Bioinformatics, 36(22-23), 2020, 5392–5397
doi: 10.1093/bioinformatics/btaa1007
Advance Access Publication Date: 8 December 2020
Original Paper

OXFORD

Gene expression

Parallelized calculation of permutation tests

Markus Ekvall[1], Michael Höhle[2] and Lukas Käll[1,*]

[1]Science for Life Laboratory, School of Engineering Sciences in Chemistry, Biotechnology and Health, KTH – Royal Institute of Technology, 171 21 Solna, Sweden and [2]Department of Mathematics, Stockholm University, 106 91 Stockholm, Sweden

*To whom correspondence should be addressed.

Associate Editor: Pier Luigi Martelli

Received on April 15, 2020; revised on November 9, 2020; editorial decision on November 18, 2020; accepted on November 19, 2020

Abstract

Motivation: Permutation tests offer a straightforward framework to assess the significance of differences in sample statistics. A significant advantage of permutation tests are the relatively few assumptions about the distribution of the test statistic are needed, as they rely on the assumption of exchangeability of the group labels. They have great value, as they allow a sensitivity analysis to determine the extent to which the assumed broad sample distribution of the test statistic applies. However, in this situation, permutation tests are rarely applied because the running time of naïve implementations is too slow and grows exponentially with the sample size. Nevertheless, continued development in the 1980s introduced dynamic programming algorithms that compute exact permutation tests in polynomial time. Albeit this significant running time reduction, the exact test has not yet become one of the predominant statistical tests for medium sample size. Here, we propose a computational parallelization of one such dynamic programming-based permutation test, the Green algorithm, which makes the permutation test more attractive.

Results: Parallelization of the Green algorithm was found possible by non-trivial rearrangement of the structure of the algorithm. A speed-up—by orders of magnitude—is achievable by executing the parallelized algorithm on a GPU. We demonstrate that the execution time essentially becomes a non-issue for sample sizes, even as high as hundreds of samples. This improvement makes our method an attractive alternative to, e.g. the widely used asymptotic Mann-Whitney U-test.

Availabilityand implementation: In Python 3 code from the GitHub repository https://github.com/statisticalbiotechnology/parallelPermutationTest under an Apache 2.0 license.

Contact: lukask@kth.se

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

Permutation tests are frequently used for non-parametric testing and are incredibly valuable within computational biology, with applications within genome-wide association studies (Browning, 2008; Dudbridge and Gusnanto, 2008; Purcell et al., 2007), Pathway Analysis (Jeuken and Käll, 2018; Subramanian et al., 2005) and expression quantitative trait loci studies (Doerge and Churchill, 1996; Sul et al., 2015). Monte Carlo-based sampling techniques (Segal et al., 2018) and exact tests that derive full permutation distributions are roughly the two ways to implement the permutation test.

Exact tests are traditionally seen as unattractive for large sample sizes, as the number of permutation grows super-exponentially with the sample size. Nonetheless, Green's dynamic programming algorithm (Green, 1977), which was made explicit by others (Pagano and Tritchler, 1983; Zimmermann, 1985), partially overcome this computational problem. This algorithm is significantly less computationally demanding than the naïve approach. However, the exact test's popularity for larger sample sizes has not attracted much attention in the last couple of decades. We report here on an extension using a Graphics processing unit (GPU) implementation to compute parallelized exact tests and found it superior to the other tested alternatives in terms of speed and accuracy.

2 Algorithm

Here, we will describe our parallelization of the Green method to calculate exact tests. First, in Section 2.1, we describe the main objective, perform hypothesis testing with an exact test, then a description of the Green algorithm in Section 2.2, and, finally, a description of how to parallelize the algorithm in Section 2.2.

2.1 Hypothesis testing

Consider a two-sample hypothesis testing setting for central tendency, i.e. we want to know if the considered response in one group B is generally larger than in another group A.

Let $\boldsymbol{x}^A = (x_1^A, \ldots, x_m^A)$ and $\boldsymbol{x}^B = (x_1^B, \ldots, x_n^B)$ be two independent samples of non-negative integers valued scores from the distributions $\mathcal{D}^A$ and $\mathcal{D}^B$, respectively. So, $x_i^A \overset{\text{iid}}{\sim} \mathcal{D}^A$ and $x_i^A \in \mathbb{N}_0$ for $1 \leq i \leq m$; and $x_i^B \overset{\text{iid}}{\sim} \mathcal{D}^B$, and $x_j^B \in \mathbb{N}_0$ for $1 \leq j \leq n$. We also

form the concatenation of the samples as $\boldsymbol{x} = (x_1^A, \ldots, x_m^A, x_1^B, \ldots, x_n^B) = (x_1, x_2, \ldots, x_{n+m})$. Our interest is in investigating the hypothesis $H_0 : \mathcal{D}^A = \mathcal{D}^B$ versus the one-sided alternative that the response in group B tends to be larger than the response in group A. This alternative can be formalized for our case of discrete distributions by, e.g. introducing $p = \Pr(x^A < x^B) + \frac{1}{2}\Pr(x^A = x^B)$ and letting $H_A : p > \frac{1}{2}$ (Fay and Proschan, 2010).

A way to perform the test is to determine how extreme the observed sum of sample $\boldsymbol{x}^A$, $s_{\text{obs}} = \sum_{i=1}^{m} x_i^A$ is under the null hypothesis. Typically, one assumes a particular parametric family of distributions and computes the P value of the test as the probability of observing s_{obs} or a more extreme value in the alternative's direction under the assumption that the null-hypothesis is true. However, it is rarely possible to analytically compute this probability, and one often has to resort to asymptotic approximations. A permutation test approach to the testing problem is to assume instead that the labels A and B are exchangeable under H_0: Under the null hypothesis the distribution of the test statistic would remain the same for any permutation of $\boldsymbol{x}$. However, this property would not hold under the alternative H_A (Huang et al., 2006). One can thus calculate how frequently samples with sample sums greater or equal than s_{obs} appears when resampling from $\boldsymbol{x}$.

We can formulate the P value as $\Pr(s_{\text{obs}} \leq S|\boldsymbol{x}, H_0)$, where $\Pr(S|\boldsymbol{x}, H_0)$ is the probability mass function, and S is a random variable denoting the sum's value in the first sample under the permutation distribution. The computationally expensive part is to obtain $\Pr(S)$ which is estimated by concatenating $\boldsymbol{x}^A$ and $\boldsymbol{x}^B$ to $\boldsymbol{x} = (x_1^A, \ldots, x_m^A, x_1^B, \ldots, x_n^B) = (x_1, x_2, \ldots, x_{n+m})$, and draw all possible subsets of length m from $\boldsymbol{x}$ and count the number of occurrences of all possible sums; when the numbers of occurrences of all possible sums are available, the distribution is accessible.

Assume a random variable $\boldsymbol{x}^* = (x_1^*, \ldots, x_m^*)$, that is a randomly sampled subset from $\boldsymbol{x}$ with j elements and its corresponding sum is $S = \sum_{i=1}^{m} x_i^*$, where $S \in [0, s_{\max}]$. Define $N[s, m]$ to be the number of ways we can sample subsets $\boldsymbol{x}^*$ with m elements in such a way that their sum $S = s$. Now $\Pr(S = s)$ can be expressed as the fraction of ways that a subset $\boldsymbol{x}^*$ can be sampled so that its sum ends up to as $S = s$ to the number of ways it can be sampled with any sum

$$\Pr(s) = \Pr(S = s) = \frac{N[s, m]}{\sum_{s'=0}^{s_{\max}} N[s', m]}, \tag{1}$$

Combinatorics gives us that the denominator of the above Equation 1 can be expressed as,

$$\binom{m+n}{m} = \frac{(m+n)!}{n!m!}.$$

The calculation of the numerator is intricate. A naïve algorithm that would exhaustively calculate the sum for each possible subset of size m and compare it to s, for all s, would need

$\mathcal{O}(s_{\max} \cdot \binom{m+n}{m}) = \mathcal{O}(m \cdot x_{\max} \cdot (m+n)^m) = \mathcal{O}(x_{\max} m^m)$ calculations, which becomes computational prohibitive even for moderate set sizes m. However, in Section 2.2, we will discuss an algorithm solving the problem within polynomial time.

Now, the sought P value can be calculated as,

$$\Pr(s_{\text{obs}} \leq S|\boldsymbol{x}) = \sum_{s=s_{\text{obs}}}^{s_{\max}} \frac{N[s, m]}{\binom{m+n}{m}} = \sum_{s=s_{\text{obs}}}^{s_{\max}} \Pr(s). \tag{2}$$

Alternatively, the same framework can be used to calculate the mid P value (Routledge, 1994) as,

$$p_{\text{mid}}(s_{\text{obs}}) = \frac{1}{2}\Pr(s_{\text{obs}}) + \sum_{s=s_{\text{obs}}+1}^{s_{\max}} \Pr(s). \tag{3}$$

As mentioned above, there is no closed formula to calculate $N[s, m]$. However, it is possible to develop a dynamic programming algorithm to obtain $N[s, m]$ (Pagano and Tritchler, 1983; Zimmermann, 1985), described thoroughly in the next section.

2.2 Efficient calculation of $N[s, m]$: the Green Algorithm

A dynamic programming algorithm for calculating $N[s, m]$ was first presented in Green (1977) and more in detail described by others (Gebhard and Schmitz, 1998; Zimmermann, 1985). Here, we will give a walk-through of the algorithm, mostly to describe its parallelization in Section 2.3. We also provide a simple use case of the algorithm in Supplementary Note S1.

We can find a recursive expression for $N[s, m]$ by considering a scenario where $\boldsymbol{x}^*$ is drawn instead of the full set $\boldsymbol{x}$ from a subset $\boldsymbol{x}^i = \{x_b\}_{b=1}^{i}$ consisting of only the first i features of $\boldsymbol{x}$. To do so; we first need some definitions. Define $N_i[s, j]$ as the number of ways we can sample j elements so that their sum becomes s from a subset $\boldsymbol{x}^i$. If we know how to calculate $N_i[s, j]$, we also know how to calculate $N[s, j] = N_{i=m+n}[s, j] = N[s, j]$, since $\{x_b\}_{b=1}^{m+n} = \boldsymbol{x}$. We also define $M_i[s, j]$ to be the number of ways we can sample $\boldsymbol{x}^i$ so that the sample elements sum to s and including the last element x_i in the sample.

We can now form a recursion of $N_i[s, j]$ as the number of ways to sample $\boldsymbol{x}^i$ has to be equal to the number of ways to sample $\boldsymbol{x}^{i-1}$ with the number of ways to sample $\boldsymbol{x}^i$ that include x_i. So,

$$N_i[s, j] = N_{i-1}[s, j] + M_i[s, j]. \tag{4}$$

We can express M_i in terms of N_{i-1} by noting that $M_i[s, j] = N_{i-1}[s - x_i, j - 1]$ when $x_i \leq s$, and otherwise zero. We can hence express the recursion 4 as,

$$N_i[s, j] = N_{i-1}[s, j] + N_{i-1}[s - x_j, j - 1], \text{ if } j > 0 \text{ and } x_i \leq s. \tag{5}$$

Let's now turn to the boundary conditions of this recursion. The empty set $\varnothing$ trivially reach the sum $s = 0$, thus

$$N_i[s, j] = 1 \quad \text{if } j = \text{s} = 0. \tag{6}$$

We cannot sample j from i elements if $j > i$. Also, $N_i[s, j]$ has to be zero when either $i \leq 0$ (i.e. checking the empty set of $\boldsymbol{x}$), when $s < 0$, or when $s_{\max} < s$ (i.e. when s is outside the boundary of possible sums). Hence,

$$N_i[s, j] = 0, \quad \text{if } i < j, i = 0, s < 0, \text{ or } s_{\max} < s. \tag{7}$$

By combining the base case 7 and 6 with the generic subrecursion 5, the final recursion is,

$$N_i[s, j] = \begin{cases} 1, & \text{if } j = s = 0. \\ 0, & \text{if } i < j, i = 0, j \leq 0, \text{ or } s < 0. \\ N_{i-1}[s, j] + & \\ +N_{i-1}[s - x_j, j - 1], & \text{otherwise.} \end{cases} \tag{8}$$

The pseudo-code of the top-down dynamic programming code of the recursion in Equation 8 is given in Supplementary Algorithm S1. The algorithm needs some explanation. Instead of using one extensive array $N[0 \ldots s_{\max}, 0 \ldots (m+n), 0 \ldots m]$, two smaller arrays, $N_{old}[0 \ldots s_{\max}, 0 \ldots m]$ and $N_{new}[0 \ldots s_{\max}, 0 \ldots m]$, are swapped and rewritten in each iteration of i in an oscillatory fashion—to save memory, see line 24. It is the complete rewriting of N_{new} in the next iteration that makes this possible (any of the conditions in the recursion relation will re-calculate each entry N_{new}).

We save quite some memory by keeping N_{new} and N_{old} instead of the full three-dimensional array. The former two arrays require $\mathcal{O}(s_{\max} \cdot m + s_{\max} \cdot m) = \mathcal{O}(2 \cdot x_{\max} m \cdot m) = \mathcal{O}(x_{\max} \cdot m^2)$, whereas the latter array requires $\mathcal{O}(s_{\max} \cdot (m+n) \cdot m) = \mathcal{O}(x_{\max} \cdot m^3)$. Moreover, this improvement in memory usage is a quintessential difference for the parallelized algorithm (GPU's memory storage can easily be a bottleneck).

A second point, notice the structure of the for-loops; they could easily have been arranged in whatever way and still obtain correct computations. However, the dimensions of the two arrays have to be adjusted appropriately related to the outer-most loop. Nevertheless, this specific order has a purpose; it is parallelizable—described in the next section.

One final note on Supplementary Algorithm S1, by plain observation on the nested loops, it easy to see that the running time is $\mathcal{O}(m \cdot (m+n) \cdot s_{\max}) = \mathcal{O}(x_{\max} \cdot m^3)$.

2.3 Parallelization over two dimensions: *s* and *j*

The meaning of parallelizing over two dimensions is, in this case, to fix one of the variables and check whether the two other variables are parallelizable given the third fixed variable. In practice, the fixed variable is the outer-most for-loop, and for each iteration of this variable, then within this loop, everything is calculated in parallel over the two other variables.

Out of the three variables, it is only necessary to find one such variable to fix, and instead of exhaustively checking all possibilities, one can check recursion 9 to see which variable is parallelizable. By comparing the left side to the right side, the only variable that is not dependent on contemporary action of itself is variable i, i.e. $N_i[s,j] = f(s, s-x_j, i-1, j, j-1)$, and, furthermore, the other two variables are not possible to fix. Below is a verification that $N_i[s,j]$ is parallelizable given that i is fixed.

Initiation: In the lines 6–9 in Supplementary Algorithm S1, constants are set, and initialization of both arrays, $N_{new}[0\ldots s_{\max}, 0\ldots m]$ and $N_{old}[0\ldots s_{\max}, 0\ldots m]$, occur. There is no conflict for the parallelization of this step.

Maintenance: $1 \leq i \leq (m+n)+1$ Consider iteration i. In lines 13–16, the array N_{new} is only dependent on constants (i.e. the boundary conditions 8 and 7). Thus, the computation of N_{new} is parallelizable. Furthermore, in lines 17–20, N_{new} is only dependent on elements from N_{old} and x, which both are invariant for $i=1$ (except for N_{old}, that switch values at the end of the iteration i, however, all computations for i are already done). Therefore, N_{new} is parallelizable here too. Finally, at line 24, N_{old} is switched with N_{new}, and no parallelization occurs here. Hence, the algorithm is parallelizable for iteration i.

When entering the next iteration of i, i.e. $i \leftarrow i+1$, the same arguments above apply.

Termination: $i = (m+n)+2$ When arriving at line 10 in Supplementary Algorithm S1 and $i=(m+n)+2$, it will not enter the loop. By the maintenance of Algorithm 2.3, one can be sure that the computation of $N(0\ldots s_{\max}, (m+n), m)$ is correct. Since there is no computation after the for-loop-block, hence, there are no more modifications on $N(0\ldots s_{\max}, (m+n), m)$, and it is safe to return this array.

2.4 Discretization of real numbered samples into integer valued samples

Our test is defined for integer valued distributions of the scores, $\mathbf{x}$. However, approximately we may use the procedure if we first discretize any real value distributed scores, $\boldsymbol{y}^A$ and $\boldsymbol{y}^B$, where $\boldsymbol{y} = (y_1^A, \ldots, y_m^A, y_1^B, \ldots, y_n^B) = (y_1, y_2, \ldots, y_{n+m})$. This was achieved by partitioning the samples range into n_w discretization windows, each of length, $l_w = \frac{|\max(\boldsymbol{y}^A,\boldsymbol{y}^B) - \min(\boldsymbol{y}^A,\boldsymbol{y}^B)|}{n_w - 1}$. Each of these windows covers $\mathcal{I}_i = [\min(\boldsymbol{y}^A,\boldsymbol{y}^B) + (i-\frac{1}{2})l_w, \min(\boldsymbol{y}^A,\boldsymbol{y}^B) + (i+\frac{1}{2})l_w]$, for $i = 0,\ldots,n_w-1$, which enables us to map any continues sample into discrete scores, as $x = x' \ni y \in \mathcal{I}_{x'}$, with values in the interval $x \in [0, n_w - 1]$. Unfortunately, this comes to the cost of discretization errors, which will be a function of the number of discretization windows, n_w. We will investigate the effects of such discretization in Section 4.

3 System and methods

3.1 Compared methods

A couple of methods were used as comparison to our implementation. We implemented t tests through the function scipy.stats.ttest_ind, and Mann-Whitney U tests through scipy.stats.mannwhitneyu, both functions in scipy version 1.4.1. We downloaded the FastPerm method from the master branch of https://github.com/bdsegal/fastPerm. When executing FastPerm we applied the default parameter-tuning as described in the implementation guide, available at the method's GitHub repository. We implemented the r-package Coin (https://CRAN.R-project.org/package=coin) version of the shift-method (exact test) as part of our python package and used it as a benchmark.

3.2 System

Performance figures were recorded on a 8 core Intel i7-9700K with an NVIDIA GeForce RTX 2070 graphics card. Some of the experiments we compared this configuration's performance to similar computers equipped with NVIDIA GeForce RTX 2060 and one with a NVIDIA Titan X Pascal graphics card.

3.3 Implementation

A python 3.6 implementation implementing the below algorithms was made available under an Apache 2.0 license. We implemented our algorithm together with our discretization strategy as a CUDA (Chakrabarti et al., 2012) enabled Python module. The implementation and all results of this paper are available in reproducible form from a GitHub repository https://github.com/statisticalbiotechnology/parallelPermutationTest.

4 Results

We implemented the algorithm described above, and set out to characterize the algorithm's performance. To establish that the strategy executes in a practically useful time scale, we first tested the running time performance as a function of the number of discretization windows and its dependence on sample size. Subsequently, we tested the accuracy of our discretization strategy to establish that it is asymptotically unbiased and precise compared to other methods. Finally, we applied the method to a relatively large-scale proteomics dataset to establish the methods' usefulness in a practical test scenario.

For some of the tests, we were able to benchmark our GPU implementation of the Green algorithm (named Green Cuda in the following text) against other methods. Here we implemented a single-thread version of the Green algorithm (Green Singlethread) on the CPU and a multi-thread version of Green algorithm (Green Multithread) on the CPU and downloaded a previously described Monte Carlo-based sampling method called the fast permutation method, FastPerm (Segal *et al.*, 2018), the shift algorithm's implementation in the popular r-package Coin (Hothorn et al., 2006; Hothorn et al., 2008) for exact permutation test (here named Shift Coin), and used Python scipy's implementation of the t test and Mann-Whitney U test.

4.1 Test of running time requirements

4.1.1 Running time as a function of sample size, *n*

We first tested the running time requirements of the testes methods as a function of sample size. Here, we selected uniformly distributed samples, $y^A \sim \mathcal{U}(0,n)$ and $y^B \sim \mathcal{U}(0,n)$, and drew samples of size $n = |\mathbf{y^A}| = |\mathbf{y^B}| \in [50,100,200,250,300,350,400,450,500] = \mathbf{n}$. For each $n \in \mathbf{n}$, using five samples replicates. The average time to calculate the corresponding P values was recorded and plotted as a function of $\mathbf{n}$ (Fig. 1). For the tested sample sizes, Green Cuda scales well with sample size, and for all sample sizes, Green Cuda was found between 15 and 50 times faster than FastPerm, see Figure 1b at $n=300$ and $n=100$, respectively. However, they seem to reach a similar running time around $n=500$.

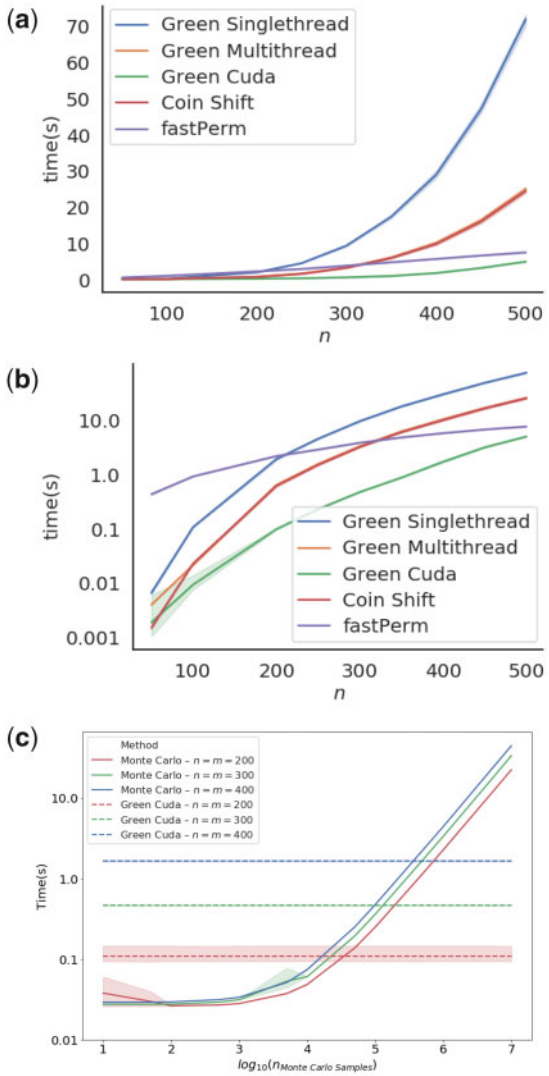

Fig. 1. Running time requirements of the compared algorithms. We plotted the time required to calculate P values for samples from $y^A \sim \mathcal{U}(0, n)$ and $y^B \sim \mathcal{U}(0, n)$ for different sample sizes **n**. The mean time and the 95% confidence interval around the mean time to calculate each of 5 replicate samplings was plotted in (**a**) linear and (**b**) log scale. It should be noted that the execution times where highly reproducible, and it might be hard to see the confidence interval in the plots. (**c**) We further investigated the running time as a function of the number of Monte Carlo samples for a MC-based approach. We compared running time for a Monte Carlo sampled to draw samples of $|y_i^A| = |y_i^B| = m = n \in [200, 300, 400]$, with five replicates. The horizontal lines represent the running time for Green Cuda to compute the same samples

The Green Singlethread should have a running time that expands as $\mathcal{O}(n_w n^3) = \mathcal{O}(n^3)$ as n_w was held constant in the experiment. This corresponds well with our observations in Figure 1.

Some Monte Carlo-based approaches are, unlike e.g. FastPerm, leaving the choice of the number of sampled permutations to their users. The number of permutations is inversely proportional to the granularity of the P values estimated by the MC-sampler, and hence governs their possible precision, at least when no other techniques such as importance sampling is used. We hence measure the running time for different amounts of permutation samples with the MC-sampling functionality of the package Coin, putting it in the context of the running time of Green Cuda (Fig. 1c). The two methods seem to have similar running times for about 10^5–10^6 permutation samples, tentatively suggesting that the Green Cuda will be the faster method of the two methods, whenever a precision of the estimated P values is desired to be better than 10^{-5}–10^{-6}.

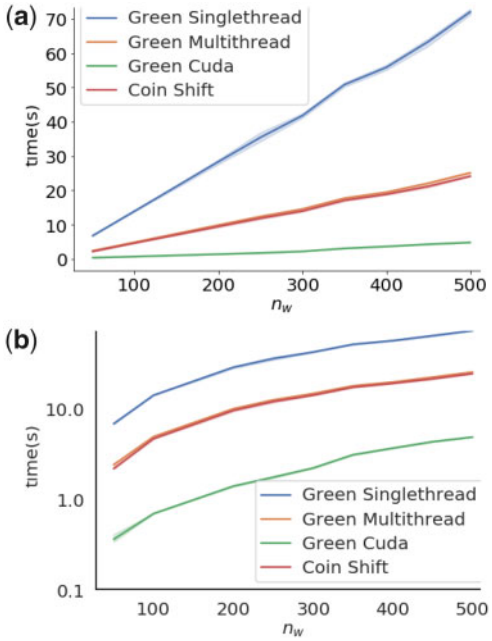

Fig. 2. Running time as a function of the number of discretization windows. We plotted the mean of the mean and standard deviation of the required running time, as wall time, as a function of the number of discretization windows, n_w. Time was plotted (**a**) in normal scale, and (**b**) log-scale. Note that the similarity in execution speed makes it hard to separate the series for Green Multithread and Coin Shift. Also note that the other methods were excluded from the plot, as the discretization step is exclusively present in the Green algorithm

Furthermore, to see how the running time depends on the hardware, the test was repeated on other GPUs. However, we found a relatively small difference in performance between the tested graphic cards (see Supplementary Fig. S1).

4.1.2 Running time as a function of the number discretization windows, n_w

We also wanted to establish that our implementation scales well with the number of discretization windows used for the test. Again, we sampled from $y^A \sim \mathcal{U}(0, 500)$ and $y^B \sim \mathcal{U}(0, 500)$ with sample size $n = m = 500$, with five replicates. We plotted the running time as a function of $n_w \in \mathbf{n_w} = [50, 100, 200, 250, 300, 350, 400, 450, 500]$ in Figure 2.

The single-threaded implementation of the Green algorithm (Green singlethread) should theoretically have a running time complexity $\mathcal{O}(n_w n^3)$. As sample size n is kept constant in the experiment, we expect the running time to expend as $\mathcal{O}(n_w)$. Indeed, Figure 2 confirms this for Green singlethread.

4.2 Memory allocation

4.2.1 Test of memory allocation

We characterized the memory requirements of Green Cuda by increasing sample size n and m, and a growing number of bins n_w. The first experiment (Fig. 3a), varied the set sizes $|\mathbf{y}^A| = |\mathbf{y}^B| = m = n \in \mathbf{n} = [10, 20, 30, \ldots, 480, 490, 500]$, for three different bin sizes n_w of 64, 128 and 256. In the second experiment (Fig. 3b), the number of bins was the variable $n_w \in \mathbf{n_w} = [10, 12, 14, \ldots, 396, 398, 400]$ for three different set sizes $|\mathbf{y}^A| = |\mathbf{y}^B| = n = m$ of 125, 250 and 500. In both experiments, the data was sampled from $y^A \sim \mathcal{N}(0, 1)$ and $y^B \sim \mathcal{N}(0, 1)$ and the number replicates was 1000.

The amount of data that can be handled by a GPU at each point in time is limited by the GPU's memory size. Here we used an NVIDIA GeForce RTX 2070, which allows for 7982 MiB memory allocation. One could imagine settings where $s_{\max}$ is so large that one cannot calculate N even for one data point. However, we would

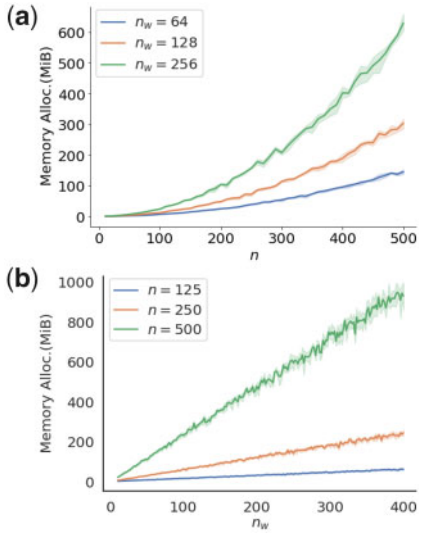

Fig. 3. Memory allocation as a function of set size and bin size for Green Cuda. We plotted the required memory allocation to calculate P values for samples from $y^A \sim \mathcal{N}(0,1)$ and $y^B \sim \mathcal{N}(0,1)$ for different sample sizes $\mathbf{n}$ and bin sizes $\mathbf{n_w}$. In (**a**) the n_w used were 64, 128 and 512 with 5 replicates, and in (**b**) n were 125, 250 and 500 with 5 replicates

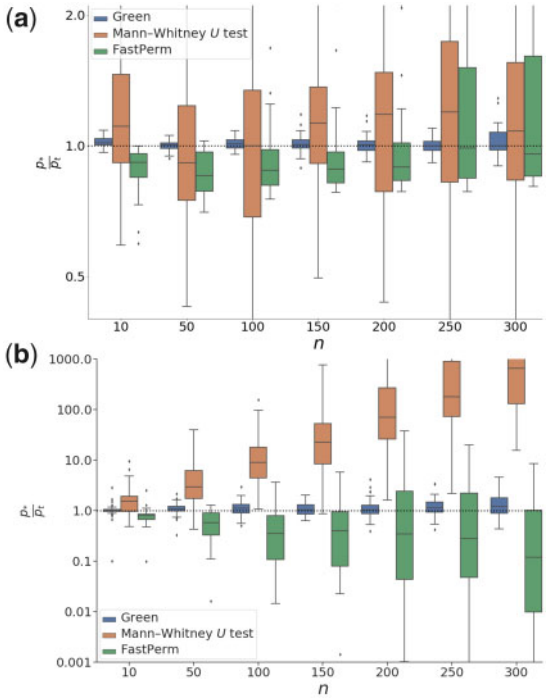

Fig. 4. Comparison of estimation error as a function of sample size. We plotted the fold change between one side Green Cuda, FastPerm and Mann-Whithey U test and on the other side a t test, as a function of the sample size, n, when $y^A \sim \mathcal{N}(0.0,1)$ and (a) $y^B \sim \mathcal{N}(0.2,1)$, and (b) $y^B \sim \mathcal{N}(1.0,1)$. For both cases we plotted results from 50 samples, and used $n_w = 100$ discretization windows. Note that the estimation errors for Green Multithread, Green Multithread, Green Cuda and Coin Shift are identical, so we compressed the reults into onse series, labeled Green

instead run into floating-point problems before reaching this type of memory problem for such cases. For large set-sizes, $1000 < |\mathbf{y^A}| + |\mathbf{y^B}|$, the sums in the entries in N start go beyond the maximum double-point precision in CUDA i.e. $\approx 1.79e^{308}$. A future improvement of our algorithm could be to reduce the values in each iteration by dividing all elements in $N_i[s,j]$ by a normalizing factor in each iteration over i, which would help us reduce the problems of N becoming too large.

4.3 Test of accuracy and precision

4.3.1 Test dependency of window size

While the Green Algorithm is a non-parametric test for discrete test statistics, the algorithm's performance will be a function of the number of windows, n_w, we use for discretizing any continuous data. We hence wanted to characterize the influence of n_w on the accuracy of our test. We selected samples from a Normal distribution and compared the computed P values with the ones from a regular t test (Supplementary Fig. S2). The results suggest that both the accuracy and precision of the test improves when increasing the number of windows. However, the effect seems to saturate for $n_w > 30$.

4.3.2 Comparison to other method's accuracy and precision against comparative methods

We subsequently wanted to test the accuracy of the estimated P values. Again we used Normal distributed samples and used t test-calculated P values as reference. As benchmark comparisons, we again used the FastPerm method and Python's asymptotic Normal distribution implementation of the Mann-Whitney U test, scipy.stats.mannwhitneyu. We plotted the ratio, $\frac{p_*}{p_t}$, where p_* is the tested P value and p_t is given by a t test, as a function of sample size for two different effect sizes (Fig. 4). Overall, the $\frac{p_*}{p_t}$ of the Green Cuda were found closer to 1, and less dependent on the sample size than the ones from the compared methods. The reason for the Mann-Whitney U test deviating from the results of the t test, particularly in Figure 4b, is that the test has comparatively low efficiency when testing on normal distributed data. It is also important to note that the Mann-Whitney U test, which depends on ranking statistics, would not be under the same null model as the compared methods in the presence of ties. However, we expect ties to be rare when sampling from continuous distributions.

Table 1. Running time for Green Cuda on the proteomics dataset as a function of the number of discretization windows n_w

n_w	16	64	256	512	1028
Time (s)	6, 31	6, 87	10, 8	17, 5	37

4.3.3 Calibration test

We also tested the parallelized shift method's P values uniformity under the null hypothesis (Murdoch et al., 2008). This property is of particular importance for studies where we test many variables for the same sample, as it is the base for efficient multiple hypothesis testing (Efron, 2012; Storey and Tibshirani, 2003). Here, we compared Green Cuda against the FastPerm method, Mann-Whitney U test and a regular t test. We picked 10000 samples from the Normal distribution and a log-Normal distribution and plotted each method has estimated P values as a function of their relative rank (Supplementary Fig. S3). We found that the calibration of the parallel Green method was on par with the $t$est. However, unsurprisingly the calibration seems to be entirely off for the t tests on log-Normal distributed data. We see that the Mann-Whitney U test's calibration appears conservative, while the FastPerm method appears anti-conservative for both tested distributions.

4.4 Running time requirements for a proteomics dataset

As the last test, we tested the algorithm's performance on a dataset of breast cancer samples from the CPTAC consortium (NCI CPTAC, 2016). For the 8051 proteins for which measurements had been obtained for all samples, we tested differential abundance between 80 non-triple-negative and 26 triple-negative samples. The run-time for Green Cuda method can be found in Table 1. For the other methods: FastPerm took 45 min, 1.5 s for a Mann-Whitney U test and 1.32 s for a t test.

5 Discussion

Statistical testing is the base for most scientific activities. Also, in most research areas, the amount of public data is rapidly increasing, and hence there is a need for ever more efficient methods to compute significance. Permutation tests offer an exciting method as they do not assume a particular sampling distribution, but instead, build one by permuting label associations to the observed data. This approach corresponds perfectly with the null hypothesis that there is no difference in case and control outcomes.

Here, we have described a parallelized dynamic programming method to perform permutation tests. We have demonstrated that it is faster and more accurate than the sampling-based methods. Previous work by Pagano and Trichler (1983) demonstrates that one can quickly expand exact tests to handle missing values, something that rank-based not easily can handle. We note that several studies are dependent on normal approximations of non-parametric tests such as the Mann-Whitney *U* test. In practice, the implementation of such tests is approximations as they are asymptotic and not exact. The Green Cuda method offers an exact test that does not appear much slower but more accurate than such tests. However, admittedly, the difference between the outcomes of asymptotic and permutation tests gets smaller with an increased sample size.

Permutation tests have been successfully used in many, if not most areas of bioinformatics as a relatively assumption-free method for assessing statistical significance in inferences. In most applications the procedures involve some flavor of Monte Carlo-based sampling methodology. Here we demonstrated that for at least in the case of statistical hypothesis testing on can instead rely on calculating a full distribution of the sampling space.

Funding

This research was supported by grants from the Swedish Research Council [2017-04030] and Swedish Foundation for Strategic Research [BD15-0043], as well as a donation of the Titan X Pascal GPU used for this research from the NVIDIA Corporation.

Conflict of Interest: none declared.

References

Browning,B.L. (2008) Presto: rapid calculation of order statistic distributions and multiple-testing adjusted p-values via permutation for one and two-stage genetic association studies. *BMC Bioinformatics*, **9**, 309.

Chakrabarti,G. *et al.* (2012) Cuda: compiling and optimizing for a GPU platform. *Procedia Comput. Sci.*, **9**, 1910–1919.

Doerge,R.W. and Churchill,G.A. (1996) Permutation tests for multiple loci affecting a quantitative character. *Genetics*, **142**, 285–294.

Dudbridge,F. and Gusnanto,A. (2008) Estimation of significance thresholds for genomewide association scans. *Genet. Epidemiol. Off. Public. Int. Genet. Epidemiol. Soc.*, **32**, 227–234.

Efron,B. (2012) *Large-Scale Inference: Empirical Bayes Methods for Estimation, Testing, and Prediction*, **Vol. 1**. Cambridge, UK: Cambridge University Press.

Fay,M.P. and Proschan,M.A. (2010) Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules. *Stat. Surv.*, **4**, 1–39.

Gebhard,J. and Schmitz,N. (1998) Permutation tests – a revival? *Stat. Papers*, **39**, 75–85.

Green,B.F. (1977) A practical interactive program for randomization tests of location. *Am. Stat.*, **31**, 37–39.

Hothorn,T. *et al.* (2006) A Lego system for conditional inference. *Am. Stat.*, **60**, 257–263.

Hothorn,T. *et al.* (2008) Implementing a class of permutation tests: the coin package. *J. Stat. Softw.*, **28**, 1–23.

Huang,Y. *et al.* (2006) To permute or not to permute. *Bioinformatics*, **22**, 2244–2248.

Jeuken,G.S. and Käll,L. (2018) A simple null model for inferences from network enrichment analysis. *PLoS One*, **13**, e0206864.

Mertins,P. *et al.*; NCI CPTAC. (2016) Proteogenomics connects somatic mutations to signalling in breast cancer. *Nature*, **534**, 55–62.

Murdoch,D.J. *et al.* (2008) P-values are random variables. *Am. Stat.*, **62**, 242–245.

Pagano,M. and Tritchler,D. (1983) On obtaining permutation distributions in polynomial time. *J. Am. Stat. Assoc.*, **78**, 435–440.

Purcell,S. *et al.* (2007) PLINK: a tool set for whole-genome association and population-based linkage analyses. *Am. J. Hum. Genet.*, **81**, 559–575.

Routledge,R.D. (1994) Practicing safe statistics with the mid-p. *Can. J. Stat.*, **22**, 103–110.

Segal,B.D. *et al.* (2018) Fast approximation of small p-values in permutation tests by partitioning the permutations. *Biometrics*, **74**, 196–206.

Storey,J.D. and Tibshirani,R. (2003) Statistical significance for genomewide studies. *Proc. Natl. Acad. Sci. USA*, **100**, 9440–9445.

Subramanian,A. *et al.* (2005) Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. *Proc. Natl. Acad. Sci. USA*, **102**, 15545–15550.

Sul,J.H. *et al.* (2015) Accurate and fast multiple-testing correction in EQTL studies. *Am. J. Hum. Genet.*, **96**, 857–868.

Zimmermann,H. (1985) Exact calculation of permutational distributions for two independent samples. *Biometrical J.*, **4**, 431–434.