AUTO-VECTORIZATION USING AVX

Advanced Programming Tutorial

Date: 10./13.12.2013

Tutor: Felix Rempe

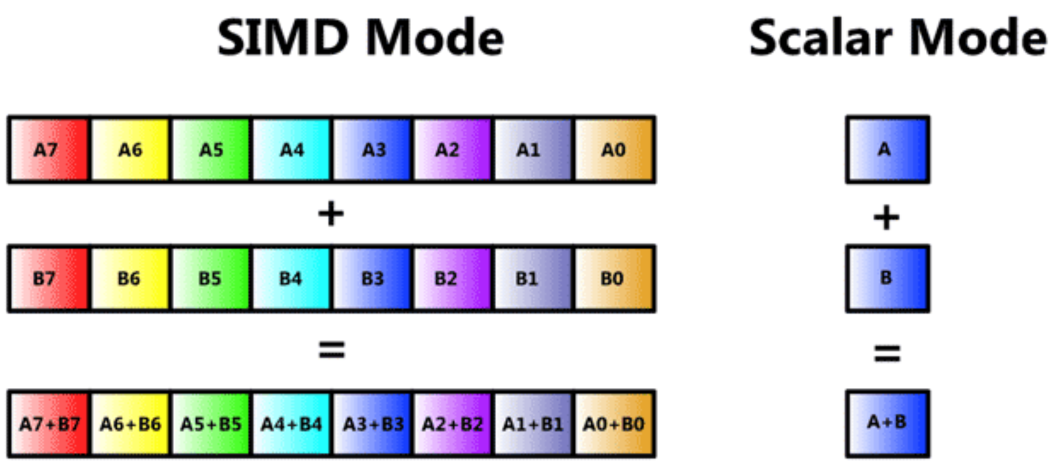
AVX and Auto-Vectorization

- Introduction / Motivation
- How to use
- Auto-Vectorization Requirements
- Useful #pragmas for Vectorization
- Guided Auto-Vectorization
- Summary

INTRODUCTION

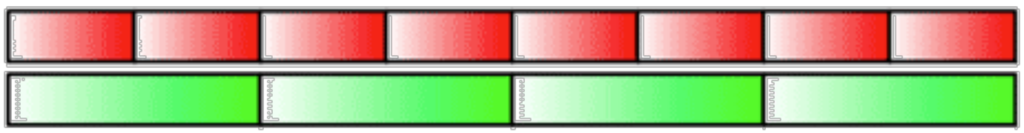
Introduction to AVX

- What is AVX? (Advanced Vector Extensions)
  - Set of instructions to do SIMD commands on Intel architectures (beginning with Sandy Bridge)
  - Previous: SSE (Streaming SIMD Extension)
  - Principle: 1 command for vector operations



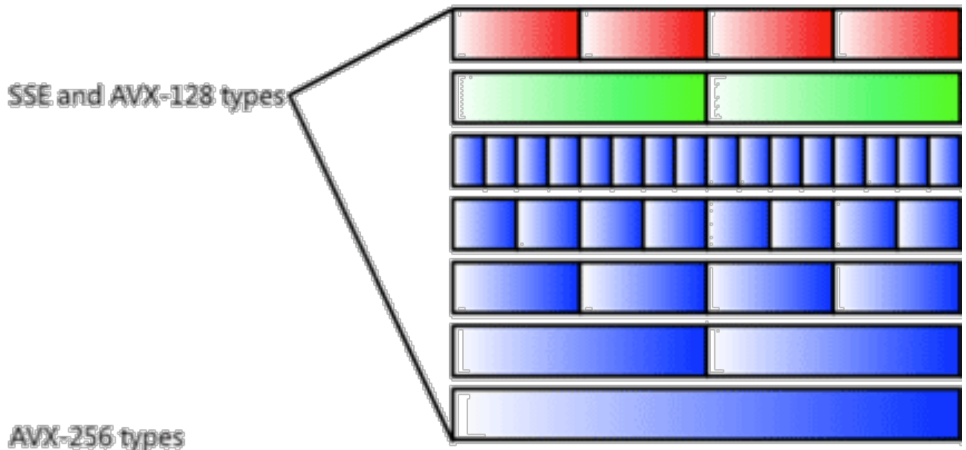
Introduction to AVX

- Features:
  - 256bit registers for operands (only FP)



8x float
4x double

  - 128bit for all other types



4x float
2x double
16x byte
word
doubleword
quadword
doublequadword

Source:http://software.intel.com/

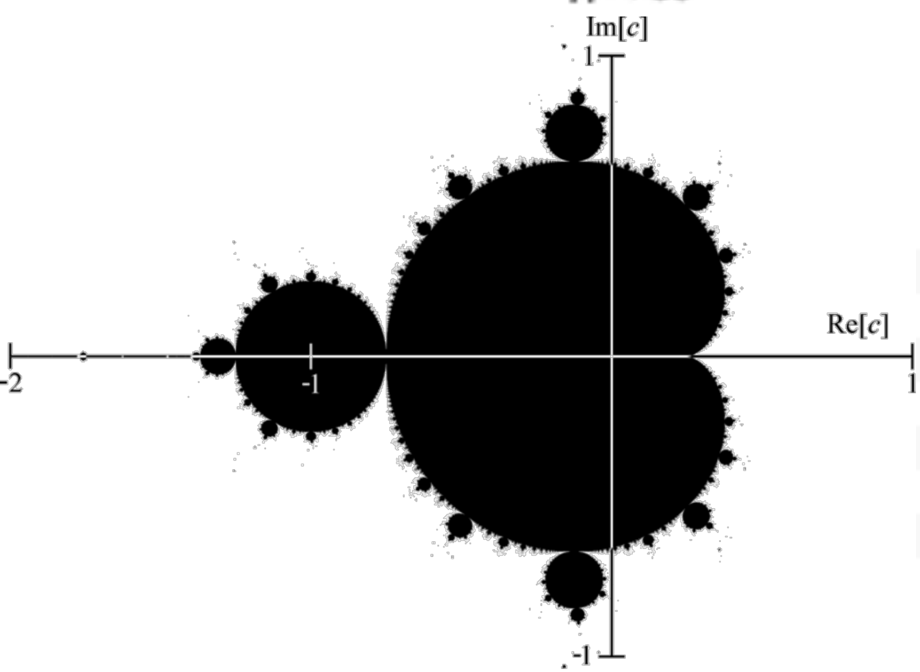
  - Many types of operations available
  - Easily computing in e.g. A = B + C (vector + vector)

Motivation: Example Mandelbrot

- Mandelbrot definition:
  - C: complex
  - N: iterations

$$z_{n+1} = z_n^2 + c$$

All c for which $\lim_{n\to\infty} z_n$ is bounded belong to the Mandelbrot set

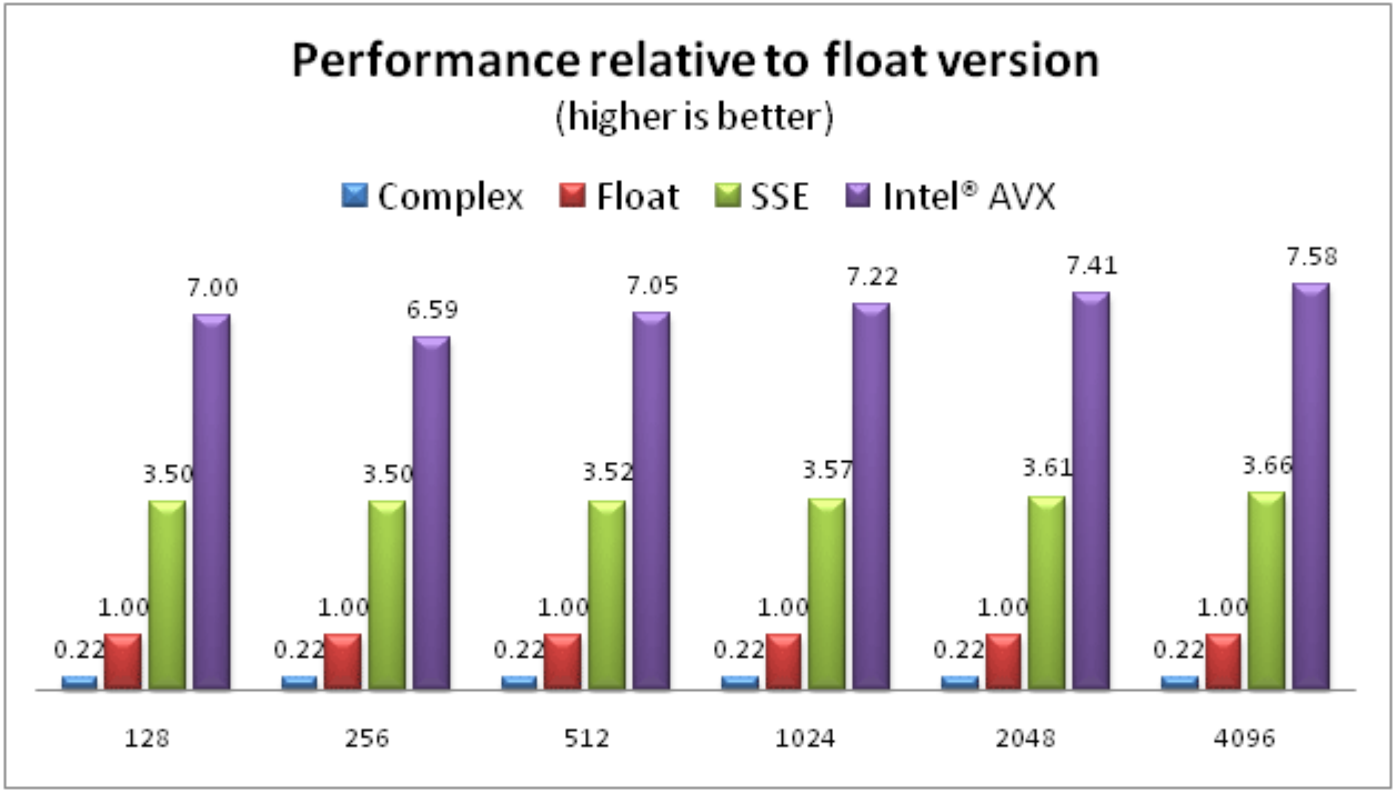


Source:http://de.wikipedia.org/

```
z,p are complex numbers
for each point p on the complex plane
    z = 0
    for count = 0 to max_iterations
        if abs(z) > 2.0
            break
        z = z*z+p
    set color at p based on count reached
```

Motivation: Performance on Mandelbrot

- Tested for different image sizes

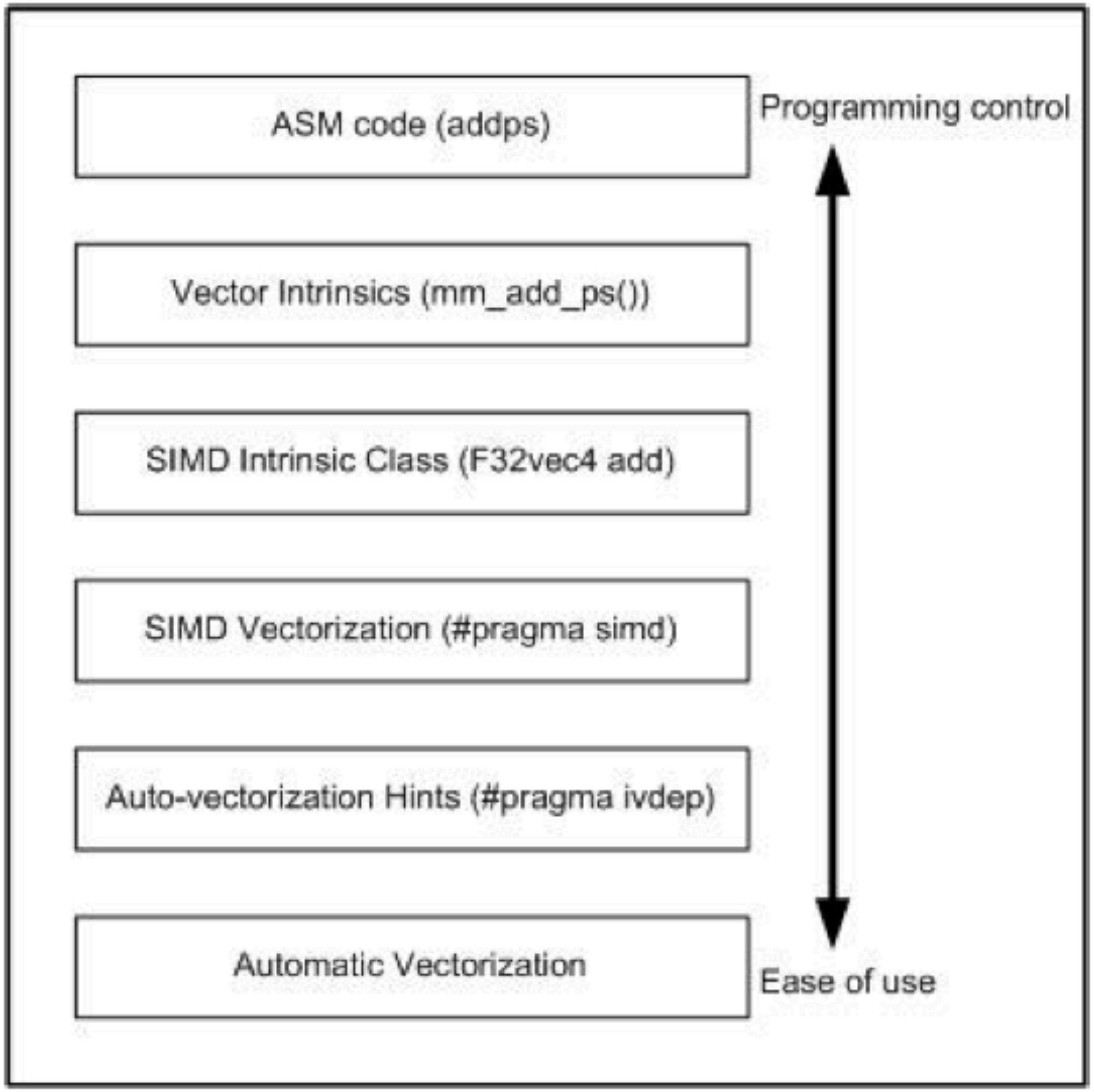


Source:http://software.intel.com/

- =>Great speed-up possible!

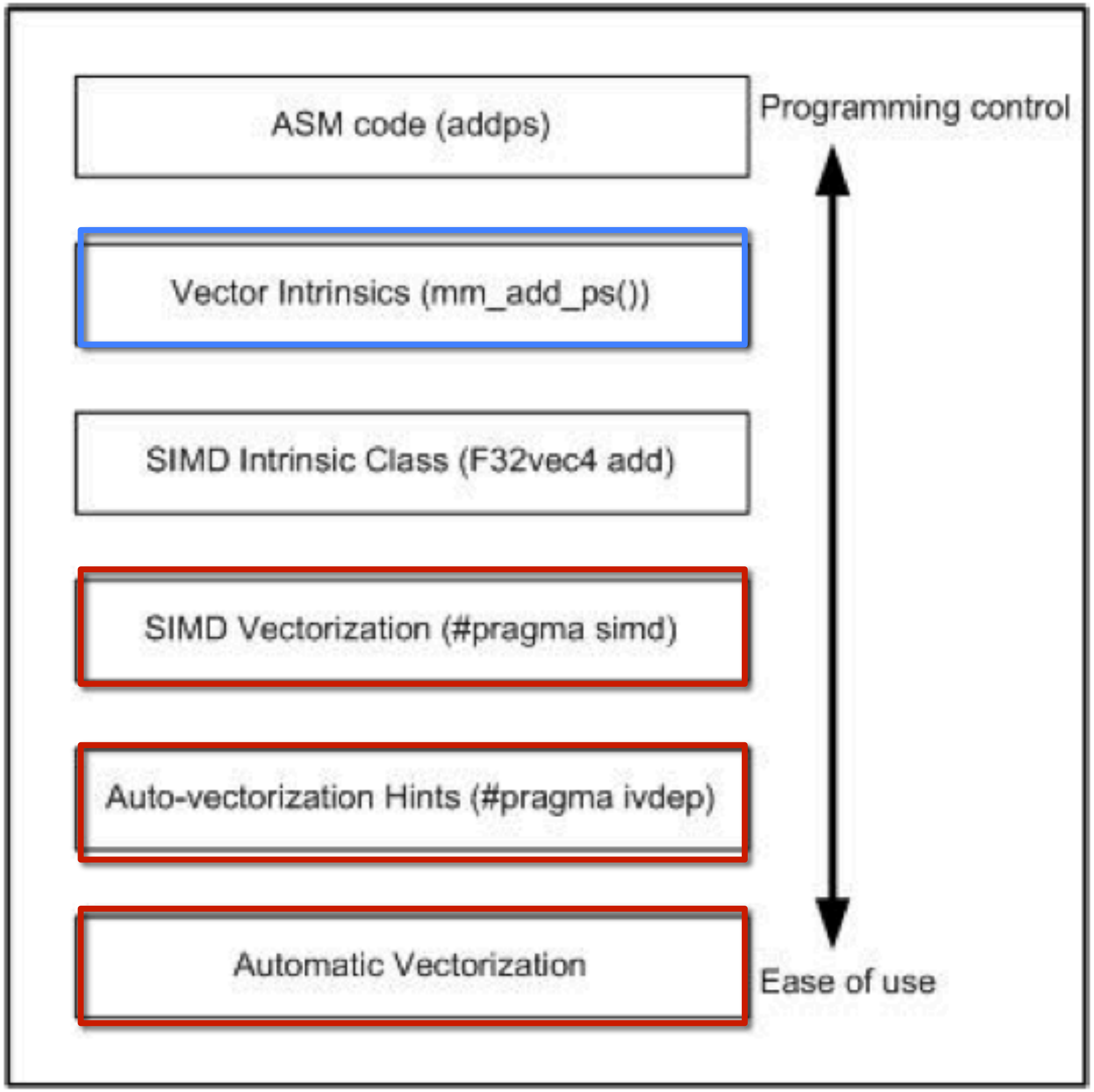
HOW TO USE

How to Apply



Source: Rettenberger, HPC Seminar, TUM, 2012

How to Apply



ASM code (addps)
Vector Intrinsics (mm_add_ps())
SIMD Intrinsic Class (F32vec4 add)
SIMD Vectorization (#pragma simd)
Auto-vectorization Hints (#pragma ivdep)
Automatic Vectorization
Programming control
Ease of use

How to Apply: Vector Intrinsics

- Example: Load a vector and multiply with another one

```
float incr[8]={0.0f,1.0f,2.0f,3.0f,4.0f,5.0f,6.0f,7.0f};
__m256 ymm7  = _mm256_load_ps(incr);
__m256 ymm8 = _mm256_mul_ps(ymm7, ymm0)
```

- In general:

```
_mm256_op_suffix(data_type param1, data_type param2, data_type param3)
```

  - Where:
    - _*mm256* is the prefix for working on the 256bit registers
    - _*op* is the operation
    - _*suffix* is the type of data (e.g. s/d for single, double FP)

Auto-Vectorization

- Let the compiler decide what to do
  - No need to fight with the low level commands
  - Portability of code
- Help the compiler by giving *Vectorization Hints*
  - E.g.

```
#pragma vector always
#pragma ivdep
...
```

- Automatic Vectorization works best if the user is aware of certain mechanisms

Auto-Vectorization: Compilation

- Commonly used compilers: icc and gcc
- Flags determine how/if vectorization is done

- **icc**:
  - Optimization level: O2 or higher
  - xHost – use the best set of optimization flags for architecture (cf. march= native)
  - xavx – use AVX optimization

- **gcc** :
  - Optimization level: O3
  - march=native – (cf. xHost)
  - mavx – use AVX optimization
  - ftree-vectorize – use vectorization
  - Try to print out the available technologies on your system

```
gcc -march=native -E -v - </dev/null 2>&1 | grep cc1
```

AUTO-VECTORIZATION

Things you should be aware of

Requirements for Auto-Vectorization

- 1. The number of iterations must be known before entering the loop
- 2. Single entry and single exit
- 3. Conditional expressions might not be vectorized
- 4. Innermost loop
- 5. No function calls (functions that can be inlined are OK)

Example for Rule No. 5

```
float trap_int (float y, float x0, float xn, int nx, float xp, float yp) {
        float x, h, sumx;
        for (int i=1;i<nx;i++) {
                x = x0 + i*h;
                sumx = sumx + func(x,y,xp,yp);
        }
        sumx = sumx * h;
        return sumx;
}
```

```
float func(float x, float y, float xp, float yp) {

        float denom = (x-xp)*(x-xp) + (y-yp)*(y-yp);
        denom = 1.f/sqrtf(denom);
        return denom;
}
```

> icc -c -vec-report2 trap_integ.c
trap_int.c(16) (col. 3): remark: LOOP WAS VECTORIZED.

G++4.8: note: not vectorized: loop contains function calls or data references that cannot be analyzed

- Is vectorizable since the function can be inlined and “sqrtf” is vectorizable

More Obstacles

- 1. Non-contiguous memory access:
  - Values that are not adjacent in memory need extra effort to be loaded into registers

```
for (int i=0; i<SIZE; i+=2) {
     b[i] += a[i] * x[i];
}
```

```
for (int j=0; j<SIZE; j++)  {
     for (int i=0; i<SIZE; i++) {
          b[i] += a[i][j] * x[j];
     }
}
```

```
for (int i=0; i<SIZE; i+=2) {
     b[i] += a[i] * x[index[i]];
}
```

More Obstacles

- 1. Non-contiguous memory access:
  - Values that are not adjacent in memory need extra effort to be loaded into registers

Stride > 1

```
for (int i=0; i<SIZE; i+=2) {
     b[i] += a[i] * x[i];
}
```

```
for (int j=0; j<SIZE; j++)  {
     for (int i=0; i<SIZE; i++) {
          b[i] += a[i][j] * x[j];
     }
}
```

```
for (int i=0; i<SIZE; i+=2) {
     b[i] += a[i] * x[index[i]];
}
```

More Obstacles

- 1. Non-contiguous memory access:
  - Values that are not adjacent in memory need extra effort to be loaded into registers

Stride > 1

```
for (int i=0; i<SIZE; i+=2) {
     b[i] += a[i] * x[i];
}
```

Innermost loop strides with j

```
for (int j=0; j<SIZE; j++)  {
     for (int i=0; i<SIZE; i++) {
          b[i] += a[i][j] * x[j];
     }
}
```

```
for (int i=0; i<SIZE; i+=2) {
     b[i] += a[i] * x[index[i]];
}
```

More Obstacles

- 1. Non-contiguous memory access:
  - Values that are not adjacent in memory need extra effort to be loaded into registers

Stride > 1

```
for (int i=0; i<SIZE; i+=2) {
     b[i] += a[i] * x[i];
}
```

Innermost loop strides with j

```
for (int j=0; j<SIZE; j++)  {
     for (int i=0; i<SIZE; i++) {
          b[i] += a[i][j] * x[j];
     }
}
```

Indirect addressing

```
for (int i=0; i<SIZE; i+=2) {
     b[i] += a[i] * x[index[i]];
}
```

More Obstacles

- 2. Data dependencies:
  - If data in the iterations have dependencies, they cannot be safely vectorized

```
for (j=1; j<MAX; j++) {
	A[j]=A[j-1]+1;
}
```

```
void test(double * a, double * b) {
	for (int i = 0; i < SIZE; i++) {
		a[i] += b[i];
	}
}
```

=> Possibility to give the compiler “hints” how to vectorize (using #pragma ivdep)

USEFUL #PRAGMAS

What are Pragmas?

- Pragmas give the compiler additional information what to do
  - E.g. #pragma once

- icc: lots of pragmas available
  - #pragma vector always:
    - Ask the compiler to vectorize even if it is not efficient
  - #pragma novector:
    - Don’t vectorize
  - #pragma vector aligned
    - Asserts that data in the loop is assigned (to 16bye boundaries, for SSE)
  - #pragma simd
    - Enforce vectorization
- gcc(4.8): only few

Some Pragmas

- #pragma ivdep:
  - Ignore potential data dependencies
  - Assert good data alignment

```
#pragma ivdep
#pragma vector aligned
void test(double * a, double * b) {
	for (int i = 0; i < SIZE; i++) {
		a[i] += b[i];
	}
}
```

Vectorization Pragmas with gcc?

- http://locklessinc.com/articles/vectorize/
- Needs some explicit statements:

```
#pragma ivdep
#pragma vector aligned
void test(double * a, double * b) {
	for (int i = 0; i < SIZE; i++) {
		a[i] += b[i];
	}
}
```

icc

```
void test(double * restrict a, double * restrict b) {
	double *x = __builtin_assume_aligned(a, 16);
	double *y = __builtin_assume_aligned(b, 16);
	for (int i = 0; i < SIZE; i++) {
		x[i] += y[i];
	}
}
```

gcc

GUIDED AUTO-VECTORIZATION

Vectorization Reports

- After vectorizing read the compilation report
  - Icc: -vec-report=<n>
  - Gcc: -ftree-vectorizer-verbose=<n>
- Report tells you what has been vectorized
- + what has NOT been vectorized and WHY

n=0: No diagnostic information
n=1: Loops successfully vectorized
n=2: Loops not vectorized - and the reason why not
n=3: Adds dependency Information
n=4: Reports only non-vectorized loops
n=5: Reports only non-vectorized loops and adds dependency info

icc

Level 0: No output at all.
Level 1: Report vectorized loops.
Level 2: Also report unvectorized "well-formed" loops and respective
reason.
Level 3: Also report alignment information (for "well-formed" loops).
Level 4: Like level 3 + report for non-well-formed inner-loops.
Level 5: Like level 3 + report for all loops.
Level 6: Print all vectorizer dump information (equivalent to former
vect-debug-details).

gcc

Exemplary Report with gcc

- Trying to Auto-vectorize a loop that is not vectorizable

g++ -march=native -O3 -ftree-vectorizer-verbose=2 testPRoject.cpp -o hallo

```
float a[N];
    for ( int i = 0; i < N; i++ ) {
            a[i] += i;
    }
    for ( int i = 1; i < N; i++ ) {
        a[i] += a[i-1];
    }
```

- GCC4.8 on supermuc returns:

Analyzing loop at mandel.cpp:12

mandel.cpp:12: note: not vectorized, possible dependence between data-refs
mandel.cpp:12: note: bad data dependence.
mandel.cpp:12: note: not vectorized, possible dependence between data-refs
mandel.cpp:12: note: bad data dependence.

Guided Vectorization

- Activate with –guide-vec=<n> to obtain more information how to vectorize (n = 1 or 2)

```
float a[N];
float c=1;
for ( int i = 0; i < N; i++) {
        if ( a[i] > 0 ) {c=a[i]; a[i] = 1 /a[i] ; }
        if ( a[i] > 1 ) {a[i] +=c;}
}
```

```
icc -O2 -guide-vec=2 testPRoject.cpp -o hallo

testPRoject.cpp(20): remark #30515: (VECT) Assign a value to the variable(s) "c" at the beginning of the body of the loop in line 20. This will allow the loop to be vectorized. [VERIFY] Make sure that, in the original program, the variable(s) "c" read in any iteration of the loop has been defined earlier in the same iteration.
```

General hints for Vectorizing

- Use array notation ([i]) rather than pointers (*a + i)
- Minimize indirect addressing (a[b[i]])
- Try to use loops with unit stride
- Align variables to 16 / 32 byte boundaries

SUMMARY

Summary

- AVX powerful extension to process lots of data
- Automated vectorization is a quick and effective way to get higher performance
- For optimal performance, a deeper understanding of the vectorization basics is necessary
- The GAP and reporting tools of icc help to locate potential

Outlook

- SIMD is becoming more important in the future
  - Performance gain
  - Energy efficiency
- AVX2 has 512bit for vector operations
- Included in the current Haswell architecture