Linux on Cell - Compilers

Automatic Simdization for Cell BE using XL Compilers

Course Code: L2T1H1-32

www.research.ibm.com/cellcompiler
Single Instruction Multiple Data (SIMD) Computation

Process multiple “b[i]+c[i]” data per operations

16-byte boundaries
Outline

- Simdization example
- Integrated simdization framework
- How to use the compiler
- Measurements and conclusions
Simdization Framework

Extract Parallelism

for (i=0; i<256; i++)

\[ a[i] = \]

Satisfy Constraints

alignment constraints

multiple targets

SIMDIZER

GENERIC

VMX

SPE
Traditional Execution of a Loop

Sequential execution of “for (i=0; i<65; i++) a[i+2] = b[i+1] + c[i+3] ”

Memory streams
(grey is 1st iteration)
SIMD Alignment Problem

- **SIMD execution of** “for(i=0;i<65;i+) a[i+2] = b[i+1] + c[i+3]”

- SIMD execution of “for(i=0;i<65;i+) a[i+2] = b[i+1] + c[i+3]”

  - vload b[1]
  - offset 4
  - offset 12
  - vload c[3]

  - vadd
  - b0+ c0
  - b1+ c1
  - b2+ c2
  - b3+ c3

  - this is not b[1]+c[3], ... because the alignments do not match
Loop-Level Simdization, Naïve Way

- **SIMD execution of** “for(i=0;i<65;i+) a[i+2] = b[i+1] + c[i+3]”

Memory stream:
- b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10

Register stream:
- b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12

- **StreamShift Left**
- **StreamShift Right**

16-byte boundaries:
- + + +
- b1+ c3 b2+ c4 b3+ c5 b4+ c6 b5+ c7 b6+ c8 b7+ c9 b8+ c10 b9+ c11 b10+ b11+ b12+ c14

16-byte boundaries:
- a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10
Solving the Alignment Problem

- **Data Reorganization Graph**
  - original graph with alignment label each load/store
  - resolve alignment conflicts

```
offset 4
vload b[i+1]

vload c[i+3]
offset 12

vadd

vstore a[i+2]
offset 8

conflict: reaching alignments are not all identical
```
Solving the Alignment Problem

- **Data Reorganization Graph**
  - original graph with alignment label each load/store
  - resolve alignment conflicts by adding “stream-shift” aligning operations

Diagram:
- `vload b[i+1]`
  - offset 4
- `vload c[i+3]`
  - offset 12
- `stream-shift(4,0)`
- `stream-shift(12,0)`
- `vadd` (black)
- `stream-shift(0,8)`
  - offset 0
- `vstore a[i+2]`
  - offset 8
- `stream-shift(0,8)`
  - offset 0

Set alignment of streams to zero

[VAST]
Loop-Level Simdization, Optimized Way

- **SIMD execution of** "for(i=0;i<65;i++) a[i+2] = b[i+1] + c[i+3]"

Diagram illustrating Memory stream and Register stream with 16-byte boundaries. Stream-Shift Left and Stream-Shift Right are shown to demonstrate the execution process.
Optimized Solving of the Alignment Problem, Eager Policy

- **Data Reorganization Graph**
  - eagerly align to store alignment

```
<table>
<thead>
<tr>
<th>vload b[i+1]</th>
<th>vload c[i+3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset 4</td>
<td>offset 12</td>
</tr>
<tr>
<td>stream-shift(4,8)</td>
<td>stream-shift(12,8)</td>
</tr>
<tr>
<td>vadd</td>
<td></td>
</tr>
<tr>
<td>vstore a[i+2]</td>
<td></td>
</tr>
<tr>
<td>offset 8</td>
<td></td>
</tr>
</tbody>
</table>
```

set to store alignment; reduce stream-shift # from 3 to 2

[PLDI04, CGO05]
Optimized Solving of the Alignment Problem, Lazy Policy

- Data Reorganization Graph (slightly different “b[i+1] + c[i+1]” example)
  - lazily align to store alignment

```
$vload \text{b}[i+1]\text{, offset 4}$
$vload \text{c}[i+1]\text{, offset 4}$
$vadd$
$\text{stream-shift}(4,8)$
$vstore \text{a}[i+2]\text{, offset 8}$
```

lazily set to store align; reduce stream-shift # from 3 to 1

[PLDI04, CGO05]
SIMD Code Generation

- SIMD codes are generated from a valid data reorganization graph

- Code generation for simdized loop

  ```c
  <simdized prologue>
  for(i=0; i<65; i+=4)
      <simdized loop body>
  <simdized epilogue>
  ```

- **simdized** loop (steady state)
- **simdized** prologue if store is misaligned
- **simdized** epilogue depending on store alignment and trip-count
When you need a stream of vectors starting at address b[1]

- vload b[1]
  - b0 b1 b2 b3
  - vload b[5]
    - b4 b5 b6 b7
  - vload b[9]
    - b8 b9 b10 b11

offset 4

streamshift(4, 0)

16-byte boundaries
Code Generation for Partial Store (beginning of stream)

\[
\text{for (i=0; i<65; i++) } a[i+2] = b[i+1] + c[i+3];
\]

![Diagram showing code generation process]
Code Generation for Partial Store (end of the stream)

for (i=0; i<65; i++) a[i+2] = b[i+1] + c[i+3];

16-byte boundaries

vstore a[62]

vstore a[66]

vload a[66]

offset 8

16-byte boundaries
**Code Generation for Loops (Multiple Statements)**

```cpp
for (i=0; i<n; i++) {
    a[i] = ...
    b[i+1] = ...
    c[i+3] = ...
}
```

**Implicit loop peeling (steady-state)**

```cpp
da[i+4] = ...
b[i+4] = ...
c[i+4] = ...
```

**loop prologue (simdized)**

```
a0 a1 a2 a3
```

```
b0 \b1 b2 b3
```

```
c0 c1 c2 c3
```

**loop steady state (simdized)**

```
a4 a5 a6 a7 ...
a96 a97 a98 a99
```

```
b4 b5 b6 b7 ...
b96 b97 b98 b99
```

```
c4 c5 c6 c7 ...
c96 c97 c98 c99
```

**loop epilogue (simdized)**

```
a100 a101 a102 a103
```

```
b100 b101 b102 b103
```

```
c100 c101 c102 c103
```
Machine-Independent Code After Simdization

- Pseudo codes after simdization (no software pipelining on adjacent loads)
  - all operations are vectors
  - loads/stores normalized to truncated address
  - splice, shiftpairl, and shiftpairr are pseudo data reorganization operations to be mapped to spu_shuffle or spu_sel

```c
i = 0;
a[i] = splice(a[i], shiftpairr(b[i-4], b[i], 4) + shiftpairl(c[i-4], c[i], 4), 8);
i = 4;
do {
    a[i] = shiftpairr(b[i-4], b[i], 4) + shiftpairl(c[i-4], c[i], 4);
    i = i + 4;
} while (i < 65-4);
a[i] = splice(shiftpairr(b[i-4], b[i], 4) + shiftpairl(c[i-4], c[i], 4), a[i], 12));
```
Machine-Dependent Intrinsic Codes after Simdization

- after software pipelining, loop normalization, and address truncation
- spu_shuffle, spu_sel, spu_add, spu_mask are SPU intrinsics
- \(<0x08090a0b, \ldots>\) is a vector literal

```c
a[0] = spu_sel(a[0], spu_add(spu_shuffle(b[-4], b[0], <0x08090a0b, 0x0c0d0e0f, 0x10111213, 0x14151617>),
spu_shuffle(c[-4], c[0], <0x0c0d0e0f, 0x10111213, 0x14151617, 0x18191a1b>), spu_maskb(15));
oldSPCopy0 = c[0];
oldSPCopy1 = b[0];
i = 0;
do {
    tc = c[i*4+4];
    tb = b[i*4+4];
    a[i*4+4] = spu_add(spu_shuffle(oldSPCopy1, tb, <0x08090a0b, 0x0c0d0e0f, 0x10111213, 0x14151617>),
spu_shuffle(oldSPCopy0, tc, <0x0c0d0e0f, 0x10111213, 0x14151617, 0x18191a1b>);
    oldSPCopy0 = tc;
    oldSPCopy1 = tb;
    i = i + 1;
} while (i < 24);

a[100] = spu_sel(spu_add(spu_shuffle(b[96], b[100], <0x08090a0b, 0x0c0d0e0f, 0x10111213, 0x14151617>),
spu_shuffle(c[96], c[100], <0x0c0d0e0f, 0x10111213, 0x14151617, 0x18191a1b>), a[100], spu_maskb(15));
```
Outline

- Simdization example
- Integrated simdization framework
- How to use the compiler
- Measurements and conclusions
Successful Simdizer

**Extract Parallelism**

- **loop level**
  - for (i=0; i<256; i++)
  - \( a[i] = \)

- **basic-block level**
  - \( a[i+0] = \)
  - \( a[i+1] = \)
  - \( a[i+2] = \)
  - \( a[i+3] = \)

- **entire short loop**
  - for (i=0; i<8; i++)
  - \( a[i] = \)

**Satisfy Constraints**

- **alignment constraints**
  - 16-byte boundaries
  - \( b_0 \rightarrow\) \( b_2 \rightarrow\) \( b_3 \rightarrow\) \( b_4 \rightarrow\) \( b_5 \rightarrow\) \( b_6 \rightarrow\) \( b_7 \rightarrow\)

- **data size conversion**
  - SHORT
  - \( \text{load } b[i] \rightarrow\) \( \text{unpack} \rightarrow\) \( \text{add} \rightarrow\) \( \text{store} \)
  - INT

- **multiple targets**

- **non stride-one**

- **SIMDIZER**

- **GENERIC**
  - VMX
  - SPE
Multiple Sources of SIMD Parallelism

- **Loop level**
  - SIMD for a single statement across consecutive iterations
  - successful at:
    - efficiently handling misaligned data
    - pattern recognition (reduction, linear recursion)
    - leverage loop transformations in most compilers
    - amortize overhead (versioning, alignment handling) and employ cost models

---

[Bik et al, IJPP 2002]
[VAST compiler, 2004]
[Eichenberger et al, PLDI 2004] [Wu et al, CGO 2005]
[Naishlos, GCC Developer’s Summit 2004]
Multiple Sources of SIMD Parallelism (cont.)

- **Basic-block level**
  - SIMD across multiple isomorphic operations
  - Successful at
    - Handling unrolled loops (manually or by compiler)
    - Extracting SIMD parallelism within structs, e.g.
      
      ```
      for (i=0; i<8; i++)
      a[i] = a[i].x = a[i].y = a[i].z = 
      s += a(i)*b(i) + a(i+1)*b(i+1) + a(i+2)*b(i+2) + a(i+3)*b(i+3) + a(i+4)*b(i+4)
      ```

  - Extracting SIMD parallelism within a statement

[Larsen et al, PLDI 2000]
[Shin et al, PACT 2002]
Multiple Sources of SIMD Parallelism (cont.)

- **Short-loop level**
  - SIMD across entire loop iterations
  - effectively collapse innermost loop
  - we can now extract SIMD at the next loop level
  - e.g. FIR

```c
for (k=0; k<248; k++)
    for (i=0; i<8; i++)
        res[k] += in[k+i] * coef[k+i];
```

```
for (i=0; i<256; i++)
a[i] =
```
Multiple SIMD Hardware Constraints

- **Alignment in SIMD units matters**
  - when alignments within inputs do not match
  - must realign the data

![Diagram showing SIMD operations and alignment constraints](image-url)
Multiple SIMD Hardware Constraints (cont.)

- Size of data in SIMD registers matters
  - When converting from short to integer:
    - We must issue 2x integer SIMD operations

- E.g. when converting from short to integer:
  - We must issue 2x integer SIMD operations

- Data size conversion
  - Alignment constraints
  - Non stride-one
Multiple SIMD Hardware Constraints (cont.)

- Hardware supports 16-byte continuous access only
- Non stride-one load requires packing

For \(i = 0; i < n; i++\)

\[... = a[i] + b[2i+1]\]

- Non stride-one store requires unpacking
Different platforms have varying SIMD support
- e.g. VMX / SPE have SIMD permute instructions
- e.g. SPE has no memory page fault, VMX does
How to support the cross product of all these?

Loop level:

For (i=0; i<256; i++)

a[i] =

Basic-block level:

a[i+0] =
a[i+1] =
a[i+2] =
a[i+3] =

Entire short loop:

For (i=0; i<8; i++)
a[i] =

Alignment constraints:

Data size conversion:

Non stride-one:

Multiple targets:

Generic

Vmx

SPE
Key Abstraction: Virtual SIMD Vector

- **Virtual SIMD Vector**
  - has arbitrary length
  - has no alignment constraints

- **Extraction of SIMD Parallelism**
  - use virtual vector as representation
  - abstract away all the hardware complexity

- **Progressive “de-virtualization” of the virtual vector**
  - until each vector in the loop satisfies all hardware constraints
  - or revert vectors back to scalars (if too much overhead)
Integrated Simdization Framework

Characteristics:
- Modular
- Integrated
- Hide complexity

Global Pointer Analysis

Constant Propagation

Dependence Elimination

Idiom Recognition

Data Layout Optimization

Basic-block level aggregation

Short-loop level aggregation

Loop-level aggregation

Alignment devirtualization

Length devirtualization

SIMD Codegen

SIMD Extraction

Virtual Vectors
- arbitrary length
- arbitrary alignment

Transition
- alignment handling
- short length handling

Physical Vectors
- 16 bytes long
- machine alignment

Machine Specific
First Example: Basic-Block & Loop Level Aggregation

Original loop

```c
for (i=0; i<256; i++) {
    a[i].x =
    a[i].y =
    a[i].z =
    b[i+1] =
}
```

Value streams

1st iter 2nd iter 3rd iter 4th iter

4 iterations shown here for the purpose of illustration

1st iter 2nd iter 3rd iter 4th iter
Phase 1: Basic-Block Level Aggregation

**Original loop**
```
for (i=0; i<256; i++) {
    a[i].x = a[i].y = a[i].z = b[i+1] =
}
```

**Value streams**

- Pack `a[i].x, a[i].y, a[i].z` into a vector of 3 elements
- Pack regardless of alignment

**Basic-Block level aggregation**
- Pack `a[i].x, a[i].y, a[i].z` into a vector of 3 elements
- Pack regardless of alignment
**Phase 2: Loop-Level Aggregation**

**BB-aggregated loop**

```c
for (i=0; i<256; i++) {
    (a[i].x,y,z) = 
    b[i+1] =
}
```

**Value streams**

- Pack each statement with itself across consecutive iterations
- Final vector lengths must be multiple of 16 bytes
- Scalar “b[i]” or vector “(a[i].x,y,z)” are treated alike
- Pack regardless of alignment

**Loop-level aggregation**

- Pack each statement with itself across consecutive iterations
- Final vector lengths must be multiple of 16 bytes
- Scalar “b[i]” or vector “(a[i].x,y,z)” are treated alike
- Pack regardless of alignment
Phase 3: Alignment Devirtualization

Loop-aggregated

Value streams

for (i=0; i<256; i+=4) {
    (a[i].x,…,a[i+3].z) =
    (b[i+1],…,b[i+4]) =
}

align access

Alignment *

- shift misaligned streams
- skew the computations so that loop computes (b[i+4]…b[i+7])

* Arrays (e.g. &a[0], &b[0],…) are assumed here 16-byte aligned.
Phase 4: Length Devirtualization

Aligned loop

(b[1]...b[3]) = 
for (i=0; i<252; i+=4) {
(a[i].x,...,a[i+3].z) =
(b[i+4],...,b[i+7]) =
}
(a[252].x,...,a[255].z) =
b[256] =

Value streams

Length

➢ break into 16-byte chunks

a0.x a0.y a0.z a1.x  a1.y a1.z a2.x a2.y  a2.z a3.x a3.y a3.z
b4  b5  b6  b7
Second Example: Data-Size Conversion and Misalignment

Original loop

```c
for (i=0; i<256; i++) {
    a[i] += (int) b[i+1]
}
```

- **Short computations**
  - Load `b[i+1]`
  - Convert
  - Add
  - Store `a[i]`

- **Integer computations**
  - Load `a[i]`
  - Convert
  - Add
  - Store `a[i]`
Phase 1: Loop-Level Aggregation

Original loop

```c
for (i=0; i<256; i++) {
    a[i] += (int) b[i+1]
}
```

Loop-level aggregation

- pack each statement with itself across consecutive iterations
- virtual vectors have uniform number of elements, even when
  - vector of 8 integer = 32 bytes of data
  - vector of 8 short = 16 bytes of data
Phase 2: Alignment Devirtualization

Original loop

```c
for (i=0; i<256; i++) {
    a[i] += (int) b[i+1]
}
```

Alignment

- shift misaligned streams
- easy to do as we are still dealing with long vectors
Phase 3: Length Devirtualization

Original loop

```
for (i=0; i<256; i++) {
    a[i] += (int) b[i+1]
}
```

Length

- 8 shorts fit into a 16-byte register
- 8 integers do not fit; must replicate integer registers and associated instructions
Outline

- Simdization example
- Integrated simdization framework
- How to use the compiler
- Measurements and conclusions
The XL Compiler Architecture

IPA IPA

Objects Objects

System System

Linker Linker

Optimized Optimized

Objects Objects

EXE

DLL

C FE

C++ FE

FORTRAN

FE

-06 (Compiled path)

Wcode

Wcode

Wcode

Wcode+

IPA IPA

Objects Objects

TOBEOY

Wcode

Wcode

Wcode+

IPA IPA

Objects Objects

FPU FPU

CELL CELL

VMX VMX

Libraries Libraries

PDF info PDF info

Instrumented Instrumented runs runs

Optimized Optimized Objects Objects

Other Other Objects Objects

System System Linker Linker

EXE

DLL

-O4 (Compile time) path -O5 (Link-time) path
Where does Simdization Occur?

- **All simdization occurs in TPO (high-level inter-procedural optimizer)**
  - computations that stream over char/short/int/float/double

- **Other important optimizations occurs in TPO**
  - data locality optimization
  - loop fusion/fission
  - inlining/cloning...

- **Low level optimization are done by Tobey (low-level backend optimizer)**
  - Tobey does most code motion/scheduling/machine specific optimizations

This talk focus mainly on TPO simdization
Simdization Optimization Process

1. Compile for the right machine
   - spuxlc for spu, ppuxlc for ppe

2. Turn on the right optimizations
   - -O5 (link-time, whole-program analysis & simdization)
   - -O4 (compile time, limited scope analysis & simdization)
   - -O3 -qhot=simd (compile time, less optimization & simdization)

3. Tune your programs
   - use TPO compiler feedback to guide you
     - -qreport gives information on what loop is simdized or not
     - -qxflag=diagnostic provides much more info help the compiler with extra info
       (directive/pragmas)
   - modify algorithms (hint: more stride-one memory accesses)

*non supported option
Program to Disable Simdization

- **Turn off the right optimizations**
  - do not invoke TPO
    - compile with `-O3` without `"-qhot"` or `"-qipa"`
    - or add `–qhot=nosimd` at `-O4`, `-O5`

- disable simdization (with at least `–O3` `–qhot=simd`)
  - for a loop: `#pragma nosimd`  |  `!IBM* NOSIMD`
  - completely: `–qhot=nosimd` or `–qdebug=nosimd`

`green is for C | red is for fortran`
Examples of TPO Simdization Success Diagnostic

Examine loop <1> on line 12
(simdizable) []

Examine loop <2> on line 20
(simdizable) [misalign(compile time) shift(3 compile-time)]

Examine loop <3> on line 26
(simdizable) [misalign(runtime)][versioned(relative-align)]
TPO Diagnostic Information on Success

- **Simdizable loops**
  - diagnostic reports ",(simdizable)[features][version]“

- **[feature]** further characterizes simdizable loops
  - “misalign(compile time store)”: simdizable loop with misaligned accesses
  - “shift(4 compile time)”: simdizable loop with 4 stream shift inserted
  - “priv”: simdizable loop has private variable
  - “reduct”: simdizable loop has a reduction construct

- **[version]** further characterizes if/why versioned loops were created
  - “relative align”: versioned for relative alignment
  - “trip count”: versioned for short runtime trip count
Examples of TPO Simdization Failure Diagnostic

Examine loop <id=1> on line 1647
contains function call
(non_simdizable)

Examine loop <id=1> on line 2373
dependence at level 0 from ( 0 73 100 )
(non_simdizable)

Examine loop <id=2> on line 2356
dependence due to aliasing
(non_simdizable)

Examine loop <1> on line 4
no intrinsic mapping for <ADD int>: a[]0[$.CIV0] + b[]0[$.CIV0]
(non_simdizable)
TPO Diagnostic Information on Failure

Alignment:
- "misalign(...)": SIMDizable loop with misaligned accesses
  - "non-natural": non naturally aligned accesses
  - "runtime": runtime alignment

Action:
- Align data for the compiler:
  ```
  double a[256] __attribute__((aligned(16)));
  ```
  - all dynamically allocated memory (malloc, alloca) are 16-byte aligned
  - all global objects are 16-byte aligned
  - inside struct / common block, you are on your own
- Tell the compiler it's aligned:
  ```
  __alignx(16, p); | call alignx(16,a[5]);
  ```
  - like a function call, no code is issued
  - can be placed anywhere in the code, preferably close to the loop
- Tell compiler that all references are naturally aligned
  - -qxflag=nataligned
- Use array references instead of pointers when possible

green is for C | red is for fortran
TPO Diagnostic Information on Failure (Cont’)

Structure of the loop
- "irregular loop structure (while-loop)“ (handle only for/do loops)
- "contains control flow“: (no if/then/else allowed)
- “contains function call“: (no function calls)
- “trip count too small“: (short loops not profitable)

Action:
- convert while loops into do loops when possible
- limited if conversion support
  - handle best if-then-else with same array defined on both sides
  - can try data select

Native Mapping and native data types
- “non supported vector element types”
- “no intrinsic mapping for <op type>“
TPO Diagnostic Information on Failure (Cont’)

- Dependence
  - “dependence due to aliasing”

  ⇒ Action:
  - help the compiler with aliasing info
    - use -O5 (does interprocedural analysis)
    - tell the compiler when its disjoint: #pragma disjoint (*a, *b)
    - use fewer pointers when possible

- Array references
  - “access not stride one”:
  - “mem accesses with unsupported alignment”

  ⇒ Action:
  - interchange the loops to enhance stride-one, when possible
  - change algorithm (e.g., split structure)
Things to Look Out for

- **Common causes of non-simdizable loops**
  - non stride-one accesses (e.g., matrix transpose, sparse codes)
  - data type double (VMX)
  - float not simdizable under -qstrict
  - dependence, control-flow, function calls

- **Common performance roadblocks for simdizable loops**
  - Alignment (unknown alignment, misalignment)
    - e.g., alignment of function parameters may be unknown, odd dimension arrays
    - permutation overhead and register pressure
    - compile-time vs. runtime alignment, compile-time vs. runtime shift
  - Spilling/high register pressure due to addressing code or alignment (VMX)
  - Short runtime trip count
    - prologue/epilogue loop overhead
  - VMX AIX ABI register saving convention
Feature Summary

- **Loop simdization**
  - inner loops with stride-one access

- **Basic-block simdization**
  - e.g., unrolled loops

- **Memory alignment**
  - advanced handling of misalignment
  - alignment versioning
  - global pointer alignment analysis

- **Idiom recognition**
  - e.g., average, max, min

- **Mixed-mode simdization**
  - mixing scalar and SIMD computation

- **Redundant integer conversion elimination**
  - maximize SIMD computation bandwidth
What coding choices impact simdization?

- **How loops are organized**
  - The simdizable loop should be the innermost loop
  - While-loops not simdizable
  - Prefer constant trip count, or trip count > 3 * # of element per vector
  - Loops with if-statement less likely simdizable

- **How data is accessed and laid out in memory**
  - Must be stride-one accesses, stride/random/indirect accesses not simdizable
  - Layout the data to maximize aligned accesses
  - Use arrays, pointer arithmetic are bad

- **Dependences inherent to the algorithm**
  - Loops with inherent dependences not simdizable yet
  - Avoid using pointers, potential aliasing may confuse compiler

- **Computation involved**
  - Operations with no native vector instructions less likely simdizable
  - Function calls not simdizable
Other Tuning

- **Loop unrolling can interact with simdization**
  - there is some support for simdizing unrolled loop, but it's harder
  - try to not manually unroll the loop for better TPO simdization
  - unroll directive: #pragma nounroll | #pragma unroll(2)

- **Math libraries:**
  - currently, we don’t simdize sqrt,…
    - we split the loop, simdize the one without sqrt
    - you can do the same, short loop that compute all the sqrt, store in a temp array
    - use optimized libraries to compute vectors of sqrt
    - then use it in the old loop, that one will simdize

- **Use literal constant loop bounds**
  - e.g. #define when possible

- **Tell compiler not to simdize a loop if not profitable (e.g., trip count too low)**
  - #pragma nosimd (right before the innermost loop)
Outline

- Simdization example
- Integrated simdization framework
- How to use the compiler
- Measurements and conclusions
Simdization Results

![Bar chart showing speedup factors for various benchmarks after simdization. The benchmarks include Linpack, Swim-I2, FIR, Autcor, Dot Product, Checksum, Alpha Blending, Saxpy, Mat Mult, and the average speedup. The speedup factors range from 2.4 to 26.2.]

- Linpack: 2.4
- Swim-I2: 2.5
- FIR: 2.9
- Autcor: 2.9
- Dot Product: 7.5
- Checksum: 8.1
- Alpha Blending: 11.4
- Saxpy: 25.3
- Mat Mult: 26.2
- Average: 9.9

single SPE, optimized, automatic simdization vs. scalar code
Conclusion

- **Integrated, modular approach to simdization**
  - extract SIMD parallelism at multiple levels
  - handle efficiently constraints such as alignment, data conversion, ...
  - target multiple ISAs, including VMX & SPU
Questions

For additional info:

www.research.ibm.com/cellcompiler