Accelerating Overlapping Community Detection: Performance Tuning a Stochastic Gradient Markov Chain Monte Carlo Algorithm

Ismail El-Helw(✉), Rutger Hofman, and Henri E. Bal

Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
ielhelw@gmail.com, {rutger,bal}@cs.vu.nl

Abstract. Building efficient algorithms for data-intensive problems requires deep analysis of data access patterns. Random data access patterns exacerbate this process. In this paper, we discuss accelerating a randomized data-intensive machine learning algorithm using multi-core CPUs and several types of GPUs. A thorough analysis of the algorithm's data dependencies enabled a 75% reduction in its memory footprint. We created custom compute kernels via code generation to identify the optimal set of data placement and computational optimizations per compute device. An empirical evaluation shows up to 245x speedup compared to an optimized sequential version. Another result from this evaluation is that achieving peak performance does not always match intuition: e.g., depending on the GPU architecture, vectorization may increase or hamper performance.

Keywords: Algorithms for accelerators and heterogeneous systems · Performance analysis · Combinatorial and data intensive application

1 Introduction

The past decade has witnessed a tremendous increase in the applicability and usefulness of artificial intelligence in our daily lives. Much of this activity was fueled by the mainstream adoption of machine learning approaches as they provide tools to solve inherently difficult problems. However, many of these techniques require a massive amount of computation which severely limits the scale of problems that can be tackled.

In this paper, we present the methodology followed in designing and implementing a high-performance version of an existing Stochastic Gradient Markov Chain Monte Carlo (SG-MCMC) machine learning algorithm that detects overlapping communities in graphs. The algorithm analyzes pair-wise interactions between entities in order to discover hidden attributes. For instance, consider a social network represented as an undirected graph where the vertices represent individuals and edges represent relations between them. Given the relation

© Springer Nature Switzerland AG 2020
M. Malawski and K. Rzadca (Eds.): Euro-Par 2020, LNCS 12247, pp. 510–526, 2020.
https://doi.org/10.1007/978-3-030-57675-2_32

information, the algorithm can identify latent groups of individuals that represent shared interests. This problem structure differs from graph partitioning or clustering as there is a many-to-many relationship between individuals and interests. For example, each individual can have multiple interests. Simultaneously, each interest group can span multiple individuals. Formally, this problem domain is known as Mixed-Membership Stochastic Blockmodels (MMSB). The theory behind the algorithm is discussed in more detail in [10].

The focus of this work is on the computational efficiency and parallel performance of the SG-MCMC algorithm. More specifically, we discuss the process of accelerating the algorithm by developing aggressive optimizations targeting multi-core CPUs and GPUs. The parallel algorithm achieves speedup factors up to 245, compared to a well-tuned sequential C++ program which itself is a factor 1000–1500 faster than the Python/Numpy program developed by the algorithm's original authors. From a computational point of view, this algorithm differs from widespread machine learning algorithms in several ways. First, it is highly data-intensive which makes GPU acceleration particularly challenging. Second, owing to the algorithm's stochastic nature, the majority of its memory access patterns and data dependencies are non-deterministic. As a result, straightforward optimization attempts of the memory access patterns either fail or lead to non-intuitive results.

Through careful analysis of the computation and data structures we show that the algorithm's full state can be reduced by roughly 75%. Compressing the state significantly reduces the data intensity and allows for tackling larger problems while maintaining all state in memory. Further, by cataloguing and accounting for the various load and store operations, we identified the highest priority locations of data reuse. In order to navigate the unclear optimization landscape, we developed an effective kernel code generation mechanism that explores all permutations of the available optimization opportunities. These optimizations include caching in shared memory, caching in the register file, loop unrolling and explicit vectorization.

In summary, the contributions of this work are:

- decrease the algorithm's data intensity by eliminating 75% of its memory footprint;
- tune the algorithm's performance by maximizing data reuse and identify the fastest combination of optimizations through dynamic kernel code generation;
- perform a comparative performance analysis of the accelerated algorithm versions on a multi-core CPU and a number of GPUs, highlighting the particular optimization combinations that were successful per device;
- achieve speedup factors of 21 and 245 over an optimized sequential program using a multi-core CPU and a GPU respectively.

The remainder of this paper is organized as follows. A description of the sequential version of the algorithm and its data structures is provided in Sect. 2. Sect. 3 discusses the design of the parallel algorithm. Section 4 provides an empirical evaluation of the contributions of this work. Sect. 5 presents an overview of related works. Finally, Sect. 6 concludes.

2 SG-MCMC Algorithm Overview

In this section we describe the computational aspects of the SG-MCMC MMSB algorithm. Moreover, we will introduce the data structures and notation that will be used throughout this paper. A detailed explanation of the algorithm is provided in [4,10].

The network graph G consists of the undirected edges $\mathcal{E}$ and has N vertices. The algorithm starts by partitioning G into the *training set*, the *validation set* $\mathcal{E}_h$ and the test set (the latter is not used in our implementations). $\mathcal{E}_h$ and the test set are much smaller than G, typically between 1% and 10% of the edges in G. The number of communities K is specified as a model parameter to the algorithm.

The algorithm progresses by iteratively improving the global state of the learning problem, using the training set. There are two pairs of data structures that hold the global state. θ, a $K{\times}2$ matrix, is used for calculating the community strength β, i.e. the probability that two members in a community share an edge. β is a vector of length K; it is the normalized version of $\theta_{k,2}$. The matrix π of dimensions $N{\times}K$ represents the probability for each vertex in G to be a member of each community. It is the normalized equivalent of the matrix ϕ of dimensions $N{\times}K$, on which the calculations occur. The definitions of β and π are:

$$\beta_k = \frac{\theta_{k,2}}{\theta_k^{sum}} \text{ where } \theta_k^{sum} = \sum_{j=1}^{2} \theta_{k,j} \tag{1}$$

$$\pi_{i,k} = \frac{\phi_{i,k}}{\phi_i^{sum}} \text{ where } \phi_i^{sum} = \sum_{j=1}^{K} \phi_{i,j} \tag{2}$$

The symbols used throughout this paper are given in Table 1.

Table 1. Definition of most important symbols

Symbol	Type	Size	Description
G			Graph
$\mathcal{K}$		K	Set of communities
$\mathcal{V}$	{vertex}	N	Vertices in G
$\mathcal{E}$	{edge}		Linked edges in G
E	{edge}		$\mathcal{V} \times \mathcal{V}$: linked and nonlinked edges
$\mathcal{E}_h$	{edge}		Held-out subset of the graph
$\mathcal{E}_n$	{edge}		Sampled mini-batch of edges in E
m			Number of vertices in $\mathcal{E}_n$
$\mathcal{V}_n$	{vertex}		Sampled neighbor set for a vertex in $\mathcal{E}_n$
θ	float vector 2-D	$K \times 2$	Reparameterization of β. $\beta_k = \theta_{k,2} / \sum_j \theta_{k,j}$
β	float vector	K	Community strength
ϕ	float vector 2-D	$N \times K$	Reparameterization of π. $\pi_{i,k} = \phi_{i,k} / \sum_j \phi_{i,j}$
π	float vector 2-D	$N \times K$	$\pi_{i,k}$ is probability that vertex i is in community k

Algorithm 1. Sequential version of SG-MCMC for a-MMSB

```
1: while sampling do
2:     sample a mini-batch ℰ_n
3:     // ℰ_n may be a subset of G, or disjoint with G
4:     for each vertex i in ℰ_n do
5:         sample n random 'neighbors' (not from G)
6:         for each edge (i, neighbor) do
7:             calculate its contribution to the gradient in φ
8:         update φ_i using stepsize s
9:     for each vertex i in ℰ_n do
10:        update π_i according to changed φ_i
11:    for the edges in ℰ_n do
12:        calculate gradients in θ and update θ
13:    for the edges in ℰ_n do
14:        update β accordingly
15:    every so many iterations: // metric is perplexity
16:    verify the quality of π and β against the validation set
```

Pseudo-code for the algorithm is presented in Algorithm 1 which is based on the description in [4,10].

An iteration in the algorithm consists of 6 compute stages. We will highlight the data accessed in the stages as this determines the opportunities for parallelism.

The first stage (line 2 in the Algorithm) randomly draws a mini-batch $\mathcal{E}_n$, using the "stratified random node" strategy [10]. In this strategy, a coin toss is used to decide between two sample types. The first sample type chooses one random vertex i and selects all of its edges to constitute the mini-batch $\mathcal{E}_n$. This sample type is referred to as *link edges*. The second sample type randomly draws a vertex i and generates random edges of the form $(i,\, j)$ such that the edges are not in G. This sample type is referred to as *nonlink edges*. The set of vertices that constitute the edges of the mini-batch $\mathcal{E}_n$ is denoted m.

In the second stage (line 5), for each vertex i in m, a neighbor set $\mathcal{V}_n$ of size n is randomly sampled with edges of the form (i, j).

Stage 3, *update_phi* (line 6–10), calculates a gradient vector $\nabla\phi_i$ for each vertex i in the mini-batch, by iterating over the edges (i, j) in i's neighbor set; the data that is used is π_i, π_j, and β. The gradient $\nabla\phi_i$ is used to update ϕ_i. Stage 4 (line 9–10), *update_pi*, updates π_i so it remains the normalized version of ϕ_i.

Stage 5, *update_theta* (line 11–12), uses β and π_a, π_b for the edges (a, b) in the minibatch $\mathcal{E}_n$ to calculate a gradient vector $\nabla\theta$. θ is updated using $\nabla\theta$. Stage 6, *update_beta* (line 13–14), recalculates β as the normalized version of $\theta_{k,2}$.

At regular intervals, the algorithm's global state is assessed by evaluating the perplexity over the edges in the validation set $\mathcal{E}_h$. Perplexity is a metric that represents the quality of the algorithm solution at a given point in time. It is used to detect the algorithm's convergence. The perplexity, as elaborated in [4,10], is

the exponential of the average over time of the negative log-likelihood of meeting a link edge. In this *calculate_perplexity* stage (lines 15–16), β is used, as are π_a and π_b for each edge (a, b) in $\mathcal{E}_h$.

The graph G is queried for membership in the stages *update_phi*, *update_beta*, and *calculate_perplexity*. The validation set $\mathcal{E}_h$ is traversed in the *calculate_perplexity* stage.

3 System Design and Implementation

The process of designing an accelerated version of the SG-MCMC MMSB algorithm involved multiple transformations. First, we describe how an efficient C++ baseline was created. Further, changes to the algorithm and the data structures were carried out, for both efficient resource utilization and parallelization. This section provides an overview of the system's evolution in incremental phases, identifying the key contributions and differences between consecutive states.

3.1 An Efficient Sequential Baseline Version

The original implementation was done in Python, as is common in the machine-learning community. It relied on Numpy [16] to perform numerical computations efficiently. However, the algorithm made heavy use of Python sets and dictionaries which have no Numpy equivalent. We ported the Python code to C++, maintaining the same program structure. This transformation yielded a speedup factor of 171.

The next step was to remove a number of inefficiencies. E.g., one recurring idiom in the Python implementation was a choice expression of the form $a^y b^{1-y}$ where y is either 0 or 1. We transformed such expressions into conditional expressions which compute either a or b, and avoid floating-point exponentiation. Other optimizations were loop strength reduction and common subexpression lifting. These optimizations yielded another speedup factor of 6.

Finally, we investigated the performance effect of reducing the floating-point precision from 64-bit to 32-bit. This reduces both the computation and data intensity leading to a lower memory footprint which frees registers and enables more effective data reuse. It has been previously shown that stochastic learning algorithms do not require high precision in the presence of statistical approximations and the addition of random noise [7]. This reduction increased the speedup by a factor of 1.5.

In conclusion, porting from Python to efficient 64-bit C++ gave a speedup factor of $\sim$ 1000, and reducing the precision to 32-bit increased that to a factor of $\sim$ 1500. We use the resulting sequential C++ version as baseline for our performance comparisons.

3.2 Restructuring for Parallelism

The design of an accelerated version of the algorithm necessitated several crucial modifications to allow for efficient parallelization. We chose OpenCL [17] as it

provides a common abstraction for a variety of compute devices which fulfilled our requirements of employing CPUs or GPUs. However, Nvidia's OpenCL SDK limits the total memory allocations within a context to 4GB which severely limits the problem sizes that can be tackled on GPUs. Therefore, we migrated the system to use the abstraction layer CLCudaAPI [15] to support both OpenCL and CUDA [14] as back-ends. We considered using OpenMP for the CPU implementation but decided against it. OpenCL offers a common platform for CPU and GPU, and hence makes performance comparison straightforward, and OpenCL allows control of the multicore vector units.

This section discusses the key contributions to attain two optimized parallel versions, catering for multi-core CPUs and GPUs. First, we present structural changes that provide compute device specific optimizations. Next, we provide an in-depth description of the optimized CPU and GPU versions respectively.

Fast Lookup of Graph Edges

The algorithm relies on a set data structure to store the edges of the graph. This set is queried frequently with randomly generated edges to check for their membership. To improve the performance of such lookups, we developed a custom set implementation that restricts its features based on its usage patterns. For example, the set is used as a container for the graph edges which are known in advance. Therefore, the set can be made immutable and does not require thread-safety.

We designed the edge set as a variant of a cuckoo hash [18]. The hash is indexed by a tuple of two 32-bit vertices that represent an edge. It uses two hash functions to address two corresponding storage spaces. Additionally, it stores 4 different 64-bit edge values per bucket as shown in Fig. 1. This set implementation allowed us to obtain a loading factor upwards of 90% which reduces the space overhead significantly.

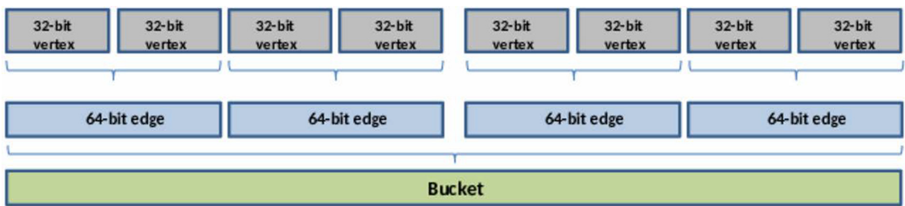

Fig. 1. Structure of a single cuckoo hash bucket containing 4 edges.

Parallelization and Data Dependencies

The original algorithm was structurally reorganized into 4 main sections with one or more kernels in each depending on data dependencies and synchronization requirements. First, the sampling of a mini-batch of edges is done on the host as it is a cheap operation. The mini-batch sampling is followed by the neighbor sampling kernel which generates uniformly random neighbors for each vertex in

the mini-batch. Second, the predominant kernel, *update_phi* is invoked to calculate the gradients for each vertex i in the mini-batch $\mathcal{E}_n$ and updates the value of $\phi_i \mid i \in \mathcal{E}_n$. Third, the *update_pi* kernel is invoked to normalize the individual ϕ_i and store the result in the corresponding π_i. Fourth, the kernels *update_theta* and *update_beta* are invoked to modify the global parameters. Finally, a dedicated kernel calculates perplexity.

Memory Footprint Reduction

A key structural change applied to the algorithm is the lossless compression of its state. This enabled the algorithm to process larger data sets while maintaining all state in memory. Moreover, as the algorithm is data-intensive, a reduction of the state is accompanied by a decrease in its data-intensity. The data structures that occupy most memory are the matrices π and ϕ of dimensions $N \times K$. However, the storage of both matrices is redundant as π is a row-normalized copy of ϕ, see Sect. 2. The storage for ϕ is discarded; ϕ_{ik} values are recalculated each time as π_{ik}/ϕ_i^{sum}, which requires maintenance of a vector ϕ^{sum} of size N. Moreover, the calculation for ϕ_{ik} can be cached. The matrix ϕ is required in two kernels only, namely, *update_phi* and *update_pi*. Both kernels access ϕ_i only for vertices i in the mini-batch $\mathcal{E}_n$, so for each iteration, the calculated values for ϕ are cached in a smaller temporary matrix of size $|\mathcal{E}_n| \times K$. This transformation trades memory storage and bandwidth for a minimal computation overhead.

Thus, the memory requirement for ϕ is reduced from $N \times K$ to a vector ϕ_{sum} of length N and a much smaller $|\mathcal{E}_n| \times K$ matrix. For sufficiently large K, this transformation roughly halves the memory footprint of the algorithm.

CPU-Specific Optimization

The multi-core CPU version of the algorithm uses OpenCL to perform its computations. The work decomposition of the CPU kernels ensures that each thread performs independent computation to avoid expensive synchronization. Edge-centric kernels that operate over mini-batch edges perform computations over every edge in parallel while vertex-centric kernels exploit parallelism across the selected vertices. Additionally, the kernels were vectorized to decrease instruction overhead and utilize the SSE capabilities of the CPU cores.

GPU-Specific Optimization

The GPU implementation builds on top of the CPU work decomposition scheme. However, instead of having every thread perform independent computations, each block of threads shares the work associated with the single edge or vertex for edge-centric and vertex-centric kernels.

Similar to the CPU optimization, we investigated the use of vector data types to decrease the instruction overhead and increase memory bandwidth for all strategies.

Since all kernels are data-bound, we investigated several memory organization strategies to exploit the GPU's memory hierarchy. As a case study, a discussion of the *update_phi* kernel is provided as it is the predominant component of the algorithm.

The *update_phi* kernel operates over every vertex i in the mini-batch and requires two temporary vectors of length K to perform its computation. For each vertex, it iterates over the randomly generated neighbors and computes a vector of probabilities $Probs_i$ of length K. The individual $Probs_i$ vectors of each (i, neighbor) tuple are used to update the gradients vector $Grads_i$ for each vertex i. Finally, the ϕ_i row is updated to reflect the changes that were accumulated in $Grads_i$ for each vertex i in the mini-batch $\mathcal{E}_n$.

A deeper analysis of the memory access patterns of the *update_phi* kernel revealed the frequency and modality of access to the data structures. The read-only accesses of π_j of the randomly generated neighbors are unique with a high probability. More precisely, each $vertex_i$ in the mini-batch randomly samples a neighbor set from the uniform distribution. Given that the total number of sampled neighbors is much smaller than N, there is a low likelihood of duplicate samples, so there is only limited potential for data reuse. Therefore, these accesses provide limited opportunities for optimization without interfering with the algorithm's entropy. The data structure usage patterns that are deterministic and most frequently accessed in read/write mode are *Probs* and *Grads*. Similarly, π_i for each vertex in the mini-batch is read repeatedly for the calculation of *Probs* per neighbor and again to update *Grads*.

The following strategies present alternative methods of handling the deterministic memory usage patterns of $Probs_i$, $Grads_i$ and π_i.

The naive strategy simply allocates temporary vectors in thread local memory for *Probs* and *Grads* which physically resides in device memory. Memory accesses are coalesced to achieve the highest possible device memory bandwidth.

The shared memory strategy allocates the temporary vectors $Probs_i$ and $Grads_i$ in shared memory. Furthermore, it copies the π_i of the selected mini-batch vertex to shared memory to avoid repeated reads of device memory.

The code generation strategy dynamically generates the code of the kernel to custom tailor its properties. It controls whether a vector is placed or cached in shared memory. Additionally, it controls which vectors explicitly reside in registers by allocating space on the stack frame, unrolling all inner loops of the kernel and substituting all vector addressing with static values. The code generation strategy allows this flexibility for the vectors of concern, namely, $Probs_i$, $Grads_i$ and π_i. Hence, this strategy allows for 8 possible configurations denoted by three letters each of which is a choice between *Register* (R) or *Shared* (S). For example, SSR denotes that $Probs_i$, $Grads_i$ and π_i were placed in *Shared*, *Shared* and *Register* respectively.

3.3 Kernel Code Generation

To support our various configuration needs, we implemented a code generator as part of the host program. It receives the model and performance tuning parameters, and produces compute kernels honoring the supplied constraints. The generated code is then compiled on the fly using the CLCudaAPI before driving the different phases of the algorithm's execution.

The code generator uses a template that defines the static structure of the kernels. Further, it employs custom syntax of placeholders that determine where dynamic content will be inserted. The code generator supports 2 forms of template substitution: type definitions and loop unrolling.

Type definitions are used to control SIMD vector widths for both CPU and GPU kernels. For example, 'Floatn' can be replaced with 'float4' when using 4-wide CUDA SIMD. Further, macros and inline functions override the standard arithmetic operations for each vector width. For example, 'ADDn(x, y)' will be replaced with 'ADD4(x, y)'. It is important to note that this substitution method influences loop lengths. For example, using a 4-wide vector instead of simple instructions reduces loop iterations by a factor of 4.

In the case of GPU kernels, type definitions also specify whether a buffer is allocated in shared or global memory.

Loop unrolling is used to force variables to be stored in registers. The kernel's static template contains placeholders that specify the type of a statement to be performed. The code generator looks for these placeholders and replaces them with one or more statements in an independent activation record.

4 Evaluation

This section discusses the performance evaluation of the various optimizations for resource utilization and parallelization from Sect. 3. First, we explore the performance benefits of parallelizing the computations on a multi-core CPU using OpenCL. Then we assess the trade-offs associated with the GPU optimization strategies and their performance effects on different types of GPUs, spanning four chip architecture generations.

All experiments were conducted on the VU Amsterdam DAS5 cluster [1]. The cluster consists of 68 compute nodes each equipped with a dual 8-core Intel Xeon E5-2630v3 CPU clocked at 2.40 GHz, 64 GB of memory and 8 TB of storage. Additionally, the cluster is fitted with a number of Nvidia GPUs including RTX 2080 Ti, GTX TitanX, GTX980, K40c and K20m; see Table 3 for an overview of their properties. The network graph used for evaluation of the algorithm's performance is *com-DBLP* from the SNAP collection [9]. It has 317 K vertices and a million edges. The focus of our paper is the effect of parallelizing the algorithm; our findings are representative for any dataset since the behavior of the calculation kernels does not depend on the dataset.

4.1 Analysis of CPU Parallelism

This section discusses the use of the multi-core CPU available on the DAS5 cluster. The parallel OpenCL version divides the work across the cores and performs independent calculations concurrently. As shown in Table 2, the dominant kernel in the computation is *update_phi*, which accounts for 66.5% of the computation time. Without exploiting the dual 8-core processor's vectorization capabilities, the speedup relative to the baseline sequential C++ version is 9.8. The model parameters for these experiments: $\mathrm{K} = 1024$, $m = 4096$, $|\mathcal{V}_n| = 32$.

In addition to applying computations in parallel, we investigated the use of the SIMD instructions to maximize resource utilization. Figure 2 presents the performance of vectorizing the kernels with varying vector widths. A key aspect in this figure is the diminishing performance benefit for higher vector widths. As the computational performance increases, the memory throughput becomes the leading performance bottleneck. Moreover, using 16-wide SIMD gave a slight performance penalty compared to 8-wide SIMD. The 8-wide vector version improves the speedup relative to the baseline version from 9.8 to 20.9.

Table 2. Multi-core CPU performance breakdown without vectorization.

Kernel	Time (seconds)
PPX CALC	0.0364737
PPX ACCUM	0.083
SAMPLING	0.535599
UPDATE_PHI	25.6598
UPDATE_PI	0.645875
THETA SUM	0.0483902
GRADS PAR	1.92919
GRADS SUM	9.31122
UPDATE THETA	0.0548013
NORM THETA	0.001
TOTAL	38.5858

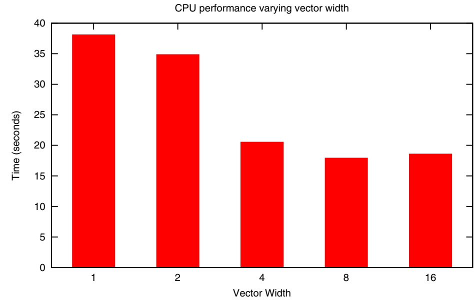

Fig. 2. Performance of CPU for varying vector widths.

4.2 Analysis of GPU Parallelism

As discussed in Sect. 3.2, we employed multiple memory organization strategies: NAIVE, SHARED and 8 variations of code generation. This section investigates the effectiveness of each on the available GPUs.

Comparison of Memory Management Strategies. Figure 3(a) presents the performance of the RTX2080 Ti GPU with an explicit vector width of 4 across the strategies. The x-axis represents *update_phi* thread block sizes while the y-axis presents the total execution time of 1000 *update_phi* invocations. The naive and shared strategies are labeled NAIVE and SHARED respectively. Further, each code generation strategy is labeled by GEN followed by the 3 choices that identify it. A zoomed-in version of Fig. 3(a) is provided as the bottom row to focus on the optimal range.

As would be expected, the naive strategy exhibits the worst performance over all thread block sizes, as it does not explicitly cache repeated device read operations.

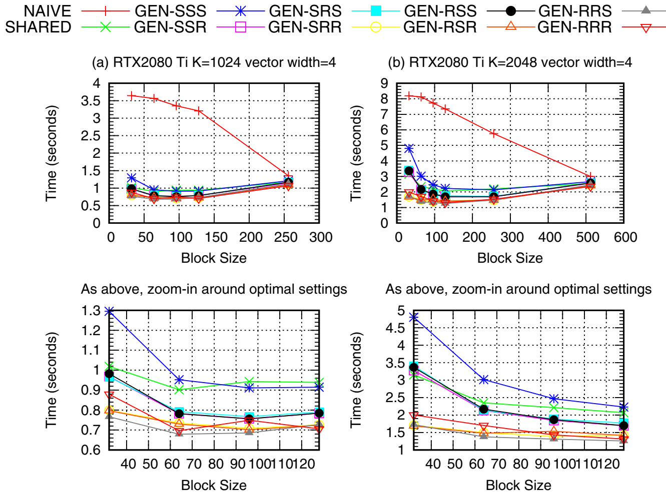

Fig. 3. Execution time of 1K *update_phi* invocations ($\mathcal{E}_n = 4096, |\mathcal{V}_n| = 32$) on the RTX2080 Ti across a sweep of *update_phi* thread block sizes. The lower figures are a zoom-in into the optimal block range of the figures above.

The SHARED and *GEN-SSS* strategies come next in terms of performance. Both strategies cache $Grads_i$, $Probs_i$ and π_i in shared memory but differ in one aspect: *GEN-SSS* explicitly unrolls the internal loops of the kernel. However, loop unrolling incurred additional overhead and made *GEN-SSS* slower than the simple SHARED strategy.

The other flavors of the code generation strategy attain higher performance as they unroll internal loops as well as cache data in registers. The RTX2080 Ti in Fig. 3(a) obtains the best performance with the *GEN-RRS* strategy. The optimal thread block size is 64 and vector width is 4. The results for other vector widths are omitted as they obtain lower performance.

A key model parameter that affects the behavior of the optimization strategies is the number of communities K. Figure 3(b) presents the same model configuration as in Fig. 3(a) but $K = 2048$ instead of $K = 1024$. The most important difference in performance between the two figures is the optimal thread block size, which grows from 64 to 128 when K is doubled. An increase in K comes with a proportional increase in the size of shared memory required by each thread block for the strategies that employ it explicitly. Similarly, GEN strategies that use the register file will require additional space. Therefore, the number of concurrent thread blocks that can execute on a single streaming multiprocessor will decrease, minimizing the GPU's occupancy and utilization. This limitation can be counteracted by selecting a larger thread block size which in turn increases the computation concurrency and occupancy. However, increasing the block sizes has diminishing returns and eventually leads to worse performance that matches the NAIVE strategy.

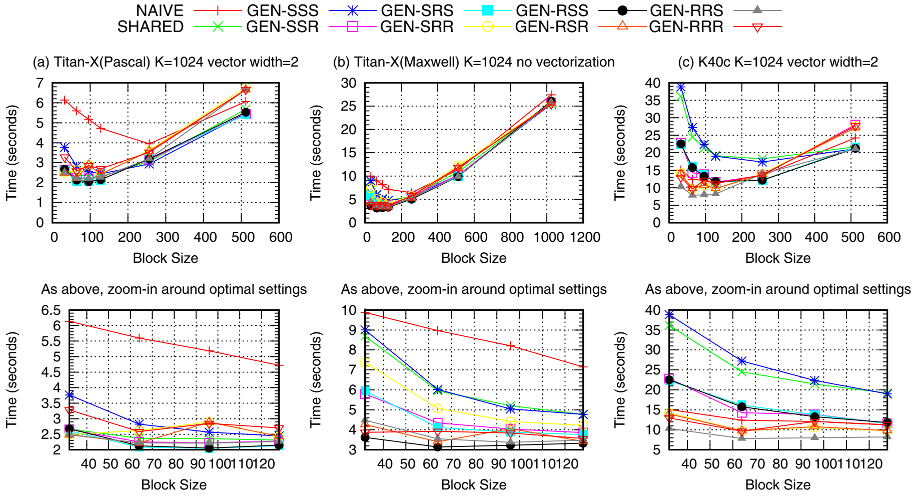

Fig. 4. Execution time of 1000 *update_phi* invocations on a Titan-X (Pascal and Maxwell) and a K40 across a sweep of *update_phi* thread block sizes. The lower figures are a zoom-in into the optimal block range of the upper figures. Other relevant model parameters: $\mathcal{E}_n = 4096$, $|\mathcal{V}_n| = 32$.

In contrast to the other GPUs, the Titan-X Pascal shows good performance with the SHARED strategy as shown in Fig. 4(a). This can be explained by its high bandwidth to computational power ratio compared to the other devices. The Pascal performs best with the *GEN-RSR* strategy, a block size of 96 and a vector width of 2. On the other hand, Fig. 4(b) shows the Titan-X Maxwell performed best with a thread block size of 64 and no vectorization.

Figure 4(c) presents the performance of the K40c GPU for the same experimental configuration as before, with a vector width of 2. The results for the versions with vector width 4 and no vectorization are omitted as they exhibit lower performance. Surprisingly, Fig. 4(c) shows that the NAIVE strategy outperforms SHARED and some of the GEN strategies. This can be explained by the unique properties of the Kepler Super Computing line of products to which the K40c belongs. These GPUs include enhanced L2 caching mechanisms that accelerate repeated and sparse memory accesses. This is especially advantageous as it caches repeated reads across streaming multiprocessors. However, the highest performance is attained by *GEN-RRS* which explicitly employs registers for both $Probs_i$ and $Grads_i$.

These performance results for a range of GPUs reinforce the importance of customizing compute kernels to each GPU's specific architecture and capabilities. For instance, each GPU achieved its highest performance by employing a different strategy. Moreover, each GPU displayed different strategy-performance orderings.

Comparison of Compute Devices. Figure 5 compares the highest speedup achieved by the RTX2080 Ti, GTX Titan-X Maxwell and Pascal, GTX980, K40c and K20m relative to the single-threaded baseline C++ version. These results are consistent with the relative capabilities of each device as listed in Table 3. For instance, the RTX2080 achieves the highest speedup of 245 relative to the baseline.

Table 3. Properties of the GPUs used in the evaluation

Device	RTX 2080 Ti	GTX Titan-X		GTX 980	K40c	K20m
Architecture	Turing	Pascal	Maxwell	Maxwell	Kepler	Kepler
Number of Cores	4352	3584	3072	2048	2880	2496
Clock (MHz)	1350	1417	1000	1126	745	706
GFlops (single)	13450	10157	6144	4612	4290	3520
GFlops (double)	420	317	192	144	1430	1170
Memory (GB)	11	12	12	4	12	5
Bandwidth (GB/s)	616	480	336.5	224	288	208

Figure 6 presents the execution time of the best-performing strategy for each GPU. In this figure, the performance is normalized over the non-vectorized kernel version for each GPU. Conforming to intuition, execution time of the RTX2080 improves with vector width. In contrast, it is notable that the Maxwell Titan-X and GTX980 achieve their highest performance with non-vectorized kernels, and the Pascal Titan-X and Kepler GPUs obtain the best performance with a vector width of 2. At one extreme, the Maxwell Titan-X exhibits an overhead factor of nearly 1.8 when using a vector width of 4. On the other hand, the RTX2080 Ti improves its performance by roughly 35% when it uses a vector width of 4 compared to the non-vectorized kernel. Therefore, explicit vectorization of the kernels can be either useful or harmful depending on the GPU architecture and the specific problem it is applied to.

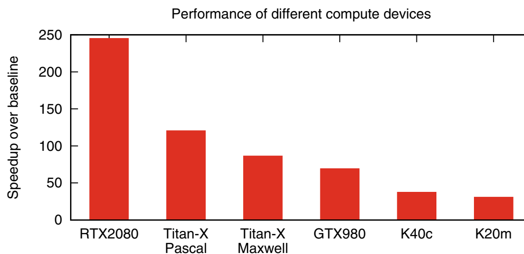

Fig. 5. Speedup comparison of best-performing parallel configurations of each compute device normalized over baseline C++ version. Relevant model parameters: K = 1024, $\mathcal{E}_n = 4096$, $|\mathcal{V}_n| = 32$.

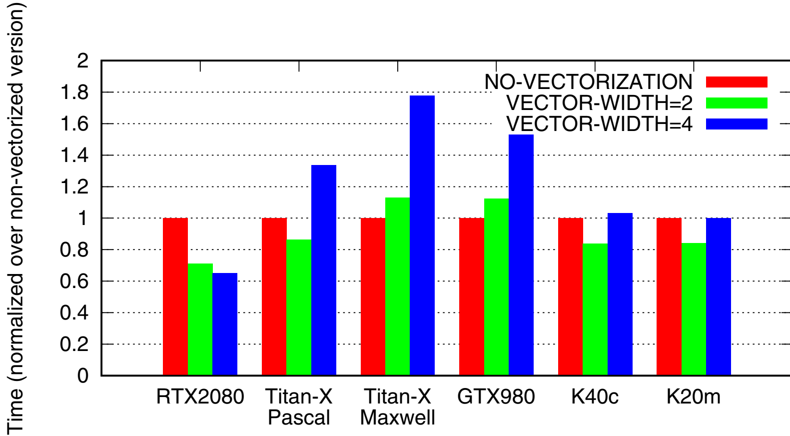

Fig. 6. Performance of the best-performing strategy per GPU, varying the vector width, normalized over the non-vectorized kernel. Relevant model parameters: K = 1024, $\mathcal{E}_n$ = 4096, $|\mathcal{V}_n|$ = 32.

5 Related Work

Several studies looked into the problem of tuning compute kernels [11,19]. Kernel Tuner [19] is a tool that facilicates the exploration of the available tuning parameters by appling multiple strategies to arrive at optimized configurations. The main focus of this work was overcoming discontinuous search spaces of established optimization techniques such as tiling and loop unrolling. On the other hand, Lim et al. [11] leveraged static code analysis to suggest tuning parameters without the need for experimentation. In contrast, our study focuses on application-specific data structures, memory and caching optimizations that required fundamental data representation changes. For example, we deduplicated matrices and re-encoded a graph as a cuckoo hash.

Recent work focused on the applicability of graph algorithms on GPUs. The common pattern is representing vertices and edges such that GPU memory hierarchies can be effectively utilized. WolfGraph [22] tackles graph processing in an edge-centric manner which prevents load imbalances associated with vertex-centric traversals. We incorporated a similar technique to avoid nondeterministic edge indirections when processing mini-batches. XBFS [5] laid out a methodology to perform breadth-first search on GPUs.

Mei et al. [13] provided a micro-benchmark that assessed the memory hierarchies of different GPU models. Similar to our evaluation results, they show how the seemingly similar memory hierarchies of different GPU models can produce non-intuitive performance outcomes.

Whereas acceleration of *deep learning* algorithms on GPUs is an ongoing success story, approximative Baysian algorithms (where our MCMC algorithm belongs) are not natural candidates for acceleration. Nevertheless, a number of projects explore this terrain. Medlar et al. [12] use GPUs with their MCMC

approach to analyze parental linkage patterns in a biology context and White and Porter [20] do the same to model terrorist activity. Latent Dirichlet Allocation, another variety of Bayesian Approximation, is used on GPUs by Yan et al. [21]. There is also related work on MCMC algorithms that use the gradient to speed up convergence. Langevin and Hamiltonian dynamics are representatives of these varieties [6]. Our algorithm uses Riemann Manifold Langevin dynamics. Beam et al. [2] use GPUs to perform Hamiltonian descent using Python interfaces to access the cuBLAS library [3]. They limit GPU optimizations to reducing data transfers between host and device memory.

Another MMSB algorithm with stochastic gradient descent on the GPU is the Online Tensor approach [8]. Their implementation uses the cuBLAS library. Unlike our work, there is no attempt to hand-optimize the GPU kernels. Since they target GPUs only, the datasets they can handle are limited by the device memory of the GPU. Our implementation can also be used, with reduced speedup, on a multicore CPU which allows much larger datasets.

6 Conclusion

Identifying optimization strategies of parallel data-intensive algorithms is a complex task. The SG-MCMC algorithm discussed in this paper posed additional challenges due to its unique stochastic nature and nondeterministic memory access patterns. We presented a methodology of improving performance by fundamentally restructuring the algorithm to cater for concurrency.

A deep analysis showed the algorithm's state can be reduced by 75%. We navigated the complex optimization landscape by dynamically generating compute kernels and testing different combinations of optimizations. This effort culminated in significant speedup factors of 21 and 245 using a multi-core CPU and a GPU respectively, compared to an optimized sequential program. Finally, we contrasted the performance of several GPUs highlighting the difference between their optimal configurations.

The outcome of this work reinforces the significance of avoiding premature optimization as it can lead to unexpected results. In particular, the success of common GPU optimizations depends on the particular device in use and the problem it is applied to. Although GPU architectures and their memory hierarchies can be leveraged to obtain significant speedups, they introduce significant complexity which hinders our ability to predict their benefits.

References

1. Bal, H., et al.: A medium-scale distributed system for computer science research: infrastructure for the long term. Computer **49**(05), 54–63 (2016). https://doi.org/10.1109/MC.2016.127

2. Beam, A.L., Ghosh, S.K., Doyle, J.: Fast hamiltonian monte carlo using GPU computing. arXiv preprint (2014). https://doi.org/10.1080/10618600.2015.1035724. http://arxiv.org/abs/1402.4089
3. cuBlas Home Page. http://docs.nvidia.com/cuda/cublas
4. El-Helw, I., Hofman, R., Li, W., Ahn, S., Welling, M., Bal, H.E.: Scalable overlapping community detection. In: IPDPS Workshops 2016, Chicago, IL, USA, 23–27 May 2016, pp. 1463–1472 (2016). https://doi.org/10.1109/IPDPSW.2016.165
5. Gaihre, A., Wu, Z., Yao, F., Liu, H.: XBFS: exploring runtime optimizations for breadth-first search on GPUs. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing HPDC 2019, pp. 121–131. ACM, New York (2019). https://doi.org/10.1145/3307681.3326606
6. Girolami, M., Calderhead, B., Chin, S.A.: Riemann manifold langevin and hamiltonian monte carlo methods. J. Roy. Stat. Soc. Ser. B (Methodological) (2012). https://doi.org/10.1111/j.1467-9868.2010.00765.x
7. Gupta, S., Agrawal, A., Gopalakrishnan, K., Narayanan, P.: Deep learning with limited numerical precision. CoRR (2015). http://arxiv.org/abs/1502.02551
8. Huang, F., Niranjan, U.N., Hakeem, M.U., Verma, P., Anandkumar, A.: Fast detection of overlapping communities via online tensor methods on GPUs. CoRR (2013). http://arxiv.org/abs/1309.0787
9. Leskovec, J., Krevl, A.: SNAP datasets: stanford large network dataset collection, June 2014. http://snap.stanford.edu/data
10. Li, W., Ahn, S., Welling, M.: Scalable MCMC for mixed membership stochastic blockmodels. CoRR (2015). http://arxiv.org/abs/1510.04815
11. Lim, R., Norris, B., Malony, A.: Autotuning GPU kernels via static and predictive analysis. In: 2017 46th International Conference on Parallel Processing (ICPP), pp. 523–532, August 2017. https://doi.org/10.1109/ICPP.2017.61
12. Medlar, A., Glowacka, D., Stanescu, H., Bryson, K., Kleta, R.: SwiftLink: parallel MCMC linkage analysis using multicore CPU and GPU. Bioinformatics **29**(4), 413–419 (2013). https://doi.org/10.1093/bioinformatics/bts704
13. Mei, X., Chu, X.: Dissecting GPU memory hierarchy through microbenchmarking. IEEE Trans. Parallel Distrib. Syst. **28**(1), 72–86 (2017). https://doi.org/10.1109/TPDS.2016.2549523
14. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6**(2), 40–53 (2008). https://doi.org/10.1145/1365490.1365500
15. Nugteren, C.: CLCudaAPI: a portable high-level API with CUDA or OpenCL back-end. https://github.com/CNugteren/CLCudaAPI
16. NumPy Home Page. http://www.numpy.org
17. OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems. http://www.khronos.org/opencl/
18. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004). https://doi.org/10.1016/j.jalgor.2003.12.002
19. van Werkhoven, B.: Kernel tuner: a search-optimizing GPU code auto-tuner. Future Gener. Comput. Syst. **90**, 347–358 (2019). https://doi.org/10.1016/j.future.2018.08.004
20. White, G., Porter, M.D.: GPU accelerated MCMC for modeling terrorist activity. Comput. Stat. Data Anal. **71**, 643–651 (2014). https://doi.org/10.1016/j.csda.2013.03.027

21. Yan, F., Xu, N., Qi, Y.: Parallel inference for latent dirichlet allocation on graphics processing units. In: 23rd Annual Conference on Neural Information Processing Systems. Vancouver, Canada, pp. 2134–2142 (2009). http://papers.nips.cc/paper/3788-parallel-inference-for-latent-dirichlet-allocation-on-graphics-processing-units
22. Zhu, H., He, L., Leeke, M., Mao, R.: WolfGraph: the edge-centric graph processing on GPU. Future Generation Computer Systems (2019). https://doi.org/10.1016/j.future.2019.09.052. http://www.sciencedirect.com/science/article/pii/S0167739X18325251