Accelerating Data-Serial Applications on Data-Parallel GPGPUs: A Systems Approach

Ashwin M. Aji and Wu-chun Feng
Department of Computer Science
Virginia Tech
{aaji,feng}@cs.vt.edu

Abstract

The general-purpose graphics processing unit (GPGPU) continues to make significant strides in high-end computing by delivering unprecedented performance at a commodity price. However, the many-core architecture of the GPGPU currently allows only data-parallel applications to extract the full potential out of the hardware. Applications that require frequent synchronization during their execution do *not* experience much performance gain out of the GPGPU. This is mainly due to the lack of explicit hardware or software support for inter-thread communication across the entire GPGPU chip.

In this paper, we design, implement, and evaluate a highly-efficient software barrier that synchronizes all the thread blocks running on an offloaded kernel on the GPGPU *without* having to transfer execution control back to the host processor. We show that our custom software barrier achieves a *three-fold* performance improvement over the existing approach, i.e., synchronization via the host processor.

To illustrate the aforementioned performance benefit, we parallelize a data-serial application, specifically an optimal sequence-search algorithm called Smith-Waterman (SWat), that requires frequent barrier synchronization across the many cores of the nVIDIA GeForce GTX 280 GPGPU. Our parallelization consists of a suite of optimization techniques — optimal data layout, coalesced memory accesses, and blocked data decomposition. Then, when coupled with our custom software-barrier implementation, we achieve nearly a *nine-fold* speed-up over the serial implementation of SWat. We also show that our solution delivers $25\times$ faster on-chip execution than the naïve implementation.

1 Introduction

While computational horsepower continues to double, it does so via doubling the number of cores in both multi-core and many-core architectures. Among the increasingly commodity many-core architectures are nVIDIA's and AMD/ATI's general-purpose graphics processing units (GPGPUs), which until recently, were used mainly for accelerating only graphics-based applications. With the elimination of key architectural limitations, the GPUs have evolved from the traditional graphics pipeline model into programmable devices that are suited for accelerating scientific applications, e.g., [9].

However, the many-core architecture of the GPGPU typically maps well only to data-parallel applications that have minimal to no inter-thread communication. Applications that require frequent synchronization during their execution do *not* experience much performance gain out of the GPGPU. This is mainly due to the lack of explicit hardware or software support for inter-thread communication across the entire GPGPU chip.

The current (implicit) barrier strategy for the CUDA platform is to simply re-launch a new kernel, which is an expensive operation. In this paper, we design, implement, and evaluate a custom software-barrier implementation for the CUDA platform that explicitly synchronizes all the threads running in the kernel without having to transfer control to the host processor. We show that our software-barrier implementation achieves a *three-fold* performance improvement over the existing approach, i.e., synchronization via the host processor.

To illustrate the effectiveness of our software barrier, we parallelize a data-serial application, specifically an optimal sequence-search algorithm called Smith-Waterman (SWat) [8], which requires frequent barrier synchronization across the many cores of the nVIDIA GeForce GTX 280 GPGPU. The SWat algorithm is used extensively in a wide range of areas from estimating evolutionary histories to identifying possible drugs to assisting in the cure of prevalent diseases. However, the exponential growth in the size of nucleotide and protein databases has made the optimal sequence-search algorithms impractical to search on these databases because of their quadratic time and space complexity. Thus, there exists a need to parallelize the Smith-Waterman algorithm. In addition, parallelizing the SWat algorithm means parallelizing the dynamic programming paradigm, which is one of the 13 dwarfs[1] [2] of parallel programming.

In this paper, we present CUDA-SWat – a highly efficient parallelization of the Smith-Waterman code on the CUDA programming platform [7] of the nVIDIA GeForce GTX 280 GPGPU. The accelerated code makes use of a series of optimizations, including optimal data-layout strategies, coalesced memory accesses, and blocked data decomposition techniques. First, we optimize the data-layout strategy so that the data is compatible with the SIMD-processing style of the CUDA platform. Next, we coalesce (align) the data accesses, according to the CUDA optimization guidelines. We then group data elements into *tiles* to improve the locality of computation. We show that these optimizations, in tandem with our custom software-barrier implementation, significantly enhances the parallel performance of the algorithm.

Specifically, we show that the cumulative effect of our optimizations results in nearly a *nine-fold* speed-up over the serial implementation of SWat. We also show that our solution delivers $25\times$ faster on-chip execution than the naïve implementation.

In summary, this paper will detail the strategies involved in efficiently mapping the (essentially) data-serial Smith-Waterman algorithm onto the data-parallel GPGPU architecture of the nVIDIA GeForce GTX 280 in order to extract the maximum performance out of the hardware.

1.1 Related Work

Smith-Waterman has previously been implemented on the GPGPU by using graphics primitives [5, 4], and more recently, using CUDA [6]. While the older implementations that use graphics primitives report decent speedups over the serial implementation, they are now obsolete, and we will not learn much from them in terms of developing general programming models on the current

[1] A dwarf is an algorithmic method that captures a pattern of computation and communication.

generation of GPGPUs, which mostly use regular C-style libraries.

The CUDA implementation of Smith-Waterman reports speed-ups of up to 30-fold, but it suffers from the following limitations. First, its approach only follows a coarse-grained parallelization by assigning a single problem instance to each thread on the device, thereby sharing the available GPGPU resources among multiple concurrent problem instances. This approach *severely* restricts the maximum problem size that can be solved by the GPGPU. Our implementation follows fine-grained parallelization by distributing the task of processing a single problem instance across all the threads on the GPGPU, thereby supporting realistic problem sizes. Lastly, to the best of our knowledge, no other work has attempted to implement a barrier synchronization mechanism across all the thread blocks in the CUDA programming environment.

The rest of this paper is organized as follows: Section 2 describes the nVIDIA GeForce GTX 280 architecture and the CUDA programming model. Section 3 presents the sequential Smith-Waterman algorithm. Section 4 describes the various optimization strategies to parallelize Smith-Waterman on the CUDA platform. Section 5 presents and discusses the results. Section 6 concludes the paper.

2 The nVIDIA GeForce GTX 280 GPGPU

This section summarizes the many-core, SIMT (Single-Instruction, Multiple-Thread) architecture of the nVIDIA Geforce GTX 280 GPGPU and the CUDA (Compute Unified Device Architecture) programming model [7].

2.1 GTX 280 Architecture

The GTX 280 GPGPU (or *device*) is implemented as a set of 30 SIMT streaming multiprocessors (SMs), where each multiprocessor consists of eight scalar processor (SP) cores running at 1.2 GHz, 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, 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 8192 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.

2.2 CUDA Programming Model

CUDA is the parallel programming model and software environment provided by nVIDIA to run applications on their graphics cards. CUDA abstracts the architecture to parallel programmers via simple extensions to C. For this paper, we have used CUDA version 2.0 as our programming interface.

CUDA follows the code off-load 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 downloaded to the device before execution.

Each kernel executes as a bunch of threads, which are logically organized in the form a grid of thread blocks. The dimensions of the blocks and the grid of thread blocks are specified as parameters before each kernel launch. Threads can be identified in code by the built-in thread and block identifiers that are generated by the CUDA run-time system. The kernel executes on the device such that each SM processes batches of blocks, one batch after the other. A thread block is mapped to execute on only one SM, but a single SM can execute multiple thread blocks simultaneously, depending on the register and memory requirements of each block. The on-chip shared memory of a SM can be accessed only by the thread block that is running on the SM, while the global, constant and texture memory modules are shared across all the the thread blocks in the kernel. The device memory can be read from or written to by the host via optimized direct memory access (DMA) calls and are persistent across kernel launches by the same application.

Threads can communicate and can be synchronized only within a thread block via the shared memory of the SM, but there exists no mechanism for threads to communicate *across* thread blocks. If threads from two different blocks communicate via global memory, the outcome of the memory transaction is undefined, and nVIDIA does not recommend inter-block communication. Thus, the launch of a kernel from the host processor to the GPGPU currently serves as an implicit barrier to all threads that were launched by the previous kernel.

The host machine, which housed the GPGPU, contains 4-GB RAM and an Intel Core 2 Duo E4500 CPU, with each core running at 2.2GHz. The installed operating system was Ubuntu 4.1.2 with the Linux kernel version 2.6.20-16.

3 The Smith-Waterman (SWat) Algorithm

Biological sequence alignment is a method of determining similar regions between two nucleotide or protein sequences. Local sequence alignment is a technique that compares sequence segments of all possible lengths and optimizes the similarity measure, which is termed as the *alignment score* of the sequences. The final step of the algorithm is to output the highest scoring local alignment. The Smith-Waterman algorithm [8] is an *optimal local sequence alignment* methodology that follows the dynamic programming paradigm, where the 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 Smith-Waterman algorithm can thus be broadly classified into two phases: (1) matrix filling and (2) backtracing.

To fill out the dynamic-programming (DP) matrix, the Smith-Waterman 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.

Using this scoring scheme, the dynamic-programming matrix is filled out following a *wave-*

front 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 1(a). Moreover, each element in the matrix can be computed only after the calculation of its north, west and northwest neighbors are computed, as shown in Figure 1(b). Thus, elements within the same anti-diagonal are independent of each other and can therefore be computed in parallel.

The recursive data dependence of the elements in the dynamic-programming matrix, combined with the scoring system can be explained by the Equations 1, 2 and 3.

$$DP_N[i,j] = e + max \begin{cases} DP_N[i-1,j] \\ DP_W[i-1,j] + o \\ DP_{NW}[i-1,j] + o \end{cases} \quad (1)$$

$$DP_W[i,j] = e + max \begin{cases} DP_N[i,j-1] + o \\ DP_W[i,j-1] \\ DP_{NW}[i,j-1] + o \end{cases} \quad (2)$$

$$DP_{NW}[i,j] = M(X_i, Y_j) + max \begin{cases} DP_N[i-1,j-1] \\ DP_W[i-1,j-1] \\ DP_{NW}[i-1,j-1] \end{cases} \quad (3)$$

The backtracing phase of the algorithm generates the highest scoring local alignment. The backtrace begins at the matrix cell that holds the optimal alignment score and proceeds in a direction opposite to that of the matrix filling, until a cell with score zero is encountered. The path thus traced yields the optimal local alignment.

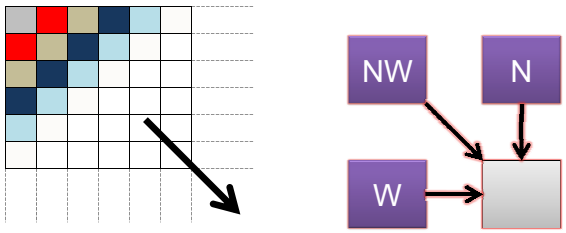

FIGURE 1: The Smith-Waterman wavefront algorithm and its dependencies.

The GTX 280 GPGPU contains 1 GB of global memory for our computational purposes. For all the tests, we chose eight randomly generated sequence pairs of sizes varying from 1 KB to 8 KB, thus covering most of the realistic sequence sizes [3]. The matrix-filling part took 99.9% of the overall execution time of the algorithm, and therefore, it was the only stage of the algorithm that had to be parallelized. We discuss the various parallelization strategies for the GPGPU in the next section.

4 CUDA-SWat: Optimizing Smith-Waterman for the GPGPU

In this section, we present a series of four optimization techniques and compare the performance of each method against each other and against the baseline naïve implementation.

4.1 Simple Kernel Offload

The first step of this optimization technique is to arrange the data in a way that is suitable for the SIMD-style processing of the streaming multiprocessors (SMs) of the GPGPU. The entire matrix is therefore physically stored in memory as a flat one-dimensional array, by storing adjacent anti-diagonals next to each other, as shown in Figure 2. We call this layout the *diagonal-major* data format. Data-parallel computation is possible only within each anti-diagonal of the matrix, and this data layout scheme significantly helps in achieving the end result.

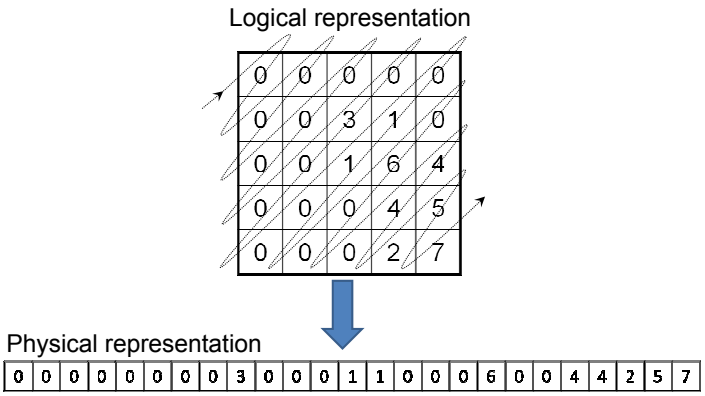

FIGURE 2: Matrix representation in memory.

The next step in the optimization process is to offload the data-intensive, matrix-filling part onto the GPGPU device. The dependence between consecutive anti-diagonals of the matrix force a synchronization operation after each anti-diagonal is computed. However, in the CUDA programming platform, only the threads within a thread block can be synchronized. Mapping the entire matrix-filling module onto a single thread block results in only a single SM of the GPGPU being utilized for computation. This causes the other 29 SMs to be idle and results in a massive waste of GPGPU chip resources.

However, the launch of a kernel serves as an implicit barrier to all the threads that were launched by the previous kernel. This forces us to re-map the problem, such that one kernel is launched to compute each anti-diagonal of the matrix. Every kernel will be made up of a one-dimensional grid of thread blocks, where each block further contains a single dimension of threads, as shown in Figure 3. We distribute the computation of the elements in every anti-diagonal uniformly among all the threads in the kernel. This method ensures the complete utilization of all the SMs available on the chip.

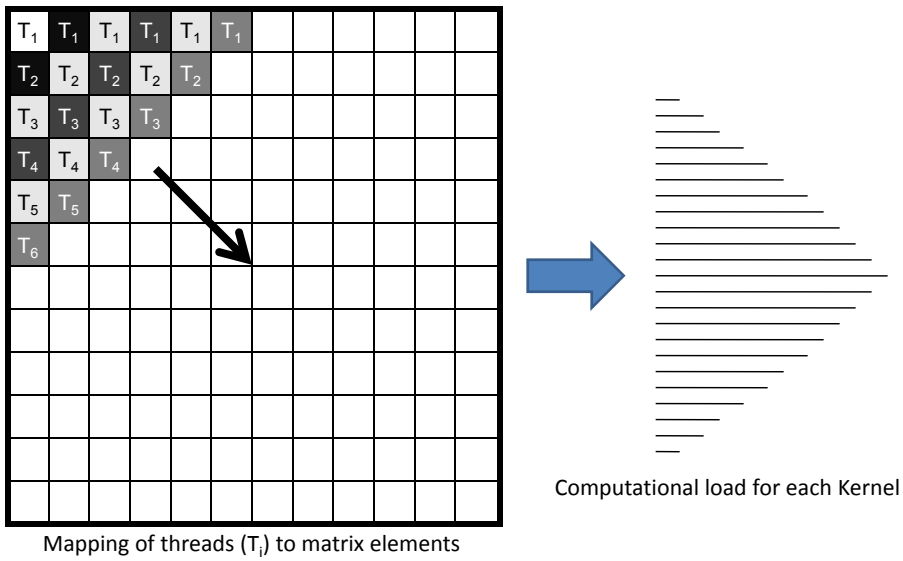

FIGURE 3: (Left) Mapping of threads to matrix elements and the (Right) variation of the computational load that is imposed on successive kernels.

The find_max optimization The backtrace operation starts at the largest element in the matrix, and it requires a find_max operation to be performed. Initially, our implementation executed this function on the host *after* the entire matrix was filled up and transferred to the host memory. We optimized this further by parallelizing the find_max function across all the threads of the kernel. The best scores that are calculated locally by each thread are passed to the host at the end of the matrix-filling phase. From this array of best scores, the host calculates the overall maximum score by performing at most T if checks, where T is the number of threads in the kernel. We achieve a *five-fold speed-up* over the naïve implementation by offloading the kernel to the device in conjunction with the diagonal-major data representation.

Once the entire matrix is filled in the device memory, it is completely transferred to the host memory before the backtrace operation is performed by the host.

4.2 Coalesced Kernel Offload

Although the previous optimization scheme shows some improvement over the host-run code, the approach still suffers from two major problems:

- Multiple kernel launches – A kernel launch per anti-diagonal means that a typical application will have thousands of kernel invocations. We developed a microbenchmark that launches empty kernels multiple times onto the device to characterize the effect of kernel invocations in an application. The results show that, for the problem sizes chosen for CUDA-SWat – i.e.

for approximately 16,000 kernel launches, about 41% of the total execution time is being spent in kernel launch alone.

- Non-coalesced global memory access – The effective global memory bandwidth significantly depends on the memory access patterns because global memory is not cached unlike the other memory modules on the device. Coalesced global memory accesses can sometimes have a significant improvement in performance over non-coalesced global memory accesses [7]. About 99% of the memory accesses in our previous optimization scheme were non-coalesced.

We deal with the non-coalesced global memory access in this section and discuss solutions for the multiple kernel launch problem in the next.

Coalesced memory accesses require that (1) only 32-bit, 64-bit, or 128-bit words should be accessed from global memory by each thread, (2) the global memory addresses that are simultaneously accessed by consecutively numbered threads during the execution of a single read or write instruction should be arranged such that, all the memory accesses can be coalesced into a single contiguous, aligned memory access and (3) when accessing x-byte words from global memory, the address location that is accessed by the thread with ID = 0 should be a multiple of $16 \times x$.

Since we currently launch separate kernels to compute every anti-diagonal and we are accessing integers (4-byte words), we make sure that the start address of every anti-diagonal is aligned to 64-byte boundaries to ensure that all the writes are coalesced as shown in Figure 4. The skewed dependence between the elements of neighboring anti-diagonals restrict the degree of coalescing among the reads from global memory. Moreover, we make sure that the dimension of the blocks and grid of blocks are all powers of 2, to enable all the thread blocks to enjoy coalesced memory accesses.

We achieve a maximum of $5.2\times$ speedup over the serial implementation by coalescing the global memory accesses in conjunction with the optimizations discussed in the previous sections.

4.3 Tiled Wavefront

This section discusses the problem of multiple kernel launches and our solution to the issue. Microbenchmark results reveal that 41% of the total execution time is spent in kernel invocations. To solve this problem, we revisit our previous work of the tiled-wavefront design that efficiently mapped SWat to the IBM Cell Broadband Engine [1], and now apply the idea to the GPGPU architecture. The tiled wavefront approach amortizes the cost of kernel launches by grouping the matrix elements into computationally independent *tiles*.

Tile Scheduling and Processing Our scheduling scheme assigns a thread block to compute a tile, and a grid of blocks (kernel) is mapped to process an entire *tile-diagonal*, thus decreasing the number of kernel launches. New kernel launches serve as implicit barriers to the threads from the previously executed kernel. Consecutive tile-diagonals are computed by different kernels one after another from the northwest corner to the southeast corner of the matrix, in the form of a *tiled wavefront*, as shown in Figure 5.

Also, the elements within a tile are computed in parallel by a thread block by following the simple wavefront pattern, starting from the northwest element of the tile. The threads within

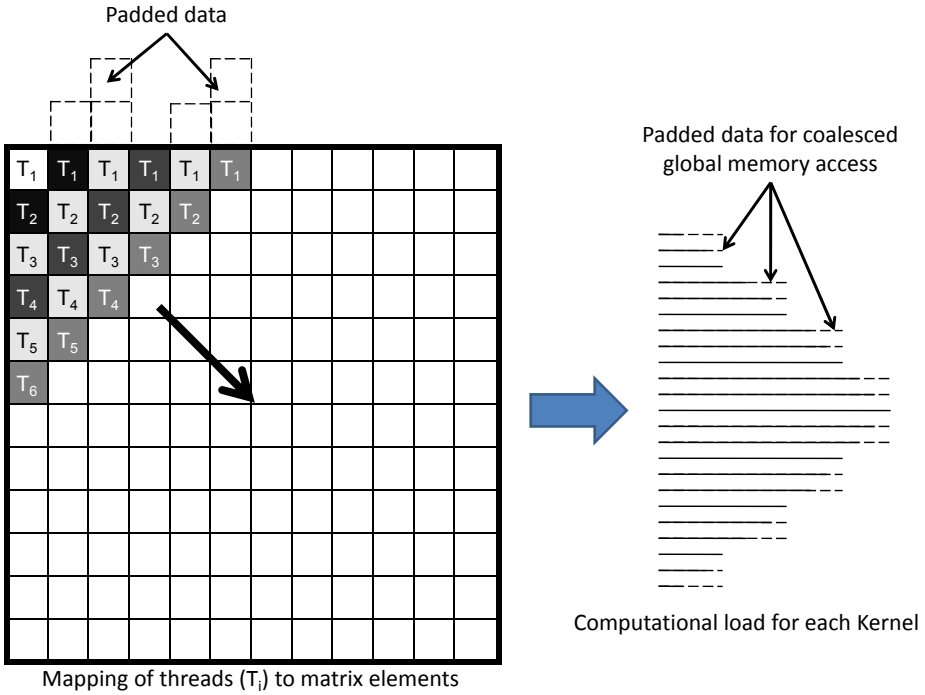

FIGURE 4: (Left) Mapping of threads to matrix elements and the (Right) variation of the computational load that is imposed on successive kernels. It also denotes the *coalesced* data representation of successive anti-diagonals in memory.

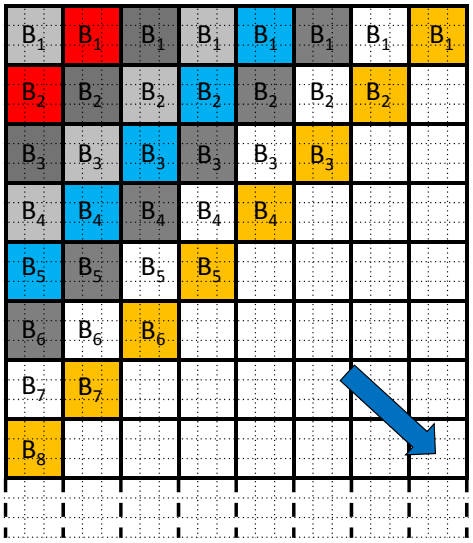

FIGURE 5: Tiled Wavefront.

each block are synchronized after computing every anti-diagonal, by explicitly calling the CUDA barrier function __syncthreads().

Computation-Communication Pattern The step-by-step communication-computation pattern performed by each thread block while processing a tile is shown in Figure 6 and explained as follows:

1. To start computing a tile, a thread block transfers the corresponding tile elements from global memory to the on-chip shared memory. This memory transfer will be coalesced because we handcraft the allocation of each tile to follow the rules for coalesced memory accesses. The boundary elements that are required for the current tile computation are assumed to be already present within the tile at this stage. Later steps will explain how to populate the tile boundary with the correct elements.

2. Next, the thread block proceeds with the tile computation within shared memory. The threads use block synchronization primitives after computing every anti-diagonal within the tile.

3. The processed tile is transferred back to its location in global memory.

4. Finally, boundary elements from the current tile are transferred to the boundaries of the neighboring tiles on the west and south. This operation is performed in parallel by using as many threads as the number of boundary elements per tile.

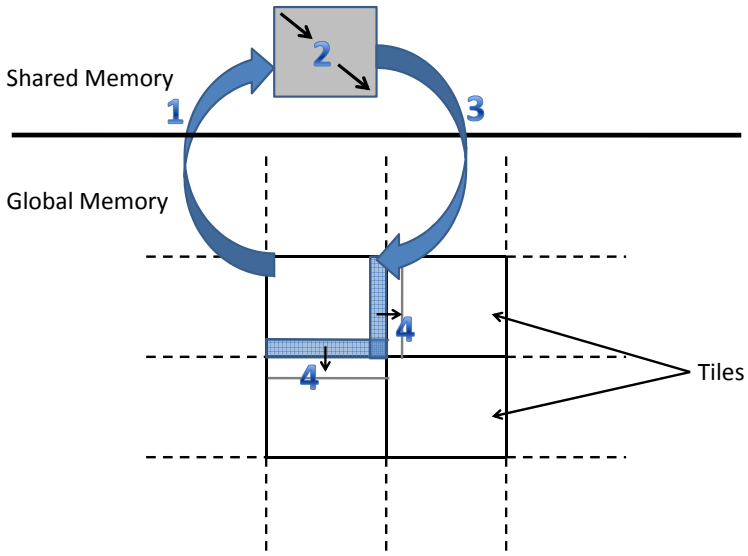

FIGURE 6: Computation-Communication pattern between tiles in global memory.

The above steps describe the method of processing non-boundary tiles. Boundary conditions can be easily checked and the redundant steps can be avoided while processing the boundary-tiles.

We achieve a maximum of $6\times$ speedup over the naïve implementation by applying the tiled wavefront design. The tiled wavefront approach achieves a 16.3% improvement in performance over the coalesced kernel offload scheme.

4.4 Coalesced + Custom Barrier

While the tiled wavefront approach reduces the impact of multiple kernel launches, it explicitly and implicitly serializes the computation both within and across tiles respectively. The ideal solution to this problem would be to revert to the coalesced kernel offload method, but introduce some sort of a synchronization mechanism across thread blocks. In this way, we avoid launching the kernel multiple times, and yet maintain coalesced memory accesses.

However, nVIDIA discourages inter-block communication via global memory and its outcome is undefined. 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 [7]. 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 a way that happens in the traditional processor environments, where one can yield the waiting process to execute other processes.

How can we derive insights into the design of the CUDA thread scheduler to be able to avoid the deadlock, and yet implement a custom barrier between the thread blocks? *We can ensure that all threads run to completion without deadlocks if and only if there exists a one-one mapping between the SM and the corresponding thread block.* For example, a kernel that is composed of not more than 30 blocks can be launched on the nVIDIA GTX 280, which has 30 streaming multiprocessors. We can also take a step further by spawning maximum permissible threads per block or allocating all of the available shared memory to each block to make sure that no two blocks can be scheduled to be executed on the same SM.

4.4.1 Custom Barrier Implementation

We implement the custom barrier by first identifying a global memory variable (g_mutex) as a shared mutex, initialized to 0. At the barrier synchronization step, each block chooses one representative thread to increment g_mutex by using the atomic function atomic_add[2]. The barrier is considered to be reached when the value of g_mutex equals the number of blocks in the kernel. The barrier can obviously be implemented as a decrementer as well. Code Snippet 1 shows the pseudo-code for the incrementer barrier.

We ran microbenchmark tests that launched empty kernels multiple times and compared its performance to a kernel that simply called the custom barrier function as many times. We found that the custom barrier method of inter-block synchronization achieved $3\times$ performance improvement over the multiple kernel launch method.

We achieve a maximum of $8.6\times$ speedup over the serial implementation by coalescing the global memory accesses in conjunction with the custom barrier implementation. The coalesced +

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

Code Snippet 1 Pseudo-code for the Custom Barrier (incrementer).

```
/* Mutex Declaration in Global Memory */
__device__ unsigned int g_mutex;

/* assume that g_mutex is already initialized to 0 */
__device__ void __barrier_incr()
{
  int thread_id = threadIdx.x;
  int thread_count = (gridDim.x * gridDim.y);
  if(thread_id == 0)
  {
    atomicAdd(&g_mutex, 1);
    while(g_mutex != thread_count)
    {
        /* do nothing */
    }
  }
  __syncthreads();
}
```

custom barrier approach achieves a 42.5% improvement over the tiled wavefront implementation.

The tiled wavefront optimization has negligible number of kernel launches and therefore, the custom barrier optimization on top of that is not expected to improve the performance of SWat any further.

5 Results

Figure 7 presents the results from our optimization methods showing the speedup values for all the possible execution configurations of the application kernel. The speedup is constant beyond 30 thread blocks for the kernel offload and the tiled wavefront optimization techniques, because the high utilization of shared resources per block forces the CUDA thread scheduler to execute only one active block per SM. This means that the performance of a kernel that has more thread blocks than the number of available SM's will not be better than the kernel with exactly 30 thread blocks (where the chip has 30 SM's). The speedup for the coalesced + custom barrier drops beyond 8 blocks because of increasing contention to the shared mutex resource. We also see that the performance is always better when there are more threads per block.

Figure 8 shows the best running times, across all kernel execution configurations, for all the presented optimizations. The coalesced + custom barrier optimization provides the best speedup ($8.6\times$ quicker overall and $25.5\times$ faster chip execution time) over the naïve implementation. Also, we can see that the kernel execution time for the custom barrier approach is roughly $3\times$ quicker than the (non-custom barrier) coalesced kernel offload method. We obtained all the above exper-

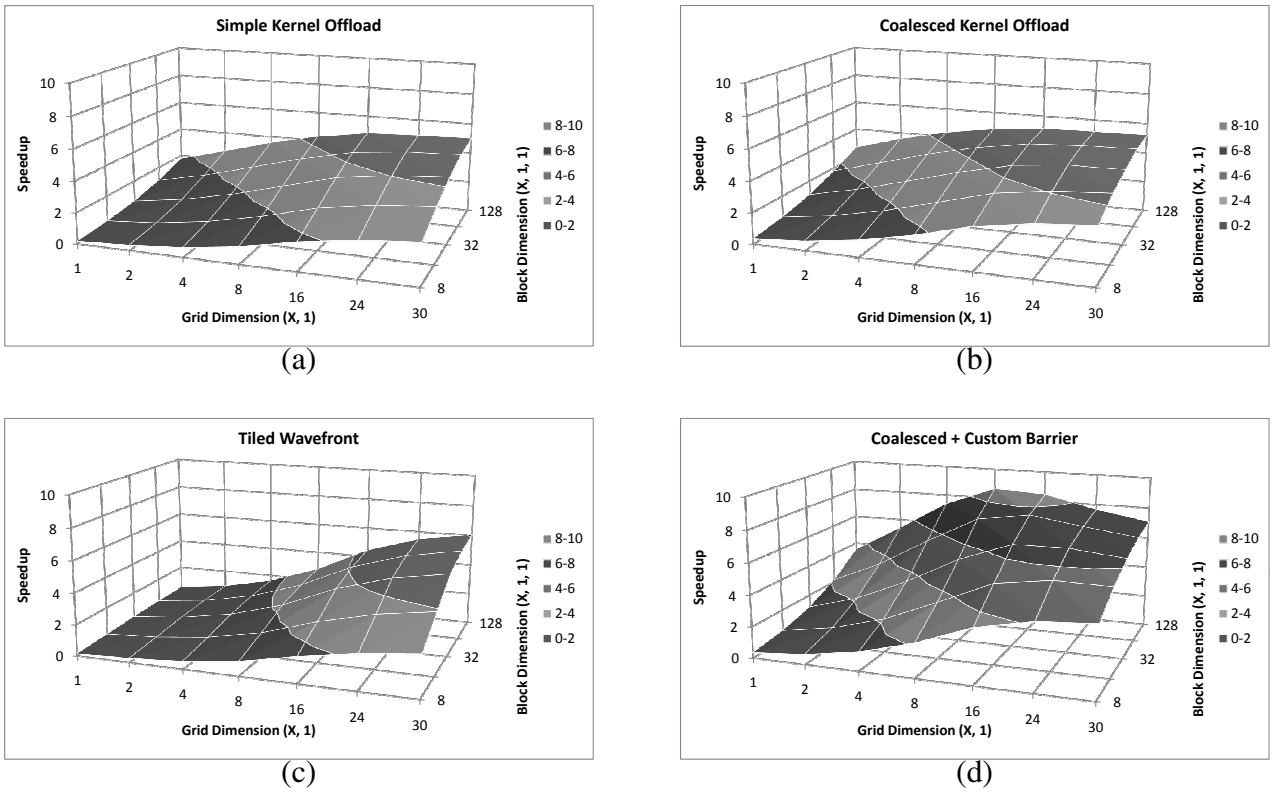

FIGURE 7: Speedup charts.

imental results while running the largest possible problem size, i.e. aligning sequences of 8KB each. Similar results were observed (but not presented here) for other problem sizes as well.

The kernel pre- and post-processing times include allocating memory for the matrix in the global memory and transferring the computed matrix back to the host memory. The non-computation time amounts to 66% of the total execution time at the end of the final optimization technique. In spite of this severe, but unavoidable extra cost, we have been able to achieve very good overall speedup. The backtrace operation has negligible effects as expected.

6 Conclusions

The GPGPU's of today are making rapid strides toward the HPC arena by delivering unprecedented high performance at commodity costs. They are packaged as simple accessories to existing systems, making the GPGPU's as viable accelerator alternatives in many supercomputing environments. However, lack of an explicit inter-block communication mechanism has made the GPGPU useless for several classes of applications that are not data-parallel. Therefore, we contribute a custom barrier implementation that synchronizes all the threads running in the kernel without transferring control out of the GPGPU. We remove the non-deterministic behavior of inter-block interac-

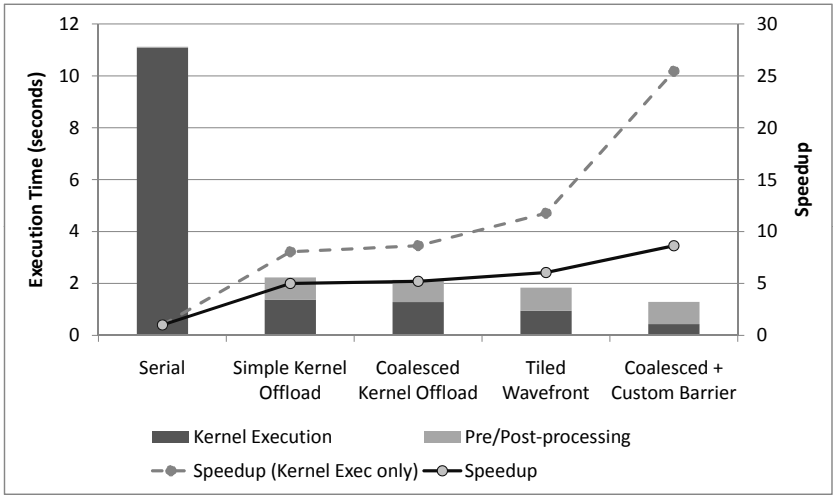

FIGURE 8: Performance comparison chart.

tion within a kernel by maintaining a one-one mapping between the thread blocks and SM's. Our barrier implementation provides a $3\times$ performance improvement over the non-barrier approach with multiple kernel launches.

For our experimental study, we chose to parallelize the Smith-Waterman algorithm, which is a highly popular and optimal biological sequence search algorithm. In this paper, we introduce and present CUDA-SWat, which is a highly parallel implementation of the Smith-Waterman code on the CUDA platform of the nVIDIA GeForce GTX 280 GPGPU. The parallelized code is cleverly optimized in four stages, which include optimal data layout strategies, coalesced memory accesses and blocked data decomposition techniques, to result in a highly efficient mapping of the non-data parallel algorithm to the completely data parallel architecture of the GPGPU.

As a result, we achieved incremental performance improvements for each optimization, with the coalesced + custom barrier method achieving a maximum speed improvement of $8.6\times$ over the serial implementation of the entire application. We also showed that our solution provides $25.5\times$ faster on-chip execution than the naïve implementation. Moreover, our experiments were conducted on realistic problem sizes and provided a *complete* solution, i.e. computed all the matrix entries and did a backtrace to output the actual sequence alignment.

As future work, we intend to integrate of our existing Smith-Waterman implementations into existing sequence alignment toolkits. We would also like to extend our design and implementation methodologies to other wave-front algorithms in general. We plan to explore optimization techniques to parallelize SWat, and other sequence search algorithms, such as BLAST and PatternHunter, within and across multiple GPGPU cards.

References

[1] Ashwin M. Aji, Wu-chun Feng, Filip 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] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] Mark K. Gardner, Wu-chun Feng, Jeremy Archuleta, Heshan Lin, and Xiaosong Ma. Parallel Genomic Sequence-Searching on an Ad-Hoc Grid: Experiences, Lessons Learned, and Implications. *Proc. of ACM/IEEE SC 2006*, Nov. 2006.

[4] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-Sequence Database Scanning on a GPU. *Proc. of the 20th International Parallel and Distributed Processing Symposium*, April 2006.

[5] Yang Liu, Wayne Huang, John Johnson, and Sheila Vaidya. GPU Accelerated Smith-Waterman. In Vassil N. Alexandrov, Geert Dick van Albada, Peter M.A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 188–195. Springer, 2006.

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

[7] NVIDIA. NVIDIA CUDA Programming Guide. *URL:http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf. Accessed: 2008-08-06. (Archived by WebCite at http://www.webcitation.org/5ZrcTQQeV)*, 2007.

[8] Temple Smith and Michael Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[9] John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, and Klaus Schulten. Accelerating Molecular Modeling Applications with Graphics Processors. *Journal of Computational Chemistry*, 28:2618–2640, 2007.