International Conference on Parallel and Distributed Systems (ICPADS'09); Shenzhen; China; Dec; 2009.

On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit

Shucai Xiao*, Ashwin M. Aji†, and Wu-chun Feng*†
*Department of Electrical and Computer Engineering
†Department of Computer Science
Virginia Tech
{shucai, aaji, wfeng}@vt.edu

Abstract

Graphics processing units (GPUs) have been widely used to accelerate algorithms that exhibit massive data parallelism or task parallelism. When such parallelism is not inherent in an algorithm, computational scientists resort to simply replicating the algorithm on every multiprocessor of a NVIDIA GPU, for example, to create such parallelism, resulting in embarrassingly parallel ensemble runs that deliver significant aggregate speed-up. However, the fundamental issue with such ensemble runs is that the problem size to achieve this speed-up is limited to the available shared memory and cache of a GPU multiprocessor.

An example of the above is dynamic programming (DP), one of the Berkeley 13 dwarfs. All known DP implementations to date use the coarse-grained approach of embarrassingly parallel ensemble runs because a fine-grained parallelization on the GPU would require extensive communication between the multiprocessors of a GPU, which could easily cripple performance as communication between multiprocessors is not natively supported in a GPU.

Consequently, we address the above by proposing a fine-grained parallelization of a single instance of the DP algorithm that is mapped to the GPU. Our parallelization incorporates a set of techniques aimed to substantially improve GPU performance: matrix re-alignment, coalesced memory access, tiling, and GPU (rather than CPU) synchronization. The specific DP algorithm that we parallelize is called Smith-Waterman (SWat), which is an optimal local-sequence alignment algorithm. We then use this SWat algorithm as a baseline to compare our GPU implementation, i.e., CUDA-SWat, to our implementation on the Cell Broadband Engine, i.e., Cell-SWat.

1 Introduction

Today, gains in computational horsepower are no longer driven by clock speeds. Instead, the gains are increasingly achieved through parallelism, both in traditional x86 multi-core architectures as well as the many-core architectures of the graphics processing unit (GPU). Amongst the most prominent many-core architectures are the GPUs from NVIDIA and AMD/ATI, which can support general-purpose computation on the GPU (GPGPU). Thus, GPUs have evolved from their traditional roots of graphics pipeline model into programmable devices that are suited for accelerating scientific applications such as sequence matching and fast N-body simulation [11–13, 15].

In general, only data- or task-parallel applications, which have little to no inter-multiprocessor communication, map well to the the many-core GPU architecture [2]. This is mainly due to the lack of explicit hardware or software support for inter-thread communication between different multiprocessors across the entire GPU chip. The current (implicit) synchronization strategy for the NVIDIA Compute Unified Device Architecture (CUDA) platform [10] synchronizes via the host CPU and then re-launches a new kernel from the CPU to GPU, which is a costly operation.

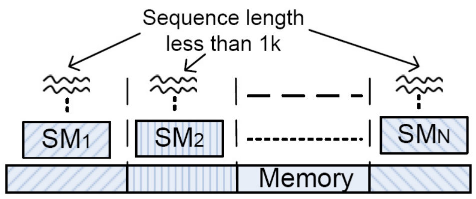
In addition, another accelerator-based parallel computing platform that we have studied, the Cell Broadband Engine (Cell/BE) [3], is also used for general-purpose computation. The Cell/BE is a heterogeneous processor, where direct data communication is supported among the Power Processing Element (PPE) and the Synergistic Processing Elements (SPEs) via the Element Interconnect Bus (EIB).

Dynamic programming (DP) is one of the Berkeley 13 dwarfs, where a dwarf is defined as an algorithmic method that captures a pattern of computation and communication.[1] Depending on the number of recursive terms and the data dependency across subproblems, dynamic programming can be divided into four classes — *serial monadic* class, *serial polyadic* class, *nonserial monadic* class, and *nonserial polyadic* class [4]. In this paper, we investigate the mapping of the Smith-Waterman (SWat) local sequence-alignment algorithm [14] — an example of the dynamic programming algorithm — onto the GPU. With the *affine gap* penalty that is used, SWat belongs to the *nonserial polyadic* class, which is the *most complex* of the four dynamic programming classes.

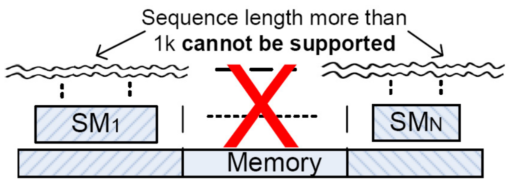
Specifically, we focus on a *fine-grained parallelization* of the SWat algorithm, in which a single problem instance is processed across all the multiprocessors of the GPU, thus *resulting in a more robust mapping of DP to the GPU* where virtually all sequence sizes within NIH's GenBank can be processed. This is in stark contrast to previous *coarse-grained parallelizations* [7, 8], where multiple problem instances are simply replicated onto each multiprocessor of a NVIDIA GPU, thus creating an "embarrassingly parallel ensemble run" that severely restricts the problem size that can be solved, as shown conceptually in Fig-

[1]The dwarfs represent different algorithmic equivalence classes, where membership in an equivalence class is defined by similarity in computation and communication.

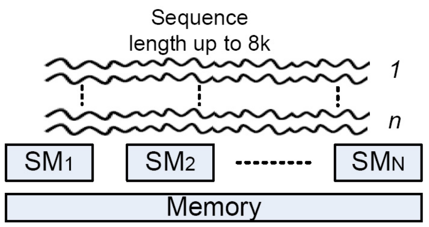
ure 1(a). When the sequence sizes exceed 1024 in length, i.e., a significant portion of the NIH GenBank database, the sequences *cannot* be processed by this coarse-grained parallelization, as shown in Figure 1(b). Our fine-grained parallelization, however, can process all sequence sizes up to 8461 in length, as shown conceptually in Figure 1(c). We then use global memory to store the alignment information due to the limited size of the shared memory and cache.



(a) Coarse-grained parallelization for small sequences (SM: Streaming Multiprocessor) [7,8,16].



(b) Coarse-grained parallelization for large sequences (SM: Streaming Multiprocessor) [7,8,16].



(c) Our fine-grained parallelization for arbitrarily sized sequences (SM: Streaming Multiprocessor).

Figure 1. Coarse-Grained vs. Fine-Grained Parallelization.

Due to the long latency-access times to the global memory of the GPU, coupled with SWat being a memory-bound application [6], we propose three techniques to more effectively access the global memory of the GPU: *matrix re-alignment*, *coalesced memory access*, and *tiling*. In addition, to achieve a fine-grained parallelization of DP on the GPU, data will need to be shared across different blocks, and hence multiprocessors, of the GPU. Thus, global synchronization across the GPU will be needed. As such, we propose a *GPU synchronization* method that does *not* involve the host CPU, thus eliminating the overhead between switching back and forth between the GPU and CPU. As mentioned above, SWat with the affine gap penalty belongs to the most complex dynamic programming class. Thus, although the above techniques are proposed for the SWat algorithm, they can be applied to any of the other classes of dynamic programming-based algorithms.

The overarching contribution of the paper is the robust mapping of our fine-grained SWat parallelization onto the GPU, thus enabling it to support the alignment of virtually all the sequences in the NIH GenBank. To achieve the above, we propose the following: (1) techniques for accelerating access to global memory: matrix re-alignment, coalesced memory access, and tiling; (2) a GPU synchronization method to improve global synchronization time, and hence, communication between multiprocessors; and (3) a performance comparison relative to the Cell/BE, another type of accelerator-based parallel platform.

The rest of this paper is organized as follows: Section 2 presents the related work. Section 3 describes the NVIDIA GTX 280 architecture and the CUDA programming model. Section 4 introduces the sequential SWat algorithm. Techniques to accelerate SWat in CUDA are described in Section 5. Section 6 compares and analyzes the performance of the various implementations on the GPU and across the GPU and the Cell/BE. Section 7 concludes the paper.

2 Related Work

The Smith-Waterman (SWat) algorithm has previously been implemented on the GPU by using graphics primitives [5, 6], and more recently, using CUDA [7, 8, 16]. Though the most recent CUDA implementations of SWat [7, 8] report speed-ups as high as 30-fold, they all suffer from a myriad of limitations. First, each of their approaches only follows a coarse-grained, embarrassingly parallel approach that assigns a single problem instance to *each* thread on the device, thereby sharing the available GPU resources among multiple concurrent problem instances, as shown in Figure 1(a). This approach *severely* restricts the maximum problem size that can be solved by the GPU to sequences of length 1024 or less. In contrast, we propose and implement a fine-grained parallelization of SWat by distributing the task of processing a single problem instance *across all the threads* on the GPU, thereby supporting realistic problem size, as large as 8461 in lengths. The above limitation is due to the physical size of global memory on the GPU. Though the global memory can support the alignment of a large sequences, the coarse-grained parallel approach forces the global memory to be shared amongst multiple instances of SWat whereas our fine-grained parallel approach leaves the entire global memory resource available to a single instance of SWat, allowing larger sequences to be processed. In [16], Striemer et al. also primarily use shared memory and constant cache for coarse-grained SWat parallelism on the GPU. Thus, their implementation is also limited to query sequences of length 1024 or less because of the limited shared memory and cache size. Finally, Tan et al. [17] parallelize the *nonserial polyadic* dynamic programming to a multi-core architecture, where they propose a parallel pipelined algorithm for filling the dynamic programming matrix by decomposing the computation operators. This technique tolerates the memory access latency using multi-thread and it is easily improved with tile technique.

With respect to GPU synchronization, the most closely related work to ours is that of Volkov et al. [18], who have also "implemented" a global software synchronization method to accelerate dense linear-algebra constructs. Our GPU synchronization differs from Volkov's in the following ways: (1) they have not actually integrated their GPU synchronization with any of the linear algebra routines. (2) they have acknowledged a severe drawback in their synchronization method, i.e., their global synchronization does *not* guarantee that previous accesses to all levels of the memory hierarchy have completed unless a memory consistency model is assumed. With respect to the first point, Volkov et al. only provide *theoretical estimates* of the performance that could possibly be obtained *if* their implementation was used in conjunction with their synchronization code. In contrast, we have implemented our GPU synchronization, integrated it in our CUDA-SWat implementation, and shown actual experimental results. For the second point, our implementation *guarantees* that transactions at all levels of the memory hierarchy have been completed at the end of the synchronization code. This is due to the introduction of the `__threadfence()` function in CUDA 2.2, which will block the calling thread until all prior writes to shared or global memory are visible to all other threads. With this function called in the barrier function, correctness of data communication can be *guaranteed*.

3 The GTX 280 GPU and the CUDA Programming Model

The GTX 280 GPU (or *device*) consists of a set of 30 Single Instruction Multiple Data (SIMD) streaming multiprocessors (SMs), where each SM consists of eight scalar processor (SP) cores running at 1.2 GHz with 16-KB on-chip shared memory and a multi-threaded instruction unit.

The *device memory*, which can be accessed by all the SMs, consists of 1 GB of read-write global memory and 64 KB of read-only constant memory and read-only texture memory. However, all the device memory modules can be read or written to by the *host* processor. Each SM has on-chip memory, which can be accessed by all the SPs within the SM and will be one of the following four types: a set of 16K local 32-bit registers; 16 KB of parallel, shared, and software-managed data cache; a read-only constant cache that caches the data from the constant device memory; and a read-only texture cache that caches the data from the texture device memory. The global memory space is not cached by the device.

CUDA (Compute Unified Device Architecture) [10] is the parallel programming model and software environment provided by NVIDIA to run applications on their GPUs. It abstracts the architecture to parallel programmers via simple extensions to the C programming language.

CUDA follows a code off-loading model, i.e. data-parallel, compute-intensive portions of applications running on the host processor are typically off-loaded onto the device. The *kernel* is the portion of the program that is compiled to the instruction set of the device and then off-loaded to the device before execution.

In CUDA, threads can communicate and synchronize only *within* a thread block via the shared memory of the SM; there exists no mechanism for threads to communicate *across* thread blocks. If threads from two different blocks try to communicate via global memory, inter-block barrier synchronization is needed. Currently, the launch of a kernel from the host processor to the GPU serves as an implicit barrier to all threads that were launched by the previous kernel.

4 Smith-Waterman (SWat) Algorithm

The SWat algorithm [14] is an *optimal local sequence alignment* methodology that follows the dynamic programming paradigm, where intermediate alignment scores are stored in a matrix before the maximum alignment score is calculated. Next, the matrix entries are inspected, and the highest-scoring local alignment is generated. The SWat algorithm can thus be broadly classified into two phases: (1) matrix filling and (2) backtracing.

To fill out the dynamic-programming (DP) matrix, the SWat algorithm follows a scoring system that consists of a scoring matrix and a gap-penalty scheme. The scoring matrix, M is a two-dimensional matrix containing the scores for aligning individual amino acid or nucleotide residues. The gap-penalty scheme provides the option of gaps being introduced within the alignments, hoping that a better alignment score can be generated; but they incur some penalty or negative score. In our implementation, we consider an affine gap penalty scheme that consists of two types of penalties. The *gap-open* penalty, o is incurred for starting (or opening) a gap in the alignment, and the *gap-extension* penalty, e is imposed for extending a previously existing gap by one unit. The gap-extension penalty is usually smaller than the gap-open penalty.

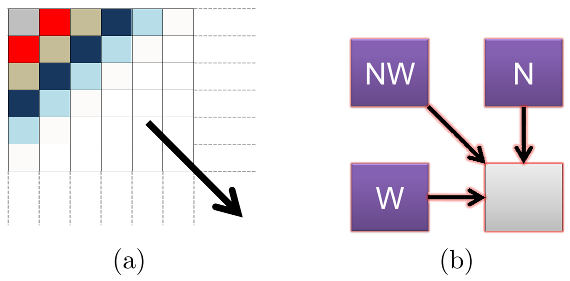


Figure 2. The SWat Wavefront Algorithm and its Dependencies.

Using this scoring scheme, the dynamic-programming matrix is populated via a *wavefront* pattern, i.e. beginning from the northwest corner element and going toward the southeast corner; the current anti-diagonal is filled after the previous anti-diagonals are computed, as shown in Figure 2(a). Moreover, each element in the matrix can be computed only after its north, west, and northwest neighbors are computed, as shown in Figure 2(b). Thus, elements within the same anti-diagonal are independent of each other and can therefore be computed in parallel. The backtracing phase of the algorithm is essentially a sequential operation that generates the highest scoring local alignment. In this paper, we mainly focus on accelerating the matrix filling because it consumes more than 99% of the execution time and it is the object to be parallelized.

5 Techniques for Accelerating SWat on the GPU

In this section, we describe a series of techniques to accelerate the access of global memory and decrease the synchronization time, and hence, communication across thread blocks.

5.1 Accelerating Global Memory Access

There are three techniques proposed for the effective global memory access, which are: 1) matrix re-alignment to optimize the sequence of memory accesses; 2) coalesced memory access to amortize the overhead of loading from and storing to memory; 3) tiling in order to sufficiently increase computational granularity so as to amortize the overhead of global memory access.

5.1.1 Matrix Re-Alignment

To leverage the SIMD processing of the GPU, we store the matrix in memory in the *diagonal-major* data format instead of the *row-major* data format, as shown in Figure 3. The matrix in Figure 3(a) is stored as the layout in Figure 3(b), where all the anti-diagonals are arranged in sequence. As a result, threads in a block can access memory in adjacent locations, and data can be transferred in blocks with larger size [10]. An implementation using this technique is referred to as a *simple implementation*.

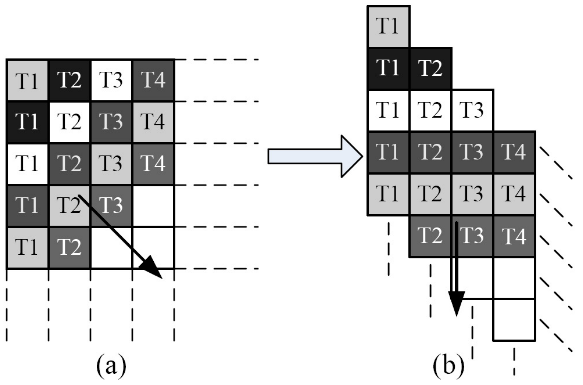


Figure 3. Matrix Representation in Memory.

With the diagonal-major data format, we offload the data-intensive, matrix-filling part onto the GPU device. If the CPU implicit synchronization is used, the dependence between consecutive anti-diagonals of the matrix forces a synchronization operation, and hence a new kernel invocation, after the computation of every anti-diagonal.

We distribute the computation of the elements in every anti-diagonal uniformly across all the threads in the kernel in order to completely utilize all the SMs on the chip and support the realistic problem size. To simplify our implementation, every kernel contains a one-dimensional grid of thread blocks, where each block further contains a single dimension of threads, as shown in Figure 4.

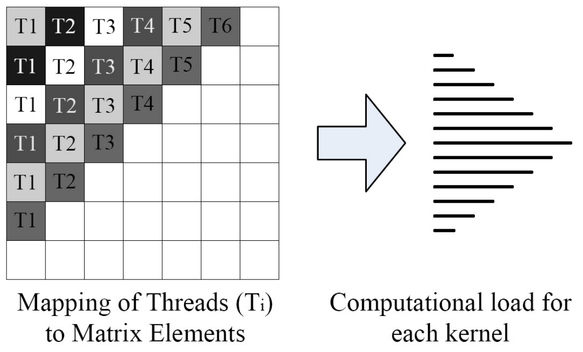

T1
T2
T3
T4
T5
T6
Mapping of Threads (Ti) to Matrix Elements
Computational load for each kernel

Figure 4. Mapping of Threads to Matrix Elements → Variation in the Computational Load Imposed on Successive Kernels.

5.1.2 Coalesced Memory Access

In addition to changing the matrix layout in Figure 3(b) to improve the effectiveness of global memory access, we transform non-coalesced data accesses into coalesced ones. Running the CUDA profiler [9] shows that 63% of memory accesses are non-coalesced if only the matrix re-alignment technique (i.e., simple implementation) is used. In this subsection, we propose a method to improve percentage of coalesced memory accesses and refer to this implementation as our *coalesced implementation.*

Since each anti-diagonal is computed by a new kernel and each thread accesses an integer (32-bit word), if the starting addresses of every anti-diagonal is aligned to 64-byte boundaries, then all the writes are coalesced, as shown in Figure 5. The skewed dependence between the elements of neighboring anti-diagonals restricts the degree of coalescing for the read operations from global memory. Also, we select the block and grid dimensions of the kernel, such that all the thread blocks have coalesced memory store.

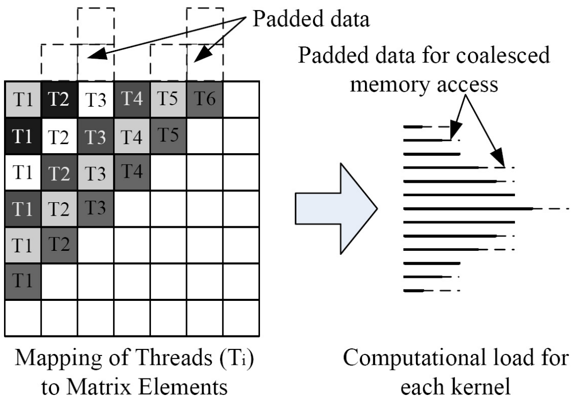

Padded data
Padded data for coalesced memory access
T1
T2
T3
T4
T5
T6
Mapping of Threads (Ti) to Matrix Elements
Computational load for each kernel

Figure 5. *Coalesced* **Data Representation of Successive Anti-Diagonals in Memory.**

5.1.3 Tiling

In the previous subsections, we proposed techniques for efficient global memory access. Here we apply the tiled-wavefront design pattern, which was used to efficiently map SWat to the IBM Cell/BE [1], to the GPU architecture. This approach amortizes the cost of kernel launches by grouping the matrix elements into computationally independent *tiles*. In addition, shared memory is used to reduce the amount of data needed to load from and store to the global memory.

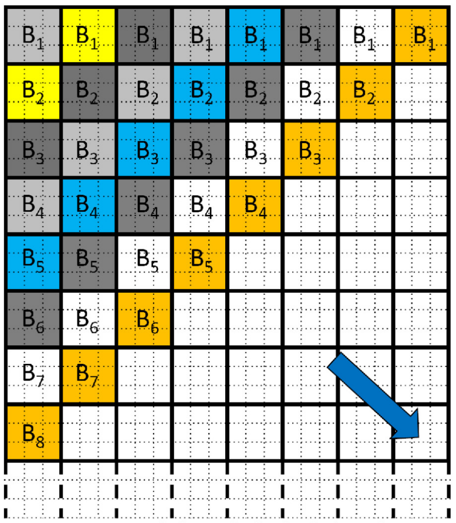
Our tile-scheduling scheme assigns a thread block to compute a tile, and a grid of blocks (kernel) is mapped to process a single *tile-diagonal*, thus decreasing the number of barriers needed to fill the alignment matrix. CPU implicit synchronization via new kernel launches or our proposed GPU synchronization (see Section 5.2) serve as barriers to threads from the previous tile-diagonal. Consecutive tile-diagonals are computed one after another from the northwest corner to the southeast corner of the matrix in the design pattern of a *tiled wavefront*, as shown in Figure 6.

The elements *within* a tile are computed by a thread block by following the simple wavefront pattern, starting from the northwest element of the tile. The threads within each block are synchronized after computing every anti-diagonal by explicitly calling CUDA's local synchronization function `__syncthreads()`.

Each thread block computes its allocated tiles within shared memory. The processed tile is then transferred back to the designated location in global memory. This memory transfer will be coalesced because we handcraft the allocation of each tile to follow the rules for coalesced memory accesses.

5.2 GPU Synchronization

Efficient global memory accesses can accelerate the computation, but it cannot accelerate the synchronization on the GPU. While the tiled-wavefront technique reduces the number of kernel launches, it explicitly and implicitly serializes the computation both within and across tiles, respectively. One solution to this problem is to introduce



B1
B2
B3
B4
B5
B6
B7
B8
Mapping of Thread Blocks (Bi) to the Tiles within the Matrix

Figure 6. Tiled Wavefront.

an efficient synchronization mechanism across thread blocks. In this way, we avoid both launching the kernel multiple times and serializing the tiled wavefront.

```
//the mutex variable
__device__ volatile int g_mutex;

//GPU synchronization function
__device__ void __GPU_sync(int goalVal)
{
    //thread ID in a block
    int tid = threadIdx.x * blockDim.y
                         + threadIdx.y;
    //memory flush to all threads
    __threadfence();
    // only thread 0 is used for
    //synchronization
    if (tid == 0)   {
        atomicAdd((int *)&g_mutex, 1);

        //only when all blocks add 1 to
        //g_mutex, will it be equal to
        //goalVal
        while(g_mutex != goalVal)   {
            //Do nothing
        }
    }
    __syncthreads();
}
```

Figure 7. Code Snapshot of GPU Synchronization.

However, NVIDIA discourages inter-block communication via global memory because its outcome can be undefined. Furthermore, a classic deadlock scenario can occur if multiple blocks are mapped to the same SM, and the active block waits on a message from the block that is yet to be scheduled by the CUDA thread scheduler [10]. CUDA threads do not yield the execution, i.e., they run to completion once spawned by the CUDA thread scheduler, and therefore, the deadlocks cannot be resolved in the same way as in traditional processor environments. This problem can be solved if and only if there exists a one-to-one mapping between the SM and the thread block. In other words, if a GPU has 'X' SMs, then at most 'X' thread blocks can be spawned in the kernel to avoid deadlock. In this paper, we use the GTX 280, which has 30 SMs, and spawn kernels with at most 30 thread blocks each. Note that this idea also works on other generations of the NVIDIA GPU with any number of SMs. To further ensure that no two blocks can be scheduled to be executed on the same SM, we can spawn maximum permissible threads per block or pre-allocate all of the available shared memory to each block.

We implement our GPU synchronization by first identifying a global memory variable (g_mutex) as a shared mutex, initialized to 0. At the synchronization step, each block chooses one representative thread to increment g_mutex by using the atomic function atomicAdd.[2] The synchronization point is considered to be reached when the value of g_mutex equals the goalVal, which is the number of blocks in the kernel. Figure 7 shows the pseudo-code for the GPU synchronization function __GPU_sync(), where correctness of inter-block communication is guaranteed by the function `__threadfence()`.

In the synchronization function __GPU_sync(), `goalVal` is set to the number of blocks N in the kernel when the barrier function is first called. The value of `goalVal` is then incremented by N each time when the barrier function is successively called. This design is more efficient than keeping `goalVal` constant and resetting `g_mutex` after each barrier because the former saves the number of instructions and avoids conditional branching.

6 Performance Analysis

This section presents the performance evaluation of various SWat implementations. From Section 5, by combining the three memory-access techniques — *matrix re-alignment*, *coalesced memory access*, and *tiled wavefront* — and the two synchronization methods — *CPU (implicit) synchronization* and *GPU synchronization*, we present six different implementations on the GPU. In addition, we compare our new GPU implementation (CUDA-SWat) to our previous Cell/BE implementation (Cell-SWat) [1].[3]

We use CUDA version 2.2 as our programming interface to the GTX 280 GPU card, which has 1-GB global memory and 30 SMs, each running at 1.2GHz. Our Cell/BE platform is a PlayStation 3 (PS3) game console, which is powered by the Cell/BE processor. The Linux kernel on the PS3 runs on top of a proprietary hypervisor that disallows the use of one of the SPE cores while another SPE core is hardware-disabled. Thus, we can effectively use only 6 SPE cores for computational purposes. The Cell processor on the PS3 executes at 3.2 GHz and has a total main memory of 256MB. For the results presented here, we choose sequence pairs of size 3K and report their alignment results. While the GPU can handle sequences of 8K in size, the Cell/BE cannot since anything larger would not fit into the 256-MB main memory of the PS3. So, although we can align sequences as long as 8461 characters in length on the NVIDIA GTX 280, we do *not* report these even better results in this paper in order to ensure a fair comparison between the Cell/BE and the GTX 280 graphics card.

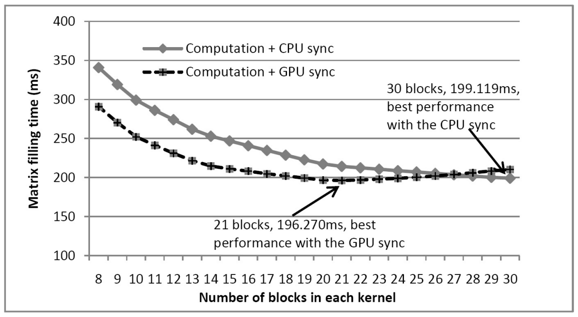
[2]Atomic operations are available on the NVIDIA graphics cards with compute capability 1.2 and up.

[3]Given the memory constraints of the GPU, and particularly the Cell, we set the tile size to 32×32, and the number of threads per block to 128.

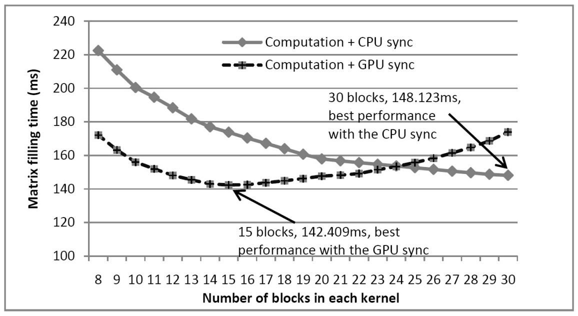
With the above SWat implementations and their associated computing platforms, we evaluate the performance of our six different implementations along three dimensions: (1) total execution (matrix filling) time, (2) profile of the time spent computing versus synchronizing in the matrix-filling phase on the GPU, and (3) comparison of the "compute + data communication" approach of the Cell/BE versus the "compute + synchronization" approach of the GPGPU, which is necessitated due to the lack of direct multiprocessor-to-multiprocessor communication.

A final note: As traditionally done, we focus on parallelizing the matrix-filling phase. All the results are the average of three runs.

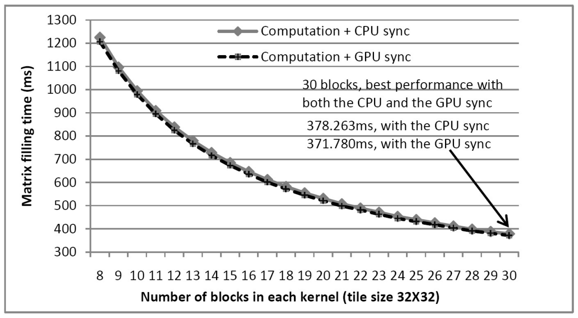
6.1 Execution (Matrix-Filling) Time



(a) Simple implementation



(b) Coalesced implementation



(c) Tiled wavefront

Figure 8. Total Execution (Matrix-Filling) Time.

Figure 8 presents the total matrix-filling time of the six GPU implementations. Figure 8(a) shows the matrix-filling time of the simple implementation with both the CPU and the GPU synchronization. We set the block number from 8 to 30 to show the matrix-filling time change against the block number. Figure 8(b) shows the results of the coalesced implementation, and Figure 8(c) is for the tiled wavefront.

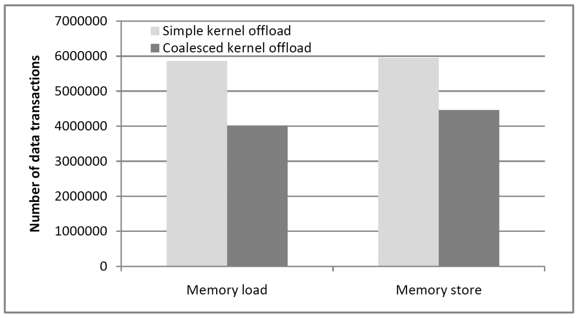


Figure 9. Number of Data Transactions to Global Memory.

Figure 8 illustrates that *for the same number of blocks, the coalesced implementation (with either CPU or GPU synchronization) is by far the fastest while the tiled wavefront is the slowest.* Why? In the coalesced implementation, fewer data transactions are needed to fill the same matrix, thus reducing the aggregate data-access time to memory and speeding up program execution. Using the CUDA profiler, we indirectly verify this by showing the difference in the number of data transactions between the simple and coalesced implementation, as shown in Figure 9.

Though the tiled wavefront reduces the synchronization time by reducing the number of kernel launches, each multiprocessor must now execute a larger computational granularity. More importantly, the occupancy of the multiprocessor in the tiled-wavefront implementation is only 0.125, much lower than that of the simple implementation at 1.000, as shown in Table 1. This means that 0.875 of the GPU "goes to waste" when using the tiled-wavefront approach. Thus, while the technique of tiling significantly improves the performance of SWat on the Cell/BE, it has the opposite effect on the GTX 280 GPU.

In addition, as mentioned in Section 5.2, the tiled wavefront serializes the matrix filling explicitly and implicitly.

When using CPU synchronization, the best SWat performance is achieved with 30 blocks per kernel launch. With GPU synchronization, however, only 21 blocks are needed in the simple implementation and 15 blocks in the coalesced implementation to achieve the best performance. The latter can be attributed to the tradeoff between the increase in synchronization time and the decrease in the computation time when more blocks are in the kernel. With more blocks, additional resources can be used for the computation, which can accelerate the computation; however, at the same time, more time is needed for GPU synchronization because of the "atomic add" in the synchronization function, which can only execute sequentially even in different blocks. With 30 blocks, in the simple and coalesced implementation, the time increase for GPU synchronization is more than the computation time decrease, when compared to that of 21 and 15 blocks in the kernel, respectively.

For tiled wavefront, the synchronization time is very small compared to the computation time. The decrease in computation time is much larger than the increase in GPU synchronization time when more blocks are used. As a result, the best performance for tiled wavefront is achieved with 30 blocks in the kernel for both the CPU and GPU synchronization.

Table 1. Simple Implementation versus Tiled Wavefront via the CUDA Profiler.

	#calls	occu-pancy	coalesced load	coalesced store	branch	divergent branch	instruc-tions	warp serialize
Simple imlementation	10359	1.0	5541640	5950074	737542	118791	4383409	640129
Tiled wavefront	335	0.125	5109771	7507480	3581961	346729	17535715	1529807

Finally, in comparing the performance of *only* the synchronization within SWat, GPU synchronization always outperforms CPU synchronization. With the CPU synchronization, the best synchronization times are 199.119 ms, 148.123 ms, and 378.263 ms for the simple implementation, coalesced implementation, and tiled wavefront, respectively, while the corresponding GPU synchronization times are 196.270 ms, 142.409 ms, and 371.780 ms, respectively. This indicates that performance can be improved with a better synchronization method on the GPU.

6.2 Computation vs. Synchronization on the GPU

To determine the time spent in computation versus synchronization on the GPU, we first calculate the total execution times for all our CUDA-SWat implementations for both the CPU and GPU synchronization methods. The total execution time can then be partitioned into the computation time and the non-computation (synchronization) time. We calculate the computation time by running the SWat implementations with GPU synchronization, but by commenting out the __GPU_sync() synchronization function. Here we assume that the computation time of all our SWat implementations are the same, irrespective of the synchronization method used. The non-computation (synchronization) time for each of the implementations will then be the difference between the corresponding total execution time (with either the CPU or the GPU synchronization) and the computation time.

Figure 10 shows the percentage of the computation time and the synchronization time corresponding to the best execution configurations of the six GPU implementations. Firstly, in Figure 10, we observe that the percentage of time to synchronize in the simple implementation is smaller than that in the coalesced implementation for both CPU and GPU synchronization, respectively. The reason is the computation time of the coalesced implementation is smaller, which makes the synchronization time occupy a larger percentage of the total matrix filling time. Secondly, percentage of the synchronization time for the GPU synchronization is always less than that of the CPU synchronization if the memory access technique is the same, which indicates that the GPU synchronization needs less time than the CPU synchronization. Thirdly, although the GPU synchronization can reduce the synchronization time, the percentage of time spent synchronizing is still 32.63% and 36.17% for the simple and coalesced implementations, respectively. This means that the synchronization consumes a large part of the total kernel execution time.

6.3 GPU vs. Cell/BE

In this section, we analyze both the computation and the non-computation time of SWat on the GPU and the Cell/BE. Figure 11 shows the composition of the matrix-filling time of all the implementations on the GPU and the Cell/BE for their best execution configurations.

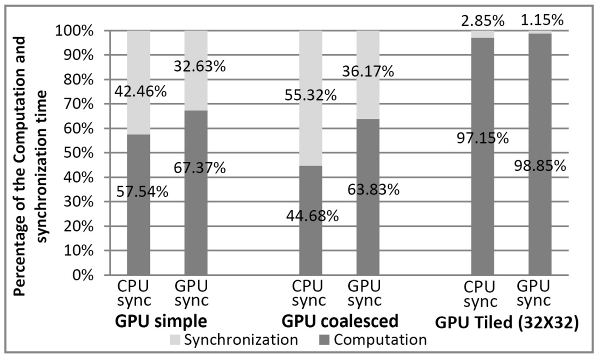


Figure 10. Computation vs. Synchronization Time on the GPU.

From Figure 11, although the tiled wavefront of Cell-SWat can achieve a good performance [1], the matrix-filling time (239.13 ms) is larger than that of CUDA-SWat's simple implementation (199.12 ms and 196.27 ms for CPU and GPU synchronization, respectively) and the coalesced implementation (148.12 ms and 142.41 ms for CPU and GPU synchronization, respectively), but it is less than that of the tiled wavefront on the GPU (378.26 ms and 371.78 ms). This indicates that the computational horsepower on the Cell/BE (PS3) is not as fast as that on the GPU.

However, the data communication time on the Cell/BE is much less than all the GPU implementations. Although the GPU tiled wavefront can reduce the synchronization time to 10.78 ms and 4.29 ms for the CPU and GPU synchronization, respectively, it is still much larger than that on the Cell/BE at 0.69 ms. Moreover, the synchronization benefits that are gained from the GPU tiled wavefront pay the cost of a significant increase in the computation time on the GPU (370 ms), which is larger than that on the Cell/BE (238.44 ms). In other words, data communication on the Cell/BE is much more efficient than that on the GPU. This is because, data communication on the Cell/BE between the Synergistic Processing Elements (SPEs) is supported via Direct Memory Access (DMA) primitives over the Element Interconnect Bus. In contrast, the GPU has no such direct communication between its "SPEs", known as multiprocessors. As a consequence, we used the high-overhead CPU (implicit) barrier synchronization as well as constructed our own software-based GPU barrier synchronization to facilitate data communication on the GPU.

7 Conclusions

In this paper, we choose one of the dynamic programming applications — Smith-Waterman — as an example to accelerate it on the GPU. In contrast to the previous coarse-grained parallelization, we implement the fine-grained parallelization of the SWat algorithm. To improve its performance, techniques to decrease both the computa-

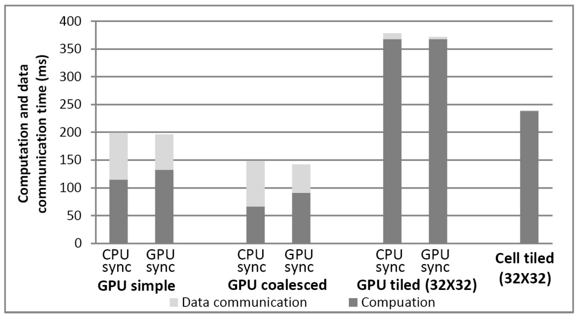


Figure 11. Performance Profile of CUDA-SWat vs. Cell-SWat.

tion time and the data communication time are proposed. For the former, it mainly focuses on the efficient global memory access; and for the latter, a new GPU synchronization method is proposed, which can synchronize the execution across different blocks without the host involved. These techniques are proposed for SWat, but they can be applied to any dynamic programming-based algorithms.

To evaluate the performance, we compare the matrix filling time of each GPU implementation. From the experiment results, compared to the simple implementation, the coalesced implementation can speedup the execution; but the tiled wavefront cannot. Also, the proposed GPU synchronization can achieve a better performance than the CPU synchronization. From the results, synchronization time percentage can decrease from 42.46% to 32.63% for the simple implementation and from 55.32% to 36.17% for the coalesced implementation with the GPU synchronization. In addition, we compare the computation time and the data communication time between the Cell/BE and the GPU. From our results, matrix filling time on Cell/BE is larger than those of the simple and coalesced implementations on the GPU, but its data communication is much less. Though we have a tiled wavefront on the GPU that can decrease the synchronization time, it is at the cost of increasing the computation time to much more than that on the Cell/BE. As a result, for algorithms such as Dynamic Programming, efficient data communication mechanism is important for them to be accelerated on multi-core and many-core architectures.

As future work, we would like to use some cached memory, e.g., texture memory and constant memory, to decrease the computation time even more for SWat. Also, from our experiment results, one problem in the GPU synchronization is the atomic operation — if more blocks are in the kernel, more time is needed for the synchronization. In future, we will try to decrease the overhead caused by the atomic operations in some degree or even totally removing them in the GPU synchronization function. Finally, we would like to extend the proposed GPU synchronization method to other algorithms.

Acknowledgments

We would like to thank Heshan Lin, Jeremy Archuleta, Tom Scogland, and Song Huang for their technical support and feedback on the manuscript.

This work was supported in part by a NVIDIA Professor Partnership Award and an IBM Faculty Award.

References

[1] A. M. Aji, W. Feng, F. Blagojevic, and D. S. Nikolopoulos. Cell-SWat: Modeling and Scheduling Wavefront Computations on the Cell Broadband Engine. In *Proc. of the ACM International Conference on Computing Frontiers*, May 2008.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[3] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and its First Implementation. *IBM developerWorks*, November 2005.

[4] A. Grama, A. Gupta, G. Karypis, and V. Kumar. Dynamic Programming. In *Introduction to Parallel Computing, Second Edition*, pages 515–536, January 2003.

[5] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-Sequence Database Scanning on a GPU. *IPDPS*, April 2006.

[6] Y. Liu, W. Huang, J. Johnson, and S. Vaidya. GPU Accelerated Smith-Waterman. In *Proc. of the 2006 International Conference on Computational Science, Lectures Notes in Computer Science Vol. 3994*, pages 188–195, June 2006.

[7] S. A. Manavski and G. Valle. CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment. *BMC Bioinformatics*, 2008.

[8] Y. Munekawa, F. Ino, and K. Hagihara. Design and Implementation of the Smith-Waterman Algorithm on the CUDA-Compatible GPU. In *Proc. of the 8th IEEE International Conference on BioInformatics and BioEngineering*, pages 1–6, October 2008.

[9] NVIDIA. CUDA Profiler, 2008. `http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/CudaVisualProfiler_linux_release_notes_1.0_13June08.txt`.

[10] NVIDIA. NVIDIA CUDA Programming Guide-2.0, 2008. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`.

[11] L. Nyland, M. Harris, and J. Prins. Fast N-Body Simulation with CUDA. *GPU Gems*, 3:677–695, 2007.

[12] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, and W. W. Hwu. GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications. In *Proc. of the Conference on Computing Frontiers*, pages 273–282, May 2008.

[13] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. High-Throughput Sequence Alignment Using Graphics Processing Units. *BMC Bioinformatics*, 8(1):474, 2007.

[14] T. Smith and M. Waterman. Identification of Common Molecular Subsequences. In *Journal of Molecular Biology*, April 1981.

[15] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating Molecular Modeling Applications with Graphics Processors. *Journal of Computational Chemistry*, 28:2618–2640, 2007.

[16] G. M. Striemer and A. Akoglu. Sequence Alignment with GPU: Performance and Design Challenges. In *IPDPS*, May 2009.

[17] G. Tan, N. Sun, and G. R. Gao. A Parallel Dynamic Programming Algorithm on a Multi-core Architecture. In *Proc. of the ACM Symp. on Parallelism in Algorithms and Architectures*, June 2007.

[18] V. Volkov and J. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, November 2008.