


VECTORIZATION


Conceptualizing Compiler Vectorization

- Vectorization transforms a program so that the same operation is performed at the same time on several vector elements.
- Think of vectorization in terms of loop unrolling – Unroll N interactions of loop, where N elements of data array fit into vector register.

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



```
for (i=0; i<N;i+=4) {  
    a[i+0]=b[i+0]+c[i+0];  
    a[i+1]=b[i+1]+c[i+1];  
    a[i+2]=b[i+2]+c[i+2];  
    a[i+3]=b[i+3]+c[i+3];  
}
```



```
Load b(i..i+3)  
Load c(i..i+3)  
Operate b+c->a  
Store a
```

SIMD Extensions

x86-64 Vector Operations - Overview

Example Instructions

: Move: (V)MOV[A/U]P[D/S]

: Comparing: (V)CMP[P/S][D/S]

: Arithmetic Operations: (V)[ADD/SUB/MUL/DIV][P/S][D/S]

Instruction Decoding

: V - AVX

: P,S - packed, scalar

: A,U - aligned, unaligned

: D,S - double, single

: B, W, D, Q - byte, word, doubleword, quadword integers

: [] - required, () - optional

Example: vmovapd ymm0, YMMWORD PTR [rdi+rax]

x86-64 Vector Registers

- AVX-512 (ZMM0-ZMM31) 512 bit
- AVX (YMM0-YMM15) 256 bit
- SSE (XMM0-XMM15) 128 bit

Requirements and Limitations:

1. Countable loops
2. No backward loop-carried dependencies
3. No function calls
 - : Except vectorizable math functions e.g. sin, sqrt,...
4. Straight-line code (only one control flow: no switch)
5. Loop to be vectorized must be innermost loop if nested

Data dependences

- The notion of dependence is the foundation of the process of vectorization.
- A statement S is said to be data dependent on statement T if
 - T executes before S in the original sequential/scalar program
 - S and T access the same data item
 - At least one of the accesses is a write.
- Dependences indicate an execution order that must be honored.
- Executing statements in the order of the dependences guarantee correct results.
- Statements not dependent on each other can be reordered, executed in parallel, or coalesced into a vector operation.

Data Dependence

- Flow dependence (True dependence)

S1: $X = A + B$

S2: $C = X + A$

- Anti dependence

S1: $A = X + B$

S2: $X = C + D$

- Output dependence

S1: $X = A + B$

S2: $X = C + D$

- Read After Write
 - Also called “flow” dependency
 - Variable written first, then read
 - Not vectorizable

```
for( i=1; i<N; i++)  
    a[i] = a[i-1] + b[i];
```

- Write after Read
 - Also called “anti” dependency
 - Variable read first, then written
 - vectorizable

```
for( i=0; i<N-1; i++)  
    a[i] = a[i+1] + b[i];
```


Data dependences and vectorization

- Loop dependences guide vectorization.
- Main idea: A statement inside a loop which is not in a cycle of the dependence graph can be vectorized.

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



```
a[0:n-1] = b[0:n-1] + 1;
```

```
for (i=1; i<n; i++){  
    a[i] = b[i] + 1;  
    c[i] = a[i-1] + 2;  
}
```



```
a[1:n] = b[1:n] + 1;  
c[1:n] = a[0:n-1] + 2;
```

Dependences in Loops

- Dependence exists within an iteration; i.e. Different iteration of the loop can be parallelized.

```
for (i=0; i<n; i++){  
    a[i] = b[i] + 1;  
    c[i] = a[i] + 2;  
}
```

iteration:	0	1	2	3
instances of S1:	S1	S1	S1	S1
instances of S2:	S2	S2	S2	S2

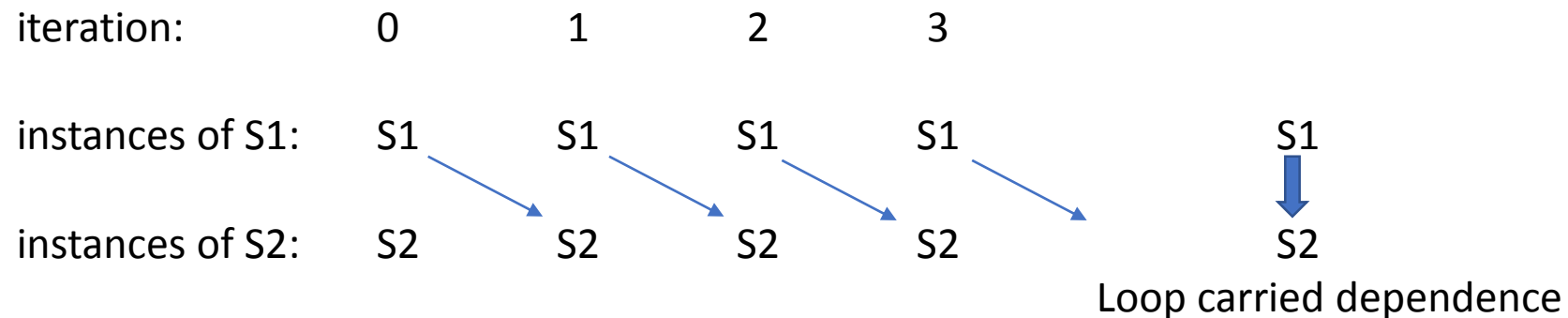
S1
↓
S2

Loop independent dependence

Dependences in Loops

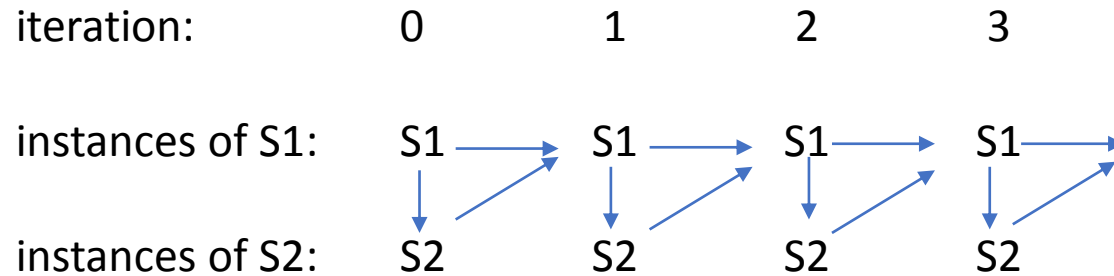
- Dependence exists across iterations; i.e., Different iteration of the loop must be executed sequentially.

```
for (i=1; i<n; i++){  
    a[i] = b[i] + 1;  
    c[i] = a[i-1] + 2;  
}
```



Dependences in Loops

```
for (i=0; i<n; i++){  
    a = b[i] + 1;  
    c[i] = a + 2;  
}
```



Loop independent dependence
Loop carried dependence

Acyclic Dependence Graphs:

Forward Dependences

```
for (i=0; i<LEN; i++) {  
  a[i]= b[i] + c[i]  
  d[i] = a[i] + (float) 1.0;  
}
```

Backward Dependences

```
for (i=0; i<LEN; i++) {  
  a[i]= b[i] + c[i]  
  d[i] = a[i+1] + (float) 1.0;  
}
```

Cycles in the DG

```
for (int i=0;i<LEN-1;i++){  
  b[i] = a[i] + (float) 1.0;  
  a[i+1] = b[i] + (float) 2.0;  
}
```

```
for (int i=1;i<LEN;i++){  
  a[i] = b[i] + c[i];  
  d[i] = a[i] + e[i-1];  
  e[i] = d[i] + c[i];  
}
```

Data Alignment

- Vector loads/stores load/store 128 consecutive bits to a vector register.
- Data addresses need to be 16-byte (128 bits) aligned to be loaded/stored.
- Some architectures need data to be aligned.
- Intel: unaligned data access possible. But: Computation Overhead
 - Multiple reads necessary
 - Additional code to extract the data

```
float A[N] __attribute__((aligned(16)));
float B[N] __attribute__((aligned(16)));
float C[N] __attribute__((aligned(16)));
void test(){
    for (int i = 0; i < N; i++){
        C[i] = A[i] + B[i];
    }
}
```

Data Alignment

- Manual 16-byte alignment can be achieved by forcing the base address to be a multiple of 16.

```
__attribute__((aligned(16))) float b[N];  
float* a = (float*) memalign(16,N*sizeof(float));
```

- When the pointer is passed to a function, the compiler should be aware of where the 16-byte aligned address of the array starts.

```
void func1(float *a, float *b, float *c) {  
    __assume_aligned(a, 16);  
    __assume_aligned(b, 16);  
    __assume_aligned(c, 16);  
    for int (i=0; i<LEN; i++) {  
        a[i] = b[i] + c[i];  
    }  
}
```


Loop Peeling

- Remove the first/s or the last/s iteration of the loop into separate code outside the loop.
- It is always legal, provided that no additional iterations are introduced.
- This transformation is useful to enforce a particular initial memory alignment on array references prior to loop vectorization.

```
for (i=0; i<LEN; i++)  
  A[i] = A[i] + A[0];
```



```
A[0] = A[0] + A[0];  
for (i=1; i<LEN; i++)  
  A[i] = A[i] + A[0];
```

After loop peeling, there are no dependences, and the loop can be vectorized

Aliasing

- Refers to memory addressed by different names.
- To vectorize, the compiler needs to guarantee that the pointers are not aliased.
- When the compiler does not know if two pointer are alias, it still vectorizes, but needs to add up-to $O(n^2)$ runtime checks, where n is the number of pointers

```
void func1(float *a, float *b, float *c)
{
    for (int i = 0; i < LEN; i++)
        a[i] = b[i] + c[i];
}
```

Aliasing

- Two solutions can be used to avoid the run-time checks
 1. static and global arrays
 2. `__restrict__` attribute: Instructs compiler to assume addresses will not overlap, ever

1. Static and Global arrays

```
__attribute__((aligned(16))) float a[LEN];
__attribute__((aligned(16))) float b[LEN];
__attribute__((aligned(16))) float c[LEN];
void func1(){
    for (int i=0; i<LEN; i++)
        a[i] = b[i] + c[i];
}
int main() {
    ...
    func1();
}
```

2. `__restrict__` keyword

```
void test(float* __restrict__ A,
float* __restrict__ B,
float* __restrict__ C,
float* __restrict__ D)
{
    for (int i = 0; i < LEN; i++){
        A[i]=B[i]+C[i]+D[i];
    }
}
```

How much programmer intervention?

- Next, three examples to illustrate what the programmer may need to do:
 - Add compiler directives
 - Transform the code
 - Program using vector intrinsics

Compiler directives

- When the compiler does not vectorize automatically due to dependences the programmer can inform the compiler that it is safe to vectorize.
- Programmer can disable vectorization of a loop when the when the vector code runs slower than the scalar code.
- **#pragma GCC ivdep**: Ignore assume data dependences
- **#pragma vector always**: override efficiency heuristics
- **#pragma novector**: disable vectorization
- **__restrict__** : assert exclusive access through pointer
- **__attribute__((aligned(int-val)))**: request memory alignment
- **memalign(int-val,size);**: malloc aligned memory
- **__assume_aligned(exp, int-val)**: assert alignment property

Loop Transformations

- Loop Distribution or loop fission
- Reordering Statements
- Loop Peeling
- Loop Unrolling
- Loop Interchanging

Loop Distribution

- It is also called loop fission.
- Divides loop control over different statements in the loop body.

```
for (i=1; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
    dummy(a,b,c);  
}
```



```
for (i=1; i<LEN; i++)  
    a[i] = b[i] + c[i];  
for (i=1; i<LEN; i++)  
    dummy(a,b,c);
```

Loop Interchanging

- This transformation switches the positions of one loop that is tightly nested within another loop.

```
for (j=1; j<LEN; j++){  
    for (i=j; i<LEN; i++){  
        A[i][j]=A[i-1][j]+(float) 1.0;  
    }  
}
```

```
for (i=1; i<LEN; i++){  
    for (j=1; j<i+1; j++){  
        A[i][j]=A[i-1][j]+(float) 1.0;  
    }  
}
```


Non-unit Stride

- An array with stride of exactly the same size as the size of each of its elements is contiguous in memory prefer as unit stride.

```
for (int i=0;i<LEN;i++){  
    sum = (float) 0.0;  
    for (int j=0;j<LEN;j++){  
        sum += A[j][i];  
    }  
    B[i] = sum;  
}
```

```
for (int i=0;i<size;i++){  
    sum[i] = 0;  
    for (int j=0;j<size;j++){  
        B[i] += A[j][i];  
    }  
}}
```

Access the SIMD through intrinsics

- Intrinsics are vendor/architecture specific.
- Intrinsics are useful when
 - the compiler fails to vectorize
 - when the programmer thinks it is possible to generate better code than the one produced by the compiler
- SSE can be accessed using intrinsics.

You must use one of the following header files:

`#include <xmmintrin.h>` (for SSE)

`#include <emmintrin.h>` (for SSE2)

`#include <pmmintrin.h>` (for SSE3)

`#include <smmintrin.h>` (for SSE4)

Intrinsics (SSE)

```
#define n 1024
__attribute__((aligned(16))) float a[n],
b[n], c[n];
int main() {
for (i = 0; i < n; i++) {
c[i]=a[i]*b[i];
}
}
```

```
#include <xmmintrin.h>
#define n 1024
__attribute__((aligned(16))) float a[n], b[n], c[n];
int main() {
__m128 rA, rB, rC;
for (i = 0; i < n; i+=4) {
rA = _mm_load_ps(&a[i]);
rB = _mm_load_ps(&b[i]);
rC= _mm_mul_ps(rA,rB);
_mm_store_ps(&c[i], rC);
}
}
```

Summary

- Microprocessor vector extensions can contribute to improve program performance and the amount of this contribution is likely to increase in the future as vector lengths grow.
- Compilers are only partially successful at vectorizing.
- When the compiler fails, programmers can
 - add compiler directives
 - apply loop transformations.
- If after transforming the code, the compiler still fails to vectorize (or the performance of the generated code is poor), the only option is to program the vector extensions directly using intrinsics or assembly language.