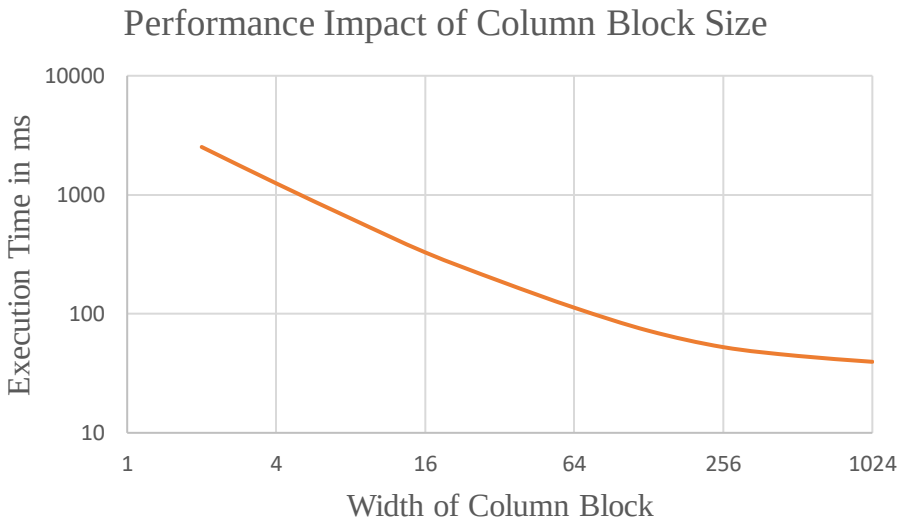

Figure 3

Figure 3. shows the variation of the column block implementation over a range of block sizes. As the number of blocks increase, the number of kernels being launched decreases. This reduces the time spent offloading control and data onto the GPU and speeds up the execution.

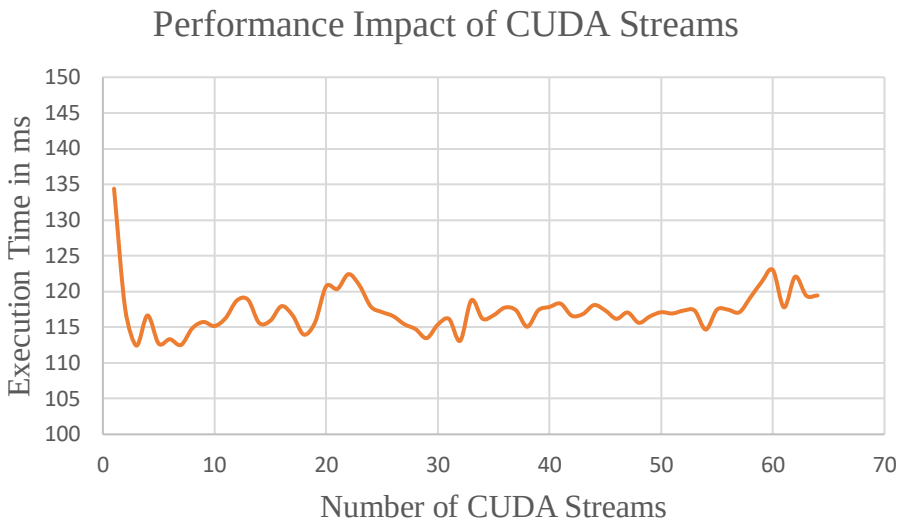

Figure 4

Figure 4. compares the impact of the number of CUDA streams used on kernel performance [10]. The GPU has a single kernel engine. All the streams used in the code will queue their kernels in this engine for scheduling. The GPU schedules operations in the default kernel serially while those in non-default streams, such as those used here, are scheduled concurrently if resources are available. Since the concept of streams is tied deeply with the hardware architecture, there has to be an upper limit before performance improvements stop. Figure 4 shows that performance improves on average till the number of streams used is 16. Beyond that the execution speeds are mostly higher than when number of streams is less than 16. This observation is similar to one made by a Stack Overflow thread [10].

5. Deliverables:

5.1. Implementation Files

The project source files are in `repo759/FinalProject759/` where repo759 is the repository base directory. Each implementation has a source file corresponding to it and three header files that contain declarations of the functions uses. In addition, a utility file defines functions used across the various implementations such as matrix print, result validation, and CUDA block size calculation. The files that are a part of the repository are

- `luFactorizaion.py (SciPy Implementation)`
- `luFactorization.hh and luFactorization.cuh (Function Declarations)`
- `luFactorizationSerial.cc (Serial)`
- `luFactorizationOMP.cc (OpenMP)`
- `luFactorizationCUDA.cu (CUDA)`
- `luFactorizationHybrid.cu (OpenMP + CUDA)`
- `columnBlockFactorization.cuh and columnBlockFactorization.cu (Column Block)`
- `main.cu (main function that invokes each of the above)`
- `utils.cuh and utils.cu (common utilities)`

5.2. Build Files

The input matrices are generated in main() function and reused across all implementations to facilitate result comparison. The repository also contains a `Makefile` and four slurm scripts that build the source code and run different tests. Running any of the slurm scripts builds the code and launches runs specified below

- `Makefile`
- `luFactorization.sh (To run scaling analysis for all implementations over different dataset sizes)`
- `threadOptimization.sh (To run all implementations for a matrix of length 1024 over different OpenMP thread counts)`
- `blockWidthOptimization.sh (To run column block implementation for a matrix of size 1024, 25 threads over a range of block widths)`
- `streamOptimization.sh (To run all implementations for a matrix of length 1024, 25 threads over a range of CUDA stream counts)`

6. Conclusions and Future Work

The CUDA version of the LU Factorization algorithm produces the best speedup amongst all while the column block version that uses both CUDA and OpenMP shows promise at very high matrix sizes. This is because the parallelizable CPU code is not very intensive while offloading the entire computation to CUDA results in much better utilization of resources. While Wu et al demonstrate the benefits of their approach at large data sizes and across multiple processors, their approach scales poorly for smaller sizes. LU Factorization is also an algorithm that has been in focus for decades. Multiple implementations have been proposed and libraries such as BLAS and LAPACK are highly optimized to exploit any possible performance benefit. While this project aims to be an independent study into learning how to implement algorithms in different frameworks, it does present further optimization opportunities. The “for” loops can be optimized further, the OpenMP threads can be restructured to reduce coherence misses, to name a few. The number of kernels can also be reduced in the CUDA implementations by clubbing together the L and U computations into one kernel, but this introduces the need for more complex synchronization. In conclusion, the project served as an excellent opportunity to leverage the ME759 material and played around with various aspects of parallel computing to improve proficiency. Additionally, it allowed exploring the effects of using too many CUDA streams on code performance. This project alone is insufficient to identify the optimal number of CUDA streams and further study needs to be conducted.

The project draws on various concepts taught over the semester. It uses OpenMP and CUDA to parallelize sections of the code to run them on heterogenous systems. It also uses concepts such as unified memory and CUDA streams to implement the hybrid variants. Finally, the project also led to some optimizations on a naïve solution that drew on suggestions to improve efficiency that were presented in several lectures.

References

[1] G. Tan, C. Shui, Y. Wang, X. Yu and Y. Yan, "Optimizing the LINPACK Algorithm for Large-Scale PCIe-Based CPU-GPU Heterogeneous Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2367-2380, September 2021.

[2] R. Wu and X. Xie, "Two-Stage Column Block Parallel LU Factorization Algorithm," *IEEE Access*, vol. 8, pp. 2645-2655, 2020.

[3] I. Lankham, B. Nachtergale and A. Schilling, "LU-Factorization," 12 March 2007. [Online]. Available: https://www.math.ucdavis.edu/~anne/WQ2007/mat67-Ln-LU_Factorization.pdf.

[4] W. H. Press, P. A. Teukolsy, W. T. Vetterling and B. P. Flannery, Numerical Recipes: The Art of Scientific Computing, Third Edition, Cambridge University Press, 2007.

[5] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, pp. 33-38, 2008.

[6] M. S. Bin Althaf and D. A. Wood, "LogCA: A High-Level Performance Model for Hardware Accelerators," in *Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[7] L. Dagum and R. Menon, OpenMP: An Industry Standard API for Shared-Memory Programming, IEEE, 1998.

[8] Nickolls, Jon, Buck, Ian, Garland, Michael, and Skadron, Kevin, Scalable Parallel Programming with CUDA, vol. 6, ACM Queue,, 2008, pp. 40-53.

[9] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov,, SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python, vol. 17, Nature Methods, 2020, pp. 261-272.

[10] M. Ebersole and M. Harris, "Stack Overflow," 3 October 2012. [Online]. Available: https://stackoverflow.com/questions/3565793/is-there-a-maximum-number-of-streams-in-cuda.