Scientific Programming 17 (2009) 97–111
DOI 10.3233/SPR-2009-0274
IOS Press

High performance protein sequence database scanning on the Cell Broadband Engine

Adrianto Wirawan, Bertil Schmidt, Huiliang Zhang and Chee Keong Kwoh
School of Computer Engineering, Nanyang Technological University, Singapore
E-mails: {adri0004, asbschmidt, hlzhang, asckkwoh}@ntu.edu.sg

Abstract. The enormous growth of biological sequence databases has caused bioinformatics to be rapidly moving towards a data-intensive, computational science. As a result, the computational power needed by bioinformatics applications is growing rapidly as well. The recent emergence of low cost parallel multicore accelerator technologies has made it possible to reduce execution times of many bioinformatics applications. In this paper, we demonstrate how the Cell Broadband Engine can be used as a computational platform to accelerate two approaches for protein sequence database scanning: exhaustive and heuristic. We present efficient parallelization techniques for two representative algorithms: the dynamic programming based Smith–Waterman algorithm and the popular BLASTP heuristic. Their implementation on a Playstation$^{®}$3 leads to significant runtime savings compared to corresponding sequential implementations.

Keywords: Heterogeneous multi-cores, Cell BE, bioinformatics, dynamic programming, BLAST

1. Introduction

Scanning genomic sequence databases is a common and often repeated task in molecular biology. The scan operation consists of finding similarities between a particular query sequence and all sequences of a bank. There are two basic algorithmic approaches to perform this scanning.

1. *Exhaustive dynamic programming algorithms*. This approach computes an optimal pairwise alignment between the query sequence and each subject sequence in the database. The dynamic programming (*DP*) based Smith–Waterman (*SW*) algorithm [20] for computing the optimal local alignment is commonly used for this task.
2. *Heuristics*. This approach generates alignments that are valid paths through the underlying alignment model but are not guaranteed to be optimal. However, these alignments can generally be calculated more rapidly than in the exhaustive approach. Heuristics are based on filtration. Filtration assumes that good alignments usually contain short exact matches. Such matches can be quickly identified using data structures such as lookup tables. Identified matches are then used as seeds for further detailed analysis. Several filtration tools for sequence database searching have been introduced, e.g. [2,9,11]. Among them, BLAST (the $\mathcal{B}$asic $\mathcal{L}$ocal $\mathcal{A}$lignment $\mathcal{S}$earch $\mathcal{T}$ool [1,2]) is the most popular software and is used to run millions of queries each day.

The computational complexity of the exhaustive approach is quadratic with respect to the lengths of the alignment targets (query sequence and subject sequence). Filtration has been introduced as a heuristic to reduce the complexity at the cost of a generally lower quality of the results [3]. However, evaluating a single query to a large database such as GenBank with BLAST still takes several minutes on a modern workstation. These runtime requirements are likely to become even more severe due to the rapid growth of the sizes of genomic sequence databases. Hence, finding fast solutions for both algorithmic approaches is of high importance to research.

In this paper we present new approaches to parallelize the scanning of protein databases using both the exhaustive SW approach and the BLASTP heuristic on the Cell Broadband Engine (*Cell BE*) processor. The design of efficient parallel algorithms on the Cell BE requires the partitioning of computation and communication between its heterogeneous cores. Furthermore, data has to be organized to deal with a highly restricted amount of local memory and to allow SIMD vectorization, if possible.

The SW algorithm has a highly regular structure and can therefore easily be vectorized. The approach chosen in this paper uses vectors of elements parallel to the query sequence. This simplifies the dependency relationship and parallel loading of the vector scores from memory can be achieved, thus accelerating the DP matrix calculation. Our implementation on a Playstation®3 (PS3) performs from 2 to 30 times faster than any other previous attempt available on commodity hardware.

Compared to the highly regular SW algorithm, parallelization of BLASTP requires a different approach since it consists of a pipeline of computations with different memory and processing requirements. In order to exploit the characteristics of the Cell BE processor we have used a compressed deterministic finite state automaton for hit detection in order to reduce memory consumption as well as a double-buffered communication scheme. Our implementation on a PS3 achieves an average speedup of 3.2 compared to the optimized FSA-BLASTP tool and 3.6 compared to the commonly used NCBI-BLASTP software on a Pentium 4, 3 GHz.

The rest of the paper is organized as follows. Section 2 highlights features of the Cell BE architecture (CBEA). An overview of the SW algorithm, the BLASTP algorithm and previous parallelization approaches is given in Section 3. Section 4 presents our parallelization approaches on the Cell BE Performance is evaluated in Section 5. Section 6 concludes the paper.

2. Cell BE architecture

The Cell BE [8] is a single-chip heterogeneous multi-core processor. It contains two types of processors: a *PowerPC Processor Element* (PPE) and eight *Synergistic Processor Elements* (SPEs) [10]. An integrated high-bandwidth bus called the *Element Interconnect Bus* (EIB) connects the processors and their ports to external memory and I/O devices. A block diagram of the Cell BE is shown in Fig. 1. The PPE is a 64-bit PowerPC architecture. It is fully compliant with the 64-bit Power Architecture specification and can run 32-bit and 64-bit operating systems and applications. Each SPE is able to run its own individual application program. It consists of a processor designed for streaming workloads, a local memory, and a globally coherent DMA engine. The SPE implements a Cell-specific set of SIMD instructions. With all eight SPEs active, the Cell BE is capable of a peak performance of around 200 GFlops using single precision floating point arithmetic.

Although it is a multiprocessor system on a chip, the Cell BE processor is not a traditional shared-memory multiprocessor. One of the major characteristics is that an SPE can execute programs and directly load and store data only from and to its private *Local Store* (LS). Since SPEs lack shared memory, they must communicate explicitly with the PPE or other SPEs using one of three available communication mechanisms:

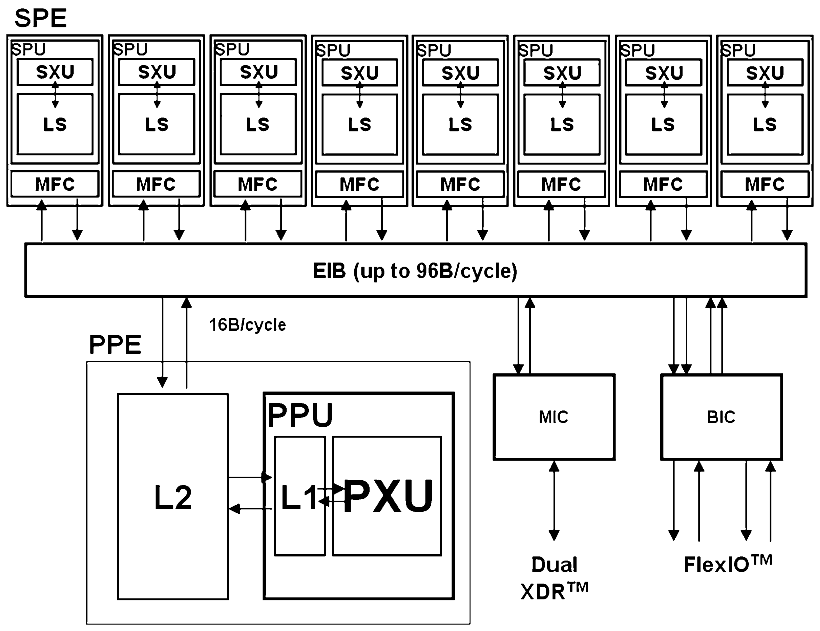

Fig. 1. Block diagram of the CBEA.

- DMA transfers,
- mailbox messages, and
- signal-notification messages.

All three communication mechanisms are controlled by the SPE's MFC (Memory Flow Controller).

The design of a parallel algorithm on the Cell BE requires an efficient partitioning of the computation between PPE and SPEs. A general approach is to perform as much as possible computations on the SPEs while the PPE is used for coordinating the control flow. Furthermore, the local memory (LS) of an SPE is very limited (only 256 kB for storing both instructions and data). Therefore, parallelized applications on the Cell BE need to be carefully structured in order to ensure that data transfers between main memory and SPEs do not become a bottleneck.

The PS3 uses the Cell BE as its processor, hence making it possible for users to create a high-powered computing environment for a fraction of the cost of a Cell Blade server. The PS3 utilizes seven of the eight SPEs, in which the eighth SPE is disabled to improve chip yields, i.e. chips do not have to be discarded if one of the SPEs is defective. Only six of the seven SPEs are accessible to developers as one is reserved by the operating system. Generally available PS3's can be used for scientific high performance computing through installation of Linux. Programs can be developed using the freely available C-based Cell BE SDK. Thus, the PS3 offers a good alternative to other accelerator technologies such as FPGAs or GPUs.

3. Related work

3.1. Smith–Waterman algorithm

Surprising relationships have been discovered between protein sequences that have little overall similarity but in which similar subsequences can be found. In that sense, the identification of similar subsequences is probably the most useful and practical method for comparing two sequences. The Smith–Waterman algorithm finds the most similar subsequences of two sequences (the local alignment) by dynamic programming (*DP*). The algorithm compares two sequences by computing a distance that represents the minimal cost of transforming one segment into another. Two elementary operations are used: substitution and insertion/deletion (also called a gap operation). Through series of such elementary operations, any segments can be transformed into any other segment. The smallest number of operations required to change one segment into another can be taken into as the measure of the distance between the segments.

Consider two strings S_1 and S_2 of length l_1 and l_2. To identify common subsequences, the Smith–Waterman algorithm computes the similarity $H(i,j)$ of two sequences ending at position i and j of the two sequences S_1 and S_2. The computation of $H(i,j)$, for $1 \leqslant i \leqslant l_1, 1 \leqslant j \leqslant l_2$, is given by the following recurrences:

$$\begin{aligned} H(i,j) &= \max\{0, E(i,j), F(i,j), \\ &\qquad H(i-1,j-1) \\ &\qquad + \mathit{sbt}(S_1[i], S_2[j])\}, \\ E(i,j) &= \max\{H(i,j-1) \\ &\qquad - \alpha, E(i,j-1) - \beta\}, \\ F(i,j) &= \max\{H(i-1,j) \\ &\qquad - \alpha, F(i-1,j) - \beta\}, \end{aligned}$$

where *sbt* is a character substitution cost table. Initialization of these values are given by $H(i,0) = E(i,0) = H(0,j) = F(0,j) = 0$ for $0 \leqslant i \leqslant l_1$, $0 \leqslant j \leqslant l_2$. Multiple gap costs are taken into account as follows: α is the cost of the first gap; β is the cost of the following gaps. This type of gap cost is known as *affine gap penalty*. Some applications also use a *linear gap penalty*, i.e. $\alpha = \beta$. For linear gap penalties the above recurrence relations can be simplified to:

$$\begin{aligned} H(i,j) = \max\{&0, H(i,j-1) - \alpha, H(i-1,j) \\ &- \alpha, H(i-1,j-1) \\ &+ \mathit{sbt}(S_1[i], S_2[j])\}. \end{aligned}$$

Each position of the matrix H is a similarity value. The two segments of S_1 and S_2 producing this value can be determined by a traceback procedure. Figure 2 illustrates an example.

3.2. Hardware-accelerated Smith–Waterman

The recent emergence of accelerator technologies such as FPGAs, GPUs, and specialized processors have made it possible to achieve an excellent improvement in execution time for many bioinformatics applications, compared to current general-purpose platforms. However, special-purpose hardware implementations such as FPGAs [12,16] tend to be very expen-

	∅	A	T	C	**T**	**C**	**G**	**T**	**A**	**T**	**G**	**A**	T	G
∅	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	2	1	0	0	2	1	0	2
T	0	0	2	1	2	1	1	4	3	2	1	1	3	2
C	0	0	1	4	3	4	3	3	3	2	1	0	2	2
T	0	0	2	3	6	5	4	5	4	5	4	3	2	1
A	0	2	2	2	5	5	4	4	7	6	5	6	5	4
T	0	1	4	3	4	4	4	6	5	9	8	7	8	7
C	0	0	3	6	5	6	5	5	5	8	8	7	7	7
A	0	2	2	5	5	5	5	4	7	7	7	10	9	8
C	0	1	1	4	4	7	6	5	6	6	6	9	9	8

Fig. 2. Example of the Smith–Waterman algorithm to compute the local alignment between two DNA sequences ATCTCGTATGATG and GTCTATCAC. The matrix $H(i, j)$ is shown for the linear gap cost $\alpha = 1$, and a substitution cost of $+2$ if the characters are identical and -1 otherwise. From the highest score ($+10$ in the example), a traceback procedure delivers the corresponding alignment (shaded cells), the two subsequences TCGTATGA and TCTATCA.

sive and hard-to-program. Hence, they are not suitable for many users. Recent usage of easily accessible accelerator technologies to improve the search time of the SW algorithm include Intel SSE2 [6] and GPUs [13,14].

Farrar [6] exploits the SSE2 SIMD multimedia extension of general-purpose CPUs. His implementation utilizes vector registers, which are parallel to the query sequence and are accessed in a striped pattern. Similar to the implementation by Rognes and Seeberg [18], a query profile is calculated only once for each database search. However, Farrar's implementation allows moving the conditional calculation of the F-matrix outside the inner loop. Therefore, this implementation achieves a speed up of factors 2–8 over the previous SIMD implementations by Wozniak [21] and Rognes and Seeberg [18].

Liu et al. [13] first reported the implementation of the Smith–Waterman algorithm on graphics hardware. The SW algorithm is implemented using the streaming architecture of GPUs by reformulating it in terms of computer graphics primitives. The implementation relies on OpenGL, in which a conversion of the problem to the graphical domain is needed, as well as a reverse procedure to convert back the results. Although, it achieves a high efficiency, programming in OpenGL requires specialized skills. Therefore, Manavski and Valle [14] re-implemented the SW algorithm on a GPU with the recently released C-based CUDA programming environment.

In this paper, we demonstrate how the PS3, a commodity hardware powered by the Cell BE, can be used as a low cost computational platform to accelerate the Smith–Waterman algorithm. Our implementation is able to outperform both the striped method as well as the CUDA-based GPU implementation. A previous implementation by Sachdeva et al. [19] ports the diagonal-based Smith–Waterman vectorization by Wozniak [21] to the Cell BE. However, it uses only eight sequences of exact length of 2048 amino acids, which is a very small dataset. The papers by Rognes and Seeberg [18] and Farrar [6] have also shown that column-based vectorization is faster than diagonal-based vectorization.

3.3. BLASTP algorithm

The basic idea for fast sequence database search is *filtration*. Filtration assumes that good alignments usually contain short exact matches. Such matches can be quickly computed by using data structures such as lookup tables. Identified matches are then used as seeds for further detailed analysis. The analysis pipeline of the BLASTP algorithm is shown in Fig. 3. It consists of four stages. Each stage progressively reduces the search space in the database for significant alignment. We briefly describe each step in the following. More details can be found in [1,2].

Stage 1: This stage identifies *hits*. Each hit is defined as an offset pair (i, j) for which $\sum_{k=0}^{w-1} sbt(Q[i + k], D[j + k]) \geqslant T$, where *sbt* is a amino acid substitution matrix (e.g. BLOSUM65), w is the user-defined word length, T is a user-defined threshold, Q is the query sequence and D is the database. BLASTP implements this stage by preprocessing Q as follows. For each position i of Q the neighborhood $N(Q[i, \ldots, i + w - 1], T)$ is computed consisting of all w-mers p for which $\sum_{k=0}^{w-1} sbt(Q[i + k], p[k]) \geqslant T$. The complete neighborhood of a query is typically stored in an efficient data structure such as a lookup table or

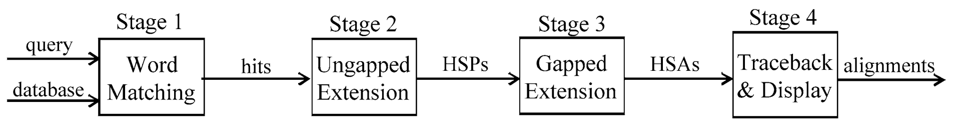

Fig. 3. The BLASTP processing pipeline.

a finite-state automaton. The default parameter values are $w = 3$ and $T = 11$.

Stage 2: Stage 2 outputs *HSPs* (high-scoring segment pairs) between Q and D. HSPs are identified by performing an ungapped extensions on a diagonal d which contains a non-overlapping hit pair $(i_1, j_1), (i_2, j_2)$ within a window A; i.e. $d = i_1 - j_1 = i_2 - j_2$ and $w \leqslant i_2 - i_1 \leqslant A$. If the resulting ungapped alignment scores above a certain threshold it is passed to Stage 3.

Stage 3: This stage outputs *HSAs* (high scoring alignments) between Q and D. HSAs are identified by performing a seeded banded gapped dynamic programming based alignment algorithm using the previously identified HSPs as seeds. Alignments that score above a certain threshold are then passed to the final stage.

Stage 4: The final alignments of the highest scoring sequences are calculated and displayed to the user. This requires the computation of the traceback path using the Smith–Waterman algorithm.

An execution profiling of the BLASTP algorithm for scanning the Genbank non-redundant protein database shows the following breakdown of execution time:

$$\text{Stage 1: } 37\%, \qquad \text{Stage 2: } 31\%,$$
$$\text{Stage 3: } 30\%, \qquad \text{Stage 4: } 2\%.$$

Hence, in order to efficiently map BLASTP on the Cell BE all stages except Stage 4 need to be parallelized. Previous work on parallelizing BLASTP has focused on distributed memory architectures such as clusters [15] and reconfigurable hardware [7]. This paper is to our knowledge the first ever reported parallelization of BLASTP on the Cell BE.

4. Parallelization of database scanning on the Cell BE

4.1. Parallelization approach

In order to achieve an efficient parallelization of protein sequence database scanning on the Cell BE processor, we need to address the following challenges:

- *Limited local storage of the SPE.* A major limitation when designing SPE kernels is that their local memory is only 256 kB for both instructions and data. Using default parameter for w and T the size of the lookup table used for Stage 1 by NCBI BLASTP is already around 400 kB for a query sequence of average length [5]. Therefore, we need to use an alternative data structure which requires significantly less memory.
- *Data transfer and coordination between PPE and SPEs.* The runtime of the SW algorithm merely depends on the length of the query and subject sequence. Therefore, a static load balancing strategy can be used for this approach; i.e. the work load is known at the start and distributed equally across the SPEs. However, the different stages of the BLASTP algorithm constitute a processing pipeline where the throughput of each stage in the pipeline depends on the filtration efficiency of the previous stage. Therefore, an efficient and flexible mechanism to transfer sequences from the database to the SPEs needs to be implemented for this approach.
- *SIMD vectorization.* There are two basic approaches to vectorize SW. All elements in the same minor diagonal of the DP matrix can be calculated independent of each other. Therefore, a possible vectorization approach is to compute the DP matrix in minor diagonal order [21]. Another approach vectorizes the DP matrix computation in a column-wise order [6,18]. By using vectors of elements parallel to the query sequence, the much-simplified dependency relationship and parallel loading of the vector scores from memory can be achieved, thus accelerating the DP matrix calculation. We have decided to use the column-based approach for vectorization on the Cell BE processor since: (1) the column-based approach outperforms the minor-diagonal approach on Intel SSE2 architectures, and (2) since we only need to store one column of the DP matrix instead of two diagonals for the minor diagonal method, the column-based approach requires less SPE memory.

4.2. Smith–Waterman

Our parallelization takes advantage of the 128-bit wide SIMD vector registers of each SPEs. The vectorization strategy is based on a column-based approach [6,18]. It also employs a static load balancing strategy, which means that the work load is known at the start and distributed equally across the SPEs. Figure 4 illustrates the mapping of different stages of SW-based protein sequence database scanning application onto the Cell BE processor. The PPE starts by reading the query sequence and the database from the respective files. It then pre-processes the query sequence such that it is suitable for vector operations. The pre-processed query sequence, together with some context data, is sent to each respective SPEs, which in turn will generate its own query profile. This process is done using DMA transfers, namely *mfc_get()* and *mfc_put()*.

Given a database D consisting of $|D|$ sequences and k SPEs. Each SPE aligns the query sequence to $|D|/k$ database sequences. The pseudocode of the SPE code is shown in Fig. 5. Scores obtained from those alignments are sorted locally in the SPEs and the b highest scores are sent to the PPE, where they are sorted once again to obtain the b overall highest scores.

The size of the SPE program in our implementation is 150 kB. The longest sequence in the Swiss-Prot database is 35,213 amino acids (accession number A2ASS6). In order to accommodate such a long protein sequence, we allocate dynamic memory of up to 64 kB to store subject sequences. Our implementation also allocates 16 kB for the read buffer. Due to these limitations, the maximum query sequence length allowed for our implementation is limited to 852.

In order to calculate $H(i, j)$ in the SW DP matrix, the value $sbt(S_1[i], S_2[j])$ needs to be added to $H(i-1, j-1)$. To avoid performing this table lookup for each element in the DP matrix, Rognes and Seeberg [18] and Farrar [6] suggested calculating a *query profile* parallel to the query sequence beforehand. Assuming that $S_1, S_2 \in \Sigma^*$ and S_1 is the query sequence, the query profile is defined as a set $P = \{P_x \mid x \in \Sigma\}$

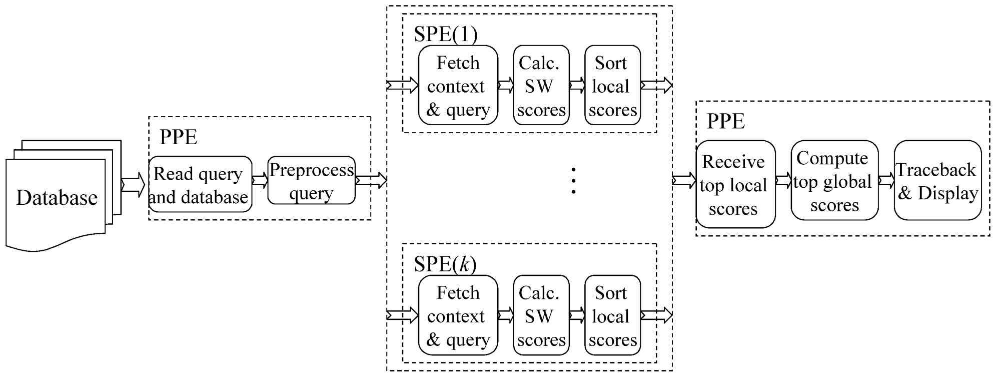

Fig. 4. Mapping of the different stages of database scanning with SW onto the Cell BE.

```
Input: SPE id $i$, total number of SPEs $num$, query sequence $Q$, chunk of databases
sequence $D_{i+j\cdot num}$, $(0 \leqslant j \leqslant |D|/num)$
Output: Database sequences with $b$ highest SW-scores
1.   Initialize
2.   Fetch context data from PPE using DMA
3.   Fetch query sequence $Q$
4.   Fetch first database sequence $D_i$
5.   For $j = 0$ to $|D|/num$ do
5a.      Compute SW-score between $Q$ and $D_{i+j\cdot num}$
5b.      Insert score into list of sorted scores
5c.      Fetch next database sequence $D_{i+(j+1)\cdot num}$
6.   Send $b$ highest scores with associated sequence id's to PPE using DMA
```

Fig. 5. Pseudocode of the program running on SPE(i).

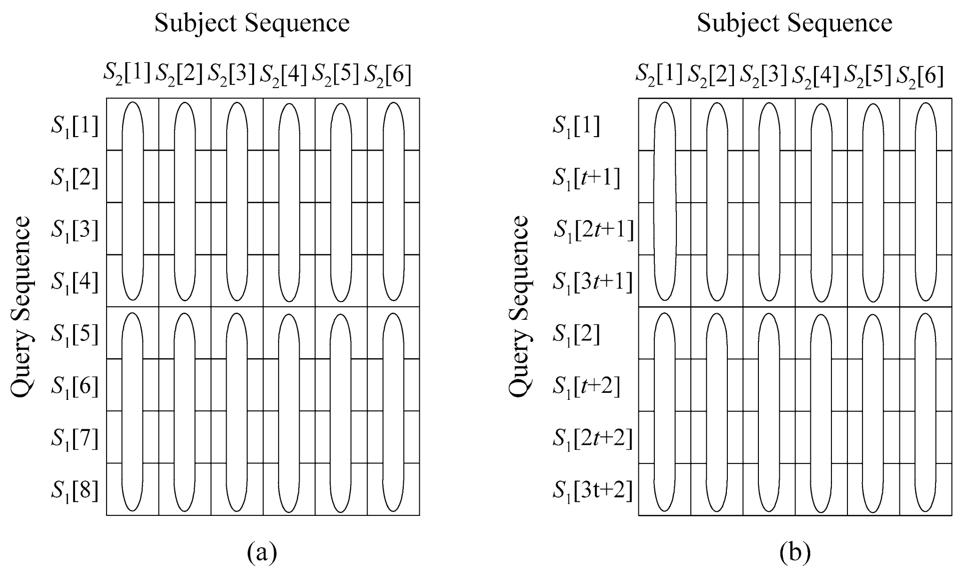

Fig. 6. The query profile layout for (a) sequential method, (b) striped method.

consisting of $|\Sigma|$ numerical strings of length l_1 each, where $l_1 = |S_1|$. Each string $P_x \in P$ consists of all substitution table values that are needed to compute a complete column j of the DP matrix for which $S_2[j] = x$. Pre-computing the query profile greatly reduces the amount of substitution table lookup in the SW DP matrix computation, since $|\Sigma|$ is usually much smaller than $|S_2|$.

The query profile can be calculated in a straightforward *sequential layout* [18] (see Fig. 6(a)) or in a more complex *striped layout* [6] (see Fig. 6(b)). The values in the query profile are defined as:

$$P_x[i] = sbt(S_1[i], x)$$
$$\text{for all } 1 \leqslant i \leqslant l_1 \quad \text{[sequential layout]},$$
$$P_x[i] = sbt\left(S_1\left[(((i-1)\%p)t) + \left\lfloor \frac{i-1}{p} \right\rfloor + 1\right], x\right)$$
$$\text{for all } 1 \leqslant i \leqslant l_1 \quad \text{[striped layout]},$$

where p is the number of segments and t is the segment length.

In the striped layout, p corresponds to the number of elements that can be processed in a SIMD vector register (e.g. for 128-bit wide SIMD registers, $p = 8$ when using 16-bit precision). The length of each segment, t, is equal to $\lceil (l_1 + p - 1)/p \rceil$.

Both approaches allow efficient vectorization on SSE2-compatible processors using the corresponding SIMD instruction set. Using the pre-calculated query profile, the computation of the DP matrix can be performed in column-wise order. Due to the simplified dependency relationship and parallel loading of the vector scores from memory, fast DP matrix calculations can be achieved. The advantage of the striped layout compared to the sequential layout is that data dependencies between vector registers are moved outside the inner loop. For instance, when calculating vectors for the DP matrices H or F with the sequential layout, the last element in the previous vector has to be moved to the first element in the current vector. When using the striped query layout, this needs to be done just once in the outer loop when processing the next subject sequence character. We have, therefore, chosen to map the striped layout onto the Cell BE processor.

In addition, the inner loop of the algorithm requires saturation arithmetic, namely saturated additions and saturated subtractions. However, different from the SSE2 instruction set, the Cell BE processor lacks the saturation arithmetic support. In order to tackle this problem, we have introduced two new functions, namely *spu_adds()* and *spu_subs()* to handle saturated additions and saturated subtractions, respectively.

4.3. BLASTP

Figure 7 shows our mapping of the different stages of the BLASTP algorithm onto the Cell BE. Stage 4

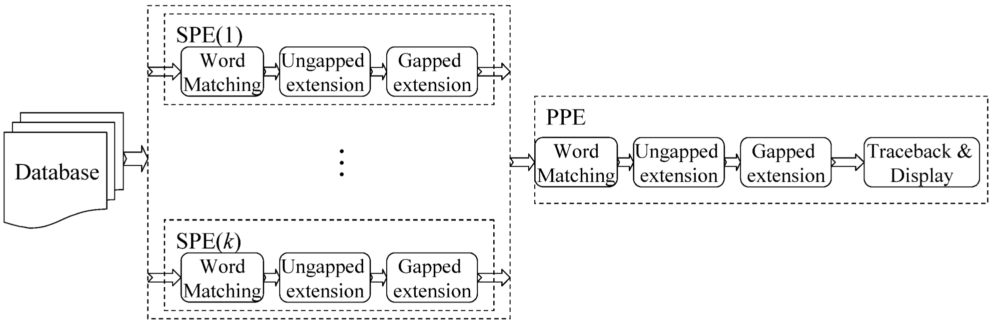

Fig. 7. Mapping of the different stages of the BLASTP algorithm onto the Cell BE.

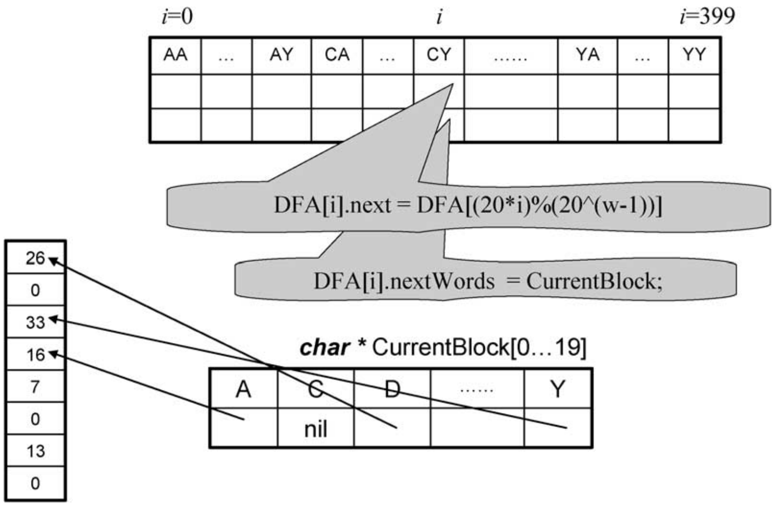
i=0
i
i=399
AA
AY
CA
CY
YA
YY
DFA[i].next = DFA[(20*i)%(20^(w-1))]
DFA[i].nextWords = CurrentBlock;
char * CurrentBlock[0…19]
A
C
D
Y
nil
26
0
33
16
7
0
13
0

Fig. 8. Illustration of the compressed FSA data structure for $w = 3$.

includes a ranking procedure on all database sequences that have passed Stages 1–3: the top 500 or less matching sequences whose scores exceed a certain threshold are displayed in descending order. Thus, this stage is performed by the PPE. SPE kernels filter the database as follows. Information about all subject sequences from the database that have passed Stages 1–3 on an SPE are sent to the main storage. Subsequently, the PPE completes Stages 1–4 for these subject sequences. The reason why not only Stage 4 is performed on the PPE is that this stage requires additional information from the previous stages. Storing this on the SPEs would be too memory-intensive in terms of both data and instructions. Therefore, we have decided to repeat Stages 1–3 on the PPE. Please note that this redundant computation is merely performed for very few subject sequences the additional runtime is relatively small (see Section 5 for details).

As mentioned above, the size of the codeword lookup data structure used by NCBI BLAST is too large for the local store of the SPEs. Therefore, we are using a more memory-efficient data structure for Stage 1. The utilized data structure is a compressed *deterministic finite-state automaton* (DFA), which is similar to the approach used by FSA-BLAST [4,5]. The compressed DFA for $w = 3$ is illustrated in Fig. 8.

Each possible prefix of lengths $w - 1$ is represented by a state; i.e. for $w = 3$ there are 400 states representing the prefixes AA to YY, which are stored in the array DFA[i] in Fig. 8. Each state has two transitions: one to the next state (DFA[i].next) and one to a list of 20 words (DFA[i].nextWords). Each entry in this list (currentBlock[0 . . . 19]) contains a pointer to an array of query positions. These query positions represent the neighborhood $N(w, T)$ of the associated w-mer. This data structure allows the compression of frequently used query positions that are in neighborhoods of similar w-mers. For example in Fig. 8, $N(\text{'CYY'}, T) = \{33, 16, 7\}$ and $N = \{\text{'CYA'}, T\} = \{16, 7\}$. By storing these positions in subsequent order terminated by

"0" it is possible to re-use memory for both neighborhoods. Our experiments have shown that the size the compressed DFA is only 43.8 kB on average. Hence, it is possible to store the complete data structure on each SPE for most queries.

The DFA is transferred into each SPE. The PPE then reads sequences from the database and transfers them to the SPEs by Direct Memory Access (DMA). In order to hide latencies and achieve good load balancing, we have implemented four buffers per SPE on main memory and two buffers on each SPE's local store (see Fig. 9). Our double buffering scheme allows SPEs to receive a new subject sequence through DMA while processing another previously received sequence. The PPE continuously prepares sequence data for free buffers. Once a buffer is filled, the PPE sends a mailbox notification to the corresponding SPE. The number of buffers in the PPE for each SPE is therefore restricted by the size of the SPE's Read Inbound Mailbox (which is four). Furthermore, the PPE dynamically assigns protein sequences to buffers depending on their lengths and the available memory. The maximum number of sequences inside a buffer is 32.

All sequences inside a buffer are filtered by Stages 1–3 on one of the SPEs. If a sequence passes all these stages, the corresponding bit in the matching signal (32 bits) is set. After all sequences are processed, this matching signal is sent back to the PPE via a mailbox. The PPE then identifies all sequences that have passed Stages 1–3 on SPE and perform Stages 1–4 on them.

Pseudocodes of the programs running on the PPE and each SPE are shown in Figs 10 and 11. The size of the SPE program is 100 kB. Thus, we have at most 156 kB for storing the DFA data structure, the two buffers as well as other parameters and intermediate results. Hence, we have assigned 10 kB to each buffer and up to 80 kB to the DFA. 80 kB is suffi-

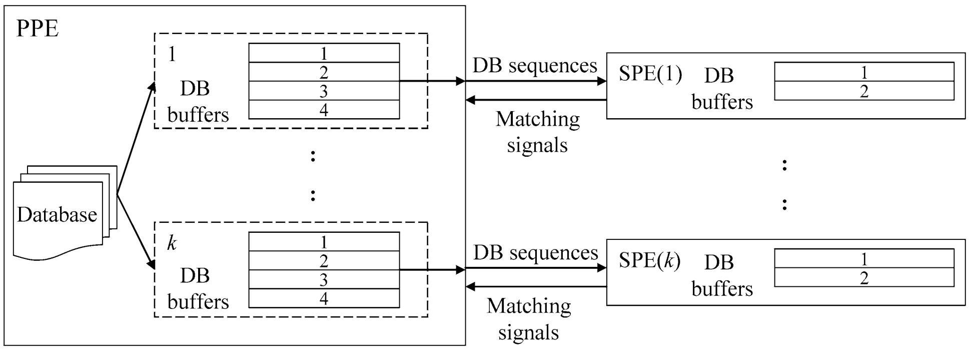

Fig. 9. Buffering scheme.

```
1. Initialization
2. Create DFA
3. Start SPEs and send parameters and DFA lookup table to SPEs
4. Check whether there is mail from SPEs
   If there is a mail
      Collect information of sequences that passed Stages 1–3 and keep in a queue
      Mark the corresponding buffer as free
5. Check whether there is a free buffer
   If a free buffer is found
      Prepare data into it and mark it as occupied
   Else
      Do BLASTP searching Stages 1 and 2 for sequences in the queue
6. Repeat 4–5 until there is no sequence in database
7. Send commands to SPEs to complete last buffered sequences
8. Wait until all buffers are marked as free
9. Do BLASTP Stages 3 and 4
```

Fig. 10. Pseudocode of the program running on the PPE.

```
1.   Initialization
2.   Receiving parameters and DFA from PPE
3.   Receiving mail with command from PPE
4.   If command is new-data-available
       DMA the new data
       If this is the 1st data
         Goto 3
       Else
4.5      Wait for last data to be completely DMA transferred
         Do Stages 1–3 for sequences in the last data
4.7      Return matching signal to PPE through SPU Write Outbound Mailbox
         Goto 3
5.   If command is finish-last-sequence
       Do Stages 1–3 for sequences in the last data
       Return matching signal to PPE through SPE Write Outbound Mailbox
       Exit
```

Fig. 11. Pseudocode of the program running on the SPE.

cient for DFAs for query sequences of up to 2000 base-pairs (bps). In our experiment, the average DFA size is 43.8 kB. If the length of a subject sequence is over 10 kBps, it will be put directly into the sequence queue of the PPE without sending it to an SPE. Furthermore, some database sequences exceed a certain memory threshold during they are processed on the SPE. Such sequences will be marked and passed to the PPE for further processing. Although, this creates additional work, the number of such sequences is usually negligible. It is also another reason why the PPE performs all stages of the BLASTP algorithm instead of only Stage 4. Furthermore, note that we do not return results of matching sequences from SPEs because we do not want to increase SPE code size by increasing program complexity to return the search results.

5. Performance evaluation

5.1. Smith–Waterman

We analyze the performance of our parallel algorithm for various query sequence lengths using sequences from Swiss-Prot database release 55.2 comprising 130,497,792 amino acids in 362,782 sequence entries. Searches for 18 query sequences with various lengths between 63–852 amino acids using the parameters $\alpha = 10$, $\beta = 2$, and BLOSUM45 have been performed. The experiments have been carried out on a standalone PS®3 with the Linux Fedora 7 operating system and the Cell Software Development Kit (SDK) 3.0.

The performance statistics measured are execution time and MCUPS (Mega Cell Updates Per Second). Given a query sequence of size Q and a database of size D, MCUPS rating (million cell updates per second) is calculated by:

$$\frac{|Q| \times |D| \times 10^6}{t},$$

where $|Q|$ = size of query sequence in amino acids, $|D|$ = size of database sequences in amino acids, t = run time (including input from file, initialization and result output).

Table 1 shows the performance of our parallel algorithm on the above mentioned datasets. By using 6 SPEs available in the PS3, our parallel algorithm reaches a peak performance of 3,663.40 MCUPS for a query sequence of length 852.

We express the execution time T of our implementation by: $T = T_{\text{overhead}} + T_{\text{comp}}$, where T_{comp} = computation time, i.e. time needed to compute all alignment score and T_{overhead} = overhead time, i.e. initialization, communication and result output.

In order to establish a theoretical saturation limit of the performance of our implementation, we have estimated the overhead time of our Cell BE implementation. This was done by measuring the total runtime for scanning Swiss-Prot 55.2 with a query sequence of length one on a PS3. This yielded a time of T = 13.13 s. Since the query sequence is very small, T_{comp}

Table 1

Performance of CBESW for different query sequences for scanning Swiss-Prot 55.2

Accession number	Query sequence length	CBESW (s)	CBESW (MCUPS)
O29181	63	14.55	565.04
P03630	127	14.74	1124.37
P02232	143	14.97	1246.57
P05013	189	15.06	1637.72
P14942	222	15.18	1908.47
P00762	246	15.35	2091.37
P53765	255	15.49	2148.29
Q8ZGB4	361	17.57	2681.26
P10318	362	17.64	2678.02
P07327	374	17.90	2726.60
P01008	464	20.89	2898.56
P10635	497	21.34	3039.24
P58229	511	22.13	3013.30
P25705	553	22.84	3159.60
P42357	657	26.14	3279.92
P21177	729	27.85	3415.90
Q38941	850	30.29	3662.04
O60341	852	30.35	3663.40

is negligible and the overhead time, T_{overhead} is similar to T, which is 13.13 s. From Table 1 we can see that the percentage of T_{overhead} compared to T continuously decreases with the query sequence length: it ranges from 90.24% (for shortest query sequence) to 43.26% (for the longest query sequence). We have therefore derived a theoretical saturation limit of the performance of our implementation by subtracting the overhead time from the runtime for the longest query sequence (852) as $(130{,}497{,}792 \times 852)/(30.35\text{ s} - 13.13\text{ s}) = 6.457$ MCUPS.

Furthermore, we have compared the performance of our CBESW implementation with other publicly available implementations of SW-based protein database scanning, namely SSEARCH [17], Striped Smith–Waterman (Striped SW) [6] and CUDA [14], as shown in Fig. 12. SSEARCH is a SW implementation which is part of the FASTA package. The SSEARCH and the Striped SW performance is benchmarked on an Intel Core Duo 2.4 GHz CPU with 1 GB RAM. Both codes use only a single thread and therefore use only one of the two available cores. The substitution matrix used was BLOSUM45. Nine query sequences with lengths of 63, 127, 255, 361, 511, 657, 729, 850 and 852 amino acids are used. The comparison with the CUDA implementation is benchmarked on a GeForce 8800GTX 512 installed in a PC with a Dual-Core AMD Opteron 2210 1.8 GHz CPU, 2 GB RAM running Fedora 6. The substitution matrix used was BLOSUM50. Seventeen query sequences with lengths of 63, 127, 143, 189, 222, 246, 255, 361, 362, 374, 464, 497, 553, 657, 729, 850 and 852 amino acids were used.

As shown in Fig. 12, for a query sequence of length 852 (accession number O60341), SSEARCH, Striped SW and CUDA achieve a performance of 121.91, 2220.09 and 1212 MCUPS, respectively. Hence, the peak performance of our implementation is about 2–30 times faster compared to the other implementations.

5.2. BLASTP

We have implemented the described Cell BE BLASTP program using CELL BE SDK 3.0 and evaluated it on a PlayStation®3 (PS3), which contains a Cell BE as its main processor. In order to evaluate the performance on a PS3, we have installed LINUX version 2.6.23-rc3 (gcc version 4.1.1 20061011 (Red Hat 4.1.1-30)). Please note that on the PS3 only six out of eight SPEs can be used for user programs. Therefore, our experiments can only use up to six SPEs.

We have compared the performance of our Cell BE BLASTP program to FSA-BLASTP (available from: www.fsa-blast.org) and NCBI-BLASTP (www.ncbi.nlm.nih.gov/BLAST/developer.shtml). FSA-BLAST uses an optimized sequential algorithm and is around 15% faster than NCBI-BLASTP with no loss in accuracy [4,5]. FSA-BLASTP and NCBI-BLASTP are tested on a HP workstation xw4200 with Dual-core Pentium®4 (P4) CPU 3 GHz, 2 GB of RAM. Both codes use only a single thread and, therefore, use only one of the two available cores. The Two-hit model [2] is used for all BLASTP programs. Default values of $W = 3$ and $T = 11$ are adopted. The produced matching results by FSA-BLASTP and Cell BE BLASTP are exactly the same.

The protein sequence database we used in our experiments is the GenBank Non-Redundant Protein Database (downloaded from: ftp://ftp.ncbi.nih.gov/blast/db/FASTA/nr.gz), which contains 6,375,605 protein sequences. We have chosen 100 random sequences from the database as queries. The lengths of the query sequences are distributed uniformly between 1 and 2000 bps.

A performance comparison of the presented parallel Cell BE BLASTP program to the sequential FSA-BLASTP and NCBI-BLASTP programs are shown in Fig. 13. It can be seen that Cell BE BLASTP is faster

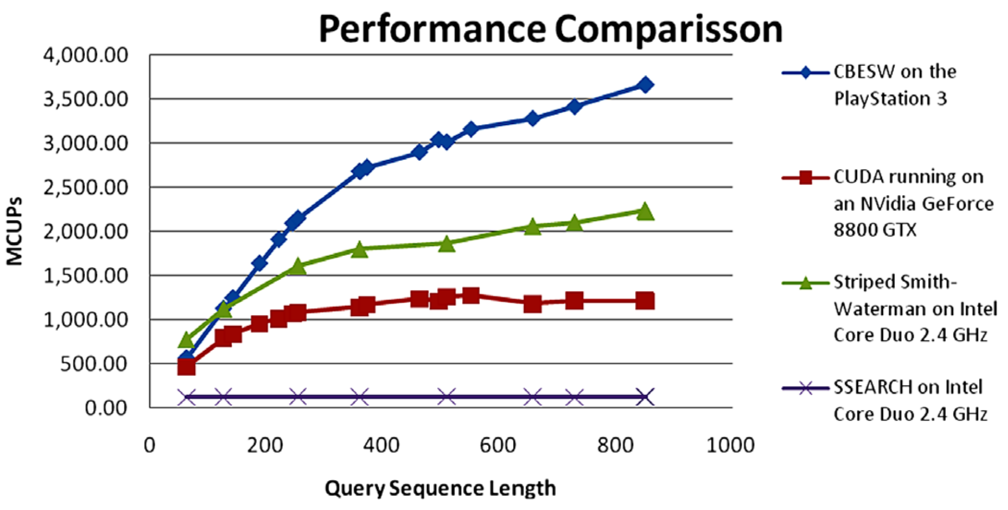

Fig. 12. Performance comparison of CBESW to CUDA-SW, Striped-SW and SSEARCH.

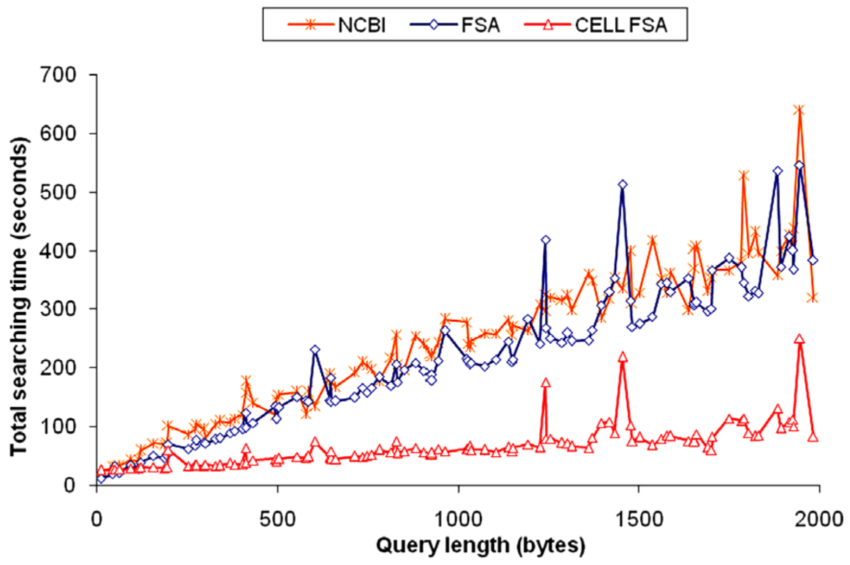

Fig. 13. Runtime comparison between FSA-BLASTP on a P4 3 GHz and Cell BE BLASTP on a PS3 for varying query sequence lengths.

than FSA-BLAST in most cases. The average searching times are 217.5 s for FSA-BLASTP, 244.75 s for NCBI-BLASTP, and 67.97 s for Cell BE BLASTP. This corresponds to an average speedup of 3.2 and 3.6, respectively.

More detailed statistics are shown in Tables 2 and 3. Table 2 shows the runtime of FSA-BLASTP for the different stages. Note that the runtime for Stages 1 and 2 are combined since they are implemented in an interleaved fashion. Separating them would require additional memory for storing hits identified in Stage 1. Table 3 shows the runtime of the corresponding stages of Cell BE BLASTP on the PPE, whereby the runtime

Table 2

Detailed runtime analysis (in seconds) of FSA-BLASTP on a P4 3 GHz

Query length range	FSA-BLASTP			
	Stages 1 and 2	Stage 3	Stage 4	Total
1–300	40.1	5.66	0.30	46.5
301–500	74.0	23.09	0.32	97.8
501–800	110.3	46.57	0.50	157.8
801–1100	151.0	50.98	0.92	203.4
1101–1400	183.0	76.32	1.80	261.6
1401–1700	216.9	109.01	3.22	329.6
1701–2000	241.8	141.53	2.02	385.9

Table 3

Detailed runtime analysis (in seconds) of Cell BE BLASTP on a PS3 (speedup is compared to the runtime in Table 2)

Query length range	Cell BE BLASTP				Speedup
	Stages 1 and 2 on PPE and all SPE computation	Stage 3 on PPE	Stage 4 on PPE	Total	
1–300	28.9	1.77	0.74	32.9	1.41
301–500	35.4	3.10	0.81	40.9	2.39
501–800	44.5	4.30	1.10	51.5	3.06
801–1100	52.8	4.74	1.83	61.6	3.33
1101–1400	61.8	10.25	4.18	79.0	3.31
1401–1700	67.2	15.19	7.98	92.4	3.57
1701–2000	83.9	18.77	4.57	109.0	3.54

Table 4

Average number of sequences processed by each stage of FSA-BLASTP on a P4 and by the PPE in Cell BE BLASTP

Query length	FSA-BLASTP			Cell BE BLASTP (PPE only)			Matching output
	Stages 1 and 2	Stage 3		Stages 1 and 2	Stage 3		
		Semi	Gapped		Semi	Gapped	
1–300		96,954	9443	2113	2062	1731	328
301–500		334,494	13,749	2591	2570	1462	324
501–800		617,225	19,602	5480	5471	3713	443
801–1100	6,375,605	586,139	24,163	5408	5402	3569	471
1101–1400		761,097	34,028	7193	7189	5178	443
1401–1700		1,096,186	43,616.1	15,404	15,402	12,901	438
1701–2000		1,206,705	38,761	6734	6733	4126	428

for Stages 1 and 2 on the PPE also includes the prefiltration steps (Stages 1–3) on the SPEs (see Fig. 7). The speedup of the Cell BE mostly comes from this step which is running on the six SPEs and the PPE in parallel. Note that Stage 3 on the PPE is significantly faster compared to Step 3 of FSA-BLASTP since it is executed for fewer sequences. The numbers of sequences that are processed in each stage by FSA-BLASTP and in the PPE by Cell BE BLASTP are shown in Table 4. In FSA-BLASTP, every database sequence is processed by Stages 1 and 2. The PPE in Cell BE BLASTP only processes a very small faction of database sequences since most sequences have been filtered by SPEs in parallel. This reduced number of sequences contributes to the less total runtime of Cell BE BLASTP. However, the ideal speedup of around six is not reached since the parallel SPE filters add some data transfer and coordination overhead and the PPE is less powerful than a P4. It should also be noted that the speedup for shorter query sequences is generally lower since the runtime is too short to effectively compensate for the associated overheads.

Also note that for Cell BE BLASTP in Table 4, the number of database sequences is larger than the number of found matching sequences. This can be explained as follows. Firstly, if a sequence is too long to be sent to the SPE, it will be processed by the PPE directly. In the experiment, 72 sequences are longer than the maximum buffer length (10 kB). Secondly, some sequences in Stages 1–3 in the SPE exceed the maximum available memory space. These sequences are returned as matches and need further processing on the PPE.

Figure 13 also shows that some query sequences require more processing time by both FSA-BLASTP and Cell BE BLASTP than queries of similar lengths. The statistics of the three such exceptional sequences is shown in Table 5. It can be seen that for these three queries, a bigger number of database sequences need to be processed than average. This increases both CPU and PPE workload.

6. Conclusion

In this paper, we have presented parallelization strategies for scanning protein sequence databases on

Table 5

Runtime statistics (in seconds) of three exceptional sequences

Query length	Method	Time				
		Stages 1 and 2	Stage 3		Stage 4	Total
			Semi	Gapped		
605	FSA-BLAST	63.55	160.89	6.40	0.80	232.13
	Cell BE	58.65	11.63	1.85	1.88	75.61
1455	FSA-BLAST	138.97	348.58	0.38	24.96	513.43
	Cell BE	84.48	65.49	1.28	66.17	219.43
1945	FSA-BLAST	225.87	316.33	1.02	2.63	546.35
	Cell BE	132.11	109.53	1.36	6.98	251.52

Query length	Method	Number of sequences			Matching output
		Stages 1 and 2	Stage 3		
			Semi	Gapped	
605	FSA-BLAST	6,375,605	1,890,358	33,061	500
	Cell BE	5536	5536	2288	500
1455	FSA-BLAST	6,375,605	2,981,242	23,895	500
	Cell BE	8344	8344	4115	500
1945	FSA-BLAST	6,375,605	1,555,474	170,541	500
	Cell BE	27,681	27,677	25,473	500

the Cell BE using two approaches: (1) the exhaustive DP SW-method, and (2) the BLASTP heuristic. In order to derive efficient mappings onto this type of heterogeneous multi-core architecture, we have utilized SIMD vectorization, parallel data partitioning and communication schemes, and a compressed deterministic finite state automaton for hit detection in order to reduce memory consumption. Our Cell BE SW implementation achieves a performance of up to 3663.46 MCUPS on a PS$^{®}$3. This performance of about 30 times faster than the straightforward C-implementation in SSEARCH. It is also 1.64 faster than the highly optimized striped SW implementation on an Intel processor and around 3 times faster than a CUDA implementation on an Nvidia GeForce 8800GTX. Our BLASTP implementation on a PS$^{®}$3 achieves an average speedup of 3.2 compared to the optimized FSA-BLASTP and 3.6 compared to NCBI-BLASTP. The very rapid growth of biological sequence databases demands even more powerful high-performance solutions in the near future. Hence, our results are especially encouraging since high performance computer architectures are developing towards heterogeneous multi-core systems. Therefore, the techniques presented in this paper are of particular importance since they compare and analyze the efficiency of parallelization approaches on different parallel architectures.

Acknowledgment

The work was supported by the A*Star BMRC Research Grant 04/1/22/19/375.

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) (1990), 403–410.

[2] S.F. Altschul, T.L. Madden, A.A. Schäffer, J. Zhang, Z. Zhang, W. Miller and D.J. Lipman, Gapped BLAST and PSI-BLAST: A new generation of protein database search programs, *Nucleic Acid Research* **25**(7) (1997), 3389–3402.

[3] S. Brenner, C. Chothia and T. Hubbard, Assessing sequence comparison methods with reliable structurally identified distant evolutionary relationships, *Biochemistry* **95**(11) (1998), 6073–6078.

[4] M. Cameron, H.E. Williams and A. Cannane, Improved gapped alignment in BLAST, *IEEE Transactions on Computational Biology and Bioinformatics* **1**(3) (2004), 116–129.

[5] M. Cameron, H.E. Williams and A. Cannane, A deterministic finite automaton for faster protein hit detection in BLAST, *Journal of Computational Biology* **13**(4) (2006), 965–978.

[6] M. Farrar, Striped Smith–Waterman speeds database searches six times over other SIMD implementations, *Bioinformatics* **23**(2) (2007), 156–161.

[7] A. Jacob, J. Lancaster, B. Harris, J. Buhler and R. Chamberlain, Mercury BLASTP: Accelerating protein sequence alignment, *ACM Transactions on Reconfigurable Technology and Systems* **1**(2) (2008), Article 9.

[8] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer and D. Shippy, Introduction the Cell multiprocessor, *IBM Journal of Research and Development* **49**(4/5) (2005), 598–604.

[9] W.J. Kent, BLAT – The BLAST-like alignment tool, *Genome Research* **12**(4) (2002), 656–664.

[10] M. Kistler, F. Perrone and F. Petrini, Cell multiprocessor communication network: built for speed, *IEEE Micro* **26**(3) (2006), 10–23.

[11] M. Li, B. Ma, D. Kisman and J. Tromp, Patternhunter II: Highly sensitive and fast homology search, *Journal of Bioinformatics and Computational Biology* **2**(3) (2004), 417–439.

[12] I.T. Li, W. Shum and K. Truong, 160-fold acceleration of the Smith–Waterman algorithm using a field programmable gate array (FPGA), *BMC Bioinformatics* **8**(185) (2007).

[13] W. Liu, B. Schmidt, G. Voss and W. Müller-Wittig, Streaming algorithms for biological sequence alignment on GPUs, *IEEE Transactions on Parallel and Distributed Systems* **18**(10) (2007), 1270–1281.

[14] S.A. Manavski and G. Valle, CUDA compatible GPU cards as efficient hardware accelerators for Smith–Waterman sequence alignment, *BMC Bioinformatics* **9**(Suppl. 2) (2008), S10.

[15] C. Oehmen and J. Nieplocha, ScalaBLAST: A scalable implementation of BLAST for high-performance data-intensive bioinformatics analysis, *IEEE Transactions on Parallel and Distributed Systems* **17**(8) (2006), 740–749.

[16] T. Oliver, B. Schmidt and D. Maskell, Reconfigurable architectures for bio-sequence database scanning on FPGAs, *IEEE Transactions on Circuits and Systems II* **52**(12) (2005), 851–855.

[17] W.R. Pearson, Flexible sequence similarity searching with the FASTA3 program package, *Methods in Molecular Biology* **132** (2000), 185–219.

[18] T. Rognes and E. Seeberg, Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors, *Bioinformatics* **16**(8) (2000), 699–706.

[19] V. Sachdeva, M. Kistler, E. Speight and K. Tzeng, Exploring the viability of the Cell Broadband Engine for bioinformatics applications, in: *6th IEEE International Workshop on High Performance Computational Biology (HiCOMB 2007)*, Long Beach, CA, USA, 2007.

[20] T.F. Smith and M.S. Waterman, Identification of common molecular subsequences, *Journal of Molecular Biology* **147**(1) (1981), 195–197.

[21] A. Wozniak, Using video-oriented instructions to speed up sequence comparison, *Computer Applications in Biosciences* **13**(2) (1997), 145–150.