Experiences with Parallelizing a Bio-informatics Program on the Cell BE

Hans Vandierendonck, Sean Rul, Michiel Questier, and Koen De Bosschere

Ghent University, Department of Electronics and Information Systems/HiPEAC, B-9000 Gent, Belgium
{hvdieren,srul,mquestie,kdb}@elis.ugent.be

Abstract. The Cell Broadband Engine Architecture is a new heterogeneous multi-core architecture targeted at compute-intensive workloads. The architecture of the Cell BE has several features that are unique in high-performance general-purpose processors, such as static instruction scheduling, extensive support for vectorization, scratch pad memories, explicit programming of DMAs, mailbox communication, multiple processor cores, etc. It is necessary to make explicit use of these features to obtain high performance. Yet, little work reports on how to apply them and how much each of them contributes to performance.

This paper presents our experiences with programming the Cell BE architecture. Our test application is Clustal W, a bio-informatics program for multiple sequence alignment. We report on how we apply the unique features of the Cell BE to Clustal W and how important each is to obtain high performance. By making extensive use of vectorization and by parallelizing the application across all cores, we speedup the pairwise alignment phase of Clustal W with a factor of 51.2 over PPU (superscalar) execution. The progressive alignment phase is sped up by a factor of 5.7 over PPU execution, resulting in an overall speedup by 9.1.

1 Introduction

Computer architectures are changing: while previous generations of processors gained performance by increasing clock frequency and instruction-level parallelism, future processor generations are likely to sustain performance improvements by increasing the number of cores on a chip. These performance improvements can, however, only be tapped when applications are parallel. This requires a large additional effort on the side of the programmer. Furthermore, it is likely that future multi-core architectures will be heterogeneous multi-cores, i.e., the chip's cores have significantly different architectures. This further increases the programming challenge.

The Cell Broadband Engine Architecture [1] is such a new heterogeneous multi-core architecture targeted at compute-intensive workloads. The Cell BE has one superscalar processor (Power processing element) and 8 SIMD *synergistic* processing elements (SPE). The SPEs have a unique architecture, with features that are uncommon in high-performance general-purpose processors:

static instruction scheduling, scratch pad memories, explicit programming of DMAs, mailbox communication, a heterogeneous multi-core architecture, etc. While these features hold promise for high performance, achieving high performance is difficult as these features are exposed to the programmer.

In this paper, we implement Clustal W [2], a bio-informatics program, on the Cell Broadband Engine and report on the optimizations that were necessary to achieve high performance. Apart from overlapping memory accesses with computation and apart from avoiding branches, we spend a lot of effort to vectorize code, modify data structures, and to remove unaligned vector memory accesses. These optimizations increase performance on an SPE. Furthermore, we extract thread-level parallelism to utilize all 8 SPEs. We report on the impact of each of these optimizations on program speed.

In the remainder of this paper, we first explain the Cell Broadband Engine Architecture (Section 2) and the Clustal W application (Section 3). We analyze Clustal W to find the most time-consuming phases (Section 4). Then, we explain our optimizations on Clustal W (Section 5) and evaluate the performance improvements from each (Section 6). Finally, Section 7 discusses related work and Section 8 concludes the paper.

2 The Cell BE Architecture

The Cell Broadband Engine [1] is a heterogeneous multi-core that is developed by Sony, Toshiba and IBM. The Cell consists of nine cores: one PowerPC processor element (PPE) and eight SIMD synergistic processor elements (SPE).[1]

The PPE serves as a controller for the SPEs and works with a conventional operating system. It is derived from a 64-bit PowerPC RISC-processor and is an in-order two-way superscalar core with simultaneous multi-threading. The instruction set is an extended PowerPC instruction set with SIMD Multimedia instructions. It uses a cache coherent memory hierarchy with a 32 KB L1 data and instruction cache and a unified L2 cache of 512 KB.

The eight SPEs [3,4] deliver the compute power of the Cell processor. These 128-bit in-order vector processors distinguish themselves by the use of explicit memory management. The SPEs each have a *local store* of 256 KB dedicated for both data and instructions. The SPE only operates on data in registers which are read from or written to the local store. Accessing data from the local store requires a constant latency of 6 cycles as opposed to processors with caches who have various memory access times due to the underlying memory hierarchy. This property allows the compiler to statically schedule the instructions for the SPE. To access data that resides in the main memory or other local stores, the SPE issues a DMA command. The register file itself has 128 registers each 128-bit wide allowing SIMD instructions with varying element width (e.g. ranging from

[1] The processing units are referred to as the power processing unit (PPU) and synergistic processing unit (SPU) for the PPE and SPE, respectively. We will use the terms PPE and PPU, and SPE and SPU, interchangeably.

2x64-bit up to 16x8-bit). There is no hardware branch predictor in order to keep the design of the SPE simple. To compensate for this, the programmer or compiler can add branch hints which notifies the hardware and allows prefetching the upcoming 32 instructions so that a correctly hinted taken branch incurs no penalty. Since there is a high branch misprediction penalty of about 18 cycles it is better to eliminate as much branches as possible. The SIMD select instruction can avoid branches by turning control flow into data flow.

All nine cores, memory controller and I/O controller are connected through the Element Interconnect Bus (EIB). The EIB consists of 4 data rings of 16 bytes wide. The EIB runs at half the frequency of the processor cores and it supports a peak bandwidth of 204.8 GBytes/s for on chip communication. The bandwidth between the DMA engine and the EIB bus is 8 bytes per core per cycle in each direction. Because of the explicit memory management that is required in the local store one has to carefully schedule DMA operations and strive for total overlap of memory latency with useful computations.

3 Clustal W

In molecular biology Clustal W [2] is an essential program for the simultaneous alignment of nucleotide or amino acid sequences. It is also part of Bioperf [5], an open benchmark suite for evaluating computer architecture on bioinformatics and life science applications.

The algorithm computes the most likely mutation of one sequence into the other by iteratively substituting amino acids in the sequences and by introducing gaps in the sequences. Each modification of the sequences impacts the score of the sequences, which measures the degree of similarity.

The alignment of two sequences is done by dynamic programming, using the Smith-Waterman algorithm [6]. This technique, however, does not scale to aligning multiple sequences, where finding a global optimum becomes NP-hard [7]. Therefore, a series of pairwise alignments is compared to each other, followed by a progressive alignment which adds the sequence most closely related to the already aligned sequences.

The algorithm consists of three stages. In the first stage, all pairs of sequences are aligned. The second stage forms a phylogenetic tree for the underlying sequences. This is achieved by using the Neighbor-Joining algorithm [8] in which the most closely related sequences, as given by the first stage, are located on the same branch of the guide tree. The third step progressively aligns the sequences according to the branching order in the guide tree obtained in the second step, starting from the leaves of the tree proceeding towards the root.

4 Analysis of Clustal W

Both the space and time complexity of the different stages of the Clustal W algorithm are influenced by the number of aligned sequences N and the typical sequence length L. Edgar [9] has computed the space and time complexity of

Table 1. Complexity of Clustal W in time and space, with N the number of sequences and L the typical sequence length

Stage	$\mathcal{O}$(Space)	$\mathcal{O}$(Time)
PW: Pairwise calculation	$N^2 + L^2$	N^2L^2
GT: Guide tree	N^2	N^4
PA: Progressive alignment	$NL + L^2$	$N^3 + NL^2$
Total	$N^2 + L^2$	$N^4 + L^2$

each phase in terms of N and L (see Table 1). This theoretical analysis indicates that the second stage of Clustal W will become more important as the number of sequences increases, but it is indifferent to the length of these sequences. In the other stages both the number of sequences and the length are of importance.

The time and space complexity indicate how each phase scales with increasing problem size, but they does not tell us the absolute amount of time spent in each phase. In order to understand this, we analyze the program by randomly creating input sets with preset number of sequences N and sequence length L. Statistical data of protein databases [10] indicates that the average length of sequences is 366 amino acids, while sequences with more than 2000 amino acids are very rare. So we randomly created input sets with a sequence length ranging from 10 to 1000 amino acids and with a number of sequences in the same range.

Figure 1 shows the percentage of execution time spent in each stage. Each bar indicates a certain configuration of the number of sequences (bottom X-value) and the sequence length (top X-value). The pairwise alignment becomes the dominant stage when the number of sequences is high, taking responsibility for more than 50% of execution time when there are at least 50 sequences. In contrast, when the number of sequences is small, then progressive alignment takes the larger share of the execution time.

The guide tree only plays an important role when the input set contains a large number of short sequences. In other cases this stage is only responsible for less than 5% of the execution time. In a previous study, G-protein coupled receptor

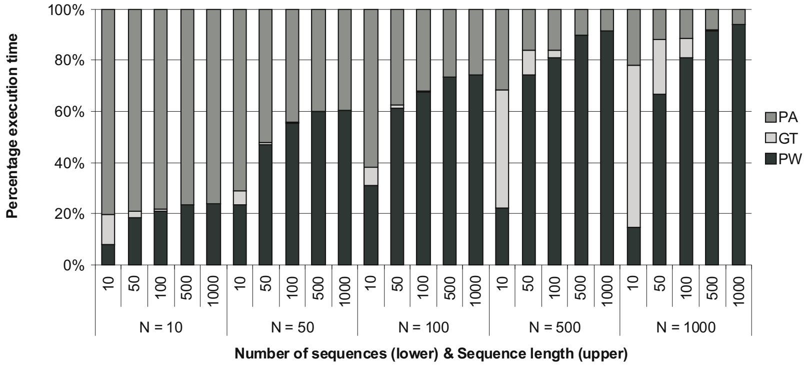


Fig. 1. Percentage of execution time of the three major stages in Clustal W

(GPCR) proteins are used as input sets [11]. These proteins are relatively short, so the guide tree plays a prominent role. A profile analysis of ClustalW-SMP [12] shows a more important role for the guide tree, but this is the effect of the SMP version of Clustal W in which both the pairwise and progressive alignment are parallelized.

The analysis above shows that the pairwise alignment and progressive alignment phases are responsible for the largest part of the execution time. In the remainder of this paper, we focus on optimizing these two phases and pay no attention to the guide tree phase (which is parallelized in [12]).

5 Optimization of Clustal W

The inner loops of the pairwise alignment and progressive alignment phases have very similar structure, so most optimizations apply to both phases. We discuss first how to optimize these phases for the SPUs. Then we turn our attention to parallelizing these phases to utilize multiple SPUs.

5.1 Optimizing for the SPU

Loop Structure. The majority of work in the PW and the PA phases is performed by 3 consecutive loop nests. Together, these loop nests compute a metric of similarity (score) for two sequences. The first loop nest iterates over the sequences in a forward fashion, i.e., it uses increasing indices for the sequence arrays. We call this loop the forward loop. The second loop nest iterates over the sequences in a backward fashion (using decreasing indices for the sequence arrays), so we call it the backward loop. From a computational point of view, a single iteration of the forward and the backward loops perform comparable computations. The third loop nest computes the desired score using intermediate values from the forward and backward loops (note that in PA the third loop nest also uses recursion besides iteration).

In both the PW and PA phases, the third loop performs an order of magnitude less work than the forward and the backward loops, so we do not bother to optimize the third loop.

In PW, the forward loop is by far more important than the backward loop, as the forward loop computes reduced limits on the iteration space of the backward loop. In PA, the forward and the backward loop have the same size of iteration space. Hence, they take a comparable share in the total execution time.

The forward and backward loop bodies contain a non-vectorizable part. In the PW loops, these are scattered accesses to a substitution matrix, while in the PA loops, these involve calls to a function called `prfscore()`. This structure limits the speedup achievable by vectorization of the surrounding loops.

We optimize the forward and backward loops using vectorization (SIMD) and loop unrolling. These optimizations are known to be very effective on the Cell BE. Before applying them, we must understand the control flow inside the inner loop bodies as well as the data dependencies between successive loop iterations.

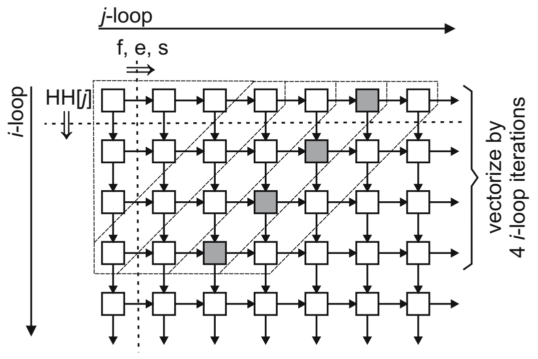


Fig. 2. Data dependencies between distinct iterations of the inner loop body of two nested loops

Vectorization of `prfscore()`. The `prfscore()` function called from the PA loops computes a vector dot product. The vector length is input-dependent but it cannot exceed 32 elements by definition of the data structures. Furthermore, the vector length remains constant during the whole PA phase.

We completely unroll the loop assuming 32 iterations of the loop and we perform a 4-way vectorization. This removes all control flow at the cost of code size increase. To take the loop iteration limit into account, we use the `spu_sel()` primitive together with a pre-computed mask array. The `spu_sel()` primitive selects only those words for which the mask contains ones, so it allows us to sum over only those values that are required.

Control Flow Optimization. The inner loop body of the backward and forward loops contains a significant amount of control flow, related to finding the maximum of a variable over all loop iterations. In the PW phase, the code also remembers the *i*-loop and *j*-loop iteration numbers where that maximum occurs. It is important to avoid this control flow, since mispredicted branch instructions have a high penalty on the SPUs. Updating the running maximum value (`if(b > a) a=b;`) can be simply avoided by using the SPUs compare and select assembly instructions to turn control flow into data flow (`a=spu_sel(a,b,spu_cmpgt(b,a));`). In the same vein, it is also possible to remember the *i*-loop and *j*-loop iteration numbers of the maximum (`imax=spu_sel(imax,i,spu_cmpgt(b,a));`).

Vectorization. The forward and backward loops in the PW and PA phases have the same data dependencies, which are depicted in Figure 2. There are two nested loops, with the *j*-loop nested inside the *i*-loop. Every box in the figure depicts one execution of the inner loop body corresponding to one pair of *i* and *j* loop indices. The execution of the inner loop body has data dependencies with previous executions of the inner loop body, as indicated by edges between the boxes. Data dependencies carry across iterations of the *j*-loop, in which case the dependencies are carried through scalar variables. Also, data dependencies carry across iterations of the *i*-loop, in which case the dependences are carried through arrays indexed by j. Thus, the execution of the inner loop body has data dependencies with two prior executions of the loop body.

To vectorize the forward and backward loops, we need to identify V data-independent loop iterations, where V is the vectorization factor. In these loops, the vectorization factor is 4, since scalar variables are 32-bit integers and the vector length is 128 bits. Data-independent loop iterations occur for skewed iteration counts for the i-loop and the j-loop. In particular, loop iterations (i_0, j_0) and (i_1, j_1) are independent when $i_0 + j_0 = i_1 + j_1$. Consequently, vectorization requires the construction of loop pre-ambles and post-ambles to deal with non-vectorizable portions of the loop body.

The PW forward loop computes the position of the maximum score. The original code records the "first" loop iteration where the maximum value occurs (`if(b > a){a=b; imax=i; jmax=j;}`). Here, the "first" loop iteration is the one that occurs first in the lexicographic ordering

$$(i,j) < (i^{'},j^{'}) \quad \text{if} \quad (i < i^{'}) \vee ((i = i^{'}) \wedge (j < j^{'})).$$

Since vectorization changes the execution order of the loop iterations, we need to take care that the *same* loop iteration is recorded in order to obtain the same output of the algorithm. In the vectorized code, we simultaneously remember 4 positions where the maximum value occurs, each one corresponding to one of the 4 vector lanes. When the vectorized loop has finished, we need to select the maximum value among the per-lane maxima and, if that maximum occurs in multiple lanes, we need to select the appropriate loop iterations corresponding to the lexicographic ordering in the original code.

Loop Unrolling to Avoid Unaligned Memory Accesses. Loop unrolling can increase performance by increasing the range across which instructions can be scheduled and by reducing control flow overhead. Loop unrolling is particularly important for statically scheduled architectures like the SPU. In the case of Clustal W, however, loop unrolling did not allow the compiler to create a better instruction schedule as the loop body already contains sufficient instruction-level parallelism. Thus, performance remains the same.

In this paper, we show that loop unrolling is also useful to enable other optimizations, in this case the removal of unaligned memory accesses. Unaligned memory accesses should be avoided as the hardware supports only aligned memory accesses. Consequently, unaligned vector loads and stores translate into a sequence of several instructions.

The interaction of loop unrolling and alignment is illustrated on the $HH[\cdot]$ array, which is used in the inner loop body (Figure 3). We assume that the vector covering elements 1 to 4 of the $HH[\cdot]$ array is aligned. This is the optimal situation since the j-loop starts at index 1.

In the vectorized loops, each loop iteration loads a 4-element vector from the $HH[\cdot]$ array. Depending on the iteration count, this vector may or may not be aligned on a natural boundary. Every iteration, the vector moves one scalar position in the $HH[\cdot]$ array, so the loaded vector is aligned exactly once every fourth iteration of the loop and it is unaligned in the other iterations.

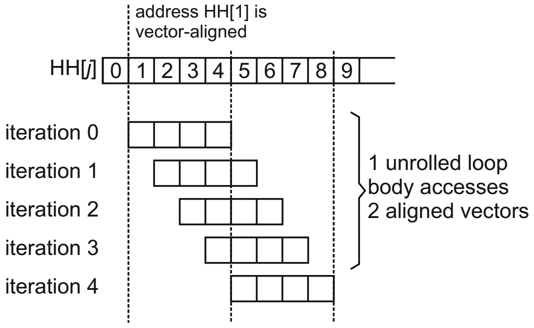

address HH[1] is vector-aligned
HH[j]
0
1
2
3
4
5
6
7
8
9
iteration 0
iteration 1
iteration 2
iteration 3
iteration 4
1 unrolled loop body accesses 2 aligned vectors

Fig. 3. Elements of the intermediary $HH[\cdot]$ array accessed by successive iterations of the vectorized loop

Four consecutive iterations of the vectorized loop access 7 distinct scalars from the $HH[\cdot]$ array (Figure 3). These 7 scalars are located in two consecutive aligned vectors, so it is possible to load them all at once into vector registers using two aligned loads, and to store them back using two aligned stores. All further references to the $HH[\cdot]$ array are now redirected to the vector registers holding the two words. This optimization removes all unaligned memory accesses to the arrays that carry dependences between iterations of the i-loop. We apply a similar optimization to the character arrays holding the sequences in the pairwise alignment phase.

5.2 Modifications to Data Structures

As the local store is not large enough to hold all data structures, we stream all large data structures in and out of the SPUs. This is true in particular for the sequence arrays (PW phase) and for the profiles (PA phase). We also carefully align all datastructures in the local store to improve vectorization.

In the PA phase, we also modify the second profile, which streams through the SPU most quickly. Each element of the profile is a 64-element vector of 32-bit integers. This vector is accessed sparsely: the first 32 elements are accessed sequentially, the next 2 elements are accessed at other locations in the code and the remainder is unused. To improve memory behavior, we create two new arrays to store the 32nd and 33rd elements. Accesses to these arrays are optimized in the same way as to the $HH[\cdot]$ array of the previous paragraph. When streaming the second profile, only the first 32 elements are fetched.

5.3 Parallelization of Pairwise Alignment

Pairwise alignment computes a score for every pair of sequences. The scores can be computed independently for all pairs, which makes parallelization trivial. We dynamically balance the work across the SPUs by dividing the work in $N-1$ work packages where N is the number of sequences. The i-th work package

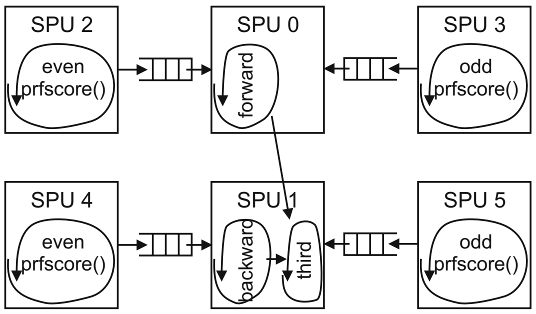


Fig. 4. Parallelization of `pdiff()` on 6 SPUs

corresponds to comparing the i-th sequence to all other sequences j where $j > i$. Work packages are sent to SPUs in order of decreasing size to maximize load balancing.

5.4 Parallelization of Progressive Alignment

Progressive alignment is more difficult to parallelize. Although the forward, backward and third loop nests are executed multiple times, there is little parallelism between executions of this set of loops. A parallelization scheme similar to the PW phase is thus not possible. Instead, we note that the first two loop nests are control- and data-independent. The third loop nests has data-dependencies with the first two loop nests, but its execution time is several orders of magnitude smaller. So a first parallelization is to execute the first two loop nests in parallel, an optimization that is also performed in the SMP version of Clustal W.

A higher degree of parallelization is obtained by observing that most of the execution time is spent in the `prfscore()` function. As the control flow through the loops is entirely independent of the data, we propose to extract DO-ACROSS parallelism from the loop. Indeed, the `prfscore()` function can be evaluated ahead of time as it is independent of the remainder of the computation. As the `prfscore()` function takes a significant amount of time, we reserve two threads to evaluate this function, each handling different values.

Thus, we instantiate three copies of the loop (Figure 4). Two copies compute each a subset of the `prfscore()`s and send these values to the third copy through a queue. The third copy of the loop performs the remaining computations and reads the results of the `prfscore()`s from the queue. As control flow is highly predictable, it is easy to divise a static distribution of work, such that each copy of the loop can proceed with minimum communication. The only communication is concerned with reading and writing the queue.

In total, a single call to `pdiff()` is executed by 6 SPU threads: one for the forward loop, one for the backward and third loop, two threads to deal with the `prfscore()`s for the forward loop and two more threads to deal with the `prfscore()`s for the backward loop.

6 Evaluation

We separately evaluate the effect of each optimization on Clustal W to understand the relative importance of each optimization. Hereto, we created distinct versions of Clustal W, with each one building upon the previous version and adding optimizations to it. The baseline version of Clustal W is taken from the BioPerf benchmark suite [5]. The programs are run with the B input from the same benchmark suite. We present results only for one input set, as distinct input sets assign different importance to each of the phases, but the performance of each phase scales similarly across input sets.

We evaluate the performance of each of our versions of Clustal W by running it on a Dual Cell BE-based blade, with two Cell Broadband Engine processors at 3.2 GHz with SMT enabled. The compiler is gcc 4.0.2 and the operating system is linux (Fedora Core 5). We added code to measure the overall wall clock time that elapses during the execution of each phase of Clustal W. Each version of Clustal W is run 5 times and the highest and lowest execution times are dropped. We report the average execution time over the 3 remaining measurements.

We first discuss the effect of SPU-specific optimizations on performance. Here, only a single SPU thread is used. Then, we discuss how performance scales with multiple SPUs.

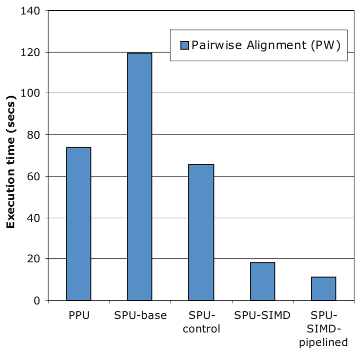
6.1 Pairwise Alignment

Figure 5(a) shows the effects of the individual optimizations on the performance of pairwise alignment. The first bar (labeled “PPU”) shows the execution time of the original code running on the PPU. The second bar (“SPU-base”) shows the execution time when the pairwise alignment is performed on a single SPU. The code running on the SPU is basically the code from the original program, extended with the necessary control, DMA transfers and mailbox operations. Although this overhead adds little to nothing to the overall execution time, we observe an important slowdown of execution. Inspection of the code shows a high density of control transfers inside the inner loop body of the important loop nests. Removing this control flow makes a single SPU already faster than the PPU (“SPU-control”).

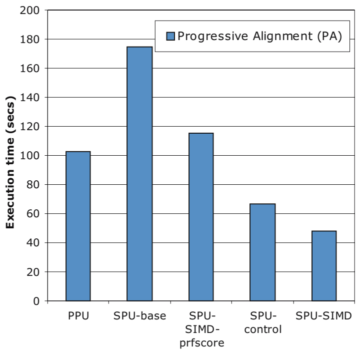
The next bar (“SPU-SIMD”) shows the performance when vectorizing the forward loop. Vectorization yields a 3.6 times speedup. The next step is to unroll the vectorized loop with the goals of removing unaligned memory accesses. This shortens the operation count in a loop iteration and improves performance by another factor 1.7. The overall speedup over PPU-only execution is a factor 6.7. At this point, the backward loop requires an order of magnitude less computation time than the forward loop, so we do not optimize it.

6.2 Progressive Alignment

We perform a similar analysis of progressive alignment (Figure 5(b)). Again we use a single SPU, so the forward and backward loops are executed sequentially on the same SPU and the `prfscore()` functions are executed by the same thread.



(a) Effect of optimizations on PW



(b) Effect of optimizations on PA

Fig. 5. The effect of each of the optimizations on the execution time of pairwise aligne-ment and progressive alignment

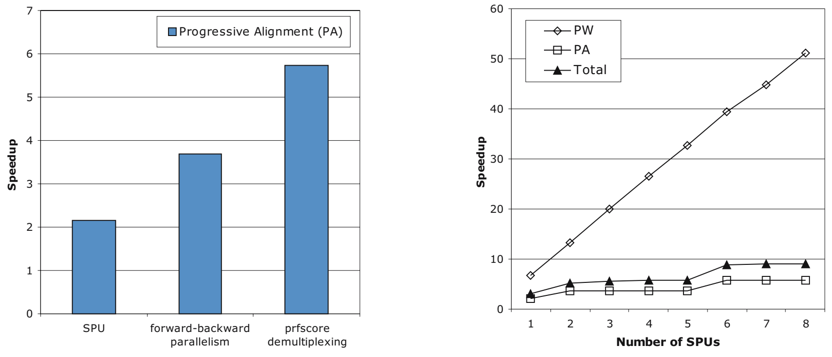
Again, executing the original code on the SPU is slower than running on the PPU (bar "SPU-base" vs. "PPU"). Again, we attribute this to excessive control flow. In PA, we identify two possible causes: the loop inside the `prfscore()` function and the remaining control flow inside the loop bodies of the forward and backward loops. First, we remove all control flow in the `prfscore()` function by unrolling the loop, vectorizing and by using pre-computed masks to deal with the loop iteration count (see Section 5.1). This brings performance close to the PPU execution time (bar "SPU-SIMD-prfscore"). Second, we remove the remaining control flow in the same way as in the PW loop nests. This gives an overall speedup of 1.6 over PPU execution (bar "SPU-control").

Vectorizing the forward and backward loops improves performance, but the effect is relatively small (bar "SPU-SIMD"). The reason is that the inner loop contains calls to `prfscore`. The execution of these calls remains sequential, which significantly reduces the benefit of vectorization. Since these calls are also responsible for most of the execution time, there is no benefit from unrolling the vectorized loops as the unaligned memory accesses are relatively unimportant compared to `prfscore()`. Furthermore, removing unaligned memory accesses requires many registers but the vectorized loop nest is already close to using all registers.

6.3 Scaling with Multiple SPUs

The final part of our analysis concerns the scaling of performance when using multiple SPUs. In the following, we use the best version of each phase. Figure 6(b) shows the speedup over PPU-only execution when using an increasing number of SPUs.

As expected, the PW phase scales very well with multiple SPUs. With 8 SPUs, the parallelized and optimized PW phase runs 51.2 times faster than the original code on the PPU.



(a) Parallelization of PA (b) Speedup against number of SPUs

Fig. 6. Speedup of a multi-threaded Clustal W over PPU-only execution

The PA phase has less parallelism than the PW phase. We present results for three versions (Figure 6(a)): a single-SPU version, a 2-SPU version where the forward and backward loops execute in parallel, and a 6-SPU version where the `prfscore()` function is evaluated by separate threads.

Executing the forward and backward loops in parallel yields a 1.7 speedup over single-SPU execution. The 6-SPU version improves the 2-SPU version by 1.6. This latter speedup is not very high, as we replace the straight-line code of `prfscore()` by control-intensive code to send the values to a different SPU. Although this communication is buffered, it is necessary to perform additional checks on buffer limits and to poll the incoming mailbox.

Figure 6(b) also shows the overall speedup for the B input of Clustal W. For this input, the guide tree phase requires virtually no execution time. The total execution time is represented by the PW and PA phases. With a single SPU, our Cell BE implementation is 3 times faster than the original PPU-only version. With 8 SPUs, our parallel version is 9.1 times faster.

6.4 Discussion

Optimizing the Clustal W program for the Cell BE has given us valuable insight into this processor. There are a few things that deserve pointing out.

First, although the SPE's local store can be perceived of as small (256 KB), there is little point in having larger local stores. Clearly, there isn't a local store that will be large enough to hold all of the application's data, regardless of input set size. So it will always be necessary to prepare a version of the SPU code that streams all major data structures in and out of the local store. Given this assumption, our experience is that 256 KB is enough to hold the compute-intensive kernels, some small data structures as well as buffers to stream the major data structures.

Second, the DMA interface is rich and easy to use, although it is necessary to carefully plan when each DMA is launched. An interesting trick is the use of barriers, which force the execution of DMAs in launch order. We use this feature when copying a buffer from one SPU to another, followed by sending a mailbox message[2] to notify the destination SPU that the buffer has arrived. By using a barrier, we can launch the message without waiting for completion of the buffer DMA. Furthermore, tag queues can be specified such that the barrier applies only to the DMAs in the specified tag queue. Thus, other DMAs (e.g., to stream data structures) are not affected by the barrier.

We experienced some properties of the Cell BE as limitations. E.g., 32-bit integer multiplies are not supported in hardware. Instead, the compiler generates multiple 16-bit multiplies. This is an important limitation in the `prfscore()` function, which extensively uses 32-bit multiplies. Also, the DMA scatter/gather functionality was not useful to us as we needed 8 byte scatter/gather operations but the cell requires that the data elements are at least 128 byte large.

Finally, although mailbox communication is easy to understand, it is a very raw device to implement parallel primitives. We find that mailboxes provide not enough programmer abstraction and are, in the end, hard to use. One problem results from sending all communication through a single mailbox. This makes it impossible to separately develop functionality to communicate with a single SPU, as this functionality can receive unexpected messages from a different SPU and it must know how to deal with these messages. An interesting solution could be the use of tag queues in the incoming mailbox, such that one can select only a particular type or source of message.

7 Related Work

The Cell BE Architecture promises high performance at low power consumption. Consequently, several researchers have investigated the utility of the Cell BE for particular application domains.

Williams et al. [13] measure performance and power consumption of the Cell BE when executing scientific computing kernels. They compare these numbers to other architectures and find potential speedups in the 10-20x range. However, the low double-precision floating-point performance of the Cell is a major down-side for scientific applications. A high-performance FFT is described in [14].

Héman et al. [15] port a relational database to the Cell BE. Only some database operations (such as projection, selection, etc.) are executed on the SPUs. The authors point out the importance of avoiding branches and of properly preparing the layout of data structures to enable vectorization.

Bader et al. [16] develop a list ranking algorithm for the Cell BE. List ranking is a combinatorial application with highly irregular memory accesses. As memory accesses are hard to predict in this application, it is proposed to use software-managed threads on the SPUs. At any one time, only one thread is running. When it initiates a DMA request, the thread blocks and control switches to another thread. This results in a kind of software fine-grain multi-threading and yields speedups up to 8.4 for this application.

[2] SPU-to-SPU mailbox communication is implemented using DMA commands.

Bagojevic et al. [17] port a randomized axelerated maximum likelihood kernel for phylogenetic tree construction to the Cell BE. They use multiple levels of parallelism and implement a scheduler that selects at runtime between loop-level parallelism and task-level parallelism.

Also, the Cell BE has been tested using bio-informatics applications. Sachdeva et al. [18] port the FASTA and Clustal W applications to the Cell BE. For Clustal W, they have only adapted the forward loop in the pairwise alignment phase for the SPU. Their implementation of the PW phase takes 3.76 seconds on 8 SPUs, whereas our implementation takes 1.44 seconds. Our implementation is faster due to the removal of unaligned memory accesses, due to the vectorization of address computations when accessing the substitution matrix and also due to optimizing control flow in the backward pass. Furthermore, Sachdeva et al. apply static load balancing while our experiments (not discussed) reveal that dynamic load balancing works better since the comparison of two sequences has variable execution time.

8 Conclusion

The Cell Broadband Engine Architecture is a recent heterogeneous multi-core architecture targeted at compute-intensive workloads. The SPUs, which are the workhorse processors, have rare architectural features that help them to sustain high performance, but they also require specific code optimizations. In this paper, we have investigated what optimizations are necessary and we measured how much they improve performance. We performed our experiments on Clustal W, a well-known bio-informatics application for multiple sequence alignment where we found that (i) executing unmodified code on an SPU is slower than execution on the PPU, (ii) removing control flow from inner loops makes the SPU code already faster than the PPU, (iii) 4-way vectorization improves performance up to 3.6x and (iv) removing unaligned memory accesses gives an important additional speedup in one loop nest. Using these optimizations, we demonstrated a speedup of 51.2 over PPU-only execution for the pairwise alignment phase, 5.7 for the progressive alignment phase and an overall 9.1 speedup.

We found the lack of support for 32-bit integer multiplies most limiting to performance and we found mailbox communication to be the most programmer-unfriendly feature of the SPUs.

Acknowledgements

Hans Vandierendonck is Postdoctoral Research Fellow with the Fund for Scientific Research - Flanders (FWO). Sean Rul is supported by the Institute for the Advancement of Science and Technology in the Industry (IWT). This research was also sponsored by Ghent University and by the European Network of Excellence on High-Performance Embedded Architectures and Compilation (HiPEAC). The authors are gratefull to the Barcelona Supercomputing Center - Centro Nacional de Supercomputacion for access to a Cell Blade.

References

1. Pham, D., et al.: The design and implementation of a first-generation Cell processor. In: IEEE International Solid-State Circuits Conference, pp. 184–592 (2005)
2. Thompson, J.D., Higgins, D.G., Gibson, T.J.: CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Res. 22(22), 4673–4680 (1994)
3. Flachs, B., et al.: The microarchitecture of the synergistic processor for a Cell processor. Solid-State Circuits, IEEE Journal of 41(1), 63–70 (2006)
4. Gschwind, M., Hofstee, P.H., Flachs, B., Hopkins, M., Watanabe, Y., Yamazaki, T.: Synergistic processing in cell's multicore architecture. IEEE Micro 26(2), 10–24 (2006)
5. Bader, D., Li, Y., Li, T., Sachdeva, V.: BioPerf: A Benchmark Suite to Evaluate High-Performance Computer Architecture on Bioinformatics Applications. In: The IEEE International Symposium on Workload Characterization, pp. 163–173 (October 2005)
6. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. Journal of Molecular Biology 147(1), 195–197 (1981)
7. Just, W.: Computational complexity of multiple sequence alignment with SP-score. Journal of Computational Biology 8(6), 615–623 (2001)
8. Saitou, N., Nei, M.: The neighbor-joining method: a new method for reconstructing phylogenetic trees. Mol. Biol. Evol. 4(4), 406–425 (1987)
9. Edgar, R.C.: Muscle: a multiple sequence alignment method with reduced time and space complexity. BMC Bioinformatics 5(1) (2004)
10. Uniprotkb/swiss-prot protein knowledgebase 52.5 statistics, `http://www.expasy.ch/sprot/relnotes/relstat.html`
11. Mikhailov, D., Cofer, H., Gomperts, R.: Performance Optimization of ClustalW: Parallel ClustalW, HT Clustal, and MULTICLUSTAL. White Paper, CA Silicon Graphics (2001)
12. Chaichoompu, K., Kittitornkun, S., Tongsima, S.: MT-ClustalW: multithreading multiple sequence alignment. In: Sixth IEEE International Workshop on High Performance Computational Biology, p. 8 (2006)
13. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the Cell processor for scientific computing. In: Proceedings of the 3rd conference on Computing frontiers, pp. 9–20 (May 2006)
14. Greene, J., Cooper, R.: A parallel 64K complex FFT algorithm for the IBM/Sony/Toshiba Cell broadband engine processor. White Paper (November 2006)
15. Heman, S., Nes, N., Zukowski, M., Boncz, P.A.: Vectorized Data Processing on the Cell Broadband Engine. In: Proceedings of the International Workshop on Data Management on New Hardware (June 2007)
16. Bader, D.A., Agarwal, V., Madduri, K.: On the design and analysis of irregular algorithms on the cell processor: A case study on list ranking. In: 21st IEEE International Parallel and Distributed Processing Symposium (March 2007)
17. Blagojevic, F., Stamatakis, A., Antonopoulos, C.D., Nikolopoulos, D.E.: RAxML-Cell: Parallel phylogenetic tree inference on the cell broadband engine. In: International Symposiumon Parallel and Distributed Processing Systems (2007)
18. Sachdeva, V., Kistler, M., Speight, E., Tzeng, T.H.K.: Exploring the viability of the Cell Broadband Engine for bioinformatics applications. In: Proceedings of the 6th Workshop on High Performance Computational Biology, p. 8 (March 2007)