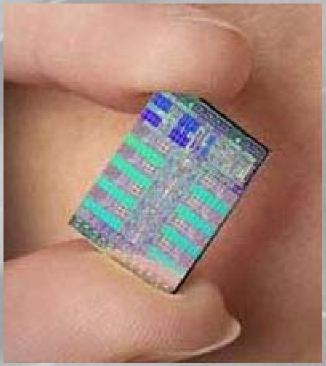
IBM T. J. Watson Research

CellSort: High Performance Sorting on the Cell Processor



Buğra Gedik, Rajesh R. Bordawekar, Philip S. Yu
IBM Thomas J. Watson Research Center

VLDB 2007

Summary of Results

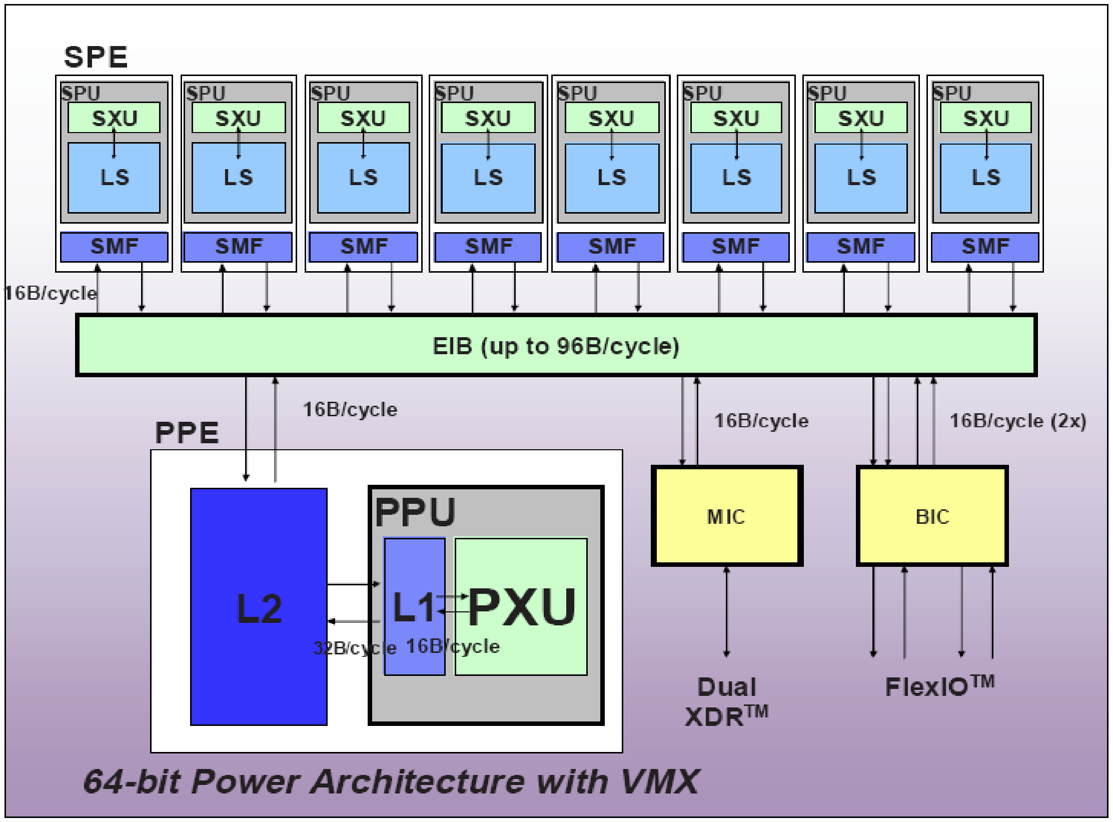
- Designed and implemented CellSort
  - A high performance sorting algorithm for Cell
  - Based on *distributed bitonic merge* with *SIMDized bitonic sorting kernel*
- Our results reported in this paper show that:
  1. SIMDized bitonic sort kernels are superior to quick sort kernels on Cell. The same does not hold for SSE-enhanced bitonic sort on Intel Xeons.
  2. Distributed in-core sort is highly scalable (SPEs). 16SPEs can sort floats up to 10 x faster, compared to quick sort on dual-core 3.2Ghz Intel Xeon.
  3. CellSort becomes memory I/O bound as we go out-of-core. Yet, 16 SPEs can sort 0.5GB of floats up to 4 x faster compared to dual-core 3.2Ghz Intel Xeon.
  - Can sort 0.5GB of ints/floats in ~4 seconds

Outline

- ➔ Cell Architecture Overview
- Sorting on the Cell: The Basics
- Sorting on the Cell: The Algorithms
  - Power-of-two Periodic Bitonic Sort
  - SIMDized Bitonic Sort Kernel
  - Distributed In-core Bitonic Sort
  - Distributed Out-of-core Bitonic Sort
- Experimental Results and Analysis
- Related Work and Future Plans
- Conclusions

The Sony-Toshiba-IBM CELL Processor

- Heterogeneous multi-core architecture
  - 1 Power Processor Element (PPE) for control tasks
  - 8 Synergistic Processor Elements (SPE) for data intensive processing
  - High-bandwidth Bus
- Each SPE has
  - Synergistic Processor Unit (SPU)
  - Synergistic Memory Flow (SMF)
  - 256 KB Local Memory Store
- Lots of parallelism!
  - 128-bit SIMD per SPE
  - Two-way ILP per SPE
  - Combination of short-vector SIMD, shared and distributed memory parallel processing



Issues in Application Development on Cell

- Every cell program potentially a parallel out-of-core application (here “core” refers to main-memory, not disk)
  - Parallelize over multiple SPEs
  - Vectorize using SIMD instructions
  - Asynchronous DMAs to hide latencies
- Weak front-end processor
  - An ideal Cell application runs only on SPEs
- Other restrictions
  - No recursive functions
  - No branch prediction hardware
  - Everything must be 128-bit aligned
- Applications should
  - Minimize function calls
  - Unroll loops wherever possible
  - Avoid comparison-based codes as they cannot fully exploit instruction-level parallelism

Outline

- Cell Architecture Overview

➔ Sorting on the Cell: The Basics

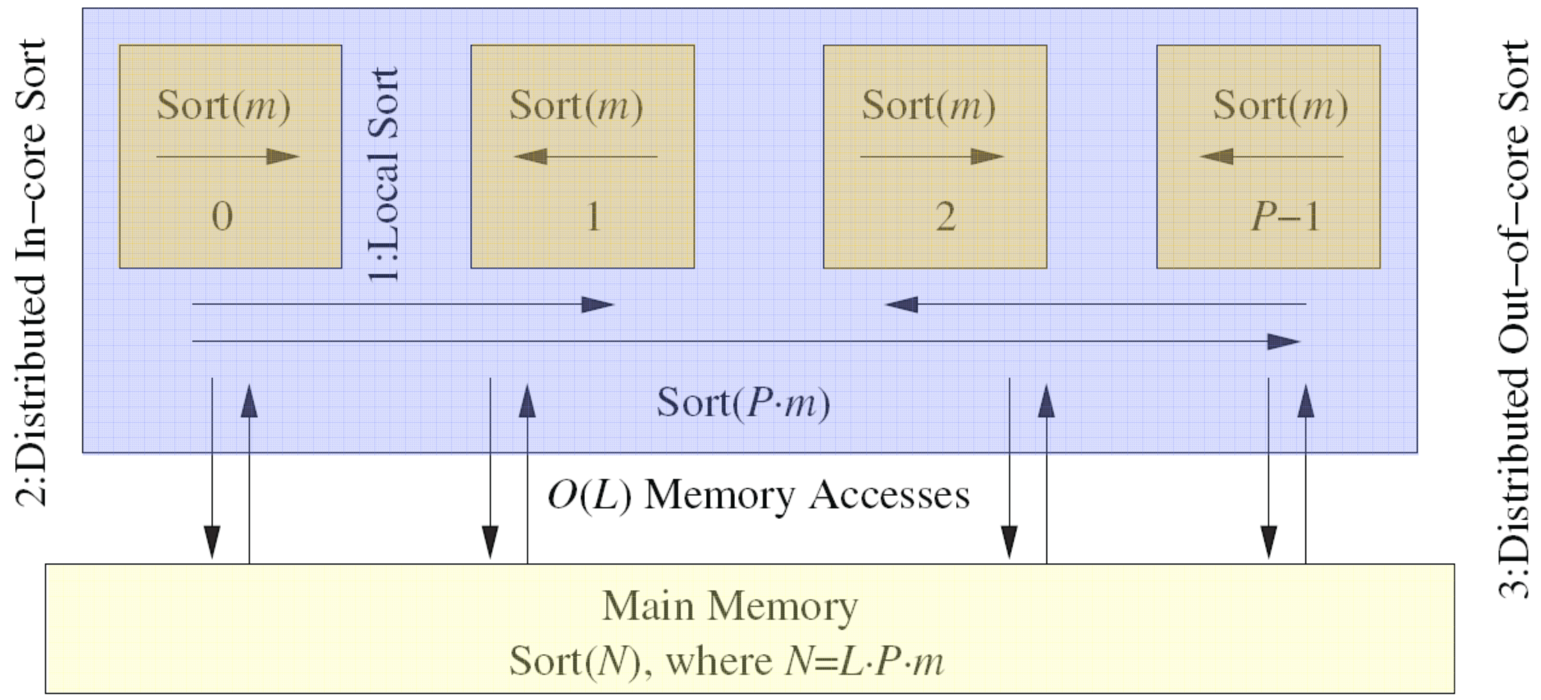
- Sorting on the Cell: The Algorithms
  - Power-of-two Periodic Bitonic Sort
  - SIMDized Bitonic Sort Kernel
  - Distributed In-core Bitonic Sort
  - Distributed Out-of-core Bitonic Sort
- Experimental Results and Analysis
- Related Work and Future Plans
- Conclusions

Sorting on the Cell: The Basics

- Aim: Out-of-core Distributed SIMD sort-merge algorithm
- CellSort follows a three-tiered approach
  1. **Single-SPE Local Sort**: An efficient per-SPE sorting kernel
     - We pick bitonic sort: no unpredictable branches, effectively SIMDizable
  2. **Distributed In-core Sort**: Inter-SPE distributed merge after local sorts
     - Use shared-address space + async. DMA capabilities, exploit high inter-SPE bandwidth
  3. **Distributed Out-of-core Sort**: Memory-based distributed merge after in-core sorts
     - Similar to in-core sort, but uses main memory and different communication patterns
     - Less bandwidth available to memory, compared to inter-SPE bandwidth

- Three levels of strip-mining
  1. Need to work with DMA access limits (16 KB per access)
  2. Need to work with small local stores (can hold up to 128 KB of data)
  3. Need to work with small collective local stores (can hold up to *P* *128 KB of data)
- We employ bitonic sort in all three tiers, SIMDized at all levels

Out-of-core Distributed SIMD Merge-Sort



- N = number of items to sort
- m = number of items that fit into a local store
- P = number of processors
- L = number of in-core runs

Outline

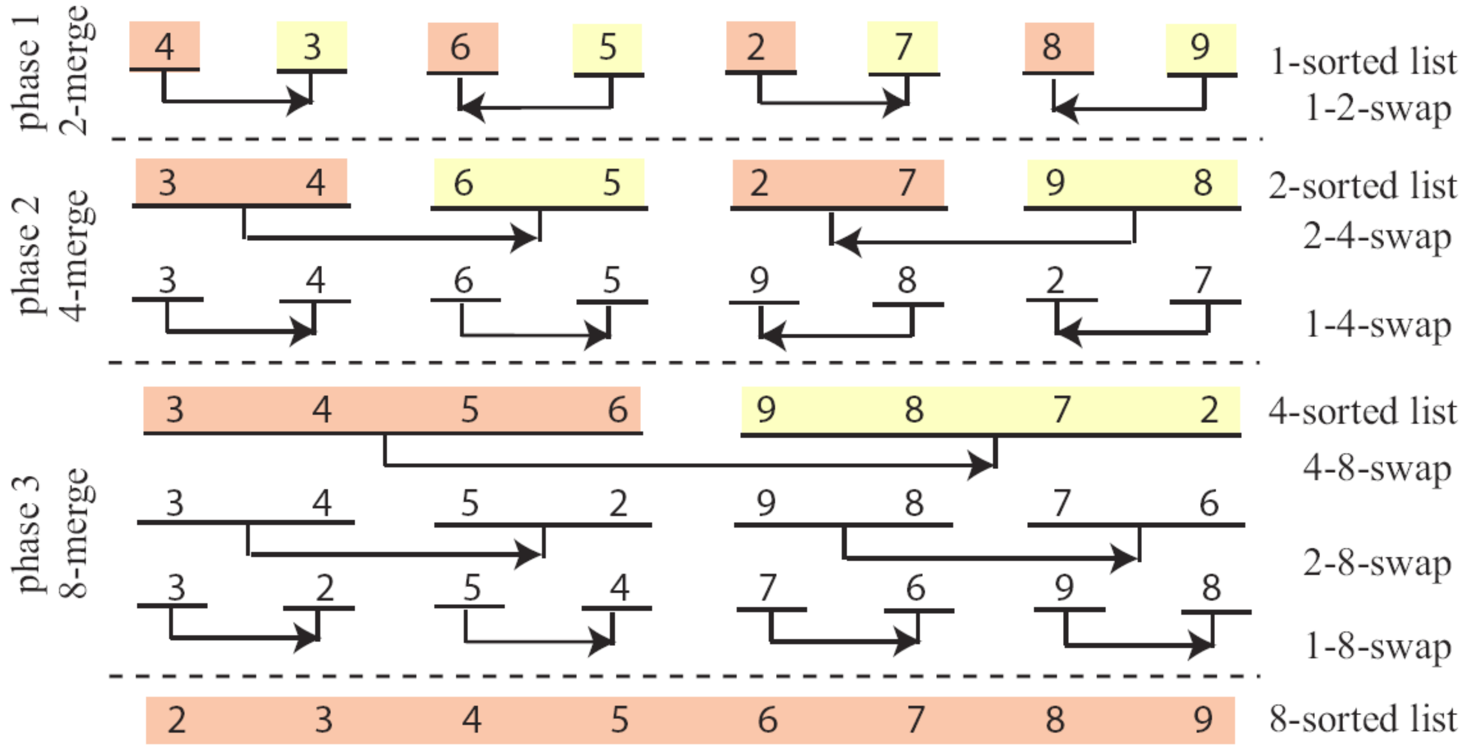
- Cell Architecture Overview
- Sorting on the Cell: The Basics
- ➔Sorting on the Cell: The Algorithms
  - ➔Power-of-two Periodic Bitonic Sort
  - SIMDized Bitonic Sort Kernel
  - Distributed In-core Bitonic Sort
  - Distributed Out-of-core Bitonic Sort
- Experimental Results and Analysis
- Related Work and Future Plans
- Conclusions

Periodic Bitonic Sort Network (PBSN)

- An $\Theta(n \lg^2 n)$ sorting algorithm
  - Best-case complexity is the same as the worst-case complexity
- Advantages
  - Fixed computation pattern irrespective of the input data values
    - Easy to unroll loops, avoid branches, make use of ILP
  - Accesses contiguous memory locations, good for SIMDization
  - Ideal for SIMDization using vector shuffle and compare operators
  - Fully parallelizable

    $\Theta(\frac{N}{P} \lg^2 N)$ complexity with P processors
- Disadvantages
  - Sub-optimal asymptotic complexity for the sequential case

Example of sorting 8 integers



- lg*N* phases, *i* th phase is called *k*-merge ($k=2^i$), produces a *k*-sorted list
- *k*-sorted list = every *k*-item block is sorted, in alternating directions
- A *k*-merge phase has lg*k* steps, *i* th step is called *j*-*k*-swap ($j=k/2^i$)
- In a *j*-*k*-swap, consecutive *j*-item blocks are *compare and swapped*, the compare-and-swap order is switched after each *k*-item block

Outline

- Cell Architecture Overview
- Sorting on the Cell: The Basics

➔Sorting on the Cell: The Algorithms

  – Power-of-two Periodic Bitonic Sort

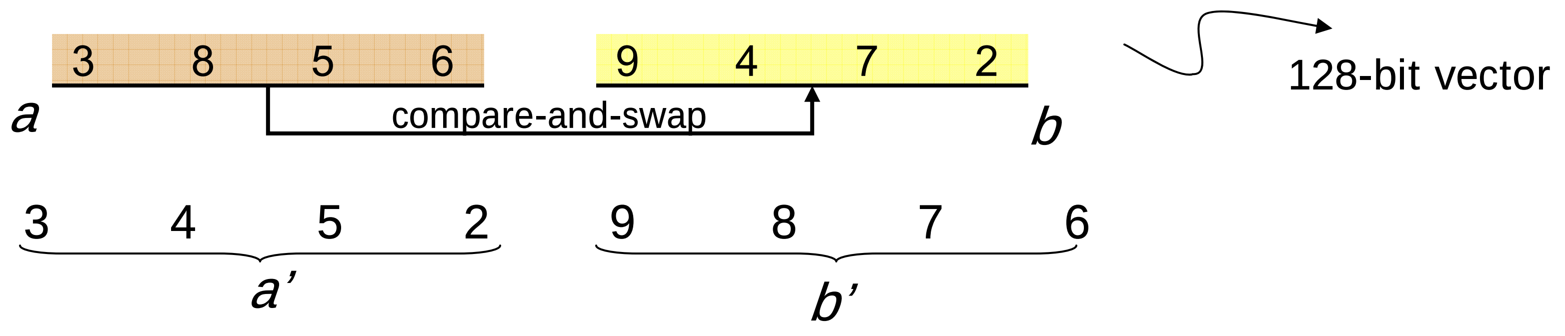
  ➔SIMDized Bitonic Sort Kernel

  – Distributed In-core Bitonic Sort

  – Distributed Out-of-core Bitonic Sort

- Experimental Results and Analysis
- Related Work and Future Plans
- Conclusions

SIMDizing the compare-and-swaps

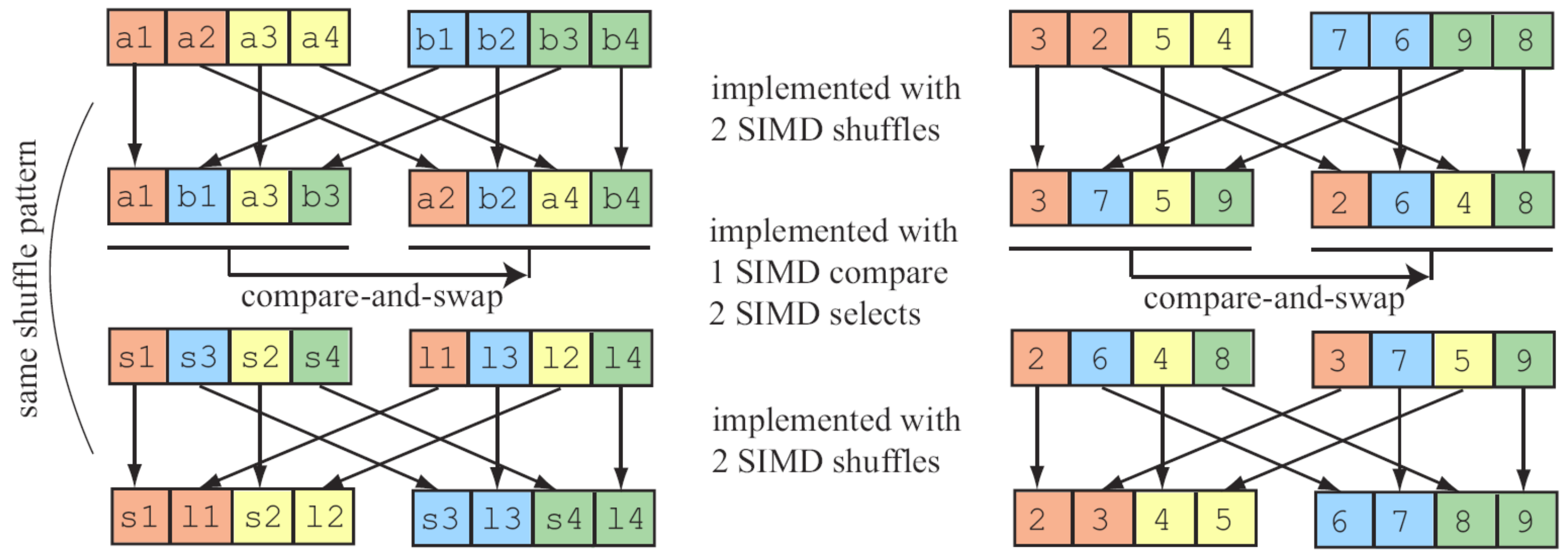


- SIMD comparison instruction *t* = cmpgt(*a*, *b*) used to create a mask *t*
- SIMD select instruction *a'* = select(*a*, *b*, *t*) used to yield the smaller of the corresponding items of the two vectors (lower half of the compare-and-swap)
- SIMD select instruction *b'* = select(*b*, *a*, *t*) used to yield the larger of the corresponding items of the two vectors (upper half of the compare-and-swap)
- In total, the SIMD implementation requires 1 comparison and 2 select instructions to complete the compare-and-swap. Total of 3 SIMD instructions.
- Furthermore, there are no branches involved !!!

SIMDizing the compare-and-swaps, special cases

- When j is smaller than 4 in a j-k-swap, that is ($j \in \{1, 2\}$):
  - Blocks to be compare-and-swapped fall into the boundaries of a single vector
  - We call these *special cases*, there are 5 of them
    - $<j=2, k=4>$, $<j=2, k \geq 8>$, $<j=1, k=2>$, $<j=1, k=4>$, and $<j=1, k \geq 8>$
  - These steps require different SIMDization strategies
  - Why do we care about these special cases?
    - For local sorts (max of 128KB of data), the fraction of j-k-swap steps with $j < 4$ constitute at least 18.75% of the total
    - When implemented in a scalar fashion, they easily dominate the overall cost
- We developed individual SIMDization techniques for each of the five special cases
  - These involve SIMD *shuffle* operations
  - Two examples follow: $<j=1, k \geq 8>$ and $<j=2, k=4>$

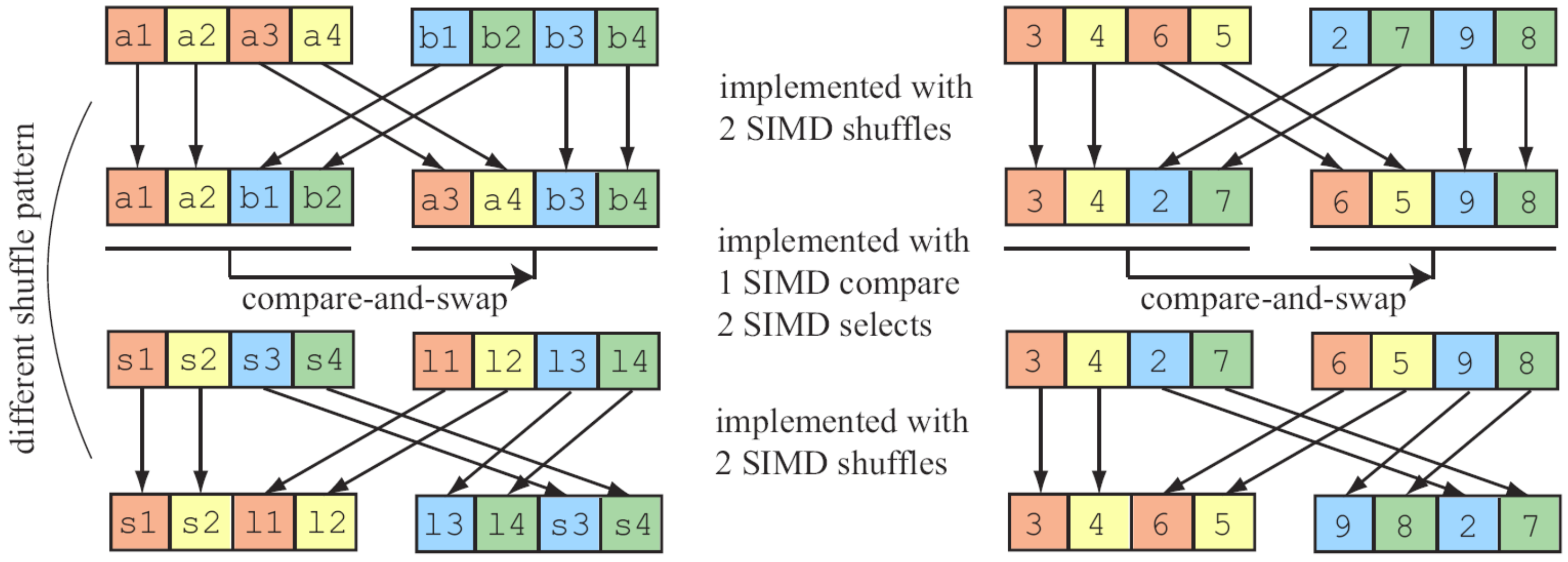
SIMDized compare-and-swaps: <j=1,k≥8>



```
s1 = min(a1, a2)  s2 = min(a3, a4)  s3 = min(b1, b2)  s4 = min(b3, b4)
l1 = max(a1, a2)  l2 = max(a3, a4)  l3 = max(b1, b2)  l4 = max(b3, b4)
```

- Total of 7 SIMD instructions, 4 more than the regular case

SIMDized compare-and-swaps: <*j*=2,*k*=4>



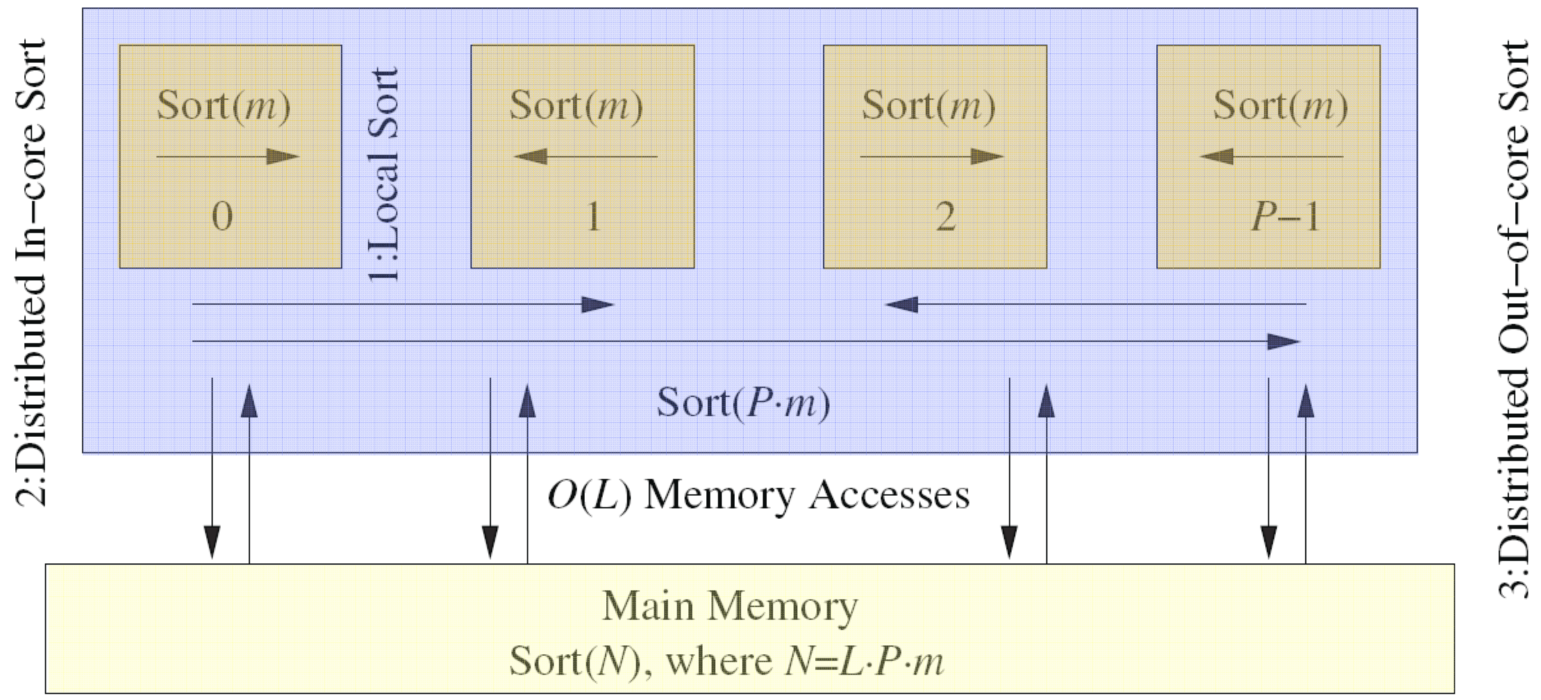
```
s1 = min(a1, a3)  s2 = min(a2, a4)  s3 = min(b1, b3)  s4 = min(b2, b4)
l1 = max(a1, a3)  l2 = max(a2, a4)  l3 = max(b1, b3)  l4 = max(b2, b4)
```

- Total of 7 SIMD instructions, 4 more than the regular case
- Note that the shuffle patterns do not match

Outline

- Cell Architecture Overview
- Sorting on the Cell: The Basics
- ➔ Sorting on the Cell: The Algorithms
  - Power-of-two Periodic Bitonic Sort
  - SIMDized Bitonic Sort Kernel
  - ➔ Distributed In-core Bitonic Sort
  - Distributed Out-of-core Bitonic Sort
- Experimental Results and Analysis
- Related Work and Future Plans
- Conclusions

Distributed SIMD Merge-Sort

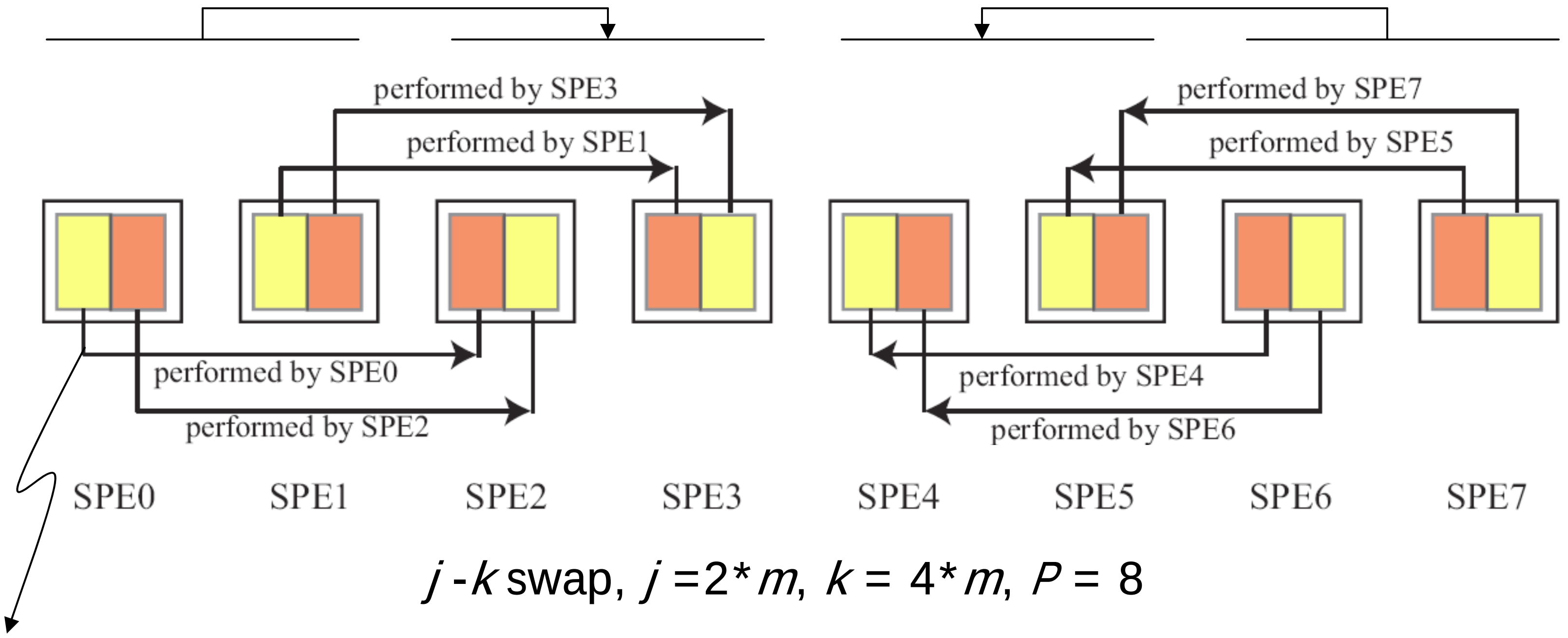


- N = number of items to sort
- m = number of items that fit into a local store
- P = number of processors
- L = number of in-core runs

Distributed In-core Bitonic Sort

- Perform *P* local sorts using the bitonic sorting kernel
- Use in-core bitonic merge to yield the final result
- The key issue is to implement distributed *j*-*k*-swaps of the in-core bitonic merge, in an efficient manner
  - Data transfer patterns
    - Make sure *half* of the processed data is *always local*
  - Data transfer latencies
    - Use double-buffering and async. DMAs to hide latencies
  - Synchronization
    - Most of the time small barriers are sufficient, avoid global barriers
- Let's see an example to illustrate these techniques

Distributed in-core *j*-*k*-swap



j-k swap, $j = 2*m$, $k = 4*m$, $P = 8$

Remote data (red) is brought into SPE0's local store one DMA block at a time, using double buffering

Red denote remote data, yellow denote local data

- After an in-core distributed *j*-*k*-swap, each consecutive $2*j/m$ SPEs do a barrier, no global barriers are needed

Outline

Cell Architecture Overview

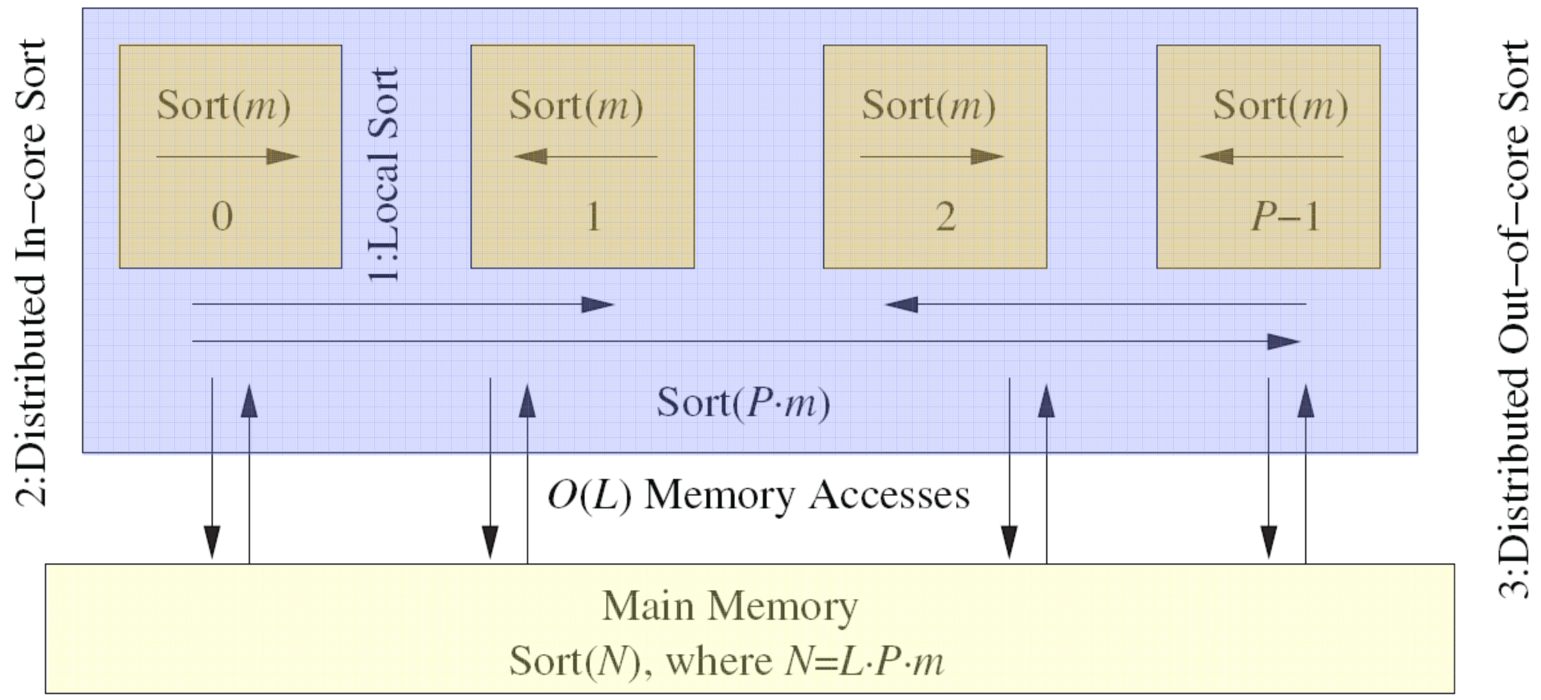
- Sorting on the Cell: The Basics

➔ Sorting on the Cell: The Algorithms
  - Power-of-two Periodic Bitonic Sort
  - SIMDized Bitonic Sort Kernel
  - Distributed In-core Bitonic Sort

  ➔ Distributed Out-of-core Bitonic Sort

- Experimental Results and Analysis
- Related Work and Future Plans
- Conclusions

Distributed SIMD Merge-Sort

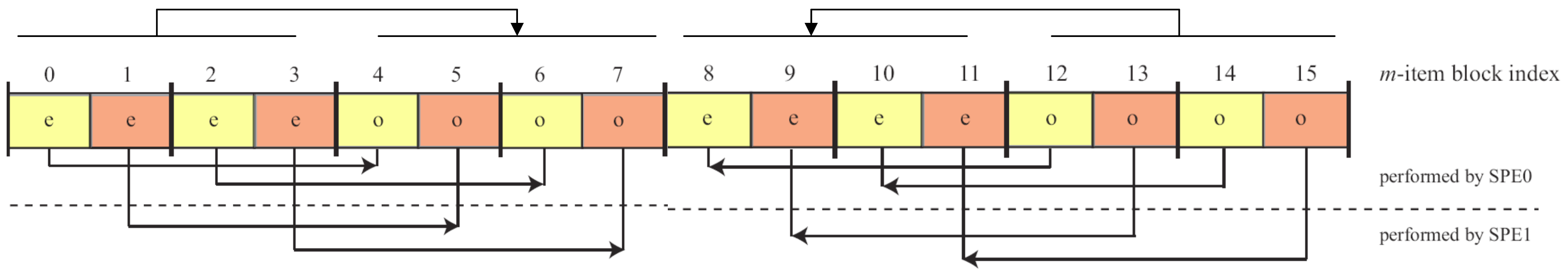


- N = number of items to sort
- m = number of items that fit into a local store
- P = number of processors
- L = number of in-core runs

Distributed Out-of-core Bitonic Sort

- Perform *L* runs using the distributed in-core sort
- Use out-of-core bitonic merge to yield the final result
- The key issue is to implement the distributed *j*-*k*-swaps of the out-of-core bitonic merge, in an efficient manner
  - Data transfer patterns
    - Different than the in-core sort case, all data is remote and in memory
    - Two times the bandwidth of the in-core case is needed, whereas only ~1/8th of the cross-SPE bandwidth is available
  - Synchronization
    - Barriers are not needed between *j*-*k*-swaps, but only after *k*-merges
  - Data transfer latencies
    - Use double-buffering and async. DMAs to hide latencies (same as earlier)
- Let’s see an example to illustrate these techniques

Distributed out-of-core *j*-*k*-swap



j-*k* swap, $j = 4*m$, $k = 8*m$, $P = 2$, $L=8$

- SPE 0 always processes the yellow blocks, and SPE 1 red blocks
- Important: The coloring is fixed over changing *j* values
  - Thus no barriers are needed between *j*-*k*-swaps
- ‘e’ (even) blocks are compare-and-swapped with ‘o’ (odd) blocks
- The ‘e’ and ‘o’ labels are different for different *j* values

Outline

- Cell Architecture Overview
- Sorting on the Cell: The Basics
- Sorting on the Cell: The Algorithms
  - Power-of-two Periodic Bitonic Sort
  - SIMDized Bitonic Sort Kernel
  - Distributed In-core Bitonic Sort
  - Distributed Out-of-core Bitonic Sort

➔ Experimental Results and Analysis

- Related Work and Future Plans
- Conclusions

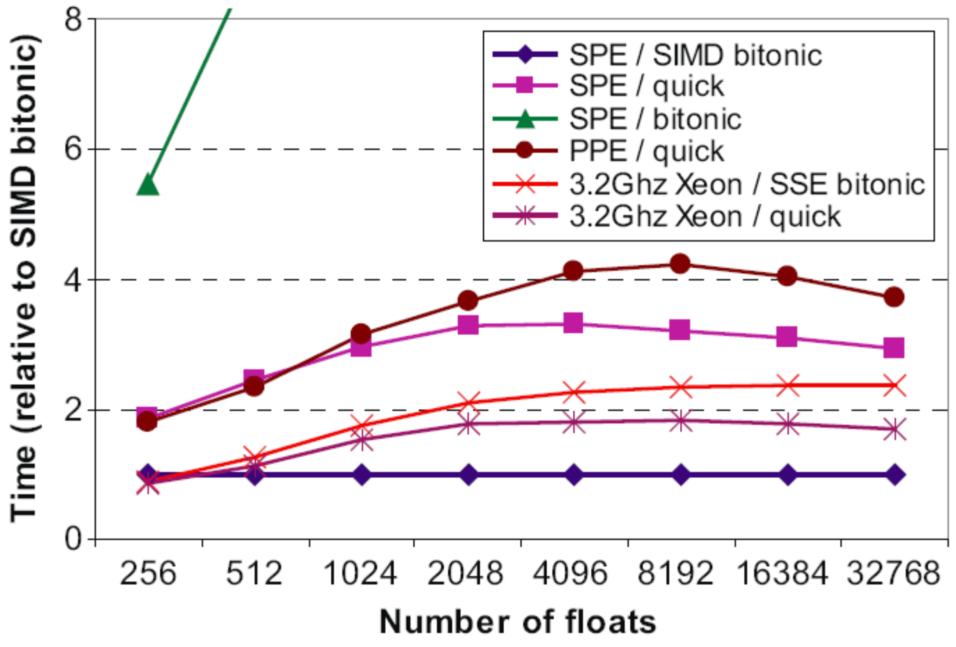
Evaluation Setup

- Distributed bitonic sort using up to P = 16 SPEs (3.2GHz) in an IBM QS20 Cell blade
  - Alternate sorting kernels: • basic bitonic, • SIMDized bitonic, • shell sort, • quick sort
- Single thread quick sort on the PPE (3.2GHz)
- Single and dual thread (using OpenMP) quick sort on 3.2GHz Intel Xeon
- Quick sort on 3GHz Intel Pentium 4
- Single-thread SSE-enhanced bitonic sort on aforementioned Intels
- Sort codes for the Intels were compiled using icc compiler with optimizations on
- Sort codes for the Cell were compiled using the gnu tool-chain
- Maximum number of items that can be sorted
  - using local sort is m = 32K (**128KBs** of data)
  - using in-core sort is $N = P * m$ = 16 * 32K = 512K (**2MBs** of data)
  - using out-of-core sort is 128M number of items (**0.5GB** of data), since the memory available to us in our test machine was 1GB
- In-core and local sorts include the time to transfer items to/from the main memory

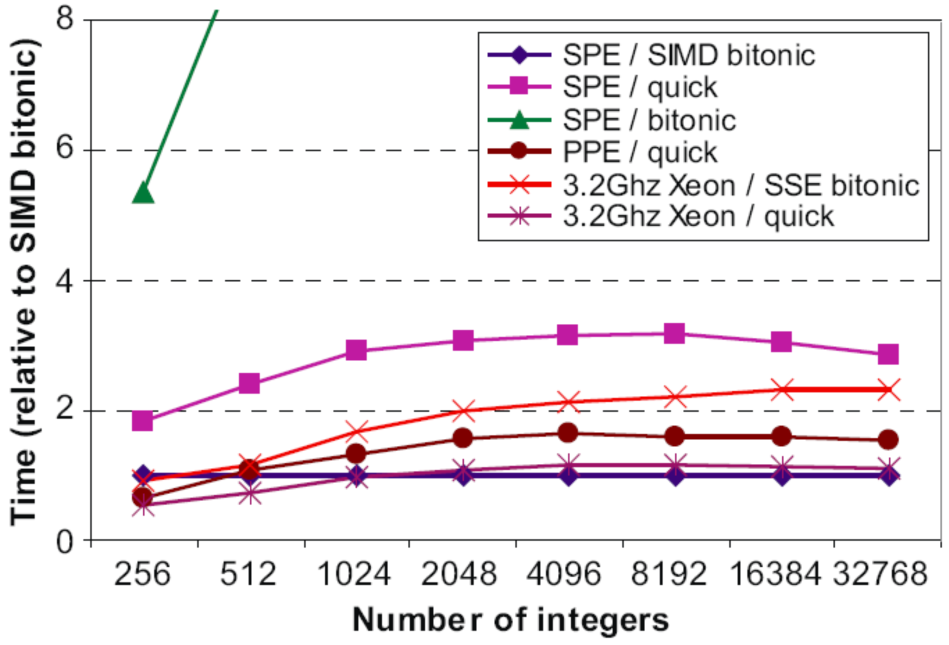
Local Sort Performance

# items 32K	SIMD bitonic	quick	no-shuffle bitonic	basic bitonic	PPE quick
ints	.0025 secs	.0073	.0178	.0550	.0038
floats	.0025 secs	.0073	.0178	.0550	.0090

Single-SPE local sort performance



Local sort, floats

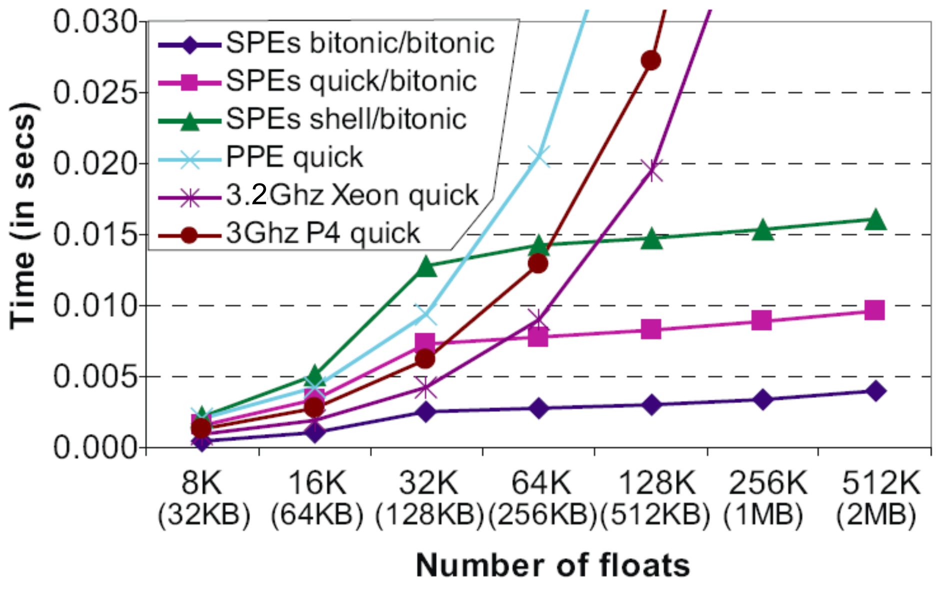


Local sort, integers

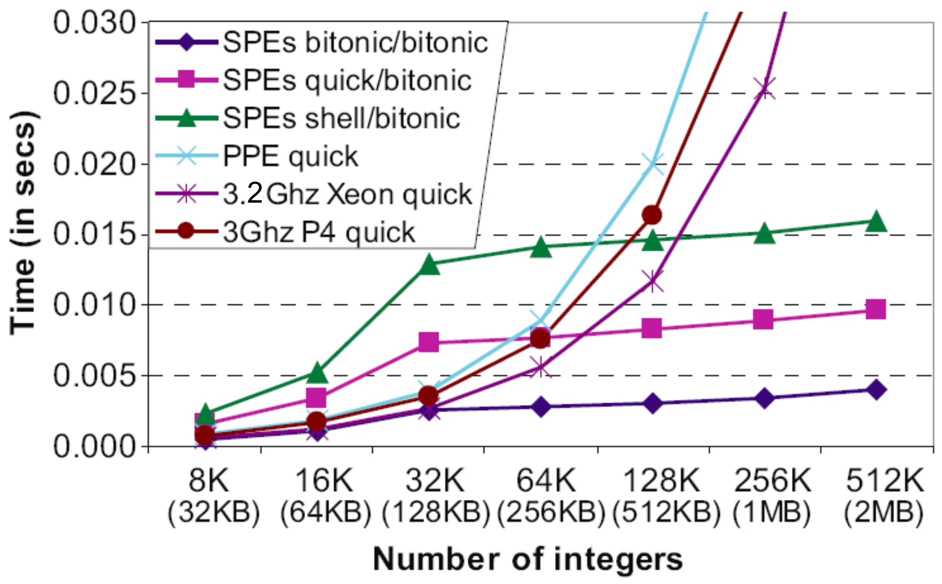
Local Sort Cycle Statistics

metrics	basic bitonic	SIMD bitonic	quick sort
CPI (cycles per instruction)	2.26	1.05	3.39
Single issued cycles	28.9%	42.5%	20.5%
Double issued cycles	3.6%	22.2%	2.6%
Stalls due to branch	40.1%	22%	40.3%
Stalls due to dependency	22.5%	10.8%	33.8%
Other (including nops)	3.7%	2.5%	2.8%

In-core Sort Performance

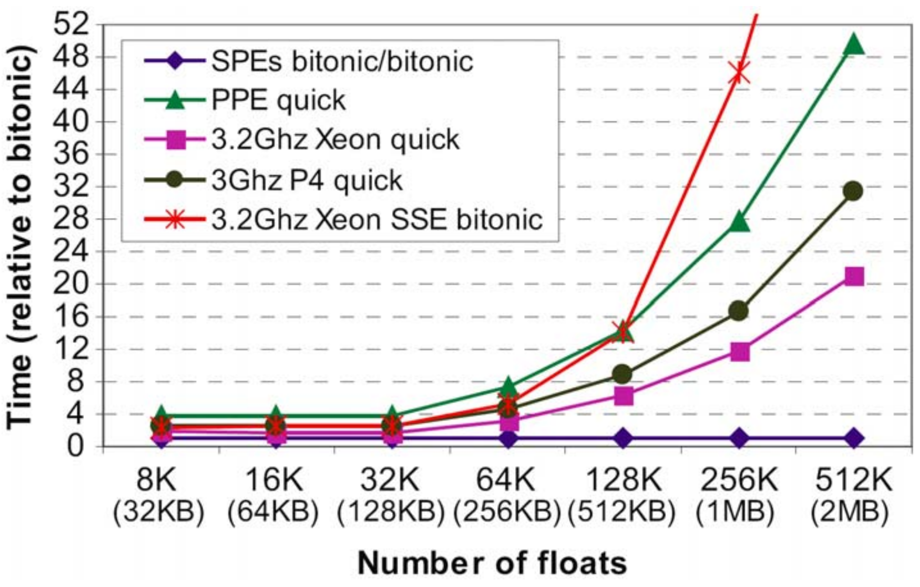


In-core sort, floats

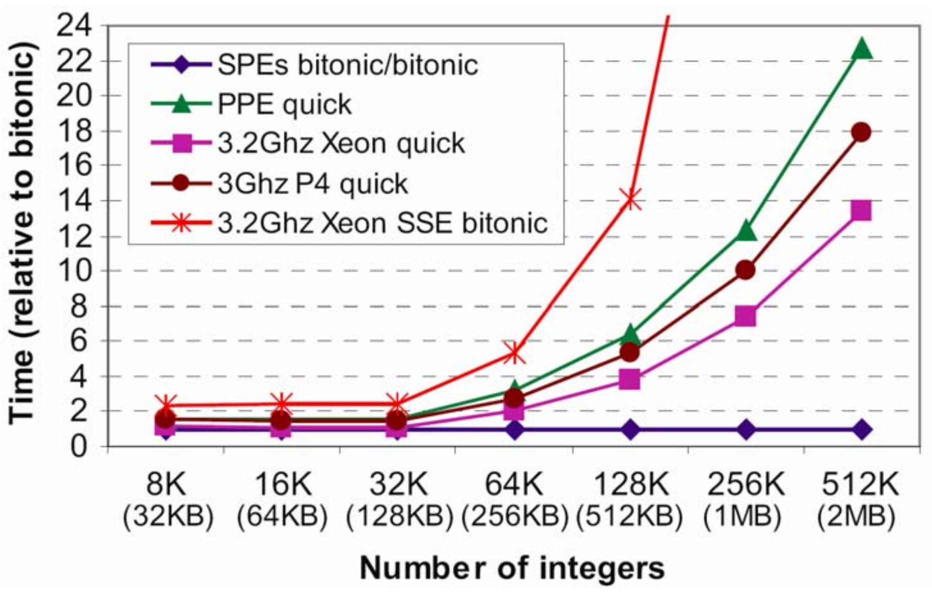


In-core sort, integers

Normalized In-core Sort Performance (lower is better)

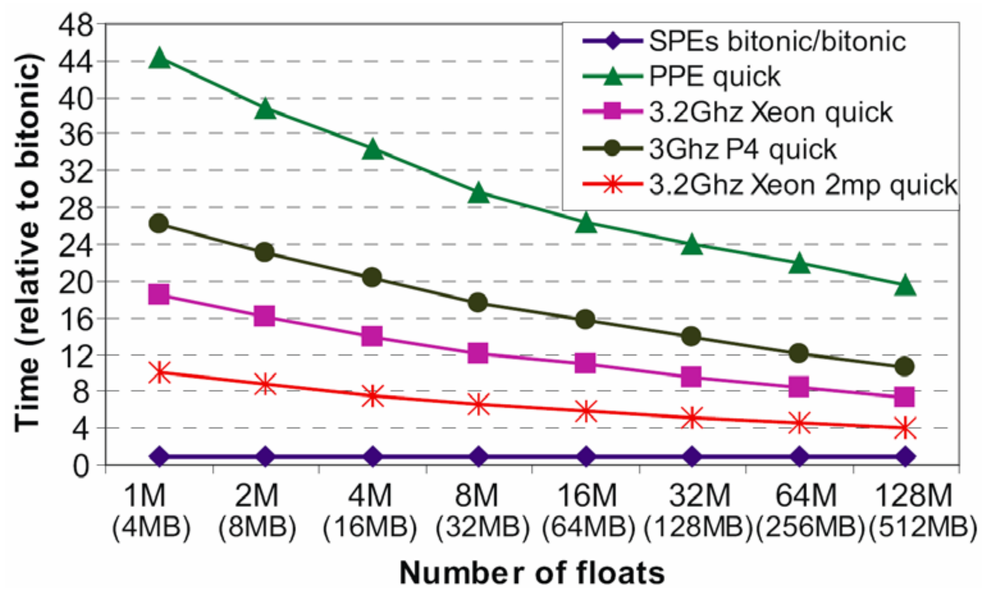


In-core sort, floats



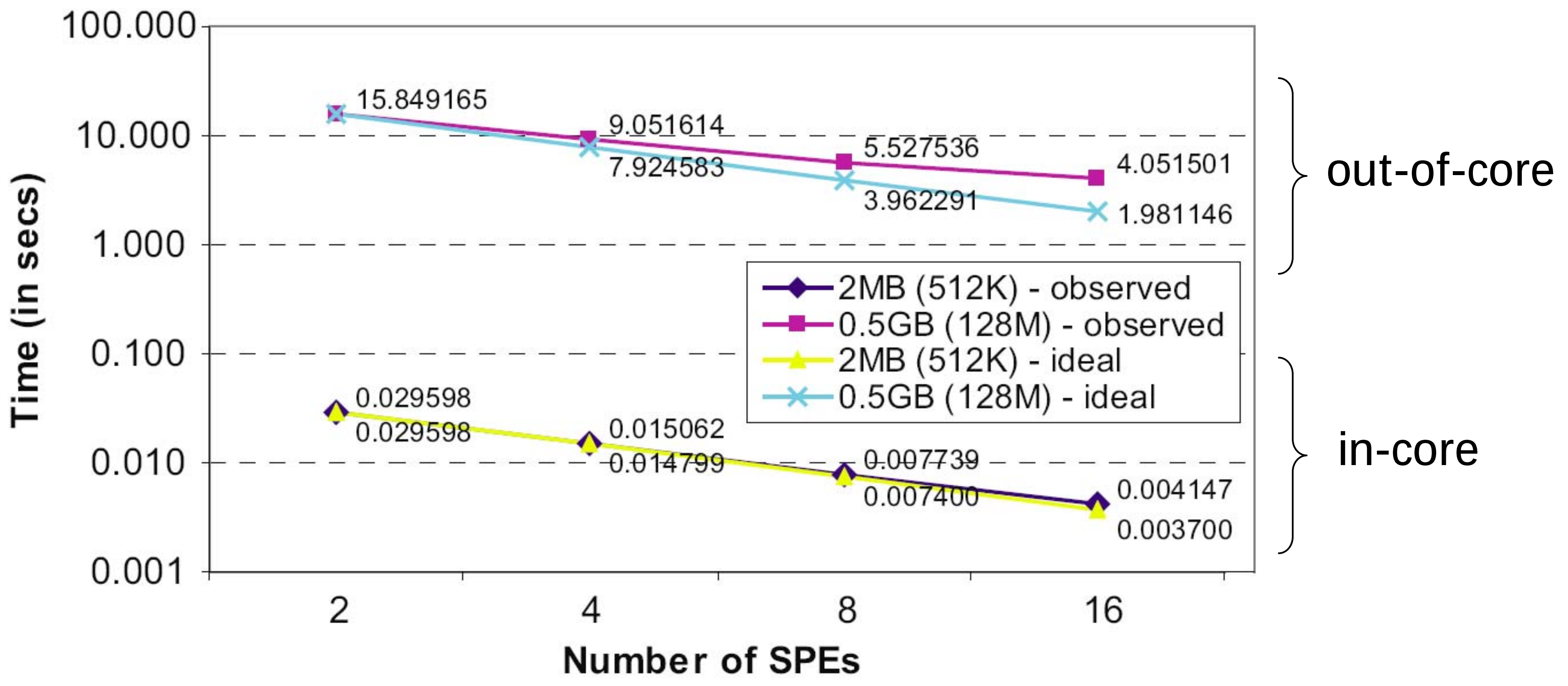
In-core sort, integers

Normalized Out-of-core Sort Performance (lower is better)

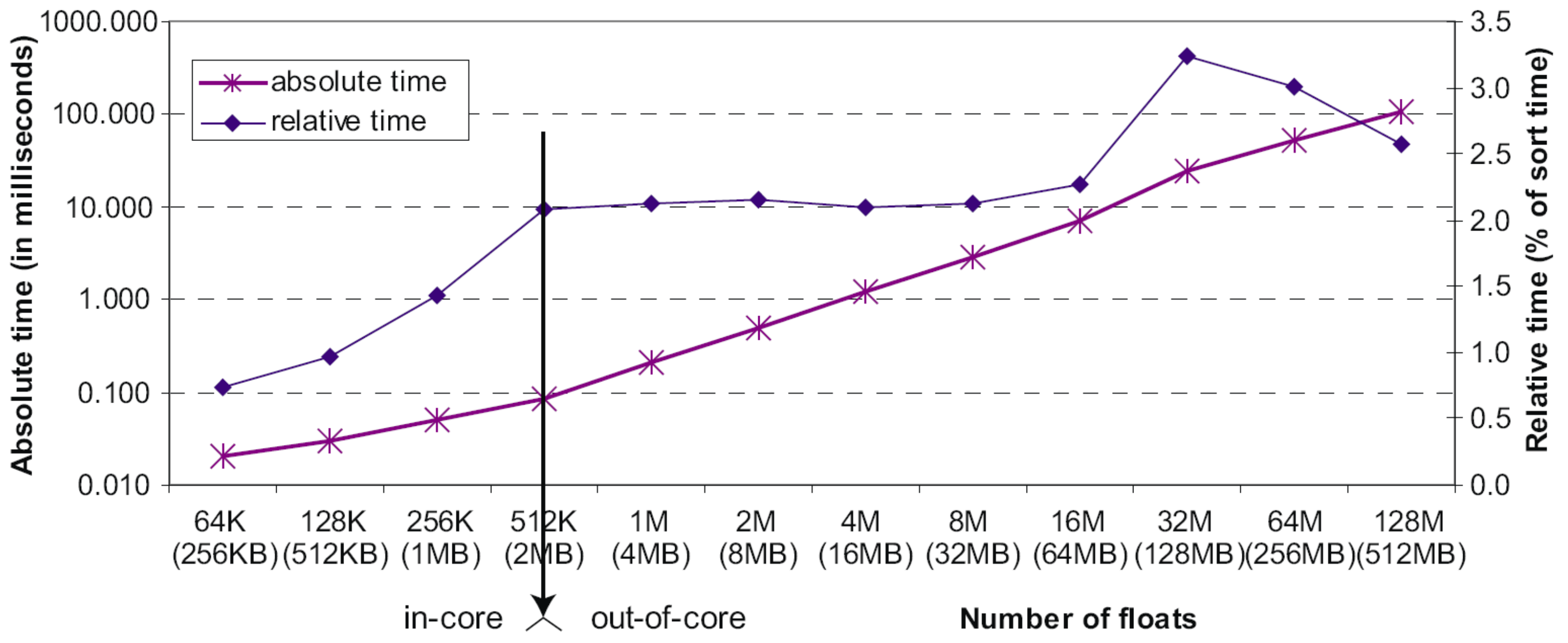


# items	16 SPEs bitonic	3.2GHz Xeon quick	3.2GHz Xeon quick 2-core	PPE quick
1M	0.0098	0.1813	0.098589	0.4333
2M	0.0234	0.3794	0.205728	0.9072
4M	0.0569	0.7941	0.429499	1.9574
8M	0.1372	1.6704	0.895168	4.0746
16M	0.3172	3.4673	1.863354	8.4577
32M	0.7461	7.1751	3.863495	18.3882
64M	1.7703	14.8731	7.946356	38.7473
128M	4.0991	30.0481	16.165578	79.9971

Scalability



Synchronization Cost



Outline

- Motivation
- Cell Architecture Overview
- Sorting on the Cell: The Basics
- Sorting on the Cell: The Algorithms
  - Power-of-two Periodic Bitonic Sort
  - SIMDized Bitonic Sort Kernel
  - Distributed In-core Bitonic Sort
  - Distributed Out-of-core Bitonic Sort
- Experimental Results and Analysis
- ➔Related Work and Future Plans
- ➔Conclusions

Related Work

- Few implementations of merge sort and shell sort that use SIMD compare operations
- Sorting on GPUs (Purcell [2003], Govindaraju [2006], etc.)
  - Map the input data on the 2-D texture memory (upto 512 MB)
    - Every 32-bit float texture element (pixel) can store upto 4 values (channels)
  - Use “large-scale” data-parallel programming over the entire texture map
    - Different than the 128-bit SIMD
  - Key advantages over Cell: large memory and cross-vector aggregation routines
  - MS Research TeraSort paper, SIGMOD 2006.

Conclusions

- CellSort provides high performance at the cost of increased complexity in programming
  - Distributed programming model, synchronization
  - Asynchronous DMA transfers, memory alignment requirements
  - Excessive loop unrolling and branch avoidance
- Unfortunately, out-of-core sort becomes memory bound
  - More SPEs with increased collective local store size
  - And/Or, higher bandwidth access to main memory

Future Plans

- Extend to <key, data> pair sorts (easy)
- Extend to arbitrary key sizes (alignment is an issue)
- Extend to disk-based sorts (need disk arrays and InifiniBand)
- Port TeraSort benchmark on the Cell and compare its performance against GPUs

Questions?

Contact us for the source code.

Lessons Learned

- Code optimization on Cell is non-trivial
  - SIMDization is only the beginning
  - Compiler support is very limited for non-numerical applications
    - Developers need to think like a compiler
      - Control the number of registers to limit register pressure
      - Determine the precise extent of unrolling for improving the CPI
  - In many cases, unrolling results in modifications to the algorithm
- Better performance tools support needed
    - Integration of static instruction scheduling information with runtime measurements and source code
      - Better determination of performance hot spots

Motivation

- Sorting is a key functionality used in a wide variety of application domains
  - Large-scale data intensive applications in many fields, ranging from databases to computer graphics to scientific computing
  - Past studies have explored parallelization and vectorization for massively parallel supercomputers
  - Few existing implementations use SIMD to accelerate key steps in the sorting function
  - No known implementation on a SIMD-only processor (like Cell)
- Sorting has been shown to be very effective on GPUs
  - Recent work by Govindaraju, Grey et al (SIGMOD'06) on implementing the TeraSort benchmark on NVIDIA 7800 GT GPU
- Goals:
  - To understand issues in developing a scalable and high-performance sorting algorithm on the Cell processor
  - To get in-depth experience with programming and optimizing for the Cell architecture