Is Your Permutation Algorithm Unbiased for $n \neq 2^m$?

Michael Waechter[1], Kay Hamacher[2], Franziska Hoffgaard[2], Sven Widmer[1], and Michael Goesele[1]

[1] GRIS, TU Darmstadt, Germany
[2] Bioinformatics and Theoretical Biology, TU Darmstadt, Germany

Abstract. Many papers on parallel random permutation algorithms assume the input size n to be a power of two and imply that these algorithms can be easily generalized to arbitrary n, e.g., by padding the input array to a power of two. We show that this simplifying assumption is not necessarily correct since it may result in a bias (i.e., not all possible permutations are generated with equal likelihood). Many of these algorithms are, however, consistent, i.e., iterating them ultimately converges against an unbiased permutation. We prove this convergence along with proving exponential convergence speed. Furthermore, we present an analysis of iterating applied to a butterfly permutation network, which works in-place and is well-suited for implementation on many-core systems such as GPUs. We also show a method that improves the convergence speed even further and yields a practical implementation of the permutation network on current GPUs.

Keywords: parallel random permutation, butterfly network, bias, consistency, GPU

1 Introduction

Parallel generation of random permutations is an important building block in parallel algorithms. It can, e.g., be used to perturb the input of a subsequent algorithm in order to make worst-case behavior unlikely [3]. It is also useful in statistical applications where a sufficient number of sample permutations is generated in order to draw conclusions about every possible input order. This is, e.g., the most important step in the bootstrapping procedure often applied in statistical science and modeling [16,17], in particular in bioinformatical phylogenetic reconstruction [14,7].

Divide and conquer is a commonly used design paradigm. With this paradigm, it is convenient to assume that the input array size n is a power of two. This assumption is frequently used (e.g., in [15,5]) to simplify the notation of the algorithm or its proof of unbiasedness, without pointing out an unbiased method for generalization to arbitrary n. In this paper we argue that this simplification may be too strong and inadmissible.

We demonstrate a butterfly style [10, Sec. 3.2] permutation network, that is well-suited for parallelization on a many-core machine with lots of processing elements (i.e., a number close to the problem size). If this algorithm is generalized to arrays, whose size is not a power of two, the algorithm does not generate all possible permutations with equal likelihood. As this algorithm and its method of generalization to arbitrary n is not pathological but seems rather natural, this issue needs to be resolved.

Our main contribution is to demonstrate and resolve this issue by showing that iterative application of any permutation algorithm, whose corresponding permutation matrix is positive, converges against an unbiased permutation, i.e., the algorithm is consistent. Furthermore we show that with an increasing number of iterations the bias diminishes exponentially. We present a method for improving the convergence behavior of the butterfly network even further and demonstrate a GPU implementation that is competitive to or even faster than a highly optimized state-of-the-art GPU algorithm.

2 Related Work

Most random permutation algorithms belong to one of five categories listed below, the first four of which are described and analyzed by Cong and Bader [3].

Rand_Sort assigns a random key to every value, and sorts the array in parallel according to these keys. Hagerup [6] gives a sorting based algorithm that runs in $\mathcal{O}(\log n)$ with n processing elements and $\mathcal{O}(n)$ space. The approach's general drawback is, however, that by construction the array is effectively doubled in size due to the sorting keys.

Rand_Dart randomly maps elements into an array of size kn with $k > 1$ (e.g., $k = 2$ [3]) and compacts the resulting array. There are two obvious drawbacks: First, space consumption is kn, and second, memory conflicts need to be resolved. This can be done by using memory locks and either re-throwing "darts" that had a collision into the same "dart board" until they find an empty cell or by throwing them onto a new "board" of smaller size.

Rand_Shuffle is essentially a parallel version of Knuth's sequential algorithm [8, Sec. 3.4.2]. Memory conflicts need to be resolved by sequentializing conflicting memory access. Anderson [1] analyzes this algorithm for machines with a small number of processing elements. For an increasing number of processing elements the likelihood of memory conflicts increases drastically, especially if the number of processing elements is of the same order as the problem size. Hagerup [6] gives a variant of this algorithm which runs in $\mathcal{O}(\log n)$ and uses n processing elements, but requires $\mathcal{O}(n^2)$ space.

Rand_Dist assigns each of the p processing elements a subset of n/p elements. Each processing element then sends all of its elements to random processing elements and sequentially permutes the elements it received. Afterwards all subsets are simply concatenated. Cong and Bader [3] show empirically that for small numbers of processing elements and in the presence of fast random number generators Rand_Dist can outperform other algorithms. However, if p is close to n the algorithm's work is mostly about redistributing the elements among the processing elements and load-balancing the work among all processing elements (i.e., ensuring that every processing element receives the same amount of work) must be traded off against implementing the algorithm in-place.

Permutation Networks: Knuth [9, Sec. 5.3.4] points out that sorting networks can be turned into permutation networks by replacing their comparers with random exchangers. Waksman [15] gives a network with size $\mathcal{O}(n \log n)$ and time $\mathcal{O}(\log n)$.

Most of the above approaches suffer from several shortcomings. If they are implemented on a system whose number of processing elements is of the same order as the problem size (e.g., many-core systems like GPUs), they either do not work in-place or

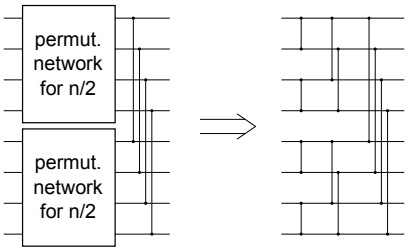

Fig. 1. Recursive construction scheme of the butterfly permutation network

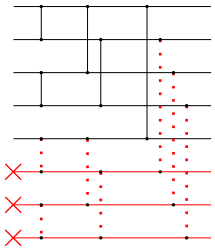

Fig. 2. Network for $n = 5$. Omitted permutations are marked dotted red.

require too much synchronization or contention resolving like memory locks. Only permutation networks seem suitable in this setting. In a permutation network each of the random exchanges, that happen in parallel in one layer of the network, can be done by one of the processing elements. This form of parallelization is extremely fine-grained, which makes it very suitable for GPU implementation. Also, the memory access pattern is determined only by the network structure and not by the result of previous computations. Therefore, it is guaranteed that no two processing elements try to access the same memory address at the same time and no contention resolving mechanism is needed.

3 A Critical Analysis of the Butterfly Permutation Network

Our algorithm is based on the butterfly network [10, Sec. 3.2]. Given an input array with size $n = 2^m$ ($m \in \mathbb{N}_0$), it recursively divides the input into two subarrays until a single element is left, permutes the subarrays, and combines them by randomly exchanging element i with element $i + \frac{n}{2}$ for all $i \in \{1, ..., \frac{n}{2}\}$ (see Fig. 1). This network has depth $\log_2 n$ and size $\frac{n}{2} \log_2 n$. It can be executed in parallel using $p = n/2$ processing elements, requires no memory locks and only $\log_2 n$ thread synchronizations. Compared to Knuth's sequential shuffling [8] the speed-up is in $\mathcal{O}(p/\log n)$ and the efficiency is in $\mathcal{O}(1/\log n)$. An important property of this algorithm is, that it works in-place (in contrast to many other algorithms, see Sec. 2). This simplifies the implementation on modern many-core systems such as GPUs, which typically have very limited shared memory with fast access.

In the following we will first prove that, if n is a power of two, the butterfly network permutes unbiased, i.e., that the probability for an element from a certain origin to be placed at a certain destination is equal for all origin/destination combinations. Afterwards we demonstrate that this is not the case for arbitrary array sizes.

3.1 Unbiasedness for $n = 2^m$

A random permutation algorithm works on an array of n elements. p_{ij} is the probability that, after a certain number of steps, element j of the original array can be found at position i. The matrix $\mathbf{M}_n = [p_{ij}]_{1 \leq i,j \leq n}$ contains all such pairwise probabilities. An

algorithm is unbiased if $p_{ij} = 1/n \; \forall i, j$ after the algorithm's termination. We show that our algorithm is unbiased for all $n = 2^m$ using induction over m.

Base case: For $m = 0$, i.e., $n = 1$ the algorithm terminates immediately. Since $i = j = 1$, $\mathbf{M}_n = [p_{1,1}] = [1] = [1/n]$ holds, which is obviously unbiased.

Induction step: The induction hypothesis is, that after $(\log_2 n) - 1$ steps the array is composed of two equally sized subarrays, both of which are permuted unbiasedly themselves. Hence, we are concerned with the following matrix:

$$\mathbf{M}_{\text{partial}} = \begin{bmatrix} 2/n & \dots & 2/n & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 2/n & \dots & 2/n & 0 & \dots & 0 \\ 0 & \dots & 0 & 2/n & \dots & 2/n \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & 2/n & \dots & 2/n \end{bmatrix}$$

In the $\log_2 n$-th step, the algorithm exchanges element i and $i + n/2$ with probability $1/2$. It follows that in the final matrix $\mathbf{M}_n$ the rows i and $i + \frac{n}{2}$, for $i \in \{1, \dots, \frac{n}{2}\}$, are the arithmetic mean of the corresponding rows from $\mathbf{M}_{\text{partial}}$:

$$\begin{aligned} [\mathbf{M}_n]_i = [\mathbf{M}_n]_{i+\frac{n}{2}} &= 1/2 \left([2/n, ..., 2/n, 0, ..., 0] + [0, ..., 0, 2/n, ..., 2/n] \right) \\ &= [1/n, ..., 1/n] \qquad \square \end{aligned}$$

3.2 Bias for $n \neq 2^m$

If n is not a power of two, two methods for generalizing the algorithm come to mind:

The first is to pad the array to the next larger power of two, do the permutation, and remove the padding in any way that preserves the relative order of the non-padding data, e.g., by using parallel prefix-sum [2]. In fact, any compaction algorithm that does not preserve the order could be used, but in that case the compaction algorithm would qualify as permutation and would thus need to be analyzed as well.

On close inspection padding proves to be biased. Via simulations with 10^7 independent runs we obtained the following matrices for $n \in \{3, 5\}$, which are clearly biased:

$$\mathbf{M}_{\text{butterf.\&pad.},3} \approx \begin{bmatrix} 0.313 & 0.313 & 0.375 \\ 0.375 & 0.375 & 0.250 \\ 0.312 & 0.312 & 0.375 \end{bmatrix} \quad \mathbf{M}_{\text{butterf.\&pad.},5} \approx \begin{bmatrix} 0.191 & 0.191 & 0.191 & 0.191 & 0.235 \\ 0.203 & 0.203 & 0.203 & 0.203 & 0.188 \\ 0.211 & 0.211 & 0.211 & 0.211 & 0.156 \\ 0.203 & 0.203 & 0.203 & 0.203 & 0.188 \\ 0.192 & 0.191 & 0.191 & 0.192 & 0.234 \end{bmatrix}$$

One might argue that the butterfly network with padding is pathological, but in fact this happens in other algorithms as well: If padding is used in Waksman's permutation network [15], (using 10^7 independent simulations) we can see the exact same effect:

$$\mathbf{M}_{\text{Waksman\&pad.},3} \approx \begin{bmatrix} 0.313 & 0.312 & 0.375 \\ 0.375 & 0.375 & 0.250 \\ 0.313 & 0.312 & 0.375 \end{bmatrix} \quad \mathbf{M}_{\text{Waksman\&pad.},5} \approx \begin{bmatrix} 0.191 & 0.191 & 0.191 & 0.191 & 0.234 \\ 0.203 & 0.203 & 0.203 & 0.203 & 0.188 \\ 0.211 & 0.211 & 0.211 & 0.211 & 0.156 \\ 0.203 & 0.203 & 0.203 & 0.203 & 0.187 \\ 0.191 & 0.191 & 0.192 & 0.191 & 0.234 \end{bmatrix}$$

The second method to generalize the butterfly network for arbitrary n is to simply use the network for the next larger power of two and omit the network's exchanges that involve non-existing array elements. An example with $n = 5$ can be seen in Fig. 2, where non-existing elements and omitted exchanges are marked in red. Using this

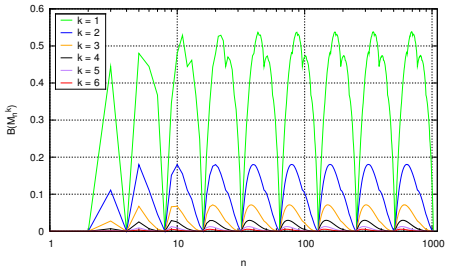

Fig. 3. Bias $\mathrm{B}(\mathbf{M}_n{}^k)$ for $n \leq 1024$ after $k = 1$ to 6 rounds of the iterating approach

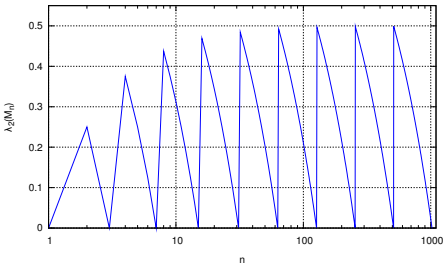

Fig. 4. Second largest eigenvalues λ_2 of $\mathbf{M}_n$ for $n \leq 1024$

method some elements in the network do not have a corresponding element they could be exchanged with, which in turn leads to a bias. E.g., for $n = 5$ we can see that the corresponding matrix is not equal to the unbiased permutation matrix:

$$\begin{bmatrix} 1&0&0&0&0\\ 0&1&0&0&0\\ 0&0&1&0&0\\ 0&0&0&1&0\\ 0&0&0&0&1 \end{bmatrix} \rightarrow \begin{bmatrix} 1/2&1/2&0&0&0\\ 1/2&1/2&0&0&0\\ 0&0&1/2&1/2&0\\ 0&0&1/2&1/2&0\\ 0&0&0&0&1 \end{bmatrix} \rightarrow \begin{bmatrix} 1/4&1/4&1/4&1/4&0\\ 1/4&1/4&1/4&1/4&0\\ 1/4&1/4&1/4&1/4&0\\ 1/4&1/4&1/4&1/4&0\\ 0&0&0&0&1 \end{bmatrix} \rightarrow \begin{bmatrix} 1/8&1/8&1/8&1/8&1/2\\ 1/4&1/4&1/4&1/4&0\\ 1/4&1/4&1/4&1/4&0\\ 1/4&1/4&1/4&1/4&0\\ 1/8&1/8&1/8&1/8&1/2 \end{bmatrix} = \mathbf{M}_5$$

The probability for an element j to end up at position i is not uniformly distributed. Especially, element 5 can only be exchanged in the last step of the butterfly network and will thus end up at position 1 or 5.

Throughout the remainder of this paper the second generalization method will be used, because it is faster than the first and preserves in-placeness.

To quantify bias, we define the following bias measure, that gives the relative deviation from a uniform random permutation matrix averaged over all matrix elements:

$$\mathrm{B}(\mathbf{M}_n) = \frac{1}{n^2} \sum_{i,j} \frac{\left|\mathbf{M}_n(i,j) - \frac{1}{n}\right|}{\frac{1}{n}} = \frac{1}{n} \sum_{i,j} \left|\mathbf{M}_n(i,j) - \frac{1}{n}\right|$$

The butterfly network's bias is shown as a function of n in the topmost curve of Fig. 3.

4 Consistency

Since many applications deal with array sizes that are not a power of two, biased permutations may lead to severe problems that are relevant in practice. We therefore show in this section that a large group of algorithms including the butterfly network are consistent even though they may be biased. An algorithm is consistent, if iterated application reduces the bias and iterated ad infinitum the bias vanishes.

Formally, the matrices $\mathbf{M}$ described in the previous section are stochastic permutation matrices. To obtain the matrix that results from applying an algorithm k times, we raise its original matrix to the k-th power. Fig. 3 displays the resulting bias $\mathrm{B}(\mathbf{M}_n^k)$ for $k \in \{1, \ldots, 6\}$ applications of the butterfly algorithm and a range of input sizes.

The figure suggests, that for $k \to \infty$ the bias converges against 0. In the following we will prove that this is indeed the case for our algorithm and actually for the larger class of all algorithms whose stochastic permutation matrix $\mathbf{M}$ is positive ($m_{ij} > 0 \ \forall i, j$) and doubly (column and row) stochastic. Furthermore, we will demonstrate that the butterfly network's matrices are positive and doubly stochastic.

4.1 Convergence

To determine the bias for $k \to \infty$, we examine the Markov chain that is associated with the matrix $\mathbf{M}$: If the Markov chain's distribution converges against a uniform distribution, the algorithm is consistent. Because $\mathbf{M}$ is row-stochastic, a vector describing a uniform distribution is an eigenvector with corresponding eigenvalue 1:

$$\mathbf{M} \cdot \left(\tfrac{1}{n}, ..., \tfrac{1}{n}\right)^T = \left(\tfrac{1}{n} \sum_j m_{1j}, ..., \tfrac{1}{n} \sum_j m_{nj}\right)^T = \left(\tfrac{1}{n}, ..., \tfrac{1}{n}\right)^T$$

Hence, the uniform distribution is a stable distribution in the Markov chain. Stability does, however, not imply that the Markov chain converges against this distribution.

Using not only $\mathbf{M}$'s row stochasticity but also column stochasticity and positiveness, the convergence can be shown using the Perron-Frobenius theorem [13,11]: It states, that all positive matrices $\mathbf{M}$ have an eigenvalue r that satisfies the condition $\min_i \sum_j m_{ij} \leq r \leq \max_i \sum_j m_{ij}$. For row stochastic matrices we obtain $r = 1$. This Perron-Frobenius eigenvalue is a simple eigenvalue and strictly greater than all other eigenvalues' absolute values. r has a corresponding right eigenvector v and a left eigenvector w. Because $\mathbf{M}$ is row stochastic,

$$\mathbf{M} \cdot \left(\tfrac{1}{\sqrt{n}}, ..., \tfrac{1}{\sqrt{n}}\right)^T = \left(\tfrac{1}{\sqrt{n}} \sum_j m_{1j}, ..., \tfrac{1}{\sqrt{n}} \sum_j m_{nj}\right)^T = \left(\tfrac{1}{\sqrt{n}}, ..., \tfrac{1}{\sqrt{n}}\right)^T$$

holds, where $(\frac{1}{\sqrt{n}}, ..., \frac{1}{\sqrt{n}})^T$ is a solution for v and (with analogous reasoning using $\mathbf{M}$'s column stochasticity) also for w. Then the Perron-Frobenius theorem states that

$$\lim_{k \to \infty} \frac{\mathbf{M}^k}{r^k} = \lim_{k \to \infty} \mathbf{M}^k = \frac{vw^T}{w^T v} = \begin{bmatrix} 1/n & \dots & 1/n \\ \vdots & \ddots & \vdots \\ 1/n & \dots & 1/n \end{bmatrix} \quad \square$$

The constraint for $\mathbf{M}$ to be doubly stochastic is not a real constraint, because every permutation algorithm's matrix is doubly stochastic. In the permutation process elements must not be lost and no new elements may be inserted into the list, hence each row and each column must sum to 1.

Therefore, any random permutation algorithm is consistent, if the probability for any element i to be moved to position j is positive.

The butterfly network's probability matrices are also doubly stochastic, but not positive, as some entries are 0. We can, however, show that ${\mathbf{M}_n}^2$ is always positive. In the following, the "$>$"-relation on matrices denotes element-wise comparisons: $\mathbf{A} > \mathbf{B} \iff a_{ij} > b_{ij} \ \forall i, j$. The algorithm's matrices contain entries greater than 0 in at least the first row and the first column (proof is omitted). It follows that

$${\mathbf{M}_n}^2 > \begin{bmatrix} m_{11} & m_{12} & \dots & m_{1n} \\ m_{21} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & 0 & \dots & 0 \end{bmatrix}^2 > \begin{bmatrix} m_{11} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & 0 & \dots & 0 \end{bmatrix} \cdot \begin{bmatrix} m_{11} & \dots & m_{1n} \\ 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix} = \begin{bmatrix} m_{11} \cdot m_{11} & \dots & m_{11} \cdot m_{1n} \\ \vdots & \ddots & \vdots \\ m_{n1} \cdot m_{11} & \dots & m_{n1} \cdot m_{1n} \end{bmatrix} > \mathbf{0}$$

for positive $m_{11}, \ldots, m_{1n}, m_{21}, \ldots, m_{n1}$. Because ${\mathbf{M}_n}^2$ is positive, $\lim_{k\to\infty} ({\mathbf{M}_n}^2)^k$ is the uniform distribution matrix.

4.2 Convergence Speed

For algorithms with positive permutation matrices $\mathbf{M}$ the convergence speed can be shown as well:
Let $\mathbf{U}$ be the uniform distribution matrix, $u = (\frac{1}{\sqrt{n}}, ..., \frac{1}{\sqrt{n}})^T$, $\mathbf{V} = (v_1, ..., v_n)$ be the matrix of $\mathbf{M}$'s eigenvectors, and $\mathbf{V}^{-1} = \mathbf{W} = (w_1, ..., w_n)^T$ be its inverse. Furthermore we assume that $\mathbf{M}$'s eigenvalues λ_l are in decreasing order. For positive, doubly stochastic matrices the Perron-Frobenius theorem ensures that $1 = r = \lambda_1 > |\lambda_l| \; \forall l \in \{2, ..., n\}$ and $v_1 = w_1^T = u$. By eigendecomposing $\mathbf{M}^k$, we obtain

$$\begin{aligned}
\|\mathbf{M}^k - \mathbf{U}\| &= \left\|v_1\lambda_1^k w_1 + ... + v_n\lambda_n^k w_n - u \cdot 1 \cdot u^T\right\| \\
&= \|u \cdot 1^k \cdot u^T + v_2\lambda_2^k w_2 + ... + v_n\lambda_n^k w_n - u \cdot 1 \cdot u^T\| \\
&\leq \sum_{l=2}^{n} |\lambda_l|^k \|v_l w_l\| \leq |\lambda_2|^k \sum_{l=2}^{n} \|v_l w_l\| \in \mathcal{O}(|\lambda_2|^k)
\end{aligned}$$

Note, that $\|v_l w_l\|$ only depends on $\mathbf{M}$ but not on k. Using $|\lambda_2| < 1$, it follows that $\|\mathbf{M}^k - \mathbf{U}\|$ decreases exponentially with k. $\square$

Since we used no assumptions besides positiveness and double stochasticity, the exponential convergence applies to all random permutation algorithms with positive permutation matrices.

Fig. 4 shows all λ_2 of the butterfly network's $\mathbf{M}_n$ for $n = 2, \ldots, 1024$. The graph displays a periodical behavior. The maximum values in each $(2^m, 2^{m+1})$-interval converge against 0.5. We note, that the local maxima can be found at $n = 2^m + 1$. This is in agreement with the fact that the maxima in every $(2^m, 2^{m+1})$-interval in Fig. 3 are further to the left for larger k. Further analysis showed that for much larger k (e.g., $k = 51$) the bias maxima are indeed located at $n = 2^m + 1$.

5 Improvement for the Butterfly Permutation Network

Even though iterating reduces the butterfly network's bias exponentially, it can still be improved by using *shifting*, i.e., the repeated application of the algorithm is interleaved with circular shifting of the data using some shifting offset l. Circular shifting moves any element i of an array to position $(i + l) \bmod n$. The underlying idea is to choose l such that array positions with a big bias are shifted to positions with a smaller bias. For $\mathbf{M}_{\text{shift},l}$ being a regular, non-stochastic permutation matrix that circularly shifts by l positions we obtain the combined permutation matrix $(\mathbf{M}_{\text{shift},l} \cdot \mathbf{M}_n)^k$.

Fig. 5 displays the bias for two array sizes and various shifting offsets. The graph for $n = 304$ reveals a global minimum for a shifting offset of 229 where the bias' magnitude is reduced to 1.8% of the bias without shifting ($l = 0$). The graph for $n = 400$ demonstrates that the bias can even exceed the bias without shifting. E.g., if we shift with an offset of 112, the bias is 3.4 times larger than without shifting. Comparing the

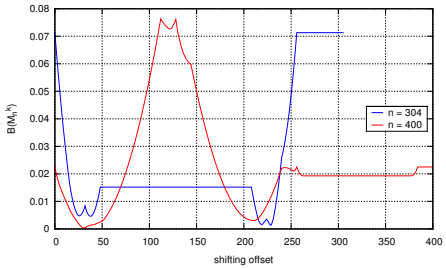

Fig. 5. Bias $\mathrm{B}(\mathbf{M}_n{}^k)$ for $k = 3$ iterations and array sizes of $n = 304$ (blue) and $n = 400$ (red) with shifting offsets ranging from 0 to n

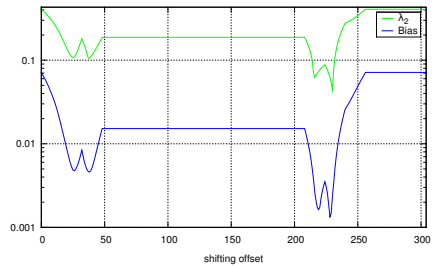

Fig. 6. Comparison of the behavior of λ_2 and bias for $n = 304$ and shifting offsets ranging from 0 to n

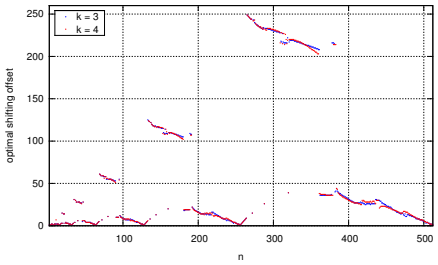

Fig. 7. Empirically determined optimal shifting offsets for $n \in \{1, \ldots, 512\}$ and $k \in \{3, 4\}$ iterations

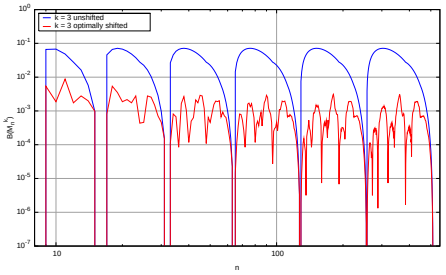

Fig. 8. Comparison of biases $\mathrm{B}(\mathbf{M}_n{}^k)$ for $k = 3$ iterations without shifting (blue) and with optimal shifting offset (red)

graphs for the two different choices of n shows large differences in the bias' overall behavior. Therefore, l needs to be carefully selected for each n to achieve optimal results.

Optimal shifting offset We empirically determined optimal shifting offsets for a range of array sizes by analyzing the permutation matrices $(\mathbf{M}_{\text{shift}} \cdot \mathbf{M}_n)^k$ for $k = 3$ and 4 (see Fig. 7). Note, that the optimal shifting offset is slightly different for different choices of k. The optimal offset can be precomputed at programming time for any combination of k and n. Using these optimal offsets, we obtained the bias graph for $k = 3$ as shown in Fig. 8, which shows a clear improvement over the basic algorithm without shifting.

Convergence of Shifting The convergence arguments detailed in Sec. 4 also apply for the algorithm's shifting extension, since $(\mathbf{M}_{\text{shift}} \cdot \mathbf{M}_n)^2$ is also positive. Furthermore, the convergence improvement by shifting is in agreement with the measure used for the convergence behavior: Fig. 6 shows a strong correlation between the λ_2 and the bias of $\mathbf{M}_n \cdot \mathbf{M}_{\text{shift,l}}$ for various shifting offsets. This shows, first, that the theoretically derived measure is valid in a practical application and second, that the bias (and thus the optimal shifting offset) can be predicted by looking at the second eigenvalues.

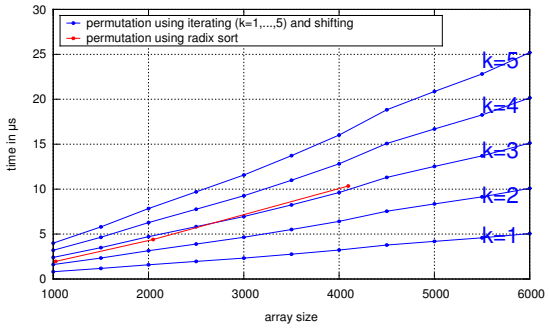

Fig. 9. Speed of shuffling arrays of various sizes with our algorithm using iterating ($k = 1, \ldots, 5$) and shifting, compared to a CUDPP-based radix sort shuffling

6 Experimental Results

To demonstrate the butterfly network's speed and suitability for many-core systems we implemented our algorithm on GPUs using NVIDIA's CUDA framework [12]. Fig. 9 shows the runtime for performing array permutations with sizes up to $n = 6000$ and $k = 1, ..., 5$ iterations with optimal shifting. Each array was permuted by a single CUDA thread block and stored in shared memory. All experiments were run on a GeForce GTX 480. The implementation is not limited to $n = 6000$ but can permute arrays of up to 12,288 4-byte or 49,152 1-byte values. 49,152 byte is the maximum shared memory that can be used by one thread block on current NVIDIA GPUs.

For comparison, we also implemented a second algorithm based on the Rand_Sort approach using radix sort for sorting. Rand_Sort is known to be unbiased but cannot be implemented in-place since random sorting keys need to be stored for each array element. We used the radix sort implementation from the CUDPP library [4], which requires array sizes smaller than 4096. This is not a fundamental limitation but merely an implementation issue. The CUDPP code is highly optimized and can therefore legitimately be regarded as suitable state-of-the-art performance reference.

Fig. 9 shows that the butterfly network with $k = 3$ and optimal shifting is about as fast as the reference approach while requiring less memory due to being in-place. If a higher bias is acceptable, using $k = 2$ or 1 (which does not perform shifting since $\mathbf{M}_n$ is applied only once) yields significant speedups of roughly 1.5 and 3, respectively.

7 Conclusions and Future Work

We showed that not all random permutation algorithms can be easily generalized for n that are not a power of two without introducing some bias. We proved that any algorithm, whose corresponding stochastic permutation matrix is positive, is nevertheless consistent, i.e., iterative application reduces the bias at an exponential rate. We also gave a specific example of a biased algorithm, the butterfly network, whose convergence behavior is further improved by shifting with carefully chosen shifting offsets. Because

the butterfly network is well-suited for implementation on a GPU (due to enabling fine-grained parallelism, needing few thread synchronizations and no contention resolving scheme), we implemented it on a current GPU and it proved to be competitive to or even faster than another highly optimized algorithm well suited for GPUs.

Further analysis needs to be done on more involved shifting strategies: In this paper we determined an optimal offset l as a function of n. There is room for further improvement, e.g., by applying varying shifting offsets after each iteration. This would require intensive pre-computations of lookup-tables for all n and k, which is beyond the scope of this work. Also, it would be interesting to analyze whether or not shifting can improve the convergence of other algorithms as well.

Acknowledgements We thank Jörg Keller for originally directing our attention towards the paper by Cong and Bader. KH is grateful for financial support by the Fonds der Chemischen Industrie and through the Graduiertenkolleg 1657 of the Deutsche Forschungsgemeinschaft. Also, we would like to thank Dominik Wodniok for implementing the radix sort based permutation approach we used as performance reference.

References

1. Anderson, R.: Parallel algorithms for generating random permutations on a shared memory machine. In: Proc. SPAA '90. pp. 95–102. ACM (1990)
2. Blelloch, G.E.: Prefix sums and their applications. Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University (Nov 1990)
3. Cong, G., Bader, D.A.: An empirical analysis of parallel random permutation algorithms on SMPs. In: Oudshoorn, M.J., Rajasekaran, S. (eds.) ISCA PDCS. pp. 27–34 (2005)
4. CUDPP – CUDA data parallel primitives library, http://code.google.com/p/cudpp/
5. Czumaj, A., Kanarek, P., Kutylowski, M., Lorys, K.: Fast generation of random permutations via networks simulation. In: Proc. ESA '96. pp. 246–260 (1996)
6. Hagerup, T.: Fast parallel generation of random permutations. In: Proc. ICALP (1991)
7. Holmes, S.: Bootstrapping Phylogenetic Trees: Theory and Methods. Statistical Science 18(2), 241–255 (2003)
8. Knuth, D.E.: The art of computer programming, volume 2 (3rd ed.) (1997)
9. Knuth, D.E.: The art of computer programming, volume 3 (2nd ed.) (1998)
10. Leighton, F.: Introduction to parallel algorithms and architectures: arrays, trees, hypercubes. No. 1, M. Kaufmann Publishers (1992)
11. Meyer, C.: Matrix Analysis and Applied Linear Algebra. SIAM (2000)
12. NVIDIA: NVIDIA CUDA C programming guide, version 3.2 (2011)
13. Perron, O.: Zur Theorie der Matrices. Mathematische Annalen 64, 248–263 (1907)
14. Soltis, P.S., Soltis, D.E.: Applying the bootstrap in phylogeny reconstruction. Statistical Science 18(2), 256–267 (2003)
15. Waksman, A.: A permutation network. J. ACM 15, 159–163 (January 1968)
16. Wu, C.F.J.: Jackknife, bootstrap and other resampling methods in regression analysis. Ann. Statist. 14(4), 1261–1295 (1986)
17. Zoubir, A.M.: Model selection: A bootstrap approach. In: Proc. ICASSP (1999)