# Massive Parallel Programming
# Low Rank Approximation

Eric Dagobert

January 17, 2017

**Abstract**

The aim of this study is to highlight the advantage of using GPUs to perform complex matrix calculations. Low Rank Approximation is an interesting topic to look at because it allows dimension reduction and also it involves complex matrix calculation such as QR decomposition. Last but not least the algorithm we present here involve randomness, which can be helpful to verify the numerical stability of our program.
The Low Rank approximation algorithm described here has been implemented in C++ within two different programs: one using CUDA parallel calculations, and the other is the baseline, using standard mathematics functions. We want to demonstrate the performances of massively parallel programs show a significant reduction of complexity costs.

## Low Rank Approximation

The idea of finding a Low Rank Approximation for a matrix is to compute an equivalent matrix with a much smaller number of *orthonormal* columns: this is literally a *compression*. The approximation means the compressed matrix is *almost* representative (to an $\epsilon$ in fact) but at a much lower computing cost.

### Rank decomposition

We know that every matrix has a rank decomposition, which means every matrix can be projected on a base of linearly independent vectors (property of Matrix Algebra). For any matrix $A$ of dimensions $m \times n$ there is an integer $r$ such that $A = V_r \times H_r^T$ with $V_r : m \times r$ and $H : n \times r$. The smallest rank decomposition is usually computed using orthonormal projections (another method is Singular Value Decomposition) and has a cost of $O(mn^2)$.

### Approximation

We also know that a Rank Decomposition is not unique: for any finite dimension matrix there is an inifinity of rank decomposition. The algorithm described below uses random multiplication to find a satisfying rank decomposition by mutliplying the target matrix with a random (Gaussian) matrix of lower dimensions. The Low Rank approximation algorithm used in this study has been described in [HMT].
Finding a low rank approximation consists in running Algorithm 1 different times by either fixing the target rank and trying different Gaussian products until $\epsilon$ is satisfying, or computing the smallest rank for which the approximation is close to a given $\epsilon$.
We can immediately see the complexity reduction offered by this algorithm : if $l$ is small enough, the complexity $O(ml^2)$ become significantly lower.
A good approximation is defined by an $\epsilon$ such that the condition $\|(I - QQ^T)A\| \leq \epsilon$ is reached, and the probability of failure is a negative power of $\epsilon$ ([HMT]).

**Algorithm 1** Randomized Range Finder

*Given a $m \times n$ matrix* $\mathbf{A}$ *and an integer $l$ compute an $m\times$ orthonormal matrix* $\mathbf{Q}$*:*

1. Draw an $n \times l$ Gaussian random matrix $\Omega$.
2. Form the $m \times l$ matrix $Y = A\Omega$.
3. Construct a $m \times l$ matrix $Q$ using the QR factorization $Y = QR$.

## CUDA toolkit

The algorithm implementation has been achieved on a Laptop with the following characteristics:

- Intel x64 i7 at 2.5 GHz
- NVIDIA GeForce 930M (Maxwell architecture) : 384 cores at 930 MHz
- Visual Studio 2013 compiler
- CUDA Toolkit 7.5

The CUDA toolkit is available at [CDT]. It provides a set of utilities to manage matrices and vectors, but above all a set of Algebra functions called cuBLAS based on LAPACK routines and especially optimized for the purpose of parallel computations. cuBLAS library is compatible with C++ compilers but it is not object oriented at all, we imagine due to performance reasons (they are to be compiled by the cu compiler). Therefore functions always carry an opaque pointer (a "context") and there is no type overloading, every data type is attached to a specific list of functions and linked to some naming conventions. Allowed data type are: float, double, complex and double complex. For instance the signature of a cuBLAS matrix multiplication function is:

```
cublasStatus_t cublas<t>gemm(cublasHandle_t handle,...)
```

Here, `<t>` is a letter representing the data type, for instance `S` for matrices of `float`.
cuBLAS also provide a list of helper functions used to manage data and memory transfer to and from the kernel.
QR decomposition is also available in CUDA Toolkit but under a different category of function called `cusolver`. This high-level package provides solutions for QR,LU and SVD decomposition, linear solver as well as some matrix and vector permutations. These functions require a specific opaque context of type `cusolver<T>_handle` and have specific optimization for Dense and Sparse matrices or Re factorization purposes, thus following the LAPACK classification. Finally, we also used cuBLAS to generate random Gaussian matrices thanks to the `curandGenerator` library, which offers random number generator based on various distributions.

### Structure of the CUDA program

The program has a kernel part (`kernel.cu`) made of extern C functions where all the CUDA computations are performed; the flow management part is made in a regular `cpp` file.
We have decided to include streams in the program for two reasons: first, as the algorithm is based on many iterations, it is interesting performance wise to run these iterations in parallel as it maximize the memory and GPU usage, particularly in the case of small matrices. The other reason is to see if mixing two levels of parallelism leads to a degradation of performance or not as it implies using two levels of barriers: `deviceSynchronize` and `streamSynchronize`.

- Main C kernel function implementing Algorithm 1:

```
extern "C"
void cudalowrank_app(
        cublasHandle_t handle,          //library opaque pointer
        cusolverDnHandle_t shandle,     //cusolver opaque pointer
        cudaStream_t streamid,          //stream identifier
        WorkMemory* wm,                 //memory chunk for the stream
        double* d_M,                    //input matrix data
        int m, int k,                   //input matrix dimensions
        int r,                          //target rank
        double* epsilon)                //result : epsilon
{
   /* ... */
}
```

- `WorkMemory` is an array of structures containing output and intermediate matrix calculations. It is allocated once and reused for different iterations for optimization purpose.

- `cudalowrank_app`'s body implementation of Algo.1 for one iteration and outputs corresponding $\epsilon$:

```
void cudalowrank_app(/*...*/)
{
        // constants for matrix multiplications
        double alpha = 1.;
        double beta = 0.;
        // variables for QR decomposition
        int bufferSize = 0;
        int *info = NULL;
        double *buffer = NULL;
        double *tau = NULL;

        // Compute Y = M x B (random gaussian)
        cublasDgemm(handle,CUBLAS_OP_N,CUBLAS_OP_N,m,r,k,&alpha,d_M,m,wm->d_B
            ,k,&beta, wm->d_Y,k);



        // Prepare QR factorization of Y
        //              buffers allocation and initialization

        cudaMalloc(&info, sizeof(int));
        cudaMalloc(&buffer, sizeof(double)*bufferSize);
        cudaMalloc((void**)&tau, sizeof(double)*min(m, r));
        cudaMemset(info, 0, sizeof(int));

        cusolverDnDgeqrf_bufferSize(shandle,m,r,wm->d_Y, m,&bufferSize);
        cusolverDnDgeqrf(shandle,m,r,wm->d_Y,m,tau,buffer,bufferSize,info);

        // Computation of QR: Q returned into wm->d_Q
        cusolverDnDormqr(shandle,CUBLAS_SIDE_LEFT,CUBLAS_OP_N,m,r,min(m, r),
            wm->d_Y,m, tau,wm->d_Q,m,buffer,bufferSize,info);

        // release temporary buffers
        cudaFree(info);
        cudaFree(buffer);
        cudaFree(tau);

        beta =-1.;

        //compute Q x QT : should be close to I
        //store the result in wm->d_WB1

        cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_T, m, k, r, &alpha, wm->
            d_Q, m, wm->d_Q, k, &beta, wm->d_WB1, m);
```

```
        beta = 1;

        //calculate (I-QQT)M
        cudaMemset(wm->d_M_, 0, m*m*sizeof(double));
        cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, m, m, &alpha, wm->
            d_WB1, m, d_M, k, &beta, wm->d_M_, m);

        // calculate epsilon = || (I-QQT)M||
        cublasDnrm2(handle, m*k, wm->d_M_, 1, epsilon);
}
```

- Note on QR decomposition: The method used for QR decomposition by cusolver and our baseline program is using **Householder matrix** as described in [NRC]. Intrinsically this method is hardly parallelisable, but cusolver computes vector dependencies as described in [PSC] during the call to helper functions `cusolverDnDgeqrf_bufferSize` and `cusolverDnDgeqrf`.
- Gaussian matrix: cuda Toolkit provides a random number generator than can fill in parallel a vector of a given size:

```
extern "C"
double*
genGauss(curandGenerator_t prngGPU,int n)
{

        double * v = 0;
        cudaMalloc((void**)&v, n*sizeof(double));
        curandGenerateNormalDouble(prngGPU, (double *)v, n, 0., 1.f);
        return v;

}
```

### Testings

To verify the program works correctly we generated test matrices that are *almost* low rank decomposable and submitted them as input. To create such matrices we used the SVD decomposition backwards. Let $M$ be a square matrix of size $(n, n)$ s.t. $M = \Omega_1 \times D \times \Omega 2$ where $\Omega 1$ and $\Omega 2$ are two Gaussian $(n, n)$ matrices, and $D$ is a diagonal matrix having the following shape:

$$D_{n,n} = \begin{pmatrix} \omega_1 & & & & & \\ & \ddots & & & & \\ & & \omega_r & & & \\ & & & \epsilon_1 & & \\ & & & & \ddots & \\ & & & & & \epsilon_{n-r} \end{pmatrix} \text{ with } \epsilon_i \ll \omega_j$$

Thus the matrix $M$ generated has $r$ significant linearly independent vectors.

## Performance results

The algorithm used here has a complexity of $O(nr^2)$ but we performed a series of tests using a worst case scenario of $r = n$, where the input matrix is extremely dense, which is also a worst case scenario for the QR decomposition ($O(n^3)$).

### No streams, variable matrix size

The first series of tests looks at the computing cost in respect of the size of the input matrix. We measured the GPU time taken to compute one iteration of algorithm 1 using function `sdkGetTimerValue`.

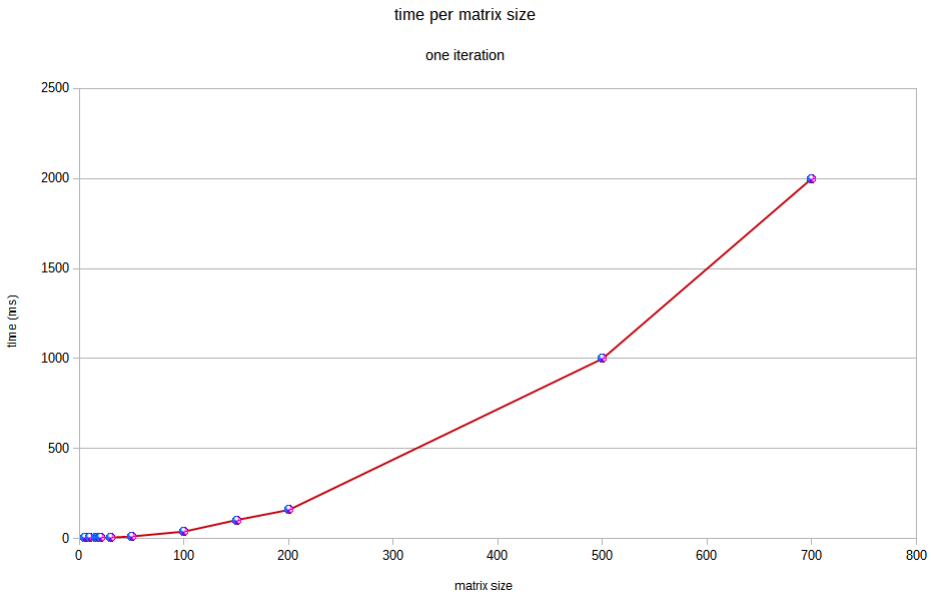



Table 1: Performance in function of matrix size

| size | 5 | 10 | 15 | 17 | 19 | 20 | 30 | 50 | 100 | 150 | 200 | 500 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ms | 5 | 5 | 5 | 5 | 5 | 6 | 9 | 14 | 40 | 104 | 160 | 1000 |

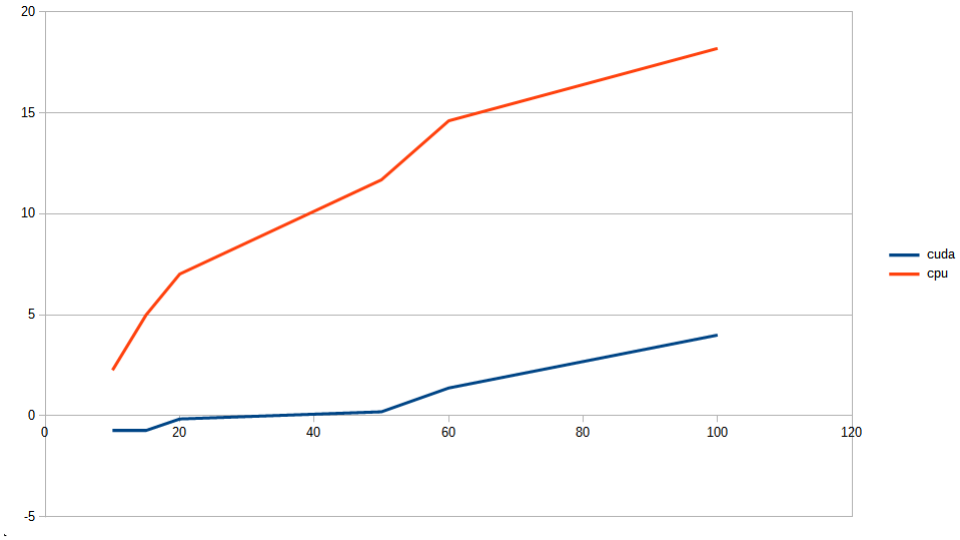
Results confirm what was expected. With 384 cores, the computation time is practically constant as long as the matrix size is small enough; we can see in Table 1 that the computing time is $O(1)$ for size $n$ such that $n \times n \leq 384$, somewhere between 19 and 20. For bigger sizes the complexity is still parabolic (i.e. pol but the performance comparison with the non parallel program will show the complexity is consistent with $O((\frac{n}{N})^3)$ with $N$ being the number of computing units.

### Performance comparison with regular C++ program.

We ony have compared the QR decomposition because it represents the heaviest part of the algorithm in term of computing cost. The baseline QR decomposition is a C++ program, using the same compiler, written from the method described in [NRC] which is equivalent to what `cusolver` implements except for the optimization part.

Table 2: Performance in function of matrix size

| size | 10 | 15 | 20 | 50 | 60 | 100 |
|---|---|---|---|---|---|---|
| cuda (ms) | 0.6 | 0.6 | 0.9 | 1.5 | 2.6 | 16 |
| cpu (ms) | 4.8 | 32 | 130 | 3300 | 25000 | 300000 |




The performance difference displayed above using logarithmic scale shows complexities for CPU and CUDA are both polynomial but in the case of CUDA, the slope is much lower, almost flat for matrix sizes up to 50, which is consistent with the expected complexity reduction of $O((\frac{n}{K})^p)$.

### Streams

Streaming is relatively easy to implement as `cusolver_handle` and other cuda opaque structures are also defined for stream synchronization purpose.

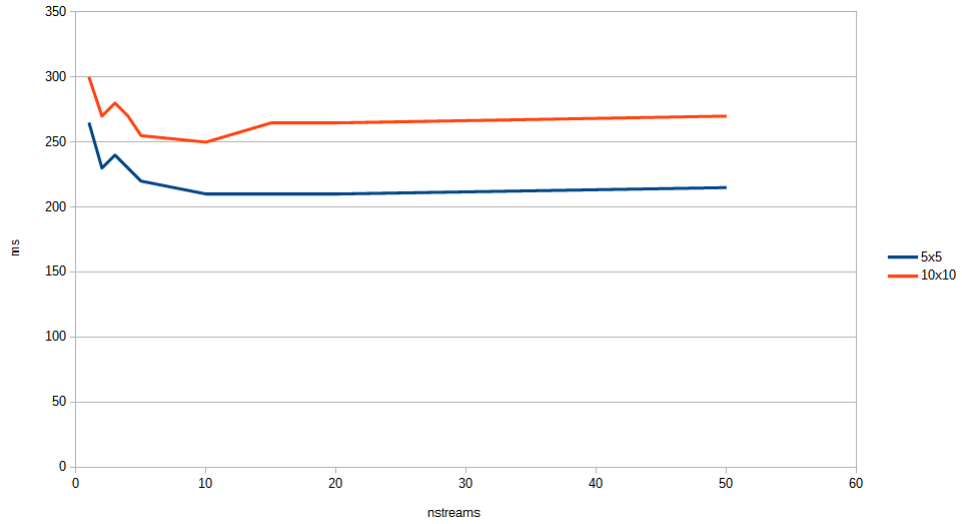
```
cudaStream_t *streams =(cudaStream_t *)malloc(nstreams*sizeof(cudaStream_t));

for (int i = 0; i < nstreams; i++)
{
        cudaStreamCreate(&streams[i]);
        cublasSetStream(handle, streams[i]);
        cusolverDnSetStream (shandle, streams[i]);
        /* ... */
```

Adding streams is effectively interesting in case of small size matrices, for which only a part of the available cores are working. Otherwise the supplementary barriers generate too much overhead to really bring some performance. We observed anyway a small gain probably due to the fact that memory copies and transfers are partially done in parallel.

Table 3: Effect of streaming/matrix size: 100 iterations

| nstreams | 1 | 2 | 3 | 4 | 5 | 10 | 15 | 20 | 50 |
|---|---|---|---|---|---|---|---|---|---|
| 5x5 | 265 | 230 | 240 | 230 | 220 | 210 | 210 | 210 | 215 |
| 10x10 | 300 | 270 | 280 | 270 | 255 | 250 | 265 | 265 | 270 |
| 100x100 | 3850 | 3850 | 3770 | 3775 | 3800 | 3780 | 3400 | 3700 | 3780 |




Only small size matrices would benefit from streaming, as seen in the table above, when the product $n \times n \times$ Nstreams is around the number of available cores. Otherwise, we see that the gain is somewhat irregular as parallel memory operation improvement compete with overhead and synchronization due to streams.Thus, instead of streaming, it is rather preferable to implement a real memory management policy, based on pre-allocation and RAM caching.

## Conclusion

We have seen using massive parallelism dramatically improve performances of complex matrix calculations, even if said calculation are not obviously parallelizable. Our approach were not optimal though because first it used too much memory for intermediate calculation, and second because the algorithm itself could have been parallelized at a finer level,i.e by having parallelism implemented at the iteration level instead of a succession of parallel operations. But both points can be worked on : the memory usage is easy to optimize by caching temporary results into RAM - this will generate an overhead but in linear time so it's acceptable in regard of the complexity of the algorithm. The other point is already addressed as [HMT] suggest a parallelization method of the algorithm. Nevertheless the method described in our study is relatively simple and cheap to implement and gives very acceptable performance improvement.

# References

[HMT] Halko, Martinsson, Tropp, *Finding Structure With Randomness: Probabilistic Algorithms For Constructing Approximate Matrix Decompositions*. SIAM review, Vol.53, No. 2, pp 217-288

[CDT] `https:://developer.nvidia.com/cublas`

[NRC] Press,Teukolsky,Vetterling, Flannery *Numerical Recipes, third edition* Cambridge University Press, pp 102-106

[PSC] *Parallel Solver with cuSOLVER* `https://devblogs.nvidia.com/parallelforall/parallel-direct-solvers-with-cusolver-batched-qr/`