

# Dependence-based Code Generation for a CELL Processor

Yuan Zhao      Ken Kennedy

Computer Science Department, Rice University, Houston, TX, USA  
Email: [yzhao|ken]@cs.rice.edu

**Abstract.** The CELL processor has attracted significant interest from the research community due to the performance it is capable of achieving through its multiple heterogeneous cores (scalar and vector) and high data transfer rates. However, obtaining high performance on CELL requires substantial programming effort because its architectural features must be explicitly managed, with separate codes required for different types of cores. A previous research effort at IBM has developed single source-image compiler for CELL that performs vectorization but uses OpenMP to specify cross-core parallelism. In this paper, we present and evaluate a dependence-based compiler approach that automatically generating parallel and vector code for CELL from a single source program with no parallelism directives. In contrast to the OpenMP model, our approach can also handle loop nests that carry dependences. To preserve the correct semantics, we use barrier and a uni-directional synchronization primitives based on the on-chip communication mechanisms. We have also implemented strategies to boost performance through effective management of DMA data movement and improved vector alignment by offloading peeled computations to the scalar core and exploiting memory reuse in the innermost loop. Our experimental evaluation demonstrates the effectiveness of our approach, including the reduction of the overhead of thread fork-join and the optimization of data movement through loop peeling onto the scalar core.

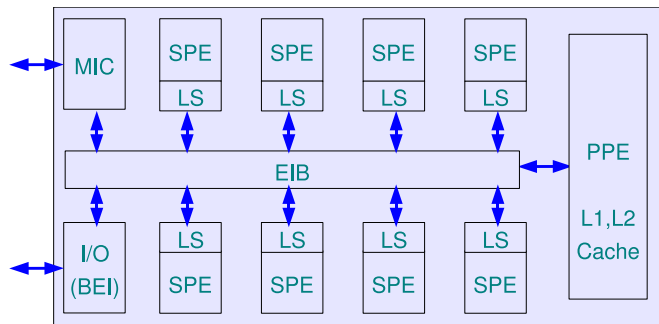
## 1 Introduction

Computing platforms with multiple processing elements—so-called “multi-core” chips—have been embraced by many chip makers. As a strategy for increasing per-chip performance, adding more cores is an alternative to scaling up operating frequency, which has become difficult because of issues such as heat dissipation. Both Intel and AMD have released dual-core (homogeneous) microprocessors with either shared or separate secondary cache.

A second emerging trend in system design is the use of heterogeneous computing components, either on or off chip. For example, The CELL processor developed by SONY, Toshiba and IBM has  $8 + 1$  (heterogeneous) processing elements. Alternatively, attached processing elements such as GPUs and FPGAs have been utilized within a single computing system to accelerate specialized applications.

One disadvantage of these multi-core approaches is that they transfer the burden of achieving high performance from the hardware to the software system or application developer. Thus, an immediate question for these platforms is: How can developers exploit the power of the parallelism? For instance, these elements can be organized by software into many different parallel computing schemes such as task parallelism and pipelined workflow. This question applies to both developing new applications and porting existing applications. Exploring trade-offs like these makes it very difficult to develop applications for these chips, particularly if a degree of portability is desired. While advanced parallel programmers can, with great effort, use expert knowledge to exploit the advanced hardware features, ordinary users may be left out in the cold. For such users, the new generation of chips desperately need automated tools and compilers that produce code with acceptable efficiency while hiding the details of the underlying hardware.

In this paper, we focus on the CELL processor, one of the most promising heterogeneous designs. We present an automatic, dependence-based code generation scheme for this chip. Figure 1, shows the architecture of a CELL processor. A single chip contains one PowerPC Processing Element (PPE) and eight Synergistic Processing Elements (SPE) on the same chip. Each SPE supports AltiVec-like vector instructions and has a 256K-byte local store memory (LS), while the PPE is a normal PowerPC core with a two-level cache. The PPE is responsible for scheduling computation tasks onto SPEs. Computation on a SPE can only access data in its own LS; data movement in between LS and main memory is explicitly controlled by DMA requests via the Element Interconnect Bus (EIB) and Memory Interface Controller (MIC).



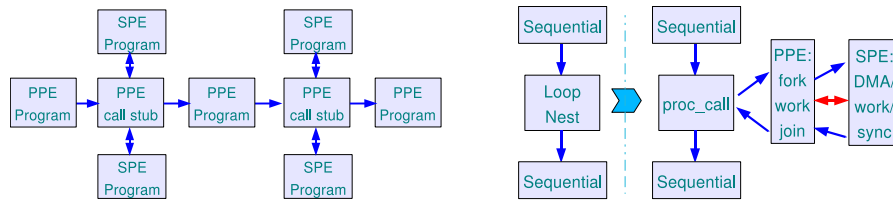
**Fig. 1.** Architecture of a CELL processor

With architecture features such as heterogeneous multi-core, parallelism at both coarse (across PEs) and fine (vector, within each PE) granularities, high data transfer bandwidth (200GB/s on chip and 25.6 GB/s off chip), and explicit local memory control through DMA, a 3.2 GHz CELL processor can, in theory, achieve a peak performance of more than 200 GFlops per second for single precision floating point computations. However, actually obtaining high performance levels requires significant programming effort because, in the distributed programming tool system, these architectural features must be explicitly managed

by the programmer. This adds a significant degree of complexity to programming to conventional uniprocessors or even simple shared-memory parallel systems.

Over the years, there have been many parallel programming models developed to reduce the burden on ordinary users. They include directive-guided task parallel models such as OpenMP, data parallel models such as HPF, and programming models with language extensions such as UPC and CoArray Fortran. Recently, the DARPA HPCS program has sponsored the development of a new generation of “high-productivity” parallel programming languages such as IBM-X10, Sun-Fortress and Cray-Chapel. However, among these models, OpenMP is to date the only one supported on CELL, and that is by an IBM research compiler [9] rather than the standard distributed tool set.

In this paper, we present a dependence-based approach to automatically generating code for CELL from a high-level language. Our system performs parallelization, vectorization and data movement optimization automatically. The input program is a single source sequential program without any parallelism directives. The output is a PPE program with many SPE programs, where parallelism is realized by the PPE thread fork-and-joining the SPE threads, as shown in Figure 2. This parallelism scheme is similar to the OpenMP approach



**Fig. 2.** Fork-and-join execution model and compilation model on CELL

used by the IBM research compiler, except that it uses fully automatic detection of parallelism in place of user-entered directives. Using information from dependence analysis, the compiler determines whether a loop nest can be parallelized across PEs and vectorized on each PE. Each such loop nest will be partitioned, using a technique called *procedure outlining* (the opposite of inlining), into a PPE call stub function and a SPE program, as shown in Figure 2. In contrast to the OpenMP approach, our strategy can also handle loops carrying dependences, preserving the correct semantics of the program using barrier and a uni-directional synchronization primitives we developed using the on-chip communication mechanisms.

Our dependence-based approach automatically orchestrates and optimizes data movement between the SPE local stores and main memory. For memory references in the original program, multiple data buffers are created on the SPE side and DMA transfer commands are placed accordingly in the SPE program. This strategy also handles the complications of data movement due to the data alignment constraints. In particular, loop peeling on the PPE side can help improve both data movement performance and vector data alignment.

We present the main parallelization and vectorization algorithm in Section 2, while the data movement analysis and placement algorithm is the subject of Sec-

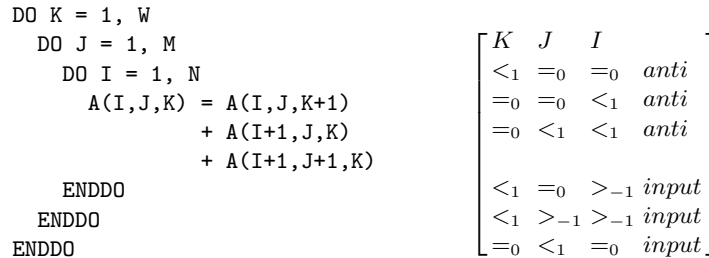
tion 3. We discuss our preliminary performance evaluation in Section 4, review related work in Section 5, and conclude in Section 6.

## 2 Parallelization and Vectorization

In this section, we present an algorithm for determining whether a loop nest can be parallelized across PEs and vectorized on each of them. Such a loop nest will be procedure outlined into a program for the SPEs, and replaced by a call to a call stub function in the PPE program that manages fork-and-join operations for SPE threads. The SPE program will contain the vectorized loop nest with automatically generated data movement. Synchronization will be generated and placed in both the SPE and PPE programs if necessary. We refer to this loop nest as *cellized* and this process as *cellization*.

To analyze if a loop nest can be cellized, we need to determine if there exists a loop in the loop nest that can be parallelized, and a loop (the innermost loop) that can be vectorized on SPE and PPE. These two candidates can be the same, *i.e.* the innermost loop can be both parallelized and vectorized, assuming that the loop is longer than the four iterations that would fit in a single vector operation.

**Data Dependences** The cellization analysis relies on the data dependence information. We assume that readers are familiar with the concept of true, anti-, output and input dependences, and with loop-carried and loop-independent dependences. Allen and Kennedy [4] have defined the *dependence matrix* for a loop nest to be a matrix in which each row is a dependence vector for some dependence in the nest and every such direction vector is included as a row. The columns from left to right correspond to the loops from the outermost to the innermost. A dependence is said to be *carried* by a loop if the corresponding entry is the first non-“=” entry in the dependence vector. Figure 3 gives an example



**Fig. 3.** Example of a loop nest and its dependences

of a loop nest and its dependence matrix. In the matrix, the anti-dependences in rows 1 to 3 corresponds to dependences from the first, second and third array references on the right hand side to the array reference on the left hand side. The dependence in the first row is carried by the K loop, the dependence in the second row is carried by the I loop, and the one in the third row is carried by the J loop. Rows 4 to 6 correspond to the input dependences among the right hand

side array references themselves. Note that for the purpose of parallelization and vectorization analysis, input dependences are omitted from consideration.

**Cellization Analysis** Our cellization analysis algorithm is modified from the Allen-Kennedy loop nest vectorization algorithm [4]. As shown in Figure 4, our algorithm searches from the outermost loop towards the innermost loop for a loop that can be parallelized in the remaining dependence matrix. At each step, if no loop is found, the current outermost loop is made sequential and all dependences carried by it are removed from the dependence matrix and the search process is repeated. After a parallel loop is identified, the loop nest is cellizable if the innermost loop is vectorizable on PPE and SPE. Obviously, the algorithm can be relaxed to accommodate loop nests that can be parallelized but not vectorized.

```

procedure IsCellizable(LN)
  // LN: loop nest {L1, L2, ..., Ln} from outermost to innermost
  obtain dependence matrix DM for LN
  initialize ParallelFlags[1:n] to UNMARKED
  while there are loops left in LN
    Search for Li in DM such that all entries in the Li column are “=”
    if Li does not exist
      if there are no loops left in LN
        ParallelFlags[n] := PARAVEC
        return IsShortVectorizable(Ln, DM)
      else
        ParallelFlags[current outermost] := SEQUENTIAL
        remove dependences carried by the current outermost loop from DM
        remove the current outermost loop from LN
    else
      ParallelFlags[i] := PARALLEL
      return IsShortVectorizable(Ln, DM)

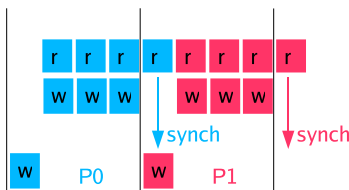
procedure IsShortVectorizable(L, DM)
  // L: the innermost loop
  // DM: the dependence matrix after a parallel loop is selected
  // or all outer loops are made sequential
  for all dependences carried by an outer loop Lk in L1 .. Ln-1 in DM
    remove such dependences from DM
  if L carries a dependence cycle among statements
    except an anti-dependence on a statement itself
    return false
  for all array references A in L
    if A is not loop invariant to L and not contiguous in memory
      return false
  return true

```

**Fig. 4.** Cellization analysis algorithm

We allow the innermost loop carrying anti-dependences to be parallelized and vectorized so long as it doesn't have a dependence cycle among different statements. As an exception, an anti-dependence cycle on a statement itself is

allowed. To preserve the semantics of a loop that has anti-dependences, we use a post-store strategy using a uni-directional synchronization. As illustrated in Figure 5,  $P1$  needs to temporarily hold the computation results in the buffer only for those writes whose main memory locations are also read by  $P0$ . The buffer for temporarily storing those results on  $P1$  can completely remain local in LS and not appear in main memory.



**Fig. 5.** Parallelizing a loop with anti-dependence

For the example loop nest shown in Figure 3, following the algorithm, after the first iteration of the search process, the K loop will be marked SEQUENTIAL and the dependence in row 1 will be eliminated from the dependence matrix; after the second iteration in the search process, the J loop will be marked SEQUENTIAL and the dependence in row 3 will be eliminated; after the third iteration in the search process, the I loop will be marked PARAVEC and sent to check for vectorization. Since the I loop carries an anti-dependence on the same statement and all array references are memory continuous to the loop, the I loop is vectorizable. Hence the loop nest is cellizable.

**Parallel Code Generation** After a loop nest is identified as cellizable, it will be procedure outlined into a SPE program and replaced with a call to a PPE call stub function which manages SPE task allocation and fork-and-joining of SPE threads. To partition  $n$  iterations of the chosen parallel loop among  $p$  PEs, each PE can get  $(n/p + (n\%p >= id?1 : 0))$  iterations, where  $id$  is the rank of the current PE. Other than partitioning, parallel code generation also needs to place synchronization accordingly, as shown in Figure 6.

The vector instruction sets on PPE and SPE are of short vector length (16 bytes) and support only contiguous memory accesses. Though certain patterns of strided accesses can be realized by data permutation afterward, we only consider contiguous memory accesses in this paper. Since there are many papers (see related work in Section 5) addressing loop based vectorization for SSE and AltiVec instruction sets, we will not discuss them in details.

For our example loop nest in Figure 3, the generated PPE code, PPE stub function code and the SPE code are shown in Figure 7. Note that the first element of the computed buffer is stored back after unidirectional synchronization in order to preserve the correct semantics of anti-dependences. The number of elements that need to be post-stored should be no less than the maximal distance of the anti-dependences.

**Load Balancing** Since PPE forks and joins SPE threads sequentially, a load balancing strategy should consider the cost of thread forking. Assume the cost is equal to that of executing  $c$  iterations of the chosen parallel loop, to partition

```

procedure Cellize(LN)
  // LN: loop nest {L1, L2, ..., Ln} from outermost to innermost
  for loop Li in LN from outermost to innermost
    if ParallelFlags[i] is SEQUENTIAL
      output Li in the new loop nest
      if ParallelFlags[i+1] is not SEQUENTIAL
        output barrier synchronization
    else if ParallelFlags[i] is UNMARKED
      output Li in the new loop nest
    else if ParallelFlags[i] is PARALLEL or PARAVEC
      output Li with partitioned iterations
  vectorize Ln
  if ParallelFlags[n] is PARAVEC and Ln carries anti-dependence
    apply post-store to the vectorized Ln
    insert uni-directional synchronization

```

**Fig. 6.** Cellization code generation algorithm

	PPEStub1(...) {	SPEmain(...) {
	put loop invariants	dma_get(data block)
	into data block;	for k = 1, w {
	for(id=1; id<PEs; id++)	for j = 1, m {
	spe_thread_fork(&block);	barrier_synch;
		compute my_lb, my_ub
call PPEStub1(...)	for k = 1, w {	dma_get(b2, b3, b4);
	for j = 1, m {	for i = my_lb, my_ub
	barrier_synch;	b1(i)=b2(i)+b3(i)+b4(i);
	compute my_lb, my_ub;	bt(1:1) = b1(my_lb:my_ub)
	for i = my_lb, my_ub	dma_put(b1(my_lb+1:my_ub));
	a(...)=a(...)+...;	uni-directional_synch;
	uni-directional_synch;	dma_put(bt(1:1));
	} }	} }
	}	}
(a) PPE program	(b) PPE stub	(c) SPE program

**Fig. 7.** Example of code generation

$n$  iterations across  $p$  SPEs, the  $k$ -th PE will get  $w_k = n/p + c(p - 2k + 1)/2$  iterations. This is derived from  $w_{k+1} = w_k - c$  and  $\sum_{k=1}^p w_k = n$ . While  $c$  could be determined by estimating the cost of each iteration with program analysis, it could also be decided with profiling information. If the measured thread forking time and total running time on one SPE are  $t_f$  and  $t$ , respectively, assuming the parallel loop is the outermost loop, then  $c = nt_f/(t - t_f)$ .

**Synchronization Implementation** The only communication mechanism on CELL other than DMA transfers that does not need special privilege is mailbox. Other mechanisms such as SPE-to-SPE signals require the executable be granted a certain capability (CAP\_SYS\_RAWIO). According to the linux man pages, the capability granting feature is still being developed in the Linux Kernel as of now. Therefore, we implemented the barrier and the uni-directional syn-

chronization using mailbox as shown in Figure 8. The number associated with each arrow (mailbox send) indicates the mail sending order.

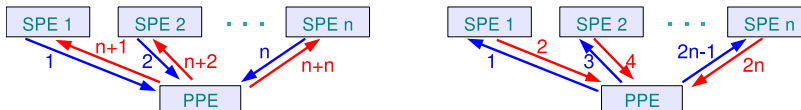


Fig. 8. Barrier and uni-directional synchronization using mailboxes

### 3 Data Movement and Alignment

In this section we discuss how to achieve an efficient data movement in between the SPEs' local memory and the main memory. To do this we must:

1. hide the latency of data movement by multi-buffering so that the DMA data transfer and computation are partially overlapped, if not fully, and
2. use the available bandwidth effectively, *i.e.* all data brought in should be used and repeated use of the same data should require the data be transfer as fewer times as possible.

The second issue is only briefly addressed in this paper. We will focus on multiple buffer allocation, DMA data transfer generation and placement.

Figure 9 gives an algorithm for analyzing and allocating multiple data buffers. The reference group partitioning in the algorithm is exactly the one used for scalar replacement [7]. We do not yet have a good buffer size estimation algorithm. The data buffer size depends on a set of predefined blocking sizes for the loop nest. Since the maximum size of a single DMA transfer is 16K bytes, it is desirable to allocate a size close to that limit, assuming that the combined code and data buffer fit into the SPE's 256K-byte local memory. For the generated code in Figure 7, following the algorithm, the I loop would be strip-mined and multi-buffering data buffers would be allocated.

Another factor affecting data movement performance on CELL is data alignment. It constraints data movement in three ways: vector offset, naturally aligned boundaries, and cacheline alignment.

**Vector Offset** DMA transfer requires that the last 4 bits of the source and sink addresses be the same. Therefore, when converting an array reference in the main memory space into a local buffer reference, we need to make sure that they have same vector offsets.

**Naturally Aligned Boundaries** DMA transfer can transfer 1, 2, 4, 8, 16,  $16*k$  bytes (max 16K Bytes) on naturally aligned boundaries. When the starting and ending addresses are not properly aligned, one has to issue multiple DMA instructions, as shown in Figure 10.

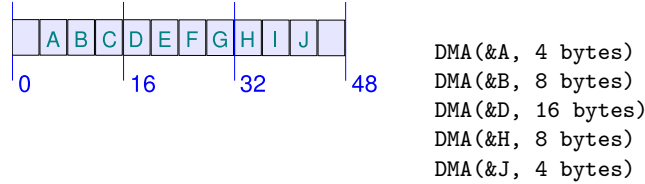
For DMA transfers that get data from the main memory to the SPE's local memory, we can always over-fetch, *i.e.* extend the memory transfer region to naturally aligned boundaries on both ends. This is always safe as long as we are

```

procedure BufferAllocation(LN)
  // LN: loop nest {L1, L2, ..., Ln} from outermost to innermost
  perform reference group partitioning for all array references
  for the group generator of each reference group
    determine the innermost loop that the generator is loop variant
    determine if the generator is memory contiguous to that loop
  for each array reference in the loop nest
    if it is a group generator
      strip-mine the innermost variant loop if memory contiguous within it
      allocate multi-buffering buffers
      place DMA prefetch current buffer before the loop
      place DMA prefetch next buffer inside the loop
      place DMA check data ready inside the loop
      place buffer index rotation computation at the end of the loop
      replace array reference with allocated buffer
    else
      replace array reference with buffer allocated by the group generator
      adjust subscript by the dependence distance to the generator

```

**Fig. 9.** Multi-buffering analysis and allocation algorithm



**Fig. 10.** DMA transferring unaligned data

careful when allocating buffers. However, we cannot easily use this strategy for the DMA puts from SPEs to the main memory. In short, we have a false sharing problem. Over-storing would require an atomic operation to get the boundary data from main memory, merge with the local data, and put data back into the main memory on the extended boundaries. Implementing such an atomic operation would incur a significant performance penalty.

**Loop Peeling** To reduce the performance penalty related to the unaligned DMA puts, we apply loop peeling to the innermost loop on the PPE side and only compute partitioned loop iterations as multiples of the SPE vector size. The peeled iterations are computed on PPE, while the starting address for a DMA put from an SPE is always at a vector boundary. Loop peeling on the PPE is always safe if the innermost loop is parallelized, or if a loop is chosen to parallelized only when there are no dependences left in the dependence matrix following the algorithm in Figure 4. There is a slight issue due to the different rounding modes for floating point on PPE and SPE. We leave it up to the application developers to decide if this discrepancy actually matters.

**Cache Line Alignment** When a CELL processor performs DMA transfers, the unit data transfer is actually a cache line, *i.e.* when transferring data smaller than a cache line size, the bandwidth of an entire cache line will be consumed and the result is masked. On the other hand, when transferring a large data

block that is not aligned on cache line boundaries, two whole cache lines will be transferred on the boundaries even though only a part of each cache line is requested. Loop peeling can also be used to ensure data alignment on the cache line boundaries. The wasted cache line bandwidth is relatively small when actually transferring a large data block. This is another reason why a large tile size should be used whenever possible.

## 4 Performance Evaluation

**Experimental Setup** A dependence-based code generator for CELL (CELLizer) that includes the parallelization, vectorization, multi-buffering DMA transfer and synchronization algorithms described in Section 2 and Section 3 is implemented as an extension of our earlier tool for vectorization and code generation for short vector machines with SSE and AltiVec, which in turn was based on the D System compilation infrastructure at Rice.

CELLizer accepts a FORTRAN 90 program as source input, performs cellization, outputs a FORTRAN 90 program which will remain on PPE, a collection of PPE stub functions written in C, and a collection of corresponding SPE programs written in C, as we described in Section 1. Each SPE program corresponds to a cellized loop nest that is procedure outlined and rewritten in C.

We use the cross-platform compilers (ppuxlc and spuxlc) in the CELL software development kit version 1.1 to compile the PPE call stub functions and the SPE programs. The remaining PPE program in Fortran is translated to C using the *f2c* converter [11], and compiled with ppuxlc respectively.

The compiled executable is transferred to a 3.2 GHz Cell Blade running Linux OS for correctness verification and performance measurement. The Cell Blade has 1G bytes memory installed, while the swap disk is turned off. We use the Linux library call `gettimeofday()` to measure the time spent in either the original loop nest, or the PPE stub functions and the SPE threads combined. Each code is run 6 times and the minimum running time is kept as record. The speedup of running the generated code on various number of SPEs over running the original loop nest on PPE alone is reported. Our CELLizer has the following compiler options that can be changed.

Option	Meaning
-cell <arg>	generate code for cell processor with <arg> SPEs
-plp	loop peeling on PPE to improve data alignment
-pwk <arg>	work on PPE (<arg> as a percentage to the work of a single SPE)
-vsr	do vectorized scalar replacement
-spls	do software pipelined aligned load/store
-novec	no vectorization, generate sequential C code

**Table 1.** CELLizer options

**Test Cases** We tested three small loop nests *LN1*, *LN2*, *ANTI*, shown in Figure 11. Each loop nest has an initialization before it and a checksum

computation after it. Note that *ANTI* carries an anti-dependence which usually requires allocating a temporary array and splitting the original loop into two so that each loop can be parallelized, as shown in *ANTITMP*. Our CELLizer, on the other hand, avoids this temporary array allocation by using post-store and uni-directional synchronization. We also tested two programs from SPEC benchmark, *swim* and *mgrid*. In all test cases, data types were converted to be single precision floating point.

```

LN1:
  DO I = 1, N-2
    B(I+1) = (A(I) + A(I+2) + C(I+1)) * 0.34
  ENDDO

LN2:
  DO J = 2, M-1
    DO I = 2, N-1
      A(I,J) = A(I,J) + (B(I-1,J)+B(I+1,J)+B(I,J-1)+B(I,J+1))*0.25
    ENDDO
  ENDDO

ANTI:
  DO I = 1, N-2
    A(I+1) = (A(I+2) + B(I+1) + C(I+1)) * 0.34
  ENDDO

ANTITMP:
  DO I = 1, N-2
    D(I) = (A(I+2) + B(I+1) + C(I+1)) * 0.34
  ENDDO
  DO I = 1, N-2
    A(I+1) = D(I)
  ENDDO

```

**Fig. 11.** Testing codes

The performance results are shown in Figure 12, Figure 13 and Figure 14, respectively. For *LN1*, both a small problem size  $N = 7555333$  and a big problem size  $N = 44222111$  are tested. Similarly,  $M = N = 2955$  and  $M = N = 7255$  are tested for *LN2*,  $N = 7555333$  and  $N = 33222111$  are tested for *ANTI* and *ANTITMP*. For SPEC benchmark programs, the reference problem sizes are tested, *i.e.*  $1335 \times 1335$  for *swim* and  $128 \times 128 \times 128$  for *mgrid*.

The default set of optimizations includes parallelization across multiple PEs (-cell <arg>), multi-buffering data movement, vectorization, vectorized scalar replacement (-vsr), software pipelined load/store (-spls), and loop peeling on PPE for data alignment (-plp). In Figure 12, “pwk10” (“pwk20”) assigns 10 percent (20 percent, respectively) of a single SPE’s work/iterations to PPE, in an attempt to balance the load across all PEs; while “noplp” turns off the loop peeling on the PPE side.

**Performance Results** Based on the performance data, we have the following observations:

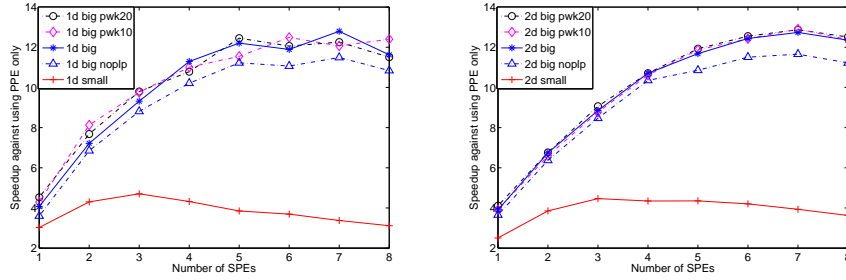


Fig. 12. Performance for 1D (*LN1*) and 2D (*LN2*) stencils

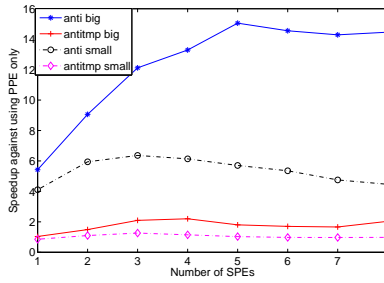


Fig. 13. Performance for loop (*ANTI*) with anti-dependence

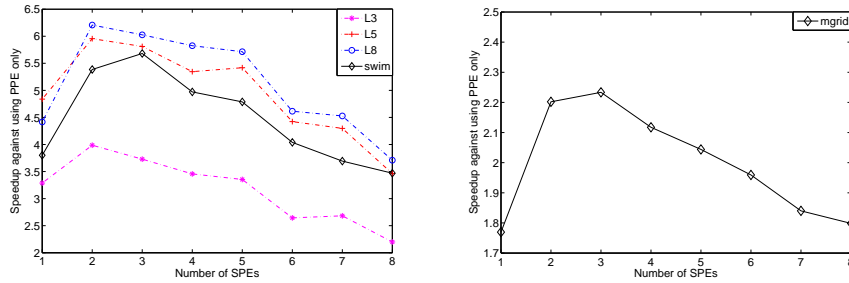


Fig. 14. Performance for *swim* and *mgrid*

- Programs run as much as 3 times faster on SPE than on PPE. There might be two reasons for this phenomenon. First, the latency of floating point instructions on SPE (6 cycles) is shorter than that on PPE (10-12 cycles). Second, PPE is also responsible for running the operating system.
- We do not get a linear speedup to the number of SPEs used. In fact, what we observe is a curve, *i.e.* the best speedup occurs when less than 8 SPEs are used for small problem sizes. The reason is that thread fork-and-join operations become higher in cost when more SPEs are used, compared to the computation cost on each SPE. We are still investigating the effectiveness of the load balancing strategy described in Section 2. Nevertheless, increasing the problem size can mitigate this problem by offering each SPE more

work so that thread fork-and-join costs become less obvious, as illustrated in Figure 12 and Figure 13.

- Our strategy of loop peeling on the PPE side to help improve DMA data alignment works well. Compared to not performing loop peeling, our strategy achieved an average of 9 percent speed improvement for the 1D stencil and 7 percent for the 2D stencil, as shown in Figure 12. Note that for the two dimensional array in 2D stencil, the loop peeling strategy will dynamically calculate the number of “I” iterations it needs to peel within each outer “J” iteration, so that on SPEs the array reference on the left hand side is always properly aligned when entering the “J” loop.
- Comparing the performance between *ANTI* and *ANTITMP* in Figure 13, we can clearly see the benefits of our parallelization strategy using post-store and uni-directional synchronization to handle the anti-dependences. The extra memory traffic in *ANTITMP* and extra thread fork-and-joining costs have caused heavy performance penalty. This result also suggests that loop fusion combined with array contraction work well for the CELL processor if the resulting loop with dependences can be parallelized with post-store strategy and proper synchronization.
- When running whole applications on CELL, we can tune each loop nest independently to determine the number of SPEs for the best speedup, instead of using the same number of SPEs for all of them. Figure 14 shows that the tuning result for three most time-consuming loop nests, *L3*, *L5*, and *L8* in the *swim* benchmark. Unfortunately, they all achieve the best speedup with two SPEs, while the whole application achieve the best speedup with three SPEs. The discrepancy is caused by collecting timing information of *L3*, *L5* and *L8* for only 20 iterations, while they are iterated for 1200 iterations in *swim*, as specified by the SPEC benchmark.
- Finally, contrary to what people may think, having PPE work on partitioned workload doesn’t seem to help much due to the relative slowness of the PPE, as shown in Figure 12. However, loop peeling on the PPE can help improve DMA data alignment. Load balancing across all PEs needs further investigation.

## 5 Related Work

Increased processor parallelism on chip has become a trend adopted by almost all major chip makers. The parallelism can be found at both coarse and fine level granularity. On coarse level, more and more cores (PEs) are built onto a single chip; on the fine level, vector instruction sets are included and regularly amended (*e.g.* SSE4). Research in compiling for parallelism at both levels have had a long history with many languages, tools and programming models. However, up till now, the OpenMP approach, used in an IBM research compiler [9], is only one implemented to support CELL.

The dependence-based approach presented in this paper is related to nearly two decades of work on loop nest parallelization and vectorization work done for

various parallel architectures [1, 3, 4, 2, 5, 8, 13, 10, 9, 21, 15, 16]. The data alignment problem on the CELL is very similar to that on SSE and AltiVec [10, 21], except on CELL data alignment also affects data movement. Scalar replacement has been developed to improve data reuse at register level [7] and later extended to vector level [17, 21]. In this paper, we use reference group partitioning to identify the array references that should share a data buffer.

Multi-buffering analysis and placement is very similar to that of software prefetching [6, 14], which was designed to hide the memory latency. On the other hand, optimizing DMA transfers using multi-buffering is similar to array copying [12, 18–20], which was proposed to eliminate conflict misses in the cache.

Effectiveness of traditional memory hierarchy optimizations for SPE’s local store, *e.g.* loop tiling, is demonstrated in the IBM work [9]. Since we have not implemented these optimizations in our compiler, we have not yet achieved same amount of speedup as the IBM work for the same benchmark programs. We are currently investigating the optimizations to improve data reuse in SPE’s local store.

## 6 Conclusion

In this paper, we presented a dependence-based automatic code generation strategy for the CELL processor. In contrast to approaches that use explicit parallelism directives, such as OpenMP, it doesn’t require that the user perform dependence analysis manually and determine the legality of the hand-specified parallelization. Besides automatic parallelization, vectorization, and multi-buffering DMA data transfer generation, our CELLizer can also insert two kinds of synchronization automatically to preserve the correctness of a program.

Our performance study verified the correctness of our code generation strategy and the usefulness of performing loop peeling on the PPE side to help improve data alignment. However, we did not observe linear speedup in terms of the number of SPEs. In fact, the performance decreases after the number of SPEs exceeds some number. To address this issue, we must improve the amount of computation per transferred data element. One way to do this is through a more thorough use of loop fusion and tiling transformations. This is a goal of our ongoing research.

## Acknowledgments

This work is supported by the Department of Energy under Contract No. 12783-001-0549 from the Los Alamos National Laboratory to William Marsh Rice University, and the CELL development systems (including CELL blades) provided by IBM to University of Tennessee and Rice University.

## References

1. J. R. Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformation*. PhD thesis, Rice University, Houston, Texas, 1983.

2. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1987.
3. R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):pp. 1290–1317, 1992.
4. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman, October 2001.
5. A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic intra-register vectorization for the intel architecture. *International Journal of Parallel Programming*, 30(2):pp. 65–98, 2002.
6. D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.
7. S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 15(3):pp. 400–462, 1994.
8. Crescent Bay Software. *VAST/AltiVec*. <http://www.crescentbaysoftware.com/vast.altivec.html>.
9. A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for a cell processor. In *PACT*, 2005.
10. A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD architectures with alignment constraints. In *PLDI'04*, June 2004.
11. S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer. A fortran-to-C converter. Technical Report 149, AT&T Bell Laboratories, Murray Hill, NJ, 1990.
12. M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS-IV: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.
13. S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, 2000.
14. T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, California, 1994.
15. D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, 2006.
16. D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *PLDI*, 2006.
17. J. Shin, J. Chame, and M. W. Hall. Compiler-controlled caching in superword register files for multimedia extension architecture. In *PACT*, 2002.
18. O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, November 1993.
19. R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–25, 2001.
20. Q. Yi. Applying data copy to improve memory performance of general array computations. In *LCPC*, October 2005.
21. Y. Zhao and K. Kennedy. Scalarization on short vector machines. In *2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 20–22, 2005.