Accelerating Genomic Analyses with Parallel Sliding Windows

Ben Kreuter, Ryan M. Layer, Michelle McDaniel, Gabriel Robins, and Kevin Skadron

Department of Computer Science
University of Virginia
Charlottesville, VA 22904
{brk7bx, rl6sf, mm4ab, robins, skadron}@cs.virginia.edu

Abstract

In recent years biology has become an information science, where an avalanche of newly sequenced genomic data has overwhelmed our existing analysis and mining tools. This paper addresses this challenge by developing a systematic way of speeding up a broad class of bioinformatics algorithms using commodity graphics processing hardware. Using the example problem of analyzing DNA structural variations, we demonstrate how such computations can be significantly accelerated in various parallel architectures, yielding over two orders of magnitude speedups at low cost and with relatively modest programming effort. Our implementation of a sliding window -based technique on the GPU and Cell architectures seems promising in its generality and extensibility to other problems and domains.

1. Introduction

While it has been long recognized that DNA sequence analysis is ripe for parallelization [4], most such work has concentrated on multiple alignment (e.g., Smith-Waterman [**?**] [**?**]) and sequence comparison (e.g., BLAST [**?**] [**?**]) algorithms. This progression is perhaps natural since these tasks are basic core steps in many experimental applications, and thus any speedup gains is these will tend to benefit the broader community. However, bioinformatics analysis is far from complete even after sequences have been aligned and compared, and parallelizing the full analysis tool chain has received considerably less attention. Improving the overall performance of common classes of bioinformatics analyses is becoming essential given the massive data volumes generated by experiments. Only with improved and faster techniques will researchers be able to fully utilize and leverage the available data.

Ultra-high throughput sequencing techniques opened the possibility of scaling previously small-scale experiments to whole genomes. While wet labs are essential to biology, the scope of any single experiment is limited by constraints such as cost, manpower, time, noise, etc. Bioinformatics analysis can augment and streamline the traditional experimental process by predicting the locations of particular properties throughout the genome. A typical experiment begins with a sequenced DNA sample; the sequences are mapped back to a reference genome, an analysis tool uses the mapping data to make predictions, and those predictions are then tested in the lab. The accuracy and granularity of the tool have a direct impact on the biologist's ability to verify the results. Sequencing data are large and complex, and attempts to tame this intractability often discard substantial portions of the data, and moreover make simplifying assumptions which paradoxically reduce prediction accuracy.

The corresponding author is Ryan Layer. This research was supported in part by National Science Foundation (NSF) grant numbers CNS-0716635, IIS-0612049, and CNS-0615277.

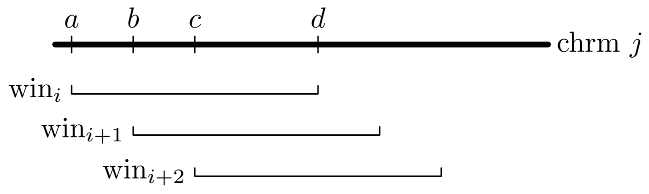


Figure 1. The sliding window algorithm considering chromosome (chrom) j; where the window length is $|d| - |a|$, and the step size is $|b| - |a|$. Each window is offset from the previous window by the same step size.

In this paper we focus on parallelizing the *sliding window* algorithm, a general bioinformatics approach used in a number of genomic analyses [14] [17] [5] [15]. In this scenario, some property (e.g., sequence density) is computed for the portion of the genome within the bounds of a fixed window. As shown in Figure 1, the window slides by a fixed amount across the genome, and the property is recomputed relative to the new window bounds. There are many different applications and variations of the sliding window approach, but they all follow this same general template.

The sliding window technique is a widely used algorithmic primitive. For example, the sliding window approach has been used to improve the spatial resolution of predicted binding sites using ChIP-Seq data [17], as well as to analyze sequence polymorphisms that can aid in understanding evolutionary forces and chromosomal functional significance [15]. In this paper we focus on its use in two structural variation detection techniques described by Shibata et al. [14] and Xie et al. [**?**]. DNA structural variations are anomalies in a genome where portions of chromosomes have either been added, deleted, or otherwise rearranged. Certain variations of this type have been linked to particular types of cancers. The Philadelphia chromosome [7], for example, is such a variation. In normal human cells, the BCR and ABL genes exist in different chromosomes (22 and 9, respectively). However, in many patients with the chronic illness myelogenous leukemia, a structural variation exists where the BCR and ABL genes have fused. This discovery eventually led to a therapy that reduced tumor growth rate [13].

The sliding window algorithm has two main parameters, windows size and step size (i.e., the distance between successive windows). While window size is generally determined by experimental factors (e.g., sequence read length), step size is a tunable parameter and has a direct impact on accuracy and performance. Each window calculates a local statistic, as the step size increases the gap between these statistics increases, which in turn decreases the resolution of any prediction (e.g., inflection points). As the step size decreases, more windows are required to analyze the genome, and the computational complexity becomes correspondingly larger.

This paper discusses our experiences in parallelizing the sliding window approach with the goal of providing researchers the flexibility to select a step size that is biologically meaningful, and not simply computationally feasible. Most recent attempts to parallelize high-throughput algorithms have been focused on algorithms that have large kernels that perform a large amount of computation per thread. In contrast, the sliding window algorithm has a small kernel and performs only a small amount of work per thread, making it a poor candidate for cluster-based parallelization, yet an ideal candidate for parallelization on Single Instruction Multiple Data (SIMD) architectures such as graphics processing units (GPUs) and highly multicore architectures such as the Cell. We therefore selected NVIDIA's Compute Unified Device Architecture (CUDA) and Cell architectures for our implementation and analysis. For comparison, we also included computational results using OpenMP (Open Multi-Processing), a popular application programming interface (API) that supports multi-platform shared memory parallel programming.

Our analysis shows that the speedup under both the CUDA and Cell architectures are inversely proportional to the step side. For example, when the step size is reduced to one, a 156x speedup is realized in CUDA. For the Cell processor, the performance is dependent upon the number of overlapping windows that are combined in DMA requests, the number of threads, and the step size; the optimal speedup is 30x. This result is encouraging given the impact that a smaller step size can have on the granularity and accuracy of the analyses. The speedup under the OpenMP architecture depends only on the number of threads, and is independent of the step size. The OpenMP implementation does not make copies of the windows to perform the analysis, and therefore cannot take advantage of the overlap between windows the way the CUDA and Cell implementations can.

The main contributions of this paper include:

- A parallel sliding window algorithm that improves both performance and accuracy.
- An implementation and performance comparison of CUDA and CELL.
- General strategies for decomposing data requirements to optimizing the performance of the particular architectural aspects of CUDA and CELL.

The remainder of this paper is organized as follows. Section 2 summarizes previous work on parallelizing biological algorithms. Section 3 reviews the sliding window algorithmic approach in bioinformatics. Section 4 describes our OpenMP, CUDA, and Cell -based parallel implementations of sliding windows, and discusses our performance benchmarks. Finally, we conclude in Section 5 with directions for future research.

2. Related Work

Commodity high-performance GPUs have become more accessible and easier to utilize due to the CUDA standard, and programmers have ported a multitude of applications to this unified parallel architecture [3]. In bioinformatics, however, the use of CUDA has been mostly limited to sequence alignment [6] [12] [16].

Similarly, parallelizations on the Cell architecture have been mostly limited to signal processing applications that have few direct memory access (DMA) requests [2]. These applications also tend to use the higher speed "mailbox" mechanism. In contrast, sliding window algorithms tend to issue a large number of DMA requests. Indeed, reducing the number of DMA requests was one of the challenges we faced, as discussed in Section 4.3.2.

In [11], the authors demonstrate that the performance of the phylogenetic likelihood function – a bioinformatics technique that uses sequence alignment to construct an evolutionary history for a group of organisms – can be improved through parallelization on the GPU and Cell/BE. We similarly follow this approach, though we are analyzing a sliding window technique common in bioinformatics.

A common bioinformatics algorithmic template for analyzing genomes is the sliding window [14] [17] [5] [15]. In general, a sliding window approach computes some property (e.g., sequence density) for the portion of the genome that lies within the bounds of a fixed-size window. The window then slides some fixed amount down the genome, and that same property is recomputed with respect to the new window bounds. Some higher-level properties (e.g., max/min, average, histograms, or other characterizing statistics) may then be computed across all individual window values. While there are many different applications and flavors of the sliding window technique, most follow this general form.

This paper discusses our experiences in improving the performance of a sliding window approach used in the structural variation detection technique described by Shibata et al. [14]. In this particular technique, we read in pair-end tag data, annotate where each tag occurs in the genome, and then use a sliding window to determine high density regions for each chromosome, in order to detect structural variations in the input sequence.

3. Sliding Window Analysis

The linear encoding (i.e., "string of symbols") nature of DNA makes the sliding window algorithms a natural template for genomic analysis. Starting from the beginning of each chromosome and proceeding to the end, some statistic is computed over the base pairs within the current bounds of a fixed window. As shown in Figure 1, the window slides down the chromosome by a fixed amount and the statistic is recomputed at each step. Beyond being a natural fit, the sliding window also helps smooth some of the noise inherent in the DNA sequencing process.

Coverage is a statistic that is important to many different types of genomic analyses. As described previously, experiments begin with the preparation of a DNA sample. The sample's coverage refers to distribution of mappable sequences within that sample with respect to a reference genome. In the case of structural variations, increases or decreases in coverage within particular areas of the genome can imply variations. Shibata et al. [14] and Xie et al. [**?**] use different sliding window techniques to calculate such coverage.

The approach proposed by Shibata et al. [14] tracks the coverage of each base pair in the genome, then slides a 200 base pair-wide window using a step size of 50 across each chromosome in order to calculate the average base pair coverage. The integer array $chrm[a][b]$ represents the coverage of base pair b on chromosome a. If sequence i maps to location $[b, c]$ on chromosome a, then each

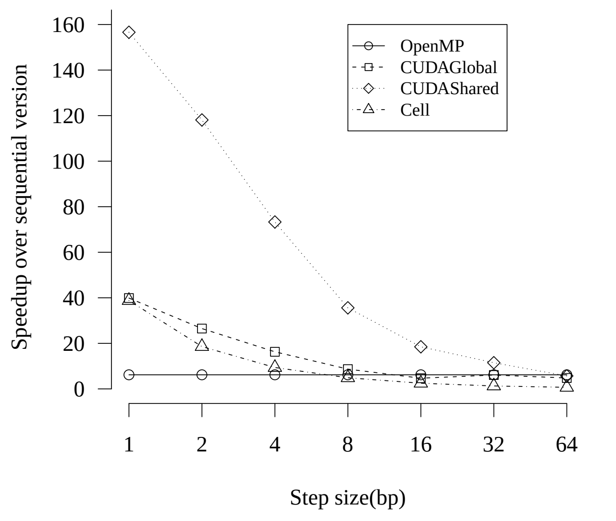


Figure 2. Speedup as a function of step size for OpenMP, CUDA, and Cell. As shown, the Cell implementation yields the greatest speedups for step sizes within Cell's constraints. CUDA, due to its flexibility, can handle smaller step sizes and can yield even greater speedups over the sequential algorithm.

value in the range $chrm[a][b] - chrm[a][c]$ is incremented by one. The value of window j on chromosome k is therefore the sum the values in $chrm[k][j * 50] - chrm[k][j * 50 + 200]$. While this approach allows for an exact accounting of coverage, it is computationally expensive, which led the authors to use a large step size. On the other hand, a smaller step size would give their analysis a finer granularity and accuracy, and possibly enabled the exact determination of variation locations.

Xie et al. [**?**] go further to improve performance by bypassing the base pair coverage step, and directly calculate window coverage. In this approach, the integer array $chrm[a][b]$ represents the coverage of window b, instead of base pair b, on chromosome a. Considering a step size of 50, if sequence i maps to location $[c, e]$ on chromosome a, then the window covering this region $chrm[a][c/50]$ is incremented by one. This approach is considerably more efficient than the Shibata et al. approach, although it sacrifices some generality and accuracy.

For this analysis we chose to parallelize the approach used by Shibata et al. [14]. While the technique used by Xie et al. apparently serves the needs of their specific analysis, we believe it to be less useful to the broader community. For example, accounting for coverage at the base pair level allows for a broader range of statics (e.g., median, mode, etc.) to be calculated.

4. Parallel Sliding Window

In the Shibata et al. [14] implementation, the sliding window size is 200 base pairs (bp) with a step size of 50 bp. Since the human genome contains over 3 billion bp, the algorithm must perform nearly 150 million sequential windowing operations. During each operation, the algorithm counts the number of tags within a region of the genome. Each window operation does not depend on the output of any other window operations, and there are no data dependencies between window operations. Therefore, the algorithm is a "high throughput" problem.

Performing a single operation independently on many different inputs is a classic parallelization scenario. However, the memory bound nature of the problem and the small kernel make it not "embarrassingly parallel". The size of the data set is beyond the memory capacity of most systems, and beyond the local storage of some architectures. Thus, threads may be forced to stall until fresh data can be loaded.

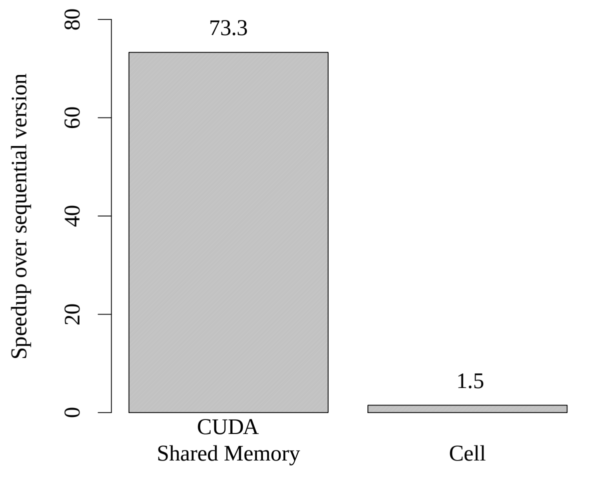
We implemented the sliding window stage using OpenMP for the multicore CPU, CUDA for the GPU, and libspe2 for the Cell. The OpenMP results were achieved by adding in one simple pragma statement. The CUDA and Cell results were more complicated, consisting of a simple translation of the code to their respective languages, with optimizations applied afterward. Speedups achieved by the OpenMP, CUDA and Cell implementations with varying step size are shown in Figure 2.

In our comparisons, we provide the OpenMP benchmarks as a reference when considering the performance of CUDA and Cell. To compare the performance between CUDA and Cell, we implemented the sliding window algorithm with similar *optimization effort*, instead of similar coding techniques. We recognize that quantifying programming effort is difficult, and we make no attempt to do so here. Instead, we followed similar strategies on both architectures; first we implemented a straight forward parallel algorithm, then we implemented the most obvious architectural optimization. While the code running on each architecture is significantly different, we believe this comparison is more insightful because it reflects real design choices that must be made when developing on these architectures and demonstrates the performance impact of those choices.

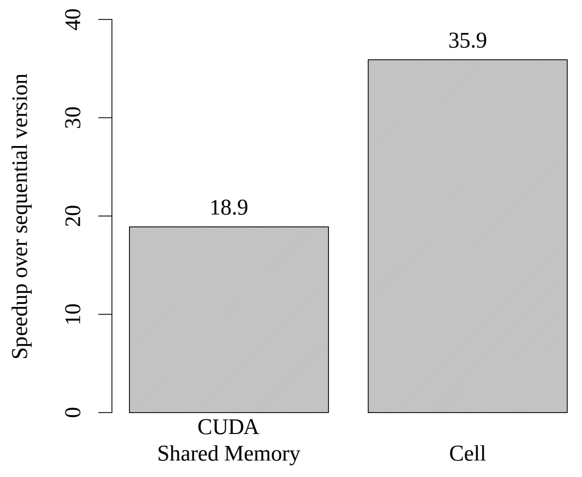
Since CUDA and Cell maximize performance in different ways, running similar code on both architectures would under-utilize at least one of the architectures and result in an artificial comparison. We demonstrate this point in Figure 3, where similar strategies are executed on both architectures. In Figure 3(a), each thread computes one window. Under this strategy, CUDA performs best because it contains a high-bandwidth shared memory space that can be concurrently accessed by threads within a thread block. Cell, on the other hand, uses a relatively low-bandwidth Direct Memory Access (DMA) channel to share data between threads. When the step size is small, there is a significant overlap between windows. CUDA shared memory allows overlapping windows to be computed in parallel, while Cell must waste cycles retransmitting data to threads. In Figure 3(b), each thread computes 120 windows. This improves performance in Cell because it reduces redundant DMA calls, and degrades CUDA performance because 120 windows that could be computed in parallel must be computed serially.

As Figure 2 shows, the CUDA and Cell architectures significantly outperform OpenMP. This is likely due to the memory-bound nature of the problem: the innermost loop has a small body and traverses a large amount of data in RAM, with a significant amount of overlap in those requests. Such algorithms are better suited to nonuniform memory access (NUMA) architectures. The Cell processor has some constraints on memory alignment; although these constraints were addressed to some degree by padding, it was still necessary to use powers of 2 for the step size, unlike in CUDA or OpenMP, where step sizes of powers of 2 were not strictly required. This suggests a trade-off between flexibility and performance that is dependent on the parameters of the sliding window, a factor that should be considered when acquiring and configuring computer systems.

Runtimes for the OpenMP implementation were measured on a 3.16 GHz, 8-way Intel Xeon CPU X5460 running Fedora 9. Runtimes for CUDA were measured on a 2.67 GHz, 4-way Intel Xeon



(a) Step size = 4; Windows per thread = 1



(b) Step size = 1; Windows per thread = 120

Figure 3. Comparison of CUDA and Cell implementations where similar strategies are executed on both architectures. 3(a): CUDA performs well with fewer windows per thread, while Cell performs poorly in this setup. 3(b): Cell performs well with more windows per thread, compared to CUDA.

CPU X5550 with a Tesla C1060 GPU and NVIDIA driver version 185.18.14. Runtimes for the Cell implementation were measured on a system with four 3.2 GHz Cell processors running Fedora 12, and a Playstation 3. The C code was compiled using gcc version 4.3.0 for the OpenMP implementation, and gcc version 4.2.4 for the CUDA implementation. The CUDA code was compiled using NVCC version 0.2.1221 and CUDA toolkit release 2.2. The Cell code was compiled using spu-gcc version 4.1.1 and gcc 4.3.0.

4.1 OpenMP

A popular option for parallelizing applications is the OpenMP standard. Converting a sequential program into a parallel program can be as simple as adding a single compiler directive, as was the case for the sliding window algorithm. We report performance benchmarks using OpenMP in Figure 4 as a comparison reference when considering alternate parallel architectures. While it did not provide the most speedup, OpenMP was the easiest to implement, with only a single pragma statement needed to be added around the iteration loop that creates windows.

4.2 CUDA

NVIDIA's Compute Unified Device Architecture (CUDA) is a SIMD architecture that provides programmers a general interface to a large number of parallel graphics processing units (GPUs) [8]. CUDA attempts to maximize resource utilization by handling large numbers of threads. Ideally, if a large enough set of threads is generated, a sizable subset of them will always be ready to execute. When the current threads are forced to stall (e.g., due to memory accesses), other threads can swap in and hide most of the latency.

Understanding how the architecture and programming constructs relate is an important part in achieving this improved resource utilization. Each CUDA processor handles a single thread. Processors and threads are grouped into multiprocessors and thread blocks, respectively. A multiprocessor executes a number of thread

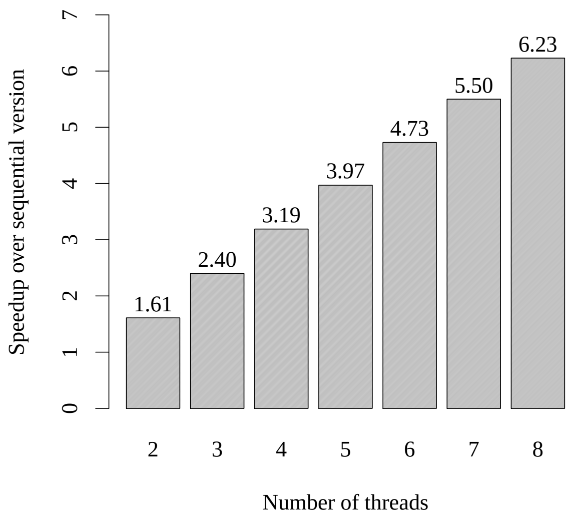


Figure 4. Speedup for the OpenMP approach over the sequential approach where the step size is equal to 32. The greatest speedup of 6.23x over the sequential version is achieved when employing 8 threads.

blocks and each thread block executes on exactly one multiprocessor. Every thread in a thread block executes the same instruction on different data in each clock cycle. Thread blocks are further subdivided into smaller groups called warps. The scheduler can easily rotate execution between warps that are stalled and warps that are ready to execute.

This scheme depends on the programmer's ability to design an algorithm that can create enough threads. In the University of Illinois CUDA lecture series [9], it was suggested that, in order to fully utilize the hardware, CUDA programs should have 32 thread blocks and each thread block should have 192 threads. For this analysis, the thread count is equal to the window count, which is in turn equal to the chromosome size divided by the step size. Since window size and chromosome size can be assumed to be fixed, the step size is the critical factor in performance. In our naive approach, we see that simply creating a large number of threads can greatly improve the performance, and that performance decreases at certain step sizes where thread blocks contain less than 192 threads.

Shared memory is a major source of optimization. Each multiprocessor has a small amount of on-chip, high-bandwidth memory (about 16K) that can be accessed only by thread blocks executing on that multiprocessor. The access time of shared memory is orders of magnitude less than global memory access, and as we demonstrate, can yield significant speedups. However these speedups are not free. Since this space can only be accessed by threads on the multiprocessor, and not the host thread, threads are responsible for loading the data from global memory into shared memory, and writing the results back to global memory. Considering the fact that the size of shared memory is limited, and that threads cannot access the shared memory of different blocks, large problems must be decomposed into small chunks. It is up to the programmer to strike a balance between the overhead associated with this decomposition and the speedup gained from using shared memory.

Figure 5 demonstrates the close relationship that can exist between shared memory and thread count. In the sliding window, when the step size is eight or greater the hardware is underutilized and performance is relatively low. A larger step size results in fewer windows and less window overlap, which in turn results in fewer threads per block and less shared data between threads. Step sizes of four and less fully utilize the hardware in both thread count and shared memory. However, simply increasing the number of threads does not necessarily translate into better performance. The peaks in step sizes of four, two, and one are the points where the shared memory is fully utilized with the minimum number of threads.

When using CUDA, it is important to consider that the GPUs exist on a card that is physically separate from the CPU. To execute code on the GPU all data must be shipped from CPU memory to GPU memory where the data is processed. Results must then be shipped back to the CPU on the same relatively slow channel. Specifically, the programmer must: allocate space on the GPU, copy data from the CPU to the GPU, process the data on the GPU, copy the data from the GPU to the CPU, and free the space previously allocated on the GPU. All of these steps take a non-trivial amount of time and must be considered when calculating any speedup. To make this point clear, we have divided the CUDA runtime analysis into these stages in Figure 6.

4.2.1 Naive Approach–Global Memory

Every thread in CUDA has full access to global memory. Therefore, the most straightforward method to parallelize the sliding window using CUDA is to load a coverage array and a result array into global memory and spawn enough threads such that each window is handled by a unique thread. Since each window operation is independent, threads can safely read from the coverage array and write to the result array without synchronization. While global memory

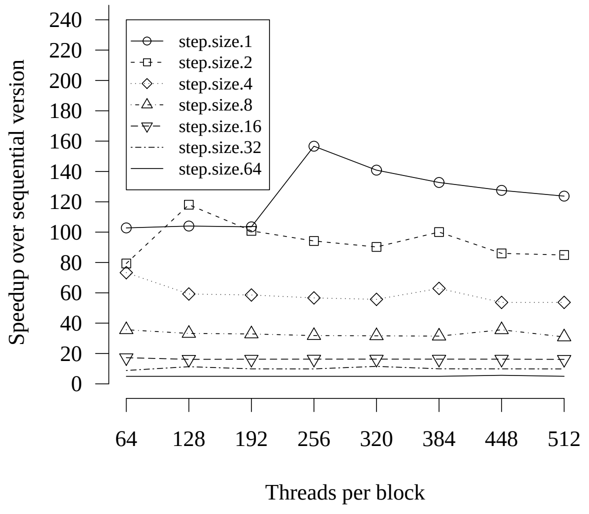


Figure 5. Speedup as a function of maximum threads per block for the CUDA shared memory approach using different step sizes. As shown, peak performance for each step size occurs when the number of threads equals, but does not exceed, the point where shared memory is fully utilized.

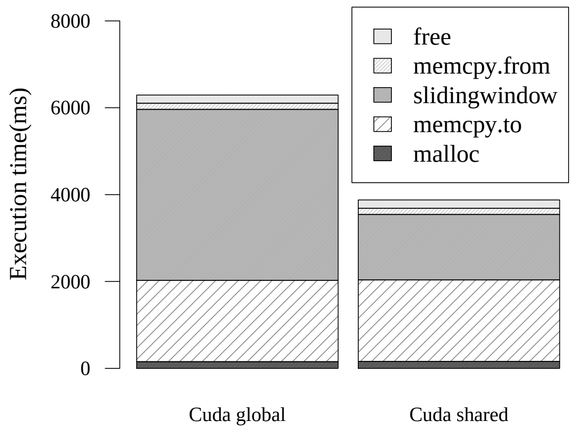


Figure 6. Breakdown of overhead and analysis runtime for CUDA global memory and shared memory approaches, demonstrating that the time to ship data from the CPU to the GPU and back is non-trivial.

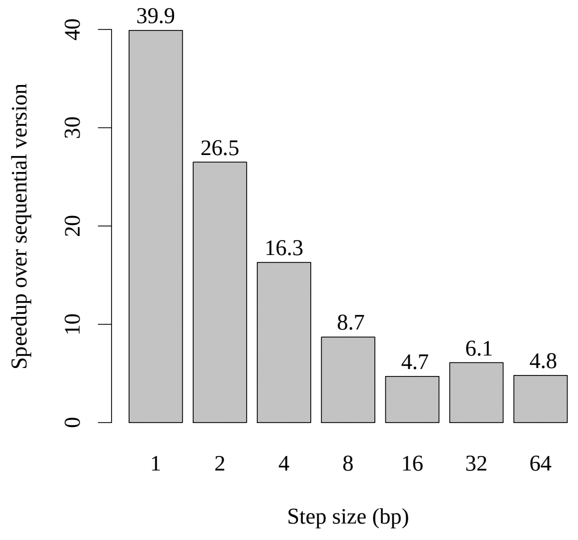


Figure 7. Speedup for the CUDA global memory approach over the sequential approach. The greatest speedup of 39.9x is found using a step size of 1. As the step size increases, the speedup decreases due to the decrease of the number of threads in a thread block.

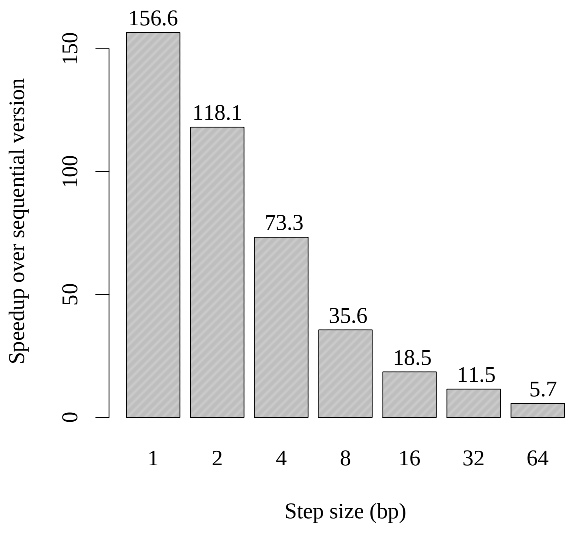


Figure 8. Speedup for the CUDA shared memory approach over the sequential approach. As the step size increases, speedup decreases because of a decrease in threads in a thread block and, therefore, a corresponding decrease in shared data between threads. The greatest speedup of 213.2 times the sequential algorithm is achieved with a step size of 1.

requests are high latency (400–600 cycles) and non-caching, and each thread makes over 200 memory requests, this approach has a non-trivial speedup over the sequential version as shown in Figure 7.

The achieved speedup, despite obvious inefficiencies, is a result of having an abundance of threads. When the executing threads read or write to global memory they must stall for hundreds of cycles. Instead of leaving the hardware idle during those cycles, the scheduler rotates in a different batch of threads that is ready to execute. Once the reads or writes have been completed, the stalled threads will be ready to execute and can be rotated back in. If the thread count is high, then it is more likely that some group of threads will always be ready to execute. We can see this phenomenon in Figure 7. Larger step sizes perform poorly because the thread count is low, while smaller step sizes perform much better due to the high thread count.

4.2.2 Shared Memory Optimization

An obvious optimization to our naive approach is to use shared memory. Given a window size of 200 and a step size of 16, one element in the coverage array is shared across 12 different windows (threads). Since global memory is non-caching, our naive approach incurs the $400 - 600$ cycle latency on all 12 accesses. If we use the low-latency (4 cycle) shared memory space, then we only incur the global memory latency once, when the value is loaded from global memory into shared memory.

There are several complexities when using shared memory related to its size and limited accessibility. The shared memory space is divided into 16K segments, and threads can only access their block's shared memory space. In our case the amount of data being analyzed is much larger than 16K, so the data must be divided into independent chunks where each chunk is handled by one thread block. Given the shared memory space access restrictions, each chunk of data must be loaded by the threads as a block from global memory to shared memory before it can be analyzed. A barrier synchronization is required after this step. Once the chunk is loaded, each window in that chunk is handled by a thread. Since the sliding window is continuous and uniform across the genome, several windows will span chunks. To ensure that every window is calculated, we must overlap these chunks.

The speedup when using shared memory (Figure 8) is significant, especially when the parameters allow for many threads in a thread block. The number of windows needed to analyze the chunk depends on the step size. Smaller step size means more windows and more threads, leading to a higher utilization of the hardware. Smaller step sizes also increase the amount of data shared between threads, which in turn increases the impact of shared memory.

While shared memory is faster than global memory, using it is not as straightforward. As previously discussed, the size of shared memory requires us to analyze the data in chucks. In many applications, including ours, there are boundary issues, and blocks must be overlapped to ensure that no data is skipped. Special care must then be taken to prevent race conditions between threads in overlapped regions.

Furthermore, each shared memory segment can only be accessed by the threads in one block, and host code cannot directly access shared memory. Threads within a block are responsible for transferring data from global memory to shared memory. In our analysis each thread loads the first s elements of their window, where s is the step size. In this approach, the last $windowsize - stepsize$ elements of each array will not be loaded, and therefore the next block must overlap by that amount to ensure that the corresponding portion of the chromosome is still analyzed.

Once the data is loaded into shared memory, execution proceeds as normal. Each thread finds the sum of the elements in its range, and the result is written to global memory. In Figure 8 we can see the performance impact of blocks without enough threads. When the step size is 32, the number of windows needed to analyze the portion of the array loaded into global memory is only 110. Therefore each thread block contains only 110 threads, which is fewer than the suggested 192 threads. A smaller step size also results in more shared values, which also contributes to the increased speedup for the shared memory approach.

4.3 Cell

The STI Cell architecture is a heterogeneous multicore architecture that features a PowerPC core and 8 simplified Synergistic Processing Elements (*SPEs*). This architecture is designed for very high throughput, particularly for floating point operations. It seeks to avoid the memory bandwidth issues that are typical of shared memory architectures by giving each SPE a special *local storage*, which is as fast as a cache but is treated as RAM by the program. An SPE may also access the system RAM through a special direct memory access (DMA) functionality that copies ranges from RAM to the local storage.

DMA functions in the Cell are performed over a relatively low bandwidth shared channel, and thus constitute a major bottleneck. Effective Cell programming requires a minimization of the number of DMA calls that are made. Ideally, each SPE should load all the data upon which it will operate into its local storage once, operate on that data, and send only the final results back to RAM. A significant loss in performance can be expected when the data size exceeds the local storage capacity, because the data must be loaded and returned multiple times. An implementation quirk of the Cell requires all DMA requests to be 16 byte aligned, and *padding* is necessary for data which cannot be aligned to such a boundary, resulting in further losses in efficiency.

In addition to using DMA, it is possible for the individual SPEs to communicate using message passing. Each SPE features a high speed, hardware-assisted message passing mechanism referred to as a *mailbox*. Mailboxes are queues of 32 bit messages that SPEs can send to each other and to the PowerPC core. The mailbox mechanism is very fast, and libspe provides both blocking and non-blocking mailbox functions.

Cell processors range from lower end Sony Playstation 3 consoles, to high end IBM blade serve systems. The Playstation 3 Cell processor makes 6 SPEs available, and has lower performance characteristics than a server system. The higher end IBM Cell systems feature multiple Cell processors per motherboard, higher memory access speeds, and a larger amount of memory. There is a measurable difference in performance between the Playstation 3 and the Cell servers, which appears to be primarily related to the different memory bandwidths of the systems. Midway through our experiments, our IBM blades broke down, and we switched to a Sony Playstation 3. Although the speedup on this system follows the same trend as the blades, the results quantitatively different, likely due to the lower performance memory. Additionally, the PS3 features fewer SPEs, and thus reduced parallelism. A comparison of the results on the two systems is found in Table 1, where a very significant difference in performance is plainly evident.

When available, utilizing multiple Cell processors requires additional programming effort, especially in memory bound problems such as the sliding window algorithm. Many multiprocessor Cell systems, including one of the test systems, are based on a nonuniform memory access (NUMA) architecture, with some memory banks being accessed faster than others depending on which CPU is performing the access. Making effective use of more than one Cell processor thus requires memory to be allocated in such a way that requests for slower memory banks are minimized. In this experiment, no such effort was made, and so utilizing more than 8 SPEs results in degraded performance; this can be seen in Figure 9, with the performance beginning to drop off for 9 and 10 SPEs. For the full complement of 32 SPEs, we found that performance was no better than the sequential algorithm.

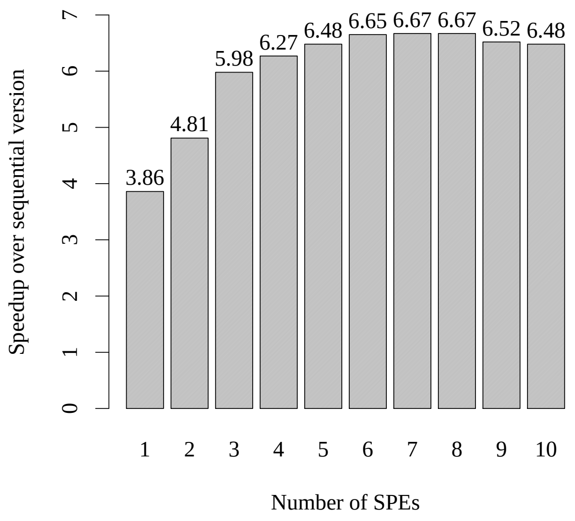


Figure 9. Speedup for Cell as the number of SPEs in use is increased; note that a maximum speedup of 6.67x is reached when 8 SPEs are in use. These results were collected using the blades system.

Step size	Blade	Playstation 3
16	30.1	2.5
32	13.9	1.3
64	6.67	0.6

Table 1. Speedup results on the blades server vs. the Playstation 3, for step sizes of 16, 32, and 64, with a chunking factor of 60. Note that the blade servers perform significantly better than the Playstation 3, due to the faster memory on those systems.

4.3.1 Naive Approach: One Window per SPE

The simplest method of programming the Cell processor for a high throughput problem is to load the data for each thread into the local store, and then have the thread execute. In the case of the sliding window algorithm, this means loading a single window into the local store, operating on it, and returning the result using a mailbox. Although this method is simple to understand and implement, it is suboptimal. In testing, it was found that even with all the SPEs in use, the execution time was close to the sequential runtime.

4.3.2 Optimizing Cell Performance

The fact that so many windows overlap with each other can be used to reduce the number of DMA calls. Although more data is transferred per DMA request, the total number of requests will be reduced by a greater factor than the increase in data transfer per request, and thus an overall improvement can be expected. This was found to be the case, as shown in Figure 10. This improvement becomes increasingly pronounced as the step size parameter

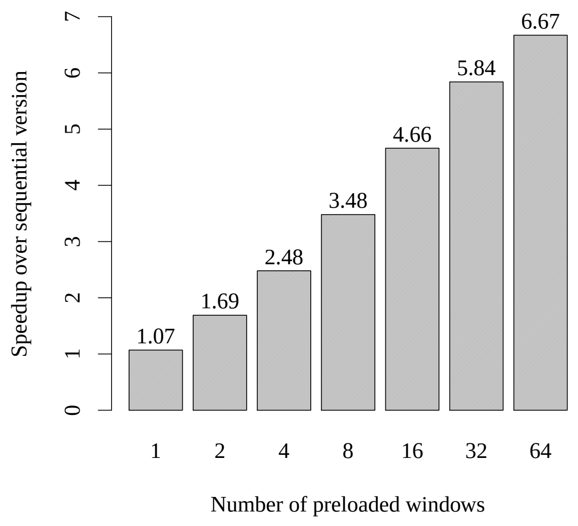


Figure 10. Speedup for Cell as overlapping windows are combined in DMA requests; a step size of 64 was used for this test, thus allowing a maximum number of combined windows of 60. As the number of preloaded windows increases, the speedup increases as well, with a maximum of speedup of 6.67x as compared to the sequential algorithm. These results were collected using the blades system.

Step Size	Optimal Chunking	Maximum Chunking
1	480	3840
2	480	1920
4	240	960
8	240	480
16	240	240
32	120	120
64	60	60

Table 2. The optimal chunking amount for each step size. This is the number of windows that will be loaded for each DMA transfer, and thus determines the amount of work that each SPE will perform between transfers. This amount decreases, because the amount of data transferred increases with the step size. For the largest step sizes, the chunking factor becomes limited by the amount of memory in the local store.

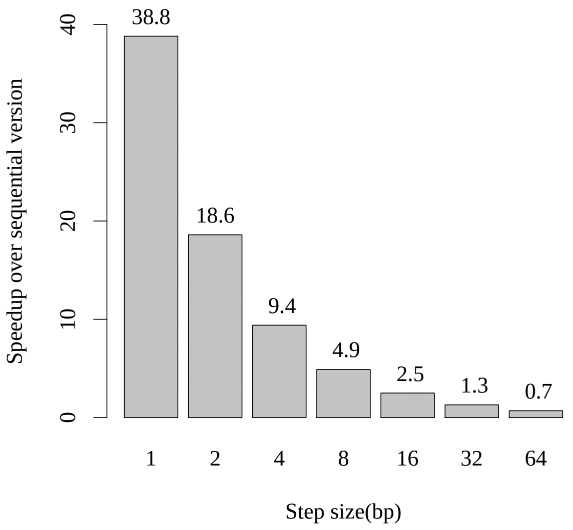


Figure 11. Speedup for Cell versus step size, with some padding added for step sizes that do not meet the alignment constraints. This graph reflects the results for the optimal chunking factor for each step size. As with CUDA, as step size decreases, speedup increases, the maximum speedup of 30.1x was observed on the blades for a step size of 16 (the smallest size tested on the blades; see Table 1), and a speedup of 38.8x was observed for a step size of 1 on the Playstation 3.

decreases, as shown in Figure 11. This result is similar to the result observed for CUDA.

However, such improvements are not reaped for free. By combining requests like this, load balancing becomes worse because of a reduction in the granularity of the subproblems, and in the extreme case the maximum number of useful parallel threads becomes unreasonably limited. This effect is mitigated somewhat by the small size of the local store, but on a system with multiple Cell processors, it is possible that some SPEs will be unused.

The SPE local store is of size 256K, but some of this memory is used to store the executable code. For a window size of 256 and a step size of 64, this only leaves room for a maximum of 60 overlapping windows which can be loaded into the local store. However, the maximum number of chunked windows is not necessarily optimal, as shown in Table 2. For smaller step sizes, the optimal chunking amount is less than the maximum chunking amount. In Figure 10, as the number of preloaded windows increases, a clear trend of continuous improvement is observed.

5. Conclusion and Future Work

Our analysis shows that the step size is a critical performance parameter in the sliding window algorithm. In [14], a step size of 50 base pairs is used. While a smaller step size would increase the granularity of results, it was deemed computationally infeasible at the time. We have demonstrated that smaller step sizes are feasible in parallel architectures and that a smaller step size actually improves performance on these parallel architectures. Researchers now have the flexibility to select a step size with biological significance in mind, rather than being limited only by what is computationally feasible.

While both parallel architectures resulted in a significant performance increase over the sequential version of the sliding window algorithm, differences between the CUDA and Cell architectures have a dramatic impact on the programming strategies. As shown in Figure 3, an implementation optimized for CUDA runs poorly on CELL, and vice versa. This demonstrates that while general parallel algorithms can be a guide for programmers, major optimizations require a deep understanding and utilization of particular architectural features. Furthermore, without good abstractions the portability of parallel code will remain questionable.

For example, the amount of work that each thread should be asked to perform is drastically different between CUDA and CELL. CUDA performance depend on the programmer's ability to create a multitude of threads. The CUDA scheduler masks memory latency by rapidly cycling threads in and out of execution. If enough threads are spawned, then chances are good that a thread will be ready to execute while another stalls. CELL, on the other hand performs best when each of a fewer number of threads computes as much as possible. Each SPE has local storage that is a fast as a cache, but the channel between SPEs is relatively low bandwidth. Therefore the ideal scenario for the CELL is where each SPE can load all the data upon which it will operate into its local storage once, operate on that data, and send only the final results back to RAM.

It may be possible to further optimize our CUDA implementation by addressing bank conflicts. In CUDA, shared memory is divided into banks which can be accessed simultaneously [8]. However, simultaneous accesses to the same bank are serialized. Since each memory element in our coverage array is accessed by a number of threads, our implementation could incur some bank conflict. However, we expect that any speedup resulting from this optimization would be minor when compared to the speedup we observed when moving from global memory to shared memory. Given a window size of 200, and a step size of one, the maximum number of simultaneous accesses to one memory location would be 200. Even if those 200 shared memory accesses occur simultaneously, and all 200 access are serialized, the time required to complete the access is on the order of one global memory access.

The Cell implementation may be further improved by developing a scheme to take better advantage of multiple Cell processors, for systems with such hardware. The drop-off in performance when employing more than 8 SPEs may be a result of how memory is allocated and accessed in the current implementation, which is not ideal for systems with more than one Cell processor [1]. Using the libnuma library for allocation and access, and possibly allocating multiple copies of the data set, may allow more SPEs to be used in an efficient manner.

More generally, it would be interesting to test our methodology on other sliding window algorithms, and even on different bioinformatics algorithms altogether, such as multiple sequence alignment, and the highly parallelizable phylogenetic tree reconstruction as described in [10]. The highly parallel nature of DNA chemistry, combined with the availability of thousands of independent genomes, strongly suggests that bioinformatics is a very ripe area for the confluence of hardware and software based parallel techniques.

References

[1] A. Arevalo, R. M. Matinata, M. Pandian, E. Peri, K. Ruby, F. Thomas, and C. Almond. *Programming the Cell Broadband Engine: Examples and Best Practices*. IBM Redbooks, 2008.

[2] C. Cantini, E. L. Rosa, A. L. Re, and A. D. Lallo. Passive coherent locator signal processor on IBM Cell Broadband Engine (Cell BE). *IEEE Radar Conference*, pages 1–6, May 2009.

[3] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing with CUDA. *IEEE Micro*, 28(4):13–27, 2008.

[4] A. S. Grimshaw, E. A. West, and W. R. Pearson. No pain and gain! - experiences with mentat on a biological application. *Concurrency: Practice and Experience*, 5(4):309–328, June 1993.

[5] P. Librado and J. Rozas. DnaSP v5: a software for comprehensive analysis of DNA polymorphism data. *Bioinformatics*, 25(11):1451–1452, 2009.

[6] S. A. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9:S10–S18, March 2008.

[7] P. C. Nowell and D. A. Hungerford. A minute chromosome in human chronic granulocytic leukemia. *Science*, 142(1497):290, 1960.

[8] NVIDIA. CUDA compute unified device architecture, v 1.0. *NVIDIA*, 2007.

[9] S. J. Patel. University of Illinois at Urbana-Champaign, ECE 498 AL : Applied Parallel Programming. URL: http://courses.ece.illinois.edu/ece498/al/, Spring 2010.

[10] W. R. Pearson, G. Robins, and T. Zhang. Generalized neighbor-joining: More reliable phylogenetic tree reconstruction. *Journal of Molecular Biology and Evolution*, 16(6):806–816, 1999.

[11] F. Pratas, P. Trancoso, A. Stamatakis, and L. Sousa. Fine-grain Parallelism Using Multi-core, Cell/BE, and GPU Systems: Accelerating the Phylogenetic Likelihood Function. *2009 International Conference on Parallel Processing*, pages 9–17, September 2009.

[12] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474–483, December 2007.

[13] T. Schindler, W. Bornmann, P. Pellicena, and W. T. Miller. A structural mechanism for STI-571 inhibition of abelson tyrosine kinase. *Science*, 289(5486):1857–1959, September 2000.

[14] Y. Shibata, A. Malhortra, S. Bekiranov, and A. Dutta. Yeast genome analysis identifies chromosomal translocation, gene conversion events, and several sites of ty element insertion. *Nucleic Acids Research*, 37 (19):6454–6465, August 2009.

[15] H. Stephan, V. Albert, and R. Julio. Genome-wide DNA polymorphism analyses using VariScan. *BMC Bioinformatics*, 7(209), 2006.

[16] G. M. Striemer and A. Akoglu. Sequence alignment with GPU: Performance and design challenges. *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, May 2009.

[17] Y. Zhang, T. Liu, C. A. Meyer, J. Eeckhoute, D. S. Johnson, B. E. Bernstein, C. Nussbaum, R. M. Myers, M. Brown, W. Li, and X. S. Liu. Model-based analysis of ChIP-Seq (MACS). *Genome Biology*, 9 (R137), September 2008.