A Compilation Framework for Irregular Memory Accesses on the Cell Broadband Engine

Pramod K. Bhatotia
High Performance Computing Group,
IBM India Research Laboratory,
New Delhi-110070, INDIA
Email:{*pbhatoti*}@in.ibm.com

Sanjeev K. Aggarwal and Mainak Chaudhuri
Department of Computer Science and Engineering,
Indian Institute of Technology,
Kanpur-208016, INDIA
Email:{*ska and mainakc*}@cse.iitk.ac.in

Abstract

A class of scientific problems represents a physical system in the form of sparse and irregular kernels. Parallelizing scientific applications that comprise of sparse data structures on the Cell Broadband Engine (Cell BE) is a challenging problem as the memory access pattern is irregular and cannot be determined at compile time. In this paper we present a compiler framework for the Cell BE that provides automatic run-time support for memory communication and parallelization to indirectly indexed applications. The memory communication scheme generates DMA communication schedule after performing data flow analysis and facilitates the gather and scatter operations. The run-time parallelization technique judiciously partitions the data and computational work taking into account any coherence issues that may arise due to irregular accesses. We evaluate the performance of our compiled code on a 3.2 *GHz Cell processor and demonstrate a parallel speedup of up to factor of* 4.7 *when using eight threads.*

1. Introduction and Motivation

Irregular memory access kernels arise in diverse scientific applications like molecular dynamics, computational fluid dynamics (CFD) solvers, n-body solvers, etc. These kernels frequently become computational bottlenecks requiring a tremendous amounts of computational power. Hence, it is important to develop efficient parallel codes for applications in which memory accesses are made through levels of indirection. The Cell BE is an attractive platform for carrying out parallel computation on a single chip with very high peak GFLOPS [12]. The high performance computing power of the Cell BE can be utilized to accelerate an application if and only if the application is effectively parallelized with proper memory communication between the Synergistic Processor Elements (SPEs) and the Power Processing Element (PPE). To take advantage of the coarse-grain parallelism available in the form of multiple SPEs, a parallelizing compiler needs to generate Single-Program-Multiple-Data (SPMD) parallel model of execution. The Cell BE performs extremely well for the applications where the memory access pattern is regular. As the access pattern can be predicted at compile time, the computation can be overlapped with communication, which yields better performance. However, parallelizing irregular accesses automatically and efficiently on the Cell BE [2] is a particularly challenging problem for the following reasons.

- **Compiler Analysis for Irregular Memory Accesses:** A memory access is irregular if no closed-form expression, in terms of the loop indices and constants, for the subscripts of the accessed variable is available at compile-time. The access patterns in irregular kernel would be known at run-time only. This results in lack of compile-time knowledge about where the DMA communication schedule for gather and scatter operation is to be placed. Hence, traditional static analysis and loop transformation techniques cannot be used for irregular memory accesses. When static analysis cannot produce the information needed, run-time techniques must be employed. A dataflow analysis framework based on run-time preprocessing of the kernel is needed.

- **Run-time Parallelization of Irregular Reduction Loops (Sparse Updates):** Parallelization of loops with irregular reads and writes leads to loop carried dependence that cannot be estimated at compile-time. As a result, loops with sparse updates may end up producing wrong results because the local stores (LS) of the SPEs are not kept coherent by the hardware. Hence, run-time analysis is required to determine the cross-iteration dependency for loops with sparse updates.

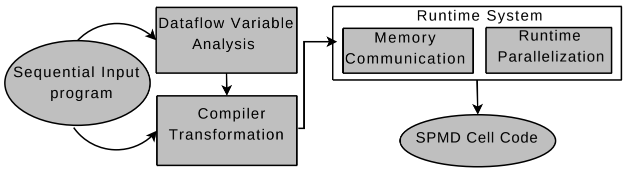

Figure 1. Overview of the compiler framework

The run-time parallelization system must provide support for automatic partitioning of computational work and data by avoiding sparse updates using barrier and other synchronization primitives.

- **Explicit Dynamic Memory Management:** The SPEs can operate only on their small LS memory (256 KB), which is shared for both instruction and data. SPEs cannot access data directly from the main memory. The data must be explicitly transferred for computation from the shared main memory into the LS through DMA operations. And due to the limited amount of LS memory per SPE, fitting all the code and data into the LS can be difficult for most applications. As the SPEs are specially designed to do large amounts of computation quickly, dynamic management of LS is needed. For this, efficient loop tiling is required to minimize the number of bus transfers between the main memory and the LS by overlapping the communication with computation.

- **Challenges Faced in Getting Better Performance from the Cell Architecture:** The SPEs are designed for streaming workload computation while the PPE manages the workload partitioning and monitors the control flow among the SPEs. So it is necessary to partition the computation and the data efficiently to get the maximum performance. Secondly, the Cell BE includes branch prediction in the PPE, but the SPEs do not include dynamic branch prediction hardware. Instead, they rely on the compiler generated branch hints. Hence, control transfer should be minimized for efficient implementation. Lastly, Since the SPEs are vector processors, scalar operations turn out to be costly. This is because an SPE is able to load and store only 16 bytes at a time from the local store locations, which are aligned on the 16-byte boundaries.

In this work, we have developed a parallelizing compilation framework for sparse scientific applications. The overview of the framework is shown in Figure 1. The dataflow framework, after performing flow variable analysis, describes accurately where the direct memory access (DMA) communication schedules are to be generated for

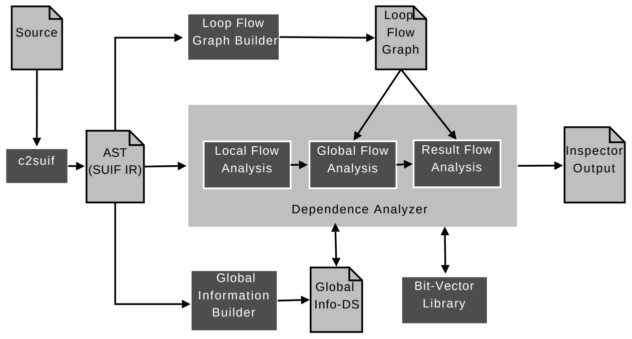

Figure 2. Dataflow analysis framework

gather and scatter operations. Then compiler framework performs transformations to facilitate the data communication and the actual computation. It also exploits situations to reuse communication schedule for amortizing the cost of the dataflow analysis. The run-time system provides support for memory communication and parallelization. The memory communication scheme provides dynamic management of data transfers between the LS and the main memory, thereby avoiding the programmer managed local stores. We have also implemented compiler-directed multi-buffering to overlap on-chip communication and SPE computation. The run-time parallelization technique judiciously partitions the data and computational work taking into account any coherence issues that may arise due to irregular accesses.

The rest of this paper is organized as follows. Section 2 explains the dataflow variable analysis for the Cell BE. Section 3 describes the compiler transformations and code generation phase. Section 4 presents the run-time system for memory communication and parallelization scheme to avoid concurrent sparse updates to the same location. Section 5 provides a brief overview of related work in the area of compilation for irregular problems. Section 6 reports performance results and Section 7 concludes the paper.

2 Dataflow Variable Analysis for the Cell BE

To generate an optimized code for the Cell memory architecture, the compiler has to schedule explicit memory communication between the main memory and the LS. To accomplish this the compiler has to transform the kernel after analyzing: (i) the kernel data access patterns and (ii) the available memory space in the LS. The dataflow analysis framework for the Cell BE analyzes the sequential input program to determine the points in the program for memory communication. Figure 2 shows the block diagram of the dataflow analysis framework for memory communication.

The dataflow framework analyzes [11] the input kernel based on run-time preprocessing to build the memory

```
Algorithm 1: Local Flow Analysis
input  : Set of $LFG$ basic nodes $\omega$, global program
         information of $AP$
output: Local flow variables $(LFV)$

for $l \leftarrow 1$ to $\omega$ do
    GET(l) : The portions read in $l$ from LS.
        $\{p|stmt\, in\, l\, reads\, portion\, p\}$
    PUT(l) : The portions written from $l$ into the LS.
        $\{p|stmt\, in\, l\, assigns\, to\, portion\, p\}$
    TILE(l) : The portions tiled into blocks. These
    portions are large compared to the memory
    available in the LS.
    $\{p|portions\, in\, l\, requires\, memory\, more\, than\, block\, size\, in\, l\}$
    BUF(l) : The portions buffered into LS on exit from
    $l$.
        $\{GET(l) \cup PUT(l)\} \cap \overline{(TILE(l))}$
    KILL(l) : The portions that may be made invalid in
    $l$ by overwriting a part of or the full portions.
        $\{p|stmt\, in\, l\, invalidates\, portion\, p\}$
end
return
```

```
Algorithm 2: Global and Result Flow Analysis
input  : Set of $LFG$ basic nodes $\omega$, Local flow variables
         $(LFV)$, global program information of $AP$
output: Result flow variable

for $l \leftarrow 1$ to $\omega$ do
    Global Flow Variable Analysis:
    LIVE^{any/all}(l): The portions needed in $l$ or along
    any/all paths starting in $l$.
    $GET(l) \cup \bigcap_{s \epsilon successor(l)} (LIVE^{all}(s) \backslash KILL(l))$
    $GET(l) \cup \bigcup_{s \epsilon successor(l)} (LIVE^{any}(s) \backslash KILL(l))$
    BUFFD (l): The portions already available when
    entering $l$.
    $BUF(l) \cup \bigcap_{p \epsilon predecessor(l)} (BUFFD(p) \backslash KILL(l))$
    HOIST (l): The portions for which gather should
    be hoisted ahead of $l$.
    $\bigcap_{p \epsilon predecessor(l)} (LIVE^{all}(p) \cup BUFFD(p))$
    FETCH(l): The portions needed in $l$ or some later
    loop. These can be hoisted before $l$.
    $GET(l) \cup \bigcap_{s \epsilon successor(l)} (HOIST(s) \cap FETCH(s))$

    Result Flow Variable Analysis:
    The result flow variable describes which portions
    have to be gathered before entering $l$.
    $FETCH(l) \backslash \bigcap_{p \epsilon predecessor(l)} (FETCH(p) \cup BUFFD(p))$
end
return
```

communication schedule. During program execution, the framework examines the data references made by processor and calculates which off-processor data needs to be fetched and where this data will be stored once it is received. The framework is implemented on SUIF 2.0 [1] compiler framework, where it analyzes the loop flow graph for communication of array portions.

Definition 1: An $array\ portion\ (AP)\ Y[X(1:n)]$ accesses array Y at subscripts generated from array X, which has lower bound 1 and upper bound n.

Definition 2: A $loop\, flow\, graph\ (LFG)$ of a program P is a control flow graph $(N, E, begin, end)$, where each node $n \epsilon N$ represents $L \cup P$. Here L is the set of loops and P contains one entry pad and one exit pad for each loop. An edge $(n, n')\epsilon E$ represents a possible flow of control from node n to node n'. The two distinguished nodes, $begin$ and end, represent, respectively, the unique entry and exit of the program P.

The dependence analyzer for local flow variable analysis is described in Algorithm 1, global and result flow variable analysis are explained in Algorithm 2.

3 Compiler Transformations and Code Generation

Once the memory communication schedule is built, then compiler transforms the input sequential program to generate Single-Program-Multiple-Data (SPMD) parallel model of execution. Then, the transformed parallelized version performs the actual communication and computation using run-time system. The block diagram of compiler transformations and code generation is shown in Figure 3.

3.1 Kernel Transformations for Double-Buffering Scheme

Since DMA transfers for gather and scatter of data take a lot of time, a great deal of cycles of SPEs gets wasted waiting for the data. In order to speed up the process, the com-

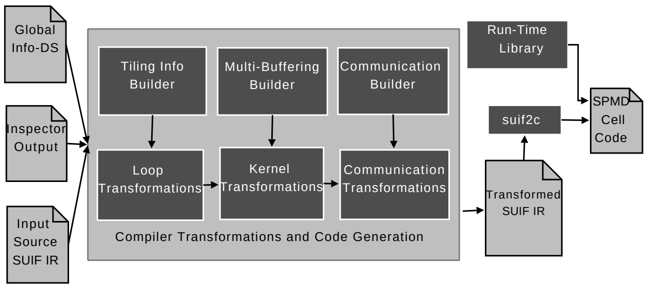

Figure 3. Compiler transformations and code generation

putation at SPE must be overlapped with the DMA communication. The current implementation of our compiler modifies the kernel's array portions wherever they are accessed, for implementing double-buffering technique. This maximizes the time spent in the compute phase and minimizes the time spent waiting for the completion of DMA transfers. It allocates two buffers for every array portion accessed in the kernel. To implement double buffering, we use unique DMA tag IDs for each buffer of an array portion using tag manager function. The tags are grouped based on the array portion, using fence command for ordering within a tag group. To ensure the ordering of DMA transfers within the MFC, barriers are implemented.

3.2 Loop Transformations for Tiling

The scientific kernels are mostly abstracted as a series of multilevel nested loops that access multi-dimensional data arrays. There is always a possibility that the data arrays needed to execute the kernel are too large to fit in the LS memory. Loop tiling is a key compilation transformation to accommodate both code and data within the limited size of the LS memory. The tiling of the loop must ensure that at all the times the data required must fit in the LS. It must also minimize the number of data transfers between the main memory and the LS thereby exploiting reuse of the data residing in the LS memory.

3.3 Communication Schedule Reuse

In iterative kernels, the access patterns repeat. Hence, the same communication schedule can be reused, provided there is no possibility of the referenced arrays having been modified. The compiler stores the communication schedule computed for the first iteration, and reuses it for subsequent iterations. To amortize the cost of the flow analysis phase used for determining the communication requirement, the current implementation of the compiler performs compile time analysis to reuse the communication schedule [7]. Algorithm 3 iterates over the loop flow graph to analyzes the schedule reuse. A data array descriptor (DAD) stores the current distribution of the array and its size. Block distribution of data arrays is implemented in our scheme. The information produced by the dataflow analysis for loop L can be reused only if the following conditions are met since the last invocation.

Algorithm 3: Communication Schedule Reuse

```
input : A $forall$ loop that contains $m$ data arrays
        $x^i_L$,$1 \le i \le m$, and $n$ indirection arrays, $ind^j_L$,
        $1 \le j \le n$.
Define:
1. $DAD(Z^p_L)$   Data access descriptor associated with
$Z^p_L$ where $Z^p_L = \{x^i_L, ind^j_L\}$ .
2. $L.DAD(Z^p_L)$   Recorded data access descriptor
associated with $Z^p_L$ when $L$ carried out its last analysis .
3. $last\_mod(DAD(ind^j_L))$   last modified for
indirection array $DAD(ind^j_L)$.
4. $L.last\_mod(L.DAD(ind^j_L))$   Recorded last
modified for indirection array $DAD(ind^j_L)$ when $L$
carried out its last analysis.
Record: Each time analysis for $L$ is carried out, store
the following information:
1. $DAD(x^i_L)$ for each unique data array
$x^i_L,\ 1 \le i \le m$
2. $DAD(ind^j_L)$ for each unique indirection array
$ind^j_L, 1 \le j \le n$
3. $last\_mod(DAD(ind^j_L))$, for $ind^j_L, 1 \le j \le n$
Check: The following checks are performed before the
subsequent execution of $L$. If any of the following
conditions is false, the analysis must be repeated for $L$.
1. $DAD(x^i_L) == L.DAD(x^i_L),\ 1 \le i \le m$
2. $DAD(ind^j_L) == L.DAD(ind^j_L),\ 1 \le j \le n$
3. $last\_mod(DAD(ind^j_L)) ==$
$L.last\_mod(L.DAD(ind^j_L)),\ 1 \le j \le n$
return
```

- Distribution of data arrays referenced in the loop L remain unchanged.
- Indirection arrays associated with loop L have not been modified.

The first time dataflow analysis for a forall loop L is carried it performs all the preprocessing. After that checks are performed before the subsequent execution of the L.

4 Compiler Controlled Runtime System

4.1 Compiler-Directed Memory Communication Mechanism

The main memory can be accessed in the following ways: (i) regular block fashion; (ii) irregular manner for

Algorithm 4: Memory communication mechanism for the Cell processor

```
input  : Number of program points $\tau$, gather-set $G$,
         scatter-set $S$, dead-set $D$
output: Data transfer through DMA

Retain-set:: $R[0:\tau]$=$\phi$
Operate-set:: $O[0:\tau]$=$\phi$
for $j \leftarrow 1$ to $\tau$ do
    Determine $G[j]$, $S[j]$, and $D[j]$ using the dataflow
    analysis result obtained for $j$th program point
    $R[j]$ = $R[j-1]$ - ( $S[j]$ + $D[j]$ )
    $O[j]$ = $G[j]$ - $R[j]$
    $\omega$ = length($O[j]$)
    for $i \leftarrow 1$ to $\omega$ do
        if $O[j][i].access\_type$ = regular access then
            Allocate twin buffer and exercise Block
            Access Method for Regular Accesses for
            $O[j][i]$th portion
        else
            Allocate twin buffer and exercise Block
            Access Method for Regular Accesses for
            the indirection portion in $O[j][i]$th portion
            if read-only access then
                Bounded Method for Irregular Read
            else
                Compiler Controlled Cache
            end
        end
    end
    $n$ = length($S[j]$)
    for $i \leftarrow 1$ to $n$ do
        if $S[j][i].access\_type$ = regular access then
            Write back using
            Block Access Method
        else
            Compiler Controlled Cache
        end
    end
end
return
```

read-only operations; and (iii) irregular manner for read as well as write operations. For each type of data access to the main memory, there is a unique way of performing the data communication. In this section, we describe the algorithm for performing the data communication between the SPE LS and the main memory. The data communication is done through direct memory access (DMA) at the program points determined after dataflow analysis. Iterating over each program point, depending on the type of the access, Algorithm 4 implements the memory communication as discussed below.

1. **Block Access Method for Regular Accesses:** To gather the portions that are accessed in a regular way, we need to retrieve the bounding box determined by tile size. For this we use the MFC block read operation. The run-time system ensures that the last 4 bits of the effective address (EA) in the main memory and the LS address are same to avoid bus errors. It also makes sure that the data is cache line aligned to utilize the bandwidth effectively.

2. **Bounded Method for Irregular Read Accesses:** Irregular read access means that the elements needed may reside in non-contiguous areas of the main memory. To access the irregular arrays, three methods for performing the data communication was proposed in Titanium [9]. The bulk method of fetching the entire indirectly accessed array into the LS is not feasible for the Cell BE because of the limited size of local memory. The gather method is too slow to run on the Cell BE because translating indirect array accesses and fetching individual elements wastes the available bandwidth.

   The bounded method with appropriate modification is useful for the Cell BE. To utilize the LS effectively, instead of finding a single bounding box that contains all the needed elements, we prepare a list of bounding boxes of all the needed elements. Since the data transfer unit of the DMA engine is a multiple of 128 bytes, we keep the size of the box 128 bytes. The executor prepares the list of boxes and initiates the communication with the MFC DMA-list command. The compiler prepares a single bounding box within the same tile for duplicated values of the referenced array to prevent communicating the same element twice. It does memcopy of the bounded box of duplicated values across tiles while implementing double buffering.

3. **Compiler Controlled Cache for Sparse Updates:** To prevent intra-SPE coherence glitches resulting from sparse updates, the compiler must ensure that the recent copy of the data is fed to the kernel instead of prefetching the stale copy from the main memory. In each SPE, the compiler simulates a direct mapped cache that has 128 lines, each of which is 128-byte long. The compiler fetches the referenced array using the block access method. It looks up the cache maintained for each indirect reference. If the line does not contain the required data, the miss handler fetches the required data from the main memory. While transferring data from the cache of an SPE to the main memory, there is a risk of data being overwritten by the garbage values in the shared cache lines residing in other SPEs. To avoid this, only the modified data items are moved to the main memory, using simulated dirty bits.

Name	Description	Problem size 1	Problem size 2	Problem size 3
IRREG	Irregular CFD mesh	2,048 nodes	4,096 nodes	10,240 nodes
NBF	Non-bonded force (GROMOS)	8,192 nodes	16,384 nodes	32,000 nodes
MOLDYN	Molecular dynamics (CHARMM)	4,096 molecules	8,192 molecules	16,384 molecules

Table 1. Scientific applications kernels

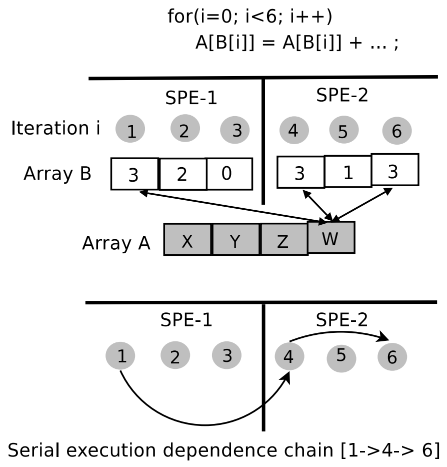
for(i=0; i<6; i++)
A[B[i]] = A[B[i]] + ... ;
SPE-1
SPE-2
Iteration i
Array B
Array A
Serial execution dependence chain [1->4-> 6]

Figure 4. Run-time parallelization of irregular reduction loops

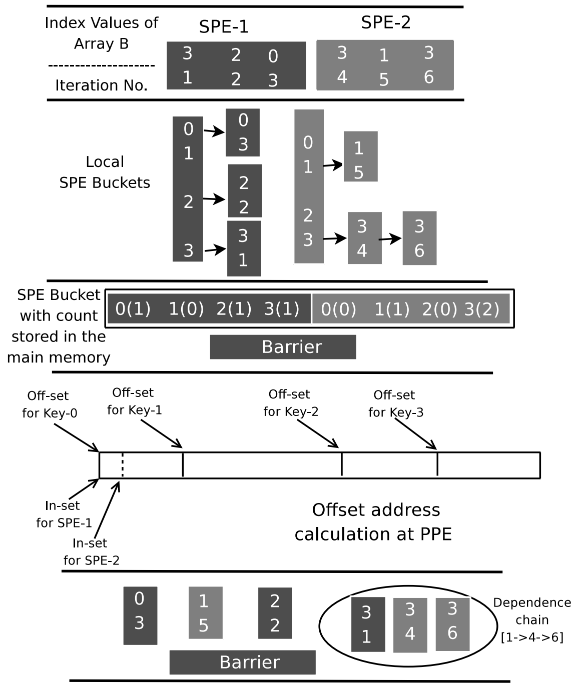

Figure 5. Parallel construction of serial dependence chain

4.2 Runtime Parallelization of Irregular Reduction Loops (Sparse Updates)

Automatic parallelization of irregular applications is different from and difficult than regular problems due to the presence of indirectly indexed array subscripts. There can be three types of dependencies between two statements depending on the memory access patterns due to duplicated values: flow dependence (read after write), anti dependence (write after read), and output dependence (write after write). A loop can be parallelized without synchronization constructs only if the loop does not have any cross-iteration dependencies. To determine whether there are any cross iteration dependencies inside the loops or not, the dependence analysis must be performed.

In our framework we have implemented the run-time parallelization of loops as shown in Figure 4. The run-time parallelization [3] system determines the cross-iteration dependence relationship by examining the data access values of the referenced arrays at run-time. And, then preserves the dependencies in accordance with the output produced by the dependence analysis. This is done by inserting synchronization constructs at proper points to respect the serial dependencies.

We gather the dependence information for the iterations that access the same memory locations. The dependence information gets stored in the ticket table that is shared amongst all the SPEs. Every chain of dependencies built in the ticket table represents the order of the memory accesses to the same location. The dependency chains are used for imposing the ordering. The iteration spaces that are not part of the dependence chain can be executed in parallel without any synchronization constructs. However, the shared memory accesses present in the dependence chain use synchronization to ensure the serial dependence.

We have implemented parallel construction of the ticket table to speedup the construction as shown in Figure 5. The ticket table is built at run-time in two phases using the inter-processor communication mechanism. To minimize the communication overhead, firstly, the local table is built at each SPE locally. And then, the global ticket table is built using the communication primitives (signals, mailbox, and

```
do t =
    do i = num_edges
        n1 = left[i]
        n2 = right[i]
        force = (x[n1]-x[n2])/4
        y[n1] += force
        y[n2] += force
```

Figure 6. Irregular memory access kernel

DMA) in the PPE.

Once the ticket table is built, each SPE speculatively executes the iteration space assigned to it as a do-all loop and at the same time it issues DMA commands to fetch the dependence chain from the main memory. In course of the execution of loop, the run-time system does the data dependence test by examining whether the accessed element is a part of the dependence chain. This is done to determine if it had any cross-iteration dependencies; if the test fails, then the current iteration gets en-queued in a buffer.

The iterations in the buffer get re-executed serially in accordance to the dependence chain. The iteration at the head of the dependence chain gets the recent copy of the accessed shared variables from the signal's inbound registers sent by the predecessor SPE in the dependence chain. The compiler inserts synchronizing constructs in order to get the recent copy of the value from the predecessor SPE in the dependence chain. Every next iteration in the buffer uses the recent copy from the current SPE. Every SPE sends the latest copy of the values to the successor SPE in the dependence chain using the signal notification channel.

5 Experiments and Results

We now present experimental results to show the performance gain of our compiler framework. We evaluated the performance of the compiler framework with three scientific applications. The real application kernels we have chosen consists of irregular reads and writes of data arrays. As an example of irregular access, consider the kernel shown in Figure 6. Here, irregular access means that subscript expressions of the data arrays x and y are not affine but are indexed through the values of other arrays. We have measured and analyzed the behavior of three kernels shown in Table 1. All our experiments are performed on a 3.2GHz Cell processor. The achieved speedup against number of active SPEs are shown in Figure 7, 8, and 9.

Results show there is no linear speedup in terms of the number of SPEs. The reason for the performance decrease is the overhead of synchronization primitives for parallelization as the number of SPEs increases.

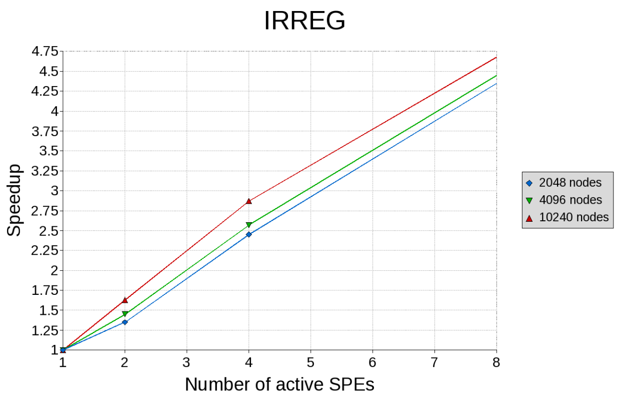

Figure 7. Performance measures for IRREG

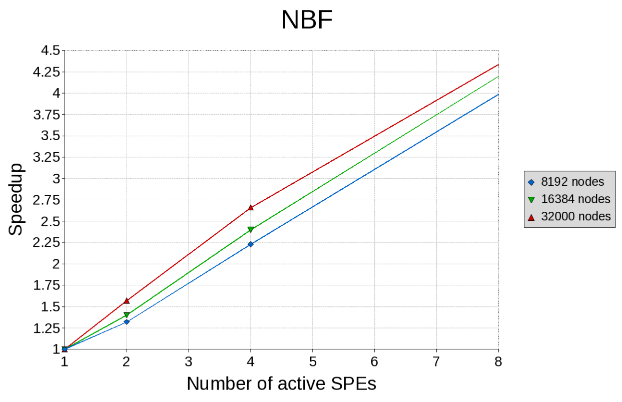

Figure 8. Performance measures for NBF

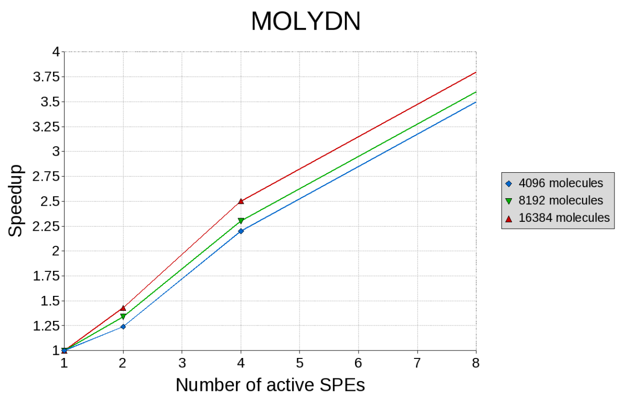

Figure 9. Performance measures for MOLDYN

6 Related Work

The Cell BE offers massive parallelism at the cost of unconventional architecture and complex programming model. Researchers have developed several strategies for overcoming the compilation challenges on the Cell BE. The IBM compiler [5] performs automatic generation of SIMD code for SPEs but follows the OpenMP based approach for parallelism. CellSs [6] proposed an alternative programming model for exploiting functional parallelism based on building task dependence graph at run-time with the help of explicit annotation. A dependence-based compiler approach for automatically generating parallel and vector code for the Cell was proposed in [13].

Researchers have investigated efficient ways for the compile-and run-time supports based on the inspector-executor technique developed by Berryman and Saltz [4]. The concept of dynamic scratch pad memory management is well-known in the context of embedded systems [10]. We use the same basic idea for managing the LS of the SPE. Prior work on run-time parallelization on multiple processors includes work and data partitioning for irregular applications [8]

In contrast to these studies, the work presented in this paper extends the compiling support for the Cell BE in the line of inspector-executor paradigm. Our compiler is able to automatically generate code for memory communication and run-time parallelization for sparse scientific applications.

7 Conclusions

In this paper, we have investigated compile-and run-time support for sparse scientific computations. We have developed automatic run-time support for memory communication and parallelization for irregular memory accesses on the Cell BE. The dataflow variable analysis generates and places the communication schedule within the limited memory available in the LS of the SPEs. The compiler facilitates the automatic data movement between the main memory and the LS of the SPEs and performs the actual computation. The run-time system overcomes the problems of data alignment constraints and explicitly managed DMAs. We have also implemented compiler-directed multi-buffering to overlap on-chip communication and SPE computation. The run-time parallelization system judiciously partitions the data and computational work to avoid sparse updates among the SPEs. It also reuses the communication schedule for amortizing the cost of dataflow analysis phase, if possible.

We have evaluated the performance of the compiler framework for irregular applications. Our preliminary results demonstrate a substantial speedup on 3.2 GHz Cell processors. Thus, we can conclude that this compiler framework would help utilize the massive parallelism in the Cell processor for irregular scientific application while relieving the programmer of the burden of carefully parallelizing the sequential code.

References

[1] Suif-2 compiler system, http://suif.stanford.edu/suif/suif2/.

[2] David A. Bader, Virat Agarwal, and Kamesh Madduri. On the design and analysis of irregular algorithms on the cell processor: A case study of list ranking. In *IPDPS*, pages 1–10. IEEE, 2007.

[3] Ding-Kai Chen, Josep Torrellas, and Pen-Chung Yew. An efficient algorithm for the run-time parallelization of doacross loops. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 518–527, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[4] R. Das, M. Uysal, J. Saltz, and Yuan-Shin S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–478, 1994.

[5] A. Eichenberger. Optimizing compiler for the cell processor, 2005.

[6] Josep M. Pérez, Pieter Bellens, Rosa M. Badia, and Jesús Labarta. Cellss: Making it easier to program the cell broadband engine processor. *IBM Journal of Research and Development*, 51(5):593–604, 2007.

[7] Ravi Ponnusamy, Joel H. Saltz, and Alok N. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Supercomputing*, pages 361–370, 1993.

[8] L. Rauchwerger, N. Amato, and D. Padua. Run-time methods for parallelizing partially parallel loops, 1995.

[9] Jimmy Su and Katherine Yelick. Array prefetching for irregular array accesses in titanium. In *Sixth Annual Workshop on Java for Parallel and Distributed Computing*.

[10] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 276–286, New York, NY, USA, 2003. ACM.

[11] Reinhard von Hanxleden, Ken Kennedy, Charles Koelbel, Raja Das, and Joel H. Saltz. Compiler analysis for irregular problems in fortran d. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 97–111, London, UK, 1993. Springer-Verlag.

[12] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.

[13] Yuan Zhao and Ken Kennedy. Dependence-based code generation for a cell processor. In *19th International Workshop, LCPC 2006, New Orleans, LA, USA, November 2-4, 2006*, pages 64–79, New Orleans, LA, USA, 2006. Springer Berlin / Heidelberg.