Performance and accuracy of the matrix multiplication routines : CUBLAS on Nvidia Tesla versus MKL and ATLAS on Intel Nehalem

Philippe Estival, Luc Giraud

A fight for performance and accuracy of the matrix multiplication routines : CUBLAS on Nvidia Tesla versus MKL and ATLAS on Intel Nehalem

Philippe Estival , Luc Giraud

CERFACS, 42 avenue Gustave Coriolis, 31057 Toulouse Cedex, France

Abstract : Scientific computation relies heavily on 64 bits arithmetic. The evolution of the Graphical Processing Units to the status of massively micro-parallel vector units and the improvement of their programmability make them stand as powerfull algebraic coprocessors for many classes of matrix calculus. But on these processors inheriting from architectures dedicated to video processing in the first place, the space for double precision is narrow yet. One building block of dense linear algebra, the GEneralized Matrix Multiply Routine has been considerably accelerated on the GPU. We figure in this paper more details regarding its speed, but first, accuracy.

1. Introduction

Before computing anything fast, one have to compute correctly. If not meeting exactitude as a requirement, and that is the case of floating point standards, then we must know how accurate the computations are. Scientific computation is a very demanding for 64 bits double precision floating point arithmetics, specially when using iterative techniques subjects to propagations of rounding errors along with schemes of high amplitude scales. The Generic Purpose Computation on GPU is very attracting exercise in physics. But on these hardwares, speed nibbled on precision. Introducing more rounding errors in exchange for speed leads to this question, asked by [Goldberg, 1991] : *"Since most floating-point calculations have rounding errors anyway, does it matter if the basic arithmetic operations introduce a little bit more rounding error than necessary"?*

When considering real-time image rendering, the answer is clearly no. Broadening to the spectrum of simulation, we give in this paper, the beginning of an answer for one class of general purpose computation hardware and one operation by investigating the performance and numerical accuracies achieved by the T10-series Nvidia GPUs on simple and double precision floating point matrix-matrix multiplication using CUBLAS, the Nvidia CUDA BLAS implementation.

Regularity and predictability of data access pattern, highly parallel computationnal requirements and reuse of cached data set this function as a good candidate to evaluate the peak GPUs performances.

Exact results will be compared against two widely used BLAS implementation as reference : ATLAS and Intel's Matrix Kernel Library, showing that the accuracy on CUDA-enabled hardware is lower with an order between one and two when compared to a CPU computation in single (32-bits) floating point and double (80-bits extended x86) precision.

2. Parallel matrix multiplication

Be the matrix product $C = A \times B$ of size $N \times N$. The parallel product without per-block computation, produces one thread handling one element for every result of the product. A and B are loaded N times from the global memory. This computation features $O(n)$ data reuse.

On a per-block basis, one thread block of size b handles one sub-matrix C_{sub} of C of size $b \times b$. A and B are only loaded N/b times from global memory wich represents the equivalent of saving as memory bandwith. The shared memory acts like a cache memory : smaller but faster than global memory, saving more bandwidth by reducing the global memory traffic. This is where the blocks are loaded to.

The kernel implemented into CUBLAS 2.1, delivered by [Volkov and Demmel, 2008] is a cache-aware strategy with explicit definition of the cache size optimal for a GTX280 GPU. The technical spe-

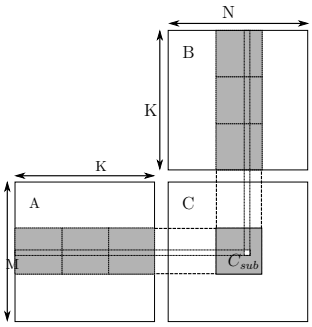

FIG. 1 – Per-block microparallel matrix product

cifications of the Tesla are kept closed by its manufacturer, but the informations given by the CUDA driver, indicate a number of core units and cache sizes equivalent on the Tesla T10.

The dimension of C_{sub} is bounded by the number of threads per block. Keeping tiles squares, and the maximum number of threads per block beeing 512, the size of the tiles is consequently 16^2. Each thread carrying a unique identifier under the combination of a unique block ID and a thread ID inside this block identifies the element to work on.

Let A, B and C be respectively of dimension (m, k), (k, n) and (m, n). C_{sub} is equal to the product of two rectangular matrices : the sub-matrix of A of dimension (b, m) that has the same line indices as C_{sub}, and the sub-matrix of B of dimension (n, b) that has the same column indices as C_{sub}. These two rectangular matrices are divided into as many square matrices of dimension b as necessary and C_{sub} is computed as the sum of the products of these square matrices. Each of these products is performed by first loading the two corresponding square matrices from global memory to shared memory with one thread loading one element of each matrix, and then by having each thread compute one element of the product. Each thread accumulates the result of each of these products into a register and once done the result is written back to global memory.

Computated this way, using the maximum of registers, and the fast level one shared memory to save bandwidth, can be seen as a more general strategy to divide the number of times the matrices are read. The idea goes beyond the scope of a single graphical unit : by spliting huge matrices among a several Tesla connected through PCI express 16, and beyond, to a hybrid cluster. All the units runs its own part of computation, and produce one $m \times n$ matrix. However this is memory and bandwidth consuming, because it leads to gather as many submatrices as there are load distributed, the computation may overlap with streaming. but doesn't scale.

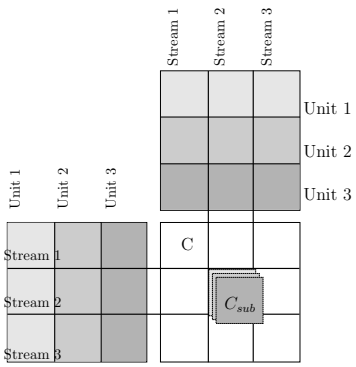

FIG. 2 – On a coarse level, load balance of the computation, featuring a $O(n \times u)$ memory overload of the matrix C, where n is the matrix size and u the number of unit.

3. Floating point specification

According to [Cud, 2009b], the binary floating point arithmetic available on the Tesla T10 series is *compatible* with the norm IEEE-754 [Goldberg, 1991]. with deviations :

- Rounding modes are not dynamically configurable and must be explicitly set.
- No mechanism for detecting a floating point exeception as occured.
- Operations behave as if the IEEE-754 exceptions are always masked, and deliver the masked reponse as defined by IEEE-754 if there is an exceptionnal event.
- For the same reason as above, while SNaN encodings are supported, they are not signaling.
- Absolute value and negation are not compliant with IEEE-754 with respect to NaNs.

4. CUBLAS Library

For now, the CUBLAS can stand for immediate substitution to an other BLAS implementation with very minor modifications. However, the default behavior of the library is to perform allocations and transferts of data on every time a host fonction call a CUBLAS routine. For an efficient use and in the general case, the memories zones allocations, desallocations and transferts between host (CPU) and device (GPU) should not occur as part of a call to the BLAS routines. Moreover, the memory allocated on the host must be aligned and non-pageable for maximum performances [Cud, 2009a]. This leads to extra code.

Where transferts operations – specially streamed transfert for stripes of matrix – occur most of the time a few instructions before the kernel call, the memory allocations – or in the same order of idea the device initialisation and warming[1] – should be tactically positioned to avoid repeating instructions, and the programmer should make a clever "use of reuse of data". This is reinforced if the computation shows any behavior of redundancy.

However memory allocation on the device is not a costly operations, and has very minor impact on the performances when compared to memory bandwidth issue.

Should the program be written in Fortran, the call requesting page-lock memory allocation requires C bindings for pointer manipulations. The performance comparison of a direct linking of CUBLAS and its optimal use is described on figure 3.

Version 2.1 of CUBLAS release is incomplete, without any BLAS 3 complex functions. which are to be partially found in the beta version 3.0.

[1] The effect is noticeable enough to worth to be mention.

Listing 1 – Extra Fortran code for allocation of page locked memory on the host

```
module cuda
use iso_c_binding
 interface
 ! cudaMallocHost
  integer(C_INT) function cudaMallocHost(buffer,
     size)  bind(C,name="cudaMallocHost")
   use iso_c_binding
   implicit none
   type (C_PTR)  :: buffer
   integer (C_SIZE_T), value :: size
  end function cudaMallocHost
end interface

! allocate m*m elements of p bytes into array A
! of page-locked memory on the host RAM,
! to ensure maximal host<->gpu bandwidth
subroutine allocateforgpu( A, m, p, cptr )
 ! page-lock constant, stack or heap data going
 ! to the gpu requires a pointer reference
 real, dimension(:,:), pointer :: A
 integer :: m
 integer :: r
 type(C_PTR) :: cptr
 integer(C_SIZE_T) :: p

 r = cudaMallocHost (cptr, m*m*p)
 ! bind the C pointer
 call c_f_pointer (cptr, A, (/ m, m /))
end subroutine allocateforgpu
```

5. Numerical accuracy

To have a guess of the numerical accuracy achieved with CUBLAS on a T10, we compute the GEMM function $C = \alpha AB + \beta C$, square matrices of size m, with B and C equals to an integer λ positive. Odd columns of A are equal to λ and even columns of A equals to a floating point value ε comprised between 0 and 1. The result is a matrix C constant.

$$A = \begin{pmatrix} \lambda & \varepsilon & \dots & \lambda & \varepsilon \\ | & | & & | & | \\ \lambda & \varepsilon & \dots & \lambda & \varepsilon \end{pmatrix} \quad B = \lambda\ , \quad C = \lambda$$

Resulting in

$$C = \alpha(\lambda^2 + \varepsilon\lambda)\frac{k}{2} + \beta\lambda$$

We compute from real to floating point space, with an approximation δ, giving the result $\bar{a}$.

$$\mathbb{R} \rightarrow \mathbb{F}$$
$$a \rightarrow \bar{a}$$
$$\bar{a} = a(1+\delta) \sim \Psi$$

where δ is the *epsilon machine*. On Intel Xeon Nehalem or Core 2 Duo CPU :
epsilon(double precision) $= 2.220446049250313E^{-16}$
epsilon(simple precision) $= 1.1920929E^{-07}$

Results

We take $\lambda = 2$ and $\varepsilon = 10^{-n}$ with $n = 1..N$. We have $B = C = 2$. As a result $C = 4m+2\varepsilon m+4$. The matrix dimension m varies from 256 to 12000. The accuracy is estimated from the relative difference between the exact result, computed on the CPU in double precision and the result given by the GPU.

The plots 6, 7, 8 are the results of the successive computation of the relative errors $\frac{||C-C'||}{||C||}$ as a a function of the matrix size, of the exact expected result C solely rounded to the epsilon machine, and the computations of the constant matrices through the BLAS implementation C', subject to cumulative rounding errors.

The variability and cancelations of precisions comes from the *denormalization* and *rounding modes* of numbers.

GEMM's MKL shows a great deal stability. ATLAS runs on one core only and gets even more accurate but the tradeoff of a computationnal power is high. Moreover, the fluctuations of accuracy is in itself a factor of inpredictabilty, hence inaccuracy.

6. Performances

In 32-bits floating point precision, the SGEMM function reaches 370 gflops of peak performances. This is acheieved on a matrix of 12K square size. 300 gflops is reached at 4K. A know fact of GPGPGU : full efficiency is achieved with matrices big enough. Below 2000^2, memory transferts latency has more impact on the overall computation and below, thread multiprocessors are left idled.

The increasing sizes of matrices has less impact on the kernel execution than on the data transfert. Using *streams*, that is, concurrent copy and execution, the input matrix A and C are split into row-major stripes and transfered asynchronously, while the kernel run on an other piece of data.

The performances achieved by MKL comes from the sustained cached access : the Pentium 4 can fetch a 128 bits SSE value (4 packed 32 bits floats) in one cycle and eight cores units are to be found on the Xeon Nehalem.

The speedup when comparing the two librairies begins in single precision for matrices bigger than 1000^2, reaching 2.6 at 7000^2. In double precision, MKL wins, but both solutions converge to a maximum computationnal power of 67 GFlops.

A division of the computationnal load using *streaming* was shown first in [Fatahalian et al., 2004] dispatching the load between the threads of multiprocessors of one GPU. As we mentionned, it is also technically possible to apply it on the macro-scale too : that is, among several Tesla cards. The blocking strategy performs sub-matrix blocks products and blocks additions inside the cache-memory before retrieval to global memory. The strategy dispatching the load would organize the two matrices in row-major and column-major stripes, with as many stripes as computation units. One submatrix is computed inside every units. In the end, on master unit has the task to perform a parallel one to one sum of every submatrix inside of its memory. A two by two fused access can hide some more of the latency, while the parallel sum of every subset is at peak speed.

One other way to do this is through a library layer such as the *CUDA Wrapper Library* [2]. It's implemented in a forced preload, such that the device allocation calls to CUDA are intercepted by it for a few different benefits. Users requesting multiple GPUs per node really need to be aware of it's transparent operation. The wrapper libary accomplishes two things :

1) Virtualizes the physical GPU devices to a dynamic mapping, that is always zero indexed. The virtual devices visible to the user map to a consistent set of physical devices, which accomplishes "user fencing" on shared systems and prevents users from accidentally trampling one another.

2) NUMA affinity, if relevant, can be mapped between CPU cores and GPU devices. This can save much memory bandwidth.

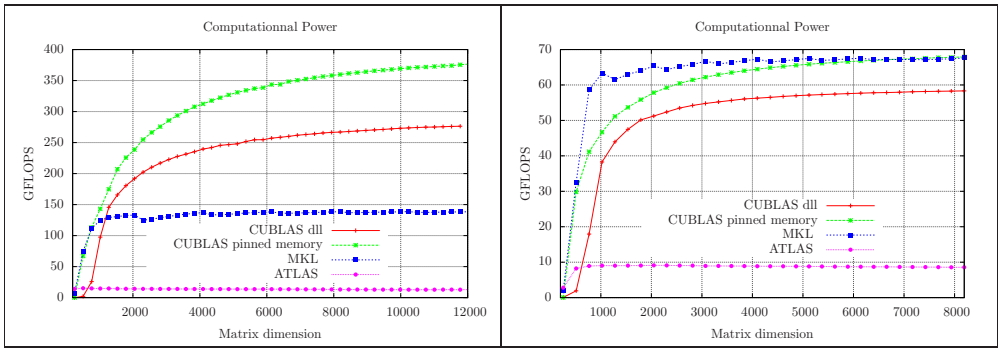

FIG. 3 – Computationnal power in single (left) and double (right) precisions.

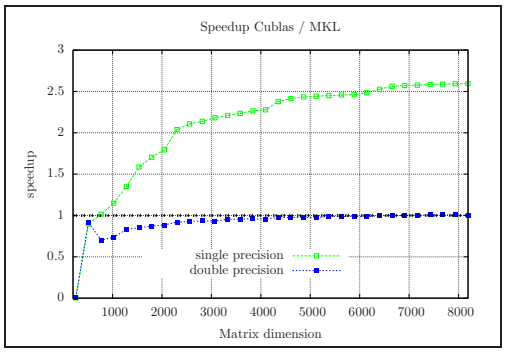

FIG. 4 – CUBLAS vs MKL : speedup as a function of matrix dimension.

7. Conclusion

We evaluated the performance and accuracy of the GEMM function, in single and double precision, showing that MKL is the stablest and fastest solution in double precision, while a maximum speedup of 2.6 can be reached in single precision for matrices bigger than 2000 square size elements. This benhmark shows that the computation on the T10 has higher relative errors when compared to one performed on a CPU. This is can get to an order as high as 10^{-4}. In the general case, on or two orders higher than MKL.

Computing many times big constant matrix on a hardware lacking of EEC, this benchmark can be extended fairly easily with a parallel prefix scan to detect a memory fault occurence and give an estimation of the mean time between faults (MTBF). For exemple, if the MTBF for one isolated SP core unit is 6 months, on a 240 core units, it would be only 16 hours and 48 minutes (by establishing an independant distribution of time).

These benchmarks will be run again on the T20 GPU, codenamed Fermi.

Regarding performances, a clearer picture of the synchronisation, and memory transferts is becoming increasingly necessary for critical kernels through examinations of the PTX assembly with dissassemblers, such as Decuda [1]. The latter can be a valulable source of information, to optimize cache movements, understand clock cycles and get more insight of the hardware.

On a final note, one should consider the very hybrid multicore architecture as a whole so to get the best of the two worlds, like what autotuning techniques [Li et al., 2009] does inspired by the ATLAS developpement. When dealing with real world problems, we can alays request the contribution of both librairies, working in concert.

References

[Cud, 2009a] (2009a). NVIDIA CUDA – Programming Best Practices Guide. Technical report, NVIDIA. Toolkit v2.3.

[Cud, 2009b] (2009b). NVIDIA CUDA – Programming Guide. Technical report, NVIDIA. Version 2.3.

[Fatahalian et al., 2004] Fatahalian, K., Sugerman, J., and Hanrahan, P. (2004). Understanding the Efficency of GPU Algorithms for Matrix-Matrix Multiplication. *Graphics Hardware*.

[Goldberg, 1991] Goldberg, D. (1991). *What every computer scientist should know about floating-point arithmetic*, volume 23. ACM, New York, NY, USA.

[Li et al., 2009] Li, Y., Dongarra, J., and Tomov, S. (2009). A Note on Auto-tuning GEMM for GPUs.

[Volkov and Demmel, 2008] Volkov, V. and Demmel, J. (2008). Benchmarking GPUs to tune dense linear algebra.

[1] Decuda, Wladimir J. van der Laan, http ://wiki.github.com/laanwj/decuda.

[2] Cuda Wrapper, University of Illinois, Innovative Systems Lab, National Center for Supercomputing Applications, http ://sourceforge.net/projects/cudawrapper/.

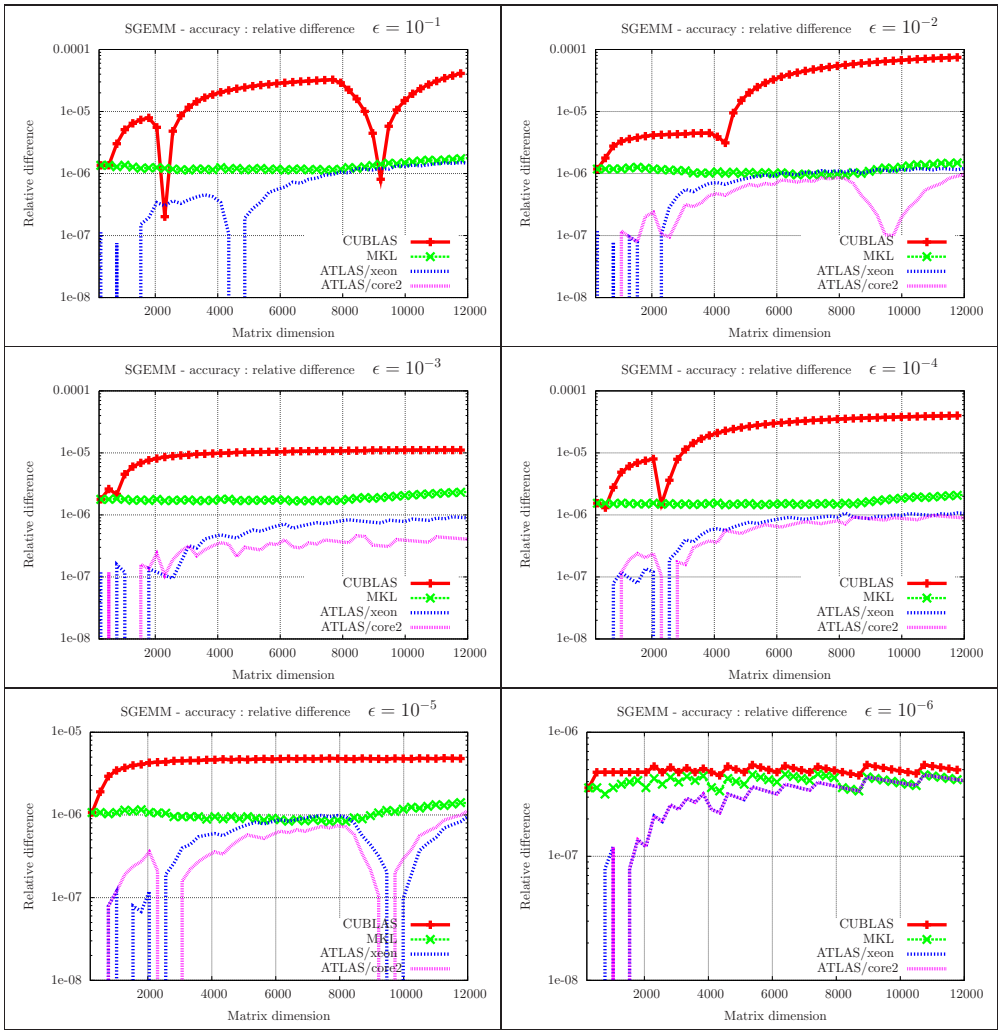

FIG. 6 – SGEMM accuracies, relative differences, epsilon ranging from 10^{-1} to 10^{-6} : CUBLAS 2.1 on Nvidia T10, ATLAS 3.9.15 and MKL 10.0 on CPU Xeon Nehalem 8 cores X5570 @ 2.93GHz

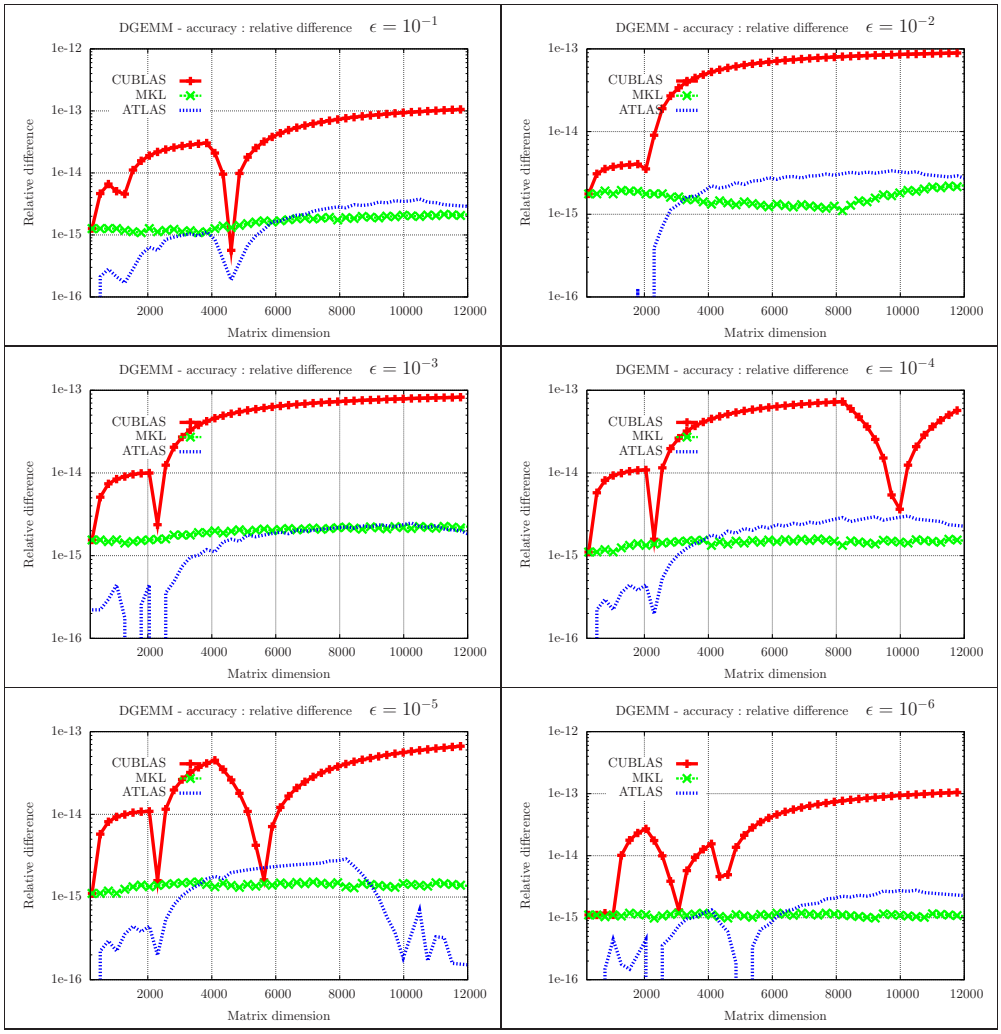

FIG. 7 – DGEMM accuracies, epsilon ranging from 10^{-1} to 10^{-6} : CUBLAS 2.1 on Nvidia T10, ATLAS 3.9.15 and MKL 10.0 on CPU Xeon Nehalem 8 cores X5570 @ 2.93GHz

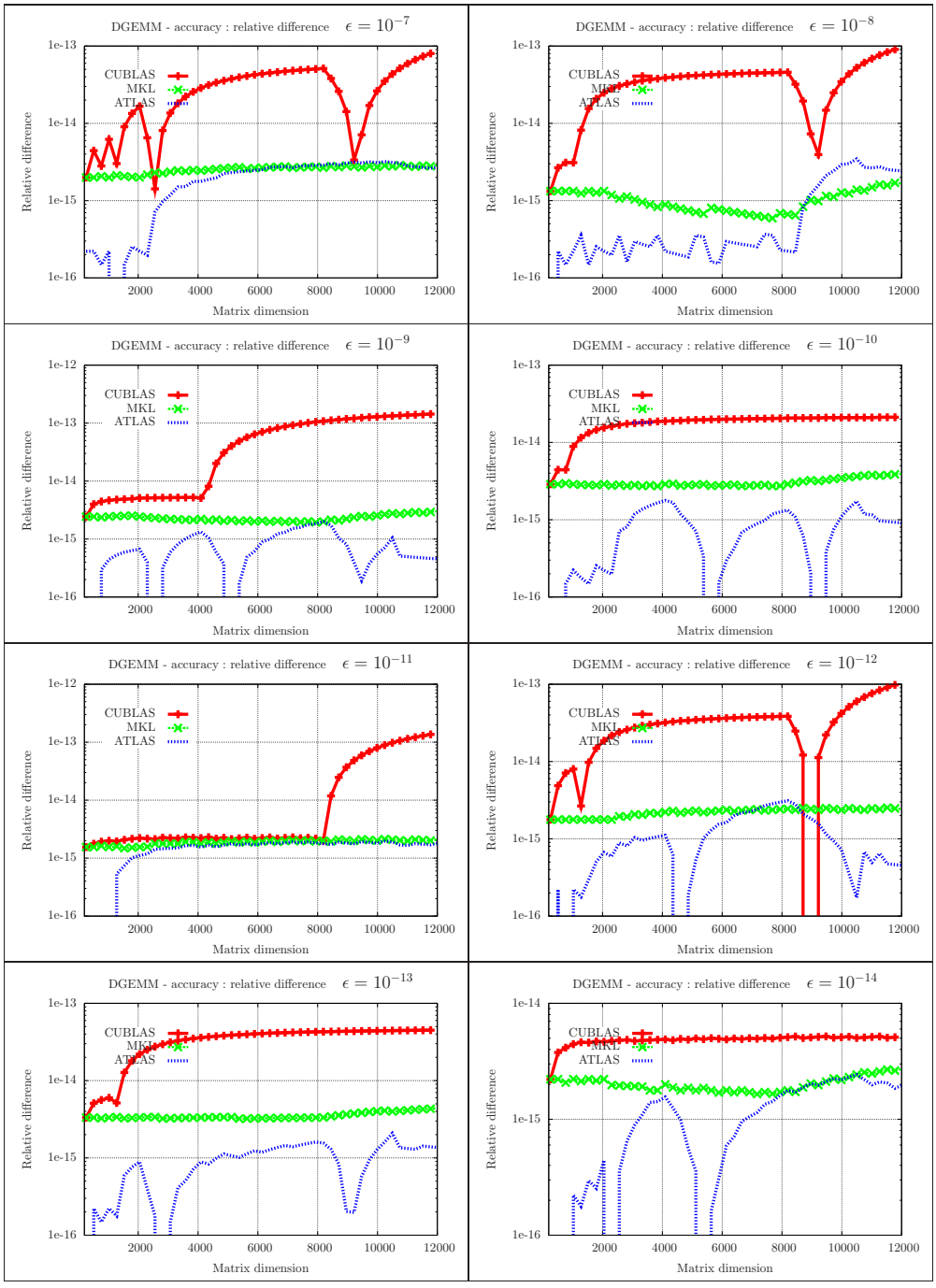

FIG. 8 – DGEMM accuracies, epsilon ranging from 10^{-7} to 10^{-14} : CUBLAS 2.1 on Nvidia T10, ATLAS 3.9.15 and MKL 10.0 on CPU Xeon Nehalem 8 cores X5570 @ 2.93GHz