Report Number: WUCSE-2015-003

2015-9

WOODSTOCC: Extracting Latent Parallelism from a DNA Sequence Aligner on a GPU

Authors: Stephen V. Cole, Jacob R. Gardner, and Jeremy D. Buhler

An exponential increase in the speed of DNA sequencing over the past decade has driven demand for fast, space-efficient algorithms to process the resultant data. The first step in processing is alignment of many short DNA sequences, or reads, against a large reference sequence. This work presents WOODSTOCC, an implementation of short-read alignment designed for Graphics Processing Unit (GPU) architectures. WOODSTOCC translates a novel CPU implementation of gapped short-read alignment, which has guaranteed optimal and complete results, to the GPU. Our implementation combines an irregular trie search with dynamic programming to expose regularly structured parallelism. We first describe this implementation, then discuss its port to the GPU. WOODSTOCC's GPU port exploits three generally useful techniques for extracting regular parallelism from irregular computations: dynamic thread mapping with a worklist, kernel stage decoupling, and kernel slicing. We discuss the performance impact of these techniques and suggest further opportunities for improvement.
... **Read complete abstract on page 2.**

WOODSTOCC: Extracting Latent Parallelism from a DNA Sequence Aligner on a GPU

Complete Abstract:

An exponential increase in the speed of DNA sequencing over the past decade has driven demand for fast, space-efficient algorithms to process the resultant data. The first step in processing is alignment of many short DNA sequences, or reads, against a large reference sequence. This work presents WOODSTOCC, an implementation of short-read alignment designed for Graphics Processing Unit (GPU) architectures. WOODSTOCC translates a novel CPU implementation of gapped short-read alignment, which has guaranteed optimal and complete results, to the GPU. Our implementation combines an irregular trie search with dynamic programming to expose regularly structured parallelism. We first describe this implementation, then discuss its port to the GPU. WOODSTOCC's GPU port exploits three generally useful techniques for extracting regular parallelism from irregular computations: dynamic thread mapping with a worklist, kernel stage decoupling, and kernel slicing. We discuss the performance impact of these techniques and suggest further opportunities for improvement.

WOODSTOCC: Extracting Latent Parallelism from a DNA Sequence Aligner on a GPU

Stephen V. Cole
Washington University
St. Louis, MO, USA
svcole@wustl.edu

Jacob R. Gardner
Washington University
St. Louis, MO, USA
gardner.jake@wustl.edu

Jeremy D. Buhler
Washington University
St. Louis, MO, USA
jbuhler@wustl.edu

***Abstract*—An exponential increase in the speed of DNA sequencing over the past decade has driven demand for fast, space-efficient algorithms to process the resultant data. The first step in processing is alignment of many short DNA sequences, or *reads*, against a large reference sequence. This work presents WOODSTOCC, an implementation of short-read alignment designed for Graphics Processing Unit (GPU) architectures. WOODSTOCC translates a novel CPU implementation of gapped short-read alignment, which has guaranteed optimal and complete results, to the GPU. Our implementation combines an irregular trie search with dynamic programming to expose regularly structured parallelism. We first describe this implementation, then discuss its port to the GPU. WOODSTOCC's GPU port exploits three generally useful techniques for extracting regular parallelism from irregular computations: dynamic thread mapping with a worklist, kernel stage decoupling, and kernel slicing. We discuss the performance impact of these techniques and suggest further opportunities for improvement.**

I. Introduction

General-purpose GPU programming languages such as CUDA and OpenCL enable a broad range of algorithms to be ported to GPUs for greater performance. Algorithms that consist of regular, homogeneous calculations repeated over many data elements are easiest to port to the GPU's many-core, SIMT architecture. However, as GPU development matures at every level, from increased architectural capabilities and device resources to more sophisticated tool suites and language functionality, it becomes increasingly interesting and feasible to map more complex, more irregularly-organized computations onto GPUs.

In this work, we consider a compute-intensive application of strong interest to the biological community: short-read DNA sequence alignment. In this application, a large number of short DNA strings ("reads") are matched against an index generated from a long DNA reference string to locate areas of correspondence. Although all reads are processed with the same algorithm, each one requires a different amount of computation to compare against the reference. This irregular per-element behavior is typical of other algorithms that filter a long stream of elements, such as Viola-Jones face detection in images [19]. A key question in porting such applications to a GPU is how to regularize their behavior for maximum performance with a SIMT architecture, even though each input fundamentally takes a different amount of work to process.

This paper describes a new short-read alignment tool, WOODSTOCC (WOODSTOCC Offers Optimal Dynamic programming - Suffix Trie alignment while optimizing OCCupancy). WOODSTOCC seeks to ameliorate the fundamental irregularity of short-read alignment in order to extract parallelism suited to a GPU. We first describe algorithmic transformations of short-read alignment, applicable to any architecture, that partly regularize it and expose SIMT parallelism. We then present a CUDA implementation of WOODSTOCC for NVIDIA GPUs. This implementation exploits several techniques to maximize *occupancy* – the fraction of the GPU's resources actively involved in computation – and thereby avoid having resources remain idle for long periods. Finally, we empirically study the performance of WOODSTOCC to assess whether it indeed boosts occupancy and to suggest avenues for future performance improvement.

We anticipate that the techniques employed by WOODSTOCC will be generally useful for GPU application designers seeking to fully occupy the GPU despite irregular behaviors in their target applications.

II. Background

A. Problem Definition

This work focuses on a variant of DNA sequence alignment known as the *short-read alignment problem*, which is motivated by the development over the last decade of experimental techniques for sampling a very large number of short substrings, or *reads*, from a long DNA sequence such as a genome [17]. This inexact matching problem is as follows:

Given a single DNA reference sequence of length n, and many DNA reads of common length $m \ll n$, find for each read all starting positions in the reference where the entire read matches with no more than k differences.

Allowable differences include substitution, insertion, or deletion of individual characters. Figure 1 shows a read alignment to part of a reference with three differences. Typically, the reference length n is tens of millions to billions of characters, while the read length m ranges from a few tens to 200 characters. A sequencing experiment generates tens to hundreds of millions of reads that must be aligned to the reference.

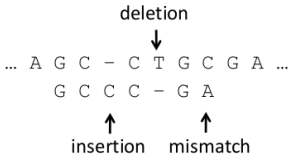

Fig. 1. Alignment of a read CGCCGA to a longer reference sequence with three differences.

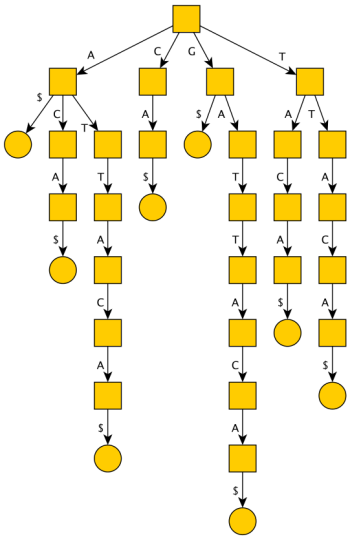

Fig. 2. Suffix trie for string GATTACA$. Each path from root to leaf is labeled with a suffix of the string.

B. Related Work

The computational demands of short-read alignment have led to several novel algorithmic strategies for rapidly matching many reads against a common reference. Methods using a hash-based index of the reference [12], [15] have largely been superseded by tools that construct a tree-based index, such as a *suffix trie* (see Figure 2). Due to the high storage cost to represent such tries explicitly, they are instead represented *virtually*, using a compact data structure such as the FM-index [5] that allows the trie to be reconstructed on the fly during alignment. FM-index-based aligners represent the current state of the art in fast short-read alignment software [9], [10], [13].

Our approach to short-read alignment emphasizes *completeness* of results. Most short-read aligners use trie-search heuristics that can quickly locate at least one match to a given read in the reference with up to k differences, if any exist, but may not find all such matches. Incomplete results are useful for read-mapping applications in which matches to multiple sites are simply discarded, but completeness is desirable for more precise analyses such as genome rearrangement history [1] and interspecies read alignment. Among CPU-based aligners, the BWA software [10] in "−N mode" is one of the few that produce complete output, so we use this tool as our CPU baseline for performance comparison.

Like WOODSTOCC, some short-read alignment tools use a dynamic programming (DP) algorithm such as Smith-Waterman [18] as part of their search algorithm. However, these tools are either optimized for aligning much longer sequences than a typical short read [7], [11] or use DP only to filter putative matches obtained by an incomplete trie-search heuristic [4], [8].

Modern short-read alignment methods have previously been ported to GPUs. SARUMAN [2] uses a hash-based index on the GPU, but its storage requirements restrict its use to microbial-sized genomes (around 10^6 characters). MUMmerGPU [16] searches an explicit representation of the suffix tree (a compressed trie), which is copied to the GPU a piece at a time; however, it does not allow for differences between reads and reference. BarraCUDA and CUSHAW [6], [14] implement BWA-like implicit trie search on the GPU, but they are limited either to incomplete trie search heuristics or to finding alignments without insertion and deletion of characters.

In contrast to previous GPU-based short-read aligners, WOODSTOCC combines virtual trie traversal with dynamic programming to guarantee completeness of its results. Its approach to regularizing GPU-based computation shares some similarities with BarraCUDA's worklist strategy, but it is quite different in detail to accommodate the needs of both dynamic programming and trie traversal.

III. Alignment Strategy

This section describes how WOODSTOCC organizes its alignment computation at a high level to expose regular structure. We exploit the highly structured nature of dynamic programming alignment to allow computations for many reads to proceed in lock-step.

A. Core Algorithm

Conceptually, we attempt to align a given read starting at every character in the reference by aligning it to each labeled path in the reference's (virtual) suffix trie. The trie is traversed depth-first beginning at the root. Each traversal step descends by one node, or equivalently one character, along an edge and computes one row of a dynamic programming matrix (a "DP row"). When the computation reaches node x of the trie, the DP row contains the least number of differences in an optimal alignment between each prefix of the read and all occurrences in the reference of the substring labeling the path from the root to x. The set of leaves below the current node indicate the reference locations at which this substring occurs. As shown in Figure 3, when the trie branches, the work done to compute the DP row at the branch point can be reused on each of the branching paths; hence, the total number of rows computed equals the number of nodes reached by the traversal.

Because the number of differences allowed in the alignment is *a priori* limited to k, traversal down a given path may be truncated once it is clear that all alignments of the read to that path must have more than k differences, i.e. when all entries of a DP row become $> k$. Truncating alignments in this way limits the depth traversed down any path from the root to $O(m)$ and so greatly curtails the cost of alignment

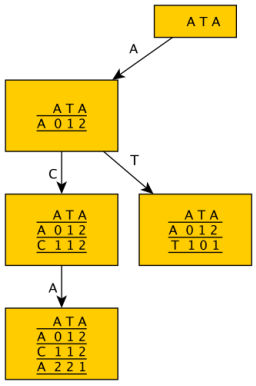

Fig. 3. Alignment of read ATA against a fragment of the reference trie of Figure 2. Each internal node shows the dynamic programming matrix (set of DP rows) between the read and the string labeling the path to that node. Each downward step computes one more row of the matrix.

overall. We also note that limiting the number of differences to k permits the use of *banded alignment*, which computes only $2k+1$ cells per DP row regardless of the read length m. In all the work described for this paper, we set $k = 3$.

Because storage of the complete suffix trie, or even a more efficient suffix tree, would be prohibitively large for genomes of 10^8 or more characters, the trie is stored implicitly using the FM-index. Each step down a path in the trie involves a computation to "discover" the current node's children, as described in [10]. A discovery computation performs several random accesses to the FM-index data structures in memory, which are comparable in size to the original reference sequence.

B. Read Batching

Table I compares the running time of the above algorithm ("naive aligner") to that of BWA v0.6.0 (using the `-N` option to ensure completeness of output) when sequentially aligning a list of 10^5 reads of length $m = 47$ against human chromosome 1 ($n = 2.5 \times 10^8$), on a single core of a 2.6 GHz Intel Core i5 CPU. The majority of the algorithm's time is spent in the discovery computations of virtual trie traversal; overall, it is slower than BWA.

To reduce the cost of trie traversal and lay the groundwork for the parallelization to follow, we implement *read batching*. In one traversal of the suffix trie, we simultaneously perform dynamic programming alignment computations for many different reads, maintaining separate DP row data structures for each. Each time a node is discovered by traversal, the DP rows for all reads are updated as the traversal moves down to this node. This process is conceptually equivalent to doing a single traversal of the trie and then reusing all the discovered nodes to perform dynamic programming for each read. Because the discovery computation is expensive, performing it once per node instead of N times for N reads substantially reduces computation at the cost of maintaining N simultaneous DP rows, each of size $O(k)$.

An important detail of read batching is that some, but not all, reads in a batch may have their alignment computation truncated on a given path due to having no alignment with $\leq k$ differences. We track the set of "live" reads in a batch during traversal (i.e. those that have not yet been truncated) and perform dynamic programming at each step only for live reads. In this way, the total DP work performed matches that of the naive implementation, while the traversal work is much less. As shown in the last row of Table I, batching all reads into a single group greatly reduces time spent in traversal and nearly halves overall running time.

Implications for Parallelization: Dynamic programming with read batching introduces a high degree of regularity to read alignment. In particular, all live reads' DP computations following a given traversal step can be performed in lockstep using SIMT parallelism. This regularity is in contrast to multithreaded CPU versions of BWA and related aligners, which use one independently executing thread per CPU core, each processing a separate stream of reads. Batching seems particularly attractive for devices with a very wide SIMT programming model, such as GPUs.

However, parallelizing the alignment of each read in a batched implementation introduces the problem of *idle reads*. Traversal down a given path in the trie continues until no reads in the current batch are live; however, some reads may become dead before others, and no further progress is made on those reads until the traversal returns to a node at which they are live. If reads are statically batched and assigned to compute resources (e.g. vector slots for SIMD instructions, or GPU threads), resources will be left idle whenever their reads become dead, limiting parallelism.

To assess the impact of idle reads, we measured the proportion of live reads encountered by the batched implementation at a range of depths in the trie (averaged over all nodes reached at that depth). These results are given in Table II. All reads remain live near the top of the trie, in particular for the first k levels where no alignment accumulates enough differences to be ruled out. Thereafter, a substantial fraction of reads are truncated at each level; after just a few levels, most reads at a given node are dead.

A wide SIMT architecture such as a GPU encourages highly regular parallelism, yet the fundamentally irregular nature of when reads' alignments are truncated in trie traversal demands that we eliminate or mitigate idle reads. Dealing with these conflicting design pressures informs our GPU implementation, which we describe next.

IV. GPU Implementation

We now describe WOODSTOCC, the implementation of our aligner on a GPU using CUDA. We first review the relevant aspects of the GPU architecture, then describe at a high level our mapping of the alignment algorithm onto the GPU. We then identify a flexible mapping strategy that maintains high utilization of the GPU's parallelism throughout the various stages of the algorithm despite the irregular aspects of the application's execution.

Algorithm	Time			
	Total(s)	Traversal (%)	DP calculation (%)	Other (%)
BWA -N	634	N/A	N/A	N/A
Naive aligner	860	51	19	30
Batched aligner	454	17	63	20

TABLE I

COST TO ALIGN 10^5 READS OF LENGTH 47 AGAINST HUMAN CHROMOSOME 1 ON ONE CORE OF 2.6 GHZ INTEL CORE I5 CPU. FOR WOODSTOCC ALGORITHMS, TIME SPENT IN EACH PORTION OF THE ALGORITHM IS PROFILED.

Depth	Liveness	Depth	Liveness
0-3	1	8	0.014
4	0.377	9	0.005
5	0.191	10	0.002
6	0.086	11	0.001
7	0.036		

TABLE II

AVERAGE FRACTION OF LIVE READS AT TRAVERSAL DEPTHS 0-11 IN THE BATCHED IMPLEMENTATION, $k = 3$.

A. CUDA GPU Model

Computation on a GPU using CUDA is organized hierarchically: a *grid* is mapped onto the device; the grid consists of *blocks*; and each block consists of *threads*. When an application *kernel* is launched on a GPU, blocks are assigned to processing elements (called Streaming Multiprocessors or SMs). The organization of threads into blocks and blocks into a grid is programmer-controlled and specified at kernel launch time. Each CUDA execution flow instance is known as a *context* and contains all information necessary for the kernel to execute.

While each thread conceptually performs an independent unit of work at each time step, the actual execution granularity of threads is an array of 32 threads, called a *warp*. During execution, the scheduler for each SM swaps active warps in and out of execution, until all warps from all blocks assigned to that SM have completed. While executing, the 32 threads within a warp follow instructional lock-step: during each execution cycle, each thread in the warp will either execute a common instruction or be idle. Code divergence between threads in the same warp is detrimental to performance, since each branch of the divergence gets executed serially, with some threads being idle during the execution of each branch.

Synchronization barriers are implemented as built-ins in CUDA at the block level only. When a synchronization barrier in the kernel code is reached, threads from all warps in a block must wait until they have all reached the barrier before proceeding. Global synchronization across blocks is theoretically possible but would be prohibitively costly.

GPUs have several different types of memory, each with different characteristics, but three are most relevant to our application: a large but slow *global memory* shared among all blocks; fast per-block *shared memory*; and very fast per-thread *registers*. If the amount of requested per-thread data storage exceeds register resources at any time during execution, the excess data is spilled into a special section of global memory called *local memory*.

All experiments reported in this work used an NVIDIA GeForce GTX Titan GPU with Kepler architecture. This card contains 14 SMs, each of which can maintain up to 64 active warps, operating at 837 MHz with 6.1 GB of global memory.

B. Application mapping to GPU

WOODSTOCC divides the set of reads to be aligned across multiple blocks on the GPU. Each block operates autonomously and runs a CUDA kernel that implements virtual suffix trie alignment on a disjoint subset of reads. All blocks share a common FM-index data structure for virtual trie traversal, which occupies the majority of global memory. However, each block independently performs DP and traversal of the virtual suffix tree to align its subset of reads. As each block discovers alignments of its reads to the reference with at most k differences, it aggregates them into a per-block global memory buffer that is copied back to the host when the block completes.

The relatively short duration and high data volume of each DP and trie traversal calculation step in our algorithm precludes mapping strategies that require frequent control transfers between CPU and GPU. In particular, we cannot put either trie traversal or DP alone on the GPU but must implement the entire algorithm. We chose to implement the algorithm as a single, monolithic kernel to enable data transfer between algorithm stages in per-block shared memory. The principal innovation in WOODSTOCC's approach is in how it organizes this application kernel, which we describe next.

C. Kernel organization to maximize occupancy

WOODSTOCC is designed to map the threads of each block to the work of the application kernel so as to maximally exploit the SIMT parallelism of the GPU. In particular, we seek to maximize GPU *occupancy* in two senses: first, all threads within a block should be kept busy doing useful work (thread occupancy); and second, there should be enough available blocks of work that no SM is idle for lack of a block to work on (block occupancy).

In a traditional, regular GPU application such as matrix multiplication, a straightforward static mapping of threads onto work units for the duration of a kernel suffices to maintain high occupancy. However, the irregular nature of short read alignment demands a more nuanced approach. Batching is an effective algorithmic optimization, but when few reads remain live, thread occupancy within a block suffers greatly. This challenge extends to block occupancy as well – some blocks may finish much sooner than others, leaving SMs partially or completely idle while the last few blocks finish.

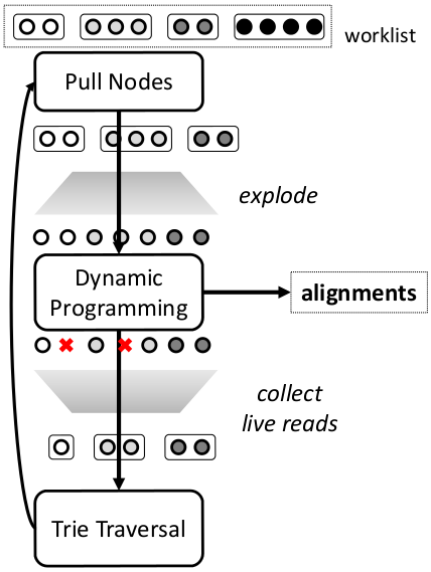

Fig. 4. Stages of the WOODSTOCC GPU kernel main loop. Trie nodes with nonempty batches of live reads (circles) are queued on a worklist. The first stage pulls enough nodes to assign a live read to most threads in the block. Nodes are "exploded" into their read sets for dynamic programming, after which the remaining live reads are collected into new, smaller read batches. Finally, nodes are subjected to trie traversal, which generates more work for the worklist. The output of the kernel, read-reference alignments, is generated by the dynamic programming stage.

Figure 4 illustrates the structure of the WOODSTOCC GPU kernel. The kernel repeatedly executes a main loop that performs alternating DP and trie traversal steps. Two key features of this kernel are the use of a *worklist* and associated *pull stage*, which ameliorate the problem of idle reads in DP, and *dynamic remapping*, which recomputes the mapping of work to threads in a block on the fly, entirely within the GPU, to enable differently shaped computations to run efficiently in a single kernel.

Eliminating idle reads: Recall that the CPU version of our alignment algorithm works on a single batch of live reads in lock-step until the batch is exhausted. The algorithm therefore works on live reads from only a single trie node at a time. In contrast, WOODSTOCC aggregates live read batches from multiple trie nodes into a common worklist. On each loop iteration, the kernel pulls and simultaneously computes on as many batches from this worklist as are needed to occupy (nearly) all threads in a block.

The worklist is maintained as follows. Starting with the root node, nodes with live reads are added the worklist as they are uncovered by traversal. Each queued node requires a DP calculation for each of its live reads, as well as a trie traversal step to discover and enqueue any of its child nodes with nonempty live read batches. Nodes become dead when all reads associated with them are dead, or when they have no more children to expose; such nodes are not enqueued. The kernel's main loop runs until the worklist is empty, at which time every possible live node in the trie has been processed.

To utilize the worklist effectively, we add a "pull" stage to the main loop prior to DP and trie traversal. In this stage, the kernel computes a progressive sum of batch sizes for the first m nodes on the worklist, for each m from 1 up to the number of threads per block. Each node may have a different-sized batch of live reads. The kernel then dequeues the maximum number of nodes such that the sum of their batch sizes fits within the number of threads for the block. For efficiency, the progressive sum is computed using a parallel scan operation in time logarithmic in the list size. The DP calculation then devotes one GPU thread to each live read from the dequeued batches.

Because a loop iteration works on either all or none of a node's batch of live reads, the worklist does not guarantee that we always have a live read for every thread in a block. However, we anticipate that most threads will have DP work to do on every loop iteration, leaving very few threads idle.

Dynamic remapping: An important feature of the WOODSTOCC kernel is that it combines operations at different granularities. Dynamic programming uses one thread per live read, while the pull and trie traversal stages of each loop work with entire nodes, rather than their component live reads. WOODSTOCC dynamically reassigns work to threads within each loop iteration of the kernel to match the granularity of the computation required by each computation stage to the granularity of the GPU's parallelism.

The pull stage of the computation assigns one GPU thread to each node pulled from the worklist. After some initial work, we must "explode" the nodes to assign one live read per GPU thread for DP. The reads from all dequeued nodes are assigned to consecutive threads, but each read must also know which node it came from. To compute each read's node efficiently in parallel, we use a parallel binary search [3] on the progressive sum of the number of live reads in each node, which was computed in the pull stage.

After DP, we know whether each read remains live, but we must then regroup *only* the live reads for each node into new, usually smaller batches. For efficiency, this operation is also parallelized, using a segmented parallel scan to map the live reads for each node into a dense array stored with the node. Finally, trie traversal is performed at node granularity, and nodes are enqueued onto the worklist for future processing.

D. Decoupling traversal from DP

A straightforward implementation of the WOODSTOCC main loop executes each of its three stages once per main loop of the kernel. However, we anticipate that trie traversal, a major part of the computation, will typically not utilize all available threads. First, the number of nodes dequeued by the pull stage may be much less than the number of threads, since each node may have multiple live reads. Second, some nodes' read batches may be empty after the DP step, in which case those nodes are discarded without the need for trie traversal.

We aggressively maximize thread occupancy for trie traversal by dynamically scheduling the stages of the kernel's main loop. In particular, we allow multiple iterations of the pull and DP stages to be executed between calls to trie traversal. Each

iteration performs DP on read batches from a different set of nodes, generating work for the traversal stage. We queue up this work (in the form of pending nodes for trie traversal) on a separate worklist until enough traversal work exists to occupy most or all threads in a block; only then do we pull nodes from this list and perform the traversal operation.

E. Boosting block occupancy via kernel slicing

Periodic kernel slicing is a technique for maintaining occupancy that operates at block rather than thread granularity. The purpose of kernel slicing is to maximize the number of *active blocks* available to execute on an SM at any given time. When a kernel is launched, it does not terminate until all blocks in its grid terminate, so all blocks must wait for the slowest block to finish. If some blocks are much slower to finish than others, the SM spends much of its time underutilized.

To avoid idle blocks, periodic kernel slicing decomposes the execution of a single monolithic kernel instance into many shorter instances called *slices*, effectively pausing the kernel after each slice and assigning any idle blocks a new set of reads to be processed. Kernel slicing exploits the fact that global memory persists between kernel launches within the same CUDA context. A statically allocated global continuation buffer holds all non-persistent block data between slices, so that control can be returned to the CPU between slices via kernel termination and re-launch, with all necessary data persisting on the device across slices. In this way, the maximum time a block could remain idle is the length of a single slice, rather than the execution time of the maximum-latency block in its grid.

We note that an effect similar to periodic kernel slicing can be achieved in CUDA simply by creating a big enough computation grid that the total number of blocks to process exceeds the maximum number of active blocks for the device. For example, the maximum number of active blocks per SM on the GTX Titan device is 16, for a total of $16 \times 14 = 224$ active blocks. As active blocks are retired, CUDA schedules the remaining blocks onto the available SMs. However, this approach fails to scale to very large data sets because the number of blocks that can be created for one kernel execution is limited by the device resources and/or CUDA implementation. Periodic kernel slicing, on the other hand, faces no such limitation because it periodically restarts the kernel after creating new blocks as needed.

To see why slicing is helpful for our computation, consider Figure 5, which shows a CDF of measured block finishing times over the course of a kernel instance. This CDF confirms that, while most blocks have a short execution time, a few take dramatically longer to complete. A monolithic kernel does not return control to the host until every block is finished; hence, blocks with low latencies will remain idle for most of the kernel's overall execution time. With slicing enabled, blocks with low latencies can be assigned new work as blocks with long latencies continue their execution, better utilizing the GPU's resources.

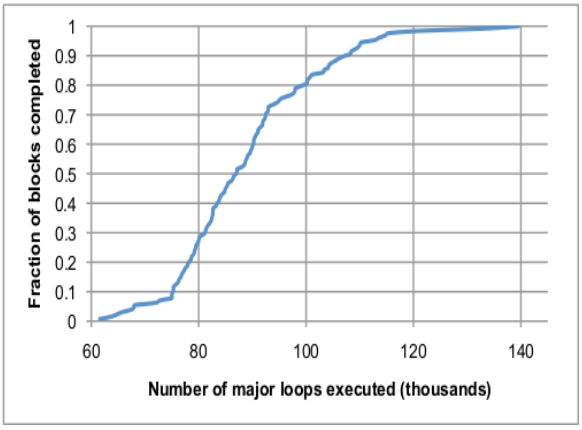

Fig. 5. Cumulative distribution function of the number of major loop iterations executed by each of 128 blocks when run to completion.

WOODSTOCC incorporates slicing at intervals of 3000 main loops, which introduces negligible overhead according to profile data. Empirical tests showed that this yields a time savings of approximately $5 - 10\%$ for input sets of size comparable to the one tested in our experiments.

V. Results

We now provide empirical measurements to support the utility of our occupancy-boosting techniques for the WOODSTOCC kernel and to assess its performance. Measurements were taken on an NVIDIA GeForce GTX Titan from a run processing an input set of 10^5 length-47 reads against human chromosome 1. Unless otherwise specified, the GPU kernel was run with 160 threads per block.

A. Impact of worklist and dynamic mapping

Table III illustrates the distribution of work (as measured by GPU cycles) among the three stages of the WOODSTOCC main loop, as well as the measured thread occupancy for each. The profiled implementation used the worklist and dynamic remapping of work between reads and threads, but not the decoupling of trie traversal from the rest of the loop. Our design successfully boosted the thread occupancy of dynamic programming to around 86%, and the thread occupancy to greater than 70% for two thirds of the application's total runtime. As predicted, the thread occupancy in trie traversal is limited by previous loop stages; it is only about a third.

Stage	Time (%)	Thread occupancy (%)
Pull from worklist	24	71
DP align	42	86
Trie traversal	34	31

TABLE III
RELATIVE CONTRIBUTION OF EACH MAIN STAGE OF THE KERNEL TO OVERALL RUNTIME, AND THREAD OCCUPANCY DURING EACH STAGE MEASURED AS THE PROPORTION OF THREADS MAPPED TO USEFUL WORK UNITS. ALL MEASUREMENTS WERE TAKEN FROM ALIGNING A SET OF 10^5 LENGTH-47 READS AGAINST HUMAN CHROMOSOME 1 AND AVERAGED ACROSS ALL GPU EXECUTION BLOCKS.

We also investigated the impact of the number of threads per block and blocks per SM on overall performance. We measured the best performance to come from using 160 threads per block and 6 blocks (30 warps) per SM. While this configuration does not maximize the number of active blocks (16) or active warps (64) per SM supported by the device, our dynamic work mapping technique does ensure high thread occupancy within the active warps present on the device, and optimizing over the thread/block configurations for performance is orthogonal to the current work.

B. Kernel stage decoupling

To probe the effect of stage decoupling, we measured thread occupancy during the third stage of the kernel for more and less aggressive queueing of nodes for trie traversal. Decoupling has the practical effect of growing the size of the main worklist, as it causes more nodes to become simultaneously available for processing. To control queueing, we chose different thresholds for how large this worklist was allowed to grow before traversal was triggered.

Table IV shows the effect of increasing the threshold from a relatively small value (twice the expected maximum depth of the trie) to a much larger value (20 times this depth). As anticipated, the occupancy of trie traversal with the more aggressive approach goes to nearly 100%, while the number of distinct calls to traversal drops by two-thirds.

The performance impact of boosting the occupancy of trie traversal from about one-third to 100% is sensitive to the intensity of the traversal stage. An earlier implementation of WOODSTOCC showed only a 6% reduction in overall runtime. We suspect that because trie traversal is extremely global-memory intensive and exhibits little coalescing of accesses across threads, boosting occupancy did not substantially reduce the number of distinct global memory transactions needed to align all reads to the reference, with correspondingly little overall performance impact. This optimization would likely be more effective for less global-memory intensive kernels, though we will also reinvestigate its impact in WOODSTOCC after anticipated optimizations to trie traversal.

C. Overall performance and limitations

On an Intel Core i7-950 processor, the best CPU software version of WOODSTOCC achieves a throughput of approximately 320 reads/sec per CPU core, while the throughput of the software tool BWA (version 0.7.9, modified to find alignments in the forward direction only) achieves throughput of approximately 360 reads/sec, on an input set of 10^5 length-47 reads against human chromosome 1. On the same input set, WOODSTOCC on an NVIDIA GeForce GTX Titan achieves throughput of approximately 4000 reads/sec – slightly more than a factor of 11 greater than BWA's per-core throughput (see Figure 6).

Threshold factor	Stage execution count	Avg thread occupancy
2	93652	34.5%
20	32346	100%

TABLE IV
STAGE EXECUTION COUNT AND AVERAGE THREAD OCCUPANCY (AS A PERCENTAGE OF MAXIMUM) DURING THE THIRD KERNEL STAGE FOR TWO THRESHOLD FACTORS. THE THRESHOLD FACTOR IS GIVEN AS THE RATIO OF THE MAIN WORKLIST SIZE TO THE MAX POSSIBLE DEPTH IN THE SEARCH TRIE. NOTE THAT, AS EXPECTED, THE THIRD STAGE IS EXECUTED FAR LESS FREQUENTLY AND ITS AVERAGE THREAD OCCUPANCY IS MAXED OUT WHEN THE HIGHER THRESHOLD FACTOR IS USED. ALL MEASUREMENTS WERE TAKEN FROM ALIGNING A SET OF 10^4 LENGTH-47 READS AGAINST HUMAN CHROMOSOME 1 AND AVERAGED ACROSS BLOCKS.

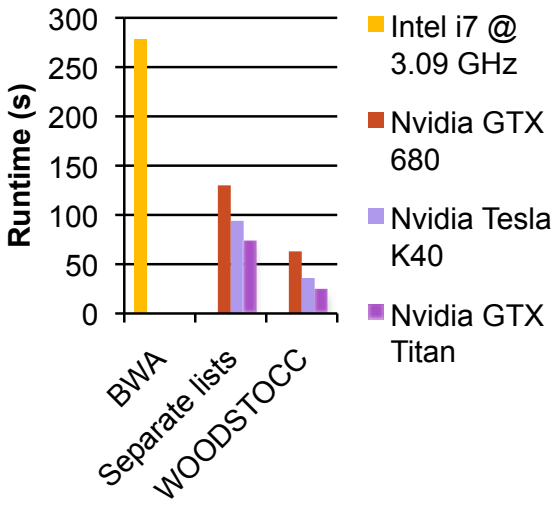

Fig. 6. Runtime comparison of BWA on a single CPU core and two versions of WOODSTOCC on three NVIDIA GPUs when aligning a dataset of 10^4 length-47 reads against human chromosome 1. The 'Separate lists' version employs a simple data-parallel strategy with independent worklists for each thread, whereas the full version incorporates dynamic thread mapping and kernel slicing.

To isolate the value added of our contributions over a pure data-parallel but task-independent approach, we implemented a version of WOODSTOCC in which threads are mapped to reads 1-to-1 for the duration of the algorithm and each thread stores its own (sequentially processed) worklist. While this version realized approximately a 3× speedup over BWA by itself, adding dynamic thread mapping with the shared worklist further increased performance by 2-3×, and kernel slicing by an additional 0.1×.

Performance of the task-independent version of WOODSTOCC and the full version including worklists and slicing is shown in Figure 6. These results show that the benefits of worklist maintenance and slicing far outweigh their overhead, and suggest their general utility as a parallel work management framework versus a pure task-independent framework.

The optimizations of this work were effective in boosting the thread occupancy of all stages of short read alignment, despite the inherent irregularity of the computation. The current bottleneck to further major performance gains is not thread occupancy. The occupancy of the DP stage, which is limited by the need to use reads from a whole number of states, could be further improved with a strategy such as node splitting or bin packing of whole states. However, we anticipate that according to Amdahl's Law the performance improvement from such a

change would be marginal, since there is only room for a further 14% increase in occupancy during this stage and it currently accounts for less than half of overall runtime.

VI. Conclusion and Future Work

We have presented WOODSTOCC, a new short-read alignment algorithm for the GPU that extracts regular, SIMT-compatible parallelism from an irregular alignment and trie search computation. WOODSTOCC employs three strategies– dynamic thread mapping, kernel stage decoupling, and periodic kernel slicing– to boost available GPU parallelism. These optimizations likely generalize to other irregular algorithms for filtering large data sets in variable amounts of time per element.

Examination of our kernel's timing results shows that execution time is dominated by dynamic programming and trie traversal. Having these as our current performance bottleneck, rather than a worklist or a thread mapping issue, is a testament to the effectiveness of the strategies presented here.

More important than further optimizing WOODSTOCC itself, we plan to investigate the applicability of our GPU optimization strategies to general DAG-specified computations. One example candidate is "decision cascades" in areas such as image processing, where variable decision times per input have until now limited the applicability of SIMT parallelism.

Acknowledgment

This work was funded by NSF award CNS-0905368. An earlier version of this work appeared at the 13th International Symposium on Parallel and Distributed Computing (ISPDC 2014).

References

[1] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, S. C. Sahinalp, R. A. Gibbs, and E. E. Eichler. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature Genetics*, 41:1061–7, 2009.

[2] J. Blom, T. Jakobi, D. Doppmeier, S. Jaenicke, J. Kalinowski, J. Stoye, and A. Goesmann. Exact and complete short read alignment to microbial genomes using GPU programming. *Bioinformatics*, 27(10):1351–1358, Mar. 2011.

[3] I. Buck and T. Purcell. *GPU Gems*, chapter A Toolkit for Computation on GPUs. Addison-Wesley, 2004.

[4] F. Fernandes, P. G. da Fonseca, L. M. Russo, A. L. Oliveira, and A. T. Freitas. Efficient alignment of pyrosequencing reads for re-sequencing applications. *BMC Bioinformatics*, 12:163, 2011.

[5] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Ann. Symp. Foundations of Computer Science*, pages 390–398, 2000.

[6] P. Klus, S. Lam, D. Lyberg, M. S. Chenug, G. Pullan, I. McFarlane, G. Yeo, and B. Lam. BarraCUDA - a fast short read sequence aligner using graphics processing units. *BMC Research Notes*, 5(1):27+, 2012.

[7] T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, Mar. 2008.

[8] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, Apr. 2012.

[9] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25+, 2009.

[10] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[11] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, Mar. 2010.

[12] H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18(11):1851–1858, Nov. 2008.

[13] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, Aug. 2009.

[14] Y. Liu, B. Schmidt, and D. L. Maskell. CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform. *Bioinformatics*, 28(14):1830–1837, July 2012.

[15] S. M. Rumble, P. Lacroute, A. V. Dalca, M. Fiume, A. Sidow, and M. Brudno. SHRiMP: accurate mapping of short color-space reads. *PLoS Computational Biology*, 5(5):e1000386+, May 2009.

[16] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. High-throughput sequence alignment using Graphics Processing Units. *BMC Bioinformatics*, 8(1):474+, 2007.

[17] J. Shendure and H. Ji. Next-generation DNA sequencing. *Nature Biotechnology*, 26:1135–45, 2008.

[18] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, Mar. 1981.

[19] P. Viola and M. Jones. Robust real-time object detection. *International Journal of Computer Vision*, 57(2):137–154, 2002.