Technical Report 2014-02

Performance Analysis of CULA on different NVIDIA GPU Architectures

Prateek Gupta

May 20, 2014

Table of Contents:

I. **Introduction**:

I. **Primary Objective**: The objective is to undertake an in-depth study of the FLOP rate performance analysis and timing scaling analysis of the CULA Dense R17 (with CUDA 5.5) functionalities: (i) GESV (ii) GETRF (iii) GBTRF on different NVIDIA CUDA-enabled GPU architectures: (i) NVIDIA Tesla C2050, (ii) NVIDIA GeForce GTX 680 (iii) NVIDIA Tesla K40C. The functionalities are implemented using CUDA programming and CULA framework functions for both single precision and double precision.

II. **Motivation**: The CULA is a GPU accelerated linear algebra library that utilizes the NVIDIA CUDA parallel computing architecture to dramatically improve the computation speed of sophisticated mathematics. CULA is an implementation of the Linear Algebra PACKage (LAPACK) interface for CUDA enabled NVIDIA GPU. The CULA is a next generation linear algebra package that uses the GPU as a co-processor to achieve speedups over existing linear algebra packages. As there is no matrix inversion operation in CUBLAS, we have to ask help from CULA. It is built on **NVIDIA CUDA** and **NVIDIA CUBLAS** and uses the **Intel® Math Kernel Library (MKL)** internally. The performance and actual speed ups of CULA depends heavily on the algorithm and the size of the data set. Additionally, the performance also varies with the GPU memory available for performing the computation, which varies with different flavors of NVIDIA GPU cards as claimed by the CULA experts in one of their blogs (). This feature can be potentially explored by using the device interface model of CULA. So, the performance analysis in terms of GFLOPS can be done on Fermi, Tesla as well as Kepler Architectures. This study is important as it will reflect the advantages of using a particular architecture for getting optimized performance for our ***Spike GPU solver*** [4,5]. For example, the performance of the first Kepler card, GeForce GTX 680 which has a downside that it has comparatively low performance numbers for double precision as compared to other traditional chips but has very good single precision performance. So, many other such interesting features will come out into play when CULA will be ported onto these architectures.

   The study stresses on finding the GFLOP performance of three different memory intensive linear algebra CULA Dense functionalities with high degree of usage in different algorithms on different types of NVIDIA GPU architectures like Fermi and Kepler.

III. **Background**: The primary reference of this study is [1]. CULA's standard Dense edition implements a much larger set of functions from LAPACK as shown in Table 1. Building from the information presented, this study does a FLOP rate performance analysis of compute / memory intensive linear algebra functionalities. In this study, the code is then further optimized by pinning the host memory and the FLOP rate analysis after optimization is presented. This optimization tries to mitigate the performance overhead in data transfer between the host and device.

IV. **Independent Study Overview**: This study exercises different linear algebra functions in single and double precision. The performance analysis will involve running different applications on CULA dense R17 and CUDA 5.5 on different NVIDIA GPU cards with different architectural specifications as shown in Table1, which are as follows:

- Dense General Matrix Solve (using LU decomposition)- DGESV, SGESV
- Dense General Matrix triangular factorization- SGETRF, DGETRF
- Dense Banded Matrix Triangular factorization- SGBTRF, DGBTRF

The number of floating point operations for factorization and solving is 0.67 times N^3 and 2 times N^2 respectively.

Table1: Architectural Specifications of Tesla C2050, GeForce GTX680, Tesla K40C

S.No.	Features	Tesla C2050 (Fermi Architecture)	GeForce GTX680 (Kepler Architecture)	Tesla K40C (Kepler Architecture)
1	CUDA cores/ (SMs/SMXs)	448/14	1536/8	2880/15
2	Memory	3072 MB (GDDR5)	2048 MB (GDDR5)	12288 MB (GDDR5)
3	Fab (nm)/Code Name	40/ GF 100	28/ GK 104	28/ GK 110B
4	Peak GFLOPS (FMA-Double precision)	515.2	3090.43	1430
5	Core Configuration (Unified shaders: Texture Mapping units: Render output units)	448:56:48	1536:128:32	2880:240:48
6	Bandwidth (GB/s)	144	192.2	288

V. CULA Libraries: CULA has two libraries as follows:-

I. CULA Dense: A GPU-accelerated implementation of dense linear algebra routines. Following are the linear equation functionalities supported by CULA Dense:

Matrix Type	Operation	S	C	D	Z
General	Factorize and solve	SGESV	CGESV	DGESV	ZGESV
	Factorize and solve with iterative refinement			DSGESV	ZCGESV
	LU factorization				
	Solve using LU factorization	SGETRF	CGETRF	DGETRF	ZGETRF
	Invert using LU factorization	SGETRS	CGETRS	DGETRS	ZGETRS
		SGETRI	CGETRI	DGETRI	ZGETRI
Positive Definite	Factorize and solve	SPOSV	CPOSV	DPOSV	ZPOSV
	Cholesky Factorization	SPOTRF	CPOTRF	DPOTRF	ZPOTRF
Triangular	Invert triangular matrix	STRTRI	CTRTRI	DTRTRI	ZTRTRI
	Solve triangular system	STRTRS	CTRTRS	DTRTRS	ZTRTRS
Banded	LU factorization	SGBTRF	CGBTRF	DGBTRF	ZGBTRF
Positive Definite Banded	Cholesky factorization	SPBTRF	CPBTRF	DPBTRF	ZPBTRF

II. CULA Sparse: A GPU-accelerated library for linear algebra that provides iterative solvers for sparse systems. Since, in this study, the focus is on CULA Dense linear functionalities so, CULA Sparse functionalities are not discussed here.

II. Performance Analysis Framework

I. Programming Considerations:

Matrix Storage: When providing data to CULA routines, it is important to consider that the data is stored in column-major order in memory. Column-major ordering is the opposite of the row-major ordering because elements of the matrix are instead stored by column, rather than by row. In this storage scheme, elements of a column are contiguous in memory, while elements of a row are not.

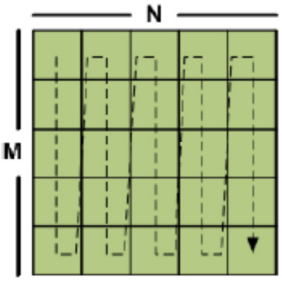

Figure1: A column-major ordered matrix. Elements are stored in memory in the order shown by the arrow.

Performing a transpose on the row-major data will convert it to column-major and vice-versa. For column-major data, the leading dimension the leading dimension is equal to the height of a column, or equivalently, the number of rows in the matrix. This is the height of the matrix as allocated and may be larger than the matrix used in the computation.

Performance Optimization: CULA is specifically designed to leverage the massively parallel computational resources of the GPU, with a particular focus on large problems whose execution on a standard CPU is too time consuming.

- **Problem Size-** As a general rule, applying CULA for larger problems will maximize performance gains with respect to other computational linear algebra packages [2]. The maximum problem size is constrained by the data type in use and the maximum GPU memory. For example, the maximum size for a problem that uses double-precision complex data is roughly one fourth of the maximum problem size of a single-precision problem for the same matrix dimensions, since the size of these data types differ by a factor of four. This can also be seen in the 'Results and Analysis' section of this report.
- **Accuracy Requirements-** CULA offers both single and double-precision floating point support for its included routines. While the latest NVIDIA GPU hardware offers support for both of these data types, it should be noted that current NVIDIA GPU hardware performs best when operating on single-precision data [2] i.e. additional performance can be achieved at the cost of accuracy through the use of single-precision routines.

- **Device Interface:** The Device interface follows the standards set forth in the NVIDIA CUBLAS package. In this interface, it is required to allocate and populate GPU memory and then call CULA functions to operate on that memory. Memory allocation is handled via **cudaMalloc** and **cudaFree**, available in the CUDA toolkit. While using pitched memory, it is our responsibility to ensure that their allocations are appropriate.

CULA's standard Dense version provides specialized allocation functions that pitch data to the optimal size for CULA. These functions are **culaDeviceMalloc()** and **culaDeviceFree()**, found in the **cula_device.h** header of CULA Dense.

- **Leading Dimension:** All LAPACK matrices are specified as a pointer and a "leading dimension" parameter. The leading dimension describes the allocated size of the matrix, which may be equal to or larger than the actual matrix height. Thus if a matrix input is described as size "(LDA, N)" it simply means that the storage for the matrix is at least LDA x N in size. The section of that array that contains valid data will be described by other parameters, often M and N.

II. Analysis Structure and Framework Functions:

Framework Function	Meaning
culaInitialize()	Initializes CULA; must be called before using any other function. Some functions have an exception to this rule like culaGetDeviceCount(), culaSelectDevice(), and version query functions.
culaShutdown()	Shuts down CULA
culaGetStatusString()	Associates a culaStatus enum with a readable error string.
culaGetStatusAsString()	Returns the culaStatus name as a string.
culaGetErrorInfo()	This function is used to provide extended functionality that LAPACK's info parameter typically provides.
culaGetErrorInfoString()	Associates a culaStatus and culaInfo with a readable error string
culaFreeBuffers	Releases any memory buffers stored internally by CULA
culaGetVersion	Reports the version number of CULA
Framework Function	Meaning
culaGetCudaMinimumVersion	Reports the CUDA_VERSION that the running version of CULA was compiled against, which indicates the minimum version of CUDA that is required to use this library
culaGetCudaRuntimeVersion()	Reports the version of the CUDA runtime that the operating system linked against when the program was loaded.
culaGetCudaDriverVersion()	Reports the version of the CUDA driver installed on the system.
culaGetCublasMinimumVersion()	Reports the CUBLAS_VERSION that the running version of CULA was compiled against, which indicates the minimum version of CUBLAS that is required to use this library.
culaGetCublasRuntimeVersion()	Reports the version of the CUBLAS runtime that operating system linked against when the program was loaded.
culaGetDeviceCount()	Reports the number of GPU devices Can be called before culaInitialize().
culaGetExecutingDevice()	Reports the id of the GPU device executing CULA.
culaGetDeviceInfo()	Prints information to a buffer about a specified device.
culaGetOptimalPitch()	Calculates a pitch that is optimal for CULA when using the device interface.
culaDeviceMalloc()	Allocates memory on the device in a pitch that is optimal for CULA.
culaDeviceFree()	Frees memory that has been allocated with culaDeviceMalloc.

III. Configuring the Environment:

The first step in this process is to set up environment variables so that your build scripts can infer the location of CULA. Add the following lines to the .bashrc:
export CULA_ROOT=/usr/local/cula
export CULA_INC_PATH=$CULA_ROOT/include
export CULA_LIB_PATH_32=$CULA_ROOT/lib
export CULA_LIB_PATH_64=$CULA_ROOT/lib64
(where CULA_ROOT is customized to the location where CULA is installed)
After setting environment variables, CULA can be built by configuring the build scripts (**module load culadense**) and by using makefile.

III. CULA Dense Functionalities

I. Benchmarking CULA R17 vs Intel MKL on Euler99 (Tesla K40C):

[pkgupta3@euler99 benchmark]$ ./benchmark
Initializing CULA...
Initializing MKL...
Benchmarking the following functions:

SGEQRF
SGETRF
SGELS
SGGLSE
SGESV
SGESVD
SSYEV
DGEQRF
DGETRF
DGELS
DGGLSE
DGESV
DGESVD
DSYEV

-- SGEQRF Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.19	0.31	1.6012
5120	0.28	0.55	1.9577
6144	0.43	0.92	2.1636
7168	0.61	1.42	2.3283
8192	0.83	2.18	2.6264

-- SGETRF Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.12	0.16	1.2944
5120	0.19	0.28	1.4750
6144	0.27	0.48	1.7634
7168	0.39	0.72	1.8318
8192	0.53	1.10	2.0923

-- SGELS Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.28	0.35	1.2325
5120	0.43	0.62	1.4481
6144	0.63	1.02	1.6208
7168	0.88	1.56	1.7674
8192	1.17	2.26	1.9336

-- SGGLSE Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.31	1.55	5.0503
5120	0.48	2.51	5.2651
6144	0.69	3.74	5.4573
7168	0.94	5.25	5.6026
8192	1.24	7.16	5.7733

-- SGESV Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.13	0.16	1.2737
5120	0.20	0.28	1.4250
6144	0.29	0.49	1.7278
7168	0.41	0.74	1.7983
8192	0.55	1.11	2.0338

-- SGESVD Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	16.80	26.05	1.5508
5120	27.29	43.68	1.6005
6144	41.12	67.82	1.6491
7168	58.05	113.88	1.9619
8192	77.90	171.80	2.2053

-- SSYEV Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	1.70	1.36	0.7978
5120	2.67	2.48	0.9272
6144	3.97	5.02	1.2641
7168	5.62	7.21	1.2819
8192	7.51	10.02	1.3341

-- DGEQRF Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.36	0.68	1.9019
5120	0.49	1.06	2.1727
6144	0.74	1.86	2.5284
7168	1.05	2.74	2.6026
8192	1.48	4.85	3.2855

-- DGETRF Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.20	0.34	1.6807
5120	0.33	0.55	1.6416
6144	0.47	1.03	2.1835
7168	0.67	1.42	2.1193
8192	0.92	3.00	3.2454

-- DGELS Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.49	0.64	1.3119
5120	0.76	1.14	1.4957
6144	1.11	1.91	1.7229
7168	1.52	2.92	1.9233
8192	2.07	4.23	2.0479

-- DGGLSE Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.53	3.27	6.1585
5120	0.79	5.07	6.4347
6144	1.15	7.76	6.7629
7168	1.57	10.70	6.7944
8192	2.13	15.31	7.2020

-- DGESV Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.21	0.36	1.7184
5120	0.33	0.56	1.6830
6144	0.49	0.97	1.9805
7168	0.69	1.43	2.0786
8192	0.96	2.97	3.0903

-- DGESVD Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	22.10	31.46	1.4231
5120	37.80	54.24	1.4352
6144	58.44	90.94	1.5561
7168	86.54	142.55	1.6471
8192	122.94	241.05	1.9607

-- DSYEV Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	2.54	3.49	1.3715
5120	3.92	5.33	1.3602
6144	5.79	8.27	1.4278
7168	7.87	11.29	1.4341
8192	10.60	19.00	1.7922
8193			

II. Benchmarking CULA R18 vs Intel MKL on Euler99 (Tesla K40C)

[pkgupta3@euler99 benchmark]$ ./benchmark
Initializing CULA...
Initializing MKL...

Benchmarking the following functions:

SGEQRF
SGETRF
SGELS
SGGLSE
SGESV
SGESVD
SSYEV
DGEQRF
DGETRF
DGELS
DGGLSE
DGESV
DGESVD
DSYEV

-- SGEQRF Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.22	0.37	1.6909
5120	0.30	0.69	2.3298
6144	0.44	1.20	2.7126
7168	0.63	1.73	2.7554
8192	0.85	2.82	3.3281

-- SGETRF Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.12	0.23	1.9259
5120	0.19	0.43	2.2097
6144	0.27	0.81	2.9622
7168	0.40	1.05	2.6517
8192	0.52	1.35	2.6126

-- SGELS Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.29	0.53	1.8400
5120	0.43	0.62	1.4372
6144	0.63	1.03	1.6335
7168	0.88	1.56	1.7679
8192	1.17	2.26	1.9355

-- SGGLSE Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.32	1.55	4.8889
5120	0.48	2.51	5.1851
6144	0.68	3.77	5.5439
7168	0.93	5.25	5.6481
8192	1.23	7.18	5.8427

-- SGESV Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.12	0.16	1.2847
5120	0.19	0.29	1.4710
6144	0.28	0.50	1.7773
7168	0.39	0.73	1.8581
8192	0.53	1.15	2.1701

-- SGESVD Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	16.89	25.82	1.5289
5120	27.64	44.77	1.6196
6144	41.47	68.58	1.6535
7168	58.70	117.67	2.0046
8192	78.55	177.53	2.2600

-- SSYEV Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	1.65	1.42	0.8588
5120	2.68	2.49	0.9283
6144	4.08	5.06	1.2392
7168	5.41	6.97	1.2889
8192	7.40	10.10	1.3662

-- DGEQRF Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.28	0.69	2.5060
5120	0.46	1.05	2.2844
6144	0.71	1.89	2.6695
7168	1.02	3.30	3.2179
8192	1.44	4.86	3.3690

-- DGETRF Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.20	0.34	1.7257
5120	0.32	0.55	1.7093
6144	0.49	1.07	2.2083
7168	0.67	1.45	2.1595
8192	0.94	3.04	3.2488

-- DGELS Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.45	0.65	1.4478
5120	0.71	1.15	1.6165
6144	1.06	1.90	1.7986
7168	1.46	2.93	2.0061
8192	2.02	4.23	2.0962

-- DGGLSE Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.47	3.30	7.0021
5120	0.74	5.13	6.9317
6144	1.09	7.69	7.0707
7168	1.51	10.67	7.0741
8192	2.05	15.26	7.4448

-- DGESV Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	0.20	0.34	1.6483
5120	0.34	0.56	1.6341
6144	0.48	1.08	2.2549
7168	0.68	1.46	2.1433
8192	0.94	3.09	3.2924

-- DGESVD Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	22.23	31.35	1.4104
5120	38.00	54.18	1.4257
6144	58.57	91.00	1.5537
7168	86.57	142.19	1.6425
8192	122.97	241.61	1.9649

-- DSYEV Benchmark --

Size	CULA (s)	MKL (s)	Speedup
4096	2.43	3.55	1.4611
5120	3.84	5.43	1.4129
6144	5.69	8.30	1.4583
7168	7.87	11.49	1.4594
8192	10.30	19.07	1.8512

III. Implementation with CulaDgesv, CulaSgesv:

GESV computes the solution to a real system of linear equations A * X = B, where A is an N-by-N matrix and X and B are N-by-NRHS matrices. The LU decomposition with partial pivoting and row interchanges is used to factor A as A = P * L * U, where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. The factored form of A is then used to solve the system of equations A * X = B.

Calling Subroutine:
culaDeviceSgesv(int n, int nrhs, culaDeviceFloat* a, int lda, culaDeviceInt* ipiv, culaDeviceFloat* b, int ldb);

culaDeviceDgesv(int n, int nrhs, culaDeviceDouble* a, int lda, culaDeviceInt* ipiv, culaDeviceDouble* b, int ldb);
where, **n**: The number of linear equations, i.e., the order of the matrix A. N >= 0.
Nrhs: The number of right hand sides, i.e., the number of columns of the matrix B. NRHS >= 0.
a: On entry, the N-by-N coefficient matrix A. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.

lda: The leading dimension of the array A. LDA >= max(1,N).

ipiv: The pivot indices that define the permutation matrix P; row i of the matrix was interchanged with row IPIV(i).

b: On entry, the N-by-NRHS matrix of right hand side matrix B.

On exit, if culaNoError is returned, the N-by-NRHS solution matrix X.
ldb: The leading dimension of the array B. LDB >= max(1,N).

Note: Results are reported in the Section 4.

IV. Implementation with CulaDgetrf, CulaSgetrf:

GETRF computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges. The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

Calling Subroutine:
culaDeviceSgetrf(int m, int n, culaDeviceFloat* a, int lda, culaDeviceInt* ipiv);
culaDeviceDgetrf(int m, int n, culaDeviceDouble* a, int lda, culaDeviceInt* ipiv);

Note: Results are reported in the Section 4.

V. Implementation with CulaSgbtrf, CulaDgbtrf:

GBTRF computes an LU factorization of an m-by-n band matrix A using partial pivoting with row interchanges. This is the blocked version of the algorithm, calling Level 3 BLAS.

Calling subroutine:
culaDeviceSgbtrf(int m, int n, int kl, int ku, culaDeviceFloat* a, int lda, culaInt* ipiv);
culaDeviceDgbtrf(int m, int n, int kl, int ku, culaDeviceDouble* a, int lda, culaInt* ipiv);

where, m: The number of rows of the matrix A. M >= 0.

n: The number of columns of the matrix A. N >= 0.

kl: The number of subdiagonals within the band of A. KL >= 0.

ku: The number of superdiagonals within the band of A. KU >= 0.
ab: On entry, the matrix A in band storage, in rows KL+1 to 2*KL+KU+1; rows 1 to KL of the array need not be set. The j-th column of A is stored in the j-th column of the array AB as follows:
AB(kl+ku+1+i-j,j) = A(i, j) for max(1,j-ku)<=i<=min(m, j+kl)
On exit, details of the factorization: U is stored as an upper triangular band matrix with KL+KU super-diagonals in rows 1 to KL+KU+1, and the multipliers used during the factorization are stored in rows KL+KU+2 to 2*KL+KU+1.
ldab: The leading dimension of the array AB. LDAB >= 2*KL+KU+1.
Ipiv: The pivot indices; for 1 <= i <= min(M,N), row i of the matrix was interchanged with row IPIV(i).

Note: Results are reported in the Section 4.

4. Results and Analysis:

I. Timing Scaling Analysis: Timing Scaling Analysis graphs are plotted for GESV, GETRF and GBTRF on GeForce GTX680, Tesla C2050 and Tesla K40C with matrix dimensions on x-axis and time taken to perform the functionalities in milliseconds on y-axis. The maximum matrix dimension is 16000.

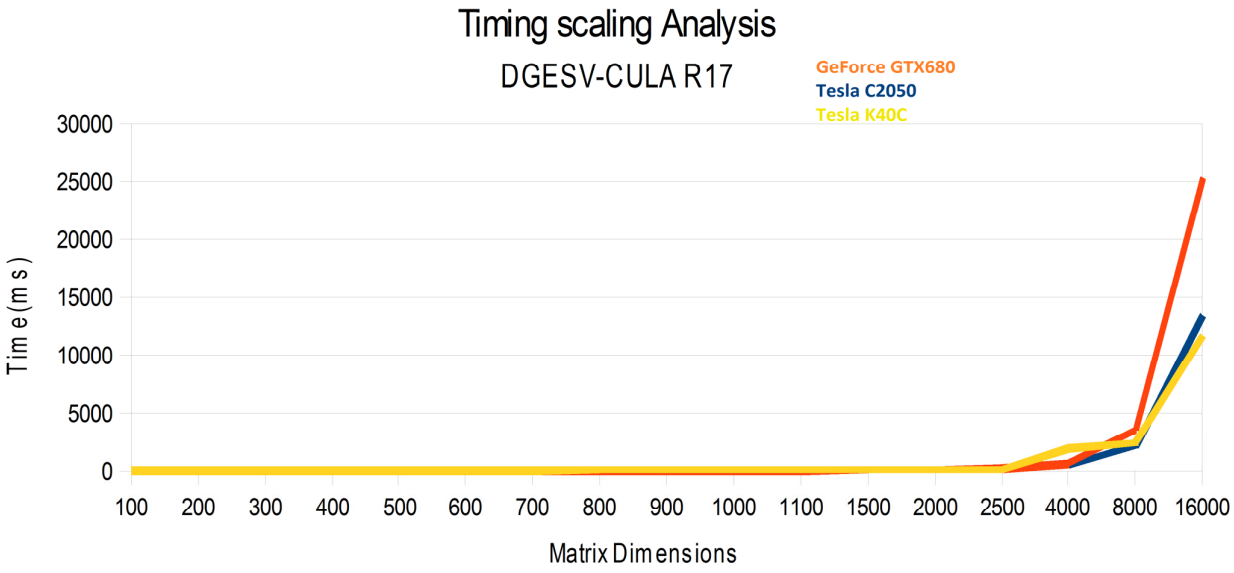

Figure1: Timing Scaling Analysis for DGESV

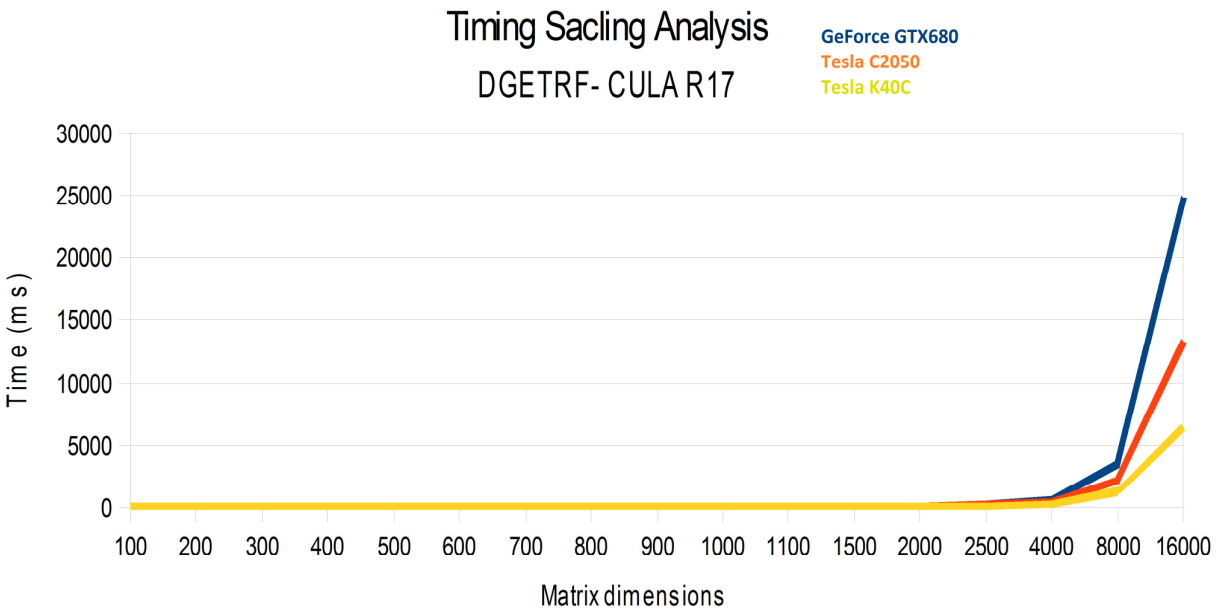

Figure2: Timing Scaling Analysis for DGETRF

II. FLOP rate Performance Scaling Analysis:
Figure3-4 below show the Performance Scaling Analysis of CulaDgesv and CulaSgesv on Tesla K40C, Tesla C2050 and GeForce GTX 680 GPU cards for matrix dimension of 16000. It can be observed that as the matrix dimension is increasing, CULA is performing better. Overall, Tesla K40C is the better performer for large matrix dimensions because of more GPU memory available.

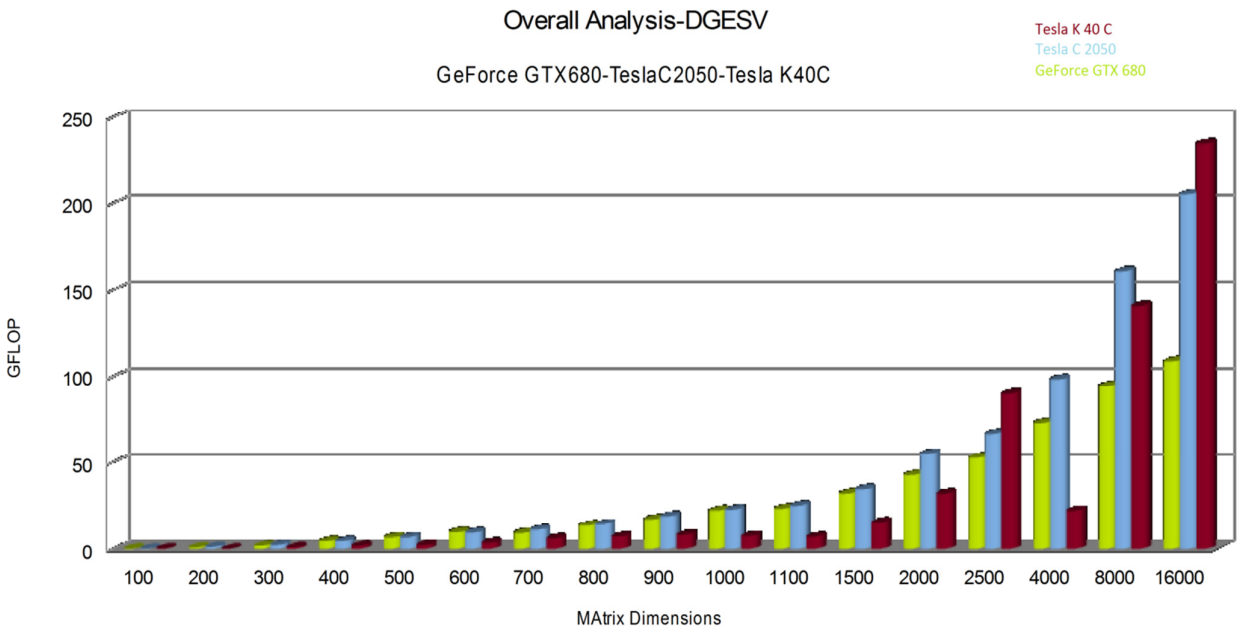

Figure3: Performance Scaling Analysis for DGESV

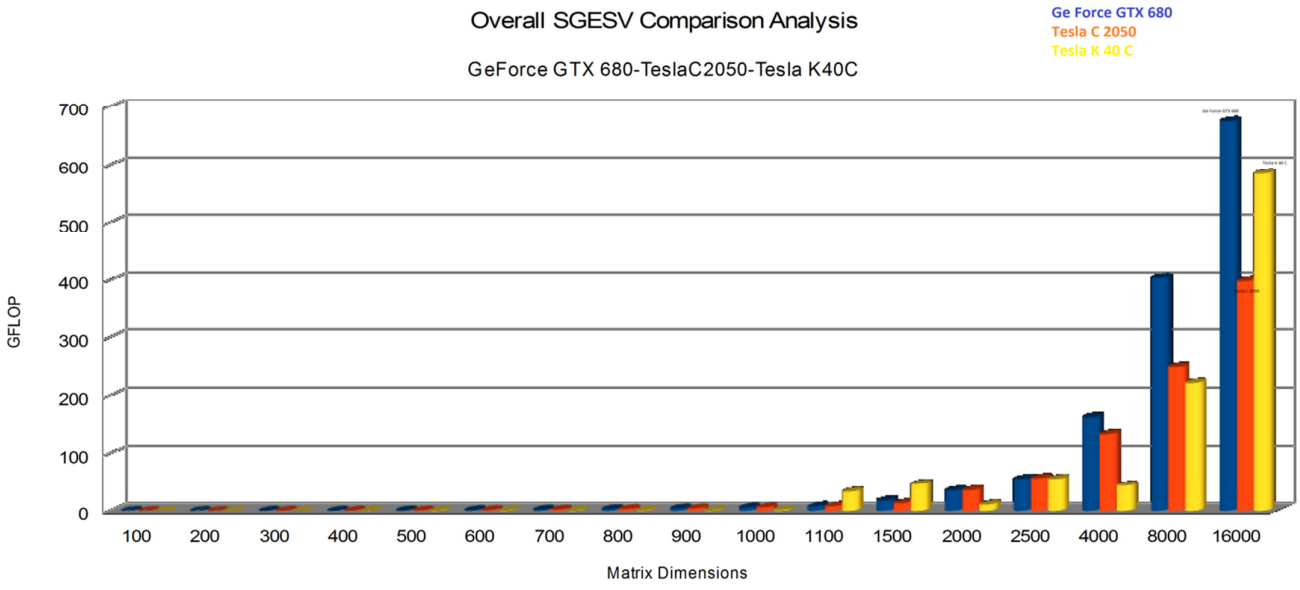

Figure4: Performance Scaling Analysis for SGESV

The GFLOP performance is measured for double precision and single precision general matrices factorization and solve on Tesla K40C, Tesla C2050 and GeForce GTX 680 GPU cards, with maximum matrix dimension of 16000. It is observed that Tesla K40C is the best performer touching around 430 GFlops for double precision and around 600-700 GFlops for maximum matrix dimension of 16000.

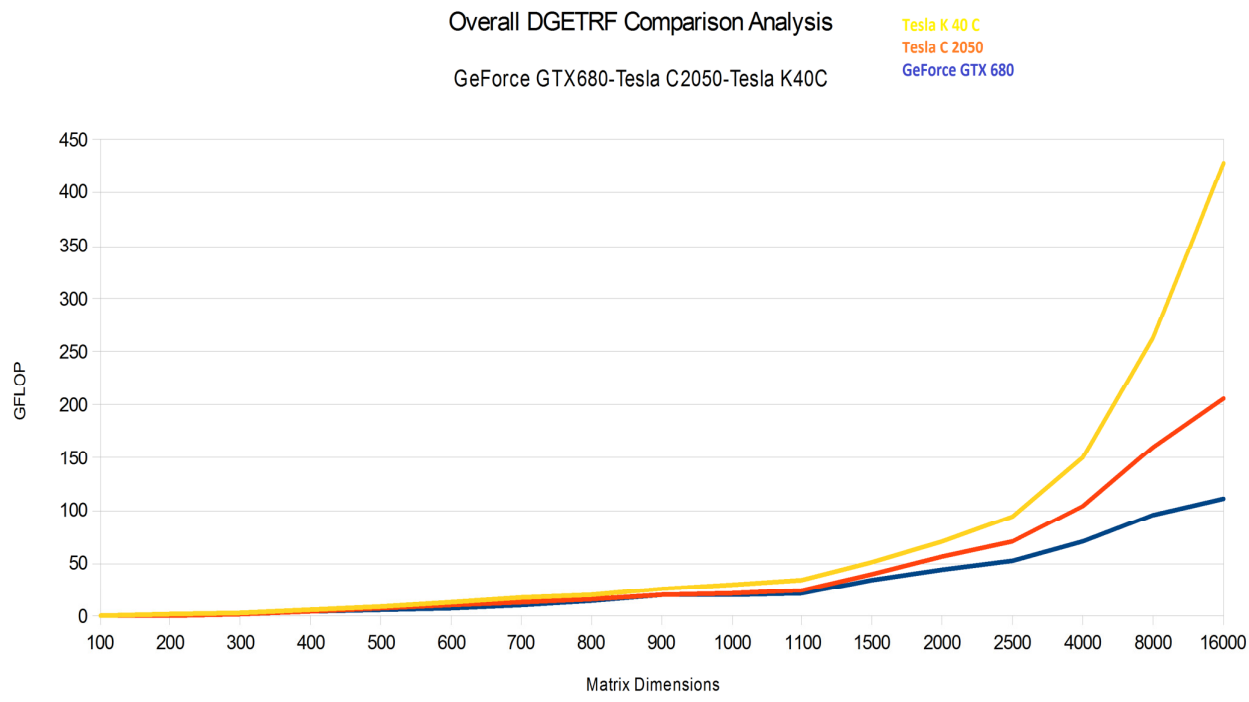

Figure5: Performance Scaling Analysis for DGETRF

Figure 5 above shows the GFLOP performance of double precision general matrices triangular factorization on Tesla K40C, Tesla C2050 and GeForce GTX 680 GPU cards, with maximum matrix dimension of 16000. It is observed that Tesla K40C is the best performer touching around 430 GFlops for matrix dimension of 16000.

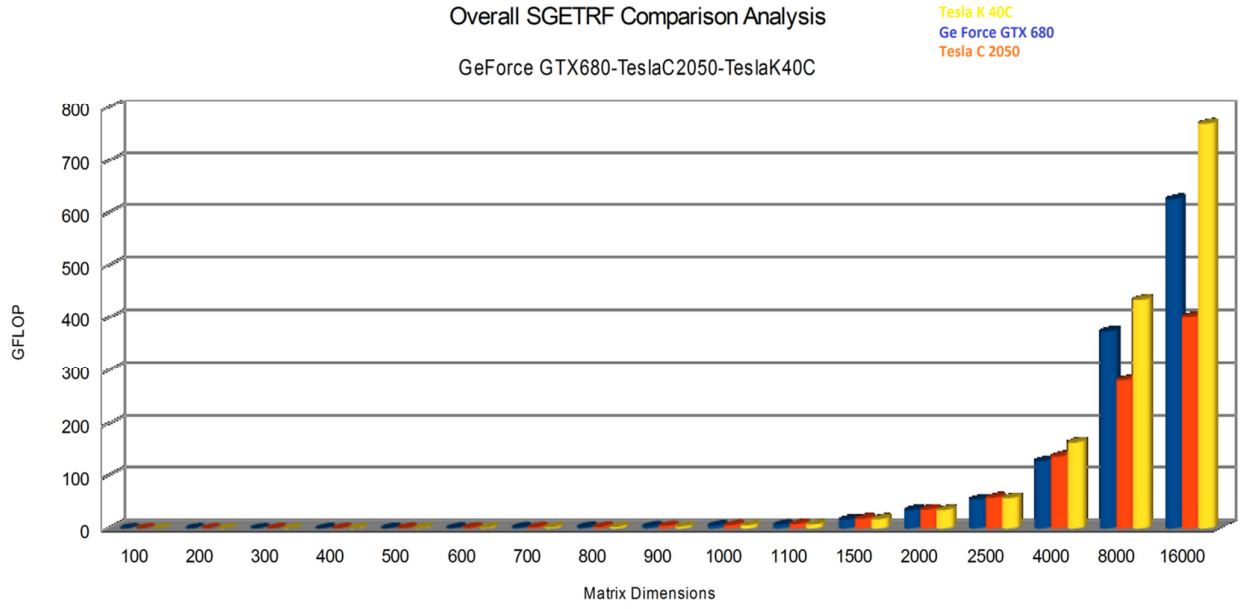

Figure6: Performance Scaling Analysis for SGETRF

Figure 6 above shows the GFLOP performance of single precision general matrices triangular factorization on Tesla K40C, Tesla C2050 and GeForce GTX 680 GPU cards, with maximum matrix dimension of 16000. It is observed that Tesla K40C is the best performer touching around 750 GFlops for matrix dimension of 16000.

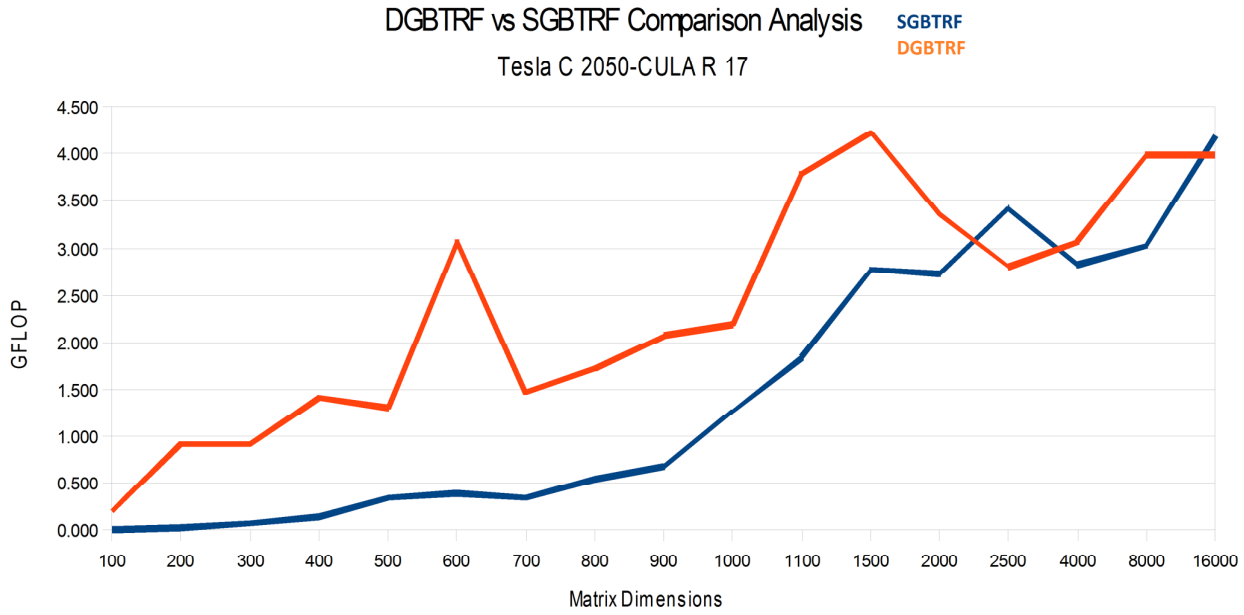

Figure7: Performance Comparison Analysis of DGBTRF vs SGBTRF on Tesla C2050 (b/w = 1/10)

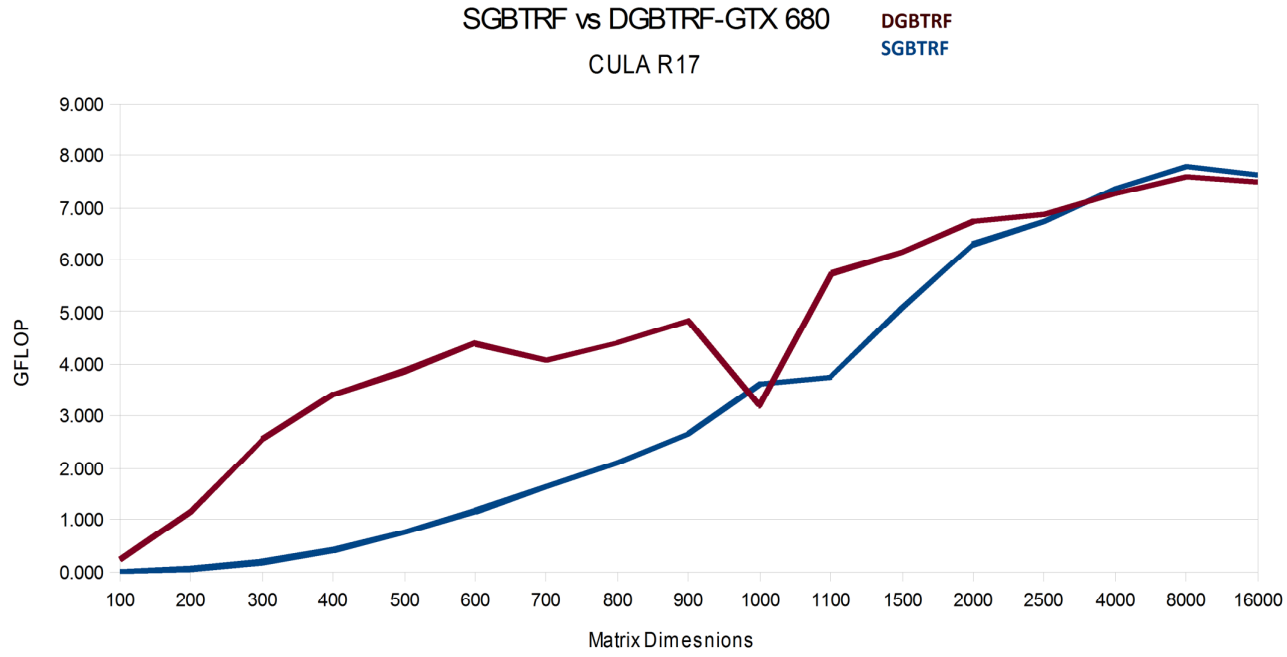

Figure8: Performance Comparison Analysis of DGBTRF vs SGBTRF on GeForce GTX680 (b/w = 1/10)

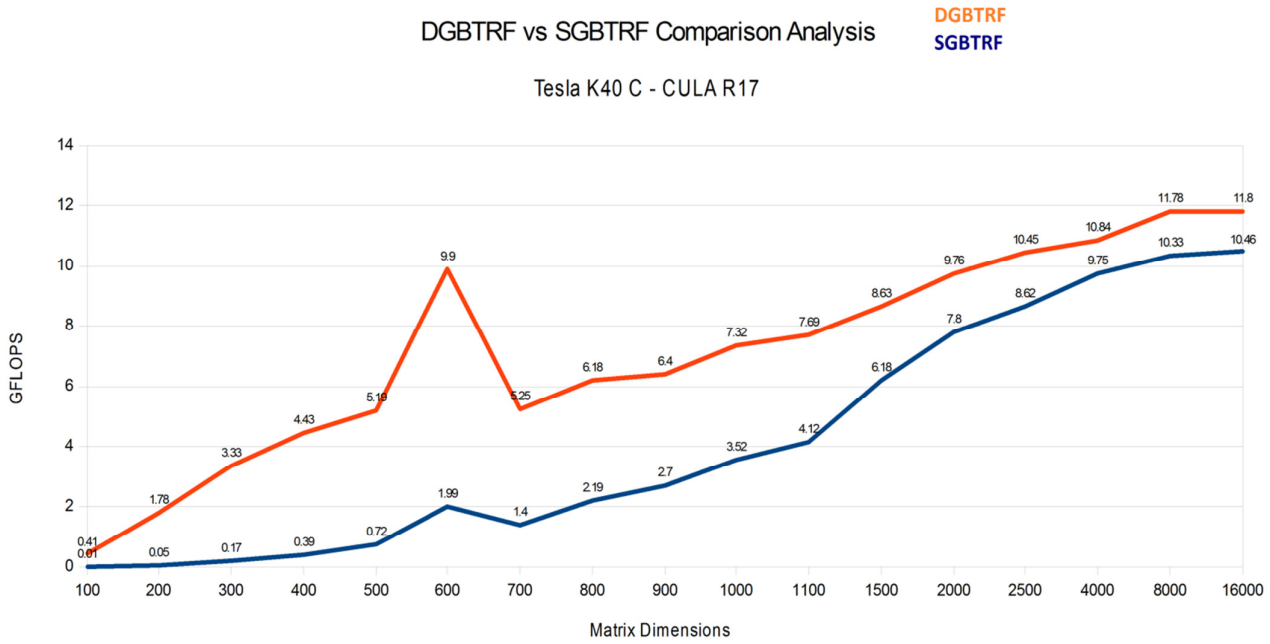

Figure9: Performance Comparison Analysis of DGBTRF vs SGBTRF on Tesla K40C (b/w = 1/10)

Figure 7-9 above show the performance comparison analysis graphs plotted for dense banded triangular factorized matrices with maximum dimension of 16000 and bandwidth = 1/10 on all the three GPU cards. The performance in terms of GFLOP is much lower for banded matrices factorization as compared to general matrices factorization as shown in figure 4-5.

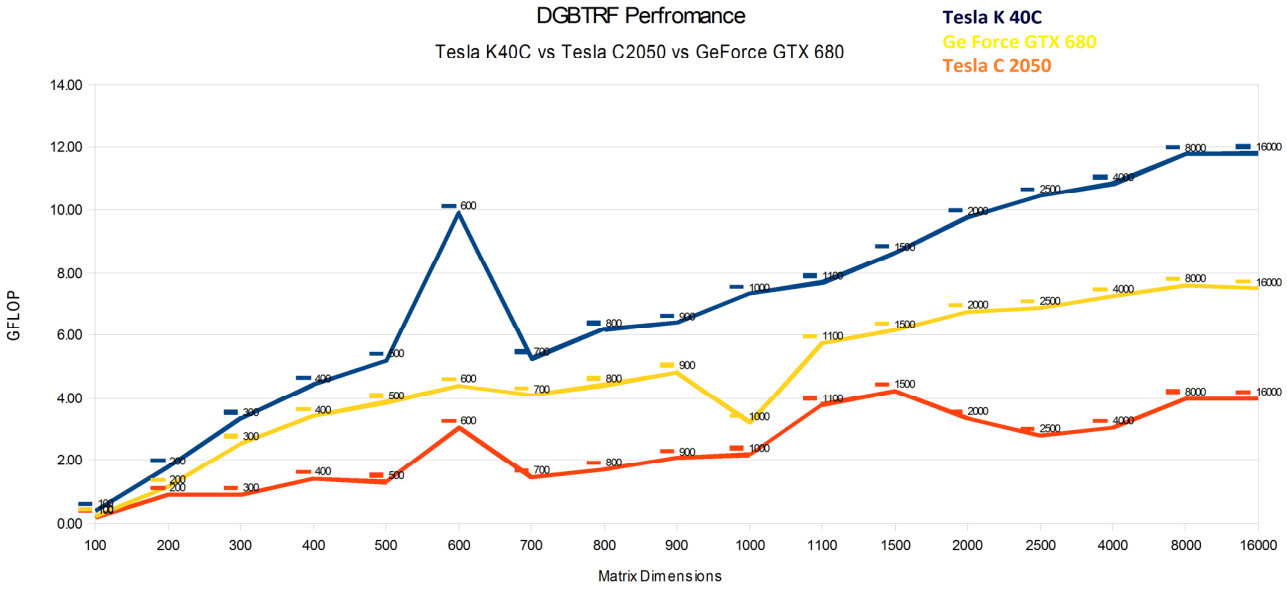

Figure10: Performance Analysis of DGBTRF with bandwidth = 1/10

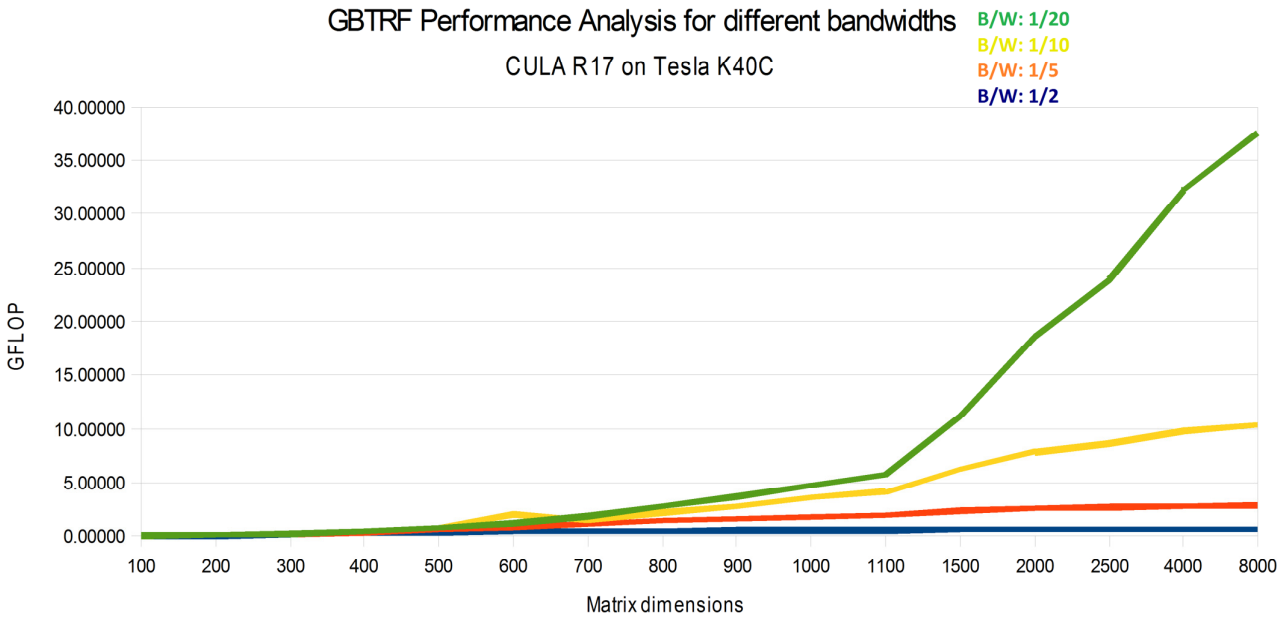

Figure11: Performance Analysis of GBTRF with different bandwidths

Figure 10-11 above are showing the Performance analysis of banded triangular matrices with bandwidth = 1/10 on all three GPU cards and performance analysis with respect to different bandwidths on Tesla K40C respectively. The main two reasons that are responsible for such behavior are cache access variation with changing the bandwidth and synchronization that is managed internally within the subroutine function call.

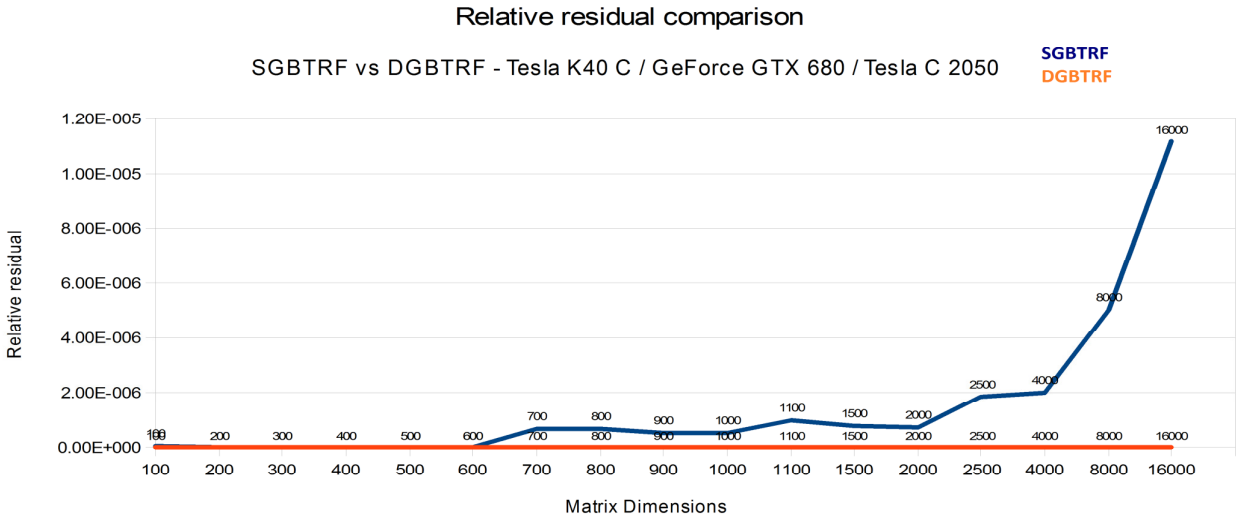

Figure12: Relative residual Analysis of DGBTRF vs SGBTRF (bandwidth = 1/10)

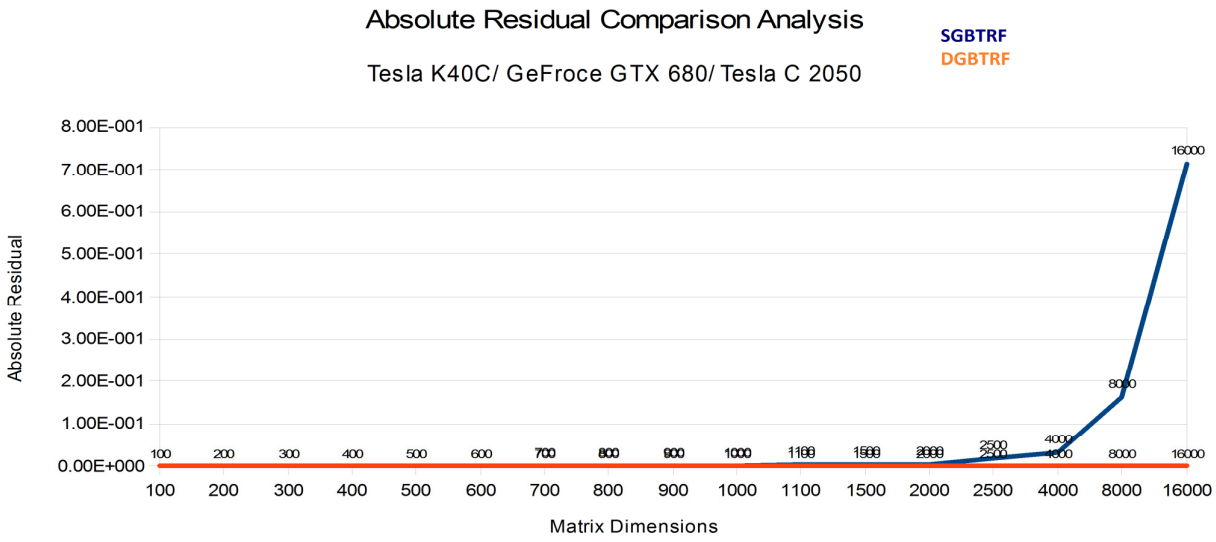

Figure13: Absolute residual Analysis of DGBTRF vs SGBTRF (bandwidth = 1/10)

Figure 12-13 above show the relative and absolute residual comparison analysis of dense banded matrices (single and double precision). The relative residual analysis shows the convergence residual and it can be interpreted from the above results that double precision functionality has much better accuracy relative to the single precision implementation.

5. Conclusion:

The performance of CULA R17 varies with different GPU cards due to the variation in architectural specifications, different GPU memory size and bandwidth. Different linear algebra functionalities were illustrated in this report, which are driven by different algorithms. For all these functionalities, it was observed that CULA gives a better performance for large matrix dimensions and on the device with more GPU memory available. Also, the implementations were optimized by pinning the host memory but it was not very helpful. However, the performance remained similar as compared to non-pinned memory. Also, the benchmarking results show that CULA R18 (requires CUDA 6.0) has better speed-ups over MKL as compared to CULA R17. CUDA 6.0 has a unified memory which manages optimized memory transfer between host and device. The key is that the system automatically migrates data allocated in Unified Memory between host and device so that it looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU.

6. References:

[1] CULA tools: http://www.culatools.com/blog/2010/04/17/11-initial-fermi-performance/

[2] CULA R17 Programmers Guide

[3] CULA R17 Reference Manual

[4] TR-2012-04: “A GPU-based LU factorization of dense banded matrices” A. Li, A. Seidl, D. Negrut.

[5] TR-2011-01: “SPIKE - A Hybrid Algorithm for Large Banded Systems” T. Heyn, D. Negrut.