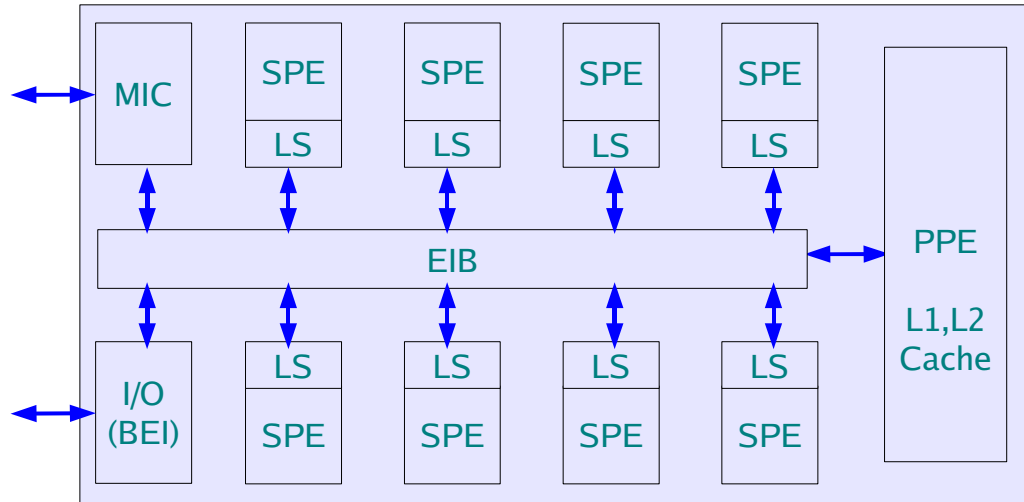


Dependence-based Code Generation for a CELL Processor

Yuan Zhao Ken Kennedy
Department of Computer Science
Rice University

Workshop on
Languages and Compilers for Parallel Computing
November 2nd, 2006
New Orleans, Louisiana

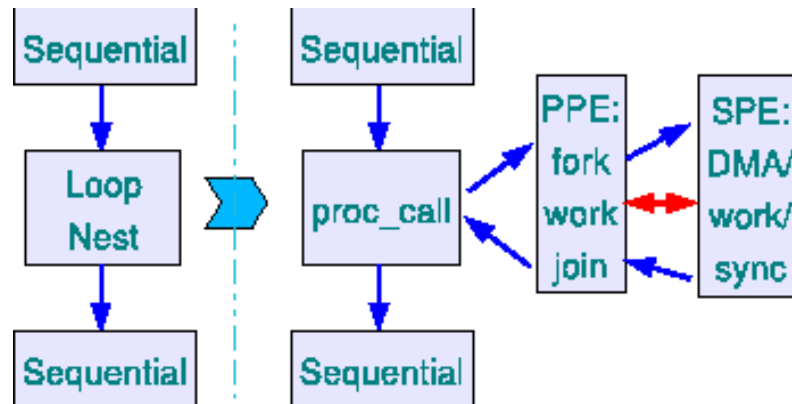
CELL Processor



- 1 PPE (AltiVec), 8 SPE (vector) each with 256KB LS
- PPE responsible for fork-and-joining SPE threads
- Each SPE has its own address space and can only access data from LS, data transfer explicitly controlled through DMA
- When running at 3.2GHz, about 200GFlops for single precision floating points, 20GFlops for double precision
- Compilation challenges
 - Code generation, finding sufficient parallelism: coarse and fine level
 - Efficient data movement strategy
 - Temporary allocation reduction

Compilation Model

- Fortran programs, array syntax supported
- Dependence-based automatic approach
 - Loop nests in regular applications
 - No parallelism directives/pragmas
- Source-to-source compiler, procedure outlining
 - Rewrite in C to utilize parallelization libraries, vector data types and intrinsics



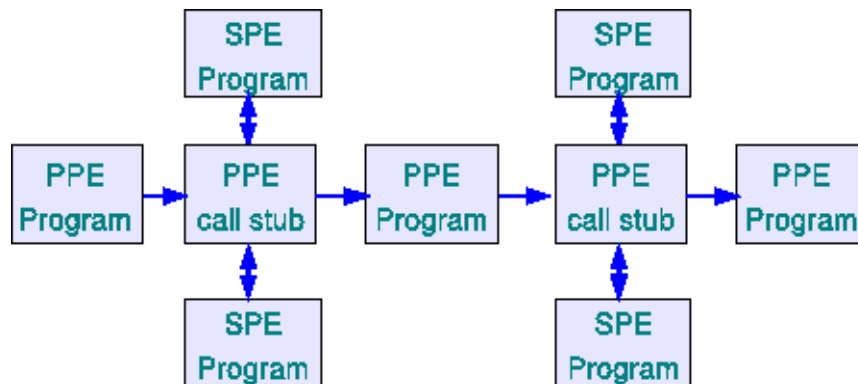
- Parallelization, vectorization (PPE and SPE), data movement, data alignment, data reuse, synchronization

Outline

- Parallelization and vectorization
- Data movement
- Experimental results
- Conclusion and future work

Related Work on Code Generation

- Parallel programming languages and models
 - HPF, OpenMP, CoArray Fortran, UPC, MPI, HPCS languages
- Related work: user specified parallelism directives/pragmas
 - Eichenberger et al, IBM research compiler, OpenMP



- Related work: automatic parallelization and vectorization
- Our approach: a dependence-based automatic code generation strategy
 - Loop nests in regular applications
 - No parallelism directives

Dependence-based Code Generation

- Parallelization and vectorization
 - A loop is parallelizable if it doesn't carry dependences
 - The innermost loop is short vectorizable if it doesn't carry dependence cycles and array references are memory contiguous
 - An anti-dependence on a statement itself is allowed
 - A short vectorizable loop can always be parallelized
 - Anti-dependence to be preserved by post-store and synchronization
 - Algorithm to identify a parallel loop and a short vectorizable loop
 - Dependences are carried at different levels
 - Search from the outermost loop towards the innermost loop
 - If the loop carries no dependence, mark parallel
 - Otherwise, make it sequential, remove the loop and all dependences it carries, repeat the search process
 - Check the innermost loop for short vectorizability

Dependence-based Code Generation .

- Example

```
DO K = 1, W
  DO J = 1, M
    DO I = 1, N
      A(I,J,K) = A(I,J,K+1) + A(I+1,J,K) + A(I+1,J+1,K)
    ENDDO
  ENDDO
ENDDO
```

K	J	I		
<1	=0	=0	anti	Step 0
=0	=0	<1	anti	
=0	<1	<1	anti	
	=0	<1	anti	Step 1
	<1	<1	anti	
		<1	anti	Step 2

- K,J sequential, with barrier synchronization
- I parallel, iterations partitioned, anti-dependence
 - post-store and uni-directional synchronization
- I vectorizable

Parallelizing Loop with Anti-dependence

- Failsafe method

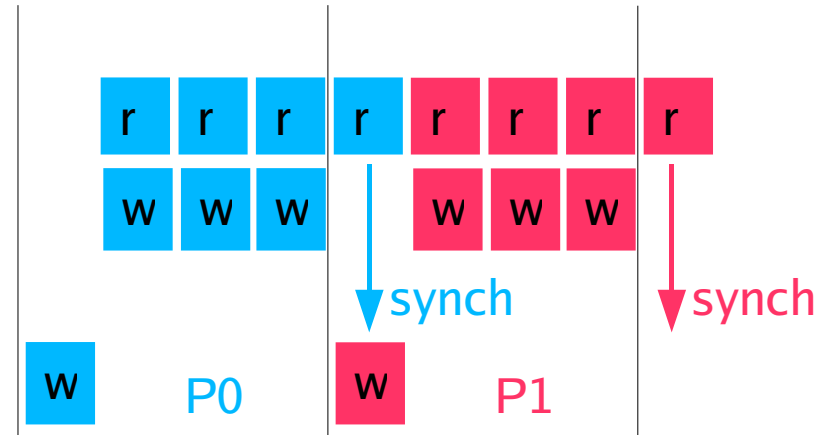
- easy CELLization
- extra data transfer

$$A(2:N-1) = B(1:N-2) + A(3:N)$$

```
DO I = 1, N-2   T(I) = B(I) + A(I+2)   ENDDO
DO I = 1, N-2   A(I+1) = T(I)          ENDDO
```

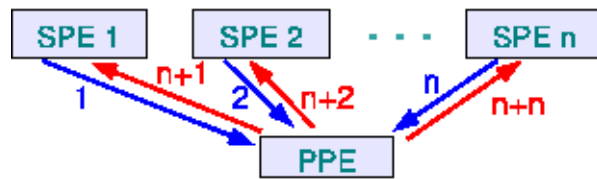
- Temporary allocation reduction

- Post-store, synchronization
- Privatize temporary for each PE
SPE's copy can reside in LS only

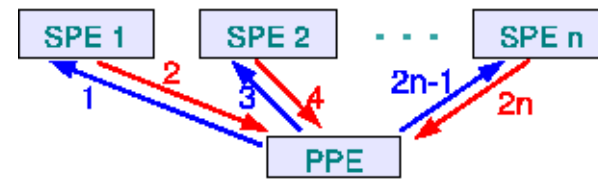


Dependence-based Code Generation ..

- Synchronization
 - Implemented with mailbox mechanism



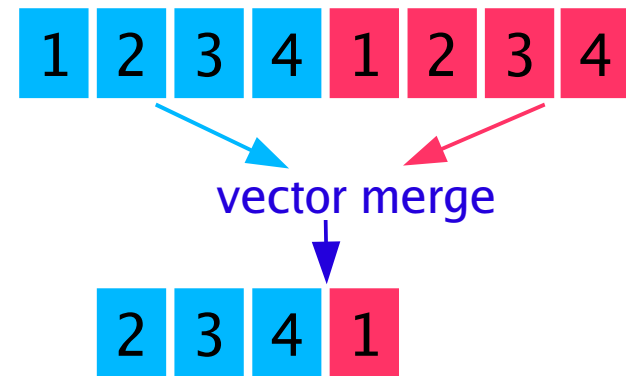
Barrier



Uni-directional

Vector Instruction Set on PPE and SPE

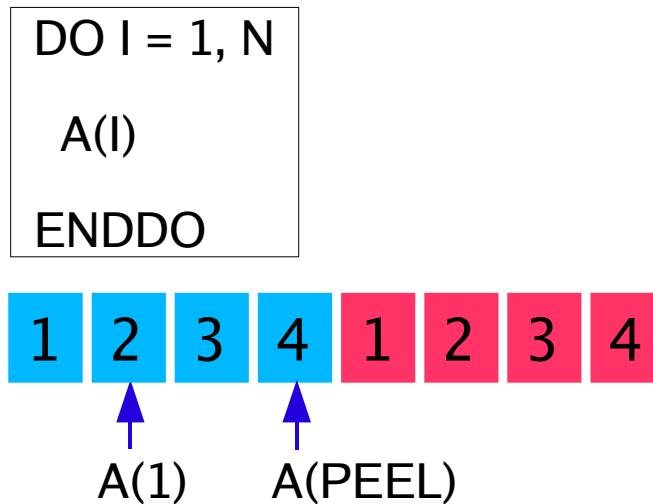
- Altivec on PPE, new SIMD on SPE
 - Limited vector length (16 bytes)
 - Contiguous memory access
 - Data alignment constraint
- Compilation challenges
 - Vectorization
 - Have as many array references aligned as possible
- Vectorization
 - Basic block based
 - S. Larsen and S. Amarasinghe
 - Loop based
 - Intel, IBM, VAST/Altivec, GNU compiler
 - many work on conventional vector machines
 - Our approach: loop based



Vector Instruction Set on PPE and SPE .

- Data alignment

- Related work: Loop peeling



```
PEEL = AlignSize(A(1))
```

```
DO I = 1, PEEL
```

```
  A(I)
```

```
ENDDO
```

```
DO I = PEEL+1, N-VL+1, VL
```

```
  A(I:I+VL-1)
```

```
ENDDO
```

- Related work: pipelined vector load and store
 - A. Eichenberger, P. Wu and K. O'Brien
- Related work: vectorized scalar replacement
 - J. Shin, J. Chame and M. Hall
- Our approach: additional method of loop alignment

Outline

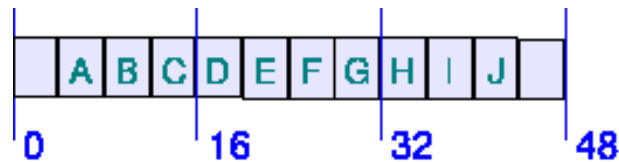
- Parallelization and vectorization
- **Data movement**
- Experimental results
- Conclusion and future work

Related Work on Data Movement

- Explicit data transfer for SPE's LS
- Related work: software cache
 - Eichenberger et al, IBM compiler, cache lookup code before each reference, 64KB, 4-way, 128bytes cache line
- Related work: loop transformations
 - Eichenberger et al, loop blocking, prefetching
- Related work: array copying
 - Lam et al, reduce cache conflict misses in blocked loop nests
 - Temam et al, cost-benefit models for array copying
 - Yi, general algorithm with heuristics for cost-benefits analysis
 - ATLAS
- Related work: software prefetching
 - Callahan et al, Mowry et al, prefetch analysis, prefetch placement

Data Movement

- Multi-buffering to hide the latency of DMA transfers
 - All loop invariants are passed to SPE once at the beginning
 - Remaining array references are partitioned into reference groups
 - For each group leader, insert DMA data transfers at the innermost loop that the reference is variant, strip-mine the loop properly
 - For other references, perform vectorized scalar replacement to get value from the group leader
- Data alignment
 - Vector offset: last 4 bits of source and destination address be same
 - Naturally aligned
 - Get: over-fetch
 - Put: loop peeling on PPE
 - Cacheline alignment



DMA(&A, 4 bytes)

DMA(&B, 8 bytes)

DMA(&D, 16 bytes)

DMA(&H, 8 bytes)

DMA(&J, 4 bytes)

Outline

- Parallelization and vectorization
- Data movement
- **Experimental results**
- Conclusion and future work

Experiments

- 3.2GHz CELL, 1GB memory, xlc compiler, f2c tool

- Stencils

$$\text{1d: } B(2:N-1) = (A(1:N-2) + A(3:N) + C(2:N-1)) * 0.34$$

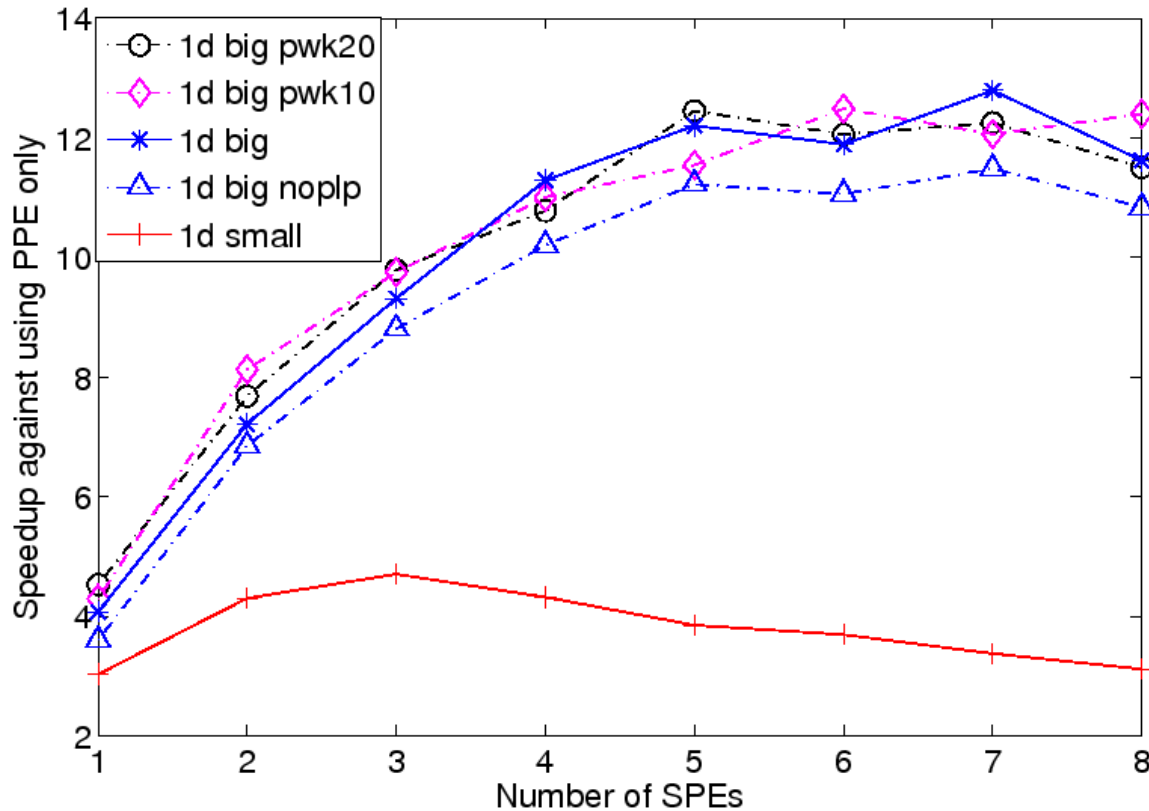
$$\begin{aligned} \text{2d: } A(2:N-1,2:M-1) = & A(2:N-1,2:M-1) + (B(1:N-2,2:M-1) + B(3:N,2:M-1) \\ & + B(2:N-1,1:M-2) + B(2:N-1,3:M)) * 0.25 \end{aligned}$$

$$\text{anti: } A(2:N-1) = (A(3:N) + B(2:N-1) + C(2:N-1)) * 0.34$$

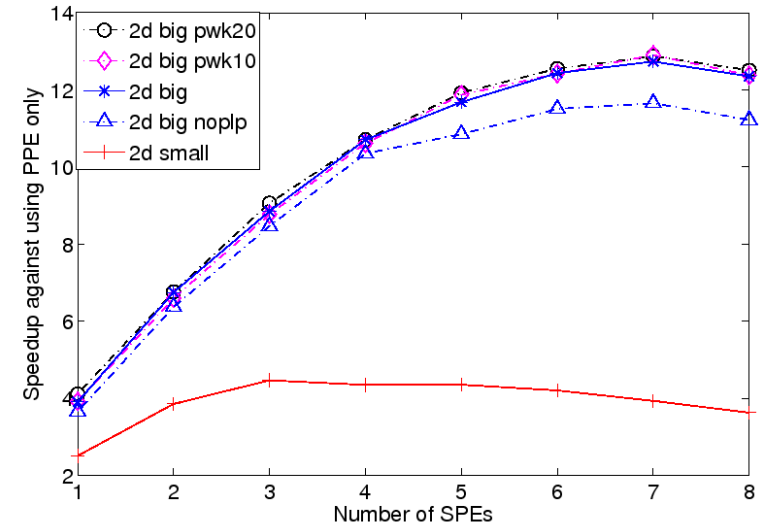
$$\begin{aligned} \text{antitmp: } T(1:N-2) = & (A(3:N) + B(2:N-1) + C(2:N-1)) * 0.34 \\ A(2:N-1) = & T(1:N-2) \end{aligned}$$

- Swimfloat: swim in SPEC benchmark with float type
 - 10 loop nests: L0...9, {L0,1,2,7 once}, {L4,6,9, 1d boundary copy}
 - L3: 2 level loop, 4 stencil statements; L5: 2 level loop, 3 stencil statements
 - L8: 2 level loop, 3 stencil statements, 3 array copy statements
- Mgridfloat: mgrid in SPEC benchmark with float type
- Optimizations: vectorized scalar replacement, software pipelined vector load/store, loop peeling on PPE, work load on PPE

Experiments .



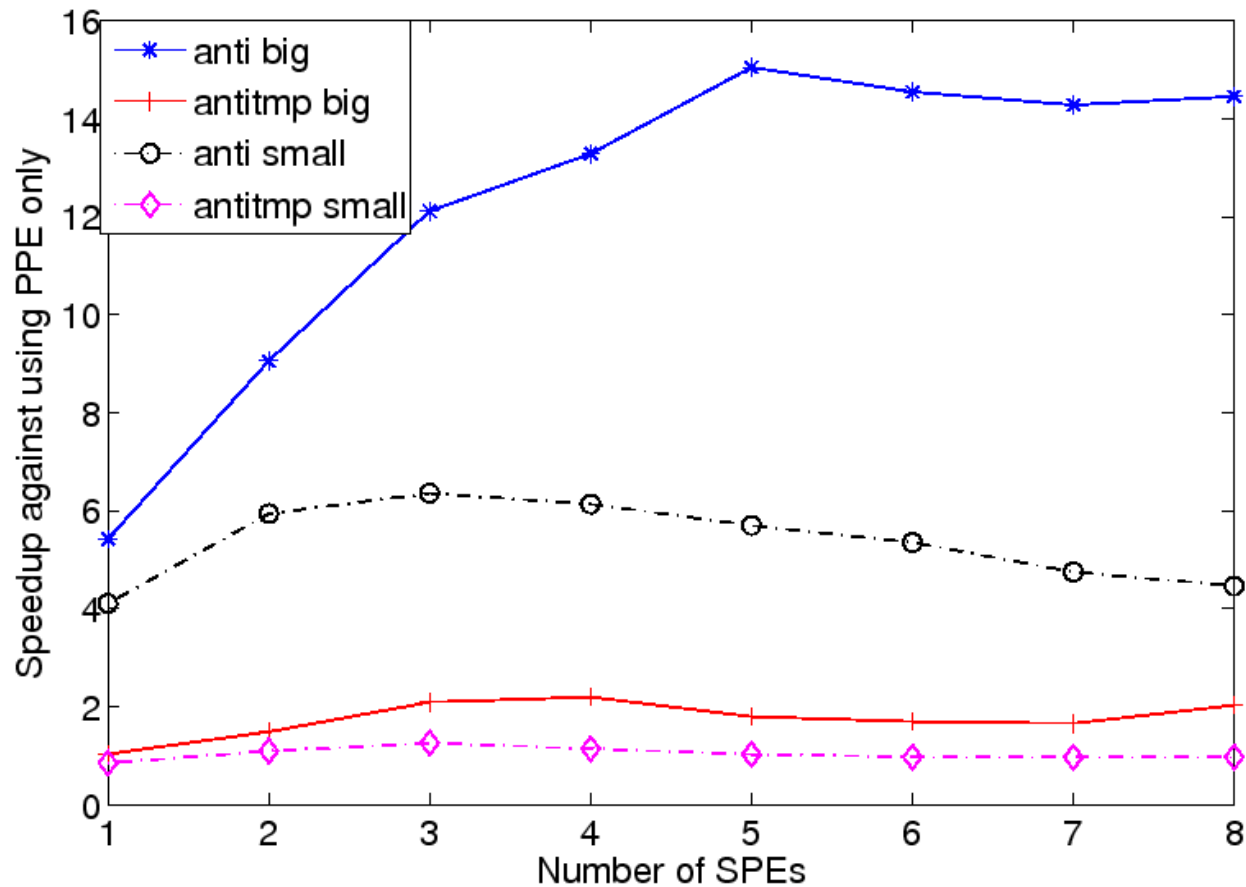
Small: N= 7555333 Big: N= 44222111



Small: M=N= 2955 Big: M=N= 7255

- SPE faster than PPE (pipeline length, OS)
- Small vs big problem size (thread cost, enough work on each PE)
- Loop peeling for DMA data alignment

Experiments ..



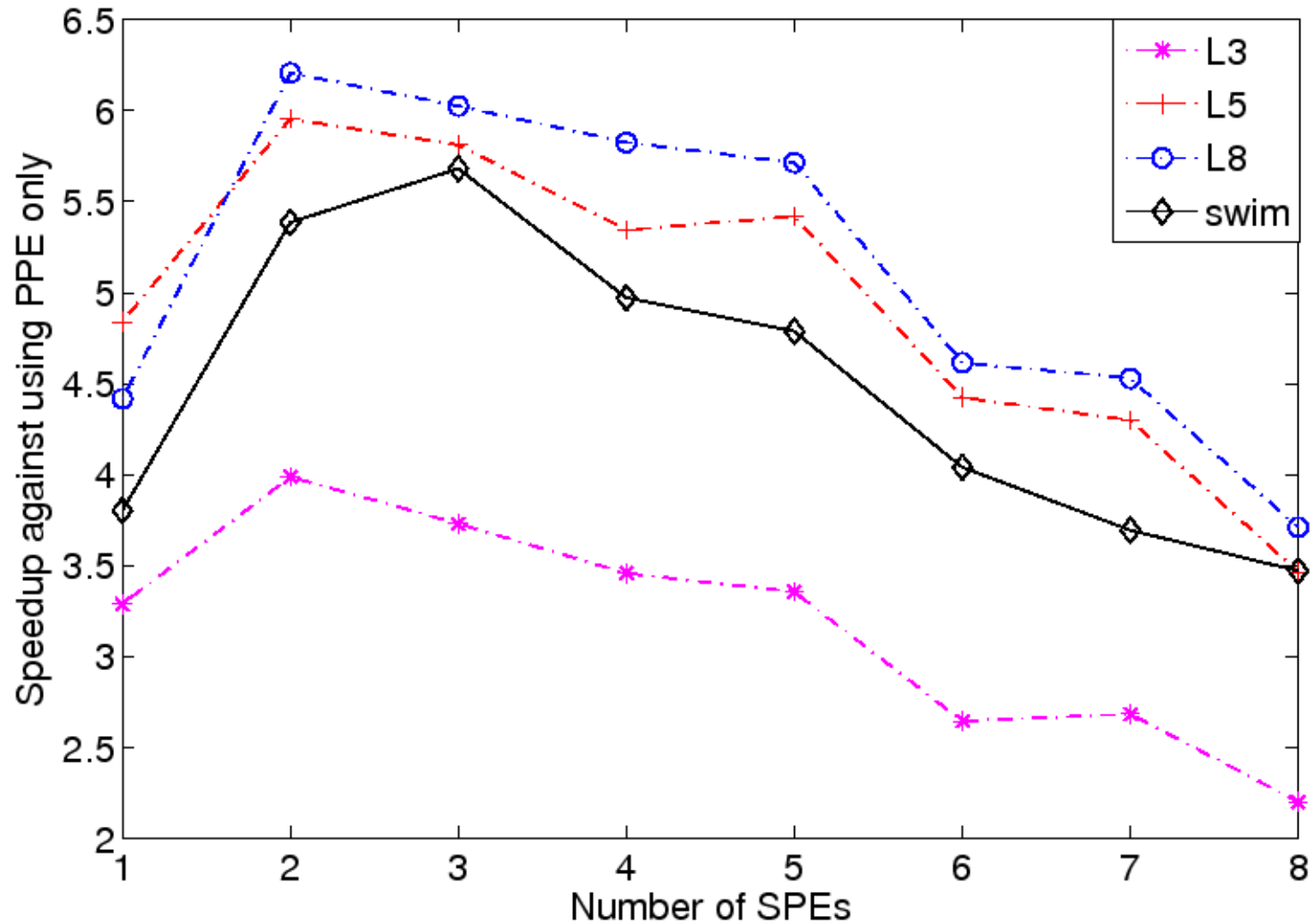
Small: N= 755333 Big: N= 33222111

- Reduced temporary array allocation leads to reduced data transfer

Experiments ...

Swim problem size: 1335x1335, 1200 iterations

(L3,5,8 collected with 20 iterations)



- Individual tuning of each loop nest
- Problem size

Conclusion

- A dependence-based automatic code generation scheme
 - No parallelism directives/pragmas
 - Parallelization, vectorization, multi-buffering data transfer, loop peeling on PPE, workload on PPE, data alignment on PEs, temporary allocation reduction, reduced lazy loop alignment, synchronization
- A source-to-source compiler
 - Procedure-outlines Fortran loop nest and rewrites in C to utilize parallelization libraries and vector data types and intrinsics
 - Adds support to vector instruction set on SPE
- Compilers developed for short vector processors can be extended for multi-core processors

Ongoing and Future Work

- Performance study
 - SPE and PPE activities (categorized performance)
 - OProfile using on-chip hardware counters (SPE events)
- Thread cost reduction
 - Loop fusion
 - Fusing parallel code regions
- More optimizations for memory hierarchy performance on CELL
 - Computation to data transfer ratio
 - Loop tiling, loop fusion
 - On-chip data reuse inside each SPE
- More parallelization schemes
 - Number of SPEs used for individual loop nest
 - Mix of loop nests on SPEs
 - On-chip data reuse among SPEs

Ongoing and Future Work .

- Load balancing
 - SPE and PPE
- Loop fusion and synchronization
 - Parallelism vs data reuse
 - Cost of on-chip synchronization
- Longer term
 - Multiple CELL chips
 - Programming models
 - Memory affinity, numactl
 - CELL chips and other chips
 - Heterogeneous computing

Thank you!