CS 610: Vectorization

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2024-25-I

Different Levels of Parallelization in Hardware

Instruction-level Parallelism

Microarchitectural techniques like pipelining, OOO execution, and superscalar instruction issue

Data-level Parallelism

Use Single Instruction Multiple Data (SIMD) vector processing instructions and units

Thread-level Parallelism

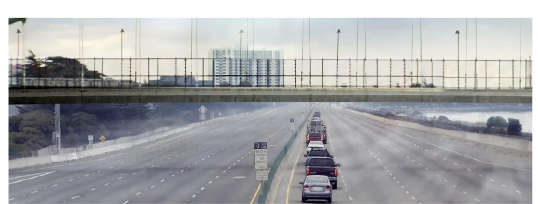
Hyperthreading

Vectorization

- Vectorization is the process of transforming a scalar operation on single data elements at a time (SISD) to an operation on multiple data elements at once (SIMD)
- Loop vectorization transforms a program so that the same operation is performed at the same time on several vector elements

Don't use a single Vector lane/thread!

Un-vectorized and un-threaded software will under perform



Permission to Design for All Lanes

Threading and Vectorization needed to fully utilize modern hardware

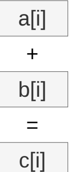
K. Rogozhin. Vectorization.

Vectorization

```c
double *a, *b, *c;
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

Scalar mode

One instruction (e.g., vaddsd/vaddss) produces one result



Vector mode

One instruction (e.g., vaddpd/vaddps) can produce multiple results

a[i+7]	a[i+6]	a[i+5]	a[i+4]	a[i+3]	a[i+2]	a[i+1]	a[i]

+

b[i+7]	b[i+6]	b[i+5]	b[i+4]	b[i+3]	b[i+2]	b[i+1]	b[i]

=

c[i+7]	c[i+6]	c[i+5]	c[i+4]	c[i+3]	c[i+2]	c[i+1]	c[i]

Vectorization

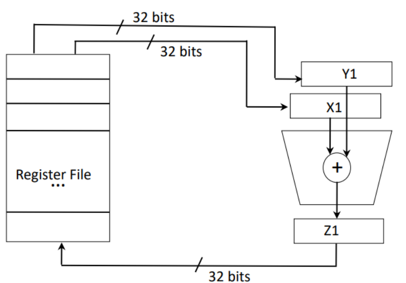
n times

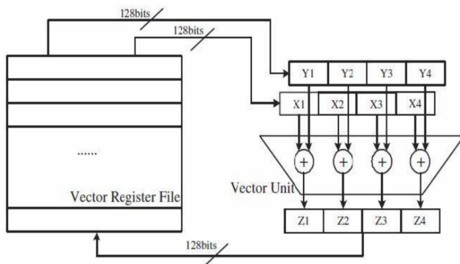
```
ld r1, addr1
ld r2, addr2
add r3, r1, r2
st r3, addr3
```

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

$\frac{n}{4}$ times

```
ldv vr1, addr1
ldv vr2, addr2
addv vr3, vr1, vr2
stv vr3, addr3
```

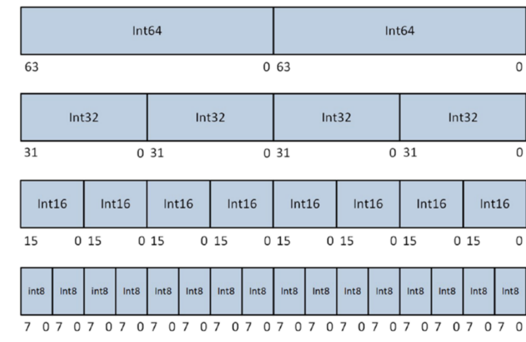




SIMD Vectorization

- Use of SIMD units can speed up the program
- Intel SSE has 128-bit vector registers and functional units
- Assuming a single ALU, these SIMD units can execute 4 single precision floating point number or 2 double precision operations in the time it takes to do only one of these operations by a scalar unit

128-bit wide operands using integer types



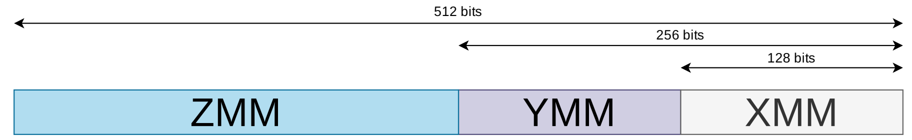
Daniel Kusswurm. Modern X86 Assembly Language Programming. 2nd edition, Apress.

Intel-Supported SIMD Extensions

SIMD Extensions	Width (bits)	SP calculations	DP calculations	Introduced
SSE2/SSE3/SSE4	128	4	2	~2001–2007
AVX/AVX2	256	8	4	~2011–2015
AVX-512	512	16	8	~2017

Other platforms that support SIMD have different extensions (e.g., ARM Neon and Power AltiVec)

Intel-Supported SIMD Extensions



64-bit architecture

SSE	XMM0–XMM15	
AVX	YMM0–YMM15	Low-order 128 bits of each YMM register is aliased to a corresponding XMM register
AVX-512	ZMM0–ZMM31	Low-order 256 and 128 bits are aliased to registers YMM0–YMM31 and XMM0–XMM31 respectively

x86_64 Vector Operations

[]	required
()	optional

Example instructions

Move	(V)MOV[A/U]P[D/S]
Comparison	(V)CMP[P/S][D/S]
Arithmetic	(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-precision
B,W,D,Q	byte, word, doubleword, quadword integers

x86_64 Vector Operations

movss xmm1, xmm2 — Move scalar single-precision floating-point value from xmm2 to xmm1

vmovapd xmm1, xmm2 — Move aligned packed double-precision floating- point values from xmm2 to xmm1

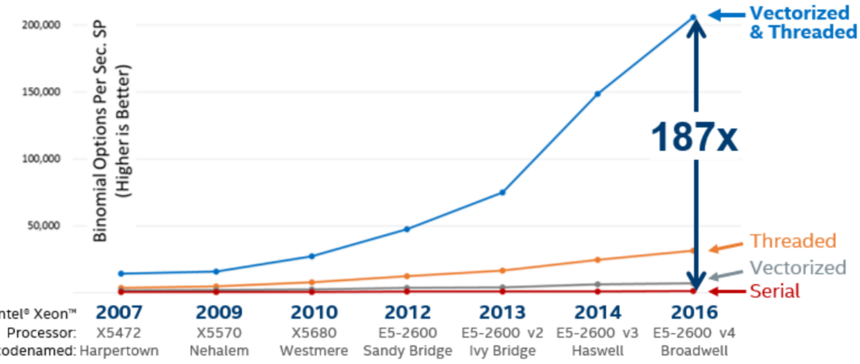
```
vaddss xmm0,xmm1,xmm2
```

```
xmm0[31:0] = xmm1[31:0] + xmm2[31:0]
xmm0[127:32] = xmm1[127:32]
ymm0[255:128] = 0
```

```
vaddsd xmm0,xmm1,xmm2
```

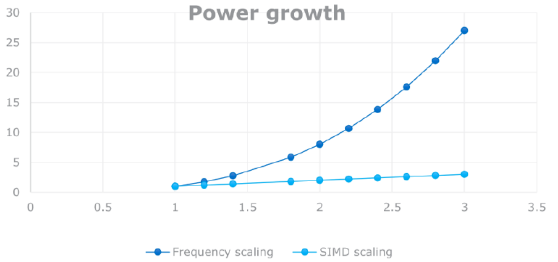
```
xmm0[63:0] = xmm1[63:0] + xmm2[63:0]
xmm0[127:64] = xmm1[127:64]
ymm0[255:128] = 0
```

The combined effect of vectorization and threading



The Difference Is Growing With Each New Generation of Hardware

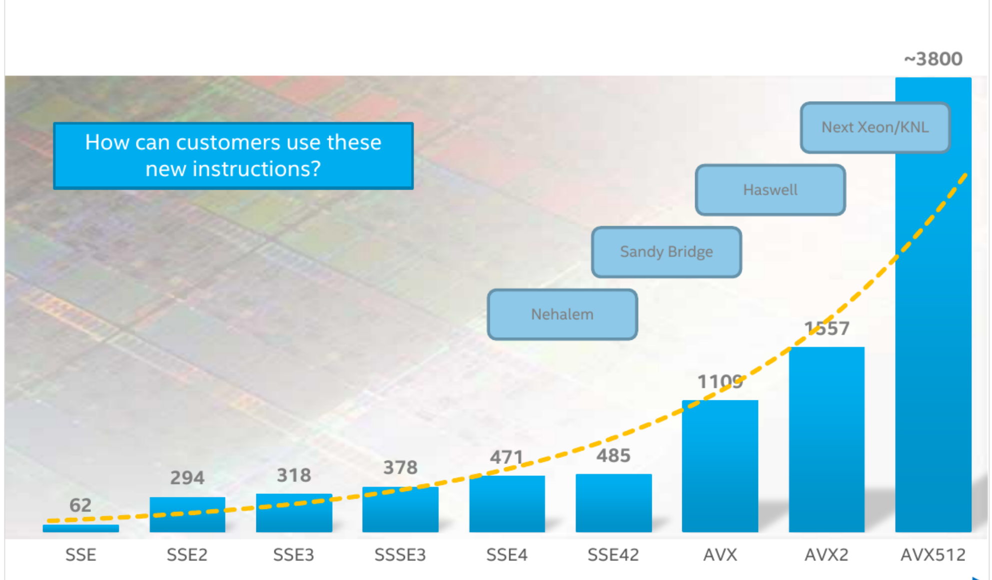
Why SIMD vector parallelism?



Wider SIMD -- Linear increase in area and power
Wider superscalar – Quadratic increase in area and power
Higher frequency – Cubic increase in power

With SIMD we can go faster with less power

Cumulative (app.) # of Vector Instructions



Ways to Vectorize Code

Ways to Vectorize Code

- Auto-vectorizing compiler

```
for (i=0; i<LEN; i++)
  c[i] = a[i] + b[i];
```

- Vector intrinsics

```
void example() {
  __m128 rA, rB, rC;
  for (int i = 0; i <LEN; i+=4) {
    rA = _mm_load_ps(&a[i]);
    rB = _mm_load_ps(&b[i]);
    rC = _mm_add_ps(rA,rB);
    _mm_store_ps(&C[i], rC);
  }
}
```

- Assembly programming

```
..B8.5
  movaps a(,%rdx,4), %xmm0
  addps b(,%rdx,4), %xmm0
  movaps %xmm0, c(,%rdx,4)
  addq $4, %rdx
  cmpq $rdi, %rdx
  jl ..B8.5
```

- Use SIMD-capable libraries like Intel Math Kernel Library (MKL)

easier, but less control

harder, but more control

Auto-Vectorization

Compiler vectorizes automatically	No code changes
Semi auto-vectorization	Use pragmas as hints to guide compiler
Explicit vector programming	OpenMP SIMD pragmas

Advantages

+ Transparent to programmers
+ Compilers can apply other transformations
+ Code is portabile across architectures
    - Vectorization instructions may differ, but compilers take care of it

Compilers may fail to vectorize

- Programmers may give hints to help the compiler
- Programmers may have to manually vectorize their code

Data Dependences and Vectorization

- Loop dependences guide vectorization
  - Statements not data dependent on each other can be reordered, executed in parallel, or coalesced into a vector operation
- A statement inside a loop which is not in a cycle of the dependence graph can be vectorized

```
    for (i=1; i<=n; i++) {
S1    a[i] = b[i] + 1;
S2    c[i] = a[i-1] + 2;
    }
```

$\Rightarrow$

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

- When cycles are present, vectorization can be achieved by
  - Separating (distributing) the statements not in a cycle
  - Removing dependences
  - Freezing loops
  - Changing the algorithm

Vectorization in Presence of Cycles

Loop Distribution

```
    for (i=1; i<n; i++) {
S1    b[i] = b[i] + c[i];
S2    a[i] = a[i-1]*a[i-2]+b[i];
S3    c[i] = a[i] + 1;
    }
```

```
S1  b[1:n-1] = b[1:n-1] + c[1:n-1];
    for (i=1; i<n; i++){
S2    a[i] = a[i-1]*a[i-2]+b[i];
    }
S3  c[1:n-1] = a[1:n-1] + 1;
```

Scalar Expansion

```
    for (i=0; i<n; i++) {
S1    a = b[i] + 1;
S2    c[i] = a + 2;
    }
```

```
    for (i=0; i<n; i++) {
S1    $a[i] = b[i] + 1;
S2    c[i] = $a[i] + 2;
    }
    a = $a[n-1]
```

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

Vectorization in Presence of Cycles

Freezing Loops

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

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

Changing the Algorithm

- When there is a recurrence, it is necessary to change the algorithm in order to vectorize
- Compilers use pattern matching to identify the recurrence and then replace it with a parallel version
- Examples of recurrences include

Reductions	`sum += A[i]`
Linear recurrences	`A[i] = B[i]*A[i-1]+C[i]`
Boolean recurrences	`if (A[i]>max) { max = A[i] }`

Loop Vectorization

- Compiler computes the dependences
  - (i) The compiler figures out dependences by
    - Solving a system of (integer) equations (with constraints)
    - Demonstrating that there is no solution to the system of equations
  - (ii) Removes cycles in the dependence graph
  - (iii) Determines data alignment
  - (iv) Determines if vectorization is profitable
    - Loop vectorization is not always a legal and profitable transformation
- Vectorizing a loop with several statements is equivalent to strip-mining the loop and then applying loop distribution

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

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

Dependence Graphs and Compiler Vectorization

- No dependences: easy case, just check for profitability
- Acyclic graphs:
  - All dependences are forward: vectorized by the compiler
  - Some backward dependences: sometimes vectorized by the compiler
- Cycles in the dependence graph
  - Self anti-dependence: vectorized by the compiler
  - Recurrence: usually not vectorized by the compiler

Acyclic Dependences

Forward dependences are vectorized

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

Backward dependences can sometimes be vectorized

```
    for (i=0; i<LEN; i++) {
S1    a[i] = b[i] + c[i]
S2    d[i] = a[i+1] + 1;
    }
```

```
    for (i=0; i<LEN; i++) {
S2    d[i] = a[i+1] + 1;
S1    a[i]= b[i] + c[i]
    }
```

```
    for (int i = 1; i<LEN; i++) {
S1    a[i] = d[i-1] + sqrt(c[i]);
S2    d[i] = b[i] + sqrt(e[i]);
    }
```

```
    for (int i = 1; i<LEN; i++) {
S2    d[i] = b[i] + sqrt(e[i]);
S1    a[i] = d[i-1] + sqrt(c[i]);
    }
```

Cycles in the DDG

Are there transformations which allow vectorizing the following loops?

```
    for (int i=0; i<LEN-1; i++) {
S1    b[i] = a[i] + 1;
S2    a[i+1] = b[i] + 2;
    }
```

Statements cannot be reordered

```
    for (int i=1; i<LEN; i++) {
S1    a[i] = b[i] + c[i];
S2    d[i] = a[i] + e[i-1];
S3    e[i] = d[i] + c[i];
    }
```

All the statements are not involved in a cycle

Cycles in the DDG

```
    for (int i=0; i<LEN-1; i++) {
S1    a[i]=a[i+1]+b[i];
    }
```

Self anti-dependence can be vectorized

```
    for (int i=1; i<LEN; i++) {
S1    a[i]=a[i-1]+b[i];
    }
```

Self true dependence cannot be vectorized

```
    for (int i=1; i<LEN; i++) {
S1    a[i]=a[i-4]+b[i];
    }
```

Self true dependence with larger distance vectors can be vectorized

Cycles in the DDG

```
    for (int i=0; i<LEN; i++) {
S1    a[r[i]] = a[r[i]] * 2;
    }
```

Are there `i` and i' such that `r[i] == r[i']` and $i \neq i'$?

Cycles can appear in the DDG because the compiler cannot prove that there cannot be dependences

Challenges in Vectorization

Loop Transformations using Compiler Directives

When the compiler does not vectorize automatically due to dependences the programmer can inform the compiler that it is safe to vectorize

```
#pragma ivdep    (ICC compiler)
```

```
for (int i=val; i<LEN-k; i++)
  a[i]=a[i+k]+b[i];
```

- Assume vector width is 4 elements
- This loop can be vectorized when `k < -3` and `k >= 0`
- Suppose programmers know that `k>0`

How can the programmer tell the compiler that `k >= 0`?

Compiler Directives

Compiler vectorizes many loops, but many more can be vectorized if appropriate directives are used

Intel ICC	
`#pragma ivdep`	Ignore data dependences
`#pragma vector always`	Override efficiency heuristics
`#pragma novector`	Disable vectorization

Aliasing

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

Aliasing

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

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

Aliasing

- To vectorize, the compiler needs to guarantee that the pointers are not aliased
- When the compiler does not know if two pointers are aliases, it can still vectorize but needs to add up to $\mathcal{O}(n^2)$ run-time checks, where n is the number of pointers
  - When the number of pointers is large, the compiler may decide to not vectorize
- Two possible workarounds
  (i) Static and globally defined arrays
  (ii) Use the `__restrict__` keyword

Resolving Aliases Using Static and Global Arrays

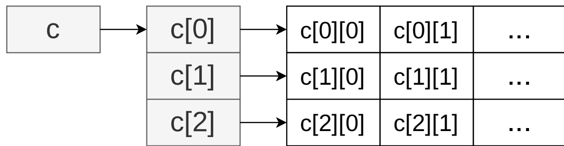
```
float A[LEN] __attribute__ ((aligned(16)));
float B[LEN] __attribute__ ((aligned(16)));
float C[LEN] __attribute__ ((aligned(16)));
void func1() {
  for (int i=0; i<LEN; i++)
    A[i] = B[i] + C[i];
}
int main() {
  ...
  func1();
}
```

Resolving Aliases Using `__restrict__` Keyword

```c
...
float *A = &B[i];
void test(float* __restrict__ A,float* __restrict__ B,
                                float* __restrict__ 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];
  }
}
int main() {
  float* A=(float*) memalign(16,LEN*sizeof(float));
  float* B=(float*) memalign(16,LEN*sizeof(float));
  float* C=(float*) memalign(16,LEN*sizeof(float));
  ...
  func1(A,B,C);
}
```

Aliasing in Multidimensional Arrays

```c
void func1(float** __restrict__ a,float** __restrict__ b,
                                  float** __restrict__ c) {
  for (int i=0; i<LEN; i++)
    for (int j=1; j<LEN; j++)
    a[i][j] = b[i][j-1] * c[i][j];
}
```



Aliasing in Multidimensional Arrays

Three solutions to try when `__restrict__` does not enable vectorization

(i) Static and global arrays
(ii) Linearize the arrays and then use `__restrict__` keyword
(iii) Use compiler directives

Static and global declaration

```
float a[N][N] __attribute__ ((aligned(16)));
void t() {
  a[i][j] ...
}
int main() {
  ...
  t();
  ...
}
```

Aliasing in Multidimensional Arrays

Linearize the array

```
void t(float* __restrict__ a){
  // Access to a[i][j] is now a[i*128+j]
  ...
}
int main() {
  float* a = (float*) memalign(16,128*128*sizeof(float));
  ...
  t(a);
}
```

Use compiler directives

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

Reductions

Reduction is an operation, such as addition, which is applied to the elements of an array to produce a result of a lesser rank

```
sum = 0;
for (int i=0; i<LEN; ++i) {
  sum += a[i];
}
```

```
x = a[0];
index = 0;
for (int i=0; i<LEN; ++i) {
  if (a[i] > x) {
    x = a[i];
    index = i;
  }
}
```

Induction Variables

Induction variables can be expressed as a function of the loop iteration variable

```
float s = 0.0;
for (int i=0; i<LEN; i++) {
  s += 2.0;
  a[i] = s * b[i];
}
```

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

Coding style may influence a compiler's ability to vectorize

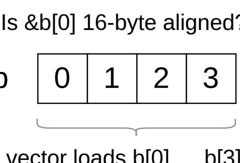
```
for (int i=0; i<LEN; i++) {
  *a = *b + *c;
  a++; b++; c++;
}
```

```
for (int i=0; i<LEN; i++) {
  a[i] = b[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
  - Intel platforms support aligned and unaligned load/stores
  - IBM platforms do not support unaligned load/stores

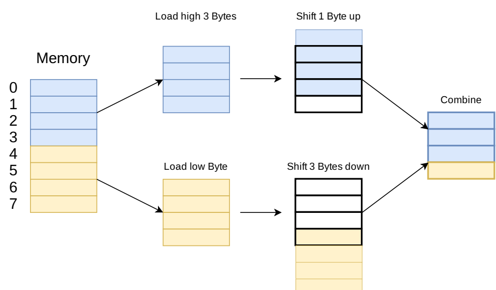
```
void test1(float *a,float *b,float *c) {
  for (int i=0;i<LEN;i++) {
    a[i] = b[i] + c[i];
  }
}
```



Why Data Alignment May Improve Efficiency?

- Vector load/store from aligned data requires one memory access
- Vector load/store from unaligned data requires multiple memory accesses and some shift operations

- A pointer is 16-byte aligned if the address is divisible by 16
  - That is, the last digit of the pointer address in hex must be 0




```
float B[1024] __attribute__ ((aligned(16)));
int main() {
  printf("%p, %p\n", &B[0], &B[4]);
}
// Output: 0x7fff1e9d8580, 0x7fff1e9d8590
```

Data Alignment

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

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

When a pointer is passed to a function, the compiler can be made aware of alignment

```
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];
}
```

Alignment in a struct

```c
struct st {
  char A;
  int B[64];
  float C;
  int D[64];
};
int main() {
  st s1;
  printf("%p\n", &s1.A); // 0x7fffe6765f00
  printf("%p\n", &s1.B); // 0x7fffe6765f04
  printf("%p\n", &s1.C); // 0x7fffe6766004
  printf("%p\n", &s1.D); // 0x7fffe6766008
}
```

```c
struct st {
  char A;
  int B[64] __attribute__ ((aligned(16)));
  float C;
  int D[64] __attribute__ ((aligned(16)));
};
int main() {
  st s1;
  printf("%p\n", &s1.A); // 0x7fffe6765f00
  printf("%p\n", &s1.B); // 0x7fff1e9d8590
  printf("%p\n", &s1.C); // 0x7fffe6766004
  printf("%p\n", &s1.D); // 0x7fff1e9d86a0
}
```

Arrays B and D are not 16-bytes aligned

Non-unit Stride

Array of structures

```
typedef struct{int x, y, z} point;
point pt[LEN];
for (int i=0; i<LEN; i++) {
  pt[i].y *= scale;
}
```

point pt[N];

x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2

vector load   vector load

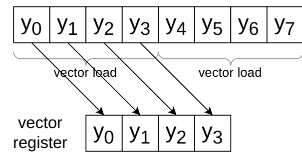
vector register

y_0	y_1	y_2	y_3

Structure of arrays

```
int ptx[LEN], pty[LEN], ptz[LEN];
for (int i=0; i<LEN; i++) {
  pty[i] *= scale;

}
```



Conditional Statements

- A compiler may not vectorize a loop with a conditional if it is unsure about the profitability
  - Furthermore, removing the condition may lead to exceptions
- You may need to introduce `#pragma vector always`
- Compiler may create multiple versions of the code (e.g., scalar and vector)
  - Compiler may remove the conditions when generating the vector version

```
#pragma vector always
for (int i = 0; i < LEN; i++) {
  if (c[i] < 0.0)
    a[i] = a[i] * b[i] + d[i];
}
```

Vectorization Examples

- Check the `Makefile` for relevant options passed to GCC
- Vectorization output can vary across compiler versions and architecture generations
- Correlate the assembly code with the high-level C++ statements

- Vectorize a loop nest with increasing control
- Understanding alignment
  - `struct.cpp`
  - `unaligned-cost-gcc.cpp`
- Makefile

Vectorization with Intrinsics

Vector Intrinsics

- Intrinsics are useful when
  - the compiler fails to vectorize, or
  - when the programmer thinks it is possible to generate better code than what is produced by the compiler
- Intrinsics are vendor/architecture specific
- We will focus on the Intel vector intrinsics

Intel Intrinsics Guide

Intel Intrinsics Header Files

You have to include one of the following header files for using intrinsics

SSE	`#include <xmmintrin.h>`
SSE2	`#include <emmintrin.h>`
SSE3	`#include <pmmintrin.h>`
SSE4	`#include <smmintrin.h>`
AVX	`#include <immintrin.h>`
AVX2	`#include <immintrin.h>`
AVX512	`#include <immintrin.h>`

Use `#include <x86intrin.h>`, it includes all relevant headers

Format of Intel Intrinsic APIs

```
_mm_instruction_suffix(...)
_mm256_instruction_suffix(...)
```

Suffix can take many forms

- **ss** scalar single precision
- **ps** packed (vector) singe precision
- **sd** scalar double precision
- **pd** packed double precision
- **si#** scalar integer (8, 16, 32, 64, 128 bits)
- **su#** scalar unsigned integer (8, 16, 32, 64, 128 bits)

Data Types

- Few examples

__m128	packed single precision (vector XMM register)
__m128d	packed double precision (vector XMM register)
__m128i	packed integer (vector XMM register)

Load four 16-byte aligned single precision values in a vector

```
float a[4]={1.0,2.0,3.0,4.0}; // a must be 16-byte aligned
__m128 x = _mm_load_ps(a);
```

Add two vectors containing four single precision values

```
__m128 a, b;
__m128 c = _mm_add_ps(a, b);
```

Examples with Intrinsics

- Check CPU features
- Understanding alignment cost with intrinsics
- Inclusive prefix sum with SSE
- Makefile

Summary

- Relevance of vectorization to improve program performance is likely to increase in the future as vector lengths grow
- Compilers are often only partially successful at vectorizing code
- When the compiler fails, programmers can
  - add compiler directives, or
  - apply loop transformations
- If after transforming the code, the compiler still fails to vectorize (or the performance of the generated code is poor), use vector extensions (e.g., intrinsics or assembly) directly

References

J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach. Sections 4.1–4.2, 6th edition, Morgan Kaufmann.

M. Garzarán et al. Program Optimization Through Loop Vectorization.

M. Voss. Topics in Loop Vectorization.

K. Rogozhin. Vectorization.