LINMA2710 - Scientific Computing Single Instruction Multiple Data (SIMD)

P.-A. Absil and B. Legat

☐ Full Width Mode ☐ Present Mode

Table of Contents

Motivation

The need for parallelism

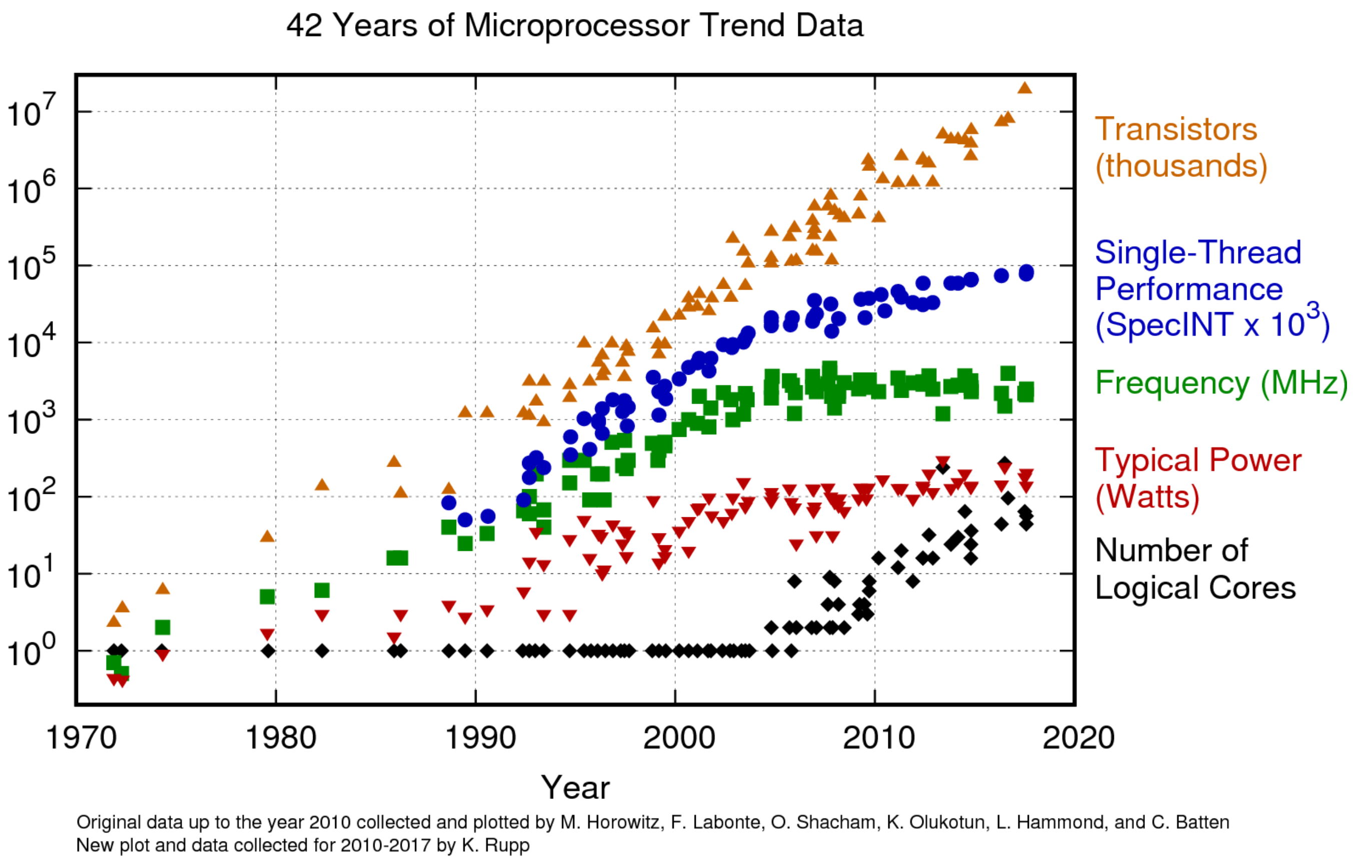
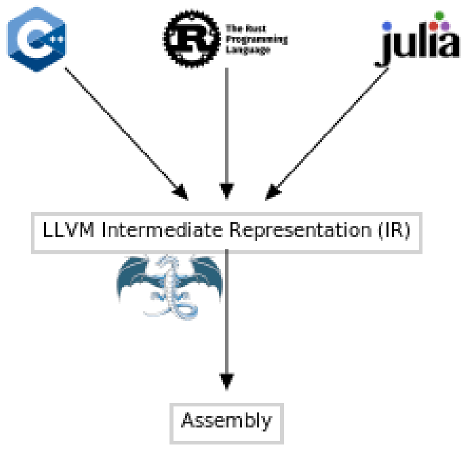


Image source

A bit of historical context

- **1972** : C language created by Dennis Ritchie and Ken Thompson to ease development of Unix (previously developed in **assembly**)
- **1985** : C++ created by Bjarne Stroustrup
- **2003** : LLVM started at University of Illinois
- **2005** : Apple hires Chris Lattner from the university
- **2007** : He then creates the LLVM-based compiler Clang
- **2009** : Mozilla start developing an LLVM-based compiler for Rust
- **2009** : Develpment starts on Julia, with LLVM-based compiler


The Rust Programming Language
julia
LLVM Intermediate Representation (IR)
Assembly

A sum function in C and Julia

```c
float sum(float *vec, int length) {
    float total = 0;
    for (int i = 0; i < length; i++) {
        total += vec[i];
    }
    return total;
}
```

```julia
c_sum(x::Vector{Cfloat}) = ccall(("sum", sum_float_lib), Cfloat, (Ptr{Cfloat},
Cint), x, length(x));
```

```
julia_sum (generic function with 1 method)
```

```julia
function julia_sum(v::Vector{T}) where {T}
    total = zero(T)
    for i in eachindex(v)
        total += v[i]
    end
    return total
end
```

Let's make a small benchmark

```
vec_float =
▸[0.900706, 0.921185, 0.46699, 0.0514815, 0.217333, 0.635073, 0.987654, 0.457924, 0.361519
```

```julia
vec_float = rand(Float32, 2^16)
```

```
32707.676f0
```

```julia
@btime c_sum($vec_float)
```

```
242.511 μs (0 allocations: 0 bytes)
```

```
32707.676f0
```

```julia
@btime julia_sum($vec_float)
```

```
60.632 μs (0 allocations: 0 bytes)
```

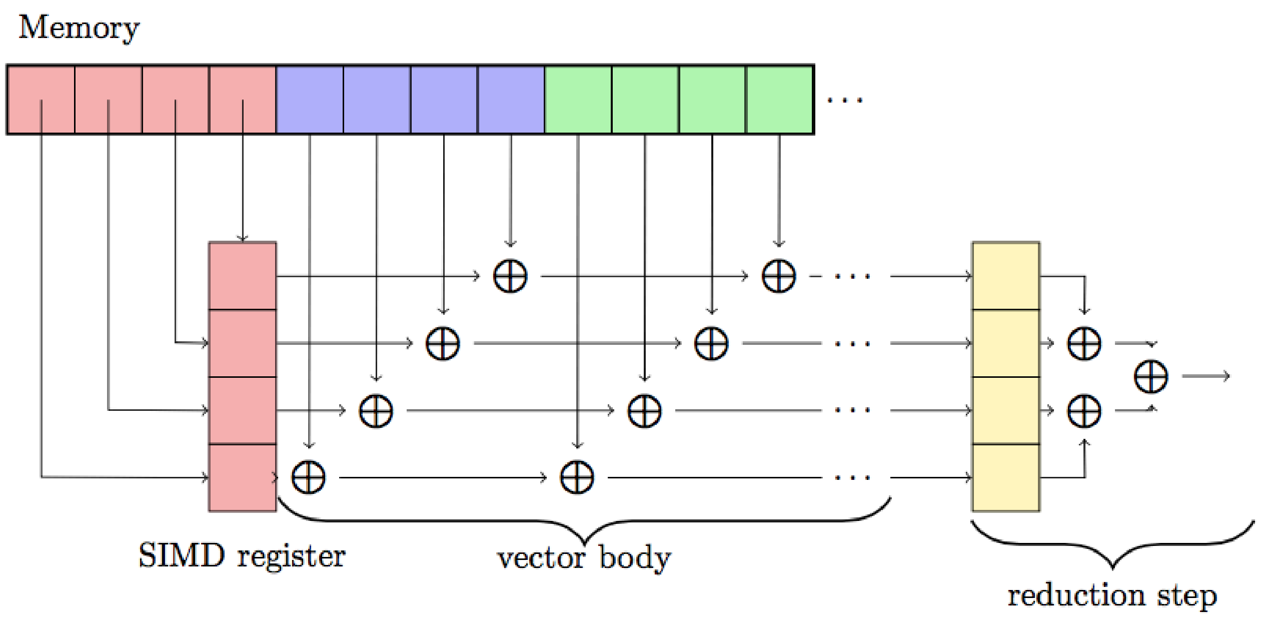
▸ **How to speed up the C code ?**

Tip

As accessing global variables is slow in Julia, it is important to add `$` in front of them when using `btime`. This is less critical in Pluto though as it handles global variables differently. To see why, try removing the `$`, you should see `1` allocations instead of zero.

```c
float sum(float *vec, int length) {
    float total = 0;
    for (int i = 0; i < length; i++) {
        total += vec[i];
    }
    return total;
}
```

Summing with SIMD



Faster Julia code

▶ **How to get the same speed up from the Julia code ?**

```
julia_sum_fast (generic function with 1 method)
```

```julia
function julia_sum_fast(v::Vector{T}) where {T}
    total = zero(T)
    for i in eachindex(v)
        @fastmath total += @inbounds v[i]
    end
    return total
end
```

```
32707.625f0
```

```julia
@btime julia_sum_fast($vec_float)
```

```
2.541 μs (0 allocations: 0 bytes)
```

```
julia_sum_simd (generic function with 1 method)
```

```
function julia_sum_simd(v::Vector{T}) where {T}
    total = zero(T)
    @simd for i in eachindex(v)
        total += v[i]
    end
    return total
end
```

```
32707.629f0
```

```
@btime julia_sum_simd($vec_float)
```

```
2.541 μs (0 allocations: 0 bytes)
```

Careful with fast math

▶ **Why are the three elements in the center of the vector ignored in this example ?**

```
test_kahan = ▸[1.0, 2.98023f-8, 2.98023f-8, 2.98023f-8, 0.000119209]
```

```
test_kahan = Cfloat[1.0, eps(Cfloat)/4, eps(Cfloat)/4, eps(Cfloat)/4,
1000eps(Cfloat)]
```

```
1.000119298696518
```

```
sum(Float64.(test_kahan))
```

```
1.0001192f0
```

```
c_sum(test_kahan[[1, 5]])
```

```
1.0001192f0
```

```
c_sum(test_kahan)
```

To improve the accuracy this, we consider the Kahan summation algorithm.

```
1.0001193f0
```

```
c_sum_kahan(test_kahan)
```

Optimization level : -O0 ⌄   Enable `-ffast-math` ? ☐

▶ **What happens when `-ffast-math` is enabled ?**

For further details, see this blog post.

Tip

`eps` gives the difference between `1` and the number closest to `1`. See also `prevfloat` and `nextfloat`.

```c
float sum_kahan(float* vec, int length) {
    float total, c, t, y;
    int i;
    total = c = 0.0f;
    for (i = 0; i < length; i++) {
      y = vec[i] - c;
      t = total + y;
      c = (t - total) - y;
      total = t;
   }
   return total;
}
```

SIMD inspection

Instruction sets

The data is **packed** on a single SIMD unit whose width and register depends on the instruction set family. The single instruction is then run in parallel on all elements of this small **vector** stored in the SIMD unit. These give the prefix `vp` to the instruction names that stands from *Vectorized Packed*.

Instruction Set Family	Width of SIMD unit	Register
`Streaming SIMD Extension (SSE)`	`128-bit`	`%xmm`
`Advanced Vector Extensions (AVX)`	`256-bit`	`%ymm`
`AVX-512`	`512-bit`	`%zmm`

```
▸ProcessChain([Process(`lscpu`, ProcessExited(0)), Process(`grep Flag`, ProcessExited(0))
```

```julia
run(pipeline(`lscpu`, `grep Flag`))
```

```
Flags:                              fpu vme de pse tsc msr pae mce cx8 ap
ic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmx
ext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl tsc_reliable nonstop
_tsc cpuid extd_apicid aperfmperf tsc_known_freq pni pclmulqdq ssse3 fma cx16
pcid sse4_1 sse4_2 movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm c
mp_legacy svm cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw topoext vmmc
all fsgsbase bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt clwb
sha_ni xsaveopt xsavec xgetbv1 xsaves user_shstk clzero xsaveerptr rdpru arat
npt nrip_save tsc_scale vmcb_clean flushbyasid decodeassists pausefilter pfthr
eshold v_vmsave_vmload umip vaes vpclmulqdq rdpid fsrm
```

Tip

To determine which instruction set is supported for your computer, look at the `Flags` list in the output of `lscpu`. We can check in the Intel® Intrinsics Guide that `avx`, `avx2` and `avx_vnni` are in the AVX family.

SIMD at LLVM level

How can you check that SIMD is enable ? Let's check at the level of LLVM IR.

```
f (generic function with 1 method)
```

```julia
function f(x1, x2, x3, x4, y1, y2, y3, y4)
    z1 = x1 + y1
    z2 = x2 + y2
    z3 = x3 + y3
    z4 = x4 + y4
    return z1, z2, z3, z4
end
```

```julia
@code_llvm debuginfo=:none f(1, 2, 3, 4, 5, 6, 7, 8)
```

```
; Function Signature: f(Int64, Int64, Int64, Int64, Int64, Int64, Int64, In
t64)
define void @julia_f_24386(ptr noalias nocapture noundef nonnull sret([4 x i6
4]) align 8 dereferenceable(32) %sret_return, i64 signext %"x1::Int64", i64 si
gnext %"x2::Int64", i64 signext %"x3::Int64", i64 signext %"x4::Int64", i64 si
gnext %"y1::Int64", i64 signext %"y2::Int64", i64 signext %"y3::Int64", i64 si
gnext %"y4::Int64") #0 {
top:
  %0 = add i64 %"y1::Int64", %"x1::Int64"
  %1 = add i64 %"y2::Int64", %"x2::Int64"
  %2 = add i64 %"y3::Int64", %"x3::Int64"
  %3 = add i64 %"y4::Int64", %"x4::Int64"
  store i64 %0, ptr %sret_return, align 8
  %"new::Tuple.sroa.2.0.sret_return.sroa_idx" = getelementptr inbounds i8, ptr
%sret_return, i64 8
  store i64 %1, ptr %"new::Tuple.sroa.2.0.sret_return.sroa_idx", align 8
  %"new::Tuple.sroa.3.0.sret_return.sroa_idx" = getelementptr inbounds i8, ptr
%sret_return, i64 16
  store i64 %2, ptr %"new::Tuple.sroa.3.0.sret_return.sroa_idx", align 8
  %"new::Tuple.sroa.4.0.sret_return.sroa_idx" = getelementptr inbounds i8, ptr
%sret_return, i64 24
  store i64 %3, ptr %"new::Tuple.sroa.4.0.sret_return.sroa_idx", align 8
  ret void
}
```

Tip

If we see `add i64`, it means that each `Int64` is added independently

Packing the data to enable SIMD

```
f_broadcast (generic function with 1 method)
```

```julia
function f_broadcast(x, y)
    z = x .+ y
    return z
end
```

```julia
@code_llvm debuginfo=:none f_broadcast((1, 2, 3, 4), (1, 2, 3, 4))
```

```
; Function Signature: f_broadcast(NTuple{4, Int64}, NTuple{4, Int64})
define void @julia_f_broadcast_23555(ptr noalias nocapture noundef nonnull sret([4 x i64]) align 8 dereferenceable(32) %sret_return, ptr nocapture noundef nonnull readonly align 8 dereferenceable(32) %"x::Tuple", ptr nocapture noundef nonnull readonly align 8 dereferenceable(32) %"y::Tuple") #0 {
top:
  %0 = load <4 x i64>, ptr %"x::Tuple", align 8
  %1 = load <4 x i64>, ptr %"y::Tuple", align 8
  %2 = add <4 x i64> %1, %0
  store <4 x i64> %2, ptr %sret_return, align 8
  ret void
}
```

Tip

`load <4 x i64>` means that 4 `Int64` are loaded into a 256-bit wide SIMD unit.

SIMD at assembly level

```julia
@code_native debuginfo=:none f_broadcast((1, 2, 3, 4), (1, 2, 3, 4))
```

```
	.text
	.file	"f_broadcast"
	.globl	julia_f_broadcast_23765         # -- Begin function julia_f_broadc
ast_23765
	.p2align	4, 0x90
	.type	julia_f_broadcast_23765,@function
julia_f_broadcast_23765:                # @julia_f_broadcast_23765
; Function Signature: f_broadcast(NTuple{4, Int64}, NTuple{4, Int64})
# %bb.0:                                # %top
	#DEBUG_VALUE: f_broadcast:x <- [DW_OP_deref] [$rsi+0]
	#DEBUG_VALUE: f_broadcast:y <- [DW_OP_deref] [$rdx+0]
	push	rbp
	vmovdqu	ymm0, ymmword ptr [rdx]
	mov	rbp, rsp
	mov	rax, rdi
	vpaddq	ymm0, ymm0, ymmword ptr [rsi]
	vmovdqu	ymmword ptr [rdi], ymm0
	pop	rbp
	vzeroupper
	ret
.Lfunc_end0:
	.size	julia_f_broadcast_23765, .Lfunc_end0-julia_f_broadcast_23765
                                        # -- End function
	.type	".L+Core.Tuple#23767",@object   # @".L+Core.Tuple#23767"
	.section	.rodata,"a",@progbits
	.p2align	3, 0x0
".L+Core.Tuple#23767":
	.quad	".L+Core.Tuple#23767.jit"
	.size	".L+Core.Tuple#23767", 8
```

Tip

The suffic `v` in front of the instruction stands for `vectorized`. It means it is using a SIMD unit.

Tuples implementing the array interface

N = 2

```julia
let
    T = Float64
    A = rand(SMatrix{N,N,T})
    x = rand(SVector{N,T})
    @code_llvm debuginfo=:none A * x
end
```

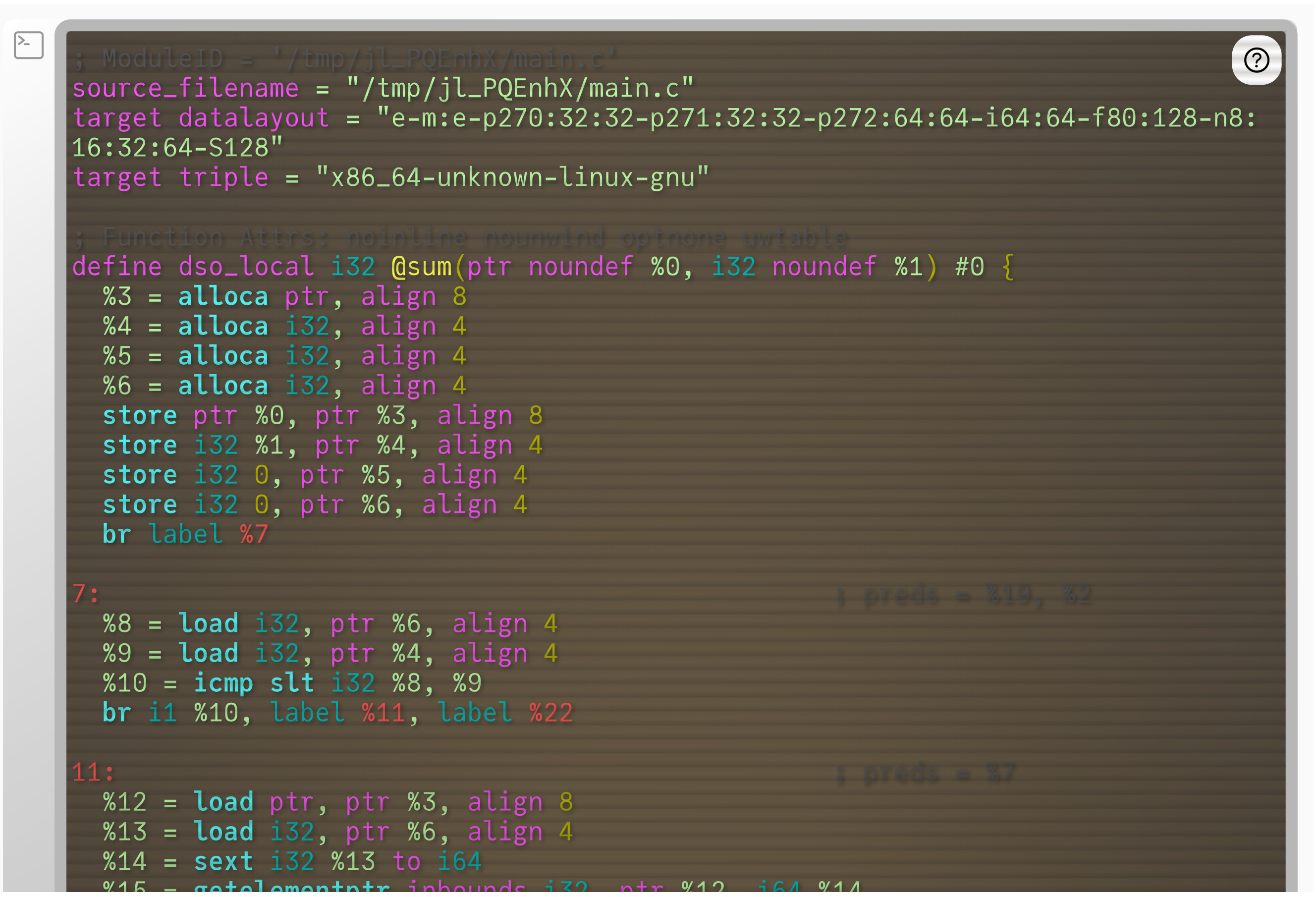
```
; Function Signature: *(StaticArraysCore.SArray{Tuple{2, 2}, Float64, 2,
4}, StaticArraysCore.SArray{Tuple{2}, Float64, 1, 2})
define void @"julia_*_24281"(ptr noalias nocapture noundef nonnull sret([1 x
[2 x double]]) align 8 dereferenceable(16) %sret_return, ptr nocapture noundef
nonnull readonly align 8 dereferenceable(32) %"A::SArray", ptr nocapture nound
ef nonnull readonly align 8 dereferenceable(16) %"B::SArray") #0 {
top:
  %"A::SArray.data_ptr[3]_ptr" = getelementptr inbounds [4 x double], ptr
%"A::SArray", i64 0, i64 2
  %0 = load <2 x double>, ptr %"B::SArray", align 8
  %1 = load <2 x double>, ptr %"A::SArray", align 8
  %2 = shufflevector <2 x double> %0, <2 x double> poison, <2 x i32> zeroiniti
alizer
  %3 = fmul contract <2 x double> %1, %2
  %4 = load <2 x double>, ptr %"A::SArray.data_ptr[3]_ptr", align 8
  %5 = shufflevector <2 x double> %0, <2 x double> poison, <2 x i32> <i32 1, i
32 1>
  %6 = fmul contract <2 x double> %4, %5
  %7 = fadd contract <2 x double> %3, %6
  store <2 x double> %7, ptr %sret_return, align 8
  ret void
}
```

Tip

Small arrays that are allocated on the stack like tuples and implemented in `StaticArrays.jl`. Operating on them leverages SIMD.

Auto-Vectorization

LLVM Loop Vectorizer for a C array



```
; ModuleID = '/tmp/jl_PQEnhX/main.c'
source_filename = "/tmp/jl_PQEnhX/main.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:
16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @sum(ptr noundef %0, i32 noundef %1) #0 {
  %3 = alloca ptr, align 8
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  store ptr %0, ptr %3, align 8
  store i32 %1, ptr %4, align 4
  store i32 0, ptr %5, align 4
  store i32 0, ptr %6, align 4
  br label %7

7:                                                ; preds = %19, %2
  %8 = load i32, ptr %6, align 4
  %9 = load i32, ptr %4, align 4
  %10 = icmp slt i32 %8, %9
  br i1 %10, label %11, label %22

11:                                               ; preds = %7
  %12 = load ptr, ptr %3, align 8
  %13 = load i32, ptr %6, align 4
  %14 = sext i32 %13 to i64
  %15 = getelementptr inbounds i32, ptr %12, i64 %14
```

```c
int sum(int *vec, int length) {
    int total = 0;
    for (int i = 0; i < length; i++) {
        total += vec[i];
    }
    return total;
}
```

No pragma

No pragma

No pragma

Element type : int

Optimization level : -O0

☐ -msse3

☐ -mavx2

☐ -mavx512f

☐ -ffast-math

LLVM Loop Vectorizer for a C++ vector

```
; ModuleID = '/tmp/jl_3aOppx/main.c'
source_filename = "/tmp/jl_3aOppx/main.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:
16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @sum(ptr noundef %0, i32 noundef %1) #0 {
  %3 = alloca ptr, align 8
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  store ptr %0, ptr %3, align 8
  store i32 %1, ptr %4, align 4
  store i32 0, ptr %5, align 4
  store i32 0, ptr %6, align 4
  br label %7

7:                                                ; preds = %19, %2
  %8 = load i32, ptr %6, align 4
  %9 = load i32, ptr %4, align 4
  %10 = icmp slt i32 %8, %9
  br i1 %10, label %11, label %22

11:                                               ; preds = %7
  %12 = load ptr, ptr %3, align 8
  %13 = load i32, ptr %6, align 4
  %14 = sext i32 %13 to i64
  %15 = getelementptr inbounds i32, ptr %12, i64 %14
```

```
32707.676f0
```

```julia
@btime cpp_sum($vec_float)
```

```
297.925 μs (0 allocations: 0 bytes)
```

```julia
cpp_sum(x::Vector{Cfloat}) = ccall(("c_sum", cpp_sum_float_lib), Cfloat,
(Ptr{Cfloat}, Cint), x, length(x));
```

```cpp
#include <vector>

int my_sum(std::vector<int> vec) {
  int total = 0;
  for (int i = 0; i < vec.size(); i++) {
    total += vec[i];
  }
  return total;
}

extern "C" {
int c_sum(int *array, int length) {
  std::vector<int> v;
  v.assign(array, array + length);
  return my_sum(v);
}}
```

No pragma

No pragma

No pragma

Element type : int

Optimization level : -O0

- ☐ -msse3
- ☐ -mavx2
- ☐ -mavx512f
- ☐ -ffast-math

Tip

Easily call C++ code from Julia or Python by adding a C interface like the `c_sum` in this example.

LLVM Superword-Level Parallelism (SLP) Vectorizer

```
f (generic function with 2 methods)
```

```julia
f(a, b) = (a[1] + b[1], a[2] + b[2], a[3] + b[3], a[4] + b[4])
```

```julia
@code_llvm debuginfo=:none f((1, 2, 3, 4), (5, 6, 7, 8))
```

```
; Function Signature: f(NTuple{4, Int64}, NTuple{4, Int64})
define void @julia_f_24422(ptr noalias nocapture noundef nonnull sret([4 x i6
4]) align 8 dereferenceable(32) %sret_return, ptr nocapture noundef nonnull re
adonly align 8 dereferenceable(32) %"a::Tuple", ptr nocapture noundef nonnull
readonly align 8 dereferenceable(32) %"b::Tuple") #0 {
top:
  %0 = load <4 x i64>, ptr %"a::Tuple", align 8
  %1 = load <4 x i64>, ptr %"b::Tuple", align 8
  %2 = add <4 x i64> %1, %0
  store <4 x i64> %2, ptr %sret_return, align 8
  ret void
}
```

Inspection with godbolt Compiler Explorer

Source Editor: C source #1

```c
void foo(int a1, int a2, int b1, int b2, int *A) {
  A[0] = a1 * (a1 + b1);
  A[1] = a2 * (a2 + b2);
  A[2] = a1 * (a1 + b1);
  A[3] = a2 * (a2 + b2);
}
```

Compiler Output: x86-64 clang 19.1.0 (Editor #1)

Flags: `-O3 -mavx2`

```asm
foo:
        add     edx, edi
        imul    edx, edi
        mov     dword ptr [r8], edx
        add     ecx, esi
        imul    ecx, esi
```

Edit on Com

Example source

Further readings

Slides inspired from:

- SIMD in Julia
- Demystifying Auto-vectorization in Julia
- Auto-Vectorization in LLVM

```
Activating project at `~/work/LINMA2710/LINMA2710/Lectures`
```