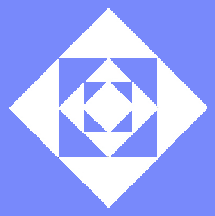


Compiling Effectively for Cell B.E. with GCC

Ira Rosen	IBM Haifa Research Lab
David Edelsohn	IBM T.J.Watson Research Center
Ben Elliston	IBM Australia Development Lab
Revital Eres	IBM Haifa Research Lab
Alan Modra	IBM Australia Development Lab
Dorit Nuzman	IBM Haifa Research Lab
Ulrich Weigand	IBM Deutschland Entwicklung GmbH
Ayal Zaks	IBM Haifa Research Lab

Talk layout

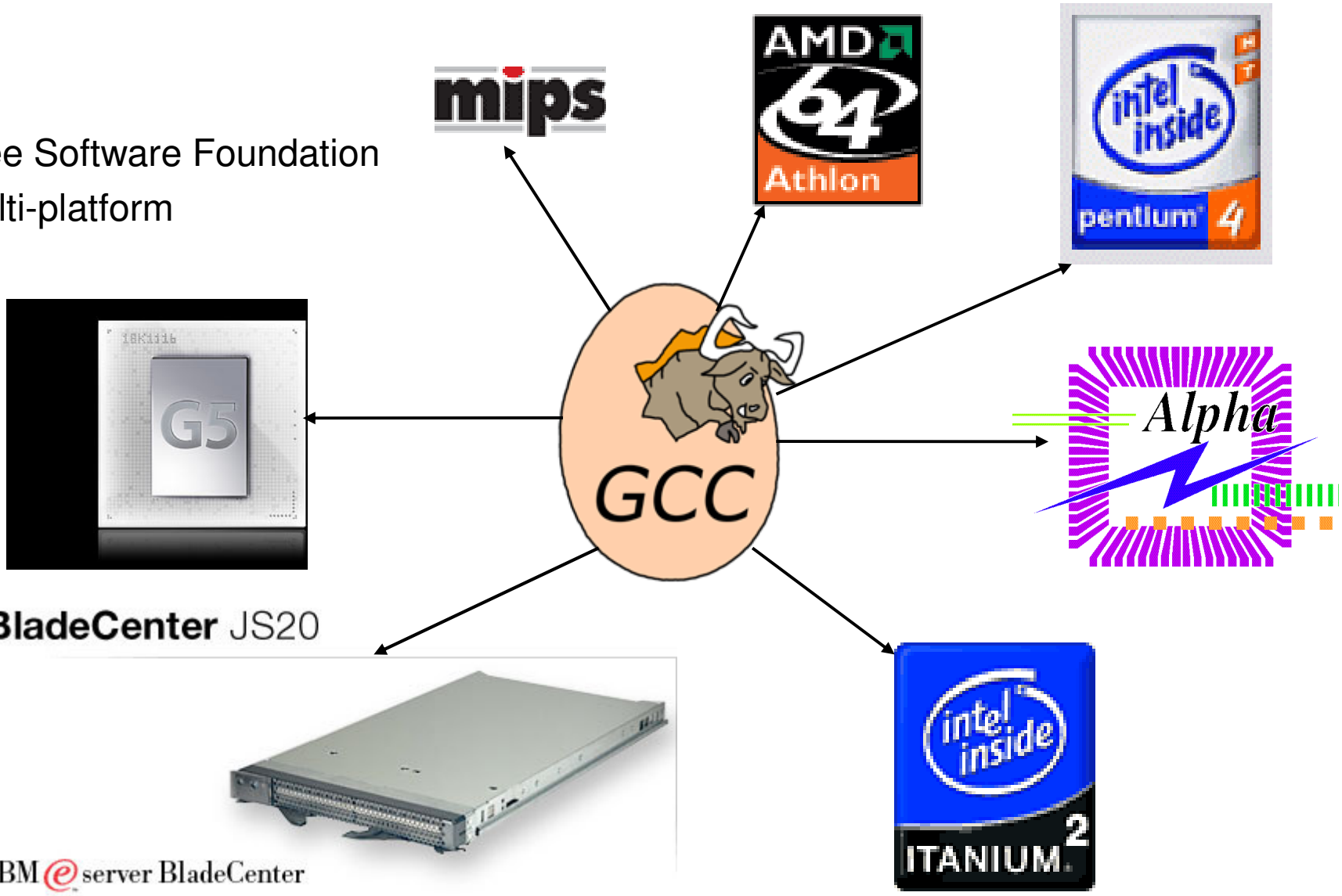
- Background: GCC
  - HRL and GCC
- Compiling Effectively for Cell B.E. with GCC
  - Cell B.E. overview
  - Auto-vectorization enhancements
  - PPE address space support on SPE
  - Supporting the overlay technique
  - Conclusions

Talk layout

- Background: GCC
  - HRL and GCC
- Compiling Effectively for Cell B.E. with GCC
  - Cell B.E. overview
  - Auto-vectorization enhancements
  - PPE address space support on SPE
  - Supporting the overlay technique
  - Conclusions

GCC

- Free Software Foundation
- Multi-platform

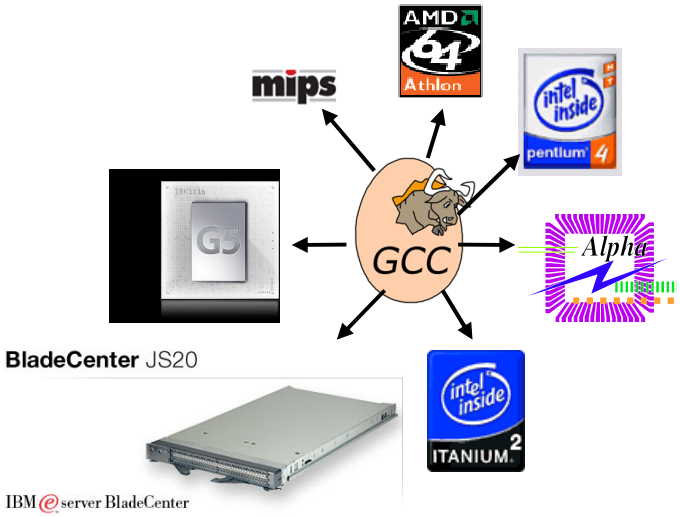


GCC

- Open Source
  Download from gcc.gnu.org
- Multi-platform
- 2.1 million lines of code, 15 years of development
- How does it work
  - svn
  - mailing list: gcc-patches@gcc.gnu.org
  - steering committee, maintainers

GCC

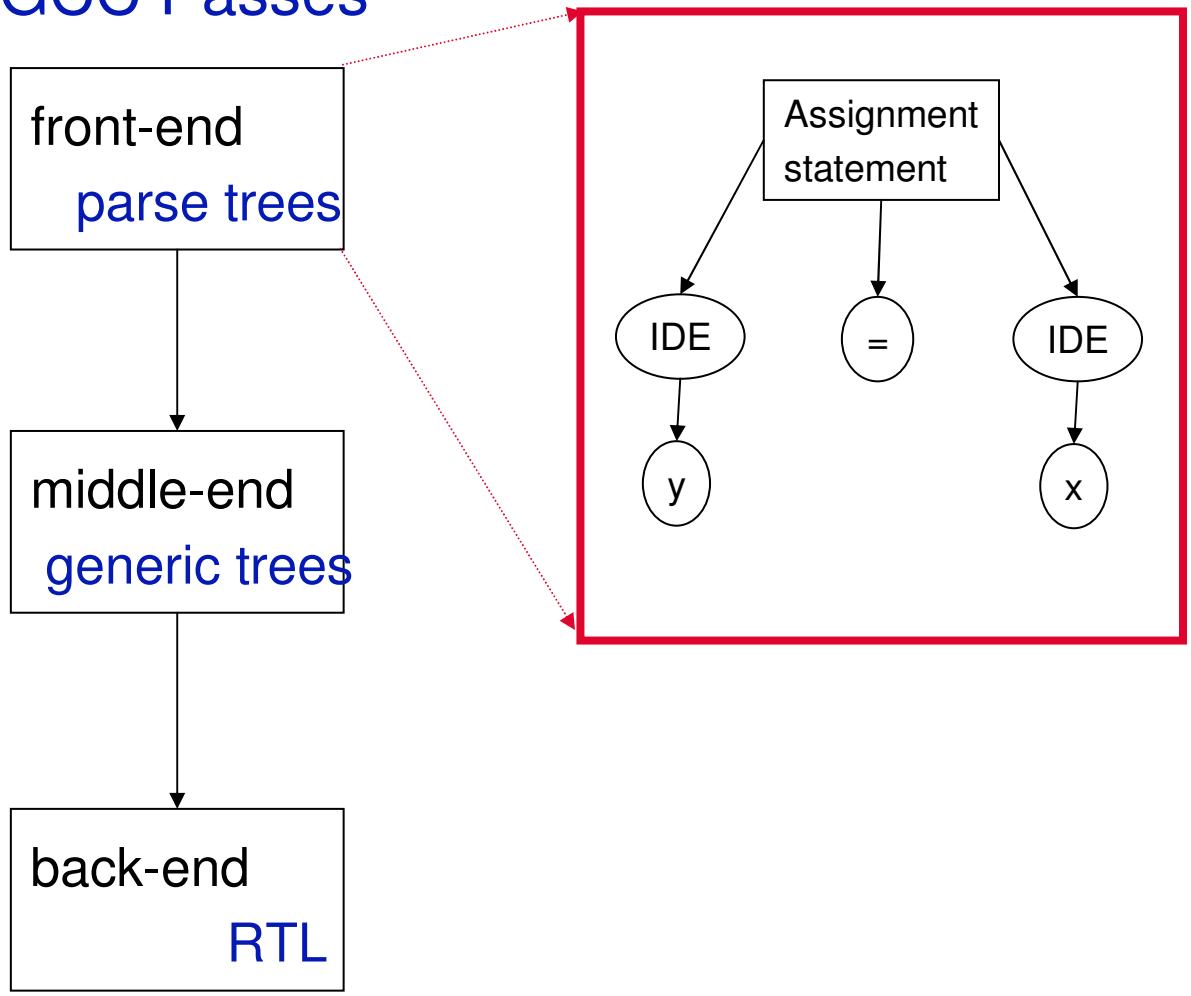
- Who's involved
  - Volunteers
  - Linux distributors (RedHat, Suse…)
  - Code Sourcery, AdaCore…
  - IBM, HP, Intel, Apple…



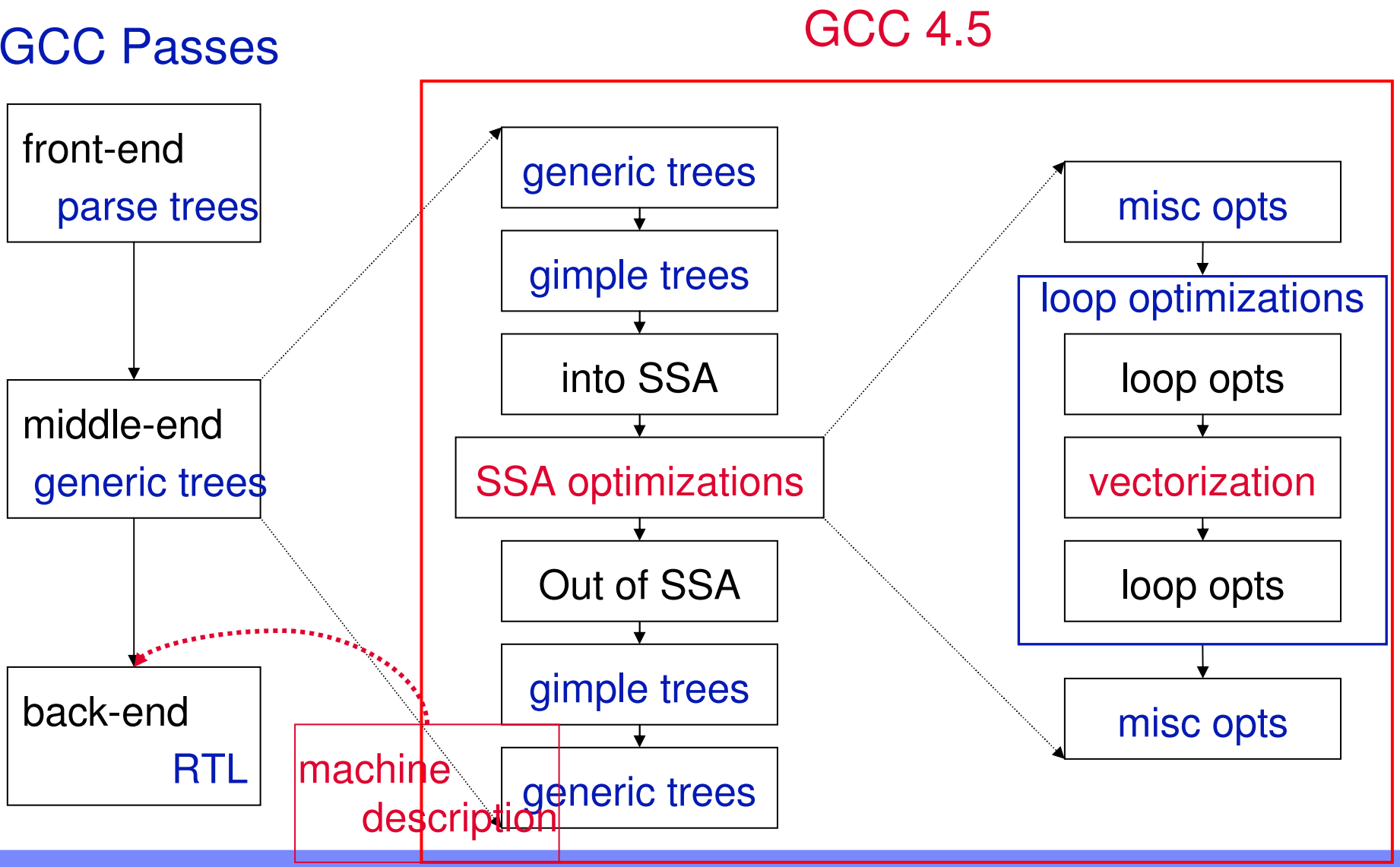
Linux on Power

Open, powerful and affordable,
a key to innovation

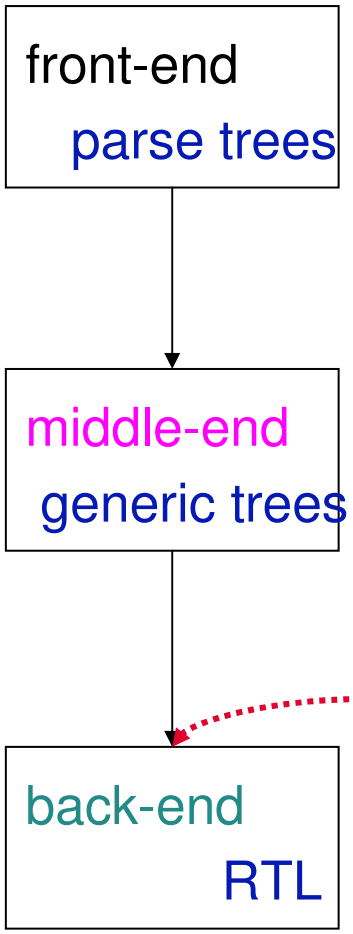
GCC Passes


front-end
parse trees
middle-end
generic trees
back-end
RTL
Assignment
statement
IDE
=
IDE
y
x

GCC Passes


GCC 4.5
front-end
parse trees
middle-end
generic trees
back-end
RTL
machine
description
generic trees
gimple trees
into SSA
SSA optimizations
Out of SSA
gimple trees
generic trees
misc opts
loop optimizations
loop opts
vectorization
loop opts
misc opts

GCC Passes



- IPO
  - CP
  - Aliasing
  - Data layout
- Vectorization
- Graphite
- Loop unrolling
- Scheduler
- Modulo Scheduling

machine description
- Power7

- The Haifa team:
  - Olga Golovanevsky
  - Razya Ladelsky
  - Dorit Nuzman
  - Mircea Namolaru
  - Ira Rosen
  - Victor Kaplansky
  - Roni Kupershtok
  - Sergei Dyshel
  - Alon Dayan
  - Revital Eres
  - Ayal Zaks

Talk outline

- Cell B.E. overview
- Auto-vectorization enhancements
- PPE address space support on SPE
- Supporting the overlay technique
- Conclusions

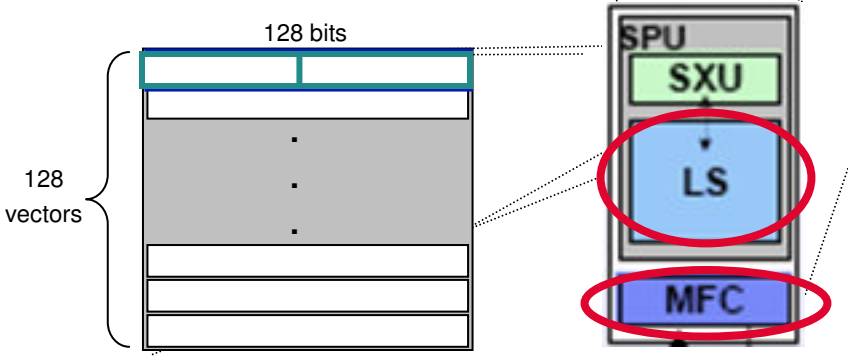
Talk outline

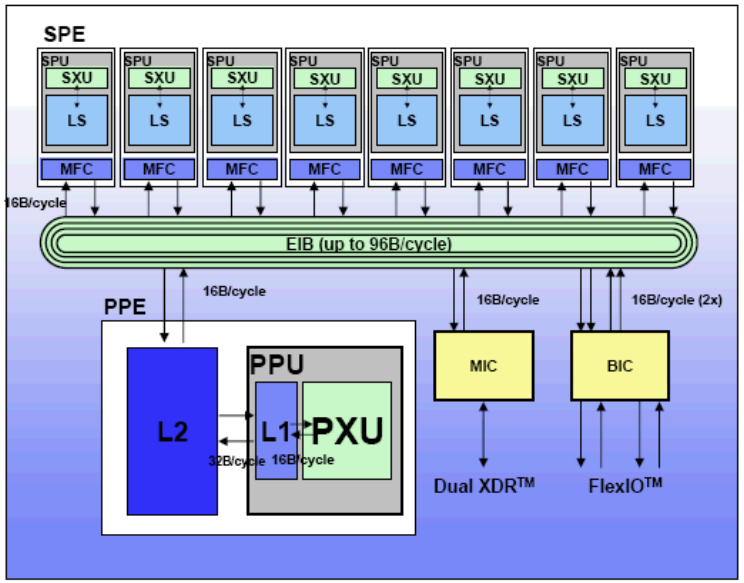
- Cell B.E. overview
- Auto-vectorization enhancements
- PPE address space support on SPE
- Supporting the overlay technique
- Conclusions

Cell B.E. overview

The CELL processor consists of:

- Power Processor Element (PPE)
- 8 Synergistic Processor Elements (SPEs)
  - Each SPE consists of:
    - Memory flow controller (MFC)
    - 256KB local memory
    - Synergistic Process Unit (SPU)
      - Supports a new SIMD instruction set.
      - 128 SIMD vector registers, each 128 bits wide.





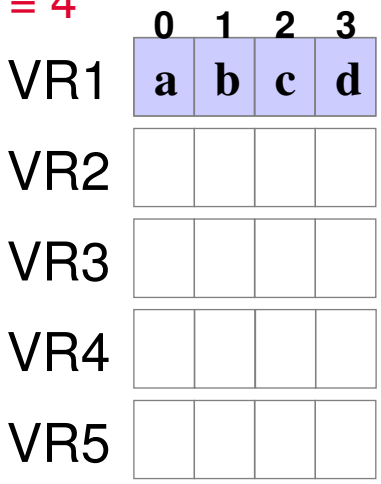
Source: M. Gschwind et al., Hot Chips-17, August 2005

Talk outline

- Cell B.E. overview
- Auto-vectorization enhancements
- PPE address space support on SPE
- Supporting the overlay technique
- Conclusions

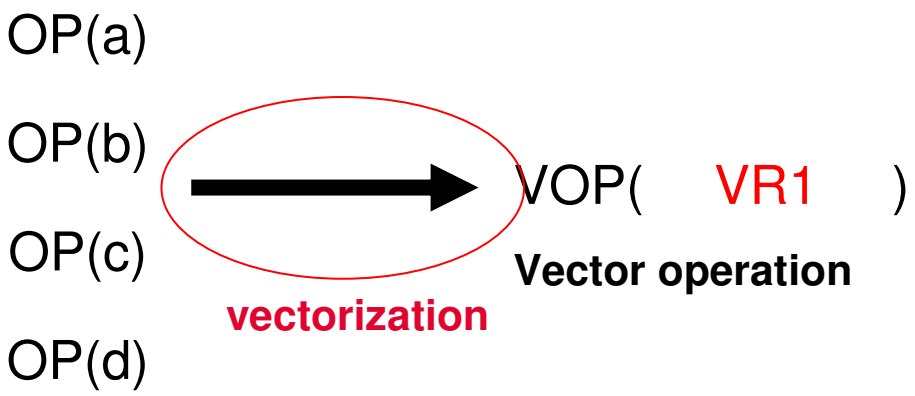
What is vectorization

VF = 4

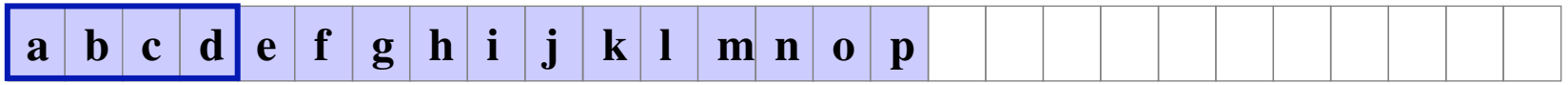


Vector Registers

- Data elements packed into vectors
- Vector length → Vectorization Factor (VF)
- No Data Dependences
- SIMD Architectural Capabilities



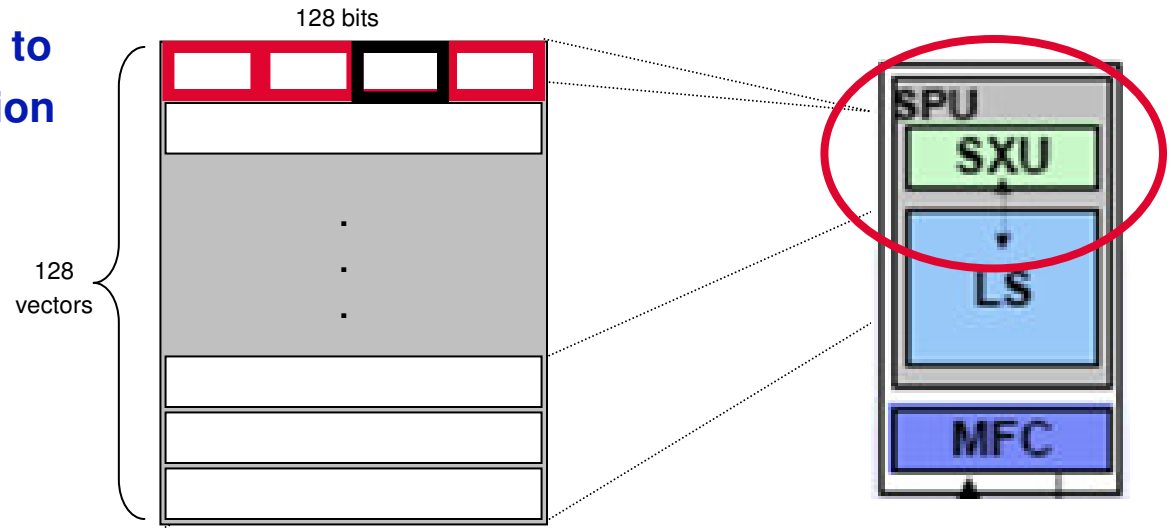
Data in Memory:



Vectorization for SPE

- **Why vectorization is especially important for Cell SPU?**
  - All instructions are SIMD instructions operating on vector registers.
  - Operations on scalar values are implemented using vector operations on vectors holding the scalar in one “preferred” element.
  - Relatively low overhead in transfers between vector and scalar data as both reside in the same vector register file.

It is therefore highly important to maximize vector code generation for the Cell SPEs.



Talk outline

- Cell B.E. overview
- Auto-vectorization enhancements
  - Outer loop vectorization
  - Intra loop vectorization
  - Vectorization of Strided Accesses
- PPE address space support on SPE
- Supporting the overlay technique
- Conclusions

Outer loop vectorization

- **loop-nest:**

```
for(i=0; i<N; i++){
   for (j=0; j<M; j++){
     a[i][j] = a[i][j] + b[i][j];
   }
}
```

Outer-Loop Vectorization

Outer-Loop Vectorization: Why?

- Inner-most loop may not be vectorizable
  - Cross iteration data-dependences

```
for (i = 0; i < N; i++) {
   for (j = 0; j < M; j++) {
      a[i][j+1] = a[i][j] + B;
    }
}
```

Outer-Loop Vectorization: Why?

- Inner-most loop may not be vectorizable
  - Cross iteration data-dependences
  - Non-Associative Reduction

```
i=0: out[0]=in[0]+in[1]+in[2]+in[3]+in[4]+in[5]+in[6]+...
i=1: out[1]=in[1]+in[2]+in[3]+in[4]+in[5]+in[6]+in[7]+...
i=2: out[2]=in[2]+in[3]+in[4]+in[5]+in[6]+in[7]+in[8]+...
i=3: out[3]=in[3]+in[4]+in[5]+in[6]+in[7]+in[8]+in[9]+...
```


```
for (i = 0; i < N; i++) {
    float sum = 0;
    for (j = 0; j < M; j++) {
      sum  += in[j+i];
    }
    out[i] = sum;
}
```

Innermost-Loop Vectorization

```
for (i = 0; i < N; i++) {
    float vector vsum = [0…0];
    for (j = 0; j < M/4; j++) {
      vsum  += in[j+i:j+3+i];
    }
    sum = reduce(vsum)
    out[i] = sum;
}
```

Outer-Loop Vectorization

```
for (i = 0; i < N/4; i++) {
    float vector vsum = [0…0];
    for (j = 0; j < M; j++) {
      vsum  += in[j+i:j+i+3];
    }
    out[i:i+3] = vsum;
}
```

Outer-Loop Vectorization: Why?

- Profitabilty
  - Elimination of reduction epilog overhead
  - Larger portion of the code vectorized
    - More register reuse, less memory bandwidth

```
for (i = 0; i < N; i++) {
    float sum = 0;
    for (j = 0; j < M; j++) {
      sum  += in[j+i];
    }
    out[i] = sum;
}
```

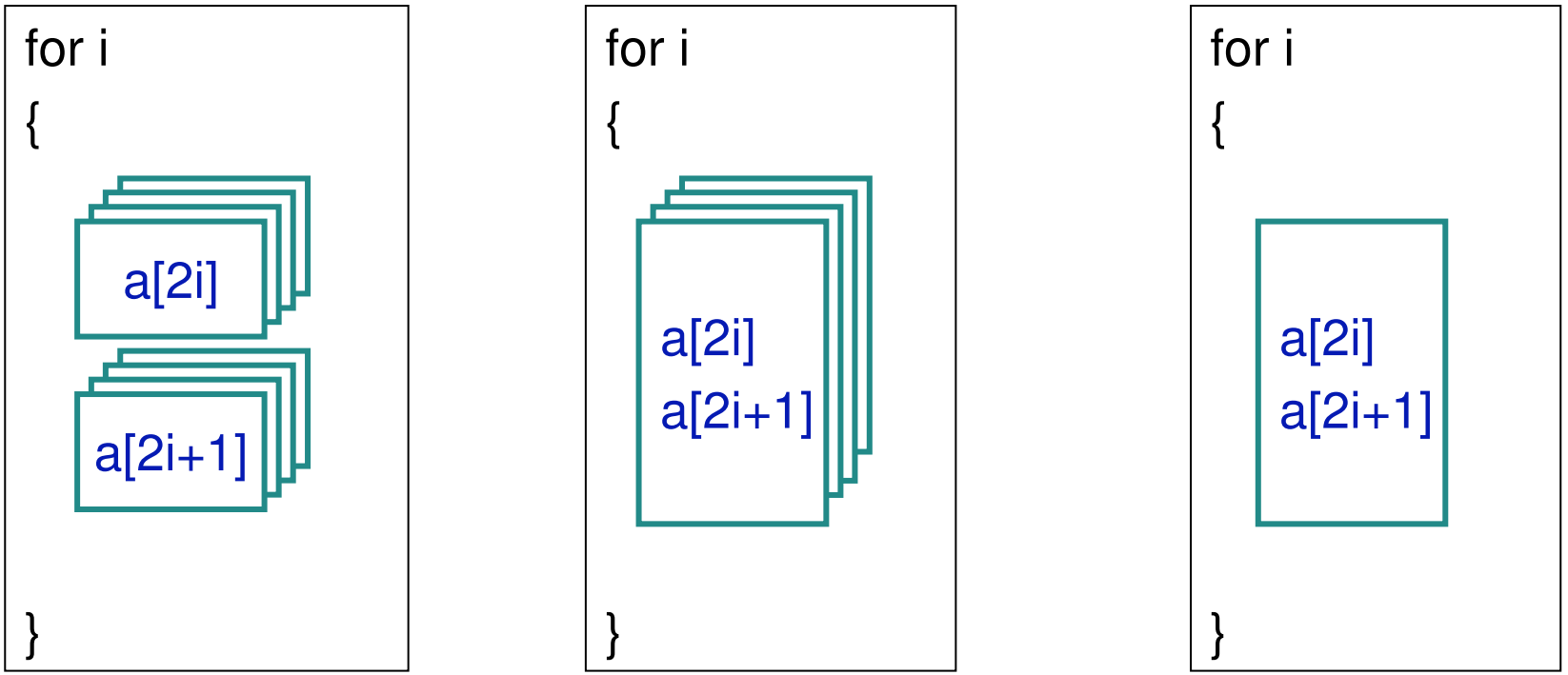
Innermost-Loop Vectorization

```
for (i = 0; i < N; i++) {
    float sum = 0;
    for (j = 0; j < M/4; j++) {
      vsum  += in[j+i:j+3+i];
    }
    sum = reduce(vsum)
    out[i] = sum;
}
```

Outer-Loop Vectorization

```
for (i = 0; i < N/4; i++) {
    float vsum = 0;
    for (j = 0; j < M; j++) {
      vsum  += in[j+i:j+i+3];
    }
    out[i:i+3] = vsum;
}
```

Auto-Vectorization for SPE



Outer-Loop Vectorization: Why?

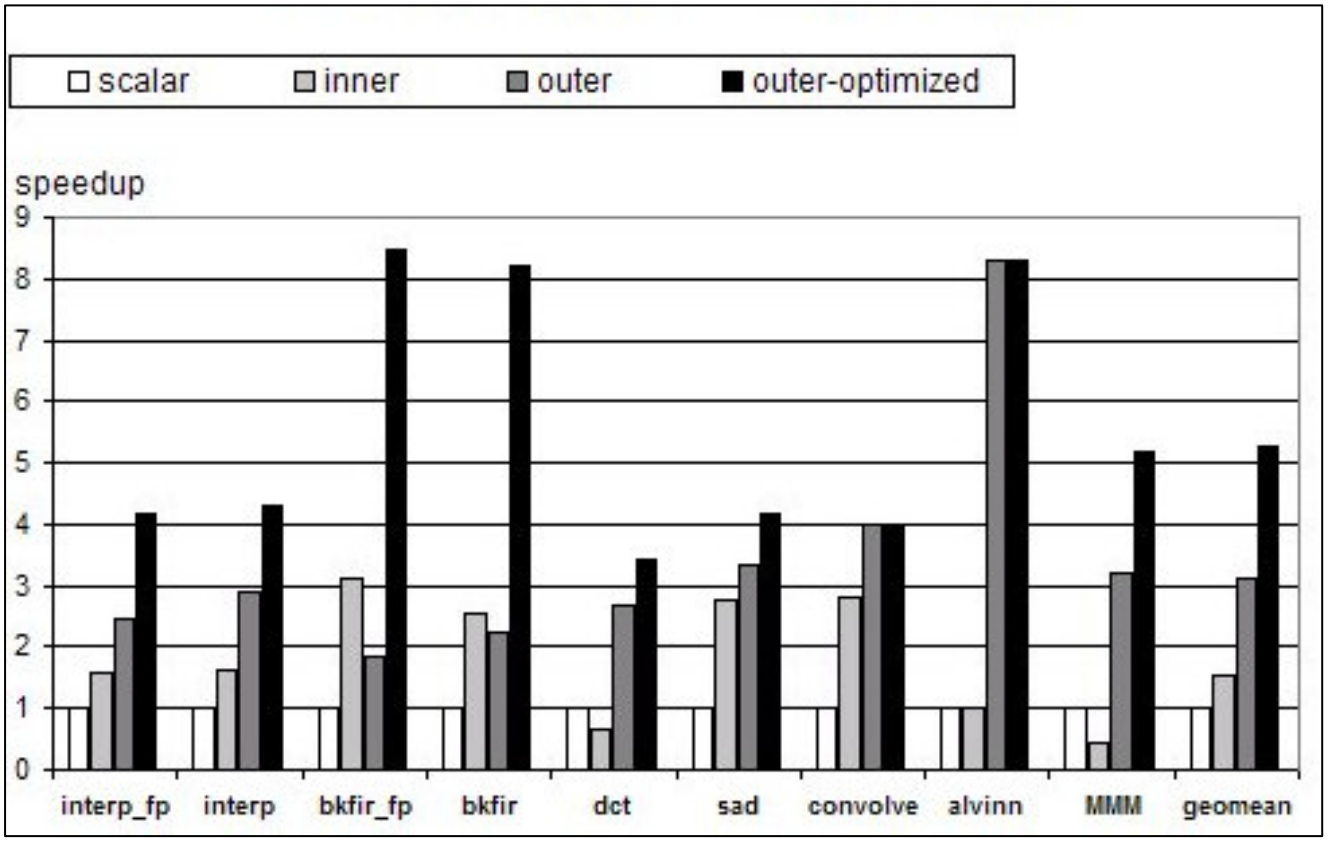
- Correctness
  - Cross iteration data-dependences
  - Non-Associative Reduction
- Profitabilty
  - Elimination of reduction epilog overhead
  - Larger portion of the code vectorized
    - More register reuse / less memory bandwidth
  - Loop vectorization may destroy perfect nests
  - Less “Per-loop” overheads
  - Longer iteration count in outer-loop
    - Multimedia: short-trip innermost loops
  - Smaller strides on outer-loop level
    - Better spatial locality in outer-loop
  - Unknown stride in innermost loop

Outer loop vectorization

- Larger portion of the code vectorized
  - More register reuse / less memory bandwidth
- **In-Place Outer-Loop Vectorization for SIMD**
  - with realignment optimization
- D. Nuzman, and A. Zaks, **"Outer-Loop Vectorization - Revisited for Short SIMD Architectures"**, *PACT 2008*

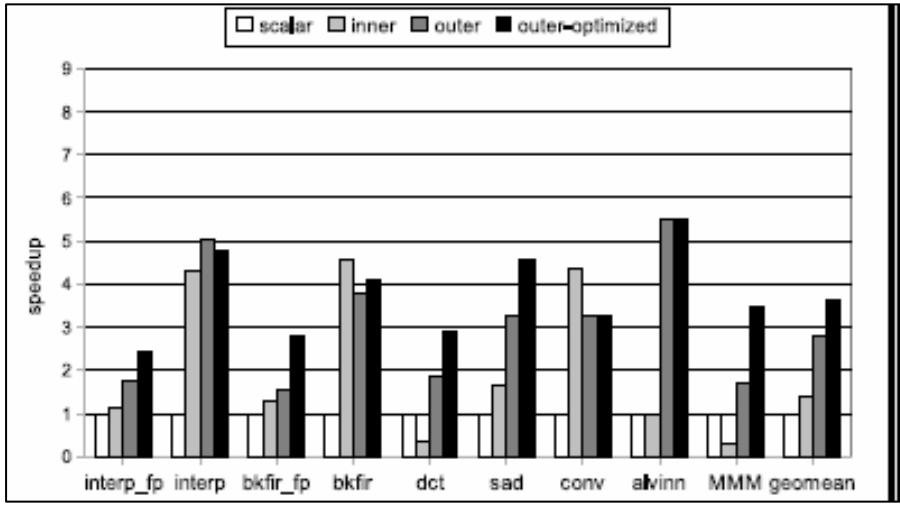
Outer loop vectorization – Experimental results

CELL SPU: Performance impact

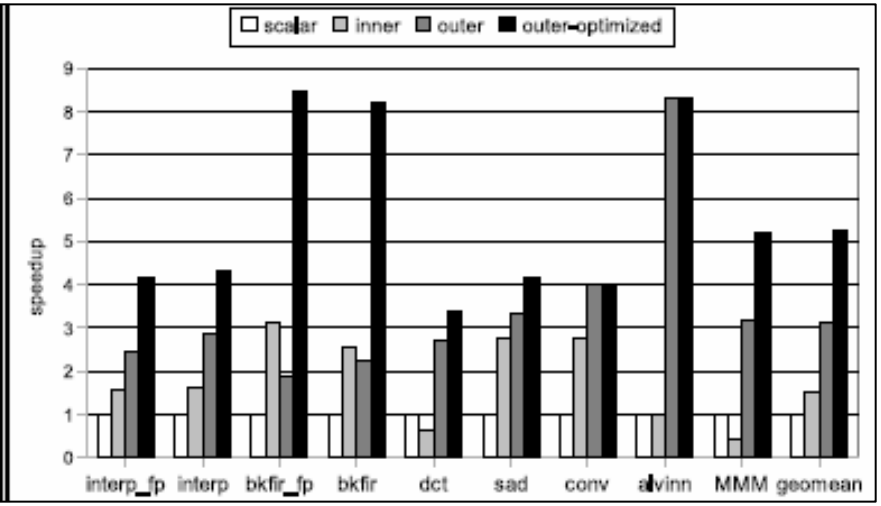


Outer loop vectorization – Experimental results

PowerPC970: Performance impact



CELL SPU: Performance impact

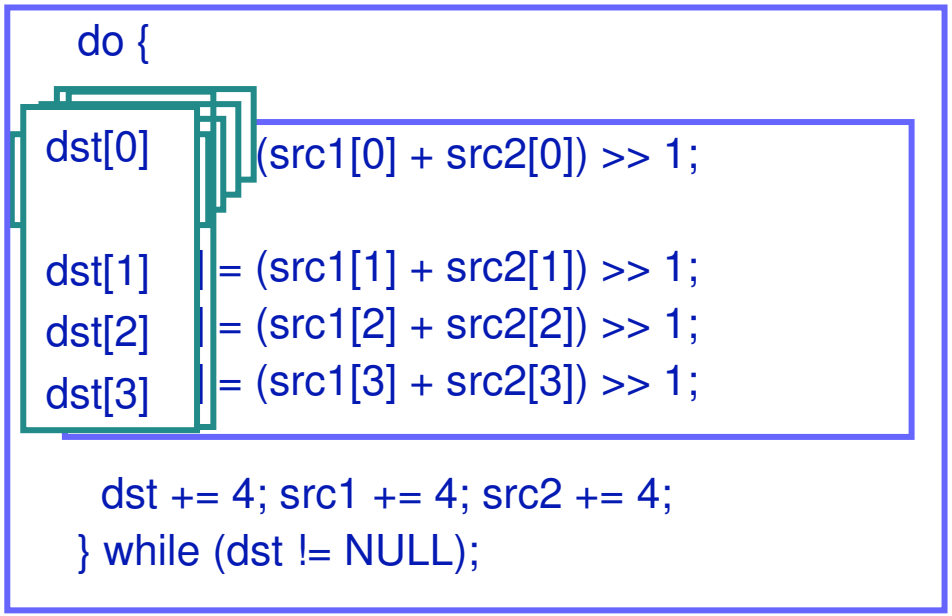


Talk outline

- Cell B.E. overview
- Auto-vectorization enhancements
  - Outer loop vectorization
  - Intra loop vectorization
  - Vectorization of Strided Accesses
- PPE address space support on SPE
- Supporting the overlay technique
- Conclusions

Intra loop vectorization

- Classic vectorization techniques exploit inter-iteration parallelism

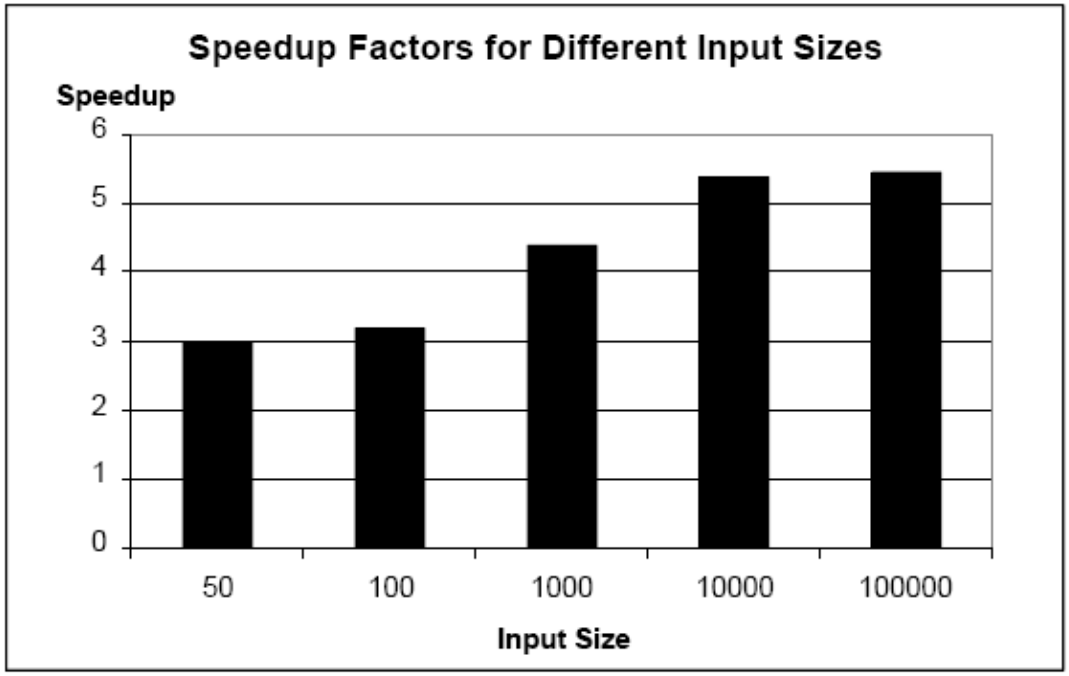


Uncountable loop

- **SLP** – straight line code vectorization - Larsen and Amarasinghe, 2000.
- **Loop-aware SLP** - exploit both inter- and intra-iteration parallelism.
- Ira Rosen, Dorit Nuzman, and Ayal Zaks, "**Loop-based SLP**", *GCC Developers' Summit 2007.*

Intra loop vectorization - Experimental Results

CELL SPU: Performance impact



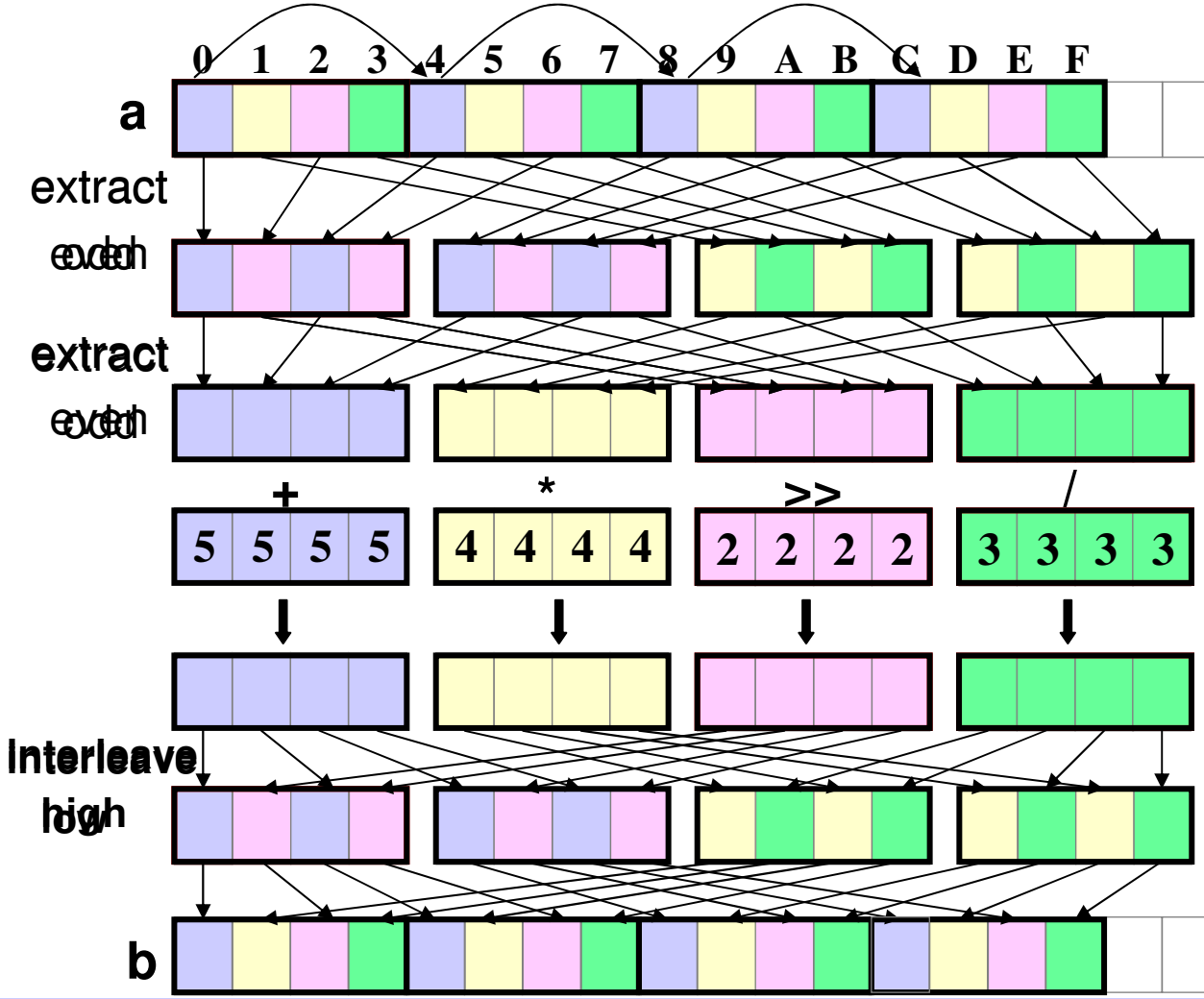
RGB to YIQ conversion

[Y] = 0.299 [R] + 0.587 [G] + 0.114 [B]
[I] = 0.596 [R] - 0.275 [G] - 0.321 [B]
[Q] = 0.212 [R] - 0.523 [G] + 0.311 [B]

Talk outline

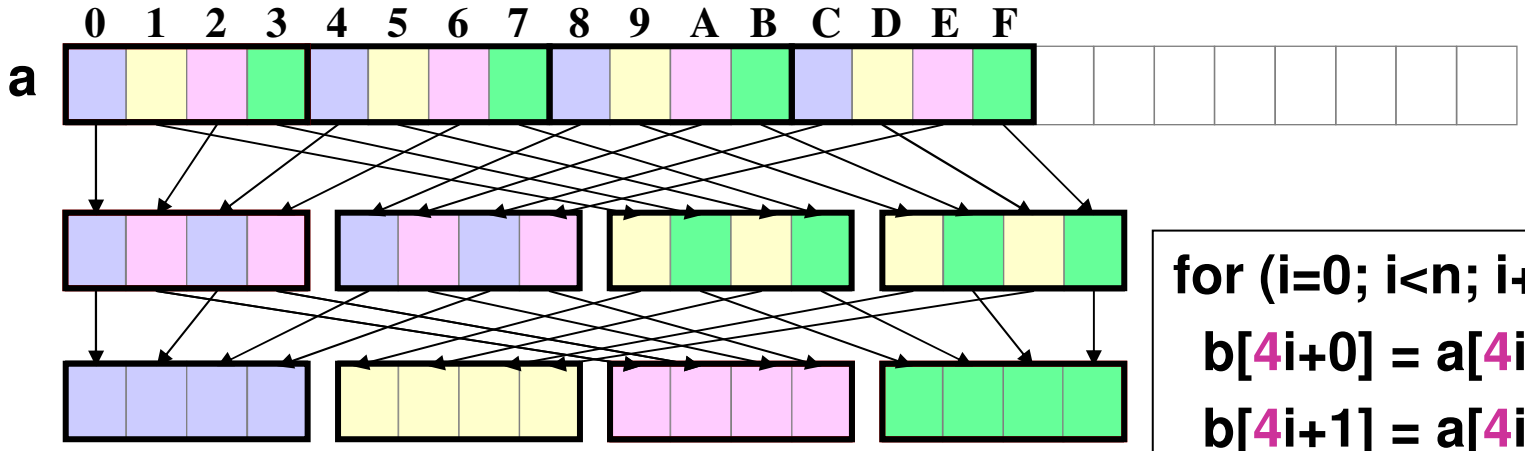
- Cell B.E. overview
- Auto-vectorization enhancements
  - Outer loop vectorization
  - Intra loop vectorization
  - Vectorization of Strided Accesses
- PPE address space support on SPE
- Supporting the overlay technique
- Conclusions

Vectorization of Strided Accesses




```
for (i=0; i<n; i++) {
  b[4i+0] = a[4i+0]  +  5;
  b[4i+1] = a[4i+1]   *  4;
  b[4i+2] = a[4i+2]  >>  2;
  b[4i+3] = a[4i+3]   /   3;
}
```

Vectorization of Strided Accesses

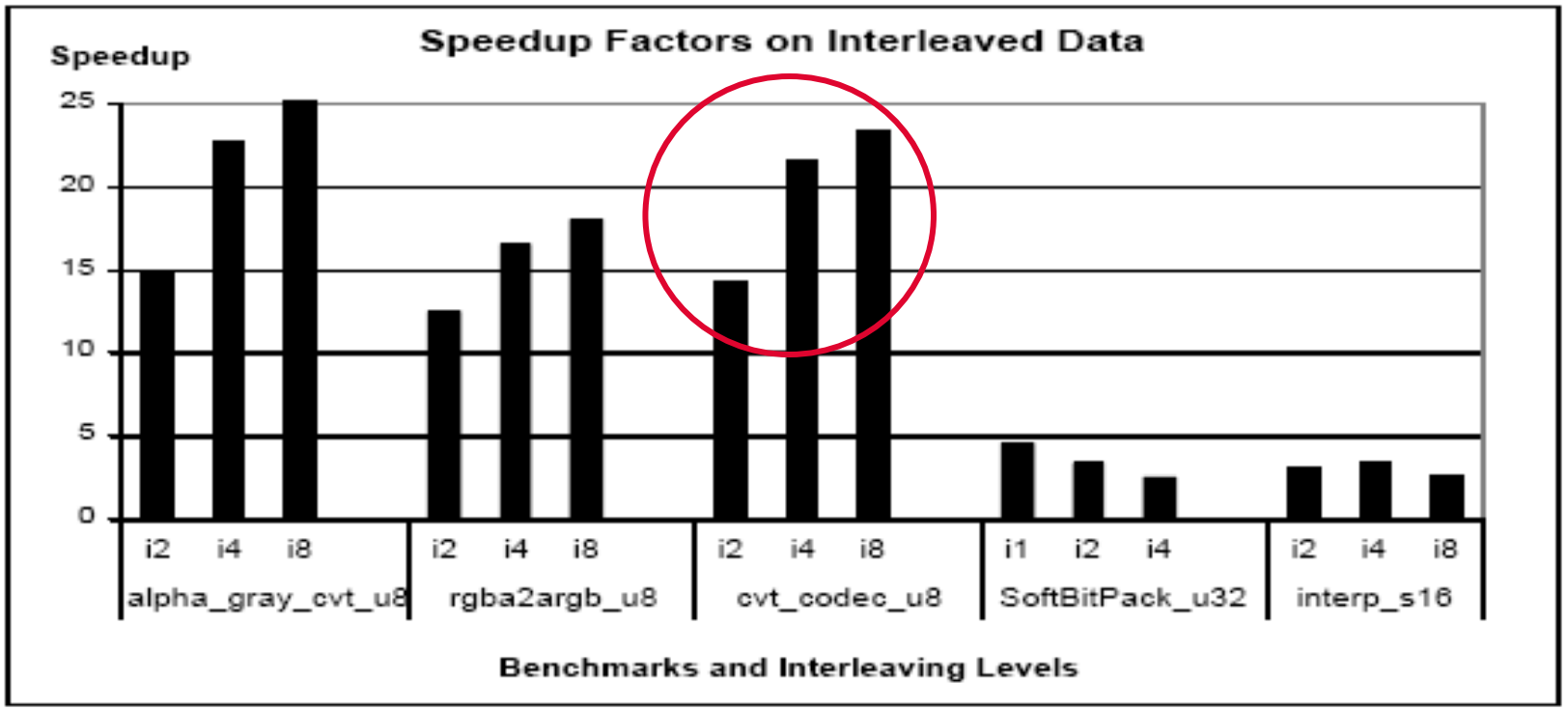



```
for (i=0; i<n; i++) {
  b[4i+0] = a[4i+0]  +  5;
  b[4i+1] = a[4i+1]   *  4;
  b[4i+2] = a[4i+2]  >>  2;
  b[4i+3] = a[4i+3]   /   3;
}
```

- Supports vectorization of non-unit stride memory accesses, with power-of-2 strides.
- "**Auto-Vectorization of Interleaved Data for SIMD**", Dorit Nuzman, Ira Rosen, and Ayal Zaks, *PLDI 20*06.

Vectorization of Strided Accesses - Experimental Results

CELL SPU: Performance impact



Talk outline

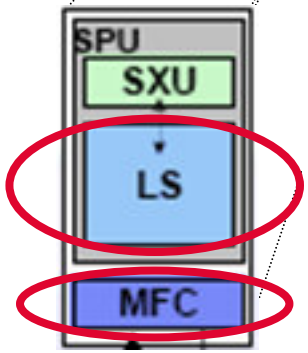
- Cell B.E. overview
- Auto-vectorization enhancements
- PPE address space support on SPE
- Supporting the overlay technique
- Conclusions

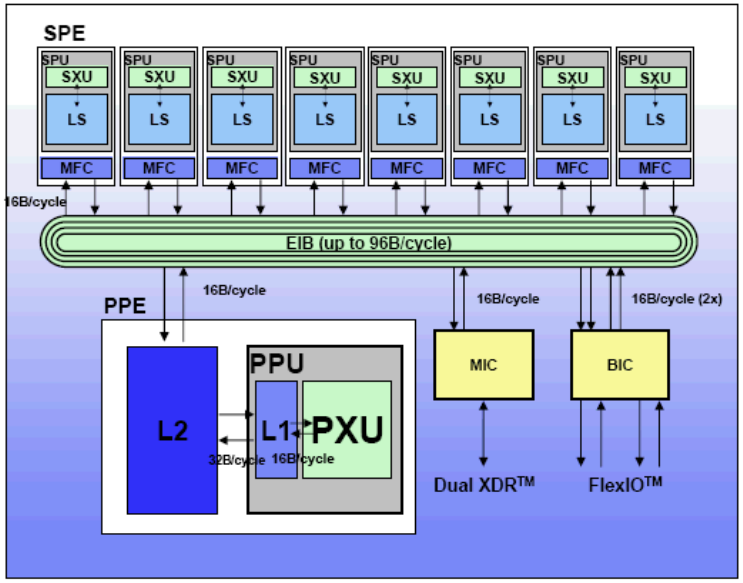
PPE Address Space Support on SPE

- Load and store instructions of an SPE access a local store of 256KB private to the SPE.
- DMA operations provided by the MFC enable the SPE to copy data between its local store and main storage.

```
inline void
dma_ls_to_mem (unsigned int mem_addr,
                volatile void *ls_addr,
                unsigned int size)
  {
   unsigned int tag = 0;
   unsigned int mask = 1;

   mfc_put (ls_addr, mem_addr, size, tag, 0, 0);
   mfc_write_tag_mask (mask);
   mfc_read_tag_status_all ();
}
```





Source: M. Gschwind et al., Hot Chips-17, August 2005

PPE Address Space Support on SPE

- GCC was extended to enable accessing PPE memory from SPE without explicitly executing DMA operations.
- Following the **embedded extension to C programming language**.
  - GCC was extended to supports multiple address spaces.
  - This extension permits variables to be qualified as belonging to a specific address space by tagging their type with an identifier recognized by the compiler.
  - The compiler can then synthesize code to access variables in these other address spaces.
- A **software-managed data cache** was developed to improve performance of programs accessing variables in the PPE address space.
  - configurable cache size chosen by the[illegible]g the SPU program.

```
inline void
dma_ls_to_mem ( __ea int *ppe_variable, int *ls_addr)
```

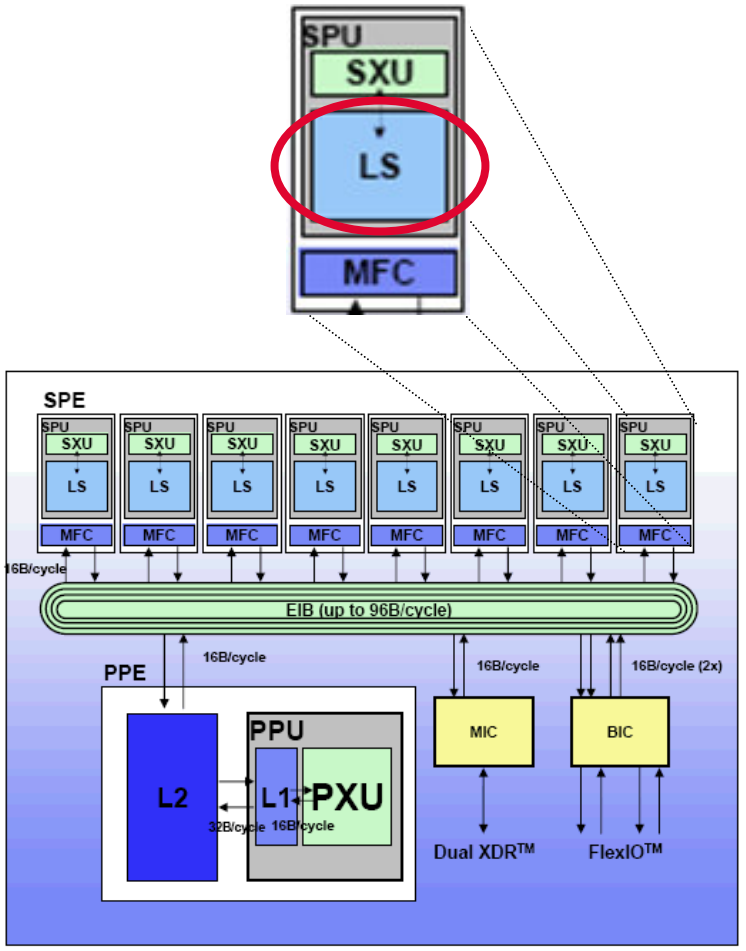
```
#include <spu_cache.h>
…
cache_fetch(_ea)
cache_evict (_ea)
…
```

Talk outline

- Cell B.E. overview
- Auto-vectorization enhancements
- PPE address space support on SPE
- Supporting the overlay technique
- Conclusions

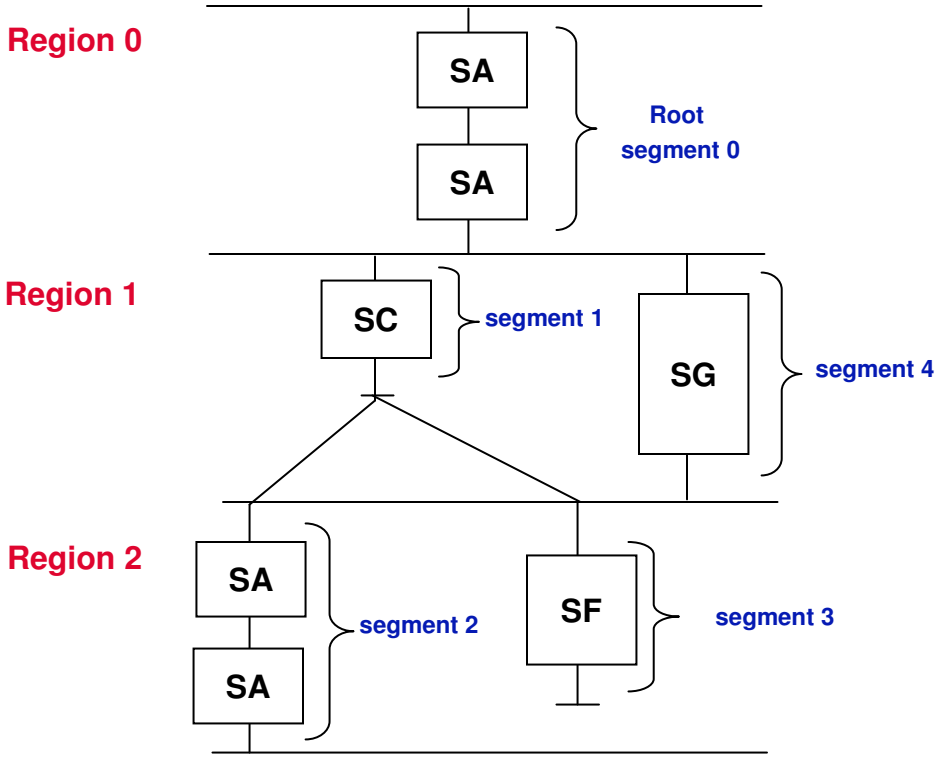
Supporting the overlay technique

- The limited amount of local store directly imposes a severe constraint on the code-size of SPE programs.
- Large programs may not fit the local store.
- Therefore, the toolchain was extended to support **overlay technique:**
  - Partition the code into multiple pieces, each sufficiently small in size, which will be swapped in and out of the local store at runtime.



Source: M. Gschwind et al., Hot Chips-17, August 2005

Supporting the overlay technique cont.



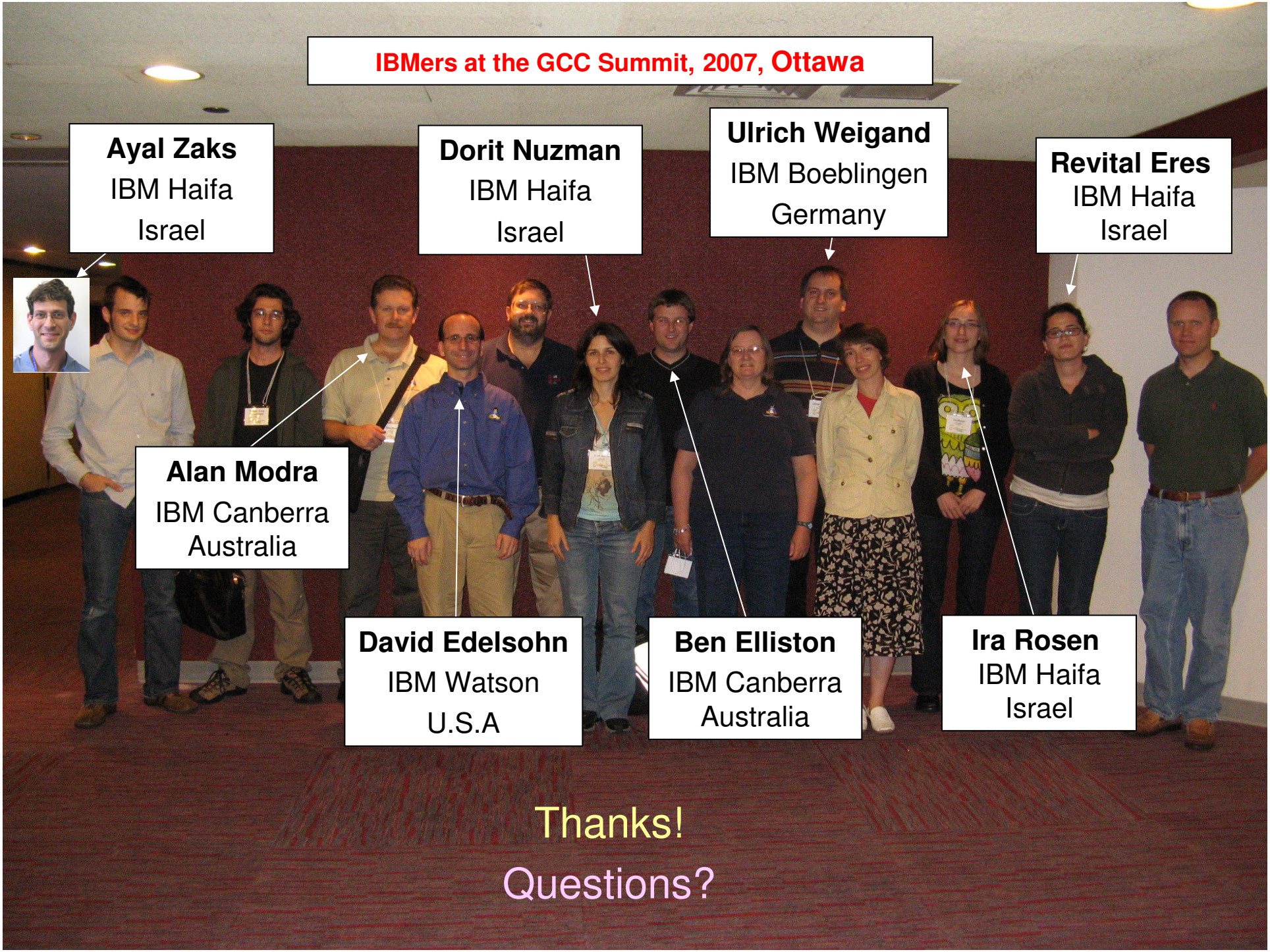
Supporting the overlay technique cont.

- Preparing code for overlay management consists of a static preprocessing stage which is done by the compiler and linker.
  - The compiler first partition the code into sections of maximum size.
    - It can break functions if needed.
      - Avoid splitting critical sections.
      - Try not breaking loops.
  - The linker construct the overlaid program from the sections.
    - replaces each branch or call which leads to a new segment by a stub which transfers the control to an overlay loading routine during execution.

```
OVERLAY {
 .segment1 {./sc.o(.text)}
 .segment4 {./sg.o(.text)}
}
OVERLAY {
 .segment2 {./sd.o(.text) ./se.o(.text)}
 .segment3 {./sf.o(.text)}
}
```

Conclusions

- The Cell Broadband Engine provides unique computational opportunities yet poses new challenges for tool-chains.
- Our collaborative effort to address some of the challenges in GCC and GNU ld includes:
  - Support the overlay technique to overcome the local store constraint on programs code size.
  - Support extension to the C language to access variables in main memory from the SPU to avoid explicit DMA operations.
  - Autovectorization extensions:
    - Innovative vectorization opportunities beyond inner-most loops and stride accesses, traditionally considered too costly.


IBMers at the GCC Summit, 2007, Ottawa
Ayal Zaks
IBM Haifa
Israel
Dorit Nuzman
IBM Haifa
Israel
Ulrich Weigand
IBM Boeblingen
Germany
Revital Eres
IBM Haifa
Israel
Alan Modra
IBM Canberra
Australia
David Edelsohn
IBM Watson
U.S.A
Ben Elliston
IBM Canberra
Australia
Ira Rosen
IBM Haifa
Israel
Thanks!
Questions?

Back-up slides

XL relevant techniques

- **Outer-Loop Vectorization in xlc -** Eliminate Inner-Loop (Short-Loop aggregation) - Wu, Eichenberger,Wang (2005)
- **SLP in xlc –** Wu et al., ICS (2005)
- **Overlay support in xlc** - the basic unit of partitioning is a function; use outlining to create smaller functions - A. E. Eichenberger et al., ”Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture”, IBM Systems Journal (2006).

Auto-Vectorization in GCC

- Classic vectorization techniques exploit inter-iteration parallelism

```
for i
{
    a[2i]
    a[2i+1]
}
```

“1-1 replace”

loop-based vectorization

Outer-Loop Vectorization: Why?

- Inner-most loop may not be vectorizable
  - Cross iteration data-dependences

```
for (i = 0; i < N; i++) {
   for (j = 0; j < M; j++) {
      a[i][j+1] = a[i][j] + B;
    }
}
```

Outer-Loop Vectorization: Why?

- Inner-most loop may not be vectorizable
  - Cross iteration data-dependences
  - Non-Associative Reduction

```
i=0: out[0]=in[0]+in[1]+in[2]+in[3]+in[4]+in[5]+in[6]+...
i=1: out[1]=in[1]+in[2]+in[3]+in[4]+in[5]+in[6]+in[7]+...
i=2: out[2]=in[2]+in[3]+in[4]+in[5]+in[6]+in[7]+in[8]+...
i=3: out[3]=in[3]+in[4]+in[5]+in[6]+in[7]+in[8]+in[9]+...
```


```
for (i = 0; i < N; i++) {
    float sum = 0;
    for (j = 0; j < M; j++) {
      sum  += in[j+i];
    }
    out[i] = sum;
}
```

Innermost-Loop Vectorization

```
for (i = 0; i < N; i++) {
    float vector vsum = [0…0];
    for (j = 0; j < M/4; j++) {
      vsum  += in[j+i:j+3+i];
    }
    sum = reduce(vsum)
    out[i] = sum;
}
```

Outer-Loop Vectorization

```
for (i = 0; i < N/4; i++) {
    float vector vsum = [0…0];
    for (j = 0; j < M; j++) {
      vsum  += in[j+i:j+i+3];
    }
    out[i:i+3] = vsum;
}
```

Outer-Loop Vectorization: Why?

- Profitabilty
  - Elimination of reduction epilog overhead
  - Larger portion of the code vectorized
    - More register reuse, less memory bandwidth

```
for (i = 0; i < N; i++) {
    float sum = 0;
    for (j = 0; j < M; j++) {
      sum  += in[j+i];
    }
    out[i] = sum;
}
```

Innermost-Loop Vectorization

```
for (i = 0; i < N; i++) {
    float sum = 0;
    for (j = 0; j < M/4; j++) {
      vsum  += in[j+i:j+3+i];
    }
    sum = reduce(vsum)
    out[i] = sum;
}
```

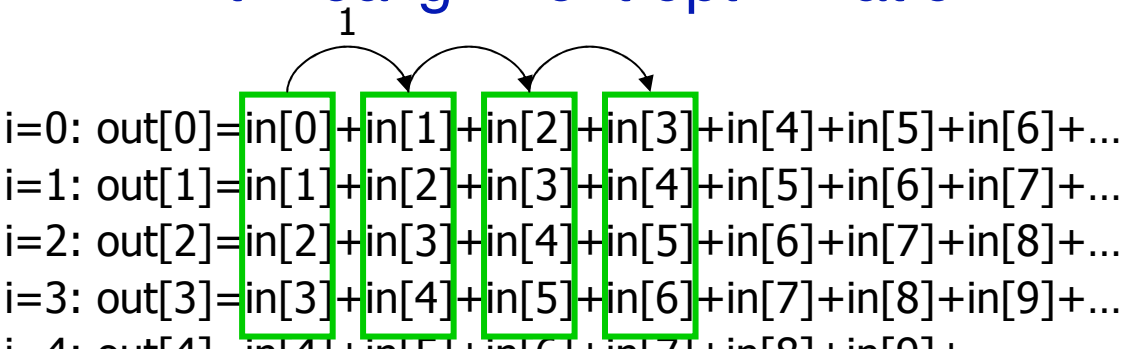
Outer-Loop Vectorization

```
for (i = 0; i < N/4; i++) {
    float vsum = 0;
    for (j = 0; j < M; j++) {
      vsum  += in[j+i:j+i+3];
    }
    out[i:i+3] = vsum;
}
```

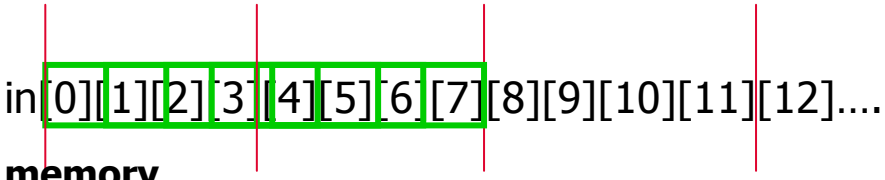
Outer-Loop Vectorization: Why?

- Correctness
  - Cross iteration data-dependences
  - Non-Associative Reduction
- Profitabilty
  - Elimination of reduction epilog overhead
  - Larger portion of the code vectorized
    - More register reuse / less memory bandwidth
  - Loop vectorization may destroy perfect nests
  - Less “Per-loop” overheads
  - Longer iteration count in outer-loop
    - Multimedia: short-trip innermost loops
  - Smaller strides on outer-loop level
    - Better spatial locality in outer-loop
  - Unknown stride in innermost loop

In-Place Outer-Loop Vectorization for SIMD with realignment optimization



- Changing misalignment




```
for (i = 0; i < N; i++) {
    float sum = 0;
    for (j = 0; j < M; j++) {
      sum  += in[j+i];
    }
    out[i] = sum;
}
```

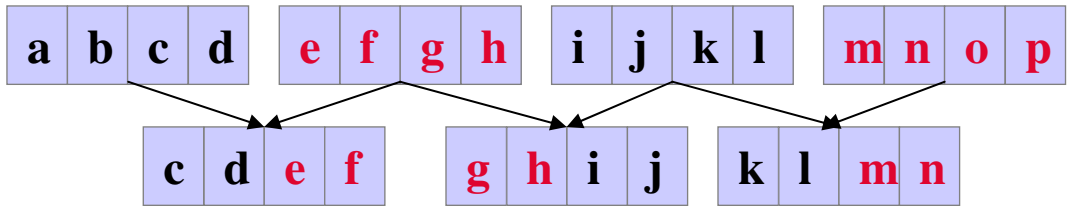
In-Place Outer-Loop Vectorization for SIMD with realignment optimization – Cont.

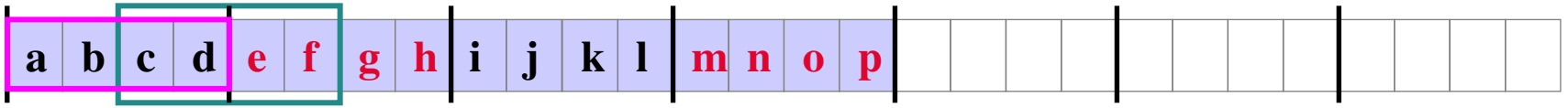
```
      p = &x[4*i];
Loop:
      mis = p & 0x3
      v1 = vload (|p|);
      v2 = vload (|p+4|);
      t = realign (v1,v2,mis);
      …
      p+=4;
      if … goto Loop;
```

Unoptimized realignment

```
      p = &x[4*i];
      mis = p & 0x3;
      v1 = vload (|p|);
Loop:
      v2 = vload (|p+4|);
      t = realign (v1,v2,mis);
      …
      p+=1; v1 = v2;
      if … goto Loop;
```

Optimized (fixed misalignment)





Data in Memory

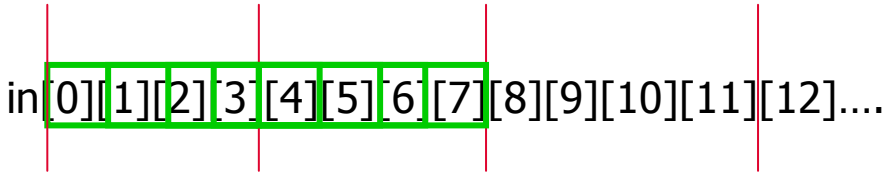
In-Place Outer-Loop Vectorization for SIMD with realignment optimization – Cont.

```
      p = &x[4*i];
Loop:
      mis = p & 0x3
      v1 = vload (|p|);
      v2 = vload (|p+4|);
      t = realign (v1,v2,mis);
      … use t …
      p+=4;
      if …< goto Loop;
```

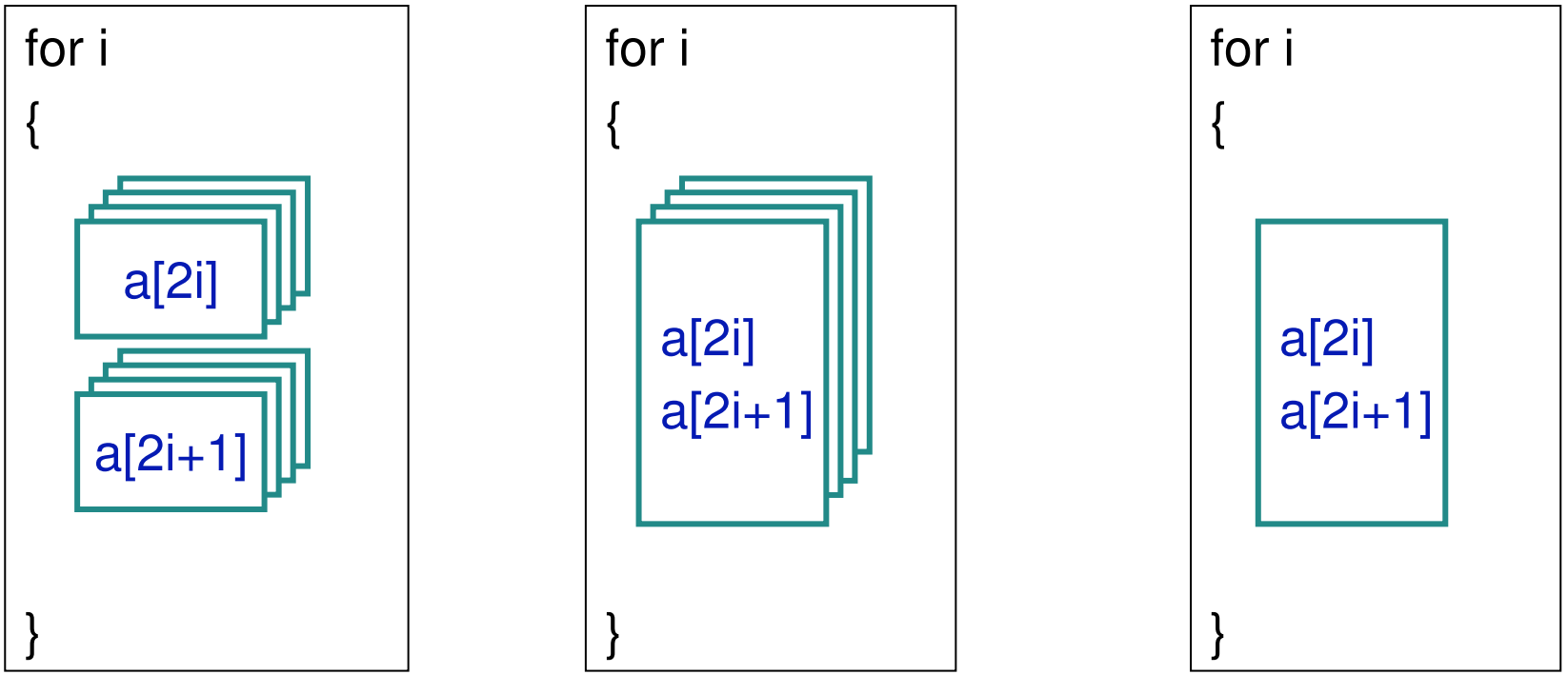
unroll by V/step

```
      p = &x[4*i];
      mis0 = p&0x3;
      mis1 = 4B        #1 elmnt
      mis2 = 8B        #2 elmnts
      mis3 = 12B       #3 elmnts
      v1 = vload (|p|);
      v2 = vload (|p|+4);
      t0 = realign (v1,v2,mis0);
Loop:
      v3 = vload (|p|+8);
      t00 = realign (v2,v3,mis0);
      t1 = realign (t0,t00,mis1);
      t2 = realign (t0,t00,mis2);
      t3 = realign (t0,t00,mis3);
      … use t0 …
      … use t1 …
      … use t2 …
      … use t3 …
      p+=4; v2=v3; t0=t00;
      if … goto Loop;
```

- Changing misalignment



Auto-Vectorization for SPE



Intra loop vectorization

- Classic vectorization techniques exploit inter-iteration parallelism

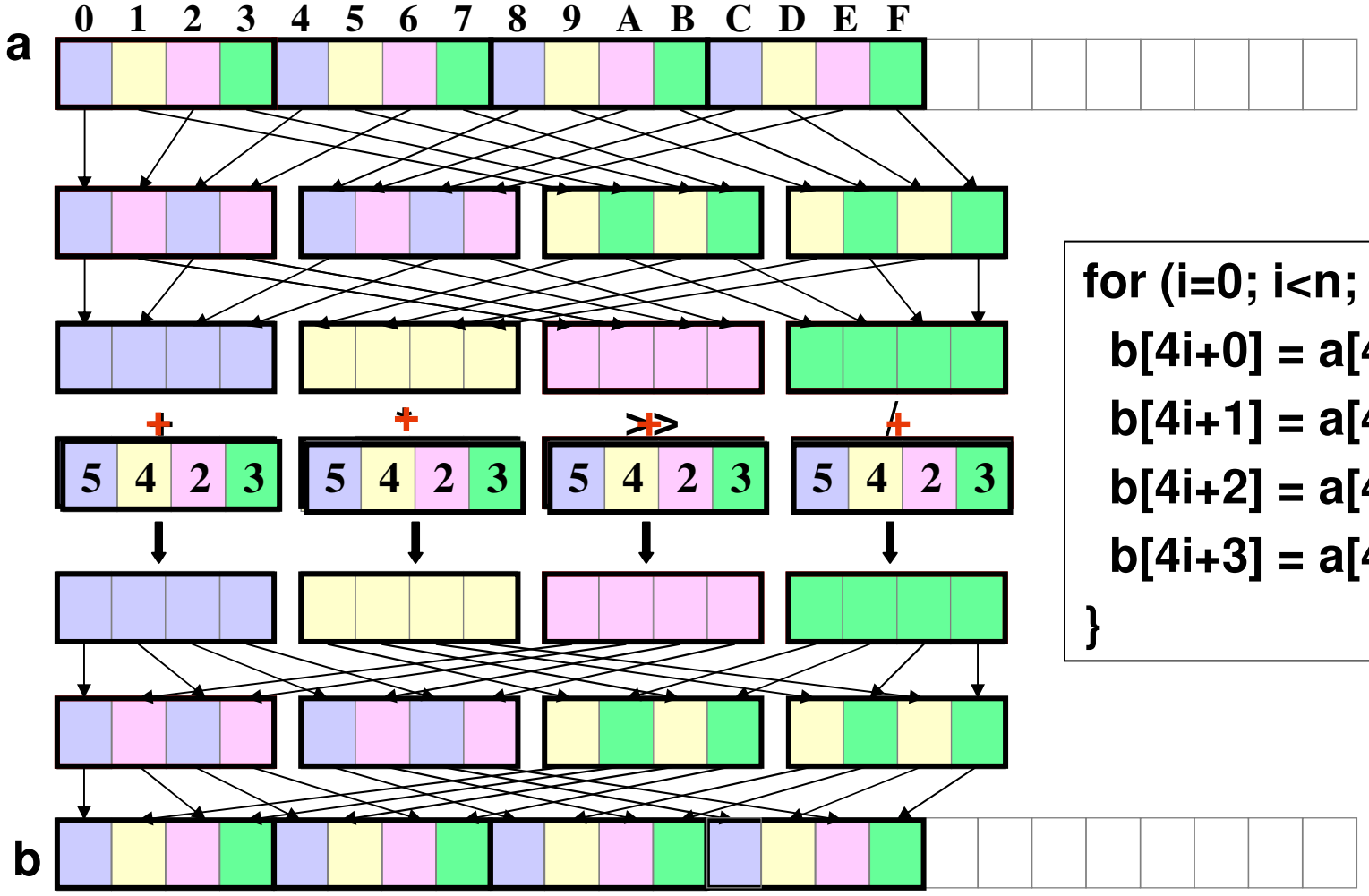
```
do {
  dst[0]  = (src1[0] + src2[0]) >> 1;
  dst[1]  = (src1[1] + src2[1]) >> 1;
  dst[2]  = (src1[2] + src2[2]) >> 1;
  dst[3]  = (src1[3] + src2[3]) >> 1;

  dst += 4; src1 += 4; src2 += 4;
} while (dst != NULL);
```

Uncountable loop

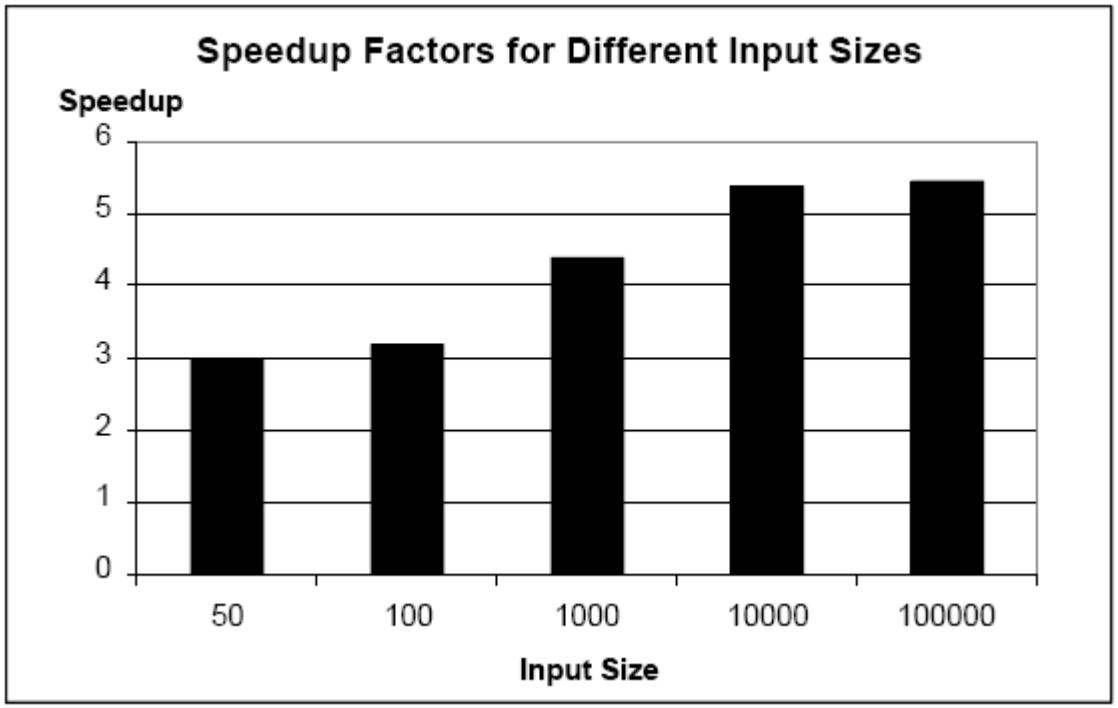
- SLP – straight line code vectorization - Larsen and Amarasinghe, 2000
- Loop-aware SLP - exploit both inter- and intra-iteration parallelism.

Vectorization of Strided Accesses



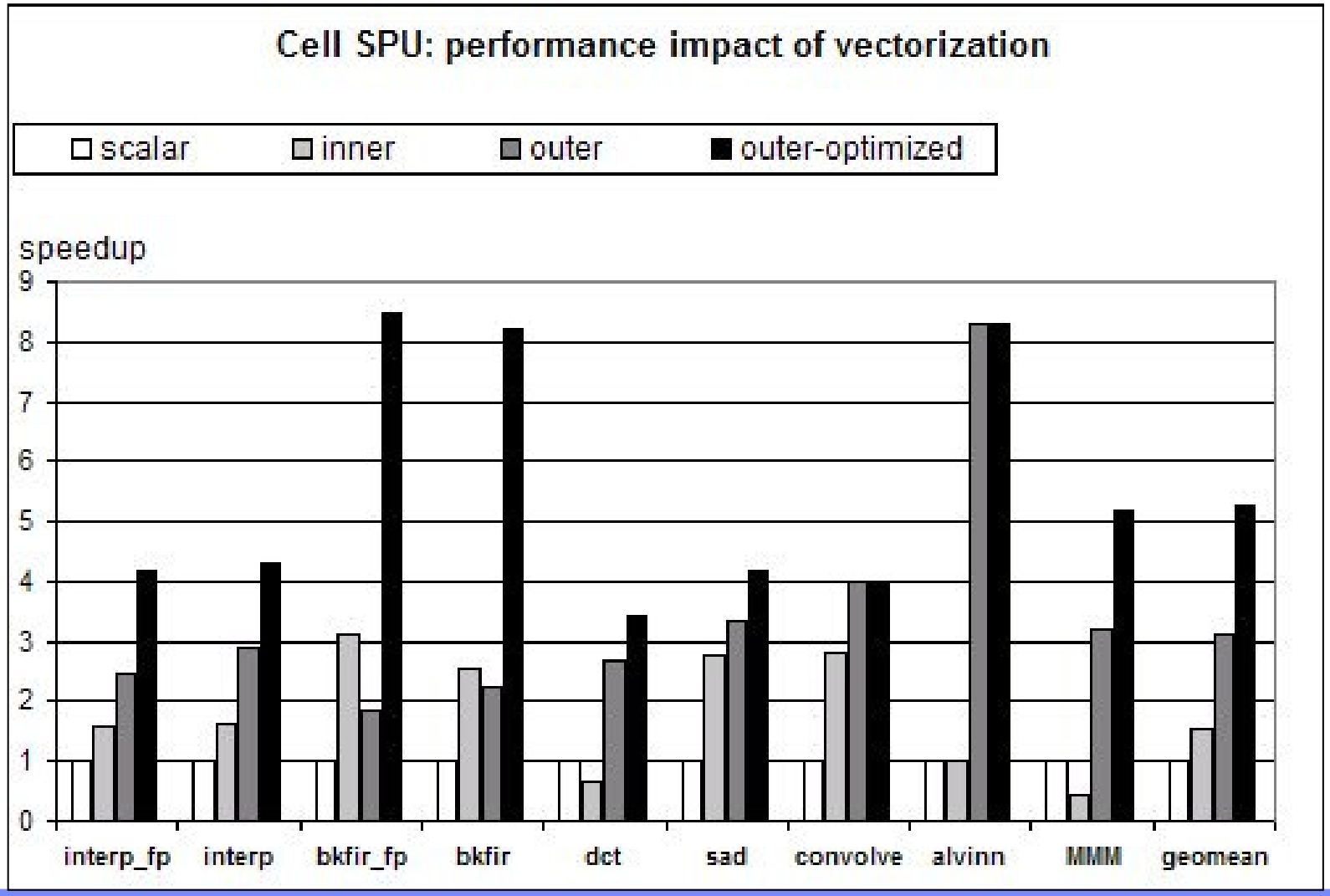

```
for (i=0; i<n; i++) {
  b[4i+0] = a[4i+0] + 5;
  b[4i+1] = a[4i+1] + 4;
  b[4i+2] = a[4i+2] + 2;
  b[4i+3] = a[4i+3] + 3;
}
```

Intra loop vectorization

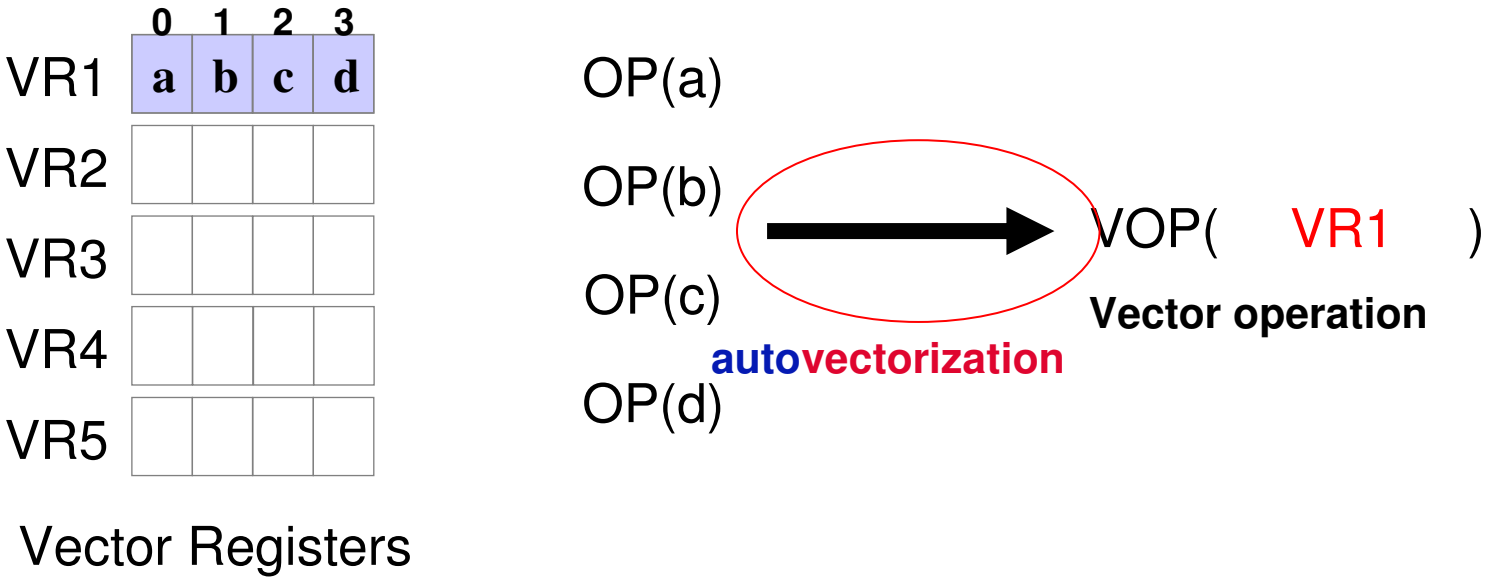


RGB to YIQ conversion

Experimental Results

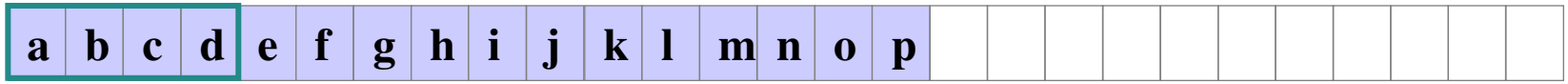


What is vectorization



- Data elements packed into vectors

Data in Memory:



Auto-Vectorization for SPE