2011 IEEE International Parallel & Distributed Processing Symposium

Efficient Nonserial Polyadic Dynamic Programming on the Cell Processor

Li Liu, Mu Wang, Jinlei Jiang, Ruizhe Li, and Guangwen Yang
Computer Science and Technology Department, Tsinghua University, Beijing, 100084 China
liuli03@mails.tsinghua.edu.cn; {jjlei, ygw}@tsinghua.edu.cn

***Abstract*—Dynamic programming (DP) is an effective technique for many search and optimization problems. However, the high arithmetic complexity limits its extensive use. Although modern processor architectures with multiple cores and SIMD (single instruction multiple data) instructions provide increasingly high computing power, even the state-of-the-art fully optimized algorithm still largely underutilizes modern multi-core processors. In this paper we propose to improve one family of DP, nonserial polyadic DP (NPDP), targeting a heterogeneous multi-core architecture, the Cell Broadband Engine. We first design a new data layout which efficiently utilizes the on-chip memory system of the Cell processor. Next we devise a *CellNPDP* algorithm with two tiers. The first tier is a SPE (a co-processor on the Cell processor) procedure which efficiently computes a block of data that can fit into one SPE's local store. The second tier is a parallel procedure which enables all SPEs to efficiently compute all blocks of data. To evaluate *CellNPDP*, we use both performance modeling and experiments. The performance model reveals that the processor utilization of NPDP can be independent of the problem size. To empirically evaluate *CellNPDP*, we use two platforms: the IBM QS20 dual-Cell blade and a CPU platform with two latest quad-core CPUs. On both platforms, the processor utilization of *CellNPDP* is larger than 60%, which demonstrates that our optimizations and *CellNPDP* can be architecture-independent. Compared to the state-of-the-art fully optimized algorithm on the CPU platform, *CellNPDP* is 44-fold faster for single-precision and 28-fold faster for double-precision, which is a significant improvement to NPDP.**

Keywords-Bioinformatics, Zuker algorithm, nonserial polyadic dynamic programming, Cell processor, multi-core, SIMD

I. INTRODUCTION

DYNAMIC programming (DP), which aims to find an optimal solution among many potential ones, is an effective technique for many search and optimization problems, e.g., inventory management, scheduling and packaging. However, the high arithmetic complexity limits its extensive use. Modern computer architectures with multiple cores and SIMD (single instruction multiple data) instructions provide increasingly high computing power to perform DP. To improve DP with modern computers, Grama et al. [13] classified DP into four classes: serial monadic DP (SMDP), serial polyadic DP (SPDP), nonserial monadic DP (NMDP) and nonserial polyadic DP (NPDP). In this classification, the terms serial and monadic correspond to uniform data dependences. This kind of DP has been well studied and optimized [19, 20]. The term nonserial polyadic stands for another family of DP with nonuniform data dependences, which is more difficult to be optimized. The applications of NPDP include optimal matrix parenthesization problem, binary search tree, *Zuker* algorithm [17], etc. In this paper, we focus on the NPDP in the *Zuker* algorithm. *Zuker* is an important bioinformatics algorithm for predicting RNA secondary structure. It searches an optimal structure with a minimal free energy.

Recently, there are continuing efforts [7, 24, 25, 26] to improve NPDP on modern multi-core processors. These works demonstrate that the optimizations of tiling (or blocking), helper threading and parallelization can significantly improve NPDP. In spite of the significant performance improvement, these works still largely underutilize modern multi-core processors. Firstly, they all focus on homogeneous processors with cache systems. NPDP on heterogeneous processors or on processors without cache systems has not been studied yet. Second, SIMD capability, which becomes increasingly important to modern processor architectures, is almost all wasted. Third, the parallel performance is not good enough, which still underutilizes the multiple cores on modern processors. Consequently, the processor utilization of the state-of-the-art fully optimized algorithm [26] is less than 4% (please refer to Section VI-C for details).

To further improve NPDP, we take a well-known heterogeneous multi-core architecture, the Cell

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.186

Broadband Engine [6], as the processor platform and identify the optimizations which make NPDP highly efficient on modern multi-core processors. Our main contributions are:

1. A new data layout which significantly improves the utilization of the on-chip memory system.
2. A two-tiered NPDP algorithm *CellNPDP*. The first tier targets the SPE (a co-processor on the Cell processor) procedure and efficiently utilizes the instruction pipelines and SIMD instructions to compute a block of data which can fit into one SPE's local store. The second tier is a parallel procedure which enables all SPEs to efficiently compute all blocks of data.
3. Performance modeling and experiments to evaluate *CellNPDP*. The performance model reveals that the processor utilization of NPDP can be independent of the problem size. Experimental results show that *CellNPDP* is highly efficient on modern multi-core processors. To the best of our knowledge, no Cell implementation of NPDP has been published. So we compare *CellNPDP* to the state-of-the-art fully optimized algorithm on the same CPU platform. On average, *CellNPDP* is 44-fold faster for single-precision and 28-fold faster for double-precision, which significantly improves NPDP.

The rest of the paper is organized as follows. Section II introduces the background and related work. Section III presents an improved data layout. Section IV presents *CellNPDP*. Section V evaluates *CellNPDP* through performance modeling. Section VI empirically evaluates *CellNPDP*. Finally, we conclude this paper in Section VII.

II. Background and Related Work

A. Background

In this subsection, we introduce the NPDP in the *Zuker* algorithm [17]. Figure 1 shows the flowchart, where n is the problem size. This algorithm corresponds to a three-level loop, with algorithmic complexity $n^3/6$. Given the problem size 12, Figure 2 shows an example of NPDP, where the gray nodes are used to compute the black node step by step and each two nodes used in one step are linked with a line in Figure 2. According to Figures 1 and 2, we find two features of NPDP. First, the logical structure is triangular. Second, the data dependences are nonuniform. Each data directly or indirectly depends on the data on its left side or below it. For example, given three data A, B and C in Figure 2, data A directly depends on data B while data B directly depends on data C. This also means that A indirectly depends on C.

B. Related Work

The previous works of optimizing DP can be classified into three categories. The first category [2, 9, 11, 15, 18] focuses on reducing the time complexity and communication overhead with different theoretical parallel models. The second category [1, 4, 10, 22, 23] focuses on the performance on distributed systems, where the communication overhead cannot be neglected. The third category [7, 24, 25, 26] focuses on the performance on modern multi-core processors. Modern multi-core processors with SIMD instructions can provide a level of performance that was formerly possible only on supercomputer and clusters. As this paper focuses on the performance of NPDP on modern multi-core processors, we next review works targeting these processors in more detail.

To improve the performance of NPDP on the multi-core processors with shared cache, Tan et al. [24, 25, 26] first designed a tiling approach to improve the cache reuse and leveraged the helper threads to hide the cache miss latency. Next they designed a parallel algorithm which performs NPDP step by step. In each step, a block of data which can fit into the shared cache is computed by all cores in parallel. Chowdhury et al. [7] further designed a cache-efficient algorithm, which develops a tiling sequence to improve the performance on the multi-core processors with different types of cache systems.

Although these works can achieve a significant performance improvement, they still largely underutilize modern multi-core processors. First, SIMD capability is almost not utilized at all. Note that SIMD becomes increasingly important to boost the computing power of modern processors. Second, when all cores are used, the parallel efficiency is less than 60%, indicating that there is still underutilization of the multiple cores on modern processors. Third, they focus on the homogeneous multi-core processors with cache systems. The performance of NPDP on heterogeneous processors or on processors without cache systems has not been studied yet.

```
for (j=0; j<n; j=j+1)
  for (i=j-1; i>-1; i=i-1)
    for (k=i; k<j; k=k+1)
      d[i][j]=min(d[i][j], d[i][k]+d[k][j])
```

Figure 1. Original flowchart of NPDP

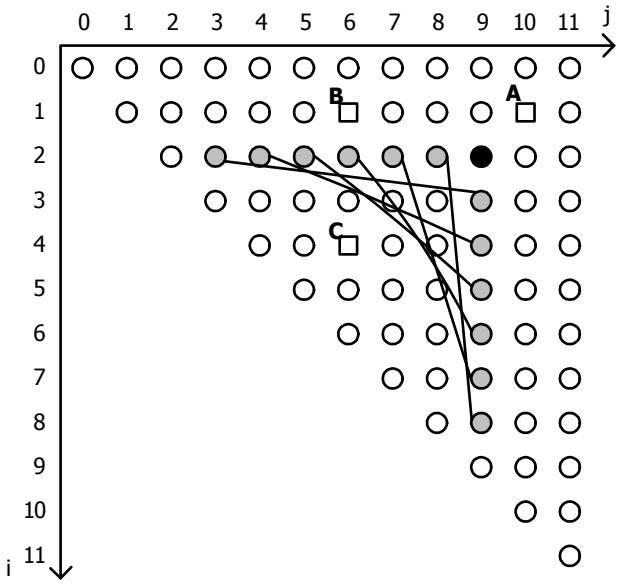


Figure 2. An example of NPDP

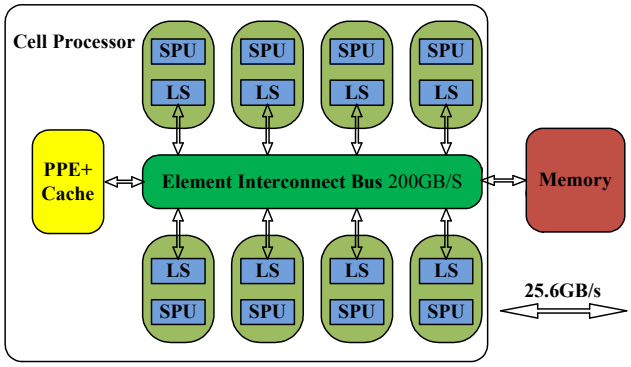
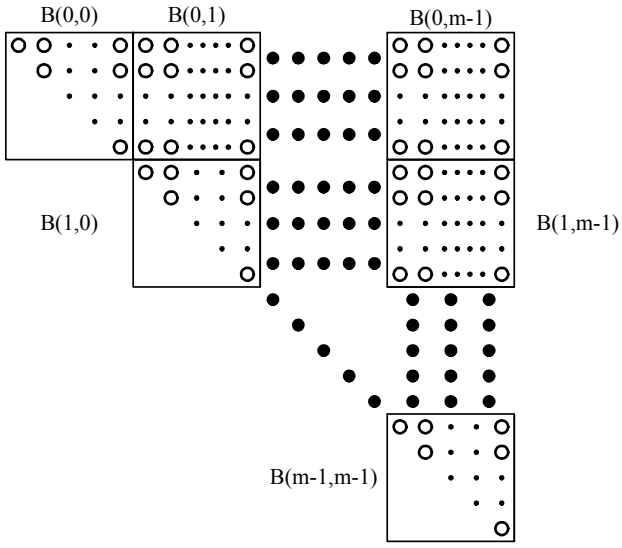


Figure 3. Architecture of the Cell processor

C. *Cell Processor*

The Cell processor is a well-known heterogeneous multi-core processor with SIMD capability. It can execute 200G 32-bit operations per second. Many recent studies [3, 5, 8, 12, 16, 21, 27] show that the Cell processor is effective to improve many algorithms and applications.



4(a): The data of NPDP after tiling. The rows in each block are not adjacent to each other in memory space.

```
for (j=0; j<m; j=j+1)
  for (i=j; i>-1; i=i-1)
    for (k=i+1; k<j-1; k=k+1)
      Compute block(i,j) with block(i,k) and block(k,j)
    Compute block(i,j) with block(i,i) and block(j,j)
```

4(b): Flowchart of NPDP after tiling

Figure 4. The tiling approach proposed by the previous works

As shown in Figure 3, the Cell processor consists of one general-purpose PowerPC processing element called PPE and eight special-purpose synergistic processor elements called SPEs. The PPE runs the operating system and provides the SPE threads control. The SPEs [14] provide the bulk of the application performance. Each SPE is a 128-bit processor, where each instruction can execute four 32-bit operations or two 64-bit operations simultaneously. It has 128 128-bit wide registers and two instruction pipelines capable of different instruction types. When the instruction types do not match, the two instructions aligned in a fetch group cannot be dual-issued. Instead of cache systems, each SPE has its own 256KB local store which holds both instructions and data. The load and store instructions can only access the data in the local store. The transfers between the local store and main memory, as well as the transfers between different local stores, are performed through asynchronous DMA (direct memory access) commands. The peak memory bandwidth is up to

25.6GB/s.

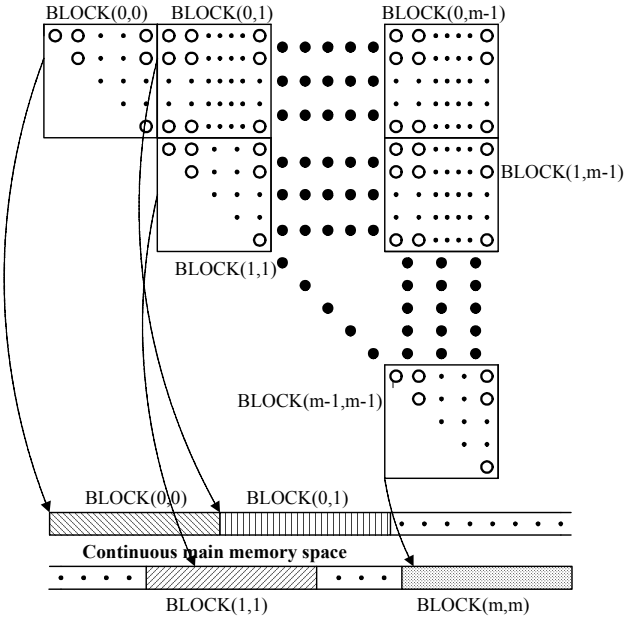
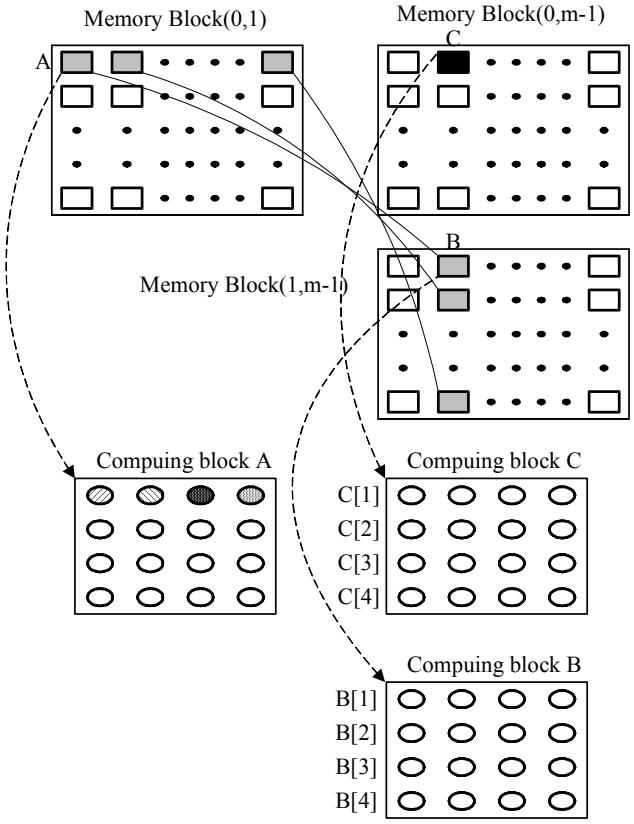
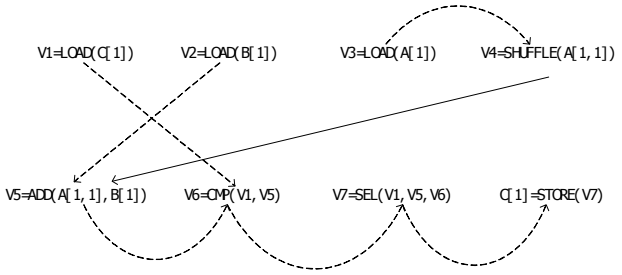


Figure 5. The new data layout of NPDP. Each memory block is a square block whose data are stored sequentially in memory space. Triangular block can be padded into square block.



6(a): Memory blocks and computing blocks. The size of each data is 32-bit.



6(b): SIMD instructions of one step for a computing block and the corresponding dependence graph

Figure 6. Computing blocks and SIMD implementation.

III. DATA LAYOUT

In order to efficiently utilize the Cell processor, we first investigate the data layout. As the logical structure of NPDP is triangular, almost all previous works [7, 24, 25, 26] use a row-major triangular matrix to store the data, as shown in Figure 2. Given the original flowchart in Figure 1, we find two problems of this data layout. First, given the innermost loop, 'd[k][j]' corresponds to discrete memory accesses with non-uniform address intervals because the row-major triangular matrix has non-uniform row sizes. This kind of accesses can be viewed as poor spatial data locality. Second, when the data of NPDP are large, the temporal data locality will be poor. For example, each iteration of the outermost loop computes a column of the data. To compute the j^{th} column, all data on the left of the j^{th} column will be used. These data will be reused for computing the $(j+1)^{th}$ column. However, when these data are too large to fit into the cache or local store, they have to be fetched from main memory when reusing them. In the previous works, Tan et al. [24, 25, 26] and Chowdhury et al. [7] have demonstrated that the tiling approach can dramatically improve the cache reuse as well as the performance of NPDP.

To efficiently utilize the local stores on the Cell processor, we propose to leverage the tiling approach. However, as this approach does not change the data layout in memory, it cannot efficiently utilize the memory bandwidth. Figure 4 shows an example of the tiling approach. When computing the block (0,*m*-1), according to the flowchart in Figure 4(b), we use the

blocks (0,1) and (1,*m*-1) in the first step, and then use the blocks (0,2) and (2,*m*-1) in the next step. In one step, the blocks used in the next step can be prefetched into the local store. As the rows in each block are not adjacent to each other in memory, we have to use a number of DMA commands to prefetch each row into the local store. We note that the efficiency of DMA depends on the size of each DMA transfer. To maximize the size of each DMA transfer, we design a new data layout for NPDP, where each block of data is stored sequentially in memory as shown in Figure 5. With this layout we can achieve the optimal memory performance.

The remaining problem is how to determine the shape and the size of each block. We do not use rectangular blocks but only use square blocks, because the reuse of the local store is determined by the smaller one of the block length and width. We determine the block size according to the size of the local store. There should be at least six buffers in the local store, where three buffers are used in the current step and the other buffers are used to prefetch the blocks for the next step. Therefore, the block size should not exceed 1/6 of the local store size. Note that, the local store also holds instructions.

As the new data layout focuses on efficiently utilizing the on-chip memory system, in what follows we refer to blocks in the new data layout as memory blocks.

IV. CELLNPDP ALGORITHM

In this section, we present our *CellNPDP* algorithm. According to the architecture of the Cell processor, we devise the algorithm with two tiers. The first tier is a SPE procedure which efficiently computes a memory block on a SPE. The second tier is a parallel procedure which enables all SPEs to efficiently compute all memory blocks.

A. SPE Procedure

To maximize the reuse of the local store, in *CellNPDP*, one SPE only computes a memory block each time. According to Section II-A, a memory block depends on the blocks on its left side and below it. For example, the memory block (0,*m*-1) in Figure 4(a) directly depends on the memory blocks on the 0^{th} row and $(m\text{-}1)^{th}$ column. Besides, as a memory block contains multiple data, there are dependences between these data. We call these kinds of dependences *inner* dependences of the block. According to Figure 1, there are strong dependences to compute a data, which results in low ILP (instruction-level parallelism) and low DLP (data-level parallelism). To improve the ILP and DLP, we should compute several independent data simultaneously. The challenge is the inner dependences. To minimize the impact of the inner dependences on the ILP and DLP, we propose to divide each memory block into a number of much smaller square blocks named *computing blocks*, as shown in Figure 6(a). As each register on the SPEs is 128-bit wide, we design the computing block with four 128-bit rows (given that the data size is 32-bit). Each 128-bit row is used to fully utilize the 128-bit wide SIMD capability and the four rows are used to fully utilize the instruction pipelines. The computing blocks still have the problem of inner dependences because a block also contains multiple data. However, the computing power wasted due to the inner dependences is trivial because the computing blocks are small. Here we'd like to point out that there are triangular computing blocks because the logical structure of NPDP is triangular. We can pad them into square blocks. The overhead of padding is also trivial.

Next we study how to compute all computing blocks in a memory block according to the various data dependences. First, each computing block depends on the computing blocks in the dependent memory blocks. For example, the black block *C* in Figure 6(a) depends on the gray blocks. Second, there are dependences between the computing blocks in the same memory block and each computing block has inner dependences. Therefore, we use two stages to compute a memory block. The first stage does not consider the inner dependences of the memory block. The second stage computes the computing blocks one by one, where the blocks on the left side and closer to the bottom are computed earlier. For each computing block, we first compute it with the dependent computing blocks in the same memory block and next use the original flowchart in Figure 1 to process its inner dependences.

Now we study the SIMD implementation. We first introduce the SIMD instructions used. According to Figure 1, there are three kinds of operations: memory, minimum and add. The memory instructions used

include load, store and shuffle. They are used for the data transfer between the local store and the registers. As the SPEs do not have the minimum instruction, we first use the compare instruction to mark the minimum values in the two registers and then use the select instruction to pick out the minimum values. With regard to the add operation, we use the add instruction. Here we'd like to point out that these SIMD instructions are not unique to the Cell SPEs [28]. VMX [29] and SSE [30] instruction sets also provide similar instructions. Next we design the SIMD procedure. Let's study an example. As shown in Figure 6(a), we want to compute the computing block *C* with blocks *A* and *B*, where each block is a 4*4 matrix and each data is 32–bit wide. As each row is 128-bit wide, we compute the four values in each row of *C* simultaneously. Let's investigate the first row *C*[1] of *C*. It depends on the first row *A*[1] of *A* and the entire matrix of *B*. There are four steps to compute *C*[1], where each step uses one value in *A*[1] and the corresponding row in *B*. For example, the first step is formulated as *C*[1]=min(*C*[1], <*A*[1][1],*A*[1][1], *A*[1][1],*A*[1][1]> + *B*[1]), where *A*[1][1] means the first value in *A*[1], and <*A*[1][1], *A*[1][1],*A*[1][1],*A*[1][1]> means a four-value vector. The SIMD procedure of this step is as follows:

1. Load C_1 into a register V_1: V_1=load(C_1)
2. Load B_1 into a register V_2: V_2=load(B_1)
3. Load A_1 into a register V_3: V_3=load(A_1)
4. Set all values in a register V_4 to A_{11}: V_4=shuffle(V_3, mask)
5. Add V_2 and V_4 to a register V_5: V_5=add(V_2,V_4)
6. Compare V_1 and V_5 to a register V_6: V_6=compare(V_1, V_5)
7. Select minimum values from V_1 and V_5, according to V_6: V_7=select(V_1,V_5,V_6)
8. Write V_7 into C_1: Store(V_7, C_1)

TABLE I
CHARACTERIZATION OF THE SIMD INSTRUCTIONS USED FOR COMPUTING A COMPUTING BLOCK WITH TWO BLOCKS. DATA TYPE IS SINGLE-PRECISION FLOATING-POINT.

Instruction	Execution number	Latency (cycles)	Pipeline type
Load	12	6	1
Shuffle	16	4	1
Add	16	6	0
Compare	16	2	0
Select	16	2	0
Store	4	6	1

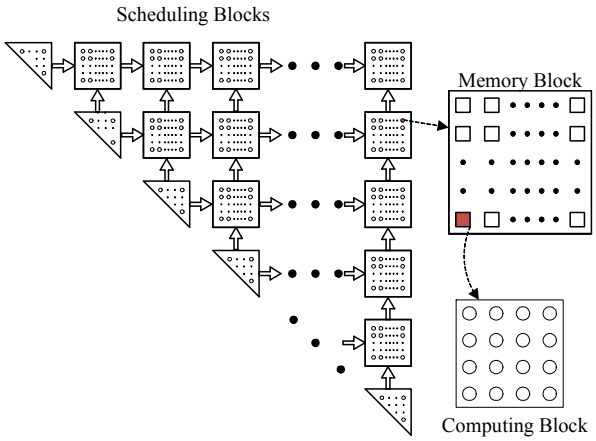


Figure 7. Scheduling blocks and task dependence graph

Subroutine: PPEprocedure
(1) Initialize the task dependence graph and task queue
(2) While there are unfinished tasks
(3) Assign the ready tasks in the task queue to SPEs
(4) Receive finished tasks from SPEs
(5) Notify the tasks according to the finished tasks and then insert the ready tasks into the task queue

Subroutine: SPEprocedure
(6) While there are unfinished tasks
(7) Fetch a ready task *T* from the PPE
(8) For each memory block *MB* in the scheduling block corresponding to *T*
(9) Compute *MB* without considering the inner dependences
(10) For each computing block *CB* in *MB*
(11) Compute *CB* according to the other computing blocks in *MB*
(12) Process the inner dependences of *CB* with the original code in Figure 1
(13) Send *T* to the PPE

Figure 8. Flowchart of CellNPDP

In sum, there are 16 steps to compute the block *C* with the blocks *A* and *B*, where 16*8=128 SIMD instructions are executed. We find that we can utilize the registers to reduce the SIMD instructions. During the 16 steps, there are multiple times of data transfer between the registers and the blocks *A*, *B* and *C*. After buffering these blocks in 12 registers, we can save 48

memory instructions. At last, only 80 SIMD instructions are executed.

Finally, we study how to explore the ILP. Table 1 characterizes the 80 SIMD instructions for a computing block with two blocks. As there are independent instructions in Figure 6(b) and the procedure of computing each row of a computing block is independent, we can hide the high instruction latency through mixing the execution of the 16 steps. However, we cannot achieve the ideal pipeline utilization because the pipeline type restriction results in more instructions executed by the pipeline 0, as shown in Table 1. We also note a problem that the first instruction executed by the pipeline 0 must be 10 cycles later than that of the pipeline 1, because of the data dependence and instruction latency. To fully utilize the pipelines, we develop an approach of software pipelining to hide the 10-cycle latency. At last, it takes only 54 cycles to execute the 80 SIMD instructions.

B. *Parallel Procedure*

To efficiently utilize all SPEs, we use a task queue model which dynamically schedules the computation of all memory blocks among SPEs. As introduced in Section IV-A, each SPE computes a memory block each time. Thus each task contains at least one memory block. When the problem size of NPDP gets bigger or the local stores get smaller (there may be other multi-core or many-core processors with smaller local stores in future), there will be more memory blocks. To reduce the overhead of task scheduling, we use the scheduling blocks each of which is a square of memory blocks. When executing a task on a SPE, the SPE computes the memory blocks in the corresponding scheduling block one by one, where the memory blocks on the left side and closer to the bottom are computed earlier.

As there are dependences between the memory blocks, there are also dependences between the scheduling blocks. Therefore, the task queue model should guarantee that the task of a scheduling block is not scheduled until the computation of all dependent scheduling blocks is finished. To further reduce the overhead of task scheduling, we build a simplified dependence graph, where a task depends on at most two tasks: the nearest task on its left side and blow it respectively. For example, Figure 7 shows the task dependence graph corresponding to Figure 4(a). In the parallel execution, when a task is finished, the two tasks depending on it will be notified. When a task has been notified twice, it becomes ready for scheduling and will be inserted into the task queue.

TABLE II
PERFORMANCE ON THE IBM QS20 CELL BLADE. TIME SECONDS

Problem size			***4,096***	***8,192***	***16,384***
Single-precision	original algorithm	one PPE	715	21961	187,945
		one SPE	3,061	24,588	198,432
	CellNPDP (16 SPEs)		0.22	1.77	13.90
Double-precision	original algorithm	one PPE	1015	27821	241,759
		one SPE	5,096	40,752	327,276
	CellNPDP (16 SPEs)		4.41	34.54	389.15

TABLE III
PERFORMANCE ON THE 8-CORE CPU PLATFORM. TIME SECONDS

Problem size		***4,096***	***8,192***	***16,384***
Single-precision	original algorithm	108.01	1041.1	11021
	CellNPDP (8 cores)	0.43	3.25	25.56
Double-precision	original algorithm	119.79	1234.3	13624
	CellNPDP (8 cores)	0.8159	6.185	48.170

C. *Putting It All Together: CellNPDP Algorithm*

Incorporating the two procedures, we construct *CellNPDP*, an efficient NPDP algorithm on the Cell processor. Figure 8 shows its flowchart. The subroutine PPEprocedure is executed by the PPE. It manages the task queue. The subroutine SPEprocedure is executed by each SPE. It executes the ready tasks one by one. There are three levels of blocking implemented. The first level is the scheduling block which reduces the overhead of task scheduling. The second level is the memory block which efficiently utilizes the on-chip memory system. Note that we have used asynchronous DMA commands in Steps from 8 through 12. The third level is the computing block which efficiently utilizes the instruction pipelines and SIMD capability. Note that, the steps 9 and 11 have been accelerated using SIMD instructions.

V. PERFORMANCE MODELING

In this section, we will answer the following two questions through performance modeling:

1. Which architecture features limit the efficiency of *CellNPDP*?
2. Does the efficiency of *CellNPDP* depend on the problem size of NPDP?

Due to the efficiency of the task queue model, *CellNPDP* can keep load balance and low overhead in

parallel execution. Thus the parallel performance of *CellNPDP* can be viewed as the ideal. As the size of the computing blocks is small, the overhead due to the inner dependences can be neglected. Therefore, we can use the fully optimized memory performance and computing performance to estimate the performance of *CellNPDP*.

The memory performance depends on the local store size and the memory bandwidth. As presented in Section II, the memory block should not exceed 1/6 of the local store. Given the local store size L_S, the maximum side length of the memory blocks is $N_2 = \sqrt{\frac{L_S}{6 * S}}$, where S is the size of each data. Given the problem size N_1, each row has at most $\frac{N_1}{N_2}$ memory blocks. Given a memory block with index (j,i), it depends on (j-i)*2 memory blocks. To compute this memory block, we need to fetch (j-i)*2 memory blocks into the local store. In sum, the total number of the memory blocks fetched into the local stores is $2\sum_{j=1}^{\frac{N_1}{N_2}}\sum_{i=1}^{j}(j-i) \approx \frac{N_1^3}{3 * N_2^3}$, with the total size $\frac{{N_1}^3 * S}{3 * N_2}$. On the other hand, the write into main memory can be neglected because each memory block is written into main memory only once. Therefore, the memory time is $T_M \approx \frac{{N_1}^3 * S}{3 * B * \sqrt{\frac{L_S}{6 * S}}}$, where B is the memory bandwidth.

Next we study the computing time. Given that each computing block is an N_3*N_3 square, each row has at most $\frac{N_1}{N_3}$ computing blocks. In sum, there are about $\frac{N_1^3}{6 * N_3^3}$ times to compute all computing blocks. Each time, we use two computing blocks to compute a computing block. Therefore, the overall computing time is $T_C \approx \frac{N_1^3 * C_C}{N_3^3 * f * C_N}$, when C_C is the number of cycles to compute a computing block each time, f is the frequency of the processor and C_N is the number of cores.

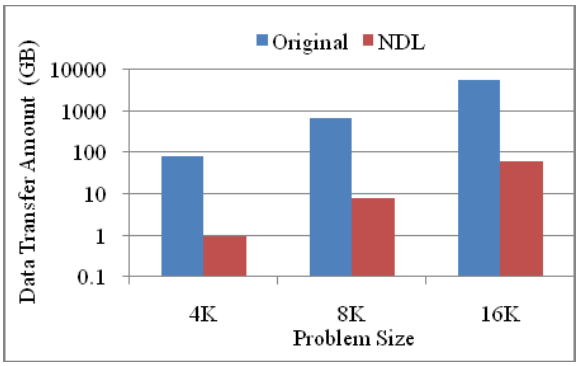
The execution time of *CellNPDP* is $T_{All} = \max(T_M, T_C)$. To fully utilize the computing power, we should guarantee $T_M \le T_C$. In this case we derive the constraint $B * \sqrt{L_S} \ge \frac{\sqrt{6} * {N_3}^3}{3} * \frac{S^{3/2} * f * C_N}{C_C}$. This formula indicates that the efficiency of *CellNPDP* depends on the memory system and is more sensitive to the memory bandwidth. The processor utilization of *CellNPDP* is $U_{All} = \frac{T_C * f * U_C}{T_{All} * f} = U_C * \min(1, \frac{T_C}{T_M})$, where U_C means the processor utilization of computing one computing block with two computing blocks. As both T_M and T_C have the factor of ${N_1}^3$, $\frac{T_C}{T_M}$ is independent to N_1. Therefore, the efficiency of *CellNPDP* is independent of the problem size of NPDP. To the best of our knowledge, this is the first work revealing that the efficiency of NPDP can be independent of the problem size on modern multi-core processors.

VI. Experimental Evaluation

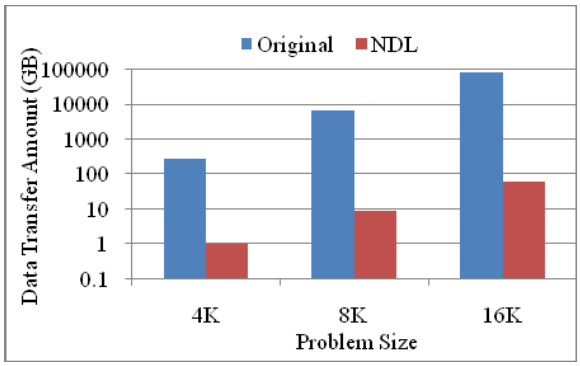
To empirically evaluate *CellNPDP*, we use two platforms: an IBM QS20 dual-Cell blade and a CPU platform with two quad-core Nehalem processors. On the two platforms, the processor utilization of *CellNPDP* is larger than 60%, which demonstrates the high efficiency of *CellNPDP* on modern multi-core processors. Tables 2 and 3 show the performance of the original NPDP algorithm in Figure 1 and *CellNPDP* on the two platforms. To the best of our knowledge, there is no available Cell implementation of NPDP. Therefore, we only use the CPU platform to compare *CellNPDP* to the state-of-the-art algorithm proposed by Tan et al. [26] (denoted as *TanNPDP*). The code of the algorithm is provided by the authors. It is a fully optimized implementation with optimizations including tiling, helper threading and parallelization. In the following context, we first evaluate the impact of each kind of optimizations, including the new data layout (denoted as NDL), SPE procedure (denoted as SPEP) and parallel procedure (denoted as PARP) on the two platforms. Next we compare *CellNPDP* to *TanNPDP* on the CPU platform. Finally, we evaluate the performance of *CellNPDP* given that the local stores get smaller.

A. Performance Anatomy on the Cell Processor

On the Cell processor, we set the memory block size to 32KB which is smaller than 1/6 of the local store size, because the local stores also hold instructions. In

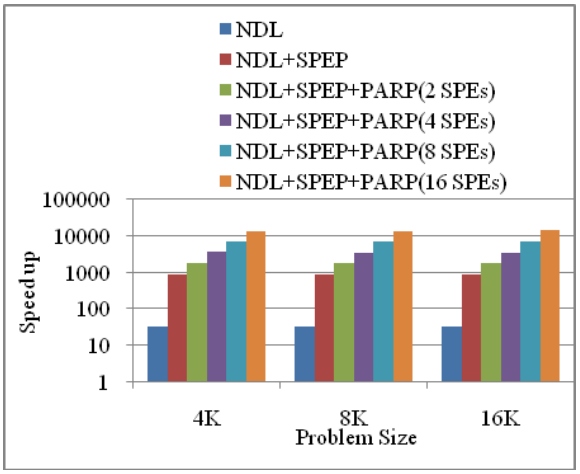


9(a): Data transfer amount on the IBM QS20 Cell blade

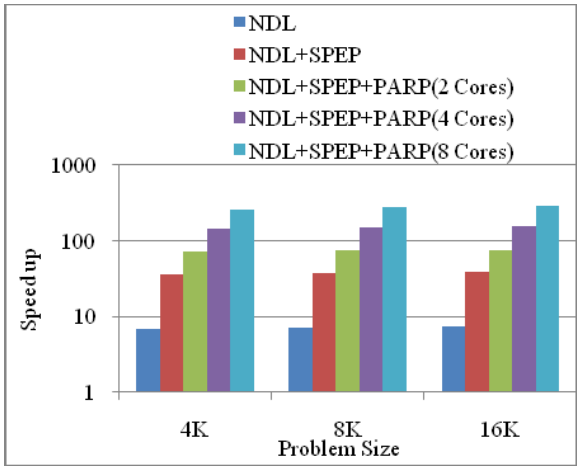


9(b): Data transfer amount on 8-core CPU platform

Figure 9. The amount of data transfer between processors and main memory. The data type is single-precision floating-point



10(a): Speedup on the IBM QS20 Cell blade



10(b): Speedup on the 8-core CPU platform

Figure 10. Performance speedup with the single-precision floating-point data

the following context, we first evaluate the performance with the single-precision floating-point (SPFP) data in Sections from VI-A. The baseline here is the original algorithm on one SPE, where each DMA command prefetches multiple data in one row or a data in one column.

1) Impact of the new data layout: As the new data layout can improve the data reuse, it significantly reduces the data transfer between the processor and main memory, as shown in Figure 9(a). Besides, it improves the efficiency of DMA transfers. As a result, it significantly improves the performance, as shown in Figure 10(a). On average, there is a 31.6-fold speedup.

2) Impact of the SPE procedure: As shown in Figure 10(a), the SPE procedure further provides a 28-fold speedup on average. Although one SIMD instruction can only execute 4 SPFP operations, the speedup achieved is much higher. This is because the SPE procedure not only efficiently utilizes the SIMD instructions but also significantly improves the ILP. Moreover, the use of the computing blocks also reduces the number of loop iterations and consequently reduces the overhead of branch instructions.

3) Impact of the parallel procedure: As shown in Figure 10(a), the parallel procedure achieves a good scaling performance with the number of SPEs. When using 16 SPEs, there is a 15.7-fold speedup on average. This result demonstrates the high efficiency of our task queue model.

4) Processor utilization: Next we evaluate the processor utilization of *CellNPDP*. When using 16 SPEs, *CellNPDP* can execute 80 scalar instructions per cycle (A useful 32-bit operation is counted as a scalar instruction. The redundant operations for padding are neglected). As the Cell blade can execute 128 scalar instructions per cycle, the processor utilization is up to 62.5%.

5) Performance with double-precision floating-point data: Now we evaluate the performance with the double-precision floating-point (DPFP) data. As shown in Figure 11(a), *CellNPDP* also significantly improves the performance of NPDP. We note that the performance of *CellNPDP* in Figure 11(a) is much worse than that in Figure 10(a). This is because of three facts. First, one SIMD instruction can execute only two DPFP operations simultaneously. Second, the latency of the DPFP instructions is 13 cycles, which is much bigger than the latency of the SPFP instructions. Third, the DPFP instructions have 6 cycles of stall, which means there are at least 6 cycles between a DPFP instruction and the successive instruction on the same pipeline. We also note that *CellNPDP* achieves lower speedup when the problem size is 16K. This is because the 1GB main memory on the Cell blade cannot hold the operating system and the 1GB data at the same time.

B. Performance Anatomy on the CPU Platform

On the CPU, we also set the memory block size to 32KB because the results in Section VI-A show that this size can achieve good enough performance. In the SPE procedure, there are also 80 SIMD instructions to compute a computing block with two computing blocks. We do not implement the software pipelining approach in the SPE procedure because the Nehalem processor does not have the pipeline type restriction. In the parallel procedure, all cores cooperatively manage the task queue.

1) Impact of the new data layout: As shown in Figures 9(b) and 10(b), the new data layout also significantly reduces the data transfer between the CPU and main memory and consequently significantly improves the performance. On average, there is a 7.14-fold performance speedup. Comparing Figures 9(a) and 9(b), we can see that the amount of original data transfer is higher on the CPU platform. This is because the size of each data transfer on the CPU platform is larger (64 bytes, a cache line). In contrast, the original algorithm can achieve much higher memory bandwidth on the CPU platform. As a result, the original performance is better and the speedup of the new data layout is lower on the CPU platform, as shown in Tables 2 and 3 and Figure 10.

2) Impact of the SPE procedure: As shown in Figure 10(b), the SPE procedure further provides a 5.28-fold speedup on average. This speedup is much smaller than the speedup on the Cell processor because the Nehalem processor does not have pipeline type restriction and the out-of-order superscalar architecture can dynamically hide the latency of the SPFP instructions.

3) Impact *of the parallel procedure:* As shown in Figure 10(b), the parallel procedure also achieves a good parallel performance on the CPU. When using 8 cores, there is a 7.22-fold speedup on average.

4) Processor utilization: Different from SPEs, Nehalem is not a SIMD processor. Each core on Nehalem is a 4-issue out-of-order superscalar core which can execute four 128-bit operations each cycle in the ideal case. In other word, the CPU platform can execute 128 32-bit operations per cycle.

Each instruction on Nehalem consists of several operations. For example, when computing a computing block with two computing blocks, the 80 SIMD instructions will be translated into 160 operations, where 64 operations compute the addresses of memory operations and the remaining are 128-bit operations. When using 8 cores, *CellNPDP* can execute 78.8 32-bit operations per cycle. As a result, the processor utilization is up to 61.6%. This result demonstrates that *CellNPDP* is also highly efficient on the homogeneous multi-core CPU architecture.

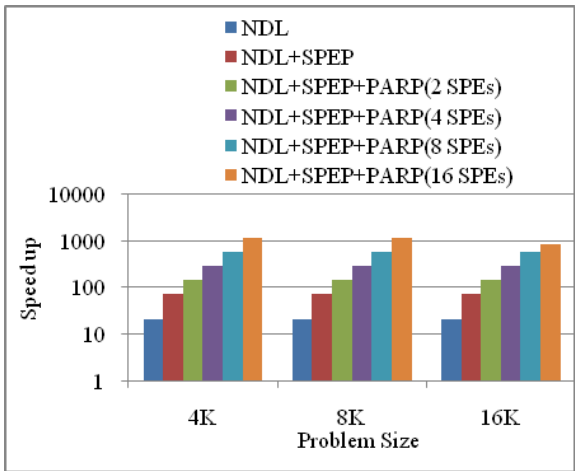
5) Performance with double-precision floating-point data: Figure 11(b) shows that *CellNPDP* also significantly improves the performance of NPDP when the data are DPFP. We note that this performance is much better than the corresponding performance on the Cell blade because the double-precision instructions on Nehalem do not have the cycles of stall.

C. Comparison with the State-of-the-art fully Optimized Algorithm

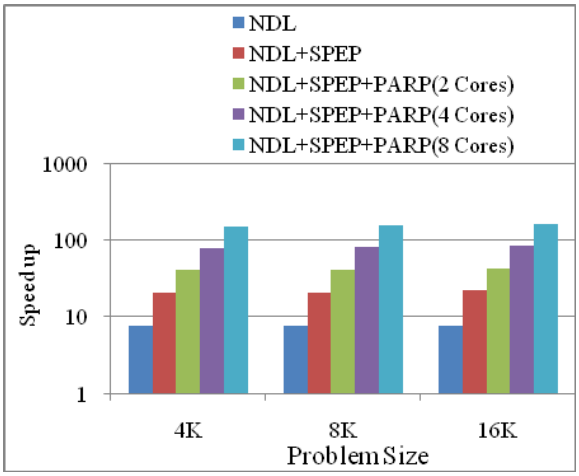
Figure 12 shows the performance of *CellNPDP* and *TanNPDP* on the CPU platform when 8 cores are used. On average, *CellNPDP* is 44-fold faster for single-precision and 28-fold faster for double-precision. This result also indicates that the processor utilization of *TanNPDP* is less than 4%, which largely underutilizes modern multi-core processors. Although *TanNPDP* is an implementation optimized with tiling, helper threading and parallelization, *CellNPDP* still significantly outperforms it. This is because *CellNPDP* fully exploits ILP, DLP and TLP (thread-level parallelism) at the same time.

D. Performance with Smaller Local Store

In the end, we'd like to evaluate the impact of the local store size on the performance of *CellNPDP*. As the maximum size of memory blocks is linear to the

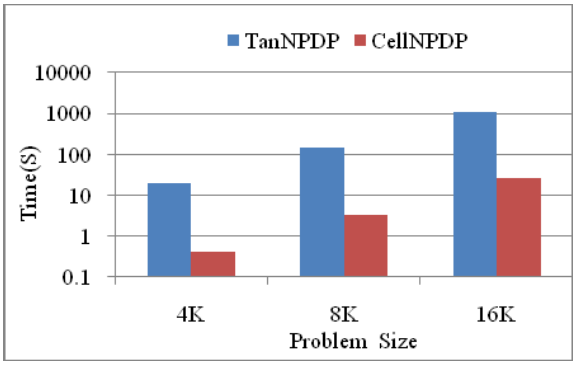


11(a): Speedup on the IBM QS20 Cell blade

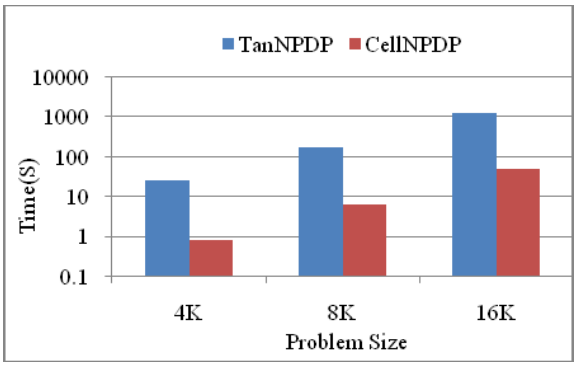


11(b): Speedup on the 8-core CPU platform

Figure 11. Performance speedup with the double-precision floating-point data



12(a): Execution time with single-precision floating-point data



12(b): Execution time with double-precision floating-point data

Figure 12. Performance of CellNPDP and TanNPDP (the state-of-the-art fully optimized algorithm) on the 8-core CPU platform. 8 cores are used.

local store size, we evaluate this impact through varying the memory block size. From Figure 13, we can find that the performance of *CellNPDP* gets poorer when the memory block size gets smaller. This is because the smaller size leads to lower efficiency of DMA transfers and more data transferred. Besides, the smaller size leads to lower efficiency of the software pipelining approach implemented in the SPE procedure, which results in poorer ILP.

VII. Conclusion

In this paper, we study the NPDP problem on modern multi-core processors. Although the previous works have achieved a significant performance improvement, they still largely underutilize modern processors. To make NPDP really efficient on modern processors, we design the new data layout and devise the two-tiered *CellNPDP* algorithm. The experimental results show that *CellNPDP* significantly improves the performance of NPDP and can achieve the processor utilization larger than 60%. The experiences of this paper demonstrate that, through carefully designing the data layout and the algorithm implementation, NPDP can efficiently utilize the instruction pipelines, SIMD capability and multiple cores on modern processors.

Acknowledgement

We would like to thank H. Peter Hofstee for improving this paper and Guangming Tan for providing us the source code of *TanNPDP*.

This work is supported by National High-Tech R&D (863) Program of China (2010AA012301, 2010AA012302), Natural Science Foundation of

China (61040048), and Tsinghua National Laboratory for Information Science and Technology (TNList) Cross-discipline Foundation.

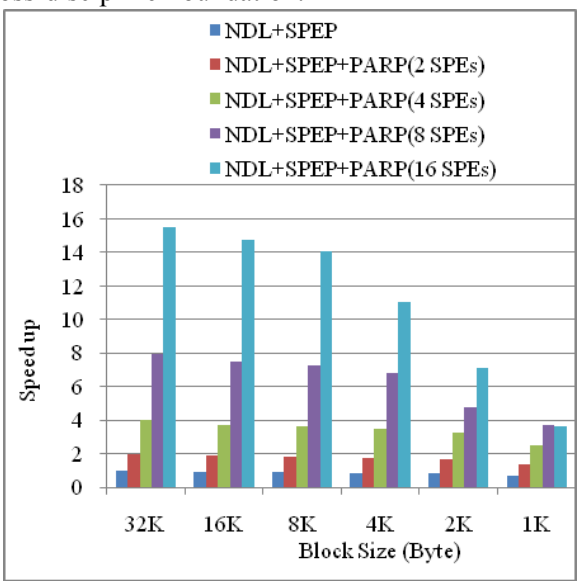


Figure 13. Performance of CellNPDP on the IBM QS20 Cell blade with different sizes of memory blocks and different numbers of SPEs. The baseline is the performance with 32KB memory block size and one SPE. The problem size is 4K. The data are single-precision floating-point.

REFERENCES

[1] F. Almeida, R. Andonov, and D. Gonzalez. Optimal tiling for RNA base pairing problem. In *Proc. SPAA*, 2002.

[2] P.G. Bradford. Efficient parallel dynamic programming. In Proc. Allerton Conf. on Communication, Control and Computing, 1992.

[3] G. Buehrer, S. Parthasarathy, and M. Goyder. Data mining on the Cell Broadband Engine. In Proc. ICS, 2008.

[4] J. H. Chen, S. Y. Le, B.A. Shapiro, and J.V. Maizel. Optimization of an RNA folding algorithm for parallel architectures. Parallel Computing, 1998.

[5] S. Chellappa, F. Franchetti, and Markus Püschel. Computer generation of fast Fourier transforms for the Cell Broadband Engine. In Proc. ICS, 2009.

[6] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine architecture and its first implementation: A performance view. IBM Journal of Research and Development, 2007.

[7] R. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In Proc. SPAA, 2008.

[8] K. Daloukas, C.D. Antonopoulos, and N.Bellas. Implementation of a wide-angle lens distortion correction algorithm on the Cell Broadband Engine, In Proc. ICS, 2009.

[9] P. Edmonds, E. Chu, and A. George. Dynamic programming on a shared memory multiprocessor. Parallel Computing, 1993.

[10] I. H. M. Fekete, and P. Stadler. Prediction of RNA base pairing possibilities for RNA secondary structure. Biopolymers, 1990.

[11] Z. Galil, and K. Park. Parallel algorithm for dynamic programming recurrences with more than O(1) dependency. In Journal of Parallel and Distributed Computing, 1994.

[12] B. Gedik, R. R. Bordawekar, and P. S. Yu. CellSort: high performance sorting on the cell processor. In Proc. VLDB, 2007.

[13] A. Grama, A. Gupta, G. Karypis, and V. Kumar. Introduction to parallel computing. Addison Wesley, 2003.

[14] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. IEEE Micro, 2006.

[15] L. Guibas, H. Kung, and C. Thomson. Direct VLSI implementation of combinatorial algorithms. In Proc. Caltech Conf. VLSI, 1979.

[16] K.Z. Ibrahim, and F. Bodin. Implementing wilson-dirac operator on the cell broadband engine". In Proc. ICS, 2008.

[17] R.B. Lyngso, and M. Zuker. Fast evaluation of internal loops in RNA secondary structure prediction. Bioinformatics, 1999.

[18] B. Louka, and M. Tchuente. Dynamic programming on two-dimensional systolic arrays. Information Processing Letters, 1988.

[19] S. A. Manavski, and G. Valle. CUDA compatible GPU cards as efficient hardware accelerator for smith-waterman sequence alignment. BMC Bioinformatics, 2008.

[20] F. Sanchez, E. Salami, A. Ramirez, and M. Valero. Performance analysis of sequence alignment applications. In Proc. IISWC, 2006.

[21] D. P. Scarpazza, and G. F. Russell. High-performance regular expression scanning on the Cell/B.E. processor. In Proc. ICS, 2009.

[22] B.A. Shapiro, J.C. Wu, D. Bengali, and M.J. Potts. The massively parallel genetic algorithm for RNA folding: MIMD implementation and population variation. Bioinformatics, 2001.

[23] G. Tan, S. Feng, and N. Sun. Load balancing algorithm in cluster-based RNA secondary structure prediction. In Proc. ISPDC 2005.

[24] G. Tan, S. Feng, and N. Sun. Locality and parallelism optimization for dynamic programming algorithm in bioinformatics. In Proc. SC, 2006.

[25] G. Tan, N. Sun, and G. R. Gao. A parallel dynamic programming algorithm on a multi-core architecture. In Proc. SPAA 2007.

[26] G. Tan, N. Sun, and G. R. Gao. Improving performance of dynamic programming via parallelism and locality on multicore architectures. IEEE Transactions on Parallel and Distributed Systems, 2009.

[27] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In Proc. CF, 2006.

[28] IBM Corp. C/C++ language extensions for Cell Broadband Engine architecture V2.5. CBEA JSRE Series, 2008.

[29] IBM Corp. PowerPC microprocessor family: Vector/SIMD multimedia extension technology programming environments manual.

[30] Intel Corp. IA-32 Intel architecture software developer's manual.