Compiling Irregular Accesses for the Cell Broadband Engine*

Pramod K. Bhatotia†, Sanjeev K. Aggarwal and Mainak Chaudhuri
Department of Computer Science and Engineering,
Indian Institute of Technology,
Kanpur-208016, INDIA
Email:{*pramod.bhatotia*}@gmail.com, {*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 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. The Cell BE performs extremely well for the applications where the memory access pattern is regular. However, parallelizing irregular accesses automatically and efficiently on the Cell BE [1] is a particularly challenging problem for the following reasons.

- **Compiler Analysis for Irregular Memory Accesses:** 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.

- **Runtime Parallelization of Irregular Reduction Loops:** 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.

- **Explicit Dynamic Memory Management:** Due to the limited amount of local store memory per SPE, dynamic management of local store memory is needed. For this, efficient loop tiling is required to minimize the number of bus transfers between the main memory and the

*Represents the work done in the Dept. of CSE at IIT Kanpur.

†Student author, currently working in the High Performance Computing (HPC) group at IBM India Research Lab - New Delhi.

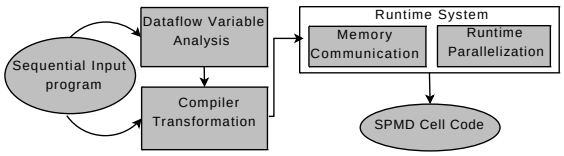

Figure 1. Overview of the compiler framework

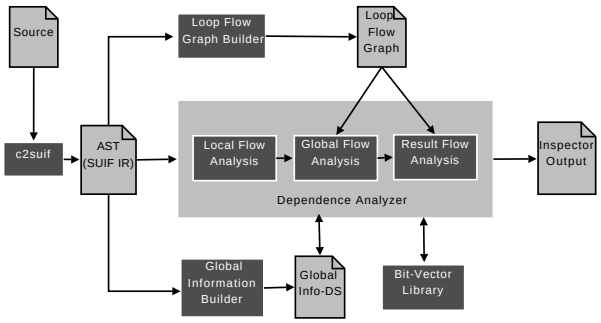

Figure 2. Dataflow analysis framework

local store by overlapping the communication with computation.

In this work, we have developed a parallelizing compilation framework for sparse scientific applications as 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 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.

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 framework analyzes [4] the input kernel based on run-time preprocessing to build the memory communication schedule as shown in Figure 2. 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.

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 as shown in Figure 3.

1) Kernel Transformations for Double-Buffering Scheme: In order to speed up the process, the computation 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. 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.

2) Loop Transformations for Tiling: Loop tiling is a key compilation transformation to accommodate both code and data within the limited size of the LS. 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.

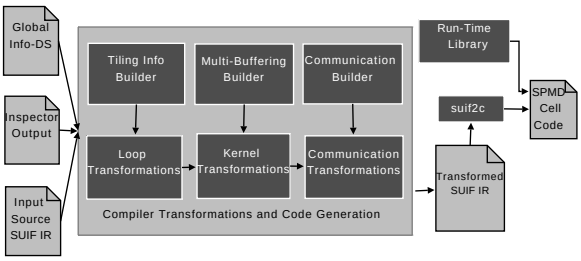

Figure 3. Compiler transformations

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 [3].

4 Compiler Controlled Runtime System

4.1 Compiler-Directed Memory Communication Mechanism

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 1 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: We prepare a list of bounding boxes of all the needed elements. The compiler 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 com-

Algorithm 1: Memory communication mechanism for the Cell processor

input : Number of program points τ, 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** τ **do**
  Determine $G[j]$, $S[j]$, and $D[j]$ using the dataflow analysis result obtained for jth program point
  $R[j]$ = $R[j-1]$ - ($S[j]$ + $D[j]$)
  $O[j]$ = $G[j]$ - $R[j]$
  ω = length($O[j]$)
  for $i \leftarrow 1$ **to** ω **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

municating 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. 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.

4.2 Runtime Parallelization of Irregular Reduction Loops (Sparse Updates)

In our framework we have implemented the run-time parallelization of loops as shown in Figure 4. The run-time parallelization [2] 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 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 dependence chain to speedup the construction. Firstly, the local chain is built at each SPE locally. And then, the global chain is built using the communication primitives (signals, mailbox, and DMA) in the PPE.

Once the dependence chain is built, each SPE speculatively executes the iteration space assigned to it as a do-all loop. The run-time system does the

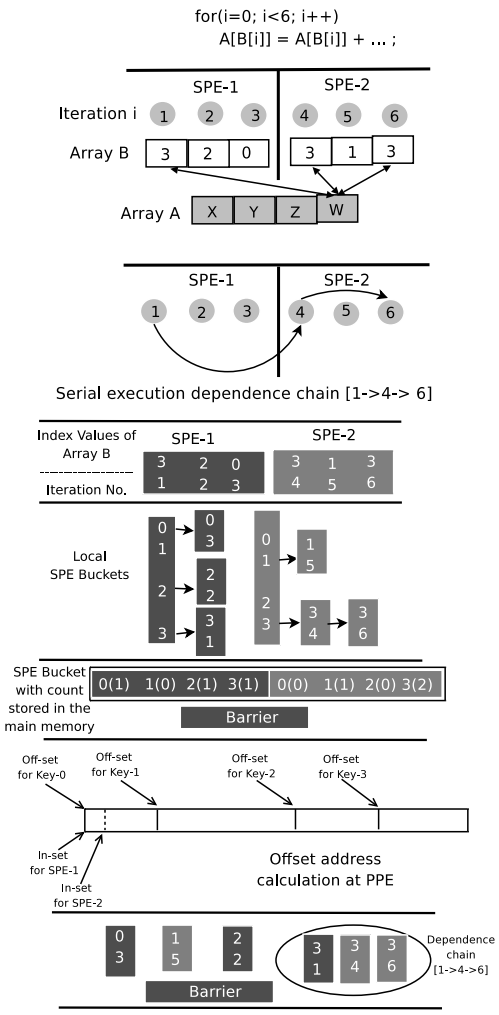

Figure 4. Run-time parallelization of irregular reduction loops

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.

5 Experiments and Results

We now present experimental results to show the performance gain of our compiler framework. We have measured and analyzed the behavior of three

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

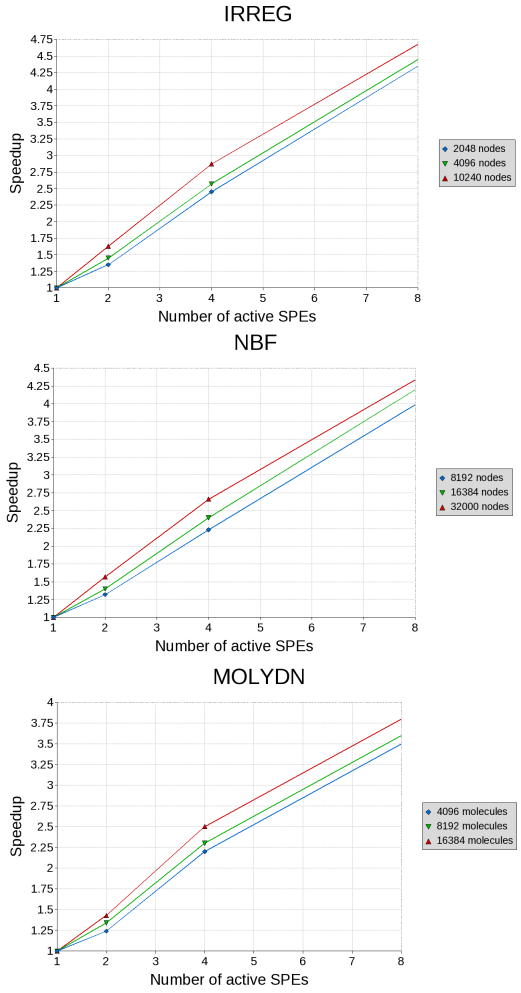

Figure 5. Performance measures

scientific applications 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 5.

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.

6 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. 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] 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*, 2007.

[2] Ding-Kai Chen, Josep Torrellas, and Pen-Chung Yew. An efficient algorithm for the run-time parallelization of doacross loops. In *SC*, 1994.

[3] Ravi Ponnusamy, Joel H. Saltz, and Alok N. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *SC*, 1993.

[4] Reinhard von Hanxleden, Ken Kennedy, Charles Koelbel, Raja Das, and Joel H. Saltz. Compiler analysis for irregular problems in fortran d. In *LCPC*, 1993.