JDEV2017

Code Vectorization & Tools

Bertrand Putigny

July 6, 2017

Outline

Vectorization (with OpenMP4)

- What is vectorization?
- Do I need vectorization
- How to "vectorize" code?

Vector Instruction

SIMD: Single Instruction Multiple Data

A0	A1	A2	A3

+

B0	B1	B2	B3

=

A0+B0	A1+B1	A2+B2	A3+B3

SIMD Instructions Sets

- SSE: 128bits
  - 2 double precision reals
  - 4 single precision reals
- AVX: 256bits
  - 4 double precision reals
  - 8 single precision reals
- coming up: AVX-512: 512bits

SIMD is here to stay:

Trends:

- larger vectors
- more instructions (FMA, gather...)

⇒ need to optimize code for SIMD

Vectorization: Using vector instructions

- automatic code vectorization (compiler)
- hand written code
  - intrinsics
  - assembly

$\Rightarrow$ poor portability, hard to write, hard to read

Vectorization: Using vector instructions

- automatic code vectorization (compiler)
- hand written code
  - intrinsics
  - assembly

  $\Rightarrow$ poor portability, hard to write, hard to read

Solution:

Understanding basics of compiler vectorization:

- code transformation
- why vectorization can fail
- help compiler is such cases

Automatic Code Vectorization

Code transformation:

Do the same thing "differently":

- keep the same semantic
- different code versions
- can be done at several level
  - source code level (source to source compilers)
  - intermediate representation (most of the time)
  - instruction level

Code transformation examples:

- instruction scheduling (optimize ILP, at assembly level)
- scalar promotion (IR level)

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        A[i][j] = (1/(double) i) * A[i][j];
    }
}
```

- loop tiling (cache access optimization, most of the time by hand)

Automatic Code Vectorization

Code Transformation:

1. rely on loop unrolling
2. turn set of instructions (scalar) into a single vector instruction

Original code:

```
for(i=0; i<SIZE; i++) {
    y[i] = x[i] + y[i];
}
```

1. Unrolled loop:

```
// peeling (if need be)
for(i=0; i<SIZE-SIZE%4; i+=4) {
    y[i] = x[i] + y[i];
    y[i+1] = x[i+1] + y[i+1];
    y[i+2] = x[i+2] + y[i+2];
    y[i+3] = x[i+3] + y[i+3];
}
// remainder...
```

2. Vectorized pseudo-code:

```
for(i=0; i<SIZE-SIZE%4; i+=4) {
    y[i:i+3] = x[i:i+3] + y[i:i+3];
}
// remainder...
```

Factor Affecting Code Vectorization: Trip Count

Scalar code:

```
for(i=0; i<7; i++) {
    y[i] = x[i] + y[i];
}
```

$\approx$ 7 cycles

Vectorized:

```
for(i=0; i<4; i+=4) {
    y[i:i+3] = x[i:i+3] + y[i:i+3];
}
y[4] = x[4] + y[4];
y[5] = x[5] + y[5];
y[6] = x[6] + y[6];
```

$\approx$ 4 cycles

Vectorized with padding:

```
for(i=0; i<8; i+=4) {
    y[i:i+3] = x[i:i+3] + y[i:i+3];
}
```

$\approx$ 2 cycles

Factor Affecting Code Vectorization: Dependencies

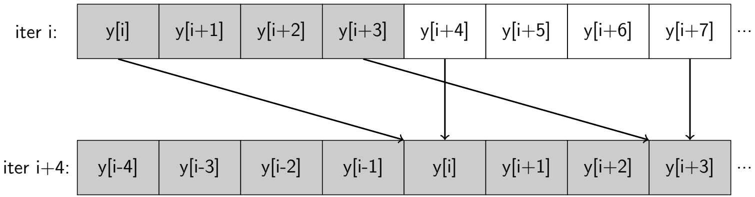
Loop-carried data dependencies:

- cannot be vectorized:

```
for(i=1; i<SIZE; i++) {
    y[i] = y[i-1] - y[i];
}
```

- can be vectorized if vector length $\leq$ 4:

```
for(i=4; i<SIZE; i++) {
    y[i] = y[i-4] - y[i];
}
```



$\Rightarrow$ use OpenMP 4.0 `pragma omp simd safelen(n)`

Factor Affecting Code Vectorization: Aliasing

Pointer Aliasing:

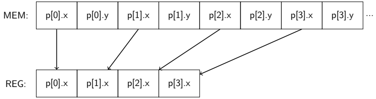
```c
void foo(double *x, double *y, int n) {
    for(i=0; i<n; i++) {
        x[i] = y[i] - x[i];
    }
}

void bar() {
    foo(x, x+1, n-1);
}
```

⇒ use compiler `-fno-alias` option (if you do not use aliasing)
⇒ `#pragma omp simd` (code level, scope)

Factor Affecting Code Vectorization: Data Layout

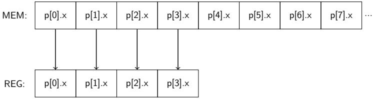
Poor memory access:

```
struct coord {
    double x;
    double y;
};

for(i=0; i<n; i++) {
    points[i].x += v.x;
    points[i].y += v.y;
}
```



Optimal memory access:

```
struct coord {
    double *x;
    double *y;
};

for(i=0; i<n; i++) {
    points.x[i] += v.x[0];
    points.y[i] += v.y[0];
}
```



Factor Affecting Code Vectorization: Control Flow

Conditionals:

```
for(i=0; i<n; i++) {
    if (x[i] > threshold) {
        x[i] = y[i];
    }
}
```

Function calls:

```
for(i=0; i<n; i++) {
    x[i] = f(y[i]);
}
```

⇒ use OpenMP 4.0 `pragma omp declare simd`

Factor Affecting Code Vectorization: Reduction

Sum:

```
r = .0;
for(i=0; i<n; i++) {
    r += x[i];
}
```

$\Rightarrow$ use `pragma omp reduction(+: r)`

Code Samples:

`https://github.com/bputigny/T8.A02`

Conclusion

Code optimization

multiprocessor $\times$ mono processor optimization

Vectorization

- rely on compiler vectorization
- help compiler with:
  - OpenMP 4
  - memory layout
  - code transformation (if need be)

Tools

Help yourself with:

- VTune: multiprocessor profiling
- Intel Advisor: vectorization advisor