

Using Butterfly-patterned Partial Sums to Draw from Discrete Distributions

GUY L. STEELE JR. and JEAN-BAPTISTE TRISTAN, Oracle Labs, USA

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 1,024 topics 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.

CCS Concepts: • **Mathematics of computing** → **Gibbs sampling**; • **Theory of computation** → **Sorting and searching**; **Concurrent algorithms**; • **Computing methodologies** → **Concurrent algorithms**; • **Computer systems organization** → **Single instruction, multiple data**;

Additional Key Words and Phrases: Butterfly, coalesced memory access, discrete distribution, GPU, latent Dirichlet allocation, LDA, machine learning, multithreading, memory bottleneck, parallel computing, random sampling, SIMD, transposed memory access

ACM Reference format:

Guy L. Steele Jr. and Jean-Baptiste Tristan. 2019. Using Butterfly-patterned Partial Sums to Draw from Discrete Distributions. *ACM Trans. Parallel Comput.* 6, 4, Article 22 (November 2019), 30 pages.

<https://doi.org/10.1145/3365662>

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. [26] 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

Authors’ addresses: G. L. Steele Jr. and J.-B. Tristan, Oracle Labs, 35 Network Drive UBUR02-313, Burlington, MA, 01803, USA; emails: {steele, jean.baptiste.tristan}@oracle.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.

2329-4949/2019/11-ART22 \$15.00

<https://doi.org/10.1145/3365662>

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 article 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 article, 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 $0 \leq j < K$ 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 article.) 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 improved to run much faster by (a) doing a bit of algebra to eliminate the division operations and (b) using a binary search:

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 $[0, 1)$.
3. Let $u' = p_{(K-1)} \times u$.
4. 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 (four-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 ($0 \leq j < 32$), lane i ($0 \leq i < 32$) fetches not $a[i][j]$ but instead $a[j][i]$. (Compare, for example, the storage of floating-point numbers as “slice-wise” rather than “field-wise” 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.

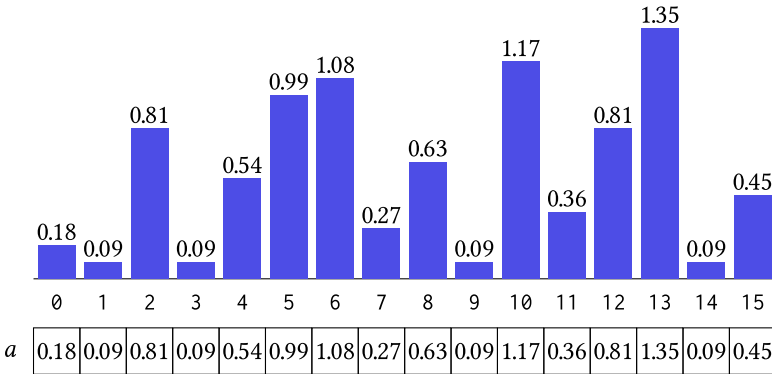
The novel contribution of this article is to observe and then exploit the fact that the binary search of the array p (which is computed from a) does not access all entries of p ; in fact, for an array of size K it examines only about $\log_2 K$ entries. Therefore, it is not necessary to compute all entries 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, 27, 33] as well as uncollapsed samplers and may also be useful for GPU implementations of other algorithms [35] whose inner loops sample from discrete distributions.

This article 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 [28] and had already been carefully tuned for speed before application of the techniques described in this article.

(This article is a revised and expanded version of Reference [25]. There is also video of a talk based on this material [22], and the slides for the talk are separately available [23].)

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 ($0 \leq j < K$, and for purposes of illustration, we will use $K = 16$):



From a , compute the prefix-sum array p :

```

1: let  $sum = 0.0$ 
2: for  $k$  from 0 through  $K - 1$  do
3:    $sum += a[k]$ 
4:    $p[k] := sum$ 
5: end for

```

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 can do the job:

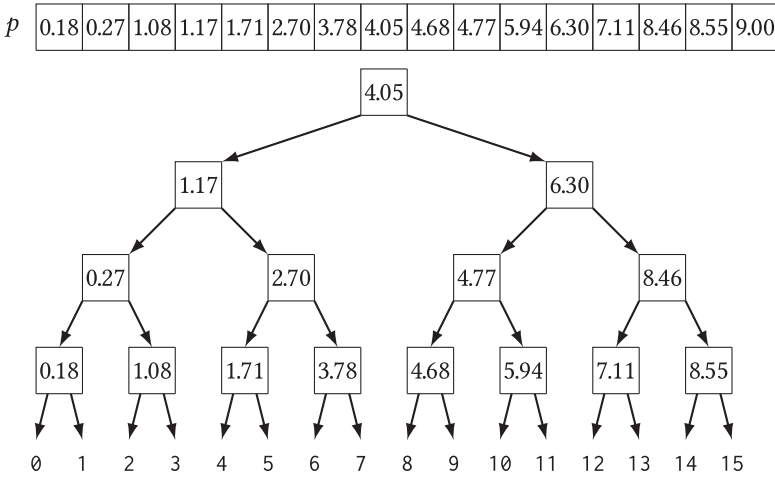
- 1: **let** u = value chosen uniformly at random (or pseudorandomly) from $[0.0, 1.0)$
- 2: **let** $u' = \text{sum} \times u$
- 3: **let** $j = 0$
- 4: **while** $j < K - 1$ and $u' \geq p[j]$ **do** $j += 1$

It is easy to see that, on completion of the search loop, j will be the index of the leftmost (lowest-indexed) element of p smaller than u' .

Alternatively, one may use a binary search:

- 1: **let** u = value chosen uniformly at random (or pseudorandomly) from $[0.0, 1.0)$
- 2: **let** $u' = \text{sum} \times u$
- 3: **let** $j = 0$
- 4: **let** $k = K - 1$
- 5: **while** $j < k$ **do**
- 6: **let** $\text{mid} = \lfloor \frac{j+k}{2} \rfloor$
- 7: **if** $u' < p[\text{mid}]$ **then** $k := \text{mid}$ **else** $j := \text{mid} + 1$
- 8: **end while**

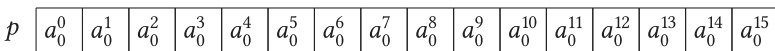
A binary search on an array amounts to walking down a binary tree whose leaves are the array indices and whose internal nodes are labeled with all entries except the last. We can draw such a tree by starting with a drawing of the array and then displacing each entry (except the last) vertically:



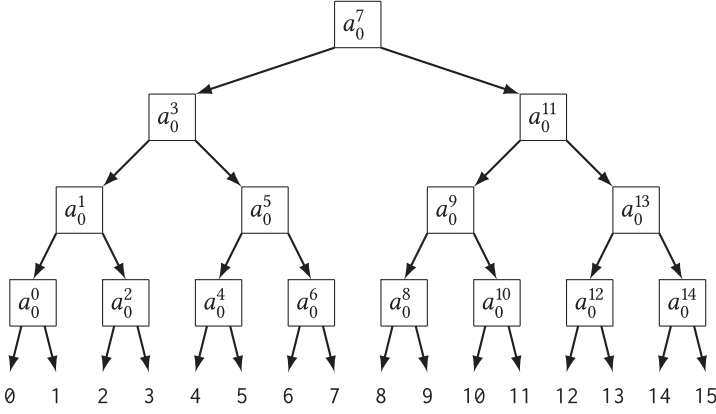
Starting from the root, we compare the quantity we are searching for to the label of each internal node that we encounter; if it is smaller, we descend to the left child, otherwise to the right child. When the process eventually arrives at a leaf, that is the desired index into the array a .

We can picture the general process by using a convenient abbreviation: a_m^n means $\sum_{m \leq i \leq n} a_i$.

Then array p looks like this:



and the corresponding binary tree looks like this:

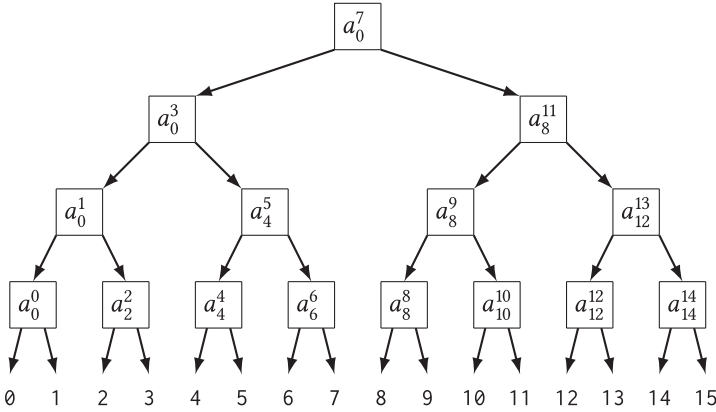


One of the central ideas in this article is that instead of computing the prefix-sum p of a , we can compute a different set of partial sums, which we will call p' :

- 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: $p'[(2i)2^{b-1} - 1] += p'[(2i - 1)2^{b-1} - 1]$
- 5: **end for**
- 6: **end for**

$$p' \quad \boxed{a_0^0} \quad \boxed{a_0^1} \quad \boxed{a_2^2} \quad \boxed{a_0^3} \quad \boxed{a_4^4} \quad \boxed{a_4^5} \quad \boxed{a_6^6} \quad \boxed{a_0^7} \quad \boxed{a_8^8} \quad \boxed{a_8^9} \quad \boxed{a_{10}^{10}} \quad \boxed{a_8^{11}} \quad \boxed{a_{12}^{12}} \quad \boxed{a_{12}^{13}} \quad \boxed{a_{14}^{14}} \quad \boxed{a_0^{15}}$$

and we can likewise treat this array p' as a binary tree to be searched:



Notice that the partial sums in p' are generally “smaller” than those in p —that is, they are sums over fewer elements of a . In fact, half of them are equal to single elements of a ; for example, $a_2^2 = \sum_{2 \leq i \leq 2} a_i = a_2$.

Here, then, is 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  $u$  = value chosen uniformly at random (or pseudorandomly) from  $[0.0, 1.0)$ 
2: let  $u' = \text{sum} \times u$ 
3: let  $j = 0$ 
4: let  $k = K - 1$ 
5: let  $\text{lowValue} = 0$ 
6: while  $j < k$  do
7:   let  $\text{mid} = \lfloor \frac{j+k}{2} \rfloor$ 
8:   let  $\text{compareValue} = \text{lowValue} + p'[\text{mid}]$ 
9:   if  $u' < \text{compareValue}$  then
10:      $k := \text{mid}$ 
11:   else
12:      $j := \text{mid} + 1$ 
13:      $\text{lowValue} := \text{compareValue}$ 
14:   end if
15: end while

```

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 $W = 16$, to make the figures a manageable size while keeping fonts at a readable size. Warps are automatically scheduled 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, 32]; 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. If the lanes happen to fetch from consecutive memory locations, then maximal coalescing occurs. For the NVIDIA Titan Black processors we used, $W = 32$ and each cache line is 128 bytes. If lane k fetches a 32-bit word at address $x + 4k$, then the accessed words occupy exactly 2 cache lines if x is a multiple of 128, and otherwise straddle 3 cache lines; if lane k fetches a 64-bit word at address $x + 8k$, then the accessed words occupy exactly 4 cache lines if x is a multiple of 128, and otherwise straddle 5 cache lines. Any of these situations is a great improvement over having to access 32 distinct cache lines.

In our LDA application, we have many documents to be processed, each with a different number of words (we pre-sort 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 k th 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 k th 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 assume $K = W$. (We lift this restriction in Section 5.)

If we use a straightforward approach, then 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 P 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. 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 pre-transposed, 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, 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 (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 $\frac{1}{2}W \log_2 W$.

lane	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
0	A ₀	B ₀	C ₀	D ₀	E ₀	F ₀	G ₀	H ₀	I ₀	J ₀	K ₀	L ₀	M ₀	N ₀	O ₀	P ₀
1	A ₁	B ₁	C ₁	D ₁	E ₁	F ₁	G ₁	H ₁	I ₁	J ₁	K ₁	L ₁	M ₁	N ₁	O ₁	P ₁
2	A ₂	B ₂	C ₂	D ₂	E ₂	F ₂	G ₂	H ₂	I ₂	J ₂	K ₂	L ₂	M ₂	N ₂	O ₂	P ₂
3	A ₃	B ₃	C ₃	D ₃	E ₃	F ₃	G ₃	H ₃	I ₃	J ₃	K ₃	L ₃	M ₃	N ₃	O ₃	P ₃
4	A ₄	B ₄	C ₄	D ₄	E ₄	F ₄	G ₄	H ₄	I ₄	J ₄	K ₄	L ₄	M ₄	N ₄	O ₄	P ₄
5	A ₅	B ₅	C ₅	D ₅	E ₅	F ₅	G ₅	H ₅	I ₅	J ₅	K ₅	L ₅	M ₅	N ₅	O ₅	P ₅
6	A ₆	B ₆	C ₆	D ₆	E ₆	F ₆	G ₆	H ₆	I ₆	J ₆	K ₆	L ₆	M ₆	N ₆	O ₆	P ₆
7	A ₇	B ₇	C ₇	D ₇	E ₇	F ₇	G ₇	H ₇	I ₇	J ₇	K ₇	L ₇	M ₇	N ₇	O ₇	P ₇
8	A ₈	B ₈	C ₈	D ₈	E ₈	F ₈	G ₈	H ₈	I ₈	J ₈	K ₈	L ₈	M ₈	N ₈	O ₈	P ₈
9	A ₉	B ₉	C ₉	D ₉	E ₉	F ₉	G ₉	H ₉	I ₉	J ₉	K ₉	L ₉	M ₉	N ₉	O ₉	P ₉
10	A ₁₀	B ₁₀	C ₁₀	D ₁₀	E ₁₀	F ₁₀	G ₁₀	H ₁₀	I ₁₀	J ₁₀	K ₁₀	L ₁₀	M ₁₀	N ₁₀	O ₁₀	P ₁₀
11	A ₁₁	B ₁₁	C ₁₁	D ₁₁	E ₁₁	F ₁₁	G ₁₁	H ₁₁	I ₁₁	J ₁₁	K ₁₁	L ₁₁	M ₁₁	N ₁₁	O ₁₁	P ₁₁
12	A ₁₂	B ₁₂	C ₁₂	D ₁₂	E ₁₂	F ₁₂	G ₁₂	H ₁₂	I ₁₂	J ₁₂	K ₁₂	L ₁₂	M ₁₂	N ₁₂	O ₁₂	P ₁₂
13	A ₁₃	B ₁₃	C ₁₃	D ₁₃	E ₁₃	F ₁₃	G ₁₃	H ₁₃	I ₁₃	J ₁₃	K ₁₃	L ₁₃	M ₁₃	N ₁₃	O ₁₃	P ₁₃
14	A ₁₄	B ₁₄	C ₁₄	D ₁₄	E ₁₄	F ₁₄	G ₁₄	H ₁₄	I ₁₄	J ₁₄	K ₁₄	L ₁₄	M ₁₄	N ₁₄	O ₁₄	P ₁₄
15	A ₁₅	B ₁₅	C ₁₅	D ₁₅	E ₁₅	F ₁₅	G ₁₅	H ₁₅	I ₁₅	J ₁₅	K ₁₅	L ₁₅	M ₁₅	N ₁₅	O ₁₅	P ₁₅

(a) Normal per-lane arrays

lane	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
0	A ₀	A ₁	A ₂	A ₃	A ₄	A ₅	A ₆	A ₇	A ₈	A ₉	A ₁₀	A ₁₁	A ₁₂	A ₁₃	A ₁₄	A ₁₅
1	B ₀	B ₁	B ₂	B ₃	B ₄	B ₅	B ₆	B ₇	B ₈	B ₉	B ₁₀	B ₁₁	B ₁₂	B ₁₃	B ₁₄	B ₁₅
2	C ₀	C ₁	C ₂	C ₃	C ₄	C ₅	C ₆	C ₇	C ₈	C ₉	C ₁₀	C ₁₁	C ₁₂	C ₁₃	C ₁₄	C ₁₅
3	D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇	D ₈	D ₉	D ₁₀	D ₁₁	D ₁₂	D ₁₃	D ₁₄	D ₁₅
4	E ₀	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	E ₇	E ₈	E ₉	E ₁₀	E ₁₁	E ₁₂	E ₁₃	E ₁₄	E ₁₅
5	F ₀	F ₁	F ₂	F ₃	F ₄	F ₅	F ₆	F ₇	F ₈	F ₉	F ₁₀	F ₁₁	F ₁₂	F ₁₃	F ₁₄	F ₁₅
6	G ₀	G ₁	G ₂	G ₃	G ₄	G ₅	G ₆	G ₇	G ₈	G ₉	G ₁₀	G ₁₁	G ₁₂	G ₁₃	G ₁₄	G ₁₅
7	H ₀	H ₁	H ₂	H ₃	H ₄	H ₅	H ₆	H ₇	H ₈	H ₉	H ₁₀	H ₁₁	H ₁₂	H ₁₃	H ₁₄	H ₁₅
8	I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇	I ₈	I ₉	I ₁₀	I ₁₁	I ₁₂	I ₁₃	I ₁₄	I ₁₅
9	J ₀	J ₁	J ₂	J ₃	J ₄	J ₅	J ₆	J ₇	J ₈	J ₉	J ₁₀	J ₁₁	J ₁₂	J ₁₃	J ₁₄	J ₁₅
10	K ₀	K ₁	K ₂	K ₃	K ₄	K ₅	K ₆	K ₇	K ₈	K ₉	K ₁₀	K ₁₁	K ₁₂	K ₁₃	K ₁₄	K ₁₅
11	L ₀	L ₁	L ₂	L ₃	L ₄	L ₅	L ₆	L ₇	L ₈	L ₉	L ₁₀	L ₁₁	L ₁₂	L ₁₃	L ₁₄	L ₁₅
12	M ₀	M ₁	M ₂	M ₃	M ₄	M ₅	M ₆	M ₇	M ₈	M ₉	M ₁₀	M ₁₁	M ₁₂	M ₁₃	M ₁₄	M ₁₅
13	N ₀	N ₁	N ₂	N ₃	N ₄	N ₅	N ₆	N ₇	N ₈	N ₉	N ₁₀	N ₁₁	N ₁₂	N ₁₃	N ₁₄	N ₁₅
14	O ₀	O ₁	O ₂	O ₃	O ₄	O ₅	O ₆	O ₇	O ₈	O ₉	O ₁₀	O ₁₁	O ₁₂	O ₁₃	O ₁₄	O ₁₅
15	P ₀	P ₁	P ₂	P ₃	P ₄	P ₅	P ₆	P ₇	P ₈	P ₉	P ₁₀	P ₁₁	P ₁₂	P ₁₃	P ₁₄	P ₁₅

(b) Transposed arrays

Fig. 1. Array storage in lane registers (illustrated for $W = 16$). Each column represents W consecutive registers in the register file of one lane. Array elements are numbered, and lanes are labeled by letters, so “C₄” denotes element 4 of the array that logically belongs to (is intended to be processed by) lane C. In the normal layout, all array elements to be processed by lane C reside in the registers of lane C, but in the transposed layout, only element C₂ resides in a register of lane C—all other elements reside in other lanes (in register 2 of each lane).

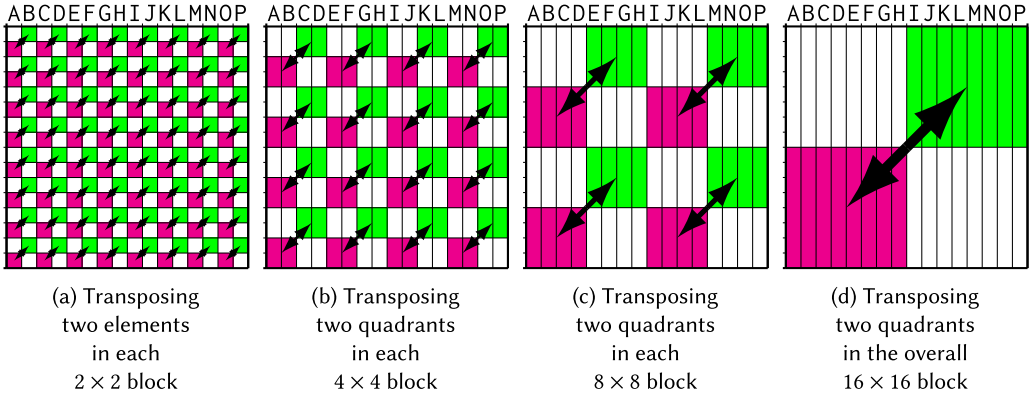


Fig. 2. Transposing a matrix consisting of W registers in each of the W lanes of the GPU (shown for $W = 16$).

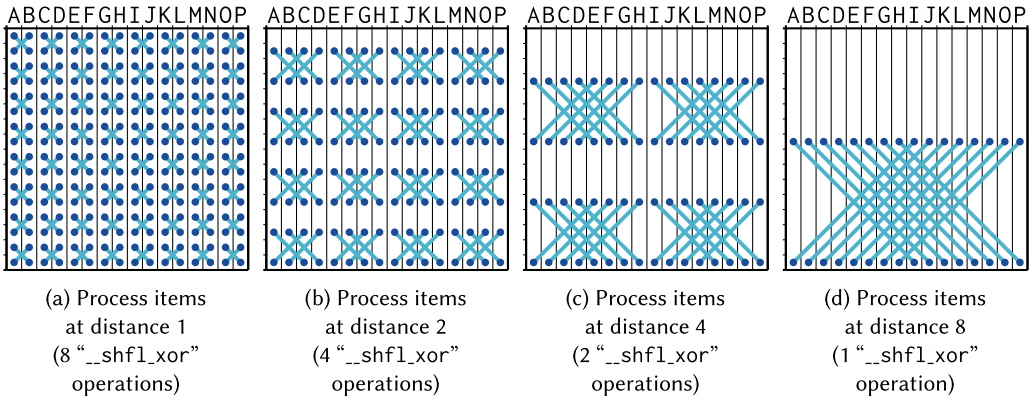


Fig. 3. Patterns of application of butterfly computations to compute butterfly-patterned p' arrays ($W = 16$).

We offer a novel approach that uses $3(W - 1)$ shuffle operations¹ 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 as in the transposed layout shown in Figure 1(b), 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 ($0 \leq i < \log_2 W$), $2^{(\log_2 W) - i - 1}$ __shfl_xor instructions are used to perform $(2^{(\log_2 W) - i - 1}) \frac{W}{2}$ butterfly computations (see Figure 3). Each butterfly computation operates on four entries, within

¹We 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 operations as a matter of mathematical principle, but we 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 the future, if GPU chips are ever built with $W \geq 64$, then the improvement in algorithmic complexity may also begin to matter.

lane	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
4	A ₀ ⁰	B ₀ ⁰	A ₂ ²	B ₂ ²	A ₄ ⁴	B ₄ ⁴	A ₆ ⁶	B ₆ ⁶	A ₈ ⁸	B ₈ ⁸	A ₁₀ ¹⁰	B ₁₀ ¹⁰	A ₁₂ ¹²	B ₁₂ ¹²	A ₁₄ ¹⁴	B ₁₄ ¹⁴
3	A ₀ ¹	B ₀ ¹	C ₀ ¹	D ₀ ¹	A ₄ ⁵	B ₄ ⁵	C ₄ ⁵	D ₄ ⁵	A ₈ ⁹	B ₈ ⁹	C ₈ ⁹	D ₈ ⁹	A ₁₂ ¹³	B ₁₂ ¹³	C ₁₂ ¹³	D ₁₂ ¹³
4	C ₀ ⁰	D ₀ ⁰	C ₂ ²	D ₂ ²	C ₄ ⁴	D ₄ ⁴	C ₆ ⁶	D ₆ ⁶	C ₈ ⁸	D ₈ ⁸	C ₁₀ ¹⁰	D ₁₀ ¹⁰	C ₁₂ ¹²	D ₁₂ ¹²	C ₁₄ ¹⁴	D ₁₄ ¹⁴
2	A ₀ ³	B ₀ ³	C ₀ ³	D ₀ ³	E ₀ ³	F ₀ ³	G ₀ ³	H ₀ ³	A ₁₁ ¹¹	B ₁₁ ¹¹	C ₁₁ ¹¹	D ₁₁ ¹¹	E ₁₁ ¹¹	F ₁₁ ¹¹	G ₁₁ ¹¹	H ₁₁ ¹¹
4	E ₀ ⁰	F ₀ ⁰	E ₂ ²	F ₂ ²	E ₄ ⁴	F ₄ ⁴	E ₆ ⁶	F ₆ ⁶	E ₈ ⁸	F ₈ ⁸	E ₁₀ ¹⁰	F ₁₀ ¹⁰	E ₁₂ ¹²	F ₁₂ ¹²	E ₁₄ ¹⁴	F ₁₄ ¹⁴
3	E ₀ ¹	F ₀ ¹	G ₀ ¹	H ₀ ¹	E ₄ ⁵	F ₄ ⁵	G ₄ ⁵	H ₄ ⁵	E ₈ ⁹	F ₈ ⁹	G ₈ ⁹	H ₈ ⁹	E ₁₂ ¹³	F ₁₂ ¹³	G ₁₂ ¹³	H ₁₂ ¹³
4	G ₀ ⁰	H ₀ ⁰	G ₂ ²	H ₂ ²	G ₄ ⁴	H ₄ ⁴	G ₆ ⁶	H ₆ ⁶	G ₈ ⁸	H ₈ ⁸	G ₁₀ ¹⁰	H ₁₀ ¹⁰	G ₁₂ ¹²	H ₁₂ ¹²	G ₁₄ ¹⁴	H ₁₄ ¹⁴
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 ₀ ⁷	J ₀ ⁷	K ₀ ⁷	L ₀ ⁷	M ₀ ⁷	N ₀ ⁷	O ₀ ⁷	P ₀ ⁷
4	I ₀ ⁰	J ₀ ⁰	I ₂ ²	J ₂ ²	I ₄ ⁴	J ₄ ⁴	I ₆ ⁶	J ₆ ⁶	I ₈ ⁸	J ₈ ⁸	I ₁₀ ¹⁰	J ₁₀ ¹⁰	I ₁₂ ¹²	J ₁₂ ¹²	I ₁₄ ¹⁴	J ₁₄ ¹⁴
3	I ₀ ¹	J ₀ ¹	K ₀ ¹	L ₀ ¹	I ₄ ⁵	J ₄ ⁵	K ₄ ⁵	L ₄ ⁵	I ₈ ⁹	J ₈ ⁹	K ₈ ⁹	L ₈ ⁹	I ₁₂ ¹³	J ₁₂ ¹³	K ₁₂ ¹³	L ₁₂ ¹³
4	K ₀ ⁰	L ₀ ⁰	K ₂ ²	L ₂ ²	K ₄ ⁴	L ₄ ⁴	K ₆ ⁶	L ₆ ⁶	K ₈ ⁸	L ₈ ⁸	K ₁₀ ¹⁰	L ₁₀ ¹⁰	K ₁₂ ¹²	L ₁₂ ¹²	K ₁₄ ¹⁴	L ₁₄ ¹⁴
2	I ₀ ³	J ₀ ³	K ₀ ³	L ₀ ³	M ₀ ³	N ₀ ³	O ₀ ³	P ₀ ³	I ₁₁ ¹¹	J ₁₁ ¹¹	K ₁₁ ¹¹	L ₁₁ ¹¹	M ₁₁ ¹¹	N ₁₁ ¹¹	O ₁₁ ¹¹	P ₁₁ ¹¹
4	M ₀ ⁰	N ₀ ⁰	M ₂ ²	N ₂ ²	M ₄ ⁴	N ₄ ⁴	M ₆ ⁶	N ₆ ⁶	M ₈ ⁸	N ₈ ⁸	M ₁₀ ¹⁰	N ₁₀ ¹⁰	M ₁₂ ¹²	N ₁₂ ¹²	M ₁₄ ¹⁴	N ₁₄ ¹⁴
3	M ₀ ¹	N ₀ ¹	O ₀ ¹	P ₀ ¹	M ₄ ⁵	N ₄ ⁵	O ₄ ⁵	P ₄ ⁵	M ₈ ⁹	N ₈ ⁹	O ₈ ⁹	P ₈ ⁹	M ₁₂ ¹³	N ₁₂ ¹³	O ₁₂ ¹³	P ₁₂ ¹³
4	O ₀ ⁰	P ₀ ⁰	O ₂ ²	P ₂ ²	O ₄ ⁴	P ₄ ⁴	O ₆ ⁶	P ₆ ⁶	O ₈ ⁸	P ₈ ⁸	O ₁₀ ¹⁰	P ₁₀ ¹⁰	O ₁₂ ¹²	P ₁₂ ¹²	O ₁₄ ¹⁴	P ₁₄ ¹⁴
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

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

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 $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$; they are replaced by $\begin{bmatrix} a & b \\ a+b & c+d \end{bmatrix}$. Each such replacement operation on four entries is symbolized by \boxtimes in the figures. Here is CUDA code for constructing the butterfly-patterned table using this replacement operation:

```

int r = threadIdx.x & 0x1f;    /* lane ID */
for (int b=1; 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;
        else a[d+(1<<b)] = v;
        a[d+(1<<b)] += a[d];
        p[d] = a[d];
    }
}
p[W-1] = a[W-1];

```

These 3 lines are replaced below to make a new version.

The result is shown in Figure 4. 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.

lane	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
4	A ⁰ ₀	B ⁰ ₀	A ² ₂	B ² ₂	A ⁴ ₄	B ⁴ ₄	A ⁶ ₆	B ⁶ ₆	A ⁸ ₈	B ⁸ ₈	A ¹⁰ ₁₀	B ¹⁰ ₁₀	A ¹² ₁₂	B ¹² ₁₂	A ¹⁴ ₁₄	B ¹⁴ ₁₄
3	A ¹ ₀	B ¹ ₀	C ¹ ₀	D ¹ ₀	A ⁵ ₄	B ⁵ ₄	C ⁵ ₄	D ⁵ ₄	A ⁹ ₈	B ⁹ ₈	C ⁹ ₈	D ⁹ ₈	A ¹³ ₁₂	B ¹³ ₁₂	C ¹³ ₁₂	D ¹³ ₁₂
4	C ⁰ ₀	D ⁰ ₀	C ² ₂	D ² ₂	C ⁴ ₄	D ⁴ ₄	C ⁶ ₆	D ⁶ ₆	C ⁸ ₈	D ⁸ ₈	C ¹⁰ ₁₀	D ¹⁰ ₁₀	C ¹² ₁₂	D ¹² ₁₂	C ¹⁴ ₁₄	D ¹⁴ ₁₄
2	A ³ ₀	B ³ ₀	C ³ ₀	D ³ ₀	E ³ ₀	F ³ ₀	G ³ ₀	H ³ ₀	A ¹¹ ₈	B ¹¹ ₈	C ¹¹ ₈	D ¹¹ ₈	E ¹¹ ₈	F ¹¹ ₈	G ¹¹ ₈	H ¹¹ ₈
3	E ⁰ ₀	F ⁰ ₀	E ² ₂	F ² ₂	E ⁴ ₄	F ⁴ ₄	E ⁶ ₆	F ⁶ ₆	E ⁸ ₈	F ⁸ ₈	E ¹⁰ ₁₀	F ¹⁰ ₁₀	E ¹² ₁₂	F ¹² ₁₂	E ¹⁴ ₁₄	F ¹⁴ ₁₄
4	E ¹ ₀	F ¹ ₀	G ¹ ₀	H ¹ ₀	E ⁵ ₄	F ⁵ ₄	G ⁵ ₄	H ⁵ ₄	E ⁹ ₈	F ⁹ ₈	G ⁹ ₈	H ⁹ ₈	E ¹³ ₁₂	F ¹³ ₁₂	G ¹³ ₁₂	H ¹³ ₁₂
3	G ⁰ ₀	H ⁰ ₀	G ² ₂	H ² ₂	G ⁴ ₄	H ⁴ ₄	G ⁶ ₆	H ⁶ ₆	G ⁸ ₈	H ⁸ ₈	G ¹⁰ ₁₀	H ¹⁰ ₁₀	G ¹² ₁₂	H ¹² ₁₂	G ¹⁴ ₁₄	H ¹⁴ ₁₄
4	A ⁷ ₀	B ⁷ ₀	C ⁷ ₀	D ⁷ ₀	E ⁷ ₀	F ⁷ ₀	G ⁷ ₀	H ⁷ ₀	I ⁷ ₀	J ⁷ ₀	K ⁷ ₀	L ⁷ ₀	M ⁷ ₀	N ⁷ ₀	O ⁷ ₀	P ⁷ ₀
4	I ⁰ ₀	J ⁰ ₀	I ² ₂	J ² ₂	I ⁴ ₄	J ⁴ ₄	I ⁶ ₆	J ⁶ ₆	I ⁸ ₈	J ⁸ ₈	I ¹⁰ ₁₀	J ¹⁰ ₁₀	I ¹² ₁₂	J ¹² ₁₂	I ¹⁴ ₁₄	J ¹⁴ ₁₄
3	I ¹ ₀	J ¹ ₀	K ¹ ₀	L ¹ ₀	I ⁵ ₄	J ⁵ ₄	K ⁵ ₄	L ⁵ ₄	I ⁹ ₈	J ⁹ ₈	K ⁹ ₈	L ⁹ ₈	I ¹³ ₁₂	J ¹³ ₁₂	K ¹³ ₁₂	L ¹³ ₁₂
4	K ⁰ ₀	L ⁰ ₀	K ² ₂	L ² ₂	K ⁴ ₄	L ⁴ ₄	K ⁶ ₆	L ⁶ ₆	K ⁸ ₈	L ⁸ ₈	K ¹⁰ ₁₀	L ¹⁰ ₁₀	K ¹² ₁₂	L ¹² ₁₂	K ¹⁴ ₁₄	L ¹⁴ ₁₄
2	I ³ ₀	J ³ ₀	K ³ ₀	L ³ ₀	M ³ ₀	N ³ ₀	O ³ ₀	P ³ ₀	I ¹¹ ₈	J ¹¹ ₈	K ¹¹ ₈	L ¹¹ ₈	M ¹¹ ₈	N ¹¹ ₈	O ¹¹ ₈	P ¹¹ ₈
4	M ⁰ ₀	N ⁰ ₀	M ² ₂	N ² ₂	M ⁴ ₄	N ⁴ ₄	M ⁶ ₆	N ⁶ ₆	M ⁸ ₈	N ⁸ ₈	M ¹⁰ ₁₀	N ¹⁰ ₁₀	M ¹² ₁₂	N ¹² ₁₂	M ¹⁴ ₁₄	N ¹⁴ ₁₄
3	M ¹ ₀	N ¹ ₀	O ¹ ₀	P ¹ ₀	M ⁵ ₄	N ⁵ ₄	O ⁵ ₄	P ⁵ ₄	M ⁹ ₈	N ⁹ ₈	O ⁹ ₈	P ⁹ ₈	M ¹³ ₁₂	N ¹³ ₁₂	O ¹³ ₁₂	P ¹³ ₁₂
4	O ⁰ ₀	P ⁰ ₀	O ² ₂	P ² ₂	O ⁴ ₄	P ⁴ ₄	O ⁶ ₆	P ⁶ ₆	O ⁸ ₈	P ⁸ ₈	O ¹⁰ ₁₀	P ¹⁰ ₁₀	O ¹² ₁₂	P ¹² ₁₂	O ¹⁴ ₁₄	P ¹⁴ ₁₄
S	A ¹⁵ ₀	B ¹⁵ ₀	C ¹⁵ ₀	D ¹⁵ ₀	E ¹⁵ ₀	F ¹⁵ ₀	G ¹⁵ ₀	H ¹⁵ ₀	I ¹⁵ ₀	J ¹⁵ ₀	K ¹⁵ ₀	L ¹⁵ ₀	M ¹⁵ ₀	N ¹⁵ ₀	O ¹⁵ ₀	P ¹⁵ ₀

Fig. 5. How the tree and total sum for lane A are embedded within the butterfly-patterned table.

Figure 5 is a copy of Figure 4, but highlights entries that logically belong to lane A; it is easy to see the sum (in the bottom row) as well as the entire binary search tree, indicated by the arrows. Similarly, Figure 6 is a copy of Figure 4 but highlights entries that logically belong to lane O. In the same manner, Figure 7 highlights entries that logically belong to lane F; the arrangement of the tree nodes is a bit more contorted, but the arrows show how they are organized. And indeed the first 15 rows of the table collectively contain a tree for each of the 16 lanes A through P, with the root of each tree appearing in the eighth line, and the last row of the table (labeled “S”) contains the overall sum for each of the 16 lanes.

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

Given this table, it is 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 ($0 \leq i < \log_2 W$), $2^{(\log_2 W) - i - 1}$ `__shfl_xor` instructions are executed 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.

A variation of this technique proved to be even faster in our experiments. It uses a modified butterfly computation: instead of replacing $\lceil \frac{a+b}{c} \rceil$ with $\lceil \frac{a}{a+b} \rceil \lceil \frac{b}{c+d} \rceil$, it replaces them with $\lceil \frac{a}{a+b} \rceil \lceil \frac{d}{c+d} \rceil$ (the only difference is providing d instead of c at the upper 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 above with two lines, to produce this code:

lane	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
4	A ₀ ⁰	B ₀ ⁰	A ₂ ²	B ₂ ²	A ₄ ⁴	B ₄ ⁴	A ₆ ⁶	B ₆ ⁶	A ₈ ⁸	B ₈ ⁸	A ₁₀ ¹⁰	B ₁₀ ¹⁰	A ₁₂ ¹²	B ₁₂ ¹²	A ₁₄ ¹⁴	B ₁₄ ¹⁴
3	A ₀ ¹	B ₀ ¹	C ₀ ¹	D ₀ ¹	A ₄ ⁵	B ₄ ⁵	C ₄ ⁵	D ₄ ⁵	A ₈ ⁹	B ₈ ⁹	C ₈ ⁹	D ₈ ⁹	A ₁₂ ¹³	B ₁₂ ¹³	C ₁₂ ¹³	D ₁₂ ¹³
4	C ₀ ⁰	D ₀ ⁰	C ₂ ²	D ₂ ²	C ₄ ⁴	D ₄ ⁴	C ₆ ⁶	D ₆ ⁶	C ₈ ⁸	D ₈ ⁸	C ₁₀ ¹⁰	D ₁₀ ¹⁰	C ₁₂ ¹²	D ₁₂ ¹²	C ₁₄ ¹⁴	D ₁₄ ¹⁴
2	A ₀ ³	B ₀ ³	C ₀ ³	D ₀ ³	E ₀ ³	F ₀ ³	G ₀ ³	H ₀ ³	A ₈ ¹¹	B ₈ ¹¹	C ₈ ¹¹	D ₈ ¹¹	E ₈ ¹¹	F ₈ ¹¹	G ₈ ¹¹	H ₈ ¹¹
4	E ₀ ⁰	F ₀ ⁰	E ₂ ²	F ₂ ²	E ₄ ⁴	F ₄ ⁴	E ₆ ⁶	F ₆ ⁶	E ₈ ⁸	F ₈ ⁸	E ₁₀ ¹⁰	F ₁₀ ¹⁰	E ₁₂ ¹²	F ₁₂ ¹²	E ₁₄ ¹⁴	F ₁₄ ¹⁴
3	E ₀ ¹	F ₀ ¹	G ₀ ¹	H ₀ ¹	E ₄ ⁵	F ₄ ⁵	G ₄ ⁵	H ₄ ⁵	E ₈ ⁹	F ₈ ⁹	G ₈ ⁹	H ₈ ⁹	E ₁₂ ¹³	F ₁₂ ¹³	G ₁₂ ¹³	H ₁₂ ¹³
4	G ₀ ⁰	H ₀ ⁰	G ₂ ²	H ₂ ²	G ₄ ⁴	H ₄ ⁴	G ₆ ⁶	H ₆ ⁶	G ₈ ⁸	H ₈ ⁸	G ₁₀ ¹⁰	H ₁₀ ¹⁰	G ₁₂ ¹²	H ₁₂ ¹²	G ₁₄ ¹⁴	H ₁₄ ¹⁴
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 ₀ ⁷	J ₀ ⁷	K ₀ ⁷	L ₀ ⁷	M ₀ ⁷	N ₀ ⁷	O ₀ ⁷	P ₀ ⁷
4	I ₀ ⁰	J ₀ ⁰	I ₂ ²	J ₂ ²	I ₄ ⁴	J ₄ ⁴	I ₆ ⁶	J ₆ ⁶	I ₈ ⁸	J ₈ ⁸	I ₁₀ ¹⁰	J ₁₀ ¹⁰	I ₁₂ ¹²	J ₁₂ ¹²	I ₁₄ ¹⁴	J ₁₄ ¹⁴
3	I ₀ ¹	J ₀ ¹	K ₀ ¹	L ₀ ¹	I ₄ ⁵	J ₄ ⁵	K ₄ ⁵	L ₄ ⁵	I ₈ ⁹	J ₈ ⁹	K ₈ ⁹	L ₈ ⁹	I ₁₂ ¹³	J ₁₂ ¹³	K ₁₂ ¹³	L ₁₂ ¹³
4	K ₀ ⁰	L ₀ ⁰	K ₂ ²	L ₂ ²	K ₄ ⁴	L ₄ ⁴	K ₆ ⁶	L ₆ ⁶	K ₈ ⁸	L ₈ ⁸	K ₁₀ ¹⁰	L ₁₀ ¹⁰	K ₁₂ ¹²	L ₁₂ ¹²	K ₁₄ ¹⁴	L ₁₄ ¹⁴
2	I ₀ ³	J ₀ ³	K ₀ ³	L ₀ ³	M ₀ ³	N ₀ ³	O ₀ ³	P ₀ ³	I ₈ ¹¹	J ₈ ¹¹	K ₈ ¹¹	L ₈ ¹¹	M ₈ ¹¹	N ₈ ¹¹	O ₈ ¹¹	P ₈ ¹¹
4	M ₀ ⁰	N ₀ ⁰	M ₂ ²	N ₂ ²	M ₄ ⁴	N ₄ ⁴	M ₆ ⁶	N ₆ ⁶	M ₈ ⁸	N ₈ ⁸	M ₁₀ ¹⁰	N ₁₀ ¹⁰	M ₁₂ ¹²	N ₁₂ ¹²	M ₁₄ ¹⁴	N ₁₄ ¹⁴
3	M ₀ ¹	N ₀ ¹	O ₀ ¹	P ₀ ¹	M ₄ ⁵	N ₄ ⁵	O ₄ ⁵	P ₄ ⁵	M ₈ ⁹	N ₈ ⁹	O ₈ ⁹	P ₈ ⁹	M ₁₂ ¹³	N ₁₂ ¹³	O ₁₂ ¹³	P ₁₂ ¹³
4	O ₀ ⁰	P ₀ ⁰	O ₂ ²	P ₂ ²	O ₄ ⁴	P ₄ ⁴	O ₆ ⁶	P ₆ ⁶	O ₈ ⁸	P ₈ ⁸	O ₁₀ ¹⁰	P ₁₀ ¹⁰	O ₁₂ ¹²	P ₁₂ ¹²	O ₁₄ ¹⁴	P ₁₄ ¹⁴
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

Fig. 6. How the tree and total sum for lane 0 are embedded within the butterfly-patterned table.

```

int r = threadIdx.x & 0x1f;    /* lane ID */
for (int b=1; 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] = a[d+(1<<b)];
        a[d+(1<<b)] = a[d] + v;
        p[d] = a[d];
    }
}
p[W-1] = a[W-1];

```

These 2 lines are different,
producing an alternate pattern.

Because the two loops are unrolled (the outer loop we actually unrolled 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 first 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 8), which in turn 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). This alternate strategy is the one we ultimately used to achieve best speed in our measurements, and is presented in more detail in Section 6.

lane	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	
4	A ⁰ ₀	B ⁰ ₀	A ² ₂	B ² ₂	A ⁴ ₄	B ⁴ ₄	A ⁶ ₆	B ⁶ ₆	A ⁸ ₈	B ⁸ ₈	A ¹⁰ ₁₀	B ¹⁰ ₁₀	A ¹² ₁₂	B ¹² ₁₂	A ¹⁴ ₁₄	B ¹⁴ ₁₄	
3	B ¹ ₀	C ¹ ₀	D ¹ ₀	A ³ ₄	B ³ ₄	C ³ ₄	D ³ ₄	A ⁵ ₄	B ⁵ ₄	C ⁵ ₄	D ⁵ ₄	A ⁷ ₈	B ⁷ ₈	C ⁷ ₈	D ⁷ ₈	A ⁹ ₁₂	B ⁹ ₁₂
4	C ⁰ ₀	D ⁰ ₀	E ⁰ ₀	F ⁰ ₀	G ⁰ ₀	H ⁰ ₀	I ⁰ ₀	J ⁰ ₀	K ⁰ ₀	L ⁰ ₀	M ⁰ ₀	N ⁰ ₀	O ⁰ ₀	P ⁰ ₀	A ¹¹ ₁₂	B ¹¹ ₁₂	
2	A ⁰ ₀	B ⁰ ₀	C ⁰ ₀	D ⁰ ₀	E ³ ₀	F ³ ₀	G ³ ₀	H ³ ₀	A ¹ ₈	B ¹ ₈	C ¹ ₈	D ¹ ₈	E ⁵ ₈	F ⁵ ₈	G ⁵ ₈	H ⁵ ₈	
4	F ⁰ ₀	E ² ₂	F ² ₂	E ⁴ ₄	F ⁴ ₄	E ⁶ ₆	F ⁶ ₆	E ⁸ ₈	F ⁸ ₈	E ¹⁰ ₁₀	F ¹⁰ ₁₀	E ¹² ₁₂	F ¹² ₁₂	E ¹⁴ ₁₄	F ¹⁴ ₁₄	E ¹⁵ ₁₅	
3	F ¹ ₀	G ¹ ₀	H ¹ ₀	I ¹ ₀	F ⁴ ₄	G ⁴ ₄	H ⁴ ₄	I ⁴ ₄	F ⁸ ₈	G ⁸ ₈	H ⁸ ₈	I ⁸ ₈	F ¹² ₁₂	G ¹² ₁₂	H ¹² ₁₂	I ¹² ₁₂	
4	G ⁰ ₀	H ⁰ ₀	I ⁰ ₀	J ⁰ ₀	G ² ₂	H ² ₂	I ² ₂	J ² ₂	G ⁴ ₄	H ⁴ ₄	I ⁴ ₄	J ⁴ ₄	G ⁶ ₆	H ⁶ ₆	I ⁶ ₆	J ⁶ ₆	
1	A ⁰ ₀	B ⁰ ₀	C ⁰ ₀	D ⁰ ₀	E ⁰ ₀	F ⁷ ₀	G ⁷ ₀	H ⁷ ₀	I ⁷ ₀	J ⁷ ₀	K ⁷ ₀	L ⁷ ₀	M ⁷ ₀	N ⁷ ₀	O ⁷ ₀	P ⁷ ₀	
4	I ⁰ ₀	J ⁰ ₀	I ² ₂	J ² ₂	I ⁴ ₄	J ⁴ ₄	I ⁶ ₆	J ⁶ ₆	I ⁸ ₈	J ⁸ ₈	I ¹⁰ ₁₀	J ¹⁰ ₁₀	I ¹² ₁₂	J ¹² ₁₂	I ¹⁴ ₁₄	J ¹⁴ ₁₄	
3	I ¹ ₀	J ¹ ₀	K ¹ ₀	L ¹ ₀	I ⁵ ₄	J ⁵ ₄	K ⁵ ₄	L ⁵ ₄	I ⁹ ₈	J ⁹ ₈	K ⁹ ₈	L ⁹ ₈	I ¹³ ₁₂	J ¹³ ₁₂	K ¹³ ₁₂	L ¹³ ₁₂	
4	K ⁰ ₀	L ⁰ ₀	K ² ₂	L ² ₂	K ⁴ ₄	L ⁴ ₄	K ⁶ ₆	L ⁶ ₆	K ⁸ ₈	L ⁸ ₈	K ¹⁰ ₁₀	L ¹⁰ ₁₀	K ¹² ₁₂	L ¹² ₁₂	K ¹⁴ ₁₄	L ¹⁴ ₁₄	
2	I ³ ₀	J ³ ₀	K ³ ₀	L ³ ₀	M ³ ₀	N ³ ₀	O ³ ₀	P ³ ₀	I ¹¹ ₈	J ¹¹ ₈	K ¹¹ ₈	L ¹¹ ₈	M ¹¹ ₈	N ¹¹ ₈	O ¹¹ ₈	P ¹¹ ₈	
4	M ⁰ ₀	N ⁰ ₀	M ² ₂	N ² ₂	M ⁴ ₄	N ⁴ ₄	M ⁶ ₆	N ⁶ ₆	M ⁸ ₈	N ⁸ ₈	M ¹⁰ ₁₀	N ¹⁰ ₁₀	M ¹² ₁₂	N ¹² ₁₂	M ¹⁴ ₁₄	N ¹⁴ ₁₄	
3	M ¹ ₀	N ¹ ₀	O ¹ ₀	P ¹ ₀	M ⁵ ₄	N ⁵ ₄	O ⁵ ₄	P ⁵ ₄	M ⁹ ₈	N ⁹ ₈	O ⁹ ₈	P ⁹ ₈	M ¹³ ₁₂	N ¹³ ₁₂	O ¹³ ₁₂	P ¹³ ₁₂	
4	O ⁰ ₀	P ⁰ ₀	O ² ₂	P ² ₂	O ⁴ ₄	P ⁴ ₄	O ⁶ ₆	P ⁶ ₆	O ⁸ ₈	P ⁸ ₈	O ¹⁰ ₁₀	P ¹⁰ ₁₀	O ¹² ₁₂	P ¹² ₁₂	O ¹⁴ ₁₄	P ¹⁴ ₁₄	
S	A ¹⁵ ₀	B ¹⁵ ₀	C ¹⁵ ₀	D ¹⁵ ₀	E ¹⁵ ₀	F ¹⁵ ₀	G ¹⁵ ₀	H ¹⁵ ₀	I ¹⁵ ₀	J ¹⁵ ₀	K ¹⁵ ₀	L ¹⁵ ₀	M ¹⁵ ₀	N ¹⁵ ₀	O ¹⁵ ₀	P ¹⁵ ₀	

Fig. 7. How the tree and total sum for lane F are embedded within the butterfly-patterned table.

The choice of whether to replace $\lceil \frac{a|b}{c|d} \rceil$ with $\lceil \frac{a}{a+b} | \frac{c}{c+d} \rceil$ or with $\lceil \frac{a}{a+b} | \frac{d}{c+d} \rceil$ (or possibly some other related pattern of values) depends on details of GPU architecture and implementation. It might well be that for another GPU architecture or another generation of NVIDIA GPU implementation, a different choice may be preferable for best speed. If a future architecture happens to provide a faster way to transpose a $W \times W$ matrix within the registers (perhaps by storing registers into a memory and then reloading them with a different access pattern, or by providing a mechanism to “index into the registers” so that each lane can access a different one of its own registers during a single SIMD instruction), then that should also be considered. In our experiments, storing and reloading proved to be much slower, and the GPU architecture we used did not support indexing into the registers, which would allow a $W \times W$ matrix such as a_{reg} (W elements in each of W lanes) to be transposed in place using just $W - 1$ shuffle operations (rather than the $3(W - 1)$ required by the butterfly pattern) and almost no computational overhead:

- 1: **for** k from 1 through $W - 1$ **do**
- 2: $a_{\text{reg}}[\text{myLaneId} \oplus k] := \text{__shfl_xor}(a_{\text{reg}}[\text{myLaneId} \oplus k], k)$
- 3: **end for**

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 (one topic to each document) 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.

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 ₁ ⁰	C ₂ ²	D ₃ ³	A ₄ ⁵	B ₅ ⁴	C ₆ ⁷	D ₇ ⁶	A ₈ ⁹	B ₉ ⁸	C ₁₀ ¹¹	D ₁₁ ¹⁰	A ₁₂ ¹³	B ₁₃ ¹²	C ₁₄ ¹⁵	D ₁₅ ¹⁴
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 ₁ ⁰	C ₂ ³	D ₃ ⁰	E ₄ ⁷	F ₅ ⁴	G ₆ ⁷	H ₇ ⁴	A ₈ ¹¹	B ₉ ⁸	C ₁₀ ¹¹	D ₁₁ ⁸	E ₁₂ ¹⁵	F ₁₃ ¹²	G ₁₄ ¹⁵	H ₁₅ ¹²
4	E ₀ ¹	F ₁ ²	E ₂ ³	F ₃ ⁴	E ₄ ⁵	F ₅ ⁶	E ₆ ⁷	F ₇ ⁸	E ₈ ⁹	F ₉ ¹⁰	E ₁₀ ¹¹	F ₁₁ ¹²	E ₁₂ ¹³	F ₁₃ ¹⁴	E ₁₄ ¹⁵	F ₁₅ ¹⁶
3	E ₀ ¹	F ₁ ⁰	G ₂ ³	H ₃ ²	E ₄ ⁵	F ₅ ⁴	G ₆ ⁷	H ₇ ⁶	E ₈ ⁹	F ₉ ⁸	G ₁₀ ¹¹	H ₁₁ ¹⁰	E ₁₂ ¹³	F ₁₃ ¹²	G ₁₄ ¹⁵	H ₁₅ ¹⁴
4	G ₀ ¹	H ₁ ²	G ₂ ³	H ₃ ⁴	G ₄ ⁵	H ₅ ⁶	G ₆ ⁷	H ₇ ⁸	G ₈ ⁹	H ₉ ¹⁰	G ₁₀ ¹¹	H ₁₁ ¹²	G ₁₂ ¹³	H ₁₃ ¹⁴	G ₁₄ ¹⁵	H ₁₅ ¹⁶
1	A ₀ ⁷	B ₁ ⁰	C ₂ ⁷	D ₃ ⁰	E ₄ ⁷	F ₅ ⁰	G ₆ ⁷	H ₇ ⁰	I ₈ ¹⁵	J ₉ ⁸	K ₁₀ ¹⁵	L ₁₁ ⁸	M ₁₂ ¹⁵	N ₁₃ ⁸	O ₁₄ ¹⁵	P ₁₅ ⁸
4	I ₀ ¹	J ₁ ²	I ₂ ³	J ₃ ⁴	I ₄ ⁵	J ₅ ⁶	I ₆ ⁷	J ₇ ⁸	I ₈ ⁹	J ₉ ¹⁰	I ₁₀ ¹¹	J ₁₁ ¹²	I ₁₂ ¹³	J ₁₃ ¹⁴	I ₁₄ ¹⁵	J ₁₅ ¹⁶
3	I ₀ ¹	J ₁ ⁰	K ₂ ³	L ₃ ²	I ₄ ⁵	J ₅ ⁴	K ₆ ⁷	L ₇ ⁶	I ₈ ⁹	J ₉ ⁸	K ₁₀ ¹¹	L ₁₁ ¹⁰	I ₁₂ ¹³	J ₁₃ ¹²	K ₁₄ ¹⁵	L ₁₅ ¹⁴
4	K ₀ ¹	L ₁ ²	K ₂ ³	L ₃ ⁴	K ₄ ⁵	L ₅ ⁶	K ₆ ⁷	L ₇ ⁸	K ₈ ⁹	L ₉ ¹⁰	K ₁₀ ¹¹	L ₁₁ ¹²	K ₁₂ ¹³	L ₁₃ ¹⁴	K ₁₄ ¹⁵	L ₁₅ ¹⁶
2	I ₀ ³	J ₁ ⁰	K ₂ ³	L ₃ ⁰	M ₄ ⁷	N ₅ ⁴	O ₆ ⁷	P ₇ ⁴	I ₈ ¹¹	J ₉ ⁸	K ₁₀ ¹¹	L ₁₁ ⁸	M ₁₂ ¹⁵	N ₁₃ ¹²	O ₁₄ ¹⁵	P ₁₅ ¹²
4	M ₀ ¹	N ₁ ²	M ₂ ³	N ₃ ⁴	M ₄ ⁵	N ₅ ⁶	M ₆ ⁷	N ₇ ⁸	M ₈ ⁹	N ₉ ¹⁰	M ₁₀ ¹¹	N ₁₁ ¹²	M ₁₂ ¹³	N ₁₃ ¹⁴	M ₁₄ ¹⁵	N ₁₅ ¹⁶
3	M ₀ ¹	N ₁ ⁰	O ₂ ³	P ₃ ²	M ₄ ⁵	N ₅ ⁴	O ₆ ⁷	P ₇ ⁶	M ₈ ⁹	N ₉ ⁸	O ₁₀ ¹¹	P ₁₁ ¹⁰	M ₁₂ ¹³	N ₁₃ ¹²	O ₁₄ ¹⁵	P ₁₅ ¹⁴
4	O ₀ ¹	P ₁ ²	O ₂ ³	P ₃ ⁴	O ₄ ⁵	P ₅ ⁶	O ₆ ⁷	P ₇ ⁸	O ₈ ⁹	P ₉ ¹⁰	O ₁₀ ¹¹	P ₁₁ ¹²	O ₁₂ ¹³	P ₁₃ ¹⁴	O ₁₄ ¹⁵	P ₁₅ ¹⁶
S	A ₀ ¹⁵	B ₁ ⁰	C ₂ ¹⁵	D ₃ ⁰	E ₄ ¹⁵	F ₅ ⁰	G ₆ ¹⁵	H ₇ ⁰	I ₈ ¹⁵	J ₉ ⁰	K ₁₀ ¹⁵	L ₁₁ ⁰	M ₁₂ ¹⁵	N ₁₃ ⁰	O ₁₄ ¹⁵	P ₁₅ ⁰

Fig. 8. Alternate “add-subtract” butterfly pattern for p' .

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 non-negative 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. Similarly, 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 to take advantage of memory cache structure.

(For presentation purposes, we likewise present array and matrices in transposed form in this article; in every diagram depicting a two-dimensional array, the axis indexed by K runs vertically.)

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 $0 \leq m < M$, $w[m]$ 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 $0 \leq m < M$ and for all i such that $0 \leq i < N[m]$, do two things: First, for all k such that $0 \leq k < K$, 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 $\sigma = \sum_{0 \leq k < K} a[m][i][k]$. Thus, $a[m][i][k']$ is a relative (unnormalized) probability, and $a[m][i][k']/\sigma$ is an absolute (normalized) probability.

ALGORITHM 1: Drawing new z values

```

1: procedure DRAWZ( $N[M], \theta[M, K], \phi[V, K], w[M][N]$ ; output  $z[M, N]$ )
2:   local array  $a[M][N][K], p[M][N][K]$ 
3:   for all  $0 \leq m < M$  do
4:     for all  $0 \leq i < N[m]$  do
5:        $\triangleright$  Compute  $\theta$ - $\phi$  products
6:       for all  $0 \leq k < K$  do
7:          $a[m][i][k] := \theta[m, k] \times \phi[w[m][i], k]$ 
8:       end for
9:        $\triangleright$  Compute partial sums of the products
10:      let  $sum = 0.0$ 
11:      for  $k$  from 0 through  $K - 1$  do
12:         $sum += a[m][i][k]$ 
13:       $p[m][i][k] := sum$ 
14:      end for
15:      let  $j = 0$ 
16:       $\langle$ search the table  $p[m][i]$  of partial sums $\rangle$ 
17:       $z[m, i] := j$ 
18:    end for
19:  end for
20: end procedure

```

ALGORITHM 2: Simple linear search

```

1: code chunk  $\langle$ search the table  $P$  of partial sums $\rangle$ :
2:   let  $u =$  value chosen uniformly at random (or pseudorandomly) from  $[0.0, 1.0)$ 
3:   let  $u' = sum \times u$ 
4:   while  $j < K - 1$  and  $u' \geq P[j]$  do  $j += 1$ 
5: end

```

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 (initially undefined) element value for each indexable position in the array; and that distinct iterations of any containing “**for**” or “**for all**” construct create distinct and independent instantiations of such local variables. The iterations of “**for** ... from ... through ...” are executed sequentially; 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 6–8 of Algorithm 1) is straightforward. The computation of partial sums (lines 10–14) is sequential; the variable sum accumulates the 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 $0 \leq k < K$, 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

ALGORITHM 3: Simple binary search

```

1: code chunk ⟨search the table  $P$  of partial sums⟩:
2:   let  $u$  = value chosen uniformly at random (or pseudorandomly) from  $[0.0, 1.0)$ 
3:   let  $u' = \text{sum} \times u$ 
4:   let  $k = K - 1$ 
5:   while  $j < k$  do
6:     let  $\text{mid} = \left\lfloor \frac{j+k}{2} \right\rfloor$ 
7:     if  $u' < P[\text{mid}]$  then  $k := \text{mid}$ 
8:     else  $j := \text{mid} + 1$ 
9:   end while
10: end

```

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 now 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 $W = 16$ and $K = 71$. 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 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 $\theta \cdot \phi$ 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 $K \geq W$. 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 .

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

ALGORITHM 4: Drawing z values (transposed access)

```

1: procedure DRAWZ( $N[M], \theta[M, K], \phi[V, K], w[M][N]$ ; output  $z[M, N]$ )
2:   local array  $p_{\text{local}}[M][K], a_{\text{local}}[M][W]$ 
3:   for all  $0 \leq q < M/W$  do
4:     local array  $c_{\text{warp}}[W, W]$ 
5:     for SIMD  $0 \leq r < W$  do
6:       let  $m = q \times W + r$ 
7:       local array  $\theta_{\text{local}}[K]$ 
8:        $\langle \text{cache } \theta \text{ values into } \theta_{\text{local}} \rangle$ 
9:       let  $i_{\text{master}} = 0$ 
10:      while  $\text{any}(i_{\text{master}} < N[m])$  do
11:        let  $i = \min(i_{\text{master}}, N[m] - 1)$ 
12:         $\langle \text{compute partial sums of } \theta\text{-}\phi \text{ products} \rangle$ 
13:        let  $j = 0$ 
14:         $\langle \text{search the table } p_{\text{local}}[m] \text{ of partial sums} \rangle$ 
15:         $z[m, i] := j$ 
16:         $i_{\text{master}} += 1$ 
17:      end while
18:    end for
19:  end for
20: end procedure

```

ALGORITHM 5: Caching θ values (transposed access)

```

1: code chunk  $\langle \text{cache } \theta \text{ values into } \theta_{\text{local}} \rangle$ :
2:   let  $j = 0$ 
3:   while  $j < (K \bmod W)$  do  $\triangleright$  Cache the remnant
4:      $\theta_{\text{local}}[j] := j1$ 
5:      $\theta[m, j] += j1$ 
6:   end while
7:   while  $j < K$  do  $\triangleright$  Cache all  $W \times W$  blocks
8:     for  $k$  from 0 through  $W - 1$  do
9:        $\triangleright$  Next line uses transposed access to  $\theta$ 
10:       $\theta_{\text{local}}[j + k] := \theta[q \times W + k, j + r]$ 
11:    end for
12:     $j += W$ 
13:  end while
14: end

```

(with $i = N[m] - 1$) multiple times, which does not 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 $W = 16$ and $K = 71$, topics 0, 1, 2, 3, 4, 5, and 6 form a remnant of length 7; topics 3–18 form a first block of length 16; topics 19–34 form a second block; topics 35–50 form a third

ALGORITHM 6: Compute partial sums (transposed access)

```

1: code chunk (compute partial sums of  $\theta$ - $\phi$  products):
2:   let  $c = w[m][i]$ 
3:   for all  $0 \leq k < W$  do
4:      $c_{\text{warp}}[k, r] := c$  ▷ Transposed access to  $c_{\text{warp}}$ 
5:   end for
6:   let  $sum = 0.0$ 
7:   let  $j = 0$ 
8:   while  $j < (K \bmod W)$  do ▷ Process the remnant
9:      $sum += (\theta_{\text{local}}[j] \times \phi[c, j])$ 
10:     $p_{\text{local}}[m][j] := sum$ 
11:     $j += 1$ 
12:  end while
13:  while  $j < K$  do ▷ Process all  $W \times W$  blocks
14:    for  $k$  from 0 through  $W - 1$  do
15:      ▷ Next line uses transposed access to  $\phi$ 
16:       $a_{\text{local}}[m, k] := \theta_{\text{local}}[j + k] \times \phi[c_{\text{warp}}[r, k], j + r]$ 
17:    end for
18:    for  $k$  from 0 through  $W - 1$  do
19:      ▷ Next line uses transposed access to  $a_{\text{local}}$ , alas
20:       $sum += a_{\text{local}}[q \times W + k, r]$ 
21:       $p_{\text{local}}[m, j + k] := sum$ 
22:    end for
23:     $j += W$ 
24:  end while
25: end

```

block; and topics 51–66 form a fourth block (see Figure 9). This organization of arrays into 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 3–6 handles the remnant, and then the following **while** loop processes successive blocks. On line 10 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 8–11 (which overall is designed to fetch one $W \times W$ block of θ values) instead of every thread in the warp fetching its k th 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 i th word of its document and calls it c ; in lines 3–5 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 8–12 computes θ - ϕ products and partial sums p in the usual way (remember that the remnant

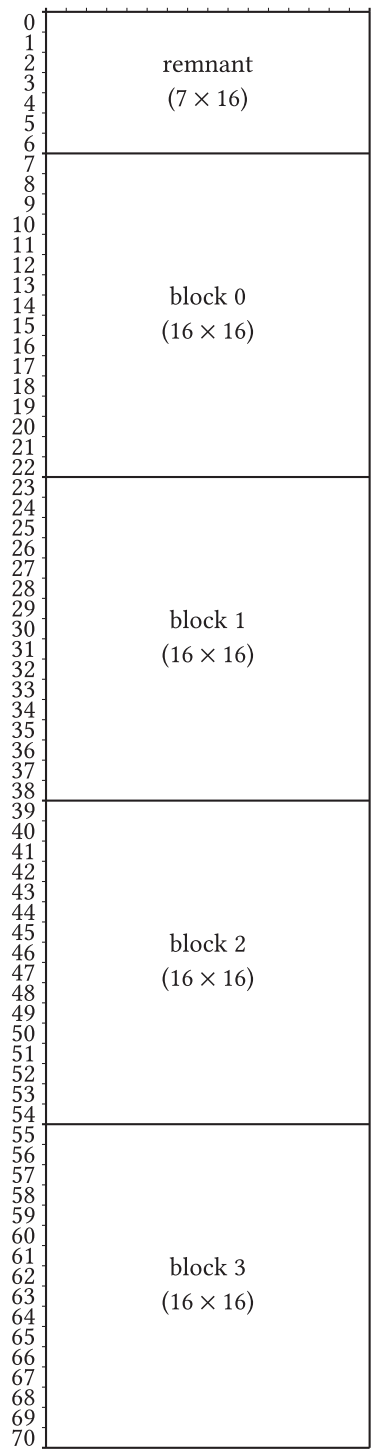


Fig. 9. Example division of an array into a remnant and four blocks ($W = 16, K = 71$).

ALGORITHM 7: Drawing new z values using a butterfly table

```

1: procedure DRAWZ( $N[M], \theta[M, K], \phi[V, K], w[M][N]$ ; output  $z[M, N]$ )
2:    $\triangleright W$  (the “warp size”) must be a power of 2, and  $M$  must be a multiple of  $W$ .
3:   for all  $0 \leq q < M/W$  do
4:     for SIMD  $0 \leq r < W$  do
5:       let  $m = q \times W + r$ 
6:       local array  $p'[K], \theta_{\text{local}}[K]$ 
7:       register array  $a_{\text{reg}}[W], c_{\text{warp}}[W]$ 
8:        $\langle \text{cache } \theta \text{ values into } \theta_{\text{local}} \rangle$ 
9:       let  $i_{\text{master}} = 0$ 
10:      while  $\text{any}(i_{\text{master}} < N[m])$  do
11:        let  $i = \min(i_{\text{master}}, N[m] - 1)$ 
12:         $\langle \text{SIMD compute butterfly partial sums} \rangle$ 
13:        let  $j = 0$ 
14:         $\langle \text{SIMD search butterfly partial sums} \rangle$ 
15:         $z[m, i] := j$ 
16:         $i_{\text{master}} += 1$ 
17:      end while
18:    end for
19:  end for
20: end procedure


```

in θ_{local} is not transposed), but the loop in lines 14–17 processes a block to compute product values to store into the a_{local} array; the access to ϕ on line 16 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 20, we finally pay the piper: 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 20; 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 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 18–30, which is implemented by the alternate (faster) version of the CUDA code shown in Section 3, that is, the one that has each butterfly  replace $\left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix} \right]$ with $\left[\begin{smallmatrix} a & d \\ a+b & c+d \end{smallmatrix} \right]$.

Within a butterfly-patterned block of partial sums, Algorithm 9 performs a binary search as follows. The u' value is 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 $K \geq W$, then some $W \times W$ 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.

ALGORITHM 8: Compute a butterfly-patterned table of sums

```

1: code chunk (SIMD compute butterfly partial sums):
2:   let  $c = w[m][i]$ 
3:   for all  $0 \leq k < W$  do
4:      $c_{\text{warp}}[k] := \_shfl(c, k)$ 
5:   end for
6:   let  $sum = 0.0$ 
7:   let  $j = 0$ 
8:   while  $j < (K \bmod W)$  do ▷ Process the remnant
9:      $sum += (\theta_{\text{local}}[j] \times \phi[c, j])$ 
10:     $p'[j] := sum$ 
11:     $j := j + 1$ 
12:   end while
13:   while  $j < K$  do ▷ Process all  $W \times W$  blocks
14:     for  $k$  from 0 through  $W - 1$  do
15:       ▷ Next line uses transposed access to  $\phi$ 
16:        $a_{\text{reg}}[k] := \theta_{\text{local}}[j + k] \times \phi[c_{\text{warp}}[k], j + r]$ 
17:     end for
18:     for  $b$  from 0 through  $(\log_2 W) - 1$  do
19:       let  $bit = 2^b$ 
20:       for  $i$  from 0 through  $\frac{W}{2 \times bit} - 1$  do
21:         let  $d = 2 \times bit \times i + (bit - 1)$ 
22:         let  $h = (\text{if } (m \ \& \ bit) \neq 0$ 
23:           then  $a_{\text{reg}}[d]$ 
24:           else  $a_{\text{reg}}[d + bit]$ )
25:         let  $v = \_shfl\_xor(h, bit)$ 
26:         if  $(r \ \& \ bit) \neq 0$  then  $a_{\text{reg}}[d] := a_{\text{reg}}[d + bit]$ 
27:          $a_{\text{reg}}[d + bit] := a_{\text{reg}}[d] + v$ 
28:          $p'[j + d] := a_{\text{reg}}[d]$ 
29:       end for
30:     end for
31:      $sum += a_{\text{reg}}[W - 1]$ 
32:      $p'[W - 1] := sum$ 
33:      $j := j + W$ 
34:   end while
35: end

```

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 $highValue = m_0^{blockBase+k}$. 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 $mid = \lfloor \frac{j+k}{2} \rfloor$; 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

ALGORITHM 9: Searching within a butterfly-patterned table

```

1: code chunk  $\langle \text{SIMD search butterfly partial sums} \rangle$ :
2:   let  $u = \text{value chosen uniformly at random (or pseudorandomly) from } [0.0, 1.0)$ 
3:   let  $u' = \text{sum} \times u$ 
4:   let  $j = 0$ 
5:   let  $k = \lfloor \frac{K}{W} \rfloor - 1$ 
6:   let  $\text{searchBase} = (K \bmod W) + (W - 1)$ 
7:    $\triangleright$  Binary search to find correct block of size  $W$ 
8:   while  $j < k$  do
9:     let  $\text{mid} = \lfloor \frac{j+k}{2} \rfloor$ 
10:    if  $u' < p'[\text{mid} \times W + \text{searchBase}]$  then  $k := \text{mid}$ 
11:    else  $j := \text{mid} + 1$ 
12:  end while
13:  let  $\text{blockBase} = (K \bmod W) + j \times W$ 
14:  if  $K \geq W$  then
15:     $\langle \text{SIMD butterfly search one block} \rangle$ 
16:  end if
17:  if  $\text{blockBase} > 0$  then
18:    if  $u' < p'[m, \text{blockBase} - 1]$  then
19:       $\triangleright$  Not in a block after all, so search the remnant
20:      for  $i$  from 0 through  $(K \bmod W) - 1$  do
21:        if  $u' < p'[i]$  then
22:           $j := i$ 
23:          break
24:        end if
25:      end for
26:    end if
27:  end if
28: end

```

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*.

The threads in a warp assist one another in fetching tree nodes using the loop in lines 12–18; the function `__shfl_xor` effects this data transfer in line 16.

7 EVALUATION

We coded four versions of a complete LDA topic-modeling algorithm using Gibbs sampling (which is a specific kind of Markov chain Monte Carlo algorithm, or MCMC) in CUDA 6.5 for an NVIDIA Titan Black GPU ($W = 32$). The LDA model has two sets of parameters, θ and ϕ , as we have already indicated; moreover, the model is designed in such a way that integrating these parameters has a closed form. Integrating a parameter introduces a tradeoff between convergence rate and the amount of parallelism of the MCMC algorithm. The four versions we consider are therefore the traditional Gibbs sampler for which both θ and ϕ are integrated out, a version in which θ is integrated but ϕ is not, a version in which ϕ is integrated but θ is not, and a version in which neither is integrated.

For each of the four versions, 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 [28]; that paper includes only a passing mention of this use of butterfly-patterned partial sums, and refers to an early version of this article [24].) All eight

ALGORITHM 10: Butterfly search within one $W \times W$ block

```

1: code chunk (SIMD butterfly search one block):
2:   let  $lowValue = (\text{if } blockBase > 0$ 
3:     then  $p'[blockBase - 1]$ 
4:     else 0)
5:   let  $highValue = p'[blockBase + (W - 1)]$ 
6:   let  $flip = 0$ 
7:    $\triangleright$  Butterfly search within the block of size  $W$ 
8:   for  $b$  from 0 through  $(\log_2 W) - 1$  do
9:     let  $bit = 2^{((\log_2 W) - 1) - b}$ 
10:    let  $mask = ((W - 1) \times (2 \times bit)) \& (W - 1)$ 
11:    let  $y = 0$ 
12:    for  $i$  from 0 through  $\frac{W}{2 \times bit} - 1$  do
13:      let  $d = (bit - 1) + 2 \times bit \times i$ 
14:      let  $him = (d \& mask) + (r \& \neg mask)$ 
15:      let  $hisBlockBase = \_shfl(blockBase, him)$ 
16:      let  $t = \_shfl\_xor(p'[hisBlockBase + d], flip)$ 
17:      if  $((r \oplus d) \& mask) = 0$  then  $y := t$ 
18:    end for
19:    let  $compareValue = (\text{if } (r \& bit) \neq 0$ 
20:      then  $highValue - y$ 
21:      else  $lowValue + y$ )
22:    if  $stop < compareValue$  then
23:       $highValue := compareValue$ 
24:       $flip := flip \oplus (bit \& r)$ 
25:    else
26:       $lowValue := compareValue$ 
27:       $flip := flip \oplus (bit \& \neg r)$ 
28:    end if
29:  end for
30:   $j := blockBase + (flip \oplus r)$ 
31: end

```

variants were tested for speed using a Wikipedia-based dataset with number of documents $M = 43,556$, vocabulary size $V = 37,286$, total number of words in corpus $\Sigma N = 3,072,662$ (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 18, and the CUDA compiler then automatically unrolled the loops that start on lines 14 and 20. The performance results are shown in Figure 10. 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.

Subsequently, we performed a more extensive set of measurements, after upgrading the CUDA software. Using CUDA 7.5 on the same hardware, we measured Algorithm 1, Algorithm 7, and a variant of Algorithm 4 that transposes in registers (as in Figure 2) for 32-bit intermediate values (Figure 11(a)) and for 64-bit intermediate values (Figure 11(b)), using the same Wikipedia-based dataset. Each of the three algorithms was measured for $K = 4, 8, 12, 16, 20, \dots, 1020, 1024$, again in each case performing 100 sampling iterations and measuring the execution time of the entire

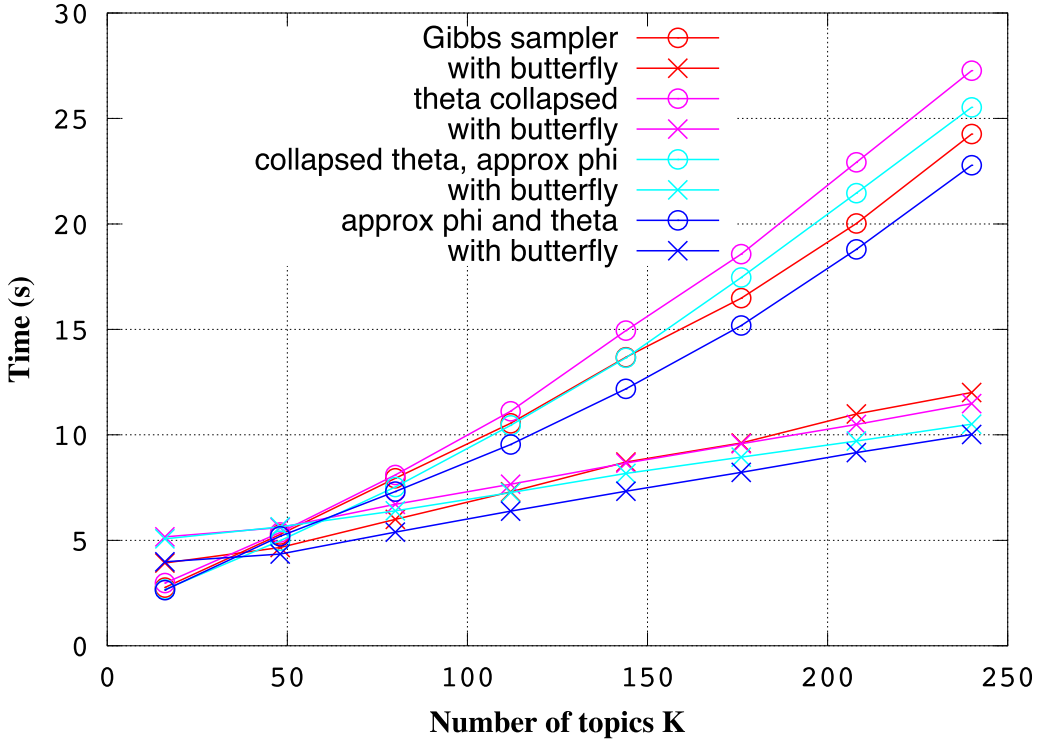


Fig. 10. Measurements of execution time, with and without butterfly-patterned partial sums, for four variations of a machine learning application (topic modeling by Gibbs sampling; CUDA 6.5; $K = 32k + 16$, $0 \leq k \leq 7$).

application, not just the part that draws z values. Each data point shown in Figure 11 (and in Figure 12) 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 binary representation of the length of the remnant (that is, the value of $K \bmod 32$).

For 32-bit intermediate data (Figure 11(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 11(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 12(a) and 12(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.

Comparing Figures 11(a) and 11(b), it may seem at first glance that the “butterfly partial sums” technique provides a substantial improvement over the “register transpose” technique in the case of 64-bit data, but hardly any improvement in the case of 32-bit data. Why? Closer inspection reveals that actually the “register transpose” technique is substantially *worse* than might be expected

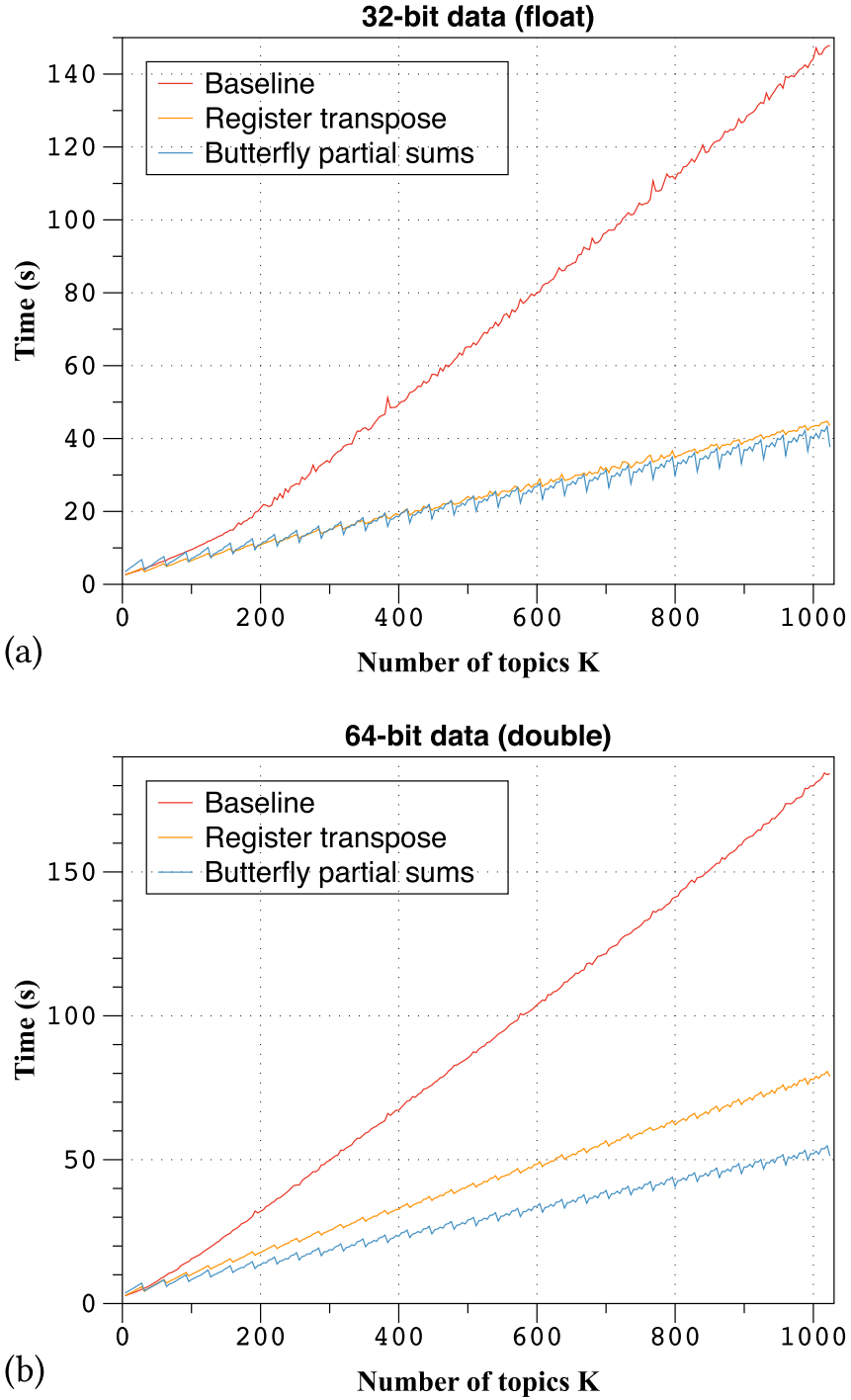


Fig. 11. Two sets of measurements of execution time for a complete machine learning application (topic modeling by Gibbs sampling; CUDA 7.5; $K = 4k + 4, 0 \leq k \leq 255$).

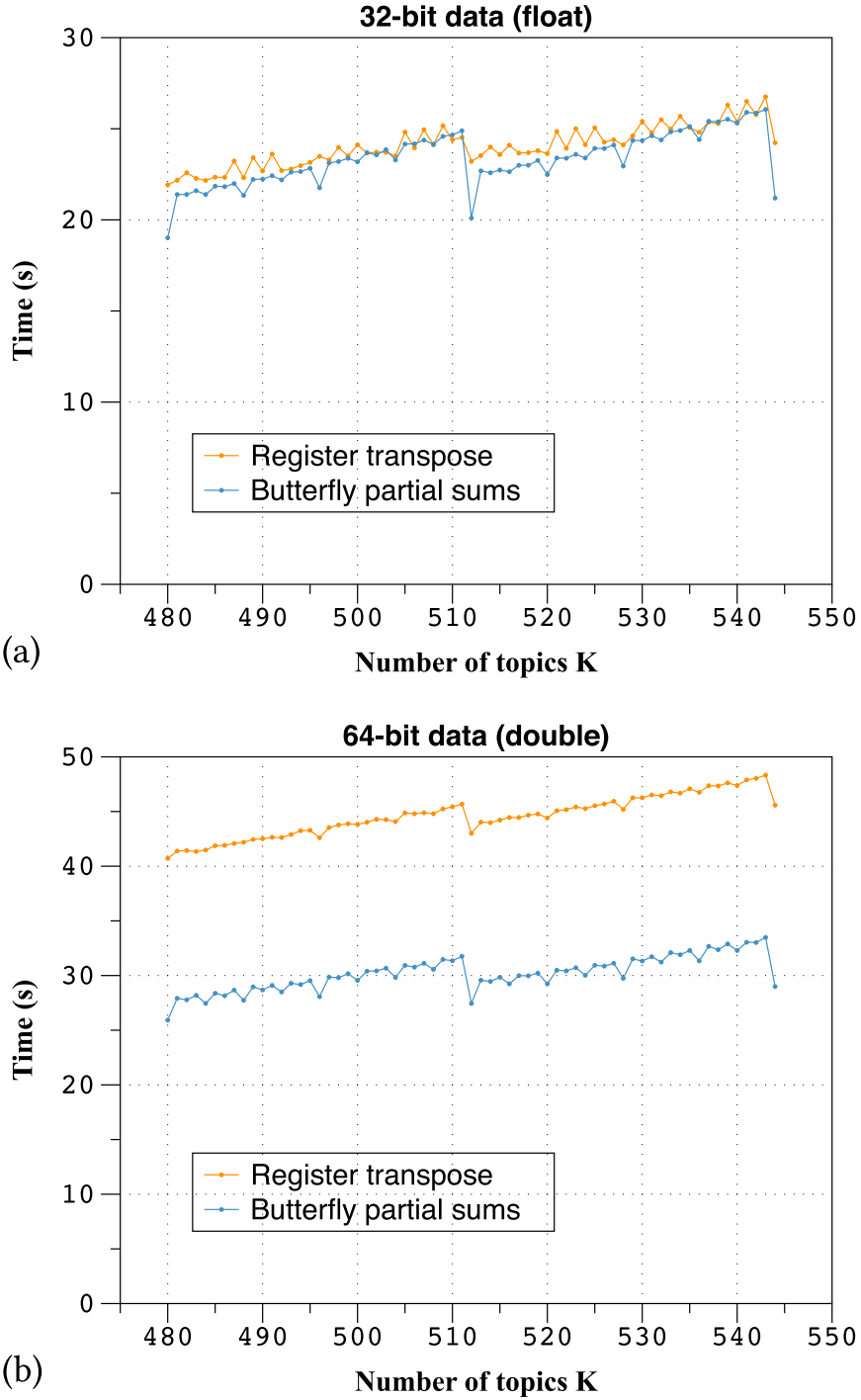


Fig. 12. Two additional (zoomed-in) sets of measurements of execution time for a complete machine learning application (topic modeling by Gibbs sampling; $480 \leq K \leq 544$).

in the case 64-bit data. In each of Figures 11(a) and 11(b), the “butterfly partial sums” runs (Algorithm 7) show an improvement over the baseline (Algorithm 1) of somewhat over 70% (74% for 32-bit data, 71% for 64-bit data, at $K = 1024$), so this improvement is consistent. However, for the “register transpose” runs (Algorithm 4) we see an improvement of just about 70% in the case of 32-bit data but of only about 56% in the case of 64-bit data, and this is a discrepancy that warrants explanation. We believe that the problem lies with the unfavorable transposed accesses that occur in line 20 of Algorithm 6. For the most part these accesses cannot be effectively coalesced. Now, computational instructions are overlapped with memory accesses; we conjecture that in the 32-bit cases these memory accesses take only slightly longer than the computational work, but the 64-bit case likely touches about twice as many lines of memory as the 32-bit case, and these additional memory accesses take much longer than the computational work.

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 [36]) 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 [30, 31] 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 $\{0, 1, 2, \dots, n-1\}$ and let u be chosen uniformly at random from the real interval $[0, 1)$. Then the value drawn is (if $u < F_k$ then k 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 [31] 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 [29] 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, 34], 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 article 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 article, 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 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. 2015. Nested Chinese restaurant franchise processes: Applications to user tracking and document modeling. In *Proceedings of the 30th International Conference on Machine Learning (ICML '13)*. Microtome Publishing, Brookline, MA, 1426–1434. Retrieved from <http://www.jmlr.org/proceedings/papers/v28/ahmed13.pdf>.
- [2] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, New York.
- [3] David M. Blei. 2012. Probabilistic topic models. *Commun. ACM* 55, 4 (Apr. 2012), 77–84. DOI: <https://doi.org/10.1145/2133806.2133826>
- [4] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet allocation. *J. Mach. Learn. Res.* 3 (Mar. 2003), 993–1022. Retrieved from <http://dl.acm.org/citation.cfm?id=944919.944937>.
- [5] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. 2008. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS '08)*. ACM, New York, 205–213. DOI: <https://doi.org/10.1145/1375527.1375559>
- [6] Peter M. Fenwick. 1994. A new data structure for cumulative frequency tables. *Software Pract. Exper.* 24, 3 (1994), 327–336. DOI: <https://doi.org/10.1002/spe.4380240306>
- [7] Thomas L. Griffiths and Mark Steyvers. 2004. Finding scientific topics. *Proc. Natl. Acad. Sci. U.S.A.* 101, suppl. 1 (2004), 5228–5235. DOI: <https://doi.org/10.1073/pnas.0307752101>
- [8] Diane Hu, Rob Hall, and Josh Attenberg. 2014. Style in the long tail: Discovering unique interests with latent variable models in large scale social E-commerce. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14)*. ACM, New York, 1640–1649. DOI: <https://doi.org/10.1145/2623330.2623338>
- [9] D. A. Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proc. IRE* 40, 9 (Sept. 1952), 1098–1101. DOI: <https://doi.org/10.1109/JRPROC.1952.273898>
- [10] S. Lennart Johnsson, Tim Harris, and Kapil K. Mathur. 1989. Matrix multiplication on the connection machine. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*. ACM, New York, NY, 326–332. <http://doi.acm.org/10.1145/76263.76298>
- [11] Joon Hee Kim, Amin Mantrach, Alejandro Jaimes, and Alice Oh. 2016. How to compete online for news audience: Modeling words that attract clicks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, 1645–1654. DOI: <https://doi.org/10.1145/2939672.2939873>
- [12] Donald E. Knuth. 1998. *Seminumerical Algorithms* (3rd edition). The Art of Computer Programming, Vol. 2. Addison-Wesley, Reading, MA.
- [13] Donald E. Knuth. 1998. *Sorting and Searching* (2nd edition). The Art of Computer Programming, Vol. 3. Addison-Wesley, Reading, MA.
- [14] Anthony Lee, Christopher Yau, Michael B. Giles, Arnaud Doucet, and Christopher C. Holmes. 2010. On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods. *J. Comput. Graph. Stat.* 19, 4 (2010), 769–789. <http://arxiv.org/pdf/0905.2441.pdf>.
- [15] Aaron Q. Li, Amr Ahmed, Sujith Ravi, and Alexander J. Smola. 2014. Reducing the sampling complexity of topic models. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14)*. ACM, New York, 891–900. DOI: <https://doi.org/10.1145/2623330.2623756>
- [16] Mian Lu, Ge Bai, Qiong Luo, Jie Tang, and Jiuxin Zhao. 2013. Accelerating topic model training on a single machine. In *Web Technologies and Applications (APWeb 2013)*, Yoshiharu Ishikawa, Jianzhong Li, Wei Wang, Rui Zhang, and Wenjie Zhang (Eds.). Lecture Notes in Computer Science, Vol. 7808. Springer, Berlin, 184–195. DOI: https://doi.org/10.1007/978-3-642-37401-2_20
- [17] Sepideh Maleki, Annie Yang, and Martin Burtscher. 2016. Higher-order and tuple-based massively-parallel prefix sums. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, 539–552. DOI: <https://doi.org/10.1145/2908080.2908089>
- [18] G. Marsaglia. 1963. Generating discrete random variables in a computer. *Commun. ACM* 6, 1 (Jan. 1963), 37–38. DOI: <https://doi.org/10.1145/366193.366228>
- [19] Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. 1993. Dynamic generation of discrete random variates. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 361–370. Retrieved from <http://dl.acm.org/citation.cfm?id=313559.313807>.

- [20] NVIDIA. 2015. Developer Zone website: CUDA Toolkit documentation: CUDA Toolkit v6.5 Programming Guide, section B.14. Warp shuffle functions. Retrieved from <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-shuffle-functions>.
- [21] Daniel Ramage, Susan Dumais, and Dan Liebling. 2010. Characterizing microblogs with topic models. In *Proceedings of the 4th International AAAI Conference on Weblogs and Social Media*. Association for the Advancement of Artificial Intelligence, Palo Alto, CA, 130–137.
- [22] Guy L. Steele Jr. 2016. Using Butterfly-Patterned Partial Sums to Draw from Discrete Distributions. *GTC website*. Retrieved from <http://on-demand.gputechconf.com/gtc/2016/video/s6665-guy-steele-fast-splittable.mp4>.
- [23] Guy L. Steele Jr. 2016. Using butterfly-patterned partial sums to draw from discrete distributions. In *NVIDIA GPU Technology Conference*. Retrieved from <http://on-demand.gputechconf.com/gtc/2016/presentation/s6666-guy-steele-butterfly-pattern.pdf>. Slides for talk S6665. Video available at Reference [22].
- [24] Guy L. Steele Jr. and Jean-Baptiste Tristan. 2015. Using butterfly-patterned partial sums to optimize GPU memory accesses for drawing from discrete distributions. *CoRR (Computing Research Repository at arXiv.org)* (May 2015). Retrieved from <http://arxiv.org/abs/1505.03851>.
- [25] Guy L. Steele Jr. and Jean-Baptiste Tristan. 2017. Using butterfly-patterned partial sums to draw from discrete distributions. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'17)*. ACM, New York, 341–355. DOI: <https://doi.org/10.1145/3018743.3018757> An early version of this paper is Reference [24].
- [26] Marc A. Suchard, Quanli Wang, Cliburn Chan, Jacob Frelinger, Andrew Cron, and Mike West. 2010. Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures. *J. Comput. Graphic. Stat.* 19, 2 (2010), 419–438.
- [27] Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C. Pocock, Stephen Green, and Guy L. Steele Jr. 2014. Augur: Data-parallel probabilistic modeling. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, 2600–2608. Retrieved from <http://papers.nips.cc/book/year-2014>.
- [28] Jean-Baptiste Tristan, Joseph Tassarotti, and Guy L. Steele Jr. 2015. Efficient training of LDA on a GPU by mean-for-mode estimation. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*. Microtome Publishing, Brookline, MA, 59–68. Retrieved from <http://jmlr.org/proceedings/papers/v37/tristan15.pdf>.
- [29] M. D. Vose. 1991. A linear algorithm for generating random numbers with a given distribution. *IEEE Trans. Software Engineer.* 17, 9 (Sept. 1991), 972–975. DOI: <https://doi.org/10.1109/32.92917>
- [30] A. J. Walker. 1974. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electron. Lett.* 10, 8 (Apr. 1974), 127–128. DOI: <https://doi.org/10.1049/el:19740097>
- [31] Alastair J. Walker. 1977. An efficient method for generating discrete random variables with general distributions. *ACM Trans. Math. Software* 3, 3 (Sept. 1977), 253–256. DOI: <https://doi.org/10.1145/355744.355749>
- [32] Nicholas Wilt. 2013. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley, Upper Saddle River, NJ.
- [33] Feng Yan, Ningyi Xu, and Yuan Qi. 2009. Parallel inference for latent Dirichlet allocation on graphics processing units. In *Advances in Neural Information Processing Systems 22*. Curran Associates, 2134–2142. Retrieved from <http://papers.nips.cc/book/year-2009>.
- [34] Shengen Yan, Guoping Long, and Yunquan Zhang. 2013. StreamScan: Fast scan algorithms for GPUs without global barrier synchronization. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, 229–238. DOI: <https://doi.org/10.1145/2442516.2442539>
- [35] Huasha Zhao, Biye Jiang, and John Canny. 2014. SAME but different: Fast and high-quality Gibbs parameter estimation. *CoRR (Computing Research Repository at arXiv.org)* (Sept. 2014). Retrieved from <http://arxiv.org/abs/1409.5402>.
- [36] Seth Zimmerman. 1959. An optimal search procedure. *Amer. Math. Monthly* 66, 8 (Oct. 1959), 690–693.

Received August 2018; revised March 2019; accepted May 2019