Auto-SIMDization Challenges

Amy Wang, Peng Zhao,
IBM Toronto Laboratory

Peng Wu, Alexandre Eichenberger
IBM T.J. Watson Research Center
Objective:

What are the new challenges in SIMD code generation that are specific to VMX?

(due to lack of time....)

- Scalar Prologue/Epilogue Code Generation (80% of the talk)
- Loop Distribution (10% of the talk)
  - Mixed-Mode SIMDization
- Future Tuning Plan (10% of the talk)
Background:

Hardware imposed misalignment problem

- More details in the CELL tutorial Tuesday afternoon
Single Instruction Multiple Data (SIMD) Computation

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

R1

b0 b1 b2 b3

R2

c0 c1 c2 c3

R3

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

16-byte boundaries
for (i=0; i<n; i++) {
    a[i] = ...;
    b[i+1] = ...;
    c[i+3] = ...
}

Implicit loop skewing (steady-state)

a[i+4] = ...

b[i+4] = ...

c[i+4] = ...

loop prologue (simdized)

loop steady state (simdized)

loop epilogue (simdized)
Code Generation for Partial Store – Vector Prologue/Epilogue

for (i=0; i<100; i++) a[i+3] = b[i+1] + c[i+2];

- Can be complicated for multi-threading and page faulting issues
Multi-threading issue

\[
\text{for (i=0; i<100; i++) } \ a[i+3] = b[i+1] + c[i+2];
\]

Thread 1 performs load of a0-a3

Thread 2 updates a0, while thread 1 is working on shuffling

WRONG!
int K1;
void ptest() {
    int i;
    for (i=0;i<UB;i++) {
        pout0[i+K1] = pin0[i+K1] + pin1[i+K1];
    }
}

After Late-SIMDization

20 | void ptest()
   | {
22 |     if (!1) goto lab_4;
24 |         @CIV0 = 0;
26 |     if (!1) goto lab_26;
28 | @ubCondTest0 = (K1 & 3) * -4 + 16;
29 | @CIV0 = 0;
30 | do { /* id=2 guarded */ /* ~27 */
31 |     /* region = 0 */
32 |     /* bump-normalized */
33 |     if ((unsigned) (@CIV0 * 4) >= @ubCondTest0) goto lab_30;
34 |     pout0[]0[K1 + @CIV0] = pin0[]0[K1 + @CIV0] + pin1[]0[K1 + @CIV0];
36 | lab_30:
37 |     /* DIR LATCH */
38 |         @CIV0 = @CIV0 + 1;
39 | } while (@CIV0 < 4); /* ~27 */
40 | @CIV0 = 0;
42 | lab_26:
if (!1) goto lab_25;

@CIV0 = 0;
do { /* id=1 guarded */ /* ~3 */
    /* region = 8 */
    @V.pout0[]02[K1 + (@CIV0 + 4)] = @V.pin0[]01[K1 + (@CIV0 + 4)] + @V.pin1[]00[K1 + (@CIV0 + 4)];
} while (@CIV0 < 8); /* ~29 */

@CIV0 = 32;

Scalar Epilogue

if (!1) goto lab_28;

@ubCondTest1 = (unsigned) ((K1 & 3) * -4 + 16);
@CIV0 = 0;
do { /* id=3 guarded */ /* ~29 */
    /* region = 0 */
    /* bump-normalized */
    if (((unsigned) (@CIV0 * 4) < @ubCondTest1) goto lab_31;
    pout0[]0[K1 + (24 + @CIV0)] = pin0[]0[K1 + (24 + @CIV0)] + pin1[]0[K1 + (24 + @CIV0)]
} while (@CIV0 < 8); /* ~29 */

SIMD Body

lab_25:

23 | @V.pout0[]02[K1 + (@CIV0 + 4)] = @V.pin0[]01[K1 + (@CIV0 + 4)] + @V.pin1[]00[K1 + (@CIV0 + 4)];
22 | /* DIR LATCH */
    @CIV0 = @CIV0 + 4;
} while (@CIV0 < 24); /* ~3 */

@mainLoopFinalCiv0 = (unsigned) @CIV0;

lab_31:

/* DIR LATCH */
@CIV0 = @CIV0 + 1;

} while (@CIV0 < 8); /* ~29 */

@CIV0 = 32;

...
Scalar Prologue/Epilogue Problems

- One loop becomes 3 loops: Scalar Prologue, SIMD Body, Scalar Epilogue
- Contains an “if” stmt, per peeled stmt, inside the Scalar P/E loops
- If there is one stmt that is misaligned, ‘every statement’ needs to be peeled
- Scalar P/E do not benefit from SIMD computation where as Vector P/E does.
The million bucks questions…

- When there is a need to generate scalar p/e, what is the threshold for a loop upper bound?
- What is the performance difference between vector p/e versus scalar p/e?
Experiments

- Written a gentest script with following parameters:
  - ./gentest -s snum -l lnum -[c/r] ratio -n ub
  - -s: number of store statements in the loop
  - -l: number of loads per stmt
  - -[c/r]: compile time or runtime misalignment, where 0 <= ratio <= 1 to specify the fraction of stmts that is aligned (i.e. known to aligned at quad word boundary during compile time).
  - -n: upper bound of the loop (compile time constant)

- Since we’re only interested in overhead introduced by p/e, load references are relatively aligned with store references. (no shifts inside body)
- Use addition operation
- Assume data type of float (i.e. 4 bytes)
- Each generated testcase is compiled at –O3 –qhot –qenablevmx –arch=ppc970, and ran on AIX ppc970 machine (c2blade24)
- Each testcase is ran 3 times with average timing recorded.
- 10 variants of the same parameters are generated.
With the lowest functional ub of 12 and in the presence of different degree of compile misalignment, it is always good to simdize!

Tobey is able to fully unroll the scalar p/e loops and fold away all the if conditions. (good job!)
When the aligned ratio is below 0.25 (i.e. misaligned ratio is greater than 0.75) at ub12, scalar p/e gives overhead too large that it is not good to simdize.

However, if we raise the ub to 16, it is always good to simdize regardless of any degree of misalignment!

Tobey is still able to fully unroll the scalar P/E loops, but can’t fold away “if”s with runtime condition.
When there is a need to generate scalar p/e, what is the threshold for a loop upper bound? Compile time 12, Run time 16.
Results

In the presence of only compile misalignment, vector p/e is always better than scalar p/e

Improvement:

Since every stmt is peeled, those that we have peeled a quad word may still be done using vector instructions
In the presence of high runtime misalignment ratio, vector p/e suffers tremendous when it needs to generate select mask using a runtime variable.

It is better to do scalar p/e when misalignment ratio is greater than 0.25!
Answer to the second question

What is the performance difference between vector p/e versus scalar p/e?

- Vector p/e is always better when there is only compile time misalignment. When there is runtime misalignment of greater than 0.25, scalar p/e proves to be better.
Motivating Example

- Not all computations are simdizable
  - Dependence cycles
  - Non-stride-one memory accesses
  - Unsupported operations and data types
- A simplified example from *GSM.encoder*, which is a speech compression application

```c
for (i = 0; i < N; i++) {
    d[i+1] = d[i] + (rp[i] * u[i]);

    t[i] = u[i] + (rp[i] * d[i]);
}
```

Fully simdizable
Current Approach: Loop Distribution

- Distribute the simdizable and non/partially simdizable statements into separated loops (after Loop distribution)
  ```c
  for (i = 0; i < N; i++) {
    1:   d[i+1] = d[i] + rp[i] * u[i];
  }
  for (i = 0; i < N; i++) {
    2:   t[i] = u[i] + rp[i] * d[i];
  }
  ```

- Simdize the loops with only simdizable statements (after SIMDization)
  ```c
  for (i = 0; i < N; i++) {
    1:   d[i+1] = d[i] + rp[i] * u[i];
  }
  for (i = 0; i < N; i+=4) {
    2:   t[i:i+3] = u[i:i+3] + rp[i:i+3] * d[i:i+3];
  }
  ```
Problems with Loop Distribution

- Increase reuse distances of memory references

```c
for (i = 0; i < N; i++) {
    1:   d[i+1] = d[i] + (rp[i] * u[i]);
    2:   t[i]   = u[i] + (rp[i] * d[i]);
}
```

- Only one unit is fully utilized for each loop

```c
for (i = 0; i < N; i++) {
    1:   d[i+1] = d[i] + (rp[i] * u[i]);
}
for (i = 0; i < N; i++) {
    2:   t[i]   = u[i] + (rp[i] * d[i]);
}
```
Preliminary results

- The prototyped mixed mode SIMDization has illustrated a gain of 2 times speed up for the SPEC95 FP swim. With loop distribution, the speed up is only 1.5 times.
Conclusion and Future tuning plan

- **Further improvement on scalar p/e code generation.**
  - Currently, finding out cases when stmt re-execution is allowed. This will allow us to fold away more if conditions
  - More experiments to determine the upper bound threshold for different data types

- **Enable Mixed-mode SIMDization**

- **Integration of SIMDization framework into TPO better**
  - e.g. predicative commoning
Acknowledgement

This work would not be possible without the technical contribution from the following individuals.

- Roch Archambault/Toronto/IBM
- Raul Silvera/Toronto/IBM
- Yaoqing Gao/Toronto/IBM
- Gang Ren/Watson/IBM