

# Exploring the Viability of the Cell Broadband Engine for Bioinformatics Applications

Vipin Sachdeva, Michael Kistler, Evan Speight and Tzy-Hwa Kathy Tzeng

{*vsachde, mkistler, speight, tzy*}@us.ibm.com

*IBM Austin Research Lab      IBM Systems and Technology Group*  
*Austin, TX      Poughkeepsie, NY*

---

## Abstract

This paper evaluates the performance of bioinformatics applications on the Cell Broadband Engine (Cell/B.E.) recently developed at IBM. In particular we focus on three highly popular bioinformatics applications – FASTA, ClustalW, and HMMER. The characteristics of these bioinformatics applications, such as small critical time-consuming code size, regular memory accesses, existing vectorized code and embarrassingly parallel computation, make them uniquely suitable for the Cell/B.E. processing platform. The price and power advantages afforded by the Cell/B.E. processor also make it an attractive alternative to general purpose processors. We report preliminary performance results for these applications, and contrast these results with the state-of-the-art hardware.

---

## 1 Computational Biology and High-Performance Computing

With the discovery of the structure of DNA and the development of new techniques for sequencing the entire genome of organisms, biology is rapidly moving towards a data-intensive, computational science. Biologists search for biomolecular sequence data to compare with other known genomes in order to determine functions and improve understanding of biochemical pathways. Computational biology has been aided by recent advances in both algorithms and technology, such as the ability to sequence short contiguous strings of DNA and from these reconstruct the whole genome [1,28,29]. In the area of technology, high-speed micro array, gene, and protein chips [25] have been developed for the study of gene expression and function determination. These high-throughput techniques have led to an exponential growth of available genomic data. As a result, the computational power needed by bioinformatics applications is growing exponentially and it is now apparent that this power will not be provided solely by traditional general-purpose processors.

The recent emergence of accelerator technologies like FPGAs, GPUs and specialized processors have made it possible to achieve an order-of-magnitude improvement in execution time for many bioinformatics applications compared to current general-purpose platforms.

Although these accelerator technologies have a performance advantage, they are also constrained by the high effort needed in porting the application to these platforms.

In this paper, we focus on the performance of sequence alignment and homology applications on the Cell Broadband Engine. The Cell Broadband Engine (Cell/B.E.) (tm) processor, jointly developed by IBM, Sony and Toshiba, is a new member of the IBM Power/PowerPC processor family [13]. The primary target is the PlayStation 3 (tm) game console, but its capabilities also make it well suited for various other applications such as visualization, image and signal processing and a range of scientific/technical workloads.

In previous research, we presented *BioPerf* [2], a suite of representative applications assembled from the computational biology community. Using our previous workload experience, we focus on three critical bioinformatics applications – FASTA (*ssearch34*), ClustalW (*clustalw*), and HMMER (*hmmpfam*). The *ssearch34* program from the FASTA package uses the Smith-Waterman kernel to perform pairwise alignment of gene sequences, ClustalW is used for multiple sequence alignment, and (*hmmpfam*) from HMMER searches a query sequence against a hidden markov model family.

A key issue in porting any application to the Cell/B.E. processor is programmer productivity: the highly specialized nature of the Cell/B.E. processor, such as multiple vectorized cores as well as programmer-managed caches, make the porting of an application to the Cell/B.E. platform more difficult than other general purpose platforms. However, our experience with these applications indicates that this effort can result in significant performance improvements over what can be achieved with current state-of-the-art general purpose microprocessors. In this paper, we discuss our implementations of FASTA, ClustalW and HMMER describing the changes required to utilize the capabilities of the Cell/B.E. processor and the resulting performance improvements.

## 2 The Cell Broadband Engine Processor

The Cell Broadband Engine processor is a heterogeneous, multi-core chip optimized for compute-intensive workloads and broadband, rich media applications. The Cell/B.E. is composed of one 64-bit Power Processor Element (PPE), 8 specialized co-processors called Synergistic Processing Elements (SPEs), a high-speed memory controller and high-bandwidth bus interface, all integrated on-chip. The PPE and SPEs communicate through an internal high-speed Element Interconnect Bus (EIB). The memory interface controller (MIC) provides a peak bandwidth of 25.6GB/s to main memory. The Cell/B.E. has a clock speed of 3.2GHz and theoretical peak performance of 204.8 GFLOPS (single precision) and 21 GFLOPS (double precision).

The PPE is the main processor of the Cell/B.E. and is responsible for running the operating system and coordinating the SPEs. It is a traditional 64-bit PowerPC (PPC) processor core with a VMX unit, 32 KB Level 1 instruction cache, 32 KB Level 1 data cache, and 512 KB Level 2 cache. The PPE is a dual issue, in-order execution design, 2-way simultaneous multi-threading (SMT).

Each SPE consists of a Synergistic Processing Unit (SPU) and a Memory Flow Controller (MFC). The SPU is a RISC processor with 128 128-bit single-instruction, multiple-data (SIMD) registers and a 256KB Local Store (LS). All SPU instructions are inherently SIMD operations that operate at four different granularities: 16-way 8b integers, 8-way 16b integers, 4-way 32b integers or single-precision floating-point numbers, or 2 64b double-precision floating point numbers. The 256 KB local store is used to hold both the instructions and data of an SPU program. The SPU cannot access main memory directly. The SPU issues DMA commands to the MFC to bring data into the LS or write the results of a computation back to main memory. Thus, the contents of the LS are explicitly managed by software. The SPU can continue program execution while the MFC independently performs these DMA transactions.

There are currently other efforts for porting computational biology to the Cell/B.E. processor: Folding@Home, a distributed computing project for protein folding was recently ported to the Cell/B.E. processor [22] with outstanding results. The Charm++ runtime system, used for NAMD simulations, is currently in the process of being ported to the Cell [14]. There are also preliminary results for the performance of BLAST on the Cell/B.E. processor[19]. Terrasoft Solutions ([www.terrasoftsolutions.com](http://www.terrasoftsolutions.com)) has recently built a facility for about 2500 playstations, to be used for bioinformatics research using popular gene-finding and sequence alignment software, available free for national laboratories and other university professionals. Folding@Home has also used the computational power of GPU's [21]. All this work points to the increasing use of accelerator technologies for achieving performance from bioinformatics kernels not available from general-purpose computing platforms. The downside of employing these technologies is that the implementation is highly specific to the accelerator technology being employed, and the effort of porting the code is non-trivial due to the non-emergence of any productive programming platform so far.

### 3 Methodology

In our work, we have ported portions of three popular bioinformatics applications to the Cell/B/E processor – FASTA, ClustalW, and HMMER. These three applications along with BLAST (Basic Local Alignment Search Tool) encompass the sub-field of computational biology known as sequence analysis. Sequence analysis is one of the most commonly performed tasks in bioinformatics and refers to the collection of techniques used to identify similar or dissimilar regions of RNA, DNA, or protein sequences. To begin our analysis for porting the applications to the Cell/B.E. processor, we used the *gprof* tool to determine the most time-consuming functions for each application. Figure 1 shows the execution profile from *gprof* for our three applications and the BLAST sequence analysis application. The results shown in Figures 1 are for the largest *class-C* inputs included in the BioPerf suite ([www.bioperf.org](http://www.bioperf.org)). These results indicate that all three of our applications spend more than half of their execution time in a single function: *dropgsw* for FASTA, *forward\_pass* for ClustalW, and *P7Viterbi* for HMMER.

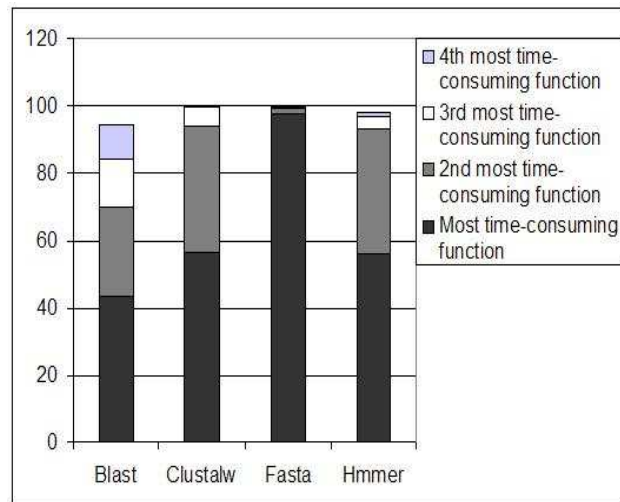


Fig. 1. Function-wise breakout of BLAST, ClustalW, FASTA, and HMMER

This is a useful fact for an implementation for the Cell/B.E. processor, as it implies that we might obtain a significant speedup for these applications by only porting these functions to run on the SPUs. In addition, all these applications perform multiple alignments, which are completely independent and thus can be performed in parallel across the 8 SPUs of the Cell/B.E. processor. This is also a common trait among many bioinformatics applications, which make them suitable for porting to the Cell/B.E. processor with relatively little effort (compared to other applications) but potentially large performance benefits.

Another important factor in porting applications to the Cell/B.E. is exploiting the *Single Instruction Multiple Data* (SIMD) nature of the SPU instruction set. SIMD instructions perform a single operation on multiple input values and produce multiple results, which can significantly improve performance for computationally intensive applications. Most high-performance, general purpose processors support SIMD as an extension of the base instruction set, the most common being the AltiVec SIMD support for PowerPC processors and SSE for x86 architecture processors. However, applications frequently must be recoded to exploit SIMD features, since data layout and computation ordering are critical to achieving the potential performance improvements of SIMD execution. This typically involves the use of *intrinsics*, which are high-level language extensions to access the underlying SIMD instructions of the processor.

In our work, two of the three applications we chose for our study already have open-source AltiVec/SSE implementations for their key kernel functions: *dropgsw* for FASTA and *P7Viterbi* for HMMER. In addition, IBM Life Sciences has developed a SIMD implementation of the *forward\_pass* kernel of ClustalW. These existing SIMD implementations make a port to the SPUs much simpler, as many of the AltiVec intrinsics map one-to-one to the SPU intrinsics. The intrinsics which do not map one-to-one can be executed using multiple instructions on the SPU. The *vmx2spu.h* file in the Cell/B.E. SDK provides SPU implementations for all the AltiVec intrinsics that do not have a corresponding SPU intrinsic.

To evaluate our implementations of these applications for the Cell/B.E. processor, we

compare the performance of the Cell/B.E. implementation to versions of the application built for current high-performance, superscalar processors. Unless otherwise stated, we compiled our implementations for the Cell/B.E. with xlc version 8.1 using the compilation flag `-O3`, which gave equal or better performance compared to gcc for both SPU and PPU. We executed the Cell/B.E. implementations on a two processor QS20 blade running at 3.2 Ghz with 1GB of XDR memory. For PowerPC G5, we used the gcc compiler with `-O3 -mcpu=G5 -mtune=G5` compiler options, and executed the code on a PowerPC G5 with two PPC 970 processors running at 2 GHz, each with 512 KB of L2 cache, and total system memory of 3.5 GB. We built the applications for Opteron and Woodcrest using gcc with the `-O3` flag. The Opteron implementation was executed on a dual-core 2.4 Ghz processor, with 1024 KB of L2 cache per processor and 8 GB of RAM. We executed the Woodcrest implementation on a two processor, dual-core 3.0 GHz Intel Xeon(R) CPU, with 4MB of L2 cache size of L2 per processor and 16 GB of memory.

In the following sections, we discuss the implementation of each application in detail along with issues in porting and bottlenecks that exist for each application. For further details on FASTA, ClustalW, and HMMER, please refer to [23], [27], and [5] respectively.

## 4 FASTA on the Cell/B.E. Processor

### 4.1 Overview

The FASTA [23] package is designed to perform fast similarity searching where uncharacterized but sequenced “query” genes are scored against vast databases of characterized sequences. FASTA uses heuristics to identify sequences in the database that have high similarity to the query sequence, and then applies the classic Smith-Waterman alignment algorithm to compute a similarity score between the query sequence and each sequence selected from the database. Matches are identified by comparisons which have scores greater than a threshold value. Such a scoring mechanism is useful for capturing a variety of biological information including identifying the coding regions of a gene, identifying similar genes, and assessing divergence among other sequences. An efficient implementation of a Altivec-enabled Smith-Waterman developed by Eric Lindahl is already included in the FASTA package.

The Smith-Waterman alignment algorithm is the classic approach to the problem of pairwise alignment, which is one of the most frequently employed and well-known techniques in sequence analysis. Pairwise alignment compares two sequences and produces a score that indicates the degree of similarity between the two sequences. Each position in the two sequences is compared; matches increase the score by a constant or nucleotide (amino-acid) specific value while mismatches decrease the score by a gap or mismatch penalty. The problem of comparing two entire sequences is called *global alignment*, and comparing portions of two sequences is called *local alignment*. Dynamic programming techniques can be used to compute optimal global and local alignments. Smith and Waterman [26] developed one such dynamic programming algorithm (referred to as “Smith-

Waterman”) for optimal pairwise global or local sequence alignment. For two sequences of length  $n$  and  $m$ , the Smith-Waterman algorithm requires  $O(nm)$  sequential computation and  $O(m)$  space. The pseudo code in Algorithm 1 describes the Smith-Waterman approach.

**Data** : (1) Two sequence  $S_1$  and  $S_2$  of length  $m$  and  $n$  respectively  
(2) Gap initiation penalty of  $W_g$  and a gap extension penalty of  $W_s$   
(3) Four two-dimensional matrices  $V, E, F, G$  for storing the intermediate values  
(3.1)  $V(i, 0) = E(i, 0) = -W_g - iW_s$   
(3.2)  $V(0, j) = F(0, j) = -W_g - jW_s$   
(4)  $W_{ij}$  which is the score of aligning the  $i^{th}$  character of the  $S_1$  sequence with the  $j^{th}$  character of the  $S_2$  sequence.

**Result** : The alignment score  $V(i, j)$  obtained by aligning the  $S_1$  sequence with the  $S_2$  sequence.

**begin**  
 $i = 0$   
**while**  $i++ \leq m$  **do**  
 $j = 0$   
**while**  $(j++ \leq n)$  **do**  
(1)  $G(i, j) = V(i - 1, j - 1) + W_{ij}$   
(2)  $E(i, j) = \max[E(i, j - 1), V(i, j - 1) - W_g] - W_s$   
(3)  $F(i, j) = \max[F(i - 1, j), V(i, j - 1) - W_g] - W_s$   
(4)  $V(i, j) = \max[E(i, j), F(i, j), G(i, j), 0]$

**end**

**Algorithm 1:** Smith-Waterman algorithm for Aligning Two Sequences

Due to the quadratic complexity of the Smith-Waterman [26] algorithm, various attempts have been made to reduce its execution time. One approach has employed MMX and SSE technology common in today’s general purpose microprocessors to achieve significant speedups for Smith-Waterman. Other approaches utilize multiple processors to perform parts of the computation in parallel. Parallel strategies for Smith-Waterman and other dynamic programming algorithms range from fine-grained ones in which processors collaborate in computing the dynamic programming matrix cell-by-cell [18] to coarse-grained ones in which query sequences are distributed amongst the processors with no communication needs [8].

Recently, several efforts have employed FPGA’s [4] or GPU’s [16] for computing Smith-Waterman alignments. An implementation of Smith-Waterman has also been developed for the Crays XD1, a hybrid platform of an AMD Opteron and FPGA connected through a hypertransport link [17]. Finally, specialized single-purpose hardware [9] has been developed for the Smith-Waterman and other dynamic programming algorithms in sequence alignment which can achieve an order of magnitude improvement in performance over most contemporary processors [24].

## 4.2 Cell Implementation

For porting the FASTA package to the Cell/B.E. processor, we chose to focus our efforts on the Smith-Waterman alignment operation since this typically accounts for the majority of the execution time of this application. The FASTA package includes a separate program, *ssearch34*, that simply performs the Smith-Waterman alignment operation on a specified set of sequences.

As noted above, our gprof analysis showed that over x% of the execution time of *ssearch34* is spent within the function *dropgsw*. The function *dropgsw* is simply a wrapper function that invokes the implementation of the Smith-Waterman kernel that is appropriate for the underlying architecture. Due to the similarity in API set, we chose the Smith-Waterman Altivec kernel *smith\_waterman\_altivec\_word* as the starting point for our Cell/B.E. implementation. We began porting this kernel to the SPUs by converting all the Altivec intrinsics to SPU intrinsics. Many of the Altivec intrinsics could be converted to SPU intrinsics with little effort (replacing the *vec\_* of Altivec to *spu\_* for the SPU), and thus such an implementation could be pretty straightforward. However, there are three Altivec intrinsics used in this kernel that are not available on the SPU, and must be implemented using multiple SPU instructions:

- *vec\_max*: This intrinsic computes the element-wise maximum of two vectors and stores the result in the output vector. The *vec\_max* operation on the SPU is implemented using *spu\_cmpgt* to create a mask from the comparison of the two input vectors, and *spu\_sel* using this mask to extract the greater of the two vectors. This intrinsic must thus required two instructions for the SPUs to be implemented. For more details of the SPU intrinsics *spu\_cmpgt* and *spu\_sel*, please refer to [11].
- *vec\_subs*: This intrinsic performs saturated subtraction, meaning that if any element of the result vector is negative, that element is set to zero. This is a very important operation for the Smith-Waterman kernel, since it needs a non-negative value at every matrix cell for local alignment. The *vec\_subs* operation on the SPU is a costly operation, requiring one *spu\_sub*, two *spu\_sel*, two *spu\_splats*, two *spu\_rlmaska*, a *spu\_nand* and a *spu\_nor* operation, which is a total of 9 instructions.
- *vec\_adds*: This intrinsic performs saturated addition, similiar to saturated subtraction. This intrinsic also needs 9 instructions for execution, thus making it a very costly operation.

To execute the Smith-Waterman kernel on the SPU, we must use DMA operations to transfer the kernel inputs from the main memory area into the SPUs local storage, and then transfer the results of the kernel back to main storage at the end of the computation. As indicated in Algorithm 1, the kernel inputs are the two sequences  $S_1$  and  $S_2$ , which are typically stored as arrays of short ints (16 bits), the gap penalties  $W_g$  and  $W_s$ , which are just two scalars, and the alignment score matrix  $W_{ij}$ , whose size depends on the size of the alphabet used in the sequences  $S_1$  and  $S_2$ .

However, the Altivec implementation transforms the alignment score matrix into an alternate structure to allow multiple elements to be computed in parallel, thus leveraging

the SIMD capabilities of the AltiVec hardware. An array is constructed in which every character of sequence  $S_1$  is scored against each of the possible characters of sequence  $S_2$  (e.g.  $a, c, g, t$  in case of DNA sequences). For the Cell/B.E. implementation, we perform this transformation on the PPU and then transfer it to the SPU. We chose this approach because this computation is inherently scalar in nature and thus poorly suited to the SPU. However, this requires a much larger transfer of data to the SPU LS. If  $S_1$  and  $S_2$  have size  $m$  and  $n$  respectively, then we must transfer approximately  $21 * m + n * sizeof(short)$  bytes of data into the SPUs LS as inputs to the kernel.

The kernel result is a single value, the score, which is small enough to be returned to main memory through a special mailbox interface between the PPU and SPU. The PPU can either block or poll for a mailbox message from and SPU, making it a convenient mechanism for the SPU to signal that it has completed the processing for a given sequence pair.

Since *ssearch34* is typically used to compute the alignment for a large number of sequence pairs, we exploit the multiple SPUs of the Cell/B.E. by distributing these independent alignment computations across the SPUs. For now, we use a simple strategy in which the sequence pairs are ordered and then allocated to the SPUs in a round-robin fashion. This is similar to the load-balancing approach we describe for ClustalW in Section 5. For FASTA, we consider extending it for bigger sequences as the more important and difficult problem.

### 4.3 Results

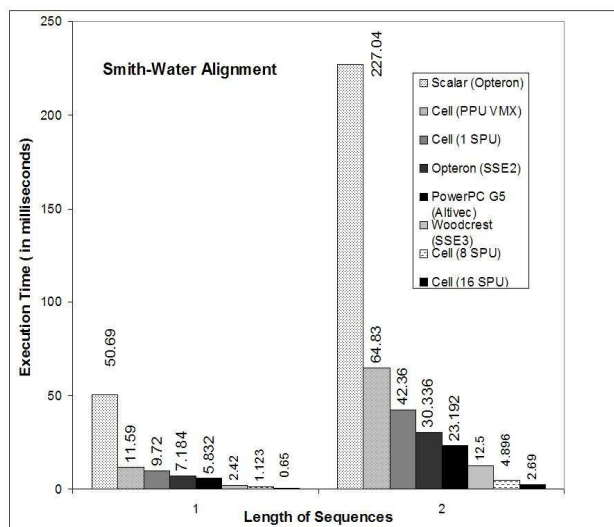


Fig. 2. Performance of Smith-Waterman Alignment for different processors

Figure 2 presents the results of the execution of the *ssearch34* on several current general-purpose processors along with our implementation for the Cell/B.E. processor. We executed a pairwise alignment of 8 pairs of sequences of 2048 characters. On the Cell/B.E., this will utilize one SPU for each pairwise alignment. Note that the Woodcrest and Opteron

implementations in FASTA use a different algorithm [7] than PPC 970 and Cell. This algorithm computes the scores vertically, since the vertical scores are mostly zero. This algorithm checks for the zero-condition on the vertical scores using the instruction `_mm_movemask_epi8`. Despite the difference in design / implementation of FASTA for these platforms, we have included all the results in Figure 2 for the benefit of the readers.

The Cell/B.E. processor, despite the absence of instructions explained above, still outperforms every superscalar processor currently in the market. This superior performance is mainly due to the presence of 8 SPU cores and the vector execution on the SPUs. We should further state that these results are still preliminary, and further optimization of the kernel performance is still underway. Further profiling of our implementation indicates that the computation dominates the total runtime (up to 99.9% considering a bandwidth of 18 GB/s for the SPUs), and hence multi-buffering is not needed for this class of computation.

The Cell implementation discussed above is not fully functional as of now: our current implementation requires both sequences to fit entirely in the SPU local store of 256 KB, which limits the sequence size to at most 2048 characters. To do genome-wide or long sequence comparisons, a pipelined approach similar to [15] among the SPUs could be implemented. In a pipelined approach, each SPU performs the Smith-Waterman alignment for a block, notifies the next SPU through a mailbox message, which then uses the boundary results of the previous SPU for its own block computation. Support of bigger sequences on the Cell is a key goal of our future research.

## 5 ClustalW on the Cell/B.E. Processor

### 5.1 Overview

ClustalW is a popular tool for multiple sequence alignment, which is needed to organize data to reflect sequence homology, identify conserved (variable) sites, perform phylogenetic analysis, and other biologically significant results. ClustalW performs a sequence alignment on all pairs of input sequences and then constructs a hierarchy for alignment. Using this hierarchy, an alignment is constructed step by step according to the guide tree. ClustalW does not give an optimal alignment, but is fast and efficient and gives reasonable alignments for similar sequences. The major time-consuming step of the ClustalW execution is the all-to-all pairwise comparisons of the input sequences, and could take 60%-80% of the execution time. The *pairalign* function performs the task of comparing all input sequences through a global alignment against each other, thus performing a total of  $\frac{n(n-1)}{2}$  alignments for  $n$  sequences. This step, called *pairalign* is explained in Algorithm 2.

<p><b>Data</b> : (1) <math>n</math> sequences <math>S_1, S_2, \dots, S_n</math> of length <math>n_1, n_2, \dots, n_n</math></p> <p><b>Result</b> : The alignment score <math>V(i, j)</math> obtained by aligning the <math>S_i</math> sequence with the <math>S_j</math> sequence, <math>i \leq n, j \leq n</math>.</p> <p><b>begin</b></p> <p style="padding-left: 2em;"><math>i = 0</math></p> <p style="padding-left: 2em;">(1) <b>while</b> <math>i++ \leq n</math> <b>do</b></p> <p style="padding-left: 4em;">(2) <math>j = i + 1</math>;</p> <p style="padding-left: 4em;">(3) <b>while</b> <math>(j++ \leq n)</math> <b>do</b></p> <p style="padding-left: 6em;">(3.1) Align sequences <math>S_i</math> and <math>S_j</math></p> <p><b>end</b></p>
--

**Algorithm 2:** Clustalw: All-to-all pairwise comparisons step

## 5.2 Cell Implementation

Our algorithm design focused on running the all-to-all comparison step on the SPUs, with the rest of the code executing on the PPU. The implementation of *pairalign* in ClustalW is broken into 4 different functions, and our profiling indicates that the *forward\_pass* function is the most time-consuming step of *pairalign*. *forward\_pass* computes the alignment of two sequences and returns the maximum score and the location within the matrix where this maximum score appears. The open source-release of ClustalW has a scalar version of *forward\_pass* which uses multiple branches used for finding the maximum at every matrix cell. Since the SPUs lack dynamic branch prediction, and the branches are not easily predictable due to the random nature of the inputs, this implementation is not well-suited for execution on the SPU. Fortunately, the IBM Deep Computing Solutions previously developed an implementation of *forward\_pass* that utilizes the AltiVec extensions of the PowerPC G5 [3]. We used this implementation as the basis for our work on ClustalW on the Cell/B.E. processor.

As with the FASTA kernel, we began porting this implementation to the SPU by converting the AltiVec intrinsics to SPU intrinsics. *forward\_pass* also uses the AltiVec *vec\_max* intrinsic as well as the saturated addition operation *vec\_adds* which must be performed with multiple instructions on the SPU. Also, the vectorized code uses the *vector status and control register* for overflow detections, while doing computation with 16-bit (*short*) data types. The overflow detects that the maximum score is greater than the range of the *short* data type, and therefore does a recomputation using integer values. Since, this mechanism is not supported inside the SPUs, we changed the code to use 32-bit (*int*) data types to begin with. This lowers the efficiency of the vector computation, since now only four values can be packed inside a vector, unlike the eight of the original code, but this was necessary for correct execution without overflow detection.

One of the bottlenecks with the ClustalW code is the alignment score lookup, in which an alignment matrix score is read for finding the cost of match/mismatch among two characters in the sequences. This lookup is a scalar operation, and hence does not perform

well on the SPU, since the SPU has only vector registers. This step can be changed to the layout of the alignment scores, in the original FASTA code, and that would give us better performance. For vector execution, four (32 bit) values are loaded into one vector, however in the code, this step is also preceded by a branch involving multiple conditions, involving both the loop variables for handling of boundary cases. Since the SPUs have only static branch prediction, such a branch, even though *mostly taken*, was difficult to predict for the SPU. We broke the inner loop of the alignment into several different loops so that the branch evaluation now depends on a single loop variable, and the boundary cases computation can be handled explicitly. This change alone helped us to get a more than 2X performance gain.

Besides the innermost kernel execution, we also focused on partitioning of the work amongst the SPUs. Assuming there are  $n$  sequences in the query sequence file, we have a total of  $\frac{n(n-1)}{2}$  computations to be performed. To distribute these computations on the SPUs, we packed all the sequences in a single array, with each sequence beginning at a multiple of sixteen bytes. This is important, as the MFC requires DMAs larger than 16 bytes to be aligned on 16-byte boundaries. This array, along with an array of the lengths of library sequences, allows the SPUs to pull in the next sequence without PPU intervention. The PPU fills in the context values, such as matrix type, gap-open and gap-extension penalties, with other inputs such as the pointers to the input array based on the command-line or default values, and the length of the sequence array.

For the SPU computation to begin, the PPU creates the threads and passes the maximum sequence size through a mailbox message. The SPUs allocate memory only once in the entire computation based on the maximum size, and then wait for the PPU to send a message for them to pull in the context data and begin the computation. Work is assigned to the SPUs using a simple round-robin strategy: each SPU is assigned a number from 0 to 7, and SPU  $k$  is responsible for comparing sequence number  $i$  against all sequences  $i + 1$  to  $n$  if  $i \bmod 8 = k$ . Algorithm 3 shows the computation for each of the SPUs.

**Data** : (1)  $n$  sequences  $S_1, S_2, \dots, S_n$  of length  $n_1, n_2, \dots, n_n$   
 (2) SPU's having `spu_id` 1, 2 .... `spu_max`

**Result** : The alignment score  $V(i, j)$  obtained by aligning the  $S_i$  sequence with the  $S_j$  sequence,  $i \leq n, j \leq n$ .

**begin**  
      $i = \text{spu\_id}$   
     (1) **while**  $i + = \text{spu\_max} \leq n$  **do**  
         (2)  $j = i + 1$ ;  
         (3) **while**  $(j + + \leq n)$  **do**  
             (3.1) Align sequences  $S_i$  and  $S_j$

**end**

**Algorithm 3:** Clustalw: All-to-all pairwise comparisons step

Such a strategy prohibits reuse of the sequence data, but since the communication costs are very low in comparison to the total computation, this strategy seems to work fairly well.

For storing of the output values, the SPUs are also passed a pointer to a array of structures, which are 16-byte aligned, in which they can store the output of the *forward\_pass* function executed for two sequences.

### 5.3 Results

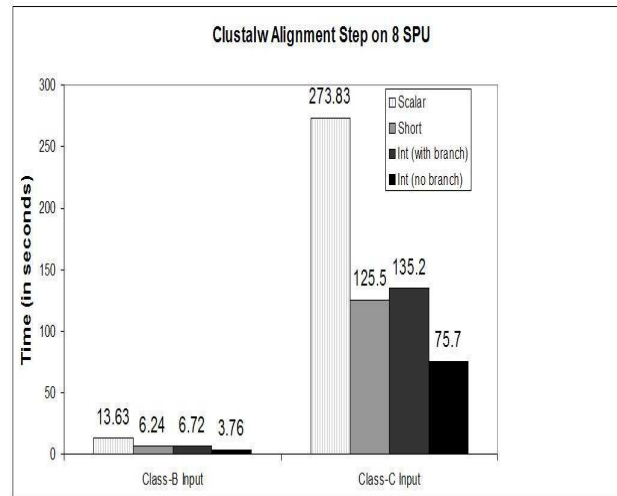


Fig. 3. Improvement of performance of ClustalW alignment function with different code changes

Figure 3 shows the time of computation of our Cell/B.E. processor implementation of ClustalW using the strategies described above for two inputs from the BioPerf suite: 1290.seq has 66 sequences of average length 1082, and 6000.seq has 318 sequences of average length 1043. The charts show the performance of the Cell/B.E. processor for scalar and vector (short and int) datatypes. It also shows how the performance notably improves with no branches.

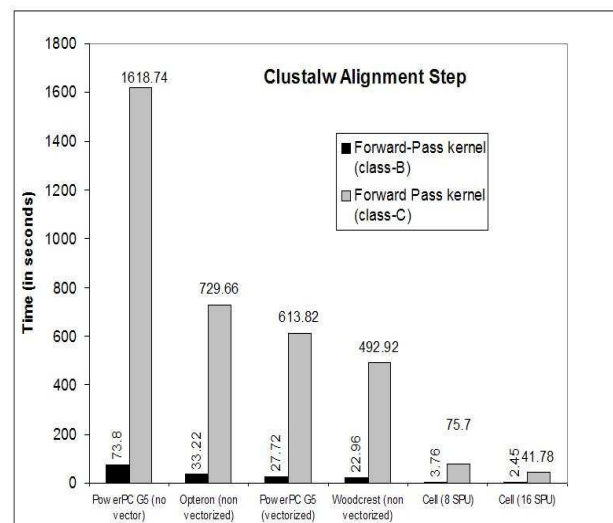


Fig. 4. Comparison of Cell Performance with other processors for only alignment function

Figure 4 shows the measured performance of the *forward\_pass* function on the Cell/B.E. platform and our reference set of general-purpose processor platforms. This graph shows that the Cell/B.E. outperforms the other processors by a significant margin, even with the simple round-robin strategy for distributing work to the SPUs. We show the best implementation strategy for the Cell/B.E. processor, namely using integer datatypes with no branches. The Opteron and the Woodcrest performance is non-vectorized, as we could not find a open-source SSE-enabled *forward\_pass* on these platforms.

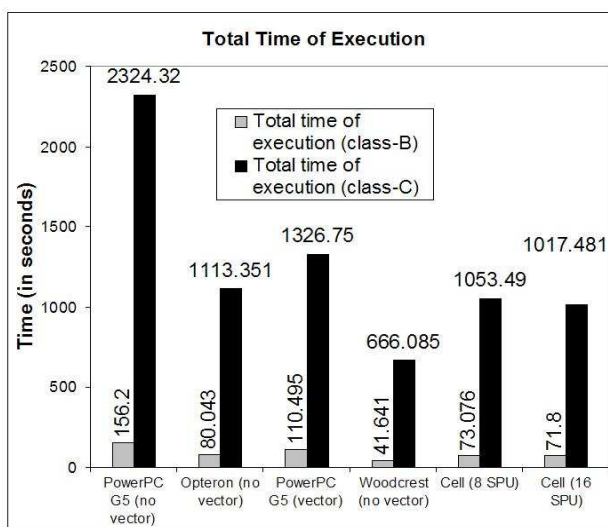


Fig. 5. Comparison of Cell Performance with other processors for total time of execution

The *ClustalW* code, executing on the PPU, uses the output for the *forward\_pass* function to generate the guide tree from the scores received from the SPU, and to compute the final alignment. Computing the final alignment takes most of the remaining execution time of the application. Due to the PPU code execution, the overall performance of *ClustalW* tends to be close in comparison to the other processors. Figure 5 shows the total time of execution of *ClustalW* for Cell/B.E. and contemporary architectures. As can be seen from the figure, the performance of our Cell/B.E. implementation is only marginally better in the overall execution time, despite being notably better in the *forward\_pass* function. This is because the remaining code executing on the PPU is much slower in comparison to the other superscalar processors. There are two ways we might address this issue: either we can attempt to migrate addition computation from the PPU to the SPUs, or we could use Cell as an accelerator, in tandem with a modern superscalar processor. The RoadRunner project [10] is already exploring such hybrid architectures, based on Opteron and Cell for accelerated application performance. Such a hybrid solution will be capable of best performance for *ClustalW*.

## 6 HMMER on the Cell/B.E. Processor

### 6.1 Overview

HMMER [6] is a freely distributable software package for protein sequence analysis that employs hidden Markov models (profile HMMs) for comparisons with sequences. Profile HMMs are statistical models of multiple sequence alignments. They capture position-specific information about how conserved is each column of the alignment, and which residues are likely. HMMER can create models based on a multiple sequence alignment as an input, which can then be stored in a database as a query into a sequence database to find (and/or align) additional homologues of the sequence family.

HMMER is written in the C language, and includes several subprograms such as *hmm-build*, *hmmcalibrate* and *hmmsearch*. *hmmpfam* searches one or more sequences to a database of profile hidden Markov models, such as the Pfam library, in order to identify known domains within a sequence, using either the Viterbi or the forward algorithm. The program reads a sequence file *seq-file* that contains the sequences that need to be stored, a score threshold and a HMM database; *hmmpfam* compares each sequence against all the HMM's in the database, and reports the sequences that score higher than the threshold. *hmmpfam* uses the kernel *P7Viterbi*, which like its name implies uses the popular Viterbi algorithm to determine the closeness between the protein family represented by the HMM and the sequence. *hmmpfam*, thus fits into the paradigm of all-to-all comparisons which could benefit from a multi-core processor like the Cell Broadband Engine.

There have been many efforts to port the *hmmpfam* to FPGAs and other accelerators [20] [12], for faster execution. However, many of these efforts utilize the FPGA or the accelerator as a screening platform; basically HMM's that score higher than a threshold value are then recomputed with traceback on the host platform. We used a different approach, in which the Cell/B.E. processor is not used as a screening platform but as a complete computing engine. However, our work on HMMER is preliminary and at an initial stage compared to FASTA and ClustalW. and handles very small sequences and HMM's (upto 100 characters and HMM states).

### 6.2 Cell Implementation

We used the same basic approach to developing our Cell/B.E. implementation as the previous applications: run the computationally expensive kernel *P7viterbi* on the SPUs, and run the remainder of the code on the PPU. The *P7Viterbi* kernel executing on the SPUs, would perform the Viterbi algorithm on its input, and return the results to the PPU for post-processing. To execute *P7Viterbi* on the SPUs, we however need to send the input parameters it uses. The *P7Viterbi* requires seven arrays as input: *dsq*, *tsc*, *bsc*, *esc*, *xsc*, *isc* and *msc*. *dsq* is the character array for the sequence, and the remaining arrays *tsc*, *bsc*, *esc*, *xsc*, *isc* and *msc* are fields of the structure representing the HMMs. These arrays need

to be DMAed to the SPUs for the *P7Viterbi* function. The memory allocation of these arrays on the PPU side is done in the similar pattern as the Altivec computation, which requires arrays to be 16-byte aligned for computation. This makes sure that the DMAs and the vector computation on the SPUs execute successfully. Besides these input arrays, the *P7Viterbi* function also uses three two-dimensional arrays *mmx*, *dmx* and *imx* which denote the match, delete and the insert parts of the HMM. These arrays are allocated on the SPUs themselves and are filled up separately, in three serial stages during the *P7Viterbi* kernel. The *mmx*, *imx* and the *dmx* denote the intermediates stages of the Viterbi algorithm in the source code.

At the end of the kernel after *P7Viterbi* is finished evaluating the score, there is a traceback step to find the most likely path through the Viterbi probabilities. This step is best suited for execution on the PPU. However, it needs the arrays *xmx*, *imx* and the *dmx* arrays, which have been filled up in the *P7Viterbi* step executing on the SPUs to be DMAed back to the PPU main memory. The memory requirements of the *P7Viterbi* kernel is more than both Smith-Waterman or ClustalW, since it requires multiple arrays for a single HMM input. The total allocation of a single HMM for the 7 arrays mentioned previously is more than 200 times the number of the HMM states (in bytes). Also, the working set size represented by the 3 arrays, is more than 12 times the product of the lengths of the HMM and the sequence (in bytes). This implies that the size of the sequences and/or HMMs we can solve inside the SPUs is significantly less than Clustalw or Smith-Waterman. Also, unlike the Clustalw or Smith-Waterman computation where only the score of the alignment was important, we need to DMA back the whole working set arrays (*xmx*, *dmx* and *imx*) for the traceback step on the PPU. Thus, there is more bandwidth requirements for HMMER, than Clustalw or Smith-Waterman. We will need to resolve all these issues, for a fully working implementation of the *hmmpfam* binary on the Cell/B.E. processor.

Apart from these data movement issues, since *hmmpfam* is already implemented using Altivec intrinsics, it required very little effort to get the code working correctly on the SPUs. One important point is that the existing Altivec code uses saturated arithmetic instructions which are not supported on the SPU. After a careful analysis of the code, we concluded that algorithm does not require saturating arithmetic and could be modified to use regular arithmetic without affecting the results. This was confirmed through email correspondence by Sean Eddy as well, and thus we replaced the *vec\_adds* with *vec\_add*. We also verified this by comparing the results of computations using saturated arithmetic with the non-saturated arithmetic for a large set of input sequences. This gave us a big boost in performance, as saturated arithmetic instructions must be performed with multiple instructions on the SPU.

### 6.3 Results

Figure 6 shows the results of the execution of *hmmpfam* on our set of evaluation platforms. These results only show a single sequence being compared against a single HMM on a single SPU. We do not have this application running on 8 SPUs as yet, but considering the embarrassingly parallel all-to-all comparison, we expect close to linear scalability

by using all the 16 SPUs in the IBM BladeCenter QS20. We do agree that the size of the sequence and the HMM is very small, for any useful output for biologists. However, the work so far was meant to explore the performance of HMMER on the Cell/B.E. processor. We also show the costs of using saturated instructions, and the DMA costs of bringing the data in, and the output data comprised of the working set arrays out. While HMMER includes AltiVec-enabled code for *P7Viterbi*, the corresponding SSE code for x86 based platforms is not included. Thus, we are only able to report results of scalar execution on the Opteron and the Woodcrest. The time is reported in seconds, from running the algorithm one million times. This is done to get an accurate measure of the time taken. We can see that with only single SPU execution, the Cell/B.E. processor is faster than the Woodcrest with the non-SSE execution. Unlike Smith-Waterman and ClustalW, we can see the DMA overhead is more than 50% of the total time of execution, and thus we need to optimize the DMA transfers for better performance. We report the time of execution without the DMA overhead; the output execution arrays will not need to be DMAed out if the Cell was only to be used as a screening platform similar to how FPGA's are employed as accelerators. With such a framework, a single Cell SPU performance is close to the PowerPC G5 AltiVec performance. These results are encouraging to attempt a fully operational *hmmpfam* binary on the Cell/B.E. processor.

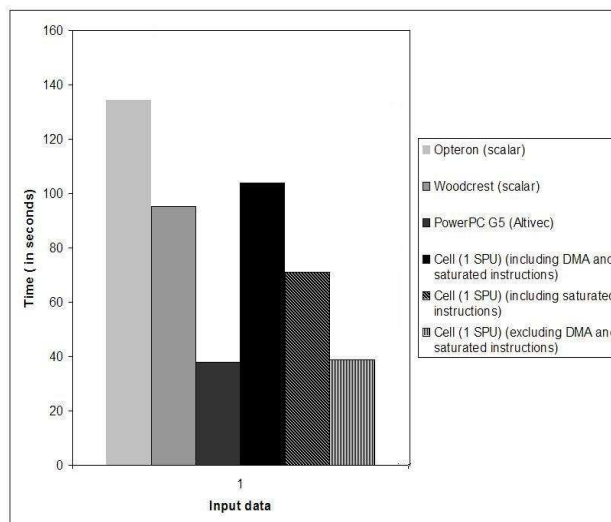


Fig. 6. Comparison of Cell Performance with other processors for total time of execution

## 7 Conclusions and Future Work

In this paper, we discussed the implementation and results of three popular bioinformatics applications, namely FASTA, ClustalW and HMMER. Our preliminary results show that the Cell Broadband Engine is an attractive platform for bioinformatics applications. Considering that the total power consumption of the Cell is less than half of a contemporary superscalar processor, we consider Cell a promising power-efficient platform for future bioinformatics computing.

Our future work will focus on making the applications discussed fully operational, and

trying to find other avenues for optimization. Both FASTA and HMMER suffer from similar issues: the size of the inputs could exceed the 256 KB local store available to each SPU. A fully operational solution for these codes would require partitioning of the input among 8 SPUs, while ensuring that the data dependency between the SPUs are fulfilled correctly. ClustalW, due to the mostly limited size of its inputs is fully functional as discussed in this paper. Besides these critical applications, we would intend to work with other applications in diverse areas such as protein docking, RNA interference, medical imaging and other avenues of computational biology to determine their applicability for the Cell/B.E. processor.

## References

- [1] E. Anson and E.W. Myers. Algorithms for whole genome shotgun sequencing. In *Proc. 3rd Ann. Int'l Conf. on Computational Molecular Biology (RECOMB99)*, Lyon, France, April 1999. ACM.
- [2] D. A. Bader, Y. Li, T. Li, and V. Sachdeva. BioPerf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *Proc. IEEE Int'l Symposium on Workload Characterization*, Austin, TX, October 2005.
- [3] Yinhe Cheng. C/c++ language extensions for cell broadband engine architecture. <http://www-03.ibm.com/servers/enable/site/technical/pdfs/1136e.pdf>, 2006.
- [4] O. Cret, S. Mathe, B. Szente, Z. Mathe, C. Vancea, F. Rusu, and A. Darabant. Fpga-based scalable implementation of the general smith-waterman algorithm. In *Proc. 18th Int'l Conf. Parallel and Distrib. Comput. Systems (PDCS 06)*, Dallas, TX, November 2006. IASTED.
- [5] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge, UK, 1998.
- [6] S. R. Eddy. Profile hidden Markov models. *Bioinformatics*, 25:755–763, 1998.
- [7] Michael Farrar. Striped smith-waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23:S156–S161, 2007.
- [8] Z. Galil and K. Park. Parallel dynamic programming. Technical Report CUCS-040-91, Computer Science Department, Columbia Univ., 1991.
- [9] R. Hughey. Parallel hardware for sequence comparison and alignment. *Comput. Applications BioSci*, 12(6):473–479, 1996.
- [10] IBM. Ibm to build world's first cell broadband engine based supercomputer. <http://www-03.ibm.com/press/us/en/pressrelease/20210.wss>, 2006.
- [11] IBM. Optimizing clustalW1.83 using altivec implementation. [http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/30B3520C93F437AB87257060006FFE5E/\\\$file/Language\\_Extensions\\_for\\_CBEA.v2.2.1.pdf](http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/30B3520C93F437AB87257060006FFE5E/\$file/Language_Extensions_for_CBEA.v2.2.1.pdf), 2006.

- [12] Arpith Jacob, Joseph Lancaster, Jeremy Buhler, and Roger Chamberlain. Preliminary results in accelerating profile hmm search on fpgas. In *Proc. 1st Workshop on High Performance Computational Biology (HiCOMB 2007)*, Long Beach, CA, April 2007.
- [13] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Systems Journal*, 49(4/5):589–605, 2005.
- [14] David Kunzman, Gengbin Zhang, Eric Bohm, and Laxmikant V. Kale. Charm++, offload api, and the cell processor. <http://charm.cs.uiuc.edu/papers/CellPMUP06.pdf>, 2006.
- [15] Weiguo Liu and Bertil Schmidt. Parallel design pattern for computational biology and scientific computing applications. In *Proc. IEEE Intl. Conf. on Cluster Computing (CLUSTER'03)*, pages 456–459, Hong Kong, December 2003.
- [16] Y. Lui, W. Huang, J. Johnson, and S. Vaidya. Gpu accelerated smith-waterman. In *Proc. GPGPU Workshop (GPGPU06)*, University of Reading, UK, May 2006. <http://www.mathematik.uni-dortmund.de/~goeddeke/iccs/index.html>.
- [17] Steve Margerm. Reconfigurable computing in real-world applications. [http://www.fpga-journal.com/articles\\_2006/20060207\\_cray.htm](http://www.fpga-journal.com/articles_2006/20060207_cray.htm), 2006.
- [18] W. S. Martins, J. B. Del Cuvillo, F. J. Useche, K. B. Theobald, and G. R. Gao. A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. In *Proc. of the Pacific Symposium on Biocomputing*, pages 311–322, Hawaii, jan 2001.
- [19] Chris Mueller. Blast on ibm's cell broadband engine. <http://www.osl.iu.edu/~chemuell/projects/presentations/cell-blast-sc06.pdf>, 2006.
- [20] Tim Oliver, Leow Yuan Yeow, and Bertil Schmidt. High performance database searching with hmmer on fpgas. In *Proc. 6th Workshop on High Performance Computational Biology (HiCOMB 2007)*, Long Beach, CA, April 2007.
- [21] Vijay Pande. Folding@home on ati gpu's: a major step forward. <http://folding.stanford.edu/FAQ-ATI.html>, 2006.
- [22] Vijay Pande. Folding@home on the ps3, December 2006. <http://folding.stanford.edu/FAQ-PS3.html>.
- [23] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences USA*, 85:2444–2448, 1988.
- [24] T. Rognes and E. Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common multiprocessors. *Bioinformatics*, 16(8):699–706, 2000.
- [25] M. Schena, D. Shalon, R.W. Davis, and P.O. Brown. Quantitative monitoring of gene expression patterns with a complementary DNA microarray. *Science*, 270(5235):467–470, 1995.
- [26] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Molecular Biology*, 147:195–197, 1981.
- [27] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTALW: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res.*, 22:4673–4680, 1994.

- [28] J.C. Venter and *et al.* The sequence of the human genome. *Science*, 291(5507):1304–1351, 2001.
- [29] J.L. Weber and E.W. Myers. Human whole-genome shotgun sequencing. *Genome Research*, 7(5):401–409, 1997.