Vectorization (SIMD)

Alessandro Casalino
a.casalino@cineca.it

July 6, 2023

CINECA

Outline

- Introduction to vectorization
- Auto-vectorization
- Vectorization in `OpenMP`

Introduction

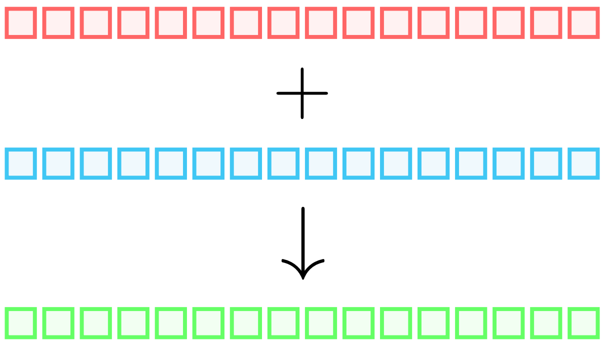
Why Vectorization?

```
void add(double* a, double* b,
         double* c)
{
    for(size_t i = 0; i < 16; i++) {
        a[i] = b[i] + c[i];
    }
}
```

- Each iteration is done sequentially one by one
- Even if parallelized, each thread do one assigned iteration one by one

Can we do better?

Scalar instruction

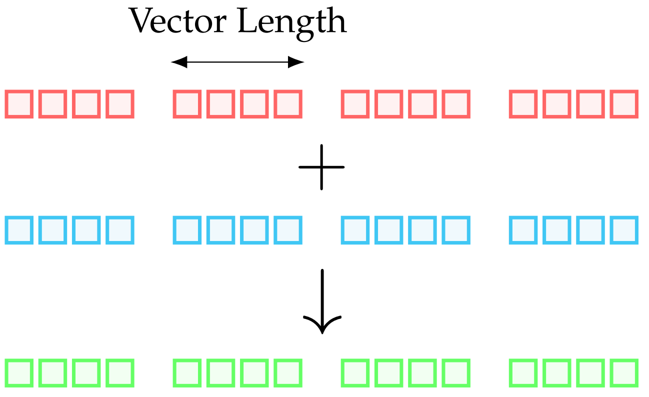


Each iteration is done sequentially one by one.

Operations

- 16 + 16 memory loads = 32 memory loads
- 16 additions
- 16 store

Vectorized instruction

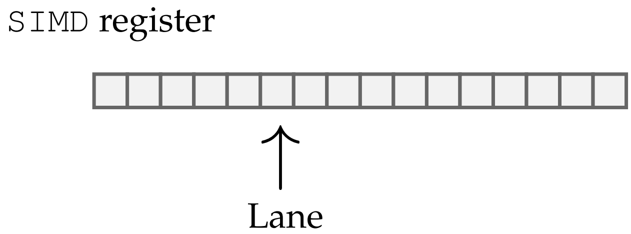


Operations are performed on a block of data (**vector**).

Operations

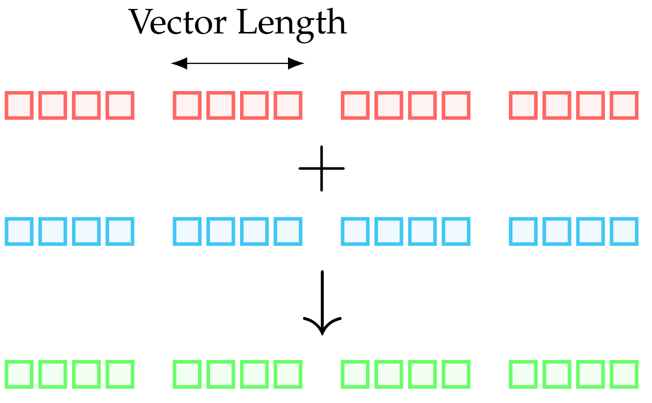
- $4 + 4$ memory loads = 8 memory loads
- 4 additions
- 4 store

SIMD (Single Instruction Multiple Data)



- A **SIMD Register** holds many values of the same datatype
- Each value is a **SIMD lane**
- SIMD hardware instructions modify the vector registers
- SIMD instructions can operate on all (or part) of the values in a SIMD register
- Width in recent CPUs up to 512 bit

Why vectorization?



- `OpenMP` vectorization = `SIMD`
- Operates at once on an entire block, i.e. **operations on multiple data concurrently**
- **Dedicated registers**
- **Maximize bandwidth**

Auto-vectorization

Auto-vectorization

Modern compilers **auto-vectorize** loops, provided that:

- complexity of the code is low enough for the compiler to select proper instructions;
- code pattern is recognized by the compiler;
- the loop benefits from a vectorization.

Compiler Reports might help

- `icc -qopt-report=N -qopt-report-phase=loop,vec`
- `gcc -ftree-vectorize -ftree-vectorize-verbose [-O3]`

Note

`icc` enables vectorization by default.
`gcc` enables vectorization with `-O3`.

Vectorize the code

```
31 for (size_t i=0; i<n; ++i)
32     for (size_t k=0; k<n; k++)
33         for (size_t j=0; j<n; ++j)
34             c[i][j] += a[i][k]*b[k][j];
```

Note

Note that only the last loop is vectorized (line `33`).

`icc` Compiler Report

```
LOOP BEGIN at mat_prod.c(33,3)
 remark #15300:  LOOP WAS VECTORIZED
 remark #15442:  entire loop may be executed in remainder
 remark #15448:  unmasked aligned unit stride loads:  2
 remark #15449:  unmasked aligned unit stride stores:  1
 remark #15475:  -- begin vector cost summary --
 remark #15476:  scalar cost:  9
 remark #15477:  vector cost:  4.000
 remark #15478:  estimated potential speedup:  2.210
 remark #15488:  -- end vector cost summary --
 remark #25015:  Estimate of max trip count of loop=125
LOOP END
```

Auto-vectorization Failing Causes

- **Data dependencies**
- **Alignment**
- **Function calls in the loop block**
- **Complex control flow/conditional branches**
- Loop not countable (e.g., upper bound not runtime constant)
- Mixed data types
- Non-unit stride between elements
- **Loop body too complex**
- Vectorization seems inefficient
- …

Data Dependency: Flow Dependency

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

RAW (Read After Write)
All the values are available within the loop iteration: **vectorization possible**.

```
i = 0

a[0] = b[0] + 1;
c[0] = a[0] + 1;
```

```
i = 1

a[1] = b[1] + 1;
c[1] = a[1] + 1;
```

Data Dependency: Anti-Dependency

```
for(size_t i = 1; i < n; i++) {
    a[i] = b[i] + 1;
    c[i] = a[i-1] + 1;
}
```

WAR (Write After Read)
A value should be loaded to a previous iteration: **vectorization not possible**.

```
i = 1

a[1] = b[1] + 1;
c[1] = a[0] + 1;
```

```
i = 2

a[2] = b[2] + 1;
c[2] = a[1] + 1;
```

Data Dependency

Flow Dependency

```
for(size_t i = 0; i < n-1; i++) {
    a[i] += a[i+1];
}
```

icc Compiler Report

```
remark #15300: LOOP WAS
VECTORIZED
```

godbolt.org/z/q5G1WcYbn

Anti-Dependency

```
for(size_t i = 1; i < n; i++) {
    a[i] += a[i-1];
}
```

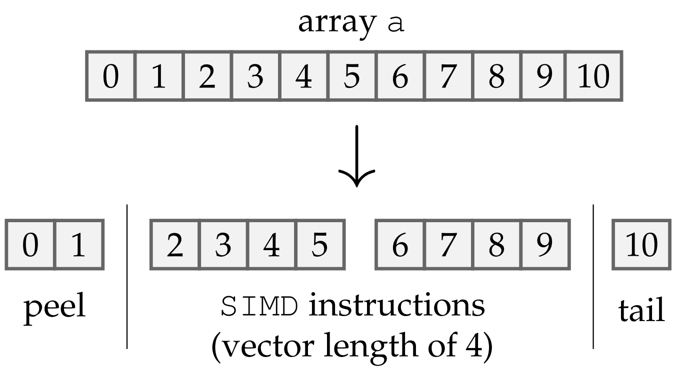
icc Compiler Report

```
remark #15344: loop was not
vectorized: vector dependence
prevents vectorization.
```

godbolt.org/z/azn5nEohM

Data Alignment

```c
for(size_t i = 0; i < 11; i++) {
    a[i] = 1.;
}
```



- For best effect: **starting address of vectors should be aligned on the correct boundary**
- Address in memory should be multiple of vector length in bytes

Otherwise **unaligned data**

- May require **multiple data loading, cache lines and instructions**
- Generates 3 different version of loops: **peel**, **kernel**, **tail**

Data Alignment

```
for (size_t i=0; i<n; ++i)
    for (size_t k=0; k<n; k++)
        for (size_t j=0; j<n; ++j)
            c[i][j] += a[i][k]*b[k][j];
```

`icc` Compiler Report

```
LOOP BEGIN at mat_prod.c(33,3)
<Remainder loop for vectorization>
 remark #15335:  remainder loop was not vectorized:
vectorization possible but seems inefficient.  Use vector always
directive or -vec-threshold0 to override
 remark #15442:  entire loop may be executed in remainder
 remark #15448:  unmasked aligned unit stride loads:  2
 remark #15449:  unmasked aligned unit stride stores:  1
 remark #15475:  -- begin vector cost summary --
 remark #15476:  scalar cost:  9
 remark #15477:  vector cost:  4.000
 remark #15478:  estimated potential speedup:  2.210
 remark #15488:  -- end vector cost summary --
LOOP END
```

Vectorization in OpenMP

SIMD CONSTRUCT

Serial

```
#pragma omp simd
for(size_t i = 0; i < n; i++) {
    a[i] = b[i] + c[i];
}
```

- `simd` tells the compiler to vectorize
- No worksharing, i.e., only enables `SIMD`
- Compiler chooses the appropriate vector length for the target architecture

Worksharing

```
#pragma omp for simd
for(size_t i = 0; i < n; i++) {
    a[i] = b[i] + c[i];
}
```

- `for` tells the compiler to start the parallelization on multiple threads
- `simd` enables the vectorization for each thread
- First the work is shared between threads, then vectorization on chunks fitting the `SIMD` register

DATA SHARING

```
void foo (double * a, double * b,
          double * c, int n) {
    int i;
    double t1, t2;

    #pragma omp simd private(t1,t2)
    for(i=0; i<n; i++) {
        t1 = func1(b[i], c[i]);
        t2 = func2(b[i], c[i]);
        a[i] = t1 + t2;
    }
}
```

- Loop iterator `i` is private
- Memory pointed by `a`, `b` and `c` is shared

`SIMD` supported clauses

- `private(list)`
- `lastprivate(list)`
- `reduction(operation:list)`

SAFELEN CLAUSE

```
#pragma omp simd safelen(4)
for(i=0; i<n-4; i++) {
    a[i] + = a[i+4];
}
```

`safelen` specifies the **distance between iterations where it is safe to vectorize**.

- Useful to specify the maximum number of iterations that run concurrently without breaking a dependence.
- In practice, maximum `SIMD` vector length.

Best practice

Avoid using this clause. Specifying explicit vector lengths builds in obsolescence to the code as hardware vector lengths continually change.

collapse CLAUSE

`simd` partitions the loop iterations into chunks. But the partitioning requires only one iteration space to be valid. We can use `collapse` to solve this issue.

```
float A[4][4]
#pragma omp simd collapse(2)
for(size_t i=0; i < 4; i++) {
    for(size_t j=0; j<4; i++) {
        A[i][j] += 1;
    }
}
```

```
float A[4][4]
#pragma omp simd
for(size_t k=0; k < 16; k++) {
    size_t i = k / 4;
    size_t j = k % 4;
    A[i][j] += 1;
}
```

Best practice

This is the correct way to collapse `for` loops for `simd`.

Warning

Access to `A` is not linear with respect to the collapsed loop.

Other SIMD Clauses

`simdlen(length)`

- Specify the preferrend length of `SIMD` registers
- Must be less or equal than `safelen`

Best practice

Again, since the hardware architectures changes, avoid using these *magic numbers* in the code.

`linear(length)`

- Specify the variable's value in relationshop with the iteration number

Take-Home Messages

Take-Home Messages

- **`OpenMP SIMD`** constructs instructs the compiler to vectorize exploiting modern CPUs vector accelerator
- The **performance improvements with vectorization can be substantial**
- **Modern compilers are capable of auto-vectorize loops**, provided some conditions are met
- **Check the compiler reports to understand if the loops are correcty vectorized**

Exercises

Exercises

Matrix Multiplication

- Add `simd` constructs
- Run and time it

Value of π

- Add `simd` constructs
- Run and time it

Thank you!