Improving CUDASW++, a Parallelization of Smith-Waterman for CUDA Enabled Devices

Doug Hains, Zach Cashero, Mark Ottenberg,
Wim Bohm and Sanjay Rajopadhye
Colorado State University
Department of Computer Science
Fort Collins, CO 80523
Telephone: (970) 491-5792

***Abstract*—CUDASW++ is a parallelization of the Smith-Waterman algorithm for CUDA graphical processing units that computes the similarity scores of a query sequence paired with each sequence in a database. The algorithm uses one of two kernel functions to compute the score between a given pair of sequences: the inter-task kernel or the intra-task kernel. We have identified the intra-task kernel as a major bottleneck in the CUDASW++ algorithm. We have developed a new intra-task kernel that is faster than the original intra-task kernel used in CUDASW++. We describe the development of our kernel as a series of incremental changes that provide insight into a number of issues that must be considered when developing any algorithm for the CUDA architecture. We analyze the performance of our kernel compared to the original and show that the use of our intra-task kernel substantially improves the overall performance of CUDASW++ on the order of three to four giga-cell updates per second on various benchmark databases.**

I. Introduction

Sequence alignment is an important problem in bioinformatics. In typical usage, a query sequence of nucleotides (DNA, RNA) or amino acids (proteins) is compared to a large database of known sequences in order to determine which database sequences have the most similarity to the query. Smith-Waterman (SW) [8] is a dynamic programming algorithm that always returns the optimal local alignment between two sequences of length m and n with $O(nm)$ time and space complexity.

Because of the time and memory demands of SW, heuristic algorithms such as the Basic Local Alignment Search Tool (BLAST) [1], [2] have been developed to quickly compare a query sequence to a large database of sequences. Such algorithms are much faster than a naive implementation of SW but do not guarantee the optimality of the alignment found.

Implementations of SW for various parallel architectures such as the Compute Unified Device Architecture (CUDA) [4], [5], Streaming SIMD Extensions (SSE) [3], [6] and Cell Broadband Engine Architecture (Cell) [9], [7] have been developed that are capable of outperforming BLAST and other heuristic algorithms [5]. In this paper we present improvements over CUDASW++ [5], the best performing CUDA implementation of SW. These improvements are not only useful for practitioners who require fast and optimal local alignments but also provide general insights into the development process that can be applied to any algorithm for the CUDA architecture.

CUDASW++ uses one of two kernel functions to compare a query sequence to each sequence in an entire database. The length of the database sequence determines which kernel is used: the *inter-task* kernel is used below a threshold length and the *intra-task* kernel is used above the threshold (the names are due to the CUDASW++ authors). The inter-task kernel uses only a single thread to perform the alignment. The intra-task kernel uses one thread block to compute the

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.182

alignment. We observed that the intra-task kernel was not highly optimized.

In the UniProtDB/Swiss-Prot database (called Swissprot from now on) used for performance benchmarks of CUDASW++ [4], [5], 99.88% of the database sequences are below the threshold, meaning the inter-task kernel is used the vast majority of the time. However, altering the threshold even slightly has a major impact in the performance of the overall algorithm, leading us to believe that the intra-task kernel is a bottleneck.

We improve the performance of the intra-task kernel by over 11 times, and when we replace the intra-task kernel in CUDASW++ with an improved implementation of the intra-task kernel, this results in over a 25% speedup in overall running time on the Swissprot database, and over 50% speedup on some databases.

Our results may seem surprising if we note that the intra-task kernel is only being called on slightly under 0.12% of the over 500,000 sequences contained in the Swissprot database. Ideally, it should be tuned to the database, since Swissprot is only one instance of one class of sequence databases. By varying the threshold between intra-task and inter-task kernel calls, we show that CUDASW++, using our improved intra-task kernel, is not nearly as sensitive to longer sequences as is the original implementation. The performance gains for the entire application, obtained by using our improved intra-task kernel can only increase for databases with a higher percentage of sequences over the threshold.

We describe the incremental improvements we made to the intra-task kernel. These improvements show that memory usage is the largest performance factor, thus indicating that SW is a memory bandwidth limited problem. Furthermore, we find several interesting aspects of the CUDA architecture which prevent the NVIDIA CUDA compiler (nvcc) from properly optimizing our code. Hand optimization is required to gain the expected performance.

The remainder of this paper is organized as follows: Section II describes the Smith-Waterman problem and previous parallel implementations, including the details of CUDASW++. Section III describes our implementation of the intra-task kernel and the impact of our incremental improvements. In Section IV, we analyze the differences between our intra-task kernel and the original kernel and compare the overall performance of CUDASW++ using both kernels on multiple databases. Section V summarizes our conclusions and provides several ideas for further improvements to CUDASW++.

II. Background

The sequence alignment problem is to find the optimal alignment between a query sequence q of length m symbols and a database sequence d of length n from some alphabet A. In most bioinformatics applications, A consists of symbols representing amino acids or nucleotides. To determine the score of a particular alignment, a scoring function $w : (a, b) \in A \times A \mapsto \mathbb{N}$ is used to score each pair of symbols in the alignment. A gap open penalty ρ, and a gap extension penalty σ, are commonly used to penalize gaps of unpaired symbols.

The Smith-Waterman (SW) algorithm is a dynamic programming algorithm that determines the optimal local alignment between q and d given w, ρ, and σ. SW fills in three m x n tables E, F and H as shown in (1). The optimal alignment score is given by the maximum score in table H. The dependencies for a cell in H are shown in Figure 1.

$$
\begin{aligned}
E_{i,j} &= \max \left\{ \begin{array}{l} E_{i,j-1} - \sigma \\ H_{i,j-1} - \rho \end{array} \right\} \\
F_{i,j} &= \max \left\{ \begin{array}{l} F_{i-1,j} - \sigma \\ H_{i-1,j} - \rho \end{array} \right\} \\
H_{i,j} &= \max \left\{ \begin{array}{l} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + w(q_i, d_j) \end{array} \right\}
\end{aligned}
\tag{1}
$$

The optimal alignment can be found by backtracking through H, but for comparisons of a

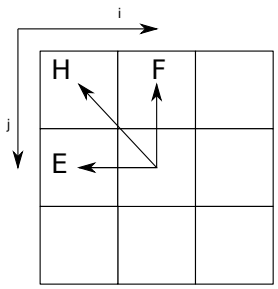


Fig. 1. Dependencies for a cell in H of the Smith-Waterman algorithm.

query sequence to an entire database, we are generally only concerned with the score and not the actual alignment. The table can be computed with computational complexity of $O(nm)$ but, when only the optimal score is required, the space complexity is linear.

A. Advances in SSE

Wozniak [10] and later Rognes and Seeberg [6] proposed an algorithmic improvement that significantly benefits vector implementations. The first implementation [10] suffered from the problem that although the cell updates were vectorized, the similarity function lookup was still sequential. To address this problem, a *query profile* creates a vectorized lookup table for the similarity scores that is unique for a given query sequence [6]. The query sequence is split into pieces with length equal to the vector length used by SSE. The profile is built by storing vectors of similarity scores for each piece of the query sequence compared to all symbols in the alphabet. This allows similarity function lookups to be performed in parallel. We observed that although CUDASW++ uses this optimization and builds a query profile, it does so only in the inter-task kernel. We make use of the query profile in our improved intra-task kernel.

B. CUDASW++

CUDASW++ [5], [4] uses either the intra-task kernel or the inter-task kernel to compare a single query sequence to a database of sequences. Each kernel uses a different strategy to find the optimal score. For each query/database sequence pair, only one kernel is used. If the database sequence length is below 3072, the *inter-task* kernel is used. Otherwise, the *intra-task* kernel is used. Our improvements are solely to the intra-task kernel, but we briefly describe both kernels to make the explanation self-contained.

1) Inter-task: The inter-task kernel uses a single thread to compare a query and a target sequence. It tiles the tables into 8×4 tiles which are computed sequentially by the same thread in row major order. Within a tile, the thread will compute cells in a tile in a column major order, storing all values needed for dependencies within a tile in registers. Once a tile is computed, the bottom row is stored in global memory and the rightmost column is retained in registers to satisfy the dependencies required by bordering tiles.

2) Intra-task: The intra-task kernel uses an entire thread block to find the optimal alignment score between a query sequence and database sequence. No tiling is used and the table is computed in the usual wavefront parallel order. Therefore, all threads in the block are busy only when the length of the minor diagonal (number of points on the wavefront) is a multiple of the number of threads per block, which is 256 by default. Global memory is used to store each wavefront as it is computed and three wavefronts need to be saved at each time step to satisfy the dependencies for the next time step.

C. CUDASW++ Performance Issues

The standard performance metric for SW is "Cell Updates Per Second" (CUPs). We measured the CUPS of each kernel separately and found that the inter-task kernel averages approximately 17 gigaCUPs (GCUPs) while the intra-task kernel averages 1.5 GCUPs when comparing the same query and database sequence on the Tesla C1060.

As previously stated, the intra-task kernel is only run on sequences longer than 3072. Using the default threshold on the Swissprot database, CUDASW++ achieves a performance of 17 GCUPs on a Tesla C1060. When we increase this threshold to 36,000 so that all sequences in

the database are compared using the inter-task kernel, the performance of CUDASW++ drops to 10 GCUPs. Because the threshold is based on sequence length it may appear that the intra-task kernel is, for some reason, better than the inter-task kernel when the database sequences are long. However, this is not the case.

To understand why the intra-task kernel is sometimes faster than the inter-task kernel, it is necessary to first understand how the database sequences are passed to the inter-task kernel. The database is sorted by length and partitioned into groups of s sequences each. As there will be one thread per database sequence in an inter-task kernel call, s is calculated at runtime based on machine parameters to maximize the occupancy, which is a measure of how well the kernel makes use of the available GPU resources. Once all the database sequences below the threshold are sorted and partitioned into groups, CUDASW++ makes one call to the inter-task kernel for each group. The kernel uses s independent threads to find the optimal alignment scores for database sequence in the group. This process continues until all groups have been processed.

What this means in terms of the performance of the inter-task kernel is that the computational time required for a single kernel call is entirely dependent on the length of the longest sequence in that group. Because the threads are synchronized between kernel calls, even if all but one of the threads have finished computing their alignment scores, they all must wait until the last thread is complete before another kernel call can be launched.

The distribution of sequence lengths in a typical protein database, such as Swissprot, resembles a log-normal distribution. If the sequences are sorted by length and broken into groups of size s, the distribution of lengths within a group are somewhat uniform except at the tail of the distribution. For this reason, CUDASW++ uses a threshold to separate the distribution into those sequences that are processed with inter-task and those processed with intra-task.

To examine exactly how the sequence length distribution of a database can impact the performance of the two kernels, we generated several random databases containing s sequences using a log-normal distribution of the sequence lengths. We set the standard deviation between 100 and 1800. Because we used a log-normal distribution the mean to varies from 1000 to 1600. We ran both the intra-task kernel and the inter-task kernel of CUDASW++ on the databases with the same query sequence of length 567.

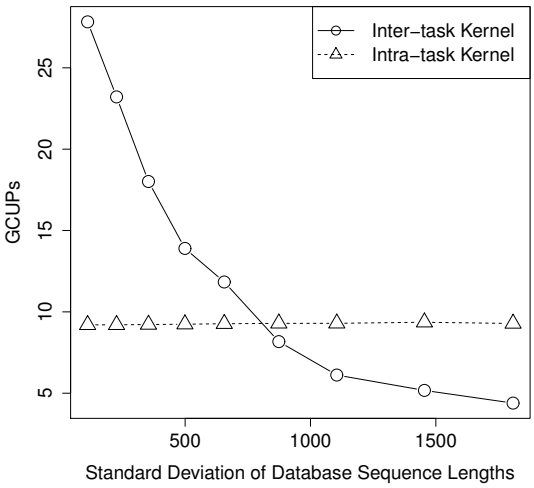


Fig. 2. The performance of the two kernels as a function of the variance of sequence lengths in the database. Observe that unlike the intra-task kernel, performance of the inter-task kernel is very sensitive to the variance. This is essentially a load balancing issue.

Figure 2 shows the GCUPs performance as a function of the standard deviation of sequence lengths in the database and indicates that unlike the intra-task kernel, the inter-task kernel is very sensitive to the distribution of these lengths. The optimal threshold is dependent on the distribution of lengths in the database and this matter is discussed further in Section VI.

When using this threshold of 3072 on the Swissprot database, as the authors of CUDASW++ [4], [5] did, only 0.12% of approximately 500,000 sequences fall above this threshold. Only this relatively small portion of the database is aligned using the intra-task ker-

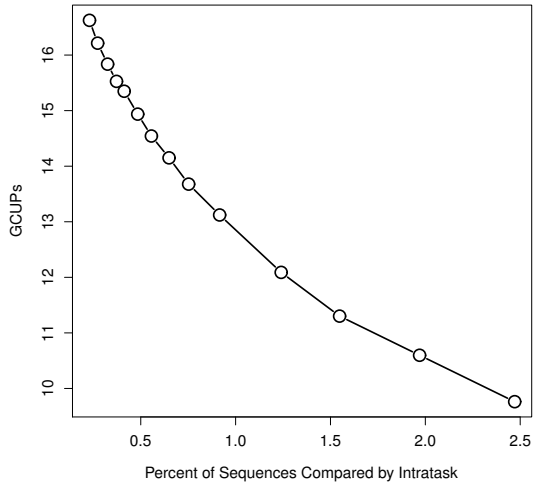


Fig. 3. GCUPs of CUDASW++ on Swissprot as a function of the threshold length. Even a small percentage increase in the number of sequences that use the intra-task kernel causes a huge drop in performance.

nel. Therefore, it may at first seem that, by an argument similar to Amdahl's law, although the intra-task kernel is not as highly optimized as the inter-task kernel, improvements to it would not yield significant performance gains.

To examine how much of an impact the intra-task kernel has on the overall performance, we changed the default threshold of 3072. We measured the GCUPs of the overall algorithm while comparing a query sequence of length 572 to the entire Swissprot database while decreasing the threshold by 100 for each of the 20 runs.

The results, shown in Figure 3 indicate that even small variations in the threshold result in large performance impacts. Therefore, the intra-task kernel is indeed a bottleneck in the algorithm even though it is only called relatively infrequently. Not only will improving the intra-task kernel yield an improvement on the Swissprot database, but it will also have an even larger impact on those databases with a more varied distribution of sequence lengths. The remainder of this paper will detail our improvements and the subsequent performance gains on both NVIDIA Tesla C1060 (Tesla) and Tesla C2050 (Fermi) GPUs.

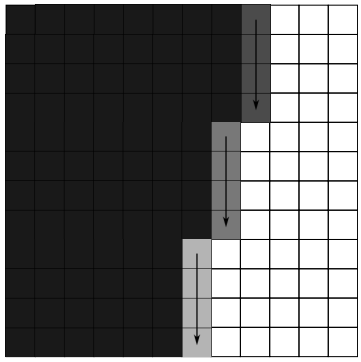


Fig. 4. The tiling of a SW table and the wavefront of threads. Each thread is responsible for four rows and fills them in column-major order. The darkest blocks indicate cells that have been computed. The lighter shaded blocks indicate tiles. The arrows indicate the update order that each thread follows.

III. Improving the Intra-task Kernel

We implemented our own intra-task kernel using a tiling approach. As in the original kernel, one thread block is responsible for computing the SW tables and returning the alignment score for a single query/database sequence pair. The tables are broken into tiles and the tiles are traversed in a wavefront order as shown in Figure 4. We empirically found that a tile size of 4×1 yields the best performance. At each time step, a thread is assigned a tile and is responsible for computing all cells in this tile. After all threads have computed the cells in their tile, the threads are synchronized, the necessary data dependencies are shared, and the threads compute the next tile in their row.

The size of a table that can be calculated in a single pass is limited by the number of threads (n_{th}) and the tile height (t_{height}). We will refer to a set of $n_{th} \cdot t_{height}$ rows as a strip. If a query sequence is longer than the size of the strip, the table must be calculated in multiple passes. Because of the dependencies inherent in the SW algorithm, the bottom row of a strip must be saved for the next pass. Although global memory access is the slowest of all available memories

on the CUDA device, we use global memory for the temporary storage of these values since the amount of memory required depends on the database sequence length, which may be arbitrarily large. Registers are used for the horizontal dependencies and shared memory is used for the vertical and diagonal dependencies between threads.

Our first implementation of this approach did not show any improvements over the original intra-task kernel from CUDASW++. Several incremental improvements allowed us to gain a significant performance increase over the original implementation. We will explain the most significant factors in detail. Unless otherwise noted, the following performance results were obtained using the Swissprot database with a block size of 256 threads and a 4×1 tile size.

A. Global Storage of Local Variables

"Local memory," as defined in CUDA, is global memory that is used for local variables when registers cannot be used. This typically occurs when the number of local variables exceeds the register capacity. However, we found two cases in which global memory was being used instead of registers even when the capacity was not exceeded.

In the first implementation of our kernel, we swapped two register arrays by swapping pointers to those arrays at the end of a loop. Thus, at subsequent iterations, the variable could be referencing one of two arrays. Because of this shallow swap, the NVIDIA compiler (`nvcc`) was unable to map the arrays directly to hardware registers and used global memory. We fixed this issue by swapping each element of the arrays in a deep swap.

We also found an issue with the loop unrolling process of `nvcc`. If texture memory is used within a loop, `nvcc` does not unroll the loop and array variables cannot be mapped to registers. Global memory will be used instead. Since the query profile uses texture memory, we hand unrolled the loops that access the query profile. Fixing both these issues yielded about a two-fold performance increase when the registers were being utilized as intended.

B. Query Profile

In the original CUDASW++ implementation, a query profile stored in texture memory is used in the inter-task kernel but is not used in the intra-task kernel. We applied the query profile to our intra-task implementation so that it stores the similarity scores of four symbols in a single variable. By making our tile height a multiple of four, only a single read is required for every four cells, reducing these memory operations by a factor of four. On pre-Fermi GPUs, the use of texture memory allows for caching, further increasing the significant performance gain we observed by using the query profile. On Fermi GPUs, caching is performed on all global memory transactions, and it is likely that the use of texture memory for the query profile is not required on these devices to achieve the same performance.

C. Parameter Space Exploration

The strip height is a product of two interrelated parameters, thread block size (n_{th}) and tile height (t_{height}). The strip height determines the number of passes required to fill the table and, therefore, the number of times intermediate values are stored and retrieved from global memory. Increasing the strip height will reduce the number of global memory operations, but it will also increase the pipeline latency. We observed that strip height is the relevant parameter to optimize. Therefore, several combinations of n_{th} and t_{height} result in essentially the same performance.

Increasing the tile width does not provide any benefit. The number of shared memory operations is not affected by tile width because the horizontal dependencies are stored in registers and the dependencies of the bottom row of a tile still need to be written to shared memory. Also, an increase in width will not change the number of rows computed in a strip and thus will not affect the number of global memory operations. The number of thread synchronizations is

reduced, but the pipeline latency is increased. Overall, the latency increase dominates the savings from reducing synchronizations. Therefore, a tile width of one is optimal.

IV. Results

Since the performance of CUDASW++ depends on the performance of both the kernels and the choice of the threshold, we first conducted a number of experiments to observe the effect of the threshold on the performance of CUDASW++ using both the original and the improved intra-task kernels.

A. Performance of the Intra-Task Kernels

To determine the optimal values for n_{th} and t_{height}, we ran CUDASW++ with our implementation of the intra-task kernel using 64, 128, 192, 256 and 320 threads per block and tile height of 4 and 8. We found that a strip size of 512 was optimal on the Tesla C1060 and 1024 was optimal on the Tesla C2050. We use these values for all of the subsequent experiments.

We then ran the same experiment used to generate Figure 3 by varying the threshold value using both the improved and the original intra-task kernels. Changing the threshold essentially controls the number and variance of the sequences compared by the inter-task and the intra-task kernels. Figure 5 shows the performance of the original and improved CUDASW++ implementations as the threshold is varied on both the Tesla C1060 and C2050. Figure 5(a) shows the GCUPs as a function of sequences over the threshold in the database and Figure 5(b) shows the percentage of time spent in the intra-task kernel as a function of sequences over the threshold.

Figure 5(a) shows that the performance increase of CUDASW++ using our intra-task kernel improves as the percentage of sequences above the threshold rises. For example, if approximately 0.12% of the database sequences are above the threshold, as is the case in Swissprot with the default threshold value of 3072, the improvement is about 25.0%. But if 2.5% of the sequences are above the threshold, the improved kernel leads to a 67.0% increase in performance. CUDASW++ with our improved intra-task kernel is not as sensitive to the percentage of sequences above the threshold as is CUDASW++ with the original kernel. This is especially important for other databases that may have a larger percentage of sequences over the threshold, as do the majority of databases in Table II.

	Total Memory Transactions	
Kernel	Query Len. 567	Query Len. 5478
Imp. Kernel	13,828	4,233,197
Orig. Kernel	28,345,473	291,179,739

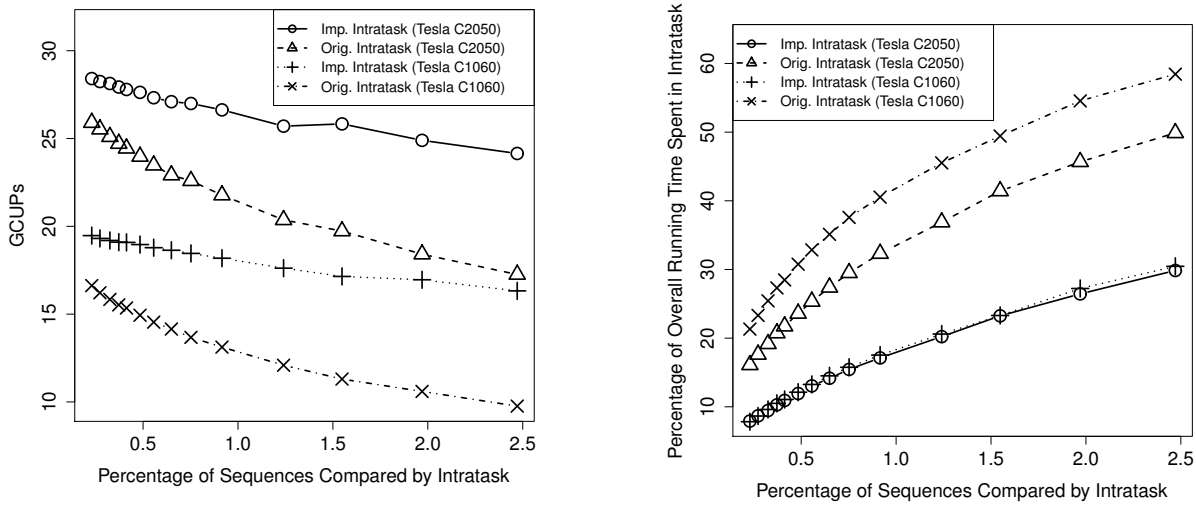
TABLE I. Number of total global memory accesses performed by both kernels on queries of two different sizes against the Swissprot database.

Figure 5(b) shows that CUDASW++ using the original kernel spends up to 50% of its running time in the intra-task kernel when run on a Tesla C1060, but shows an improvement when run on a Tesla C2050. While our improved implementation reduces the percentage of time spent in the intra-task kernel kernel by more than half, there is no significant difference between the C1060 and C2050.

This is because the C2050 introduces two levels of cache to global memory accesses: an L1 cache, which is shared by threads of the same multi-processor, and an L2 cache, which is shared by all threads. While both the original intra-task kernel and the improved intra-task kernel can benefit from the L1 and L2 cache, the original intra-task kernel makes many more global memory accesses than the improved kernel, which allows it to gain more of a benefit from caching.

We used a profiler to count the number of global memory accesses of both the improved and the original kernel. We used a query sequence of length 567 and a query sequence of length 5478 and ran each against the Swissprot database. The total number of memory transactions are reported in Table I.

We generated the improved intra-task kernel results using 256 threads per block and a tile height of four. Thus, a strip size is 1024. Five full



(a) GCUPs

(b) Percentage of time in intra-task

Fig. 5. The four curves in each figure show (a) the performance in GCUPs and (b) the percentage of time spent in the intra-task kernel by CUDASW++ on C1060 and C2050 with and without our improved kernel, as a function of the percentage of sequences compared by the intra-task kernel. Our kernel always improves performance. The gain is at least 6.7% on the C2050 (17.5% on the C1060) and as much as 39.3% on the C2050 (67.0% on the C1060). We ran this experiment on Swissprot with a single query of length 576.

passes are required on a query size of 5478, and we can approximate about 10^6 global memory transactions per strip from the data of Table I. The original kernel requires global memory accesses for *every* cell updated. Therefore, dividing the values of the original kernel by the respective query lengths in Table I gives us an approximation of 50,000 global memory accesses per symbol in the query length when run against the Swissprot database. This means the original kernel performs approximately 5×10^7 global memory accesses per 1024 query symbols compared to 10^6 accesses in the improved kernel.

The improved kernel needs to communicate data to and from global memory for each strip of 1024 elements in the query sequence, using the default parameters. The original kernel needs to perform global communications for each element of the query sequence. This is the main source of our performance gain and also the reason for the improvement found in the original intra-task kernel on the Tesla C2050 as seen in Figure 5(b).

Because the L1 and L2 caches only affect global memory transactions, the performance gained by caching is only significant in the original kernel because of the large number of global memory transactions performed by that kernel. To show that the cache is indeed responsible for the improvement shown in Figure 5(b), we performed the same experiment on a Tesla C2050 with both of the L1 and L2 caches turned off. These results are shown in Figure 6. This shows that the improvements gained by the original kernel on a Tesla C2050 are almost completely attributed to the cache.

Any kernel making extensive use of global memory can gain a performance boost by the L1 and L2 cache introduced in the Fermi line of GPUs, but it is not enough to compete with a kernel which eliminates the majority of the transactions altogether. Even though L1 has the same throughput as shared memory, if the access pattern results in L1 misses, tiling for the explicit use of shared memory is ideal.

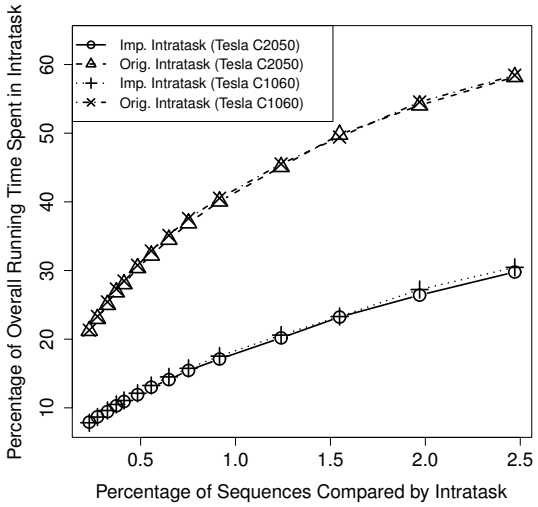


Fig. 6. Results of varying the threshold between inter-task and intra-task kernel calls on the Swissprot database with L1 and L2 cache turned off.

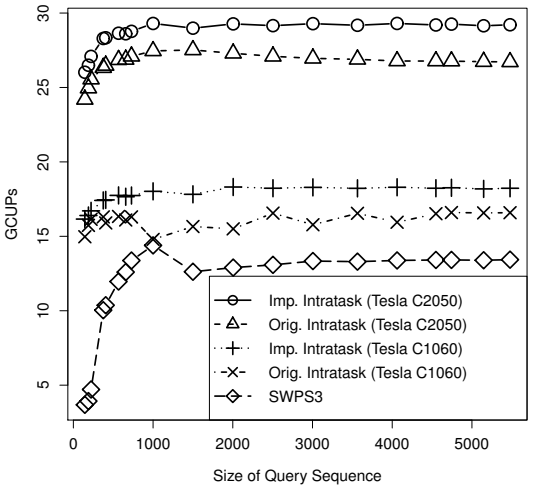


Fig. 7. GCUPs of CUDASW++ with our improved intra-task kernel and with the original intra-task task kernel on the Swissprot database.

B. Performance of the Complete Application

To determine the performance gain of the overall CUDASW++ algorithm using our improved intra-task kernel, we ran CUDASW++ with the improved kernel and with the original kernel against the Swissprot database. We used the same query sequences from the original CUDASW++ study [4]. The length of the query sequences ranges from 144 to 5478 residues in length.

We ran the algorithms on a single Tesla C1060 GPU and a single Tesla C2050 GPU. While we did not run the algorithm on multiple GPU cards, we note that the kernel tasks are independent, and thus the running time will scale almost linearly with the number of GPUs available, as seen in previous studies [4], [5]. For both GPUs, we used CUDA version 3.2. We measure the GCUPs from multiple query sequences against the Swissprot database. As a point of reference, we also ran SWPS3 [9], a vectorized SSE implementation of Smith-Waterman using four cores of an Intel Xeon processor clocked at 2.33 GHz. The results are shown in Figure 7.

A benefit of both versions of CUDASW++ is that it is not as sensitive to query length as SWPS3. This is mainly due to the need of SWPS3 to correct errors which are a result of a vertical traversal through the SW tables. The correction requires at least another pass, which is known as the Lazy-F loop that is required for efficient vectorization [3], [9]. CUDASW++ outperforms SWPS3 at all points tested using one GPU card. We note that while SWPS3 can be run on more processors to increase the performance, CUDASW++ can similarly be run on multiple GPUs. Using eight x86 cores will give SWPS3 roughly a two times increase in speed; CUDASW++ will likewise see a twofold increase if two GPUs are used.

When our improved intra-task kernel is incorporated into CUDASW++, the performance is consistently higher than the original CUDASW++ by an average of about four GCUPs or 25%. In addition, the performance is consistent for query lengths above 1000. In general, our improved CUDASW++ implementation is less sensitive to varying query lengths and outperforms both the original CUDASW++ implementation and SWPS3.

In practice, many different protein, RNA, or DNA databases are routinely used for compar-

ison purposes. To test our improved kernel on others databases, we chose several databases from various sources. The databases used were the most recent versions as of this writing. Table II shows the performance of CUDASW++ with both the original intra-task kernel and the improved version on the Tesla C1060 and Tesla C2050.

We see that the improved intra-task kernel increases the performance of CUDASW++ on all databases tested. The performance gain is typically more pronounced when there are more sequences over the threshold, with the lowest performance gain occurring on the TAIR database with only 0.06% of the sequences over the threshold. This is to be expected since the inter-task kernel remains unchanged and very few sequences are being compared by the intra-task kernel. The gains of the improved intra-task kernel are also more noticeable on the Tesla C1060 than the C2050. Again, this is a result of the performance boost gained by the original implementation due to the global memory caching available on the Fermi.

However, we have not addressed another benefit our improved intra-task kernel provides the CUDASW++ algorithm. By increasing the performance of the intra-task kernel, we have changed the tradeoff point where intra-task is better than inter-task as the distribution of sequence lengths becomes non-uniform, i.e. the intersection point shown in Figure 2. This means the algorithm may benefit from lowering the threshold on some databases from the default 3072.

We decreased the threshold from 3072 to 1500 and reran CUDASW++ with our improved kernel on the TAIR database. At this threshold setting, 0.96% of the sequences were over the threshold. For query sequences longer than 144, the performance increased to over 21 GCUPs in all cases on the C2050. This is close to a 4 GCUPs increase over the performance reported in Table II by simply decreasing the threshold. Exploratory experiments into the optimal threshold value have shown us that we can gain similar performance increases in almost all databases by lowering the threshold when CUDASW++ uses our improved kernel.

V. Conclusions

In this paper we started with a detailed analysis of the CUDASW++ algorithm and found the original intra-task kernel to be a large bottleneck. Although the intra-task kernel is slower than the inter-task kernel when performing at its peak, we have shown that the performance of the inter-task kernel is highly dependent on the distribution of sequence lengths in the database, but the intra-task kernel does not have this dependency. Thus, when the distribution of sequence lengths deviates far enough from a uniform distribution, the intra-task kernel becomes necessary.

Even though the intra-task kernel is only used to compare the query sequence to a small portion of the database sequences, we have shown when as few as 2% of the sequences in the database are compared using the intra-task kernel, more than 50% of the overall running time is used by this kernel. We have shown that the bottleneck is primarily due to the large number of global memory accesses performed by the intra-task kernel.

We have presented an improved implementation of the intra-task kernel used in CUDASW++, which substantially improves on the performance of the original intra-task kernel on both Tesla and Fermi GPUs. Our kernel was designed to tile the computation in order to reduce the number of global memory accesses and make use of the shared memory. We have shown that our kernel achieves an approximate 50:1 reduction in the number of global memory accesses on the Swissprot database. We have also found and described our workarounds for several non-trivial instances when global memory was being used instead of registers as we had intended. Our improved kernel is pleasantly parallel [1] at the scope of kernel calls, allowing CUDASW++ with our improved implementation

[1] many authors call this embarrassingly parallel

Database	% over Thresh.	GPU	Kernel	GCUPs for Query Sequences of Length					
				144	1000	2005	3005	4061	5478
Ensembl Dog Proteins	.53%	C1060	Original	7.85	8.78	8.82	9.05	9.19	9.72
			Improved	8.87	11.37	11.53	11.51	11.50	11.41
		C2050	Original	12.01	15.44	15.46	15.27	15.09	15.02
			Improved	13.28	17.29	17.35	17.38	17.41	17.29
Ensembl Rat Proteins	.35%	C1060	Original	7.90	8.61	8.68	8.87	9.02	9.55
			Improved	8.83	10.98	11.14	11.10	11.11	11.04
		C2050	Original	11.77	14.61	14.66	14.47	14.32	14.26
			Improved	12.98	16.11	16.16	16.19	16.23	16.16
NCBI RefSeqHuman Proteins	.56%	C1060	Original	6.71	7.12	7.70	7.91	8.04	8.52
			Improved	7.77	10.34	10.43	10.42	10.40	10.31
		C2050	Original	11.19	15.10	15.06	14.83	14.61	14.54
			Improved	12.37	17.48	17.54	17.61	17.60	17.42
NCBI RefSeq Mouse Proteins	.54%	C1060	Original	6.77	7.72	7.93	8.11	8.24	8.73
			Improved	7.91	11.05	10.95	10.88	10.87	10.76
		C2050	Original	9.44	12.49	12.40	12.21	12.10	12.06
			Improved	10.29	14.21	14.18	14.14	14.17	14.08
TAIR Arabidopsis Proteins	.06%	C1060	Original	9.83	10.50	10.74	10.83	10.87	11.01
			Improved	10.03	11.04	11.24	11.30	11.28	11.28
		C2050	Original	15.29	17.40	17.53	17.63	17.52	17.54
			Improved	15.31	17.69	17.81	17.94	17.89	17.89
UniProtDB/Swiss-Prot	.12%	C1060	Original	14.98	14.79	15.50	15.77	15.95	16.59
			Improved	16.16	18.02	18.31	18.29	18.30	18.23
		C2050	Original	24.20	27.46	27.29	26.97	26.77	26.71
			Improved	26.02	29.29	29.26	29.28	29.19	29.20

TABLE II. Results for both versions of CUDASW++ on several databases.

to linearly scale with multiple GPUs as does the original CUDASW++ [5].

Our results show that using our improved kernel yields a significant performance boost to CUDASW++ when run on multiple real world databases of protein sequences on both a Tesla C1060 and a Tesla C2050. Even though the Tesla C2050 provides two levels of caching to increase the performance of global memory accesses and the L1 cache has the same throughput as shared memory, we have shown that tiling for explicit shared memory usage outperforms the original intra-task kernel despite the performance gained by caching.

VI. Future Work

We believe that there are several key improvements to the algorithm that would cause significant increases in the performance. One, the global memory accesses can be coalesced for improved performance. Currently, the last thread in the bottom row of the strip must write out its values to global memory one at a time. This can be improved by adding an intermediary step to first write these values to a shared memory buffer. Once this buffer is full, all threads in the block would be responsible for moving these values from shared memory to global memory in a coalesced fashion. The same concept can be applied to the global memory reads when starting a new strip. It is also possible to use the increased amount of shared memory on the Fermi to completely eliminate global memory for shorter sequences. Our implementation uses such a small amount of global memory that it can all be done in shared memory for sequence lengths less than 10,000.

Since we know from our tile height experiments that the latency for filling and flushing the pipeline can impact the performance, the algorithm can be changed so that only one pipeline fill/flush is required for the entire alignment, rather than one fill/flush for every strip. When a thread finishes its last tile on a strip, it can immediately start working on the next strip.

A more high-level change can be incorporated

that would streamline the host to device memory copy. Rather than copy the entire database to device memory before starting any alignments, the algorithm could copy over a small portion of the database and start performing alignments on those sequences. Then the rest of the database can be copied in the background, essentially hiding the majority of the host to device memory transfer time. Additionally, this would allow large databases to be used, such as the NR database or TrEMBL, which are currently too large to fit in the memory of a single Tesla C1060 or C2050.

Finally, as shown by our initial experiments conducted by varying the threshold on the TAIR database, the automatic detection of the optimal threshold value could have a significant impact on the performance. It is possible to characterize the relative performance of the inter-task and intra-task kernels based on the mean and maximum lengths of a given group of sequences. In this way, during the database preprocessing step, we can find the transition point where the intra-task kernel will outperform the inter-task kernel to determine the optimal threshold value.

VII. Acknowledgement

The authors would like to thank the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, for the use of their resources.

References

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[2] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389, 1997.

[3] M. Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156, 2007.

[4] Y. Liu, D.L. Maskell, and B. Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73, 2009.

[5] Y. Liu, B. Schmidt, and D.L. Maskell. CUDASW++ 2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*, 3(1):93, 2010.

[6] T. Rognes and E. Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699, 2000.

[7] W.R. Rudnicki, A. Jankowski, A. Modzelewski, A. Piotrowski, and A. Zadrożny. The new SIMD implementation of the Smith-Waterman algorithm on Cell microprocessor. *Fundamenta Informaticae*, 96(1):181–194, 2009.

[8] TF Smith and MS Waterman. Identification of common molecular subsequences. *J. Mol. Bwl*, 147:195–197, 1981.

[9] A. Szalkowski, C. Ledergerber, P. Krahenbuhl, and C. Dessimoz. SWPS 3 – fast multi-threaded vectorized Smith-Waterman for IBM Cell/B. E. and $\times$ 86/SSE 2. *BMC Research Notes*, 1(1):107, 2008.

[10] A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Bioinformatics*, 13(2):145, 1997.