Using Butterfly-Patterned Partial Sums to Draw from Discrete Distributions

Guy L. Steele Jr.
Oracle Labs
guy.steele@oracle.com

Jean-Baptiste Tristan
Oracle Labs
jean.baptiste.tristan@oracle.com

Abstract

We describe a SIMD technique for drawing values from multiple discrete distributions, such as sampling from the random variables of a mixture model, that avoids computing a complete table of partial sums of the relative probabilities. A table of alternate ("butterfly-patterned") form is faster to compute, making better use of coalesced memory accesses; from this table, complete partial sums are computed on the fly during a binary search. Measurements using CUDA 7.5 on an NVIDIA Titan Black GPU show that this technique makes an entire machine-learning application that uses a Latent Dirichlet Allocation topic model with 1024 topics about about 13% faster (when using single-precision floating-point data) or about 35% faster (when using double-precision floating-point data) than doing a straightforward matrix transposition after using coalesced accesses.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; E.1 [*Data Structures*]: Distributed data structures

Keywords butterfly, coalesced memory access, discrete distribution, GPU, latent Dirichlet allocation, LDA, machine learning, multithreading, memory bottleneck, parallel computing, random sampling, SIMD, transposed memory access

1. Overview

The successful use of Graphics Processing Units (GPUs) to train neural networks is a great example of how machine learning can benefit from such massively parallel architecture. Generative probabilistic modeling [3] and associated inference methods (such as Monte Carlo methods) can also benefit. Indeed, authors such as Suchard *et al.* [23] and Lee *et al.* [14] have pointed out that many algorithms of interest are embarrassingly parallel. However, the potential for massively parallel computation is only the first step toward full use of GPU capacity. One bottleneck that such embarrassingly parallel algorithms run into is related to memory bandwidth; one must design key probabilistic primitives with such constraints in mind.

We address the case where parallel threads draw independently from distinct discrete distributions. This can arise when implementing any mixture model, and Latent Dirichlet Allocation (LDA) models in particular, which are probabilistic mixture models used to discover abstract "topics" in a collection of documents (a *corpus*) [4]. This model can be fitted (or "trained") in an unsupervised fashion using sampling methods [2, chapter 11][7]. Each document is modeled as a distribution θ over topics, and each word in a document is assumed to be drawn from a distribution ϕ of words. Understanding the methods described in this paper does not require a deep understanding of sampling algorithms for LDA. What is important is that each word in a corpus is associated with a so-called "latent" random variable [2, chapter 9], usually referred to as z, that takes on one of K integer values, indicating a topic to which the word belongs. Broadly speaking, the iterative training process works by tentatively choosing a topic (that is, sampling the random latent variable z) for a given word using relative probabilities calculated from θ and ϕ, then updating θ and ϕ accordingly.

In this paper, we focus on the step that, given a discrete distribution represented as a length-K array a of relative probabilities, chooses an integer j such that $0 \le j < K$ in such a way that the probability of choosing any specific value j' is $a_{j'}/\sigma$, where σ is the sum of all elements of a. (We use zero-based indexing throughout this paper.) This is easily done using a three-step process:

1. Normalize a (divide each entry by the sum of all entries).
2. Let u be chosen uniformly at random (or pseudorandomly) from the real interval $[0, 1)$.
3. Find the smallest index j such that the sum of all entries of a at or below index j is larger than u.

In practice, this sequence of steps may be optimized by doing a bit of algebra and using a binary search:

PPoPP '17, February 04-08, 2017, Austin, TX, USA.
 ISBN 978-1-4503-4493-7/17/02…$15.00.
http://dx.doi.org/10.1145/3018743.3018757

1. Compute p, the prefix-sum of a, such that $p_j = \sum_{i=0}^{j} a_i$.
2. Choose u uniformly at random (or pseudorandomly) from the real interval $[0, 1)$, and let $u' = p_{(K-1)} \times u$.
3. Use a binary search to find the smallest index j such that the entry at index p_j is larger than u'.

This works because all elements of a are nonnegative and therefore elements of p are monotonically nondecreasing.

Now suppose that we have many discrete distributions (thousands or millions) and wish to draw one sample from each, using a SIMD-style GPU. An obvious approach is to assign each distribution to a separate thread and have each thread execute the optimized three-step algorithm. However, in the context of the LDA application, a problem arises: when the threads fetch entries from their respective arrays (especially the ϕ arrays), the values to be fetched will likely reside at unrelated locations in memory, resulting in poor memory-fetch performance. A standard technique is to have all the lanes in a warp (a group of threads being executed simultaneously by the SIMD engine) cooperate with each other. For concreteness, suppose there are 32 threads in a warp, and for simplicity, assume that each array a of relative probabilities is also of length 32. We can furthermore assume that the elements of any single a array are stored sequentially in memory (and therefore fit within a small number of cache lines); the problem arises solely because we cannot assume any specific relationship within memory among the 32 a instances to be processed simultaneously. The technique that gets around this problem is *transposed memory access*: as the 32 elements are fetched for each of 32 instances of a, on each step j of 32 steps ($0 \le j < 32$), lane i ($0 \le i < 32$) fetches not $a[i][j]$ but instead $a[j][i]$. (Compare, for example, the storage of floating-point numbers as "slicewise" rather than "fieldwise" in the architecture of the Connection Machine Model CM-2, so that 32 1-bit processors cooperate on each of 32 clock cycles to fetch and store an entire 32-bit floating-point value that logically belongs to just one of the 32 processors [10].) In words, on step j all 32 lanes fetch the 32 values needed by lane j; as a result, on each memory cycle all 32 values being fetched are in a contiguous region of memory, allowing improved memory-fetch performance.

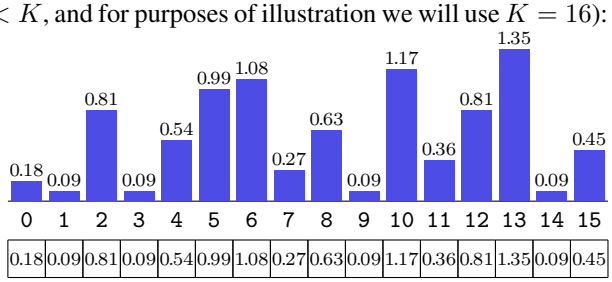
It is then necessary for the lanes to exchange information among themselves so that the rest of the algorithm may be carried out, including the summation arithmetic.

The novel contribution of this paper is to observe and then exploit the fact that the binary search of the array p (which is computed from a) does not access all entries of p; in fact, for an array of size K it examines only about $\log_2 K$ entries. Therefore it is not necessary to compute all entries of the prefix-sum table. We present an alternate technique that computes a "butterfly-patterned" partial-sums table, using less computational and communication effort when implemented on a GPU; a modified binary search then uses this table to compute, on the fly, entries that would have been in the original complete prefix-sum table. This requires more work per table entry during the binary search, but because the search examines only a few table entries, the result is a net reduction in execution time. This technique may be effective for collapsed LDA Gibbs samplers [16, 24, 30] as well as uncollapsed samplers, and may also be useful for GPU implementations of other algorithms [32] whose inner loops sample from discrete distributions.

This paper is not about machine-learning algorithms in general or LDA in particular; rather, we use an LDA application as a convenient and practical benchmark for evaluating data-parallel sampling algorithms. The specific LDA algorithm that we use is state-of-the-art [25] and had already been carefully tuned for speed before application of the techniques described in this paper.

2. Background

Suppose we are given a discrete distribution described as an array a of length K such that a_j is the relative probability that sampling the distribution will produce the value j ($0 \le j < K$, and for purposes of illustration we will use $K = 16$):



a	0.18	0.09	0.81	0.09	0.54	0.99	1.08	0.27	0.63	0.09	1.17	0.36	0.81	1.35	0.09	0.45

From a, compute the prefix-sum array p:

1: **let** $sum = 0.0$
2: **for** k from 0 through $K - 1$ **do**
3: $\quad sum \mathrel{+}= a[k];\ p[k] := sum$

p	0.18	0.27	1.08	1.17	1.71	2.70	3.78	4.05	4.68	4.77	5.94	6.30	7.11	8.46	8.55	9.00

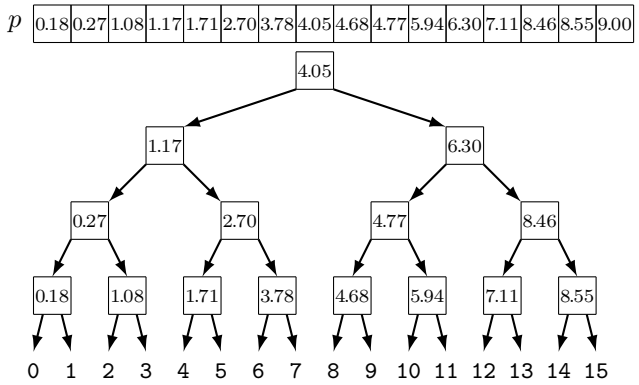
To sample the distribution, a simple linear search will do:

1: **let** $u =$ random value chosen from $[0.0, 1.0)$
2: **let** $u' = sum \times u$, $j = 0$
3: **while** $j < K - 1$ and $u' \ge p[j]$ **do** $j \mathrel{+}= 1$

Alternatively, one may use a binary search:

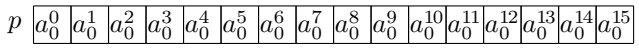
1: **let** $u =$ random value chosen from $[0.0, 1.0)$
2: **let** $u' = sum \times u$, $j = 0$, $k = K - 1$
3: **while** $j < k$ **do**
4: $\quad$ **let** $mid = \left\lfloor \dfrac{j + k}{2} \right\rfloor$
5: $\quad$ **if** $u' < p[mid]$ **then** $k := mid$ **else** $j := mid + 1$

A binary search on an array amounts to walking down a binary tree whose leaves are the array indices and whose internal nodes are labeled with all entries except the last. We can draw such a tree by starting with a drawing of the array and then displacing each entry (except the last) vertically:

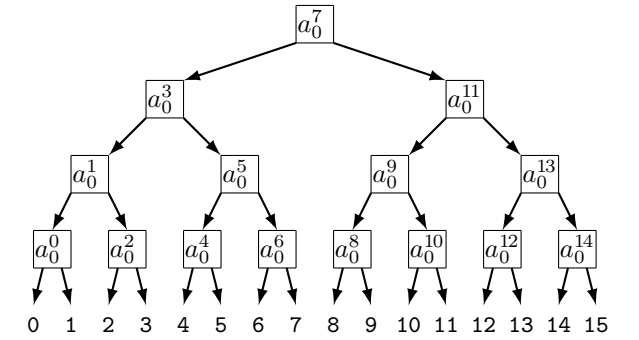


Starting from the root, we compare the quantity we are searching for to the label of each internal node that we encounter; if it is smaller, we descend to the left child, otherwise to the right child.

We can picture the general process by using a convenient abbreviation: a_m^n means $\sum_{m \le i \le n} a_i$. Then array p looks like:

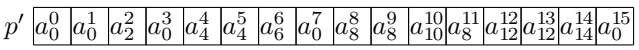


and the corresponding binary tree looks like this:

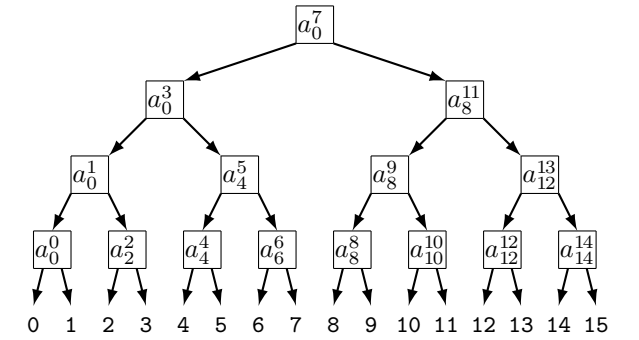


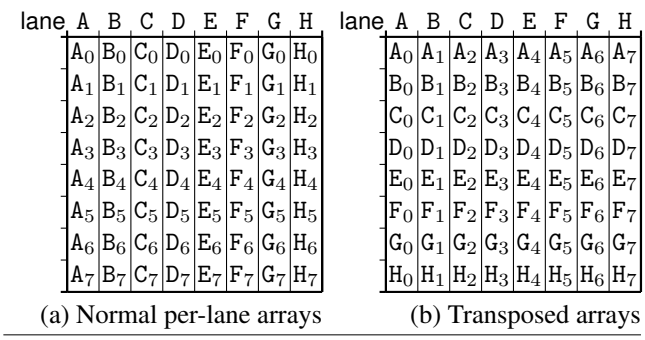
Instead of computing the prefix-sum of a, we can compute a different set of partial sums, which we will call p':

1: **for** k from 0 through $K-1$ **do** $p'[k] := a[k]$
2: **for** b from 1 through $\log_2 K$ **do**
3: **for** i from 1 through $2^{(\log_2 K)-b}$ **do**
4: $p'[(2i)2^{b-1}-1] \mathrel{+}= p'[(2i-1)2^{b-1}-1]$



and also treat it as a binary tree to be searched:





(a) Normal per-lane arrays (b) Transposed arrays

Figure 1. Array storage in lane registers ($W = 8$). Each column represents W consecutive registers in the register file of one lane. Arrays elements are numbered, lanes are labeled by letters; so "$\mathtt{C}_4$" denotes element 4 of the array that logically belongs to (is intended to be processed by) lane C.

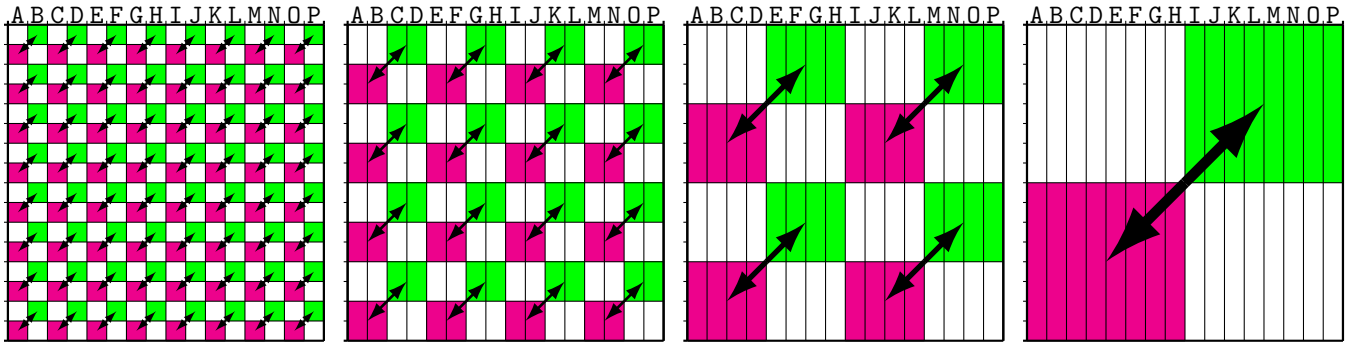
Here's the trick: while the internal nodes of this tree do not contain the complete partial sums of p needed for comparison, the values of p that we actually need can be computed from p' on the fly as we walk down the tree.

1: **let** $u =$ random value chosen from $[0.0, 1.0)$
2: **let** $u' = sum \times u$, $j = 0$, $k = K - 1$
3: **let** $lowValue = 0$
4: **while** $j < k$ **do**
5: **let** $mid = \left\lfloor \dfrac{j+k}{2} \right\rfloor$
6: **let** $compareValue = lowValue + p'[m]$
7: **if** $u' < p[mid]$ **then**
8: $k := mid$
9: **else**
10: $j := mid + 1$
11: $lowValue := compareValue$

This tree p' is familiar: it is an intermediate state in one parallel algorithm for computing the prefix-sum array p. Because the iterations of the inner **for** loop used to construct p' are independent, they may be executed in parallel, and so p' can be constructed from a in $\log_2 K$ steps, building the tree from bottom to top; and p can likewise be constructed from p' in $\log_2 K$ steps, passing information down the tree.

3. Parallel SIMD Implementation on a GPU

In the CUDA programming model, as used on GPU products from NVIDIA, one may pretend that one has thousands or millions of threads, each with its own local memory, stack, and registers, and one may assign to each such thread an instance of a computation. The operating system multiplexes the hardware resources to give the illusion of more or less simultaneous execution. For management purposes, threads are grouped into *warps*, each having exactly W threads. In the actual hardware used for our experiments, $W = 32$; however, for purposes of illustration, we will use $W = 8$ or $W = 16$, to make the figures a manageable size while keeping fonts at a readable size. Warps are automatically sched-



(a) Transposing two elements in each 2×2 block (b) Transposing two quadrants in each 4×4 block (c) Transposing two quadrants in each 8×8 block (d) Transposing two quadrants in the overall 16×16 block

Figure 2. Transposing a matrix consisting of W registers in each of the W lanes of the GPU (shown for $W = 16$)

uled onto multiple SIMD processing engines, where each engine has W *lanes*, and each lane performs the computation for one thread. When an engine has completed computation for one warp, it goes on to process another warp, and so on, until all warps have been processed.

The programmer can count on absolutely simultaneous execution of the W threads that constitute any single warp. There are a few low-level operations that can exchange data synchronously among the lanes of a warp [20, 29]; two are of interest here. The expression `__shfl`$(value, lane)$, when executed (necessarily simultaneously) as a SIMD operation by the threads in a warp, causes each lane to make its computed $value$ available to all lanes, and then returns the $value$ provided by lane $lane$; in short, each lane gets to decide which of the other lanes to read from. The expression `__shfl_xor`$(value, m)$ is, in effect, an abbreviation for `__shfl`$(value, myLaneId \oplus m)$, where $myLaneId$ has the value i in lane i ($0 \leq i < W$), and where $\oplus$ is the bitwise exclusive-OR operation on unsigned integers. If the same value of m is provided on all lanes, then `__shfl_xor` performs a permutation specified by m; for example, if $m = 2$, then lanes whose numbers differ in exactly one bit position (the second-least-significant bit) will exchange data.

Certain instructions can cause individual lanes of a warp to become conditionally inactive; others can force some or all lanes to become active again. These are typically used by a compiler (such as the CUDA compiler) to implement **if-then-else** statements and loops.

When the lanes of a warp execute a "load" instruction, then W words are read from memory; more precisely, at most W words are read from memory, because the memory controller performs memory reads only for active lanes (but this nicety will not matter for our purposes). The memory controller makes an effort to *coalesce* these read requests, so that if multiple words to be read happen to reside in the same cache line, then the cache line is read only once.

In our LDA application, we have many documents to be processed. each with a different number of words (we presort the documents so that those of any one warp likely have similar lengths). We assign one document to each thread. The thread processes each word in the document; for that word it computes a discrete distribution to be sampled, and then samples it once. All threads in a warp process the kth word of their respective documents simultaneously.

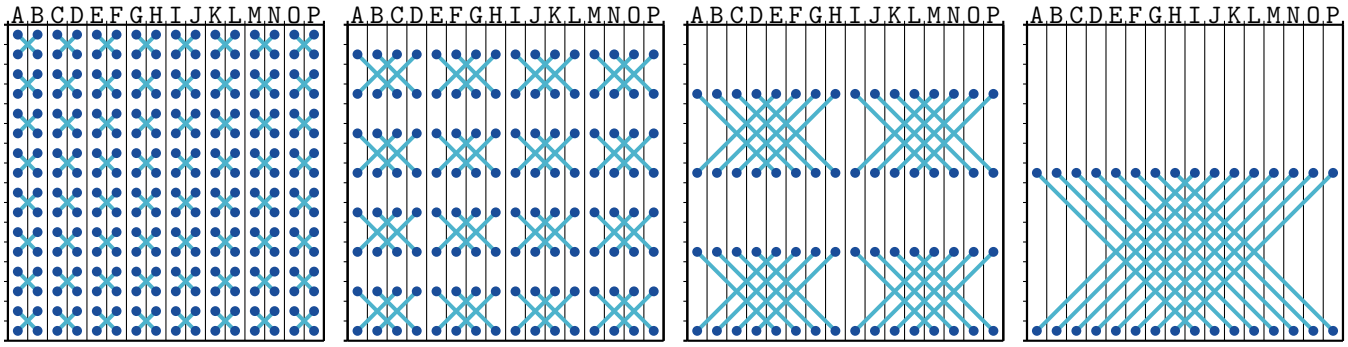
Within each thread, the array a representing the distribution to be sampled is computed on the fly, used once, and then discarded, so a resides in registers. It is the elementwise product of two other arrays, θ and ϕ; θ represents a discrete distribution associated with the document, so it is kept in the local memory of the thread, but ϕ represents a discrete distribution associated with the word type, and comes from a very large table that must reside in main memory. There is no reason to expect significant correlation of word types among the kth words of the W documents in a warp, so very likely their ϕ arrays are scattered throughout main memory.

To keep the discussion simple, for now we will assume $K = W$. (We lift this restriction in Section 5.)

If we use a straightforward approach, every lane loads only the data it needs and computes its own a values. This situation is pictured in Figure 1(a), where for each element the variable name a is replaced with one of the letters `A` through `H` to indicate which lane that particular element logically belongs to. In this case, every a element resides in some register of the lane to which it logically belongs, which logically is just what we want. The downside is poor performance because few memory accesses will be coalesced.

An alternative approach is to use transposed memory access. The lanes work together so that memory accesses are coalesced, then compute the a values for the data they happen to have (the precomputed θ arrays can also be pretransposed, so that every lane will have the elements of θ that it needs in its own local memory). This situation is pictured in Figure 1(b). The downside is that no lane has direct access to all the a values that it needs to compute its p values.

One way out is to arrange to have lanes use `__shfl_xor` operations to exchange data, so as to turn the situation of Figure 1(b) into that of Figure 1(a). This is a well-known problem with well-known solutions; one that works as well as could be expected is illustrated in Figure 2. On step i ($0 \leq i < \log_2 W$), $W/2$ `__shfl_xor` instructions are used



(a) Process distance-1 items (8 `__shfl_xor` operations)
(b) Process distance-2 items (4 `__shfl_xor` operations)
(c) Process distance-4 items (2 `__shfl_xor` operations)
(d) Process distance-8 items (1 `__shfl_xor` operation)

Figure 3. Patterns of application of butterfly computations to compute butterfly-patterned p' arrays ($W = 16$)

(one for every pair of register positions) to exchange registers whose numbers differ by 2^i between lanes whose numbers differ by 2^i (see Figure 2). First, $W/2$ `__shfl_xor` instructions are used (one for every pair of register positions) to exchange registers whose numbers differ by 1 between lanes whose numbers differ by 1 (Figure 2(a)). Second, $W/2$ `__shfl_xor` instructions are used to exchange registers whose numbers differ by 2 between lanes whose numbers differ by 2 (Figure 2(b)). Third, $W/2$ `__shfl_xor` instructions are used to exchange registers whose numbers differ by 4 between lanes whose numbers differ by 4 (Figure 2(c)). Finally, $W/2$ `__shfl_xor` instructions are used to exchange registers whose numbers differ by 8 between lanes whose numbers differ by 8 (Figure 2(d)). Thus the total number of `__shfl_xor` operations is $\frac{1}{2}W \log_2 W$.

We offer a novel approach that uses $3(W-1)$ shuffle operations[1] and avoids all scattered memory access. The idea is to have the lanes cooperate to construct a partial sums table that is related to the p' arrays discussed in Section 2. Instead of ending up with every lane having all its own p' array elements, each array is distributed across multiple lanes—but instead of every lane containing exactly one element of every p' array, they are distributed in a more complicated way: we call it a "butterfly-patterned partial sums table." The construction of this table requires only $W-1$ `__shfl_xor` operations. During the binary search, an additional $2(W-1)$ `__shfl` and `__shfl_xor` operations are used as the lanes assist each other in accessing array elements.

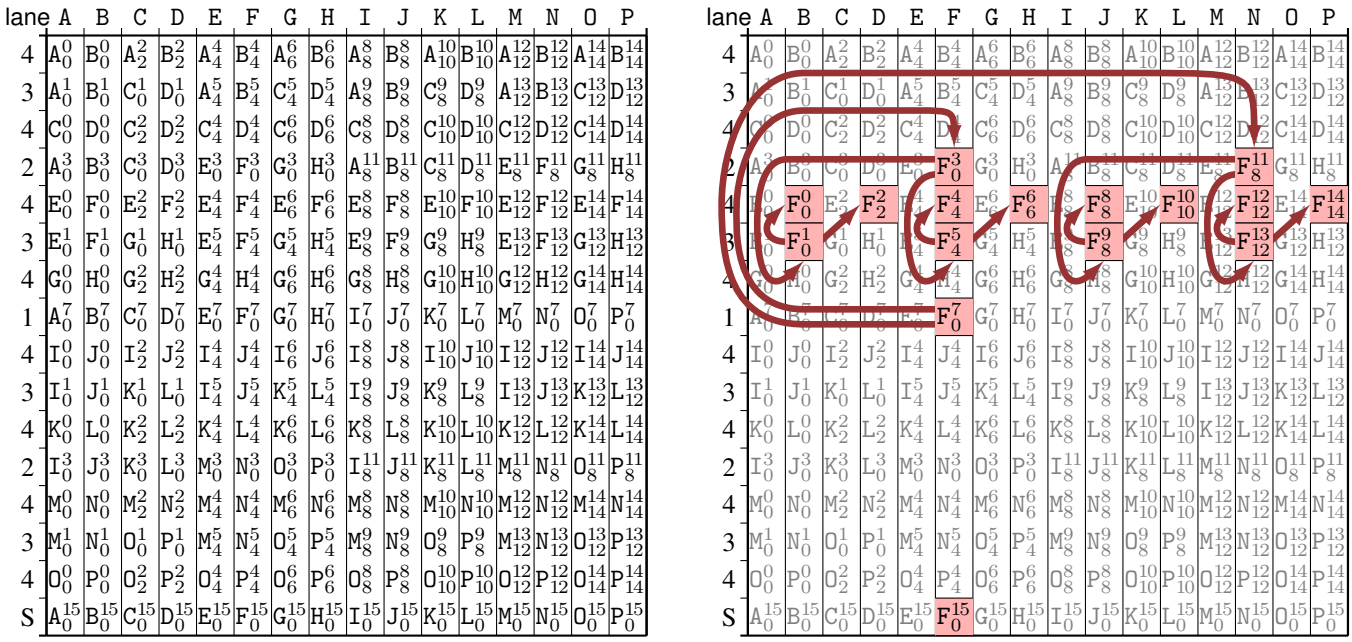
Here is how the butterfly-patterned table is constructed. There are $\log_2 W$ steps, and during step i ($0 \le i < \log_2 W$), $2^{(\log_2 W)-i-1}$ `__shfl_xor` instructions are used to perform $\left(2^{(\log_2 W)-i-1}\right)\frac{W}{2}$ butterfly computations (see Figure 3). Each butterfly computation operates on four entries, within the W p' arrays, that are at the intersection of two rows whose indices differ by a power of 2 and two columns whose indices differ by that same power of 2. Suppose the four values in those entries are $\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]$; they are replaced by $\left[\begin{array}{c|c} a & c \\ \hline a+b & c+d \end{array}\right]$. Each such replacement operation on four entries is symbolized by  in the figures. Here is CUDA code for constructing the butterfly-patterned table:

```
int r = threadIdx.x & 0x1f;   /* lane ID */
for (int b=0; b < W; b+=b) {  /* 1,2,4,8,...*/
  for (int j=0; j < (W>>(b+1)); j++) {
    d = (((j << 1) + 1) << b) - 1;
    h = (r & (1<<b)) ? a[d] : a[d+(1<<b)];
    v = __shfl_xor(h, 1<<b);
    if (r & (1<<b)) a[d] = v;   | These 3 lines are
    else a[d+(1<<b)] = v;       | replaced below to
    a[d+(1<<b)] += a[d];        | make a new version.
    p[d] = a[d]; } }
p[W-1] = a[W-1];
```

The result is shown in Figure 4(a). The rows of this figure are labeled with tree levels (1 through 4) and S. Observe that in the row labeled S, every lane has the sum of its own a array. Observe also in the row labeled 1, every lane has the root of its own binary search tree for p'. The two rows labeled 2 collectively contain all the level-2 internal nodes of the trees, the four rows labeled 3 contain all the level-3 nodes, and the eight rows labeled 4 contain all the level-4 nodes. Figure 4(b) is the same, but highlights entries that logically belong to lane `F`; one can see the sum (in the bottom row) as well as the entire binary search tree, indicated by the arrows.

Here is a precise description of the pattern: the entry in row i and column j contains the value $\mathtt{X}_v^w$ where $m = i \oplus (i+1)$, $k = \lfloor \frac{m}{2} \rfloor$, $\ell = (i \,\&\, \neg m) + (j \,\&\, m)$, $\mathtt{X} =$ "`ABCDEFGHIJKLMNOP`"$[\ell]$, $v = j \,\&\, (\neg k)$, and $w = v + k$. We use "$\neg$" to indicate bitwise NOT, "&" to indicate bitwise AND, and (as earlier) "$\oplus$" to indicate bitwise XOR.

[1] We realize that $3(W-1) > \frac{1}{2}W \log_2 W$ for $W = 16$ or even for $W = 32$. We remark on the algorithmic complexity as a function of the number of shuffle perations as a matter of mathematical principle, but recognize that this comparison does not explain the faster speed observed in practice with the butterfly-patterned partial sums for $W = 32$. Rather, the observed speedup comes from an overall reduction of instructions and the manner in which the CUDA compiler manages to schedule them. In thr future, if GPU chips are ever built with $W \ge 64$, the improvement in algorithmic complexity may also begin to matter.



(a) Partial sums resulting from butterfly-pattern summation

(b) The tree and total sum for lane F

Figure 4. The butterfly-patterned table that results from the butterfly computations of Figure 3 ($W = 16$)

Given this table, it's just a matter of making sure that each lane has access to the tree nodes it needs during the binary search. During step i of the binary search ($0 \le i < \log_2 W$), $2^{(\log_2 W)-i-1}$ `__shfl_xor` instructions are executed so as to make available all entries in rows labeled $(\log_2 W) - i$, and each lane picks off the entry of interest by computing which lane contains it and which `__shfl_xor` instruction will post it; $2^{(\log_2 W)-i-1}$ `__shfl` instructions are also needed to exchange array-address information.

There a variation of this technique that in our experiments turned out to be even faster. It uses a tiny modification of the butterfly computation: instead of replacing $\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]$ with $\left[\begin{array}{c|c} a & c \\ \hline a+b & c+d \end{array}\right]$, it replaces them with $\left[\begin{array}{c|c} a & d \\ \hline a+b & c+d \end{array}\right]$. This alternate replacement saves one machine instruction at the lowest level in the innermost loop. Merely replace the three lines of CUDA code indicated with these two lines:

```
if (r & (1<<b)) a[d] = a[d+(1<<b)];
a[d+(1<<b)] = a[d] + v;
```

Because the two loops are unrolled (the outer loop we actually unroll manually, and the CUDA compiler then automatically unrolls the inner one), many of the expressions are reduced to constants or hoisted out of the loops at compile time. As a result, the `if` statement represents a single conditional move operation, whereas in the previous version of the code, the `if-else` statement represents two conditional move operations. Eliminating one conditional move operation results in a substantial speed improvement (about 15%).

The resulting table has a slightly more complicated pattern (see Figure 5). It requires a slightly more complicated version of the binary search code, one that either adds or subtracts at each step, and whether to add or subtract at a given step depends on a bit of the lane number (that is, at each step some lanes must add while others subtract).

4. Use in an LDA Application

In Sections 4, 5, and 6 we show how to use these sampling techniques in the context of a kernel for sampling z values for a complete machine-learning application.

For a Latent Dirichlet Allocation model of, for example, a set of documents to which we want to assign topics probabilistically using Gibbs sampling, let M be the number of documents, K be the number of topics, and V be the size of the vocabulary, which is a set of distinct words. Each document is a bag of words, each of which belongs to the vocabulary; any given word can appear in any number of documents, and may appear any number of times in any single document. The documents may be of different lengths.

We are interested in the phase of an uncollapsed Gibbs sampler that draws new z values, given θ and ϕ distributions. Because no z value directly depends on any other z value in this formulation, new z values may all be computed independently (and therefore in parallel to any extent desired).

We assume that we are given an $M \times K$ matrix θ and a $V \times K$ matrix ϕ; the elements of these matrices are nonnegative numbers, typically represented as floating-point values. Row m of θ (that is, $\theta[m, \cdot]$) is the (currently assumed) distribution of topics for document m, that is, the relative probabilities (weights) for each of the K possible topics to which the document might be assigned. Note that columns of θ are *not* to be considered as distributions. Sim-

lane	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
4	A_0^0	B_1^1	A_2^2	B_3^3	A_4^4	B_5^5	A_6^6	B_7^7	A_8^8	B_9^9	A_{10}^{10}	B_{11}^{11}	A_{12}^{12}	B_{13}^{13}	A_{14}^{14}	B_{15}^{15}
3	A_0^1	B_0^1	C_2^3	D_2^3	A_4^5	B_4^5	C_6^7	D_6^7	A_8^9	B_8^9	C_{10}^{11}	D_{10}^{11}	A_{12}^{13}	B_{12}^{13}	C_{14}^{15}	D_{14}^{15}
4	C_0^0	D_1^1	C_2^2	D_3^3	C_4^4	D_5^5	C_6^6	D_7^7	C_8^8	D_9^9	C_{10}^{10}	D_{11}^{11}	C_{12}^{12}	D_{13}^{13}	C_{14}^{14}	D_{15}^{15}
2	A_0^3	B_0^3	C_0^3	D_0^3	E_4^7	F_4^7	G_4^7	H_4^7	A_8^{11}	B_8^{11}	C_8^{11}	D_8^{11}	E_{12}^{15}	F_{12}^{15}	G_{12}^{15}	H_{12}^{15}
4	E_0^0	F_1^1	E_2^2	F_3^3	E_4^4	F_5^5	E_6^6	F_7^7	E_8^8	F_9^9	E_{10}^{10}	F_{11}^{11}	E_{12}^{12}	F_{13}^{13}	E_{14}^{14}	F_{15}^{15}
3	E_0^1	F_0^1	G_2^3	H_2^3	E_4^5	F_4^5	G_6^7	H_6^7	E_8^9	F_8^9	G_{10}^{11}	H_{10}^{11}	E_{12}^{13}	F_{12}^{13}	G_{14}^{15}	H_{14}^{15}
4	G_0^0	H_1^1	G_2^2	H_3^3	G_4^4	H_5^5	G_6^6	H_7^7	G_8^8	H_9^9	G_{10}^{10}	H_{11}^{11}	G_{12}^{12}	H_{13}^{13}	G_{14}^{14}	H_{15}^{15}
1	A_0^7	B_0^7	C_0^7	D_0^7	E_0^7	F_0^7	G_0^7	H_0^7	I_8^{15}	J_8^{15}	K_8^{15}	L_8^{15}	M_8^{15}	N_8^{15}	O_8^{15}	P_8^{15}
4	I_0^0	J_1^1	I_2^2	J_3^3	I_4^4	J_5^5	I_6^6	J_7^7	I_8^8	J_9^9	I_{10}^{10}	J_{11}^{11}	I_{12}^{12}	J_{13}^{13}	I_{14}^{14}	J_{15}^{15}
3	I_0^1	J_0^1	K_2^3	L_2^3	I_4^5	J_4^5	K_6^7	L_6^7	I_8^9	J_8^9	K_{10}^{11}	L_{10}^{11}	I_{12}^{13}	J_{12}^{13}	K_{14}^{15}	L_{14}^{15}
4	K_0^0	L_1^1	K_2^2	L_3^3	K_4^4	L_5^5	K_6^6	L_7^7	K_8^8	L_9^9	K_{10}^{10}	L_{11}^{11}	K_{12}^{12}	L_{13}^{13}	K_{14}^{14}	L_{15}^{15}
2	I_0^3	J_0^3	K_0^3	L_0^3	M_4^7	N_4^7	O_4^7	P_4^7	I_8^{11}	J_8^{11}	K_8^{11}	L_8^{11}	M_{12}^{15}	N_{12}^{15}	O_{12}^{15}	P_{12}^{15}
4	M_0^0	N_1^1	M_2^2	N_3^3	M_4^4	N_5^5	M_6^6	N_7^7	M_8^8	N_9^9	M_{10}^{10}	N_{11}^{11}	M_{12}^{12}	N_{13}^{13}	M_{14}^{14}	N_{15}^{15}
3	M_0^1	N_0^1	O_2^3	P_2^3	M_4^5	N_4^5	O_6^7	P_6^7	M_8^9	N_8^9	O_{10}^{11}	P_{10}^{11}	M_{12}^{13}	N_{12}^{13}	O_{14}^{15}	P_{14}^{15}
4	O_0^0	P_1^1	O_2^2	P_3^3	O_4^4	P_5^5	O_6^6	P_7^7	O_8^8	P_9^9	O_{10}^{10}	P_{11}^{11}	O_{12}^{12}	P_{13}^{13}	O_{14}^{14}	P_{15}^{15}
S	A_0^{15}	B_0^{15}	C_0^{15}	D_0^{15}	E_0^{15}	F_0^{15}	G_0^{15}	H_0^{15}	I_0^{15}	J_0^{15}	K_0^{15}	L_0^{15}	M_0^{15}	N_0^{15}	O_0^{15}	P_0^{15}

Figure 5. Alternate "add-subtract" butterfly pattern for p'

ilarly, column k of ϕ (that is, $\phi[\cdot, k]$) is the (currently assumed) distribution of words for topic k, that is, the weights with which the V possible words in the vocabulary are associated with the topic. Note that rows of ϕ are *not* to be considered as distributions. We organize θ as rows and ϕ as columns for engineering reasons: we want the K entries obtained by ranging over all possible topics to be contiguous in memory so as to take advantage of memory cache structure.

We also assume that we are given (i) a length-M vector of nonnegative integers N such that $N[m]$ is the number of words in document m, and (ii) an $M \times N$ ragged array w, by which we mean that for $0 \le m < M$, $w[m]$ is a vector of length $N[m]$. Each element of w is a word type (a nonnegative integer less than V) and may therefore be used as a first index for ϕ. Our goal, given K, M, V, N, ϕ, θ, and w and assuming the use of a temporary $M \times N \times K$ ragged work array a (which we will later optimize away), is to compute all the elements for an $M \times N$ ragged array z as follows: For all m such that $0 \le m < M$ and for all i such that $0 \le i < N[m]$, do two things: first, for all k such that $0 \le k < K$, let $a[m][i][k] = \theta[m, k] \times \phi[w[m, i], k]$; second, let $z[m, i]$ be a nonnegative integer less than K, chosen randomly in such a way that the probability of choosing the value k' is $a[m][i][k']/\sigma$ where $\sigma = \sum_{0 \le k < K} a[m][i][k]$. Thus, $a[m][i][k']$ is a relative (unnormalized) probability, and $a[m][i][k']/\sigma$ is an absolute (normalized) probability.

Algorithm 1 is a basic implementation of this process. We remark that a "**let**" statement creates a local binding of a scalar (single-valued) variable and gives it a value, that a "**local array**" declaration creates a local binding of an array variable (containing an element value for each indexable position in the array), and that distinct iterations of a containing "**for**" or "**for all**" construct are understood to create distinct and independent instantiations of such local variables for each iteration. The iterations of "**for** … from … through …" are executed sequentially in a specific order; but the iterations of a "**for all**" construct are intended to be computationally independent and therefore may be executed in any order, or in parallel, or in any sequential-parallel combination. We use angle brackets to indicate the use of a "code chunk" that is defined as a separate algorithm; such a use indicates that the definition of the code chunk should be inserted at the use site, possibly with parameter substitution, as if it were a C macro, but surrounded by **begin** and **end** (this programming-language technicality ensures that the scope of any variable declared within the code chunk is confined to that code chunk).

The computation of the θ-ϕ products (lines 7–8 of Algorithm 1) is straightforward. The computation of partial sums (lines 10–12) is sequential; the variable sum accumulates the

Algorithm 1 Drawing new z values

```
1:  procedure DrawZ(N[M], θ[M,K], φ[V,K],
2:                  w[M][N]; output z[M,N])
3:    local array a[M][N][K], p[M][N][K]
4:    for all 0 ≤ m < M do
5:      for all 0 ≤ i < N[m] do
6:        ▷ Compute θ-φ products
7:        for all 0 ≤ k < K do
8:          a[m][i][k] := θ[m,k] × φ[w[m][i],k]
9:        ▷ Compute partials sums of the products
10:       let sum = 0.0
11:       for k from 0 through K − 1 do
12:         sum += a[m][i][k]; p[m][i][k] := sum
13:       let j = 0
14:       ⟨search the table p[m][i] of partial sums⟩
15:       z[m,i] := j
16:   end
```

Algorithm 2 Simple linear search

```
1: code chunk ⟨search the table P of partial sums⟩:
2:   let u = random value chosen from [0.0, 1.0)
3:   let u' = sum × u
4:   while j < K − 1 and u' ≥ P[j] do j += 1
5: end
```

Algorithm 3 Simple binary search

```
1: code chunk ⟨search the table P of partial sums⟩:
2:   let u = random value chosen from [0.0, 1.0)
3:   let u' = sum × u, k = K − 1
4:   while j < k do
5:     let mid = ⌊(j + k)/2⌋
6:     if u' < P[mid] then k := mid
7:     else j := mid + 1
8: end
```

Algorithm 4 Drawing z values (transposed access)

```
1:  procedure DrawZ(N[M], θ[M,K], φ[V,K],
2:                  w[M][N]; output z[M,N])
3:    local array p_local[M][K], a_local[M][W]
4:    for all 0 ≤ q < M/W do
5:      local array c_warp[W,W]
6:      for SIMD 0 ≤ r < W do
7:        let m = q × W + r
8:        local array θ_local[K]
9:        ⟨cache θ values into θ_local⟩
10:       let i_master = 0
11:       while any(i_master < N[m]) do
12:         let i = min(i_master, N[m] − 1)
13:         ⟨compute partial sums of θ-φ products⟩
14:         let j = 0
15:         ⟨search the table p_local[m] of partial sums⟩
16:         z[m,i] := j
17:         i_master += 1
18: end
```

products, and successive values of sum are stored into the array p. A random integer is chosen for $z[m,i]$ by choosing a random value uniformly from the range $[0,0,1.0)$, scaling it by the final value of sum (which has the same algorithmic effect as dividing each $p[m][i][k]$ by that value, for all $0 \le k < K$, to turn it into an absolute probability), and then searching the subarray $p[m][i]$ to find the smallest entry that is larger than the scaled value (and if there are several such entries, all equal, then the one with the smallest index is chosen); the index j of that entry is used as the desired randomly chosen integer. A simple linear search (Algorithm 2) can do the job, but a binary search (Algorithm 3) can be used instead, which is faster, on average, for K sufficiently large [13, exercise 6.2.1-5].

5. Blocking and Transposition

Anticipating certain characteristics of the hardware, we make some commitments as to how the algorithm will be executed. We assume that arrays are laid out in row-major order (as they are when using C or CUDA). Let W be a machine-dependent constant (typically 16 or 32, but for now we do not require that W be a power of 2). For purposes of illustration we assume $W = 8$ and also $K = 19$. We divide the documents into groups of size W and assume that M is an exact multiple of W. (In the overall application, the set of documents can be padded with empty documents so as to make M be an exact multiple of W without affecting the overall behavior of the algorithm on the "real" documents.) We turn the outermost loop of Algorithm 1 (with index variable m) into two nested loops with index variables q and r, from which the equivalent value for m is then computed. We commit to making the loop with index variable i sequential, to treating the iterations of the loop on q as independent (and therefore possibly parallel), and to treating the iterations of the loop on r as executed by a SIMD "thread warp" of size W, that is, parallel and implicitly lock-step synchronized. As a result, we view each of the M documents as being processed by a separate thread. A benefit of making the loop on i sequential is that the array p can be made two-dimensional and non-ragged, having size $M \times K$. We fuse the loop that computes θ-ϕ products with the loop that computes partial sums; this eliminates the need for the array a, but instead (for reasons explained below) we use a_{local} as a two-dimensional, non-ragged array of size $M \times W$ that is used only when $K \ge W$. Within the loop on q we declare a local work array c_{warp} of size $W \times W$ that will be used to exchange information by the W threads within a warp; our eventual intent is that this array will reside in GPU registers. We cache values from the array θ in a per-thread array θ_{local} of length K, anticipating that such cached values will reside in a faster memory and be used repeatedly by the loop on i.

Algorithm 5 Caching θ values (transposed access)

```
1:  code chunk ⟨cache θ values into θ_local⟩:
2:    let j = 0
3:    while j < (K mod W) do          ▷ Cache the remnant
4:      θ_local[j] := θ[m,j]; j += 1
5:    while j < K do                  ▷ Cache all W × W blocks
6:      for k from 0 through W − 1 do
7:        ▷ Next line uses transposed access to θ
8:        θ_local[j + k] := θ[q × W + k, j + r]
9:      j += W
10: end
```

There is, however, a subtle problem with the loop controlling index variable i: the upper bound $N[m]$ for the loop variable may be different for different threads. As a result, in the last iterations it may be that some threads have "gone to sleep" because they reached their upper loop bound earlier than other threads in the warp. This is undesirable because, as we shall see, we rely on all threads "staying awake" so that they can assist each other. Therefore, we rewrite the loop control to use a "master index" idiom and exploit the trick of allowing a thread to perform its last iteration (with $i = N[m] - 1$) multiple times, which doesn't work for many algorithms but is acceptable for LDA Gibbs.

The result of all these code transformations is Algorithm 4, which makes use of three code chunks: Algorithm 5, Algorithm 6, and either Algorithm 2 or Algorithm 3. Algorithms 5 and 6, besides using SIMD thread warps of size W to process documents in groups of size W, also process topics in blocks of size W. This allows the innermost loops to process "little" arrays of size $W \times W$. If K (the number of topics) is not a multiple of W, then there will be a *remnant* of size $K \bmod W$. To make looping code slightly simpler, we put the remnant at the *front* of each array, rather than at the end. For $W = 8$ and $K = 19$, topics 0, 1, and 2 form the remnant; topics 3–10 form a block of length 8; and topics 11–18 form a second block. This organization of arrays into

Algorithm 6 Compute partial sums (transposed access)

```
1:  code chunk ⟨compute partial sums of θ-φ products⟩:
2:    let c = w[m][i]
3:    for all 0 ≤ k < W do
4:      c_warp[k, r] := c                ▷ Transposed access to c_warp
5:    let sum = 0.0
6:    let j = 0
7:    while j < (K mod W) do             ▷ Process the remnant
8:      sum += (θ_local[j] × φ[c, j]); p_local[m]j] := sum
9:      j += 1
10:   while j < K do                     ▷ Process all W × W blocks
11:     for k from 0 through W − 1 do
12:       ▷ Next line uses transposed access to φ
13:       a_local[m, k] := θ_local[j + k] × φ[c_warp[r, k], j + r]
14:     for k from 0 through W − 1 do
15:       ▷ Next line uses transposed access to a_local, alas
16:       sum += a_local[q × W + k, r]
17:       p_local[m, j + k] := sum
18:     j += W
19: end
```

Algorithm 7 Drawing new z values using a butterfly table

```
1:  procedure DrawZ(N[M], θ[M, K], φ[V, K],
2:                  w[M][N]; output z[M, N])
3:    ▷ W (the "warp size") must be a power of 2,
4:    ▷ and M must be a multiple of W.
5:    for all 0 ≤ q < M/W do
6:      for SIMD 0 ≤ r < W do
7:        let m = q × W + r
8:        local array p′[K], θ_local[K]
9:        register array a_reg[W], c_warp[W]
10:       ⟨cache θ values into θ_local⟩
11:       let i_master = 0
12:       while any(i_master < N[m]) do
13:         let i = min(i_master, N[m] − 1)
14:         ⟨SIMD compute butterfly partial sums⟩
15:         let j = 0
16:         ⟨SIMD search butterfly partial sums⟩
17:         z[m, i] := j
18:         i_master += 1
19: end
```

blocks allows reduction of the cost of accessing data in main memory by performing *transposed accesses*.

The simplest use of transposed memory access occurs in Algorithm 5. For every document, a θ value is fetched for every topic. The topics are regarded as divided into a leading remnant (if any) and then a sequence of blocks of length W. The **while** loop on lines 3–4 handles the remnant, and then the following **while** loop processes successive blocks. On line 8 within the inner loop, note that the reference is to $\theta[q\times W+k, j+r]$ rather than the expected $\theta[q\times W+r, j+k]$ (which would be the same as $\theta[m, j + k]$ because $m = q \times W + r$). The result is that when the W threads of a SIMD warp execute this code and all access θ simultaneously, they access W consecutive memory locations, which can typically be fetched by a hardware memory controller much more efficiently than W memory locations separated by stride K. Another way to think about it is that on any given single iteration of the loop on lines 6–8 (which overall is designed to fetch one $W \times W$ block of θ values) instead of every thread in the warp fetching its kth value from the θ array, all the threads work together to fetch all W values that are needed by thread k of the warp. Each thread then stores what it has fetched into its local copy of the array θ_{local}.

Algorithm 6 compensates for this transposition of θ. The idea is also to divide each row of ϕ into blocks (possibly preceded by a remnant) and perform transposed accesses to ϕ. To do this, each thread needs to know which row of ϕ every other thread is interested in; this is done through the $W \times W$ local work array c_{warp}. In line 2, each thread figures out which word is the ith word of its document and calls it c; in lines 3–4 it then stores its value for c into every element of row r of the array c_{warp}. This is not an especially fast operation, but it pays for itself later on. The loop in lines 7–9 computes θ-ϕ products and partial sums p in the usual way (remember that the remnant in θ_{local} is not transposed), but the loop in lines 11–13 processes a block to compute product values to store into the a_{local} array; the access to ϕ on line 13 is transposed (note that the accesses to θ_{local} and c_{warp} are *not* transposed; because they were constructed and stored in transposed form, normal fetches cause their values to line up correctly with the ϕ values obtained by a transposed fetch). So this is pretty good; but in line 16 we finally pay the piper: in order to have the finally computed partial sums p reside in the correct lane for the binary search, it is necessary to perform a transposed access to a_{local} on line 16; but a_{local} is a local array, so transposed accesses are bad rather than good, and this occurs in an inner loop, so performance still suffers.

6. Using Butterfly-patterned Partial Sums

We avoid the cost of the final transposition of a_{local} by not requiring the partial sums table p for each thread to be entirely in the local memory of that thread. Instead, for each $W \times W$ block we use a butterfly-patterned table p'.

Our final version is Algorithm 7. It is similar to Algorithm 4, but declares all local arrays so as to be thread-local (and specifies that arrays a_{reg} and c_{warp} should reside in registers). It uses Algorithm 5 to cache θ values in θ_{local}, and also uses three new code chunks: Algorithms 8, 9, and 10. For Algorithm 7 to work properly, W must be a power of 2.

Algorithm 8 computes the butterfly-patterned table. The tricky part is the loop on lines 13–23, which is implemented by the alternate (faster) version of the CUDA code shown in Section 3.

Within a butterfly-patterned block of partial sums, Algorithm 9 performs a binary search as follows. The u' value is

Algorithm 8 Compute a butterfly-patterned table of sums

code chunk ⟨SIMD compute butterfly partial sums⟩:
  let $c = w[m][i]$
  for all $0 \le k < W$ **do**
    $c_{\text{warp}}[k] :=$ `__shfl`(c, k)
  let $sum = 0.0$, $j = 0$
  while $j < (K \bmod W)$ **do** ▷ Process the remnant
    $sum \mathrel{+}= (\theta_{\text{local}}[j] \times \phi[c, j])$; $p'[j] := sum$
    $j := j + 1$
  while $j < K$ **do** ▷ Process all $W \times W$ blocks
    for k from 0 through $W - 1$ **do**
      ▷ Next line uses transposed access to ϕ
      $a_{\text{reg}}[k] := \theta_{\text{local}}[j + k] \times \phi[c_{\text{warp}}[k], j + r]$
    for b from 0 through $(\log_2 W) - 1$ **do**
      let $bit = 2^b$
      for i from 0 through $\frac{W}{2 \times bit} - 1$ **do**
        let $d = 2 \times bit \times i + (bit - 1)$
        let $h =$ (**if** $(m \,\&\, bit) \neq 0$
                then $a_{\text{reg}}[d]$
                else $a_{\text{reg}}[d + bit]$)
        let $v =$ `__shfl_xor`(h, bit)
        if $(r \,\&\, bit) \neq 0$ **then** $a_{\text{reg}}[d] := a_{\text{reg}}[d + bit]$
        $a_{\text{reg}}[d + bit] := a_{\text{reg}}[d] + v$
        $p'[j + d] := a_{\text{reg}}[d]$
    $sum \mathrel{+}= a_{\text{reg}}[W - 1]$; $p'[W - 1] := sum$
    $j := j + W$
end

Algorithm 9 Searching within a butterfly-patterned table

code chunk ⟨SIMD search butterfly partial sums⟩:
  let $u =$ random value chosen from $[0.0, 1.0)$
  let $u' = sum \times u$, $j = 0$, $k = \left\lfloor \frac{K}{W} \right\rfloor - 1$
  let $searchBase = (K \bmod W) + (W - 1)$
  ▷ Binary search to find correct block of size W
  while $j < k$ **do**
    let $mid = \left\lfloor \dfrac{j + k}{2} \right\rfloor$
    if $u' < p'[mid \times W + searchBase]$ **then** $k := mid$
    else $j := mid + 1$
  let $blockBase = (K \bmod W) + j \times W$
  if $K \ge W$ **then**
    ⟨SIMD butterfly search one block⟩
  if $blockBase > 0$ **then**
    if $u' < p'[m, blockBase - 1]$ **then**
      ▷ Not in a block after all, so search remnant
      for i from 0 through $(K \bmod W) - 1$ **do**
        if $u' < p'[i]$ **then** { $j := i$; **break** }
end

computed exactly as in Algorithms 2 and 3, and a block to be searched is identified by performing a binary search on the subarray consisting of just the last row of each block; this identifies a specific block to search. If $K \ge W$, then some

Algorithm 10 Butterfly search within one $W \times W$ block

code chunk ⟨SIMD butterfly search one block⟩:
  let $lowValue =$ (**if** $blockBase > 0$
              then $p'[blockBase - 1]$
              else 0)
  let $highValue = p'[blockBase + (W - 1)]$
  let *flip* $= 0$
  ▷ Butterfly search within the block of size W
  for b from 0 through $(\log_2 W) - 1$ **do**
    let $bit = 2^{((\log_2 W) - 1) - b}$
    let $mask = ((W - 1) \times (2 \times bit)) \,\&\, (W - 1)$
    let $y = 0$
    for i from 0 through $\frac{W}{2 \times bit} - 1$ **do**
      let $d = (bit - 1) + 2 \times bit \times i$
      let $him = (d \,\&\, mask) + (r \,\&\, \neg mask)$
      let $hisBlockBase =$ `__shfl`$(blockBase, him)$
      let $t =$ `__shfl_xor`$(p'[hisBlockBase + d], flip)$
      if $((r \oplus d) \,\&\, mask) = 0$ **then** $y := t$
    let $compareValue =$ (**if** $(r \,\&\, bit) \neq 0$
                then $highValue - y$
                else $lowValue + y$)
    if $stop < compareValue$ **then**
      $highValue := compareValue$
      flip := *flip* $\oplus$ $(bit \,\&\, r)$
    else
      $lowValue := compareValue$
      flip := *flip* $\oplus$ $(bit \,\&\, \neg r)$
  $j := blockBase +$ (*flip* $\oplus$ r)
end

$W \times W$ block is identified, and it is searched, but it is possible that the desired u' value does not lie within that block; in that case, the remnant is searched using a linear search.

In order to search within a block, Algorithm 10 maintains two additional state variables $lowValue$ and $highValue$. An invariant is that if lane m has indices j through k of a block still under consideration, then $lowValue = m_0^{blockBase+j-1}$ and $highValue = m_0^{blockBase+k}$. In order to cut the search range in half, the binary search needs to compare the u' value to the midpoint value $m_0^{blockBase+mid}$ where $mid = \left\lfloor \frac{j+k}{2} \right\rfloor$; in Algorithm 3 this value is of course an entry in the p array, but in Algorithm 10 the midpoint value is *calculated* by choosing an appropriate entry from the butterfly-patterned p' array and then either adding it to $lowValue$ or subtracting it from $highValue$. Whether to add or subtract on iteration number b (where the $\log_2 W$ iterations are numbered starting from 0) depends on whether bit b (counting from the right starting at 0) of the binary representation of m is 0 or 1, respectively. Depending on the result of the comparison of the midpoint value with the u' value, the midpoint value is assigned to either $lowValue$ and $highValue$, maintaining the invariant, and a bit of a third state variable *flip* (initially 0) is updated. When the binary search is complete, the correct index to select is computed from the value in *flip*.

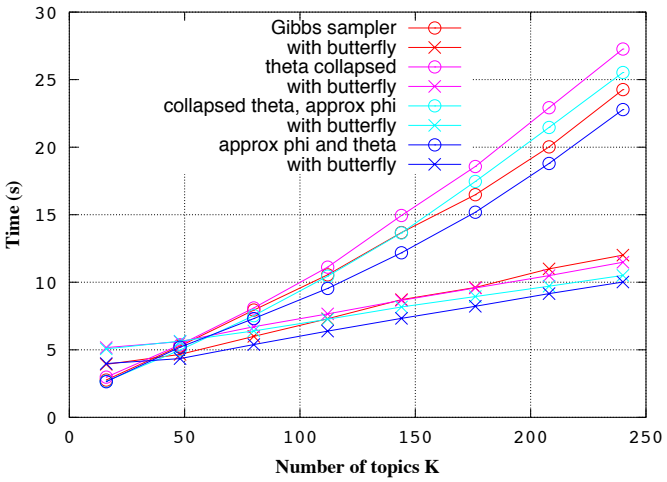


Figure 6. Measurements of execution time, with and without butterfly-patterned partial sums, for four variations of a machine-learning application ($K = 32k + 16, 0 \leq k \leq 7$)

The threads in a warp assist one another in fetching tree nodes using the loop in lines 12–17; the function `__shfl_xor` effects this data transfer in line 16.

7. Evaluation

We coded four versions of a complete LDA Gibbs-sampler topic-modeling algorithm in CUDA 6.5 for an NVIDIA Titan Black GPU ($W = 32$). For each version we tested two variants, one using Algorithm 1 (using the binary search of Algorithm 3) and one using Algorithm 7. (These algorithms are the ones on which we reported at ICML 2015 [25]; that paper includes only a passing mention of this use of butterfly-patterned partial sums, and refers to an early version of this paper [22].) All eight variants were tested for speed using a Wikipedia-based dataset with number of documents $M = 43556$, vocabulary size $V = 37286$, total number of words in corpus $\Sigma N = 3072662$ (therefore average document size $(\Sigma N)/M \approx 70.5$), and maximum document size $\max N = 307$. Each variant was measured using eight different values for the number of topics K (16, 48, 80, 112, 144, 176, 208, and 240), in each case performing 100 sampling iterations and measuring the execution time of the entire application, not just the part that draws z values. Best performance requires unrolling three loops in Algorithm 8; we had to manually unroll the loop that starts on line 13, and the CUDA compiler then automatically unrolled the loops that start on lines 10 and 15. The performance results are shown in Figure 6. The butterfly variants are faster for $K \geq 80$. For $K \geq 200$, for each of the four versions the butterfly variant is more than twice as fast.

In CUDA 7.5 on the same hardware, we measured Algorithms 1, 7, and a variant of 4 that transposes in registers (as in Figure 2) for 32-bit intermediate values (Figure 7(a)) and for 64-bit intermediate values (Figure 7(b)), using the same Wikipedia-based dataset. Each of the three algorithms was measured for $K = 4, 8, 12, 16, \ldots, 1024$, again in each case performing 100 sampling iterations and measuring the execution time of the entire application, not just the part that draws z values. Each data point shown in Figure 7 (and in Figure 8) is an average of five runs; the maximum relative standard deviation was 0.38. One can see that the measurements for Algorithms 4 and 7 have a distinctive sawtooth pattern: the amount by which a measurement dips below an upper-bounding line depends on the number of trailing zero-bits in the bionary representation of the length of the remnant (that is, the value of $K \bmod 32$).

For 32-bit intermediate data (Figure 7(a)), Algorithm 7 is faster than Algorithm 4 for all $K > 576$; moreover, it is also faster for all multiples of 32 greater than 64. For $K = 512$, Algorithm 7 is 8% faster than Algorithm 4; for $K = 1024$, it is 13% faster. For 64-bit intermediate data (Figure 7(b)), Algorithm 7 is faster than Algorithm 4 for all $K > 64$. For $K = 512$, Algorithm 7 is 33% faster than Algorithm 4; for $K = 1024$, it is 35% faster.

Measurements of just Algorithms 4 and 7, for all $480 \leq K \leq 544$ (not just multiples of 4), are shown in Figures 8(a) and 8(b), which exhibit the sawtooth pattern in greater detail in the measurements for both algorithms.

It is difficult to measure GPU memory bandwidth and computational costs directly, because CUDA "abstracts" the hardware architecture, and the optimizing compiler does extraordinarily complex instruction scheduling, so we relied entirely on measuring wall-clock time for entire application executions. Nevertheless, we can draw some inferences.

In each of Figures 7(a) and 7(b), the "register transpose" runs (Algorithm 4) shows the improvement over the baseline (Algorithm 1) that comes from the combination of using transposed (therefore coalesced) memory accesses, compensating for the transposition by shuffling in-register among lanes, and otherwise keeping the computation exactly the same. We may infer that the net improvement over the baseline is attributable entirely to improved use of memory bandwidth (less the cost of in-register shuffling). The "butterfly partial sums" run (Algorithm 7) shows additional improvement from applying the butterfly-patterned partial-sums trick to Algorithm 4, and this additional net improvement is attributable entirely to a combination of reduced computation and reduced register shuffling. Ideally, one would like to "complete the diamond" by evaluating the impact of the butterfly directly over the baseline, but the butterfly makes sense only in conjunction with transposed memory access. For $K = 1024$, then, it is fair to say that for 32-bit intermediate data, taking advantage of memory coalescing provides a 70% speed improvement over the baseline algorithm, and the butterfly-patterned partial-sums technique provides an additional improvement of 4 percentage points relative to baseline, for an overall speedup of 74% over the baseline; and also that for 64-bit intermediate data, taking advantage of memory coalescing provides

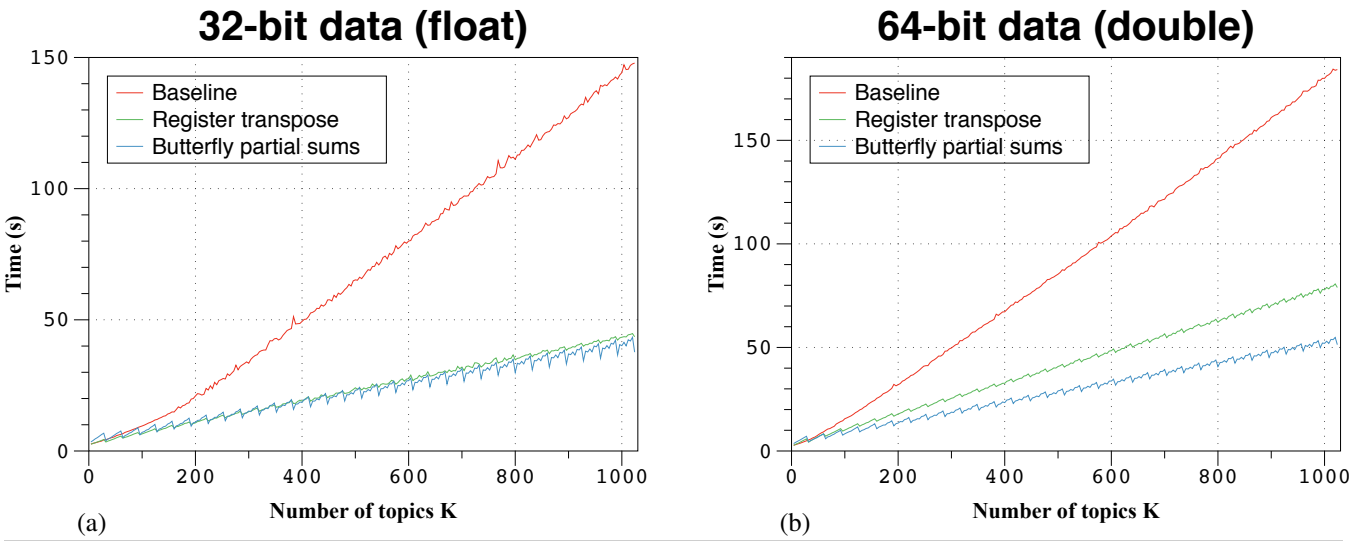


Figure 7. Two sets of measurements of execution time for a complete machine-learning application ($K = 4k + 4, 0 \le k \le 255$)

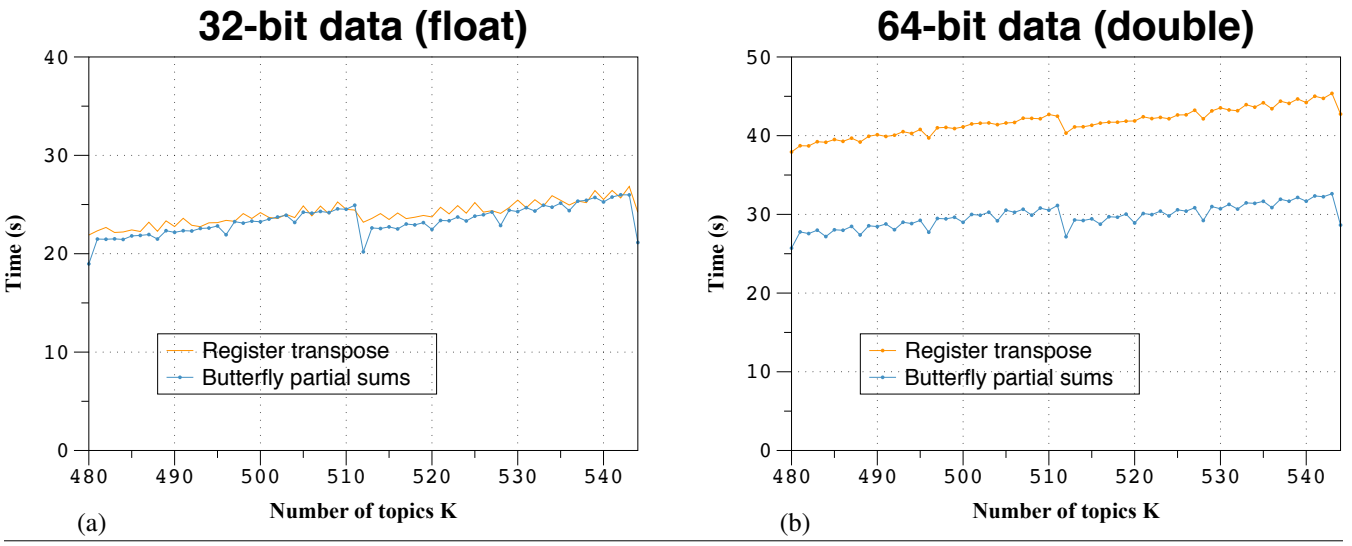


Figure 8. Two more sets of measurements of execution time for a complete machine-learning application ($480 \le K \le 544$)

a 56% speed improvement over the baseline algorithm, and the butterfly-patterned partial-sums technique provides an additional improvement of 15 percentage points relative to baseline, for an overall speedup of 71% over the baseline.

8. Related Work

Because the computed probabilities are relative in our LDA application, it is necessary to compute all of them and then to compute, if nothing else, their sum, so that the relative probabilities can be effectively normalized. Therefore every method for drawing from a discrete distribution represented by a set of relative probabilities involves some amount of preprocessing before drawing from the distribution. The various algorithms in the literature have differing tradeoffs according to what technique is used for preprocessing and what technique is used for drawing; some algorithms also accommodate incremental updating of the relative probabilities by providing a technique for incremental preprocessing.

Instead of doing a binary search on the partial sums, one can instead (as Marsaglia [18] observes in passing) construct a search tree using the principles of Huffman encoding [9] (independently rediscovered by Zimmerman[33]) to minimize the expected number of comparisons. In either case the complexity of the search is $O(\log n)$, but the optimized search may have a smaller constant, obtained at the expense of a preprocessing step that must sort the relative probabilities and therefore has complexity $\Omega(n \log n)$.

Walker [27, 28] describes what we now call the "alias" method, in which n relative probabilities are preprocessed into two additional tables F and A of length n. To draw a value from the distribution, let k be a integer chosen uniformly at random from $\{0, 1, 2, \ldots, n-1\}$ and let u be chosen uniformly at random from the real interval $[0, 1)$. Then

the value drawn is ($\textbf{if } u < F_k \textbf{ then } k \textbf{ else } A_k$). Therefore, once the tables F and A have been produced, the complexity of drawing a value from the distribution is $O(1)$, assuming that the cost of an array access is $O(1)$. Walker's method [28] for producing the tables F and A requires time $\Theta(n^2)$; it is easy to reduce this to $\Omega(n \log n)$ by sorting the probabilities [12, exercise 3.4.1-7] and then using, say, priority heaps instead of a list for the intermediate data structure. Either version heuristically attempts to minimize the probability of having to access the table A.

Vose [26] describes a preprocessing algorithm, with proof, that further reduces the preprocessing complexity of the alias method to $\Theta(n)$. The tradeoff that permits this improvement is that the preprocessing algorithm makes no attempt to minimize the probability of accessing the array A.

Matias *et al.* [19] describe a technique for preprocessing a set of relative probabilities into a set of trees, after which a sequence of intermixed generate (draw) and update operations can be performed, where an update operation changes just one of the relative probabilities; a single generate operation takes $O(\log^* n)$ expected time, and a single update operation takes $O(\log^* n)$ amortized expected time.

Li *et al.* [15] describe a modified LDA topic modeling algorithm, which they call Metropolis-Hastings-Walker sampling, that uses Walker's alias method but amortizes the cost of constructing the table by drawing from the same table during multiple consecutive sampling iterations of a Metropolis-Hastings sampler; their paper provides some justification for why it is acceptable to use a "slightly stale" alias table (their words) for the purposes of this application.

The trees embedded in the butterfly-patterned partial-sums table are reminiscent of Fenwick's binary-indexed trees [6], in that tree nodes containing partial sums are stored as array elements whose addresses are calculated through bit-manipulation of indices. However, the butterfly-patterned table as formulated here differs in three ways: (a) it stores partial sums for multiple distributions in a two-dimensional format rather than partial sums for a single distribution in a one-dimensional format; (b) it requires maintenance of two running values rather than one as a search descends the tree; and (c) at each step of the search it will always perform either an addition to one of the running values or subtraction from the other running value, whereas the Fenwick search always uses subtraction on its single running value, but at each step the subtraction is conditional.

There is a growing literature on interesting and clever techniques for improving the speed of parallel-prefix computations, especially on GPU architectures [5, 17, 31], and these techniques could possibly also be profitably applied to the problem of sampling from discrete distributions. However, we emphasize that, in contrast, the entire point of the butterfly-patterned partial-sums algorithm presented here is not to compute a complete prefix-sum table *faster*, but to *avoid* computing most of it in the first place.

9. Conclusions

This paper focuses on one low-level "utility algorithm": independent sampling from a large number of discrete distributions. The technique presented here can be compared to an optimization that applies "only" to sorting. But sorting is an operation of broad utility that can be exploited in a wide variety of applications. In the same way, drawing from multiple discrete distributions is the most important component of a wide class of machine learning algorithms: discrete latent variable models. This class encompasses mixture models (such as Gaussian mixture models), mixed membership models (such as topic models), mixtures of experts (such as probabilistic decision trees), learning meta-algorithms (such as Bayesian averaging), and more. Just in the specific case of LDA, we are currently aware of more than 50 variants. The currently most scalable and statistically efficient inference training procedures for LDA are based on the Stochastic Expectation Maximization variant of the Gibbs sampling procedure; they all use independent sampling as their most costly step, so improving the speed of independent sampling greatly improves the overall speed of these training algorithms. From an academic perspective, much recent work on Bayesian non-parametrics and hierarchical modeling builds upon LDA (for example, the Pachinko Allocation model and the Chinese Restaurant Franchise model). Speeding up LDA is a fundamental first step toward making such more advanced models practical. From an industrial perspective, LDA and its extensions are now making their way into useful tools and services, for example in product recommendation systems [8], consumer personalization systems [1], audience expansion systems for online advertisement [11], and document summarization [21]. It is precisely as this technology is being deployed that engineering for speed, such as we do in this paper, is most useful. Independent sampling is also used in chemistry and physics, for example to compute the ground state of Ising models (and Potts models) and to simulate Stochastic Cellular Automata.

The technique of constructing butterfly-patterned partial sums appears to be best suited for situations where a SIMD processor is used to compute tables of relative probabilities for multiple discrete distributions, each of which is then used just once to draw a single value, and where each thread, when computing its table, must fetch data from a contiguous region of memory whose address is computed from other data. The LDA application for which we developed the technique has these characteristics. The technique uses transposed memory access in order to allow a SIMD memory controller to touch at most three cache lines on each fetch, then cheaply constructs a butterfly-patterned set of partial sums that are just adequate to allow partial sums actually needed to be constructed on the fly during the course of a binary search. This butterfly-pattern approach provides significant speedup (up to 35%) over a transposition-only approach for our LDA machine-learning application.

References

[1] Amr Ahmed, Linagjie Hong, and Alexander J. Smola. Nested Chinese Restaurant Franchise Processes: Applications to user tracking and document modeling. In *Proc. ICML 2013: 30th International Conference on Machine Learning*, pages 1426–1434, Brookline, MA, June 2015. Microtome Publishing. URL: `http://www.jmlr.org/proceedings/papers/v28/ahmed13.pdf`.

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.

[3] David M. Blei. Probabilistic topic models. *Commun. ACM*, 55(4):77–84, April 2012. `doi:10.1145/2133806.2133826`.

[4] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet allocation. *J. Machine Learning Research*, 3:993–1022, March 2003. URL: `http://dl.acm.org/citation.cfm?id=944919.944937`.

[5] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms on graphics processors. In *Proc. 22nd Annual International Conference on Supercomputing*, ICS '08, pages 205–213, New York, 2008. ACM. `doi:10.1145/1375527.1375559`.

[6] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994. `doi:10.1002/spe.4380240306`.

[7] Thomas L. Griffiths and Mark Steyvers. Finding scientific topics. *Proc. National Academy of Sciences of the United States of America*, 101(suppl 1):5228–5235, 2004. `doi:10.1073/pnas.0307752101`.

[8] Diane Hu, Rob Hall, and Josh Attenberg. Style in the long tail: Discovering unique interests with latent variable models in large scale social E-commerce. In *Proc. 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1640–1649, New York, August 2014. ACM. URL: `http://doi.acm.org/10.1145/2623330.2623338`, `doi:10.1145/2623330.2623338`.

[9] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. IRE*, 40(9):1098–1101, Sept 1952. `doi:10.1109/JRPROC.1952.273898`.

[10] S. Lennart Johnsson, Tim Harris, and Kapil K. Mathur. Matrix multiplication on the Connection Machine. In *Proc. 1989 ACM/IEEE Conference on Supercomputing*, pages 326–332, New York, NY, USA, 1989. ACM. URL: `http://doi.acm.org/10.1145/76263.76298`.

[11] Joon Hee Kim, Amin Mantrach, Alejandro Jaimes, and Alice Oh. How to compete online for news audience: Modeling words that attract clicks. In *Proc. 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1645–1654, New York, August 2016. ACM. URL: `http://doi.acm.org/10.1145/2939672.2939873`, `doi:10.1145/2939672.2939873`.

[12] Donald E. Knuth. *Seminumerical Algorithms* (third edition), volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1998.

[13] Donald E. Knuth. *Sorting and Searching* (second edition), volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1998.

[14] Anthony Lee, Christopher Yau, Michael B. Giles, Arnaud Doucet, and Christopher C. Holmes. On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods. *J. Computational and Graphical Statistics*, 19(4):769–789, 2010. URL: `http://arxiv.org/pdf/0905.2441.pdf`.

[15] Aaron Q. Li, Amr Ahmed, Sujith Ravi, and Alexander J. Smola. Reducing the sampling complexity of topic models. In *Proc. 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 891–900, New York, 2014. ACM. `doi:10.1145/2623330.2623756`.

[16] Mian Lu, Ge Bai, Qiong Luo, Jie Tang, and Jiuxin Zhao. Accelerating topic model training on a single machine. In Yoshiharu Ishikawa, Jianzhong Li, Wei Wang, Rui Zhang, and Wenjie Zhang, editors, *Web Technologies and Applications (APWeb 2013)*, volume 7808 of *Lecture Notes in Computer Science*, pages 184–195. Springer Berlin Heidelberg, 2013. `doi:10.1007/978-3-642-37401-2_20`.

[17] Sepideh Maleki, Annie Yang, and Martin Burtscher. Higher-order and tuple-based massively-parallel prefix sums. In *Proc. 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 539–552, New York, 2016. ACM. URL: `http://doi.acm.org/10.1145/2908080.2908089`, `doi:10.1145/2908080.2908089`.

[18] G. Marsaglia. Generating discrete random variables in a computer. *Commun. ACM*, 6(1):37–38, January 1963. `doi:10.1145/366193.366228`.

[19] Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. Dynamic generation of discrete random variates. In *Proc. Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 361–370, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics. URL: `http://dl.acm.org/citation.cfm?id=313559.313807`.

[20] NVIDIA. Developer zone website: Cuda toolkit documentation: Cuda toolkit v6.5 programming guide, section B.14. warp shuffle functions, 2015. Online documentation. Accessed February 6, 2015. URL: `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-shuffle-functions`.

[21] Daniel Ramage, Susan Dumais, and Dan Liebling. Characterizing microblogs with topic models. In *Proc. 4th International AAAI Conference on Weblogs and Social Media*, pages 130–137, Palo Alto, CA, July 2010. Association for the Advancement of Artificial Intelligence.

[22] Guy L. Steele Jr. and Jean-Baptiste Tristan. Using butterfly-patterned partial sums to optimize GPU memory accesses for drawing from discrete distributions. *CoRR (Computing Research Repository at arXiv.org)*, May 2015. URL: `http://arxiv.org/abs/1505.03851`.

[23] Marc A. Suchard, Quanli Wang, Cliburn Chan, Jacob Frelinger, Andrew Cron, and Mike West. Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures. *J. Computational and Graphical Statistics*, 19(2):419–438, 2010.

[24] Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C. Pocock, Stephen Green, and Guy L. Steele Jr. Augur: Data-parallel probabilistic modeling. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2600–2608. Curran Associates, Inc., 2014. URL: `http://papers.nips.cc/book/year-2014`.

[25] Jean-Baptiste Tristan, Joseph Tassarotti, and Guy L. Steele Jr. Efficient training of LDA on a GPU by mean-for-mode estimation. In *Proc. ICML 2015: 32nd International Conference on Machine Learning*, pages 59–68, Brookline, MA, July 2015. Microtome Publishing. URL: `http://jmlr.org/proceedings/papers/v37/tristan15.pdf`.

[26] M. D. Vose. A linear algorithm for generating random numbers with a given distribution. *IEEE Trans. Software Engineering*, 17(9):972–975, Sept 1991. `doi:10.1109/32.92917`.

[27] A. J. Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters*, 10(8):127–128, April 1974. `doi:10.1049/el:19740097`.

[28] Alastair J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Trans. Math. Software*, 3(3):253–256, September 1977. `doi:10.1145/355744.355749`.

[29] Nicholas Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley, Upper Saddle River, New Jersey, 2013.

[30] Feng Yan, Ningyi Xu, and Yuan Qi. Parallel inference for latent Dirichlet allocation on graphics processing units. In *Advances in Neural Information Processing Systems 22*, pages 2134–2142. Curran Associates, Inc., 2009. URL: `http://papers.nips.cc/book/year-2009`.

[31] Shengen Yan, Guoping Long, and Yunquan Zhang. StreamScan: Fast scan algorithms for GPUs without global barrier synchronization. In *Proc. 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 229–238, New York, 2013. ACM. URL: `http://doi.acm.org/10.1145/2442516.2442539`, `doi:10.1145/2442516.2442539`.

[32] Huasha Zhao, Biye Jiang, and John Canny. SAME but different: Fast and high-quality Gibbs parameter estimation. *CoRR (Computing Research Repository at arXiv.org)*, September 2014. URL: `http://arxiv.org/abs/1409.5402`.

[33] Seth Zimmerman. An optimal search procedure. *American Mathematical Monthly*, 66(8):690–693, Oct 1959.