Batched Least Squares of Tall Skinny Matrices on GPU

Tim Schmielau* and Jacques du Toit†

1 Summary

NAG has produced a highly efficient batched least squares solver for NVIDIA GPUs. The solver allows matrices in a batch to have different sizes and content. The code is optimized for tall skinny matrices which frequently arise in data fitting problems (e.g. XVA in finance) and are typically not that easy to parallelize. The code is 20x to 40x faster than building a batched GPU least squares solver using the NVIDIA libraries (`cuBLAS`, `cuSolver`). This gives a pronounced speedup for applications where the matrices are already in GPU memory.

For CPU-only applications, the cost of transferring the matrices from CPU memory to the GPU can dominate. We observed speedups including all transfer costs of between 1.5x and 12x for large enough problem sizes. Hence CPU-only applications can see a healthy speedup by adding a GPU and using NAG's software. NAG can provide code to benchmark users' systems to determine likely benefits and minimum batch sizes.

If the matrices have structure (e.g. polynomial basis functions), then much less data needs to be transferred. Evaluating the basis functions on the GPU means current CPU-only applications can see a 10x to 20x speedup for large enough problems. NAG can help users write the small amount of additional GPU code needed to do this.

2 Batched Linear Least Squares

The linear least squares problem

$$\min_{\beta \in \mathbb{R}^n} \left\| y - X\beta \right\|_2$$

for $y \in \mathbb{R}^m$ and $X \in \mathbb{R}^{m \times n}$ arises in many applications where a model is fitted to data. Often the design matrix X is "tall and skinny" ($m \gg n$), meaning there are many more observations than variables β. Usually this is handled via a QR factorization of X followed by an SVD factorization of the triangular matrix R. Since R is $n \times n$ and hence very small, the SVD has negligible impact on runtime. All the work is in the QR factorization. Unfortunately traditional QR algorithms for tall skinny matrices don't parallelize well, so better performance can be expected if independent least squares problems are batched together into a larger problem.

In finance, tall skinny least squares problems arise in American Monte Carlo (or whenever conditional expectations are approximated). In applications such as XVA with independent assets in a netting set, there is good scope for batching least squares calculations together.

3 Options for batching on CPU and NVIDIA GPU

To do batched least squares on a CPU one parallelizes over the least squares problems and calls the LAPACK routines `dgelss` or `sgelss` to solve each problem. If the LAPACK routines are efficient, e.g. from MKL, then this gives good performance.

On NVIDIA GPU there is batched QR decomposition in `cuBLAS`, but no routines for SVD and no batched `dormqr` or `sormqr` needed to apply Q to y. The SVD is done on the host (e.g. by calling MKL) and Q applied via streaming calls to single-matrix `dormqr` or `sormqr` in `cuSolver`.

NAG developed a custom batched least squares solver optimized for $500 \leq m \leq 100,000$ and $3 \leq n \leq 40$. Performance will gradually decrease as the number of columns becomes large. The SVD is done by LAPACK (MKL) on the host. The solver allows matrices in a batch to have different sizes.

*email: tim.schmielau@nag.co.uk
†email: jacques@nag.co.uk

4 The PCI Express 3.0 Bus

A motherboard with good PCIe 3 connections will typically be able to copy data from host to GPU at a sustained rate of around 12GB/s. This translates to 1.5×10^9 matrix elements per second for double precision data, and 3×10^9 matrix elements per second for single precision data. As we'll shortly see, on our test system with 8 threads on a Intel Core i7-7700K running at 4.2GHz, the CPU approach outlined above has performance comparable to the PCIe3 transfer speed, at least for small matrix and batch sizes. This means for small problems, the CPU runtime is comparable to the time it would take to move all the matrices over the PCI Express bus.

We therefore split our discussion into two. In the next section we consider applications where the matrices are already resident on the GPU. This will be applications that already run partly or entirely on the GPU.

We then consider applications where the matrices are resident on the CPU. Typically this will be applications which do not have any GPU component at all. It turns out that even here our batched least squares code can give benefits.

5 Applications with Matrices Resident on GPU

We compare our GPU implementation against the approach using the NVIDIA libraries outlined in Secion 3. For small batch sizes (less than 50) our code has *significant room for further improvement*. We display performance in terms of *throughput*, which is the total number of matrix elements in the batch divided by the runtime.

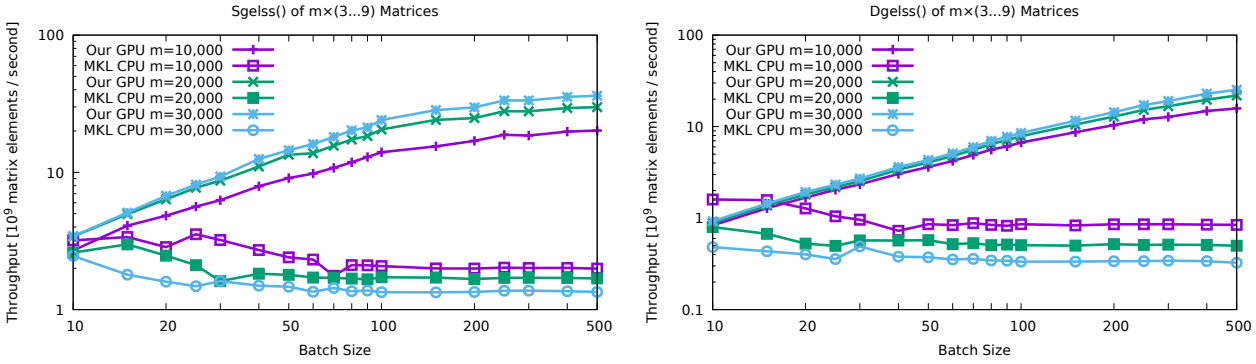

Figure 1: Single and double precision performance vs. batch size for $m \times n$ matrices with $n \sim \text{Uniform}(3, 9)$. Our code on P100 GPU vs. CPU code from Section 3 with 8 threads on Intel Core i7-7700K.

Figure 1 shows performance of our code with batches of differently sized matrices (same number of rows but columns distributed uniformly in the range $[3, 9]$). Note the NVIDIA libraries cannot handle batches with different size matrices. CPU runtimes are included to show the difference in behaviour.

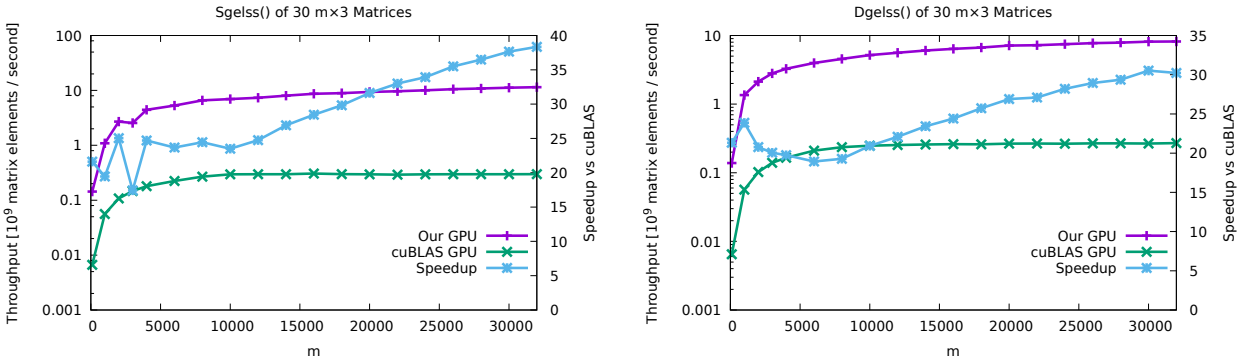

Figure 2: Single and double precision performance vs. number of rows for batches of 30 $m \times 3$ matrices. Our code is compared to NVIDIA libraries, both on P100. Performance for other problem sizes is similar.

Figure 2 shows performance of our code compared to batched least squares constructed using the NVIDIA libraries on batches of uniformly sized matrices. The speedup seen here is quite typical. Similar results are seen with larger batch sizes and with larger matrices.

6 Applications with Matrices Resident on CPU

As mentioned in Section 4 the PCIe3 bus can transfer double precision data at 1.5×10^9 matrix elements per second and single precision data at 3×10^9 matrix elements per second. Comparing this with Figure 1 we see that for small batch sizes the parallel CPU code can solve the problem faster than it would take to transfer to the GPU. However as the batch and matrix sizes increase, CPU performance decreases. Consequently, on our test system, there is a point where it becomes beneficial to copy the data to the GPU and solve the problem there[a]. The figures below measure performance in terms of *throughput*, which is the total number of matrix elements in the batch divided by the runtime.

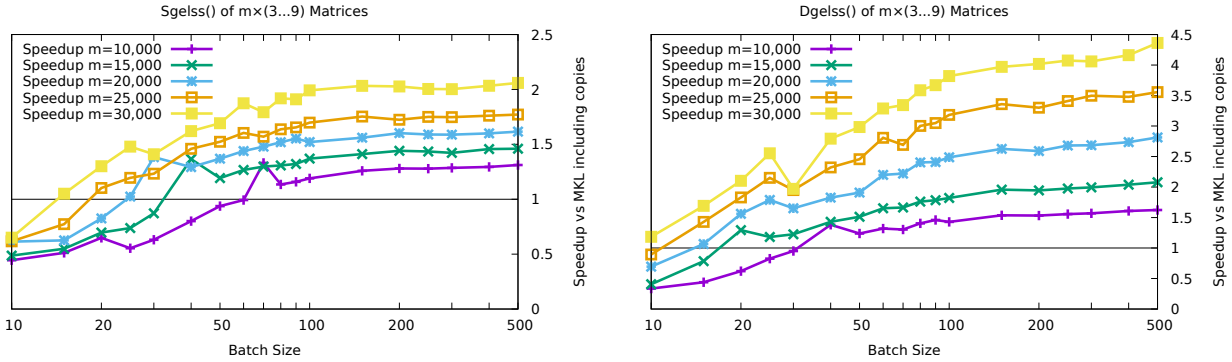

Figure 3: Speedup of P100 over CPU (8 threads + MKL, Intel Core i7-7700K) including PCIe transfer times (12GB/s). For large enough batch or matrix size the GPU is faster.

Figure 3 shows speedup including PCIe transfer times for batches of differently sized matrices. For large enough batch or matrix sizes it is faster to copy the problem to the GPU and solve it there. In double precision the crossover points are quite low. Figures 4 and 5 show the situation for uniformly sized batches: the result is the same.

6.1 Key Performance Factors for Matrices Resident on CPU

The PCIe transfer speed, the CPU processing speed, and amount of data transferred are the key factors determining if a speedup can be had by moving the problem to the GPU[b]. Our test system has quite a modern CPU (released Q1 2017), placing the GPU at a relative disadvantage since the CPU can solve many problems faster than it takes to move the data through the PCIe bus. Systems with older CPUs may see greater benefits from solving the problem on the GPU. NAG can provide software to benchmark a user's system and determine likely speedups. Achieving the 12GB/s rates quoted here will require page locked host memory. This is straightforward to incorporate into host applications that don't use GPUs.

If the input matrices have structure (e.g. polynomial basis functions) substantial improvements are possible. Transferring only the underlying factors and evaluating the basis functions on the GPU should result in overall speedups of more than 10x for big enough problem sizes. NAG can assist customers to implement this, it is a fairly small amount of additional GPU code.

7 Further Optimization for Small Batches

Our GPU implementation has significant room for further improvement at small batch sizes (below 50 matrices). Depending on whether the matrices are resident on the GPU or CPU, and on the problem sizes, these additional improvements may be beneficial.

[a]The SVD is done on the host, hence results are produced on the host

[b]The actual GPU used is not that crucial since most GPUs will solve the problem much faster than the data can be moved through the PCIe bus. Double precision data requires a GPU with sufficient double precision performance.

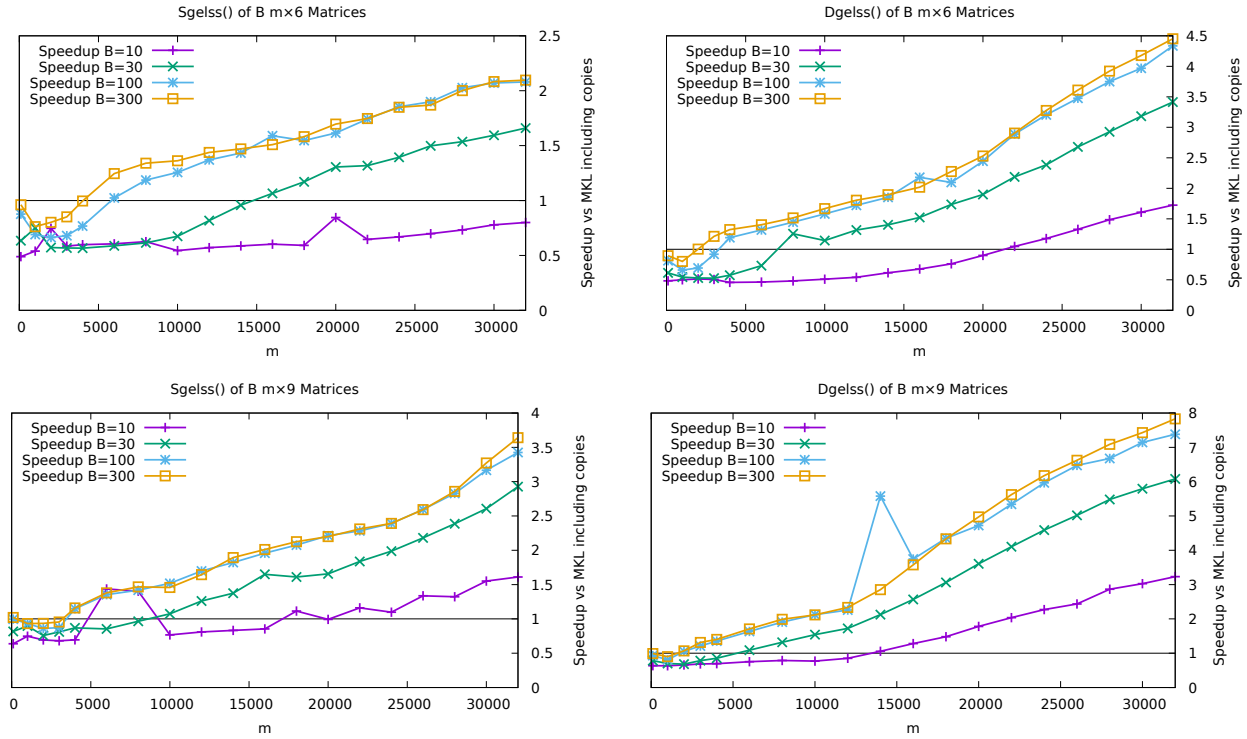

Figure 4: Speedup of P100 over CPU (8 threads + MKL, Intel Core i7-7700K) including PCIe transfer times (12GB/s) for batches of matrices with 6 and 9 columns. For large enough problems the GPU is faster.

8 Getting Access to the Code

The code is available to trial. To arrange access please contact `support@nag.co.uk`. The code can accept input data residing on the GPU or the CPU. In the latter case it handles all data transfers transparently so that from a user's perspective, it behaves like a regular CPU library routine. In particular, users need no knowledge of GPU programming.

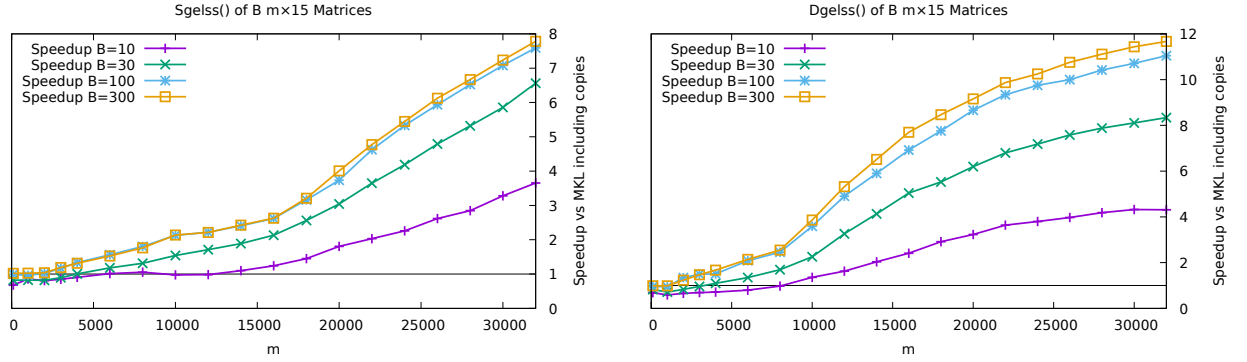

Figure 5: Speedup of P100 over CPU (8 threads + MKL, Intel Core i7-7700K) including PCIe transfer times (12GB/s) for batches of matrices with 15 columns. For large enough problems the GPU is faster.