Optimizing cluster-robust variance estimates for GPUs

Scott McNeil
Department of Economics
Carleton University
Ottawa, Canada K1S 5B6
scott.mcneil@carleton.ca

December 16, 2016

Abstract

In this paper, I show that the cluster-robust variance estimator can be optimized for use on a Graphics Processing Unit. The key to achieving this is maximizing the use of each thread's register memory while also taking advantage of the warp-shuffle instruction on newer NVIDIA architectures. This paper presents different implementations for two situations where the CRVE is used in high volume: Monte Carlo simulation and bootstrap resampling. In the case of Monte Carlo, speedups from a linear C++ process are significant. In the boostrap situation, speed-up is still reasonable but less due to the latency of DRAM memory access.

1 Introduction

One of the challenges of General Programming with Graphical Processing Units (GPGPU) is overcoming the high latency of memory access between the off-chip DRAM and the processing units. Linear algebra problems typically provide a sufficient work-to-memory access ratio but, when operating on many small matrices or vectors, this may not be the case. Recent research and architecture developments present opporunities for overcoming these hurdles.

Cluster robust variance estimation (CRVE) is a statistical technique for correcting inference when variance is correlated within groups. It is an important and often underused technique for econometric analysis [25]. The calculation of the CRVE itself is not particularly expensive but there are two scenarios when the CRVE is re-calculated many times. First, the CRVE has shown to be best when combined with bootstrap resampling, specifically the wild cluster bootstrap (WCB), which involves recalculating the CRVE hundreds if not thousands of times [6]. Secondly, reasoning about the CRVE often involves Monte Carlo simulations with tens of thousands of replications.

In both of the above cases, the relevant problem is of the "embarassingly parallel" variety—meaning a GPU implementation could provide considerable benefits. There are challenges, however, to optimizing the CRVE on a GPU. First, the CRVE requires calculating many independent matrix-vector and vector-vector products. Further, data from the first set of independent problems (matrix-vector product) must be re-used in a second and third (vector-vector multiplication and summation).

Recent research has shown that even small, independent linear algebra problems can be optimized on the GPU, especially via batched processing [24]. However, using current

batched implementations for the CRVE would mean needlessly moving data in and out of global memory on the GPU. Another recent development is the addition of the warp-shuffle instruction to NVIDIA GPUs via Compute Unified Device Architecture (CUDA). This has been shown to considerably reducde latency when data needs to be shared among threads. This paper will present an implementation that combines the approaches draws on existing batched implementations while making use of the warp-shuffle. All of the GPU results presented here were computed using an NVIDIA K80 GPU from Amazon Web Services. All CPU results were computed with a Intel Xeon E5-2686v4 (Broadwell) Processors, also from Amazon Web Services.

2 Literature Review

The CRVE is a generalization of the heteroskedasticity-robust estimator proposed by White [32], with the original CRVE estimator proposed by Froot [13]. The use of the CRVE with bootstrapping was proposed by Cameron, Gelbach and Miller, often referred to as the Wild Cluster Bootstrap (WCB) [6]. A good review of the CRVE is also provided by Cameron, Gelbach and Miller [7].

There are a number of software implementations of CRVE. This includes an implementation in the standard distribution of the Stata statistical language [29] and the Sandwich package for the R programming language [33]. Finally, the WCB is available through boottest package for Stata [30].

There has been considerable work done to optimize dense linear algebra (DLA) algorithms for GPUs. For a review of this, including batched processing, see [12]. Implementations of batched DLA algorithms are currently available both in NVIDIA's cuBLAS library [27] and the MAGMA library from University of Tennessee's Innovative Computing Laboratory (ICL) [5].

A number of applications have been presented for Generalized Matrix-Matrix Multiplication (GEMM) for many small matrices. For example, matrix exponentiation and high-order finite element methods can be optimized as batched GEMMs [20, 15]. The performance of tensor contractions has been shown to be improved through batched GEMMs and implemented in the MAGMA library [1].

Batched GEMMs are incorporated in a number of other batched DLA algorithms, including QR decomposition, Cholsky decomposition and LU factorizations [14, 10, 2, 9, 11]. Batched processing has also been used for vector reduction and vector scaling [28].

This paper will especially draw on work done by the ICL group for the MAGMA library. The group has optimized multiplication of many small matrices for GPUs [4, 24] and showed that optimizing register usage is important for achieving peak performance. Further work from the ICL group looks at batches of variable sized matrices [3], however this paper will focus solely situations where problems are of identical sizes.

The warp shuffle is an additional instructiona added in the Kepler and later architectures of NVIDA GPUs. This paper will particularly draw on examples for computing parallel reductions with a warp shuffle provided by NVIDIA [21]. The instruction has been shown to work well when applied to techniques comparable to the CRVE. This includes sparse Matrix-Vector multiplication [19, 18, 23], option pricing Monte Carlo [8], dynamic programming [16, 22] and linear algebra [17].

Finally, this paper will make the use of the CURAND library from NVIDIA for generating random numbers [26] and the Armadillo C++ linear algebra library [31] for sequential

comparisons.

3 Algorithm

3.1 Sandwich Matrix

The CRVE is typially used in the standard ordinary least squares model:

$$Y = X\beta + u$$

Where Y is a vector of dependent variables, X is a matrix of regressors with first column being a vector of ones, β is a vector of regression coefficient and u is a vector of errors. Using the oridnary least squares estimator:

$$\begin{aligned} \hat{\beta} &= (X'X)^{-1}X'Y \\ \hat{u} &= Y - X\hat{\beta} \end{aligned} \tag{1}$$

For cluster robust variance estimation, each observation (row of X) is assumed to have a group indice g. Then, the CRVE estimate of the variance of entry k in β is the (k, k) element of the following, often referred to as a sandwich matrix:

$$(X'X)^{-1} \sum_{g=1}^{G} x_g' \hat{u}_g \hat{u}_g' x_g (X'X)^{-1}$$

Where G is the total number of groups and x_g and $\hat{u}_g$ are the elements of X and $\hat{u}$, respectively, that correspond to group g. In terms of notation, this paper will refer to the number of rows of X and $\hat{u}$ as n and the number of rows of x_g and $\hat{u}_g$ as m. The number of columns of both X and x_g will be k.

This paper is particularly interested in the summation term of the sandwich matrix. The calculation of this term can be thought of in three steps:

1. Calculate the inner product of x_g and u_g for each group, the result of which will be a k length vector
2. Calculate the outer product of the above vector and itself for each group, the result of which will be a $k \times k$ symmetric matrix
3. Sum all the above products for all groups, the result of which will also be a $k \times k$ matrix

Scope-wise, this paper will focus on situations where k is relatively small (often 2) and m is constant. Under these situation, the inner summation term is particularly important for working with a GPU since it involves many small, independent calculations. Further, the CRVE summation term differs in two important ways from the matrix-multiplication problems targeted by existing batched implementations. First, since the final result is symmetrical, only the upper triangle needs to be calculated. Second, as noted above, the results of each set of calculations will be re-used in the next, giving an opportunity for within-block optimization.

3.2 Bootstrap Resampling

This paper will focus on two situations where the CRVE will be calculated many times: Monte Carlo simulation and boostrap resampling. For the purposes of this paper, the key difference between these two is in the source of the data. For the boostrap, the X matrix and Y will be pre-determined data while for the Monte Carlo new data will be generated at each stage.

For the boostrap, $\hat{u} = Y - X\hat{\beta}$ must be calculated before computing the sandwich matrix. This paper will assume $\hat{\beta}$ has already been calculated, since this involves a fairly trivial matrix-matrix multiplication that is handled well by existing implementations. Typically the number of bootstrap replications is fixed at 999. Therefore, this paper will focus on situations where n is relatively large. Specifically, it will look at situations where G is large and m is fixed at 32 as well as situations where G is relatively small and m is quite large. Both these situations are comparable to real data one could expect in econometric analaysis. For example, the first situation would correspond to a large panel data set tracking many individuals over several years. The second situation would correspond to state-level data with many thousand observations per state.

In both situations, this paper will assume that the total data set is too large to be contained in the shared memory of a thread block, which is typically only 48kb. Therefore, the calculation of $\hat{u}$ and the sandwich matrix must done in a separate kernel than the calculation of $\hat{\beta}$. Therefore, the implementation presented below takes $\hat{\beta}$ as already calculated. To calculate the sandwich matrix, first elements of Y, X and $\hat{\beta}$ must be fetched from DRAM, and the elements of $\hat{u}$ must be calculated. The algorithm will then be:

Algorithm 1 CRVE Summation Term

1: **procedure** CRVE(G, m, k)
2: **for** $g = 1$ `through` G **do**
3: **for** $i = 1$ `through` m **do**
4: $\hat{u}_{ig} = y_{ig} - \sum_{j=1}^{k} \beta_j x_{ij}$
5: **for** $j = 1$ `through` k **do**
6: $(x_g' u_g)_j = \sum_{i=1}^{m} u_{ig} x_{igj}$ ▷ Element j of inner product
7: **for** $j = 1$ `through` k **do**
8: **for** $h = j$ `through` k **do**
9: $(x_g' u_g u_g' x_g)_{jk} = (x_g' u_g)_j (x_g' u_g)_k$ ▷ Element j,k of outer product
10: result $= \sum_{g=1}^{G} x_g' u_g u_g' x_g$ ▷ Sum outer product for all G

The implementation presented here calculates each bootrap replication within a single thread block. This allows the entire boostrap replication to be computed within a single kernel and avoid reading or writing from DRAM more than once. Since the total replications are set at 999, there will be 999 total thread blocks, which should be sufficient work for the scheduler to distribute among the multiprocessors in the GPU.

Further, each k-length inner product vector will be computed within a single thread. This means that each thread-block will have G threads. The benefit to computing the entire inner product within a single thread is that all of x_{ig1} through x_{ig1} will be neccesary to compute $u\hat{i}g$ and then each element of the inner product will be combined with each other element to calculate the upper triangle of the outer product. Therefore, thread g will recursively access each row of x_g and element of y_g from DRAM. Then it will use these

values to compute $\hat{u_{ig}}$ and finally add $u_{ig}x_{igj}$ to each of the j elements of the inner product. Ultimately, thread g will calculate the vector below:

$$\begin{bmatrix} \sum_{i=1}^{m} u_{ig}x_{ig1} \\ \sum_{i=1}^{m} u_{ig}x_{ig1} \\ \vdots \\ \sum_{i=1}^{m} u_{ig}x_{igk} \end{bmatrix}$$

An alternative approach would be to compute partial summations of each inner product on each thread and then complete the summations via warp-shuffle. Then, the final thread would could compute the outer product. This would increase the number of threads per block and allow the parallelism of the algorithm to be increased considerably. In a situation where many GPUs were being used, this would allow the algorithm to be completed in fewer steps. However, the implementations in this paper were only tested on a single GPU and so this extra approach was not implemented.

Once the inner prouct has been calculated, thread g will calculate the upper triangle of the outer product as:

$$\begin{bmatrix} (\sum_{i=1}^{m} u_{ig}x_{ig1})^2 & (\sum_{i=1}^{m} u_{ig}x_{ig1})(\sum_{i=1}^{m} u_{ig}x_{ig2}) & \dots & (\sum_{i=1}^{m} u_{ig}x_{ig1})(\sum_{i=1}^{m} u_{ig}x_{igk}) \\ & \sum_{i=1}^{m} u_{ig}x_{ig2})^2 & \dots & (\sum_{i=1}^{m} u_{ig}x_{ig2})(\sum_{i=1}^{m} u_{ig}x_{igk}) \\ & & \ddots & \vdots \\ & & & (\sum_{i=1}^{m} u_{ig}x_{igk})^2 \end{bmatrix}$$

And then finally, all the outer products will be summed by the warp shuffle instruction. This allows each thread to pass data in its registers to other threads in a single warp with effectively no latency. This can be done recusrively to compute a summation in $\log(p)$ time. In this case, each element of the outer product will be passed to succesive elements of the same warp. Then, once the last thread in each warp has the total value, it will put the values in the thread block shared memory. One final thread warp will then sum the remaining results to complete the summation term. A reduced version of eight warps summing their respective outer products in presented in Figure 1, where $\sum_i$ is the respective outer product.

Finally, each element of the outer product from each of the 999 replicaitions will be stored in DRAM for further calcualtions.

3.3 Monte Carlo Simulation

In papers related to the CRVE [6], Monte Carlo simulations are used to test the effectiveness of the technique in different situation. This often takes the form of varying the data-generation process (DGP) and its resepective parameters. For each parameter set, the CRVE is then calculated several thousand times. For this paper, the repetition count R will be fixed at 50,000. In terms of DGP, this paper will use a standard cluster-style DGP presented by Cameron, Gelbach and Miller and will primarily look at varying G [6]. As per this set-up, k will be fixed at 2, meaning that there will be one column of random variables and one column of ones in X. Also, m will be fixed at 32. Y and X will be generated as:

$$x_{ig} = e_i + e_g$$
$$y_{ig} = x_{ig} + u_i + u_g$$

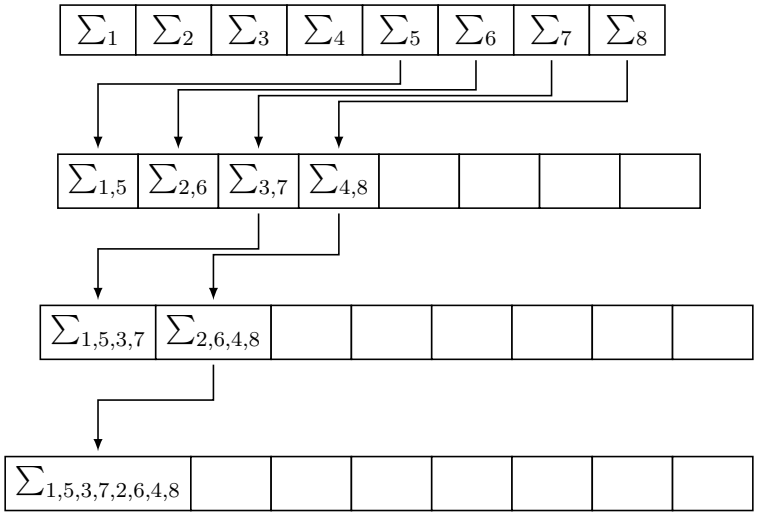

Figure 1: Demonstration of warp shuffle reduction with 8 threads

Where e_i and u_i are random numbers specific to obervations and e_g and u_g are group-specific random numbers. All four are drawn from $\sim N(0,1)$. In this case, the implementation presented in this paper will not assume $\hat{\beta}$ has been calculated since this is not a trivial. It will, however, take advantage of the collapsed formula for ordinary least squares of one variable:

$$\beta_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

One key element of this implementation is to make use of cuRAND, a library build by NVIDIA for generating random numbers in CUDA. This library makes it possible to generate properly independent pseudo-random numbers at the thread level. This means that, as long as each respective data sets fits within the registers of a thread block, the Monte Carlo process never needs to access the off-chip DRAM. As will be seen in the results, this will provide a much greater speed-up than in the bootstrap situation.

Beyond each data set being calculated at the thread block level by CURAND and having to calculate $\hat{\beta}$, this implementation for the Monte Carlo will be itentical to that for the bootstrap. In particular, both the terms for calculating $\hat{\beta}$ and the CRVE summation term are computed using a warp-shuffle reduction, as demonstrated before. As will be noted in the results section, the fact that this algorithm never needs to access DRAM memory provides a considerable speed-up compared with a linear C++ implementation.

The algorithm for calculating the CRVE for Monte Carlo is presented in Algorithm 2.

4 Results

The results from first set of tests of this algorithm are presented in Figure 2. In this case, m is fixed at 32 and then k and G are varied. As expected the time to calculate the full 999 summation terms increases mostly linearly as k and G increase. The time to calculate does see a small uptick when the number of groups is slightly higher than a multiple of 64, which

Algorithm 2 CRVE Monte Carlo

procedure OLS(m, k, n)
  $\bar{y} = \frac{\sum_{i=1}^{2} y_i}{n}$
  $\bar{x} = \frac{\sum_{i=1}^{2} x_i}{n}$
  $\hat{\beta}_1 = y - \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$
  $\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$
  for $g = 1$ `through` G **do**
    for $i = 1$ `through` m **do**
      $\hat{u}_{ig} = y_{ig} - \hat{\beta}_0 - \hat{\beta}_1 x_{ig}$
    for $j = 1$ `through` k **do**
      $(x_g' u_g)_0 = \sum_{i=1}^{m} u_{ig}$
      $(x_g' u_g)_1 = \sum_{i=1}^{m} u_{ig} x_{ig}$
    for $j = 1$ `through` k **do**
      for $h = j$ `through` k **do**
        $(x_g' u_g u_g' x_g)_{00} = (x_g' u_g)_0^2$
        $(x_g' u_g u_g' x_g)_{01} = (x_g' u_g)_0 (x_g' u_g)_1$
        $(x_g' u_g u_g' x_g)_{11} = (x_g' u_g)_1^2$
  result $= \sum_{g=1}^{G} x_g' u_g u_g' x_g$ ▷ Sum outer product for all G

is likely due to the fact that warp size for NVIDIA GPUs is set at 32, and so this would result in an extra warp being launched that would only calculate a few groups. However, the mostly-linear increase in time suggests that the main bottle-neck in the computation is memory access rather than computation. This will be in constrast to the Monte Carlo implementation.

The speed-up in this situation is compared with a linear C++ implementation using the Armadillo linear algebra library. The speed-up here is certainly not optimal, since the GPU used for the test has over 2,000 cores. However, given the memory latency of DRAM on the GPU, this speed-up is non-trivial. The speed-up is also fairly constant across different values of G, with the previously mentioned drops just above multiples of 64.

A second group of bootstrap tests are presented in Figure 4. In this case, G is kept relatively low while m is set to a very large value. This is potentially one of the most relevant situations when a GPU might be used to compute the WCB since it is consistent with many forms of big data that an econometrician might encounter. In this case, the implementation struggles when the number of thread blocks is less than 64, which is to be expected. This would be another case when increasing the number of threads and having each thread compute a partial sum of the inner product would likely add benefit. However, after $G = 64$, the time for computing the 999 summation terms increases in a fairly linear fashion. This is again consistent with the main bottleneck being memory access. Further, the relative speed-up is fairly consistent across values of G. This is comparable to the situation where where m was fixed at 32.

Finally, in Figure 4, the results for the Monte Carlo implementation are presented. As can be seen in the first chart, the time for calculation is highly sensitive to the number of groups being a multiple of 32— much more-so than the bootstrap implementation.

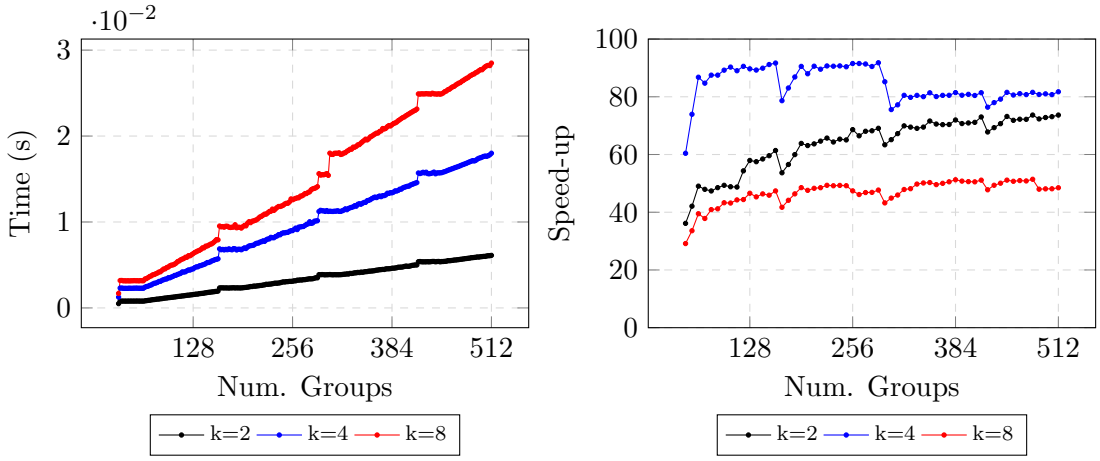

Figure 2: 999 bootstrap replications with $m = 32$

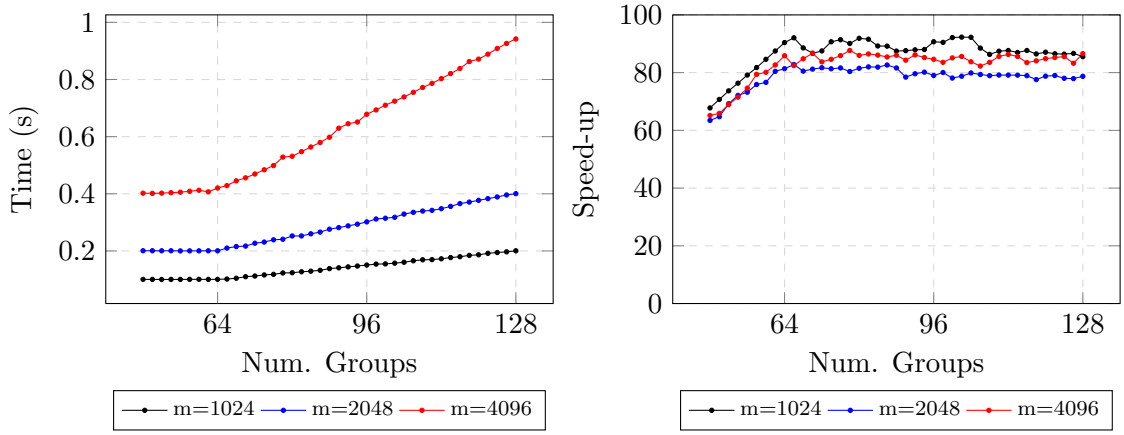

Figure 3: 999 bootstrap replications with $k = 8$

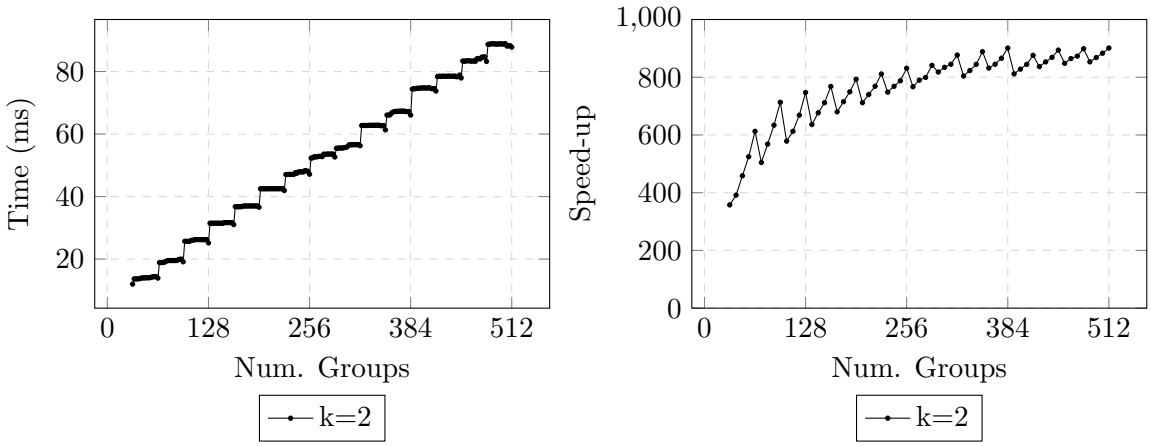

Figure 4: 50,000 Monte Carlo replications with $k = 8$ and $m = 32$

This is to be expected, since the threads never need to access DRAM and so the calculation should be entirely compute-bound. Therefore, when the number of groups is not a multiple of 32 and some number of threads are inactive, speed is very adversely affected. In fact, the difference in calculation time between, say, $G = 65$ and $G = 96$ is fairly negligible.

Where this implementation particularly excels is compared with a linear C++ version. Again, the comparison for the Monte Carlo was built using the Armadillo C++ linear algebra library. The speed-up remains highly sensitive to G being a multiple of 32. However, at the lower end, speed-ups were nearly 400 times. And, at the high end, speed-ups were around 900 times. This is still not optimal, since the total processors in the GPUs used was over 2,000, but it is still much better than the results in the bootstrap implementation.

Further, in econometrics research terms, these results make it possible to compute large Monte Carlo simulations for the CRVE in fractions of a second. This would make testing the limits and effectiveness of the estimator much easier than using say, a multicore implementation.

5 Conclusion

This paper showed that, while the CRVE estimator consists of a large number of small, independent problems, it can still be optimized for a GPU. In particular, by taking advantage of thread registers and the warp-shuffle instruction, the CRVE can be calculated much faster on a GPU than using a serial C++ implemtation, even one with an optimized linear algebra library.

The paper also showed that one of the main obstacles to calculating a problem such as the CRVE is memory access latency on the GPU. For calculating the CRVE as part of a bootstrap resampling, for example, when DRAM access is unavoidable, this can reduce speed-ups relative to a serial implementation by a factor of 10. When calculating the CRVE as part of a Monte Carlo simulation, however, data can be generated on-thread, and therefore the memory bottle-neck is completely avoided. In this case, the total speed-up factor for 50,000 replications can reach as high as 900.

As noted earlier, there is very likely a situation where the parallelism of the bootstrap implementation could be improved using partial sums. This would allow many replications of either a bootsrap or Monte Carlo to be computed in a much smaller number of steps using a large number of GPUs. Another further point of research would be an implementation of Monte Carlo simulations of bootstrap resampling, which is another important avenue of research and, as expected, can be extremely expensive to compute.

References

[1] Ahmad Abdelfattah, Marc Baboulin, Veselin Dobrev, Jack Dongarra, Christopher Earl, Joel Falcou, Azzam Haidar, Ian Karlin, Tzanio Kolev, Ian Masliah, et al. High-performance tensor contractions for gpus. *Procedia Computer Science*, 80:108–118, 2016.

[2] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Performance tuning and optimization techniques of fixed and variable size batched cholesky factorization on gpus. In *International Conference on Computational Science (ICCS'16)*, San Diego, CA, 06-2016 2015.9.

[3] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. On the development of variable size batched computation for heterogeneous parallel architectures. In *The 17th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2016), IPDPS 2016*, Chicago, IL, 05-2016 2016. IEEE, IEEE.

[4] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Performance, design, and autotuning of batched gemm for gpus. In *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, volume 9697, page 21. Springer, 2016.

[5] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[6] A Colin Cameron, Jonah B Gelbach, and Douglas L Miller. Bootstrap-based improvements for inference with clustered errors. *The Review of Economics and Statistics*, 90(3):414–427, 2008.

[7] A Colin Cameron, Douglas L Miller, et al. Robust inference with clustered data. *Handbook of empirical economics and finance*, pages 1–28, 2010.

[8] Aurelien Cassagnes, Yu Chen, and Hirotada Ohashi. Shuffle up and deal: accelerating gpgpu monte carlo simulation with application to option pricing. *Concurrency and Computation: Practice and Experience*, 27(17):5203–5213, 2015.

[9] Tingxing Dong, Azzam Haidar, Piotr Luszczek, James Austin Harris, Stanimire Tomov, and Jack Dongarra. Lu factorization of small matrices: Accelerating batched dgetrf on the gpu. In *16th IEEE International Conference on High Performance Computing and Communications (HPCC)*, Paris, France, 08-2014 2014. IEEE, IEEE.

[10] Tingxing Dong, Azzam Haidar, Piotr Luszczek, Stanimire Tomov, Ahmad Abdelfattah, and Jack Dongarra. Magma batched: A batched blas approach for small matrix factorizations and applications on gpus. Technical report, 08/2016 2016.

[11] Tingxing Dong, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. A fast batched cholesky factorization on a gpu. In *International Conference on Parallel Processing (ICPP-2014)*, Minneapolis, MN, 09-2014 2014.

[12] Jack Dongarra, M Abalenkovs, A Abdelfattah, M Gates, A Haidar, J Kurzak, P Luszczek, S Tomov, I Yamazaki, and A YarKhan. Parallel programming models for dense linear algebra on heterogeneous systems. *Supercomputing frontiers and innovations*, 2(4):67–86, 2016.

[13] Kenneth A Froot. Consistent covariance matrix estimation with cross-sectional dependence and heteroskedasticity in financial data. *Journal of Financial and Quantitative Analysis*, 24(03):333–355, 1989.

[14] Azzam Haidar, Tingxing Dong, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Batched matrix computations on hardware accelerators based on gpus. *International Journal of High Performance Computing Applications*, 29:2, 2015.

[15] Chetan Jhurani and Paul Mullowney. A gemm interface and implementation on nvidia gpus for multiple small matrices. *Journal of Parallel and Distributed Computing*, 75:133–140, 2015.

[16] Martin Kruliš, David Bednárek, and Michal Brabec. Improving parallel processing of matrix-based similarity measures on modern gpus. In *International Conference on Similarity Search and Applications*, pages 283–294. Springer, 2015.

[17] Jakub Kurzak, Hartwig Anzt, Mark Gates, and Jack Dongarra. Implementation and tuning of batched cholesky factorization and solve for nvidia gpus. 2015.

[18] Weifeng Liu and Brian Vinter. Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. *Parallel Computing*, 49:179–193, 2015.

[19] Yongchao Liu and Bertil Schmidt. Lightspmv: faster csr-based sparse matrix-vector multiplication on cuda-enabled gpus. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 82–89. IEEE, 2015.

[20] M Graham Lopez and Mitchel D Horton. Batch matrix exponentiation. In *Numerical Computations with GPUs*, pages 45–67. Springer, 2014.

[21] Justin Luitjens. Faster parallel reductions on kepler. `https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/`, 2014. Accessed: 2016-12-16.

[22] Marco Maggioni. *Sparse Convex Optimization on GPUs*. PhD thesis, Politecnico di Milano, 2015.

[23] Marco Maggioni and Tanya Berger-Wolf. Optimization techniques for sparse matrix–vector multiplication on gpus. *Journal of Parallel and Distributed Computing*, 93:66–86, 2016.

[24] Ian Masliah, Ahmad Abdelfattah, A Haidar, S Tomov, Marc Baboulin, J Falcou, and Jack Dongarra. High-performance matrix-matrix multiplications of very small matrices. In *European Conference on Parallel Processing*, pages 659–671. Springer, 2016.

[25] Brent R Moulton. An illustration of a pitfall in estimating the effects of aggregate variables on micro units. *The Review of Economics and Statistics*, 72(2):334–338, 1990.

[26] CUDA NVIDIA. Curand library, 2010.

[27] CUDA NVIDIA. Basic linear algebra subroutines (cublas) library, 2013.

[28] Lukas Polok and Pavel Smrz. Fast linear algebra on gpu. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 439–444. IEEE, 2012.

[29] William Rogers. Regression standard errors in clustered samples. *Stata technical bulletin*, 3(13), 1994.

[30] David Roodman. BOOTTEST: Stata module to provide fast execution of the wild bootstrap with null imposed. Statistical Software Components, Boston College Department of Economics, December 2015.

[31] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based c++ library for linear algebra. *Journal of Open Source Software*, 1:26, 2016.

[32] Halbert White. A Heteroskedasticity-Consistent Covariance Matrix Estimator and a Direct Test for Heteroskedasticity. *Econometrica*, 48(4):817–38, May 1980.

[33] Achim Zeileis. Econometric computing with hc and hac covariance matrix estimators. *Journal of Statistical Software*, 11(10):1–17, 2004.