RADBOUD RADIO LAB

Compiling and optimization for high performance systems

Roel Jordans
Radboud University & Eindhoven University of Technology

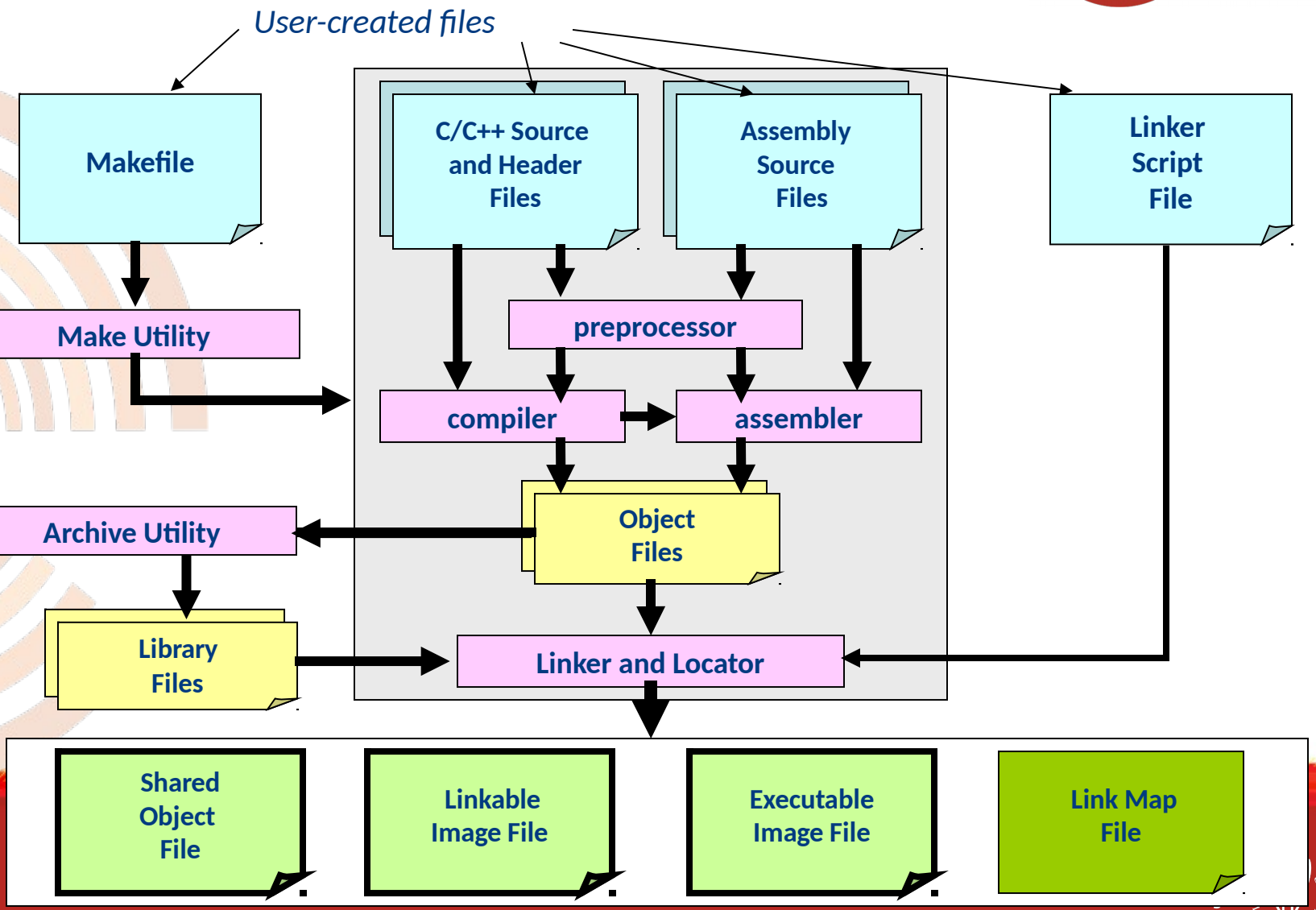
Radboud Universiteit Nijmegen

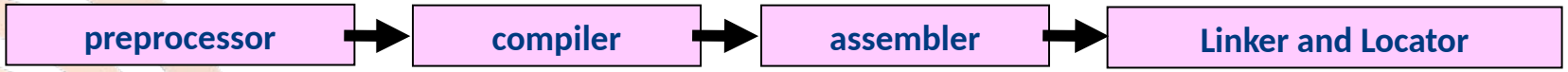
RADBOUD RADIO LAB

Chapter 1: Down the rabbit hole

Compiler organization

Compilation flow overview



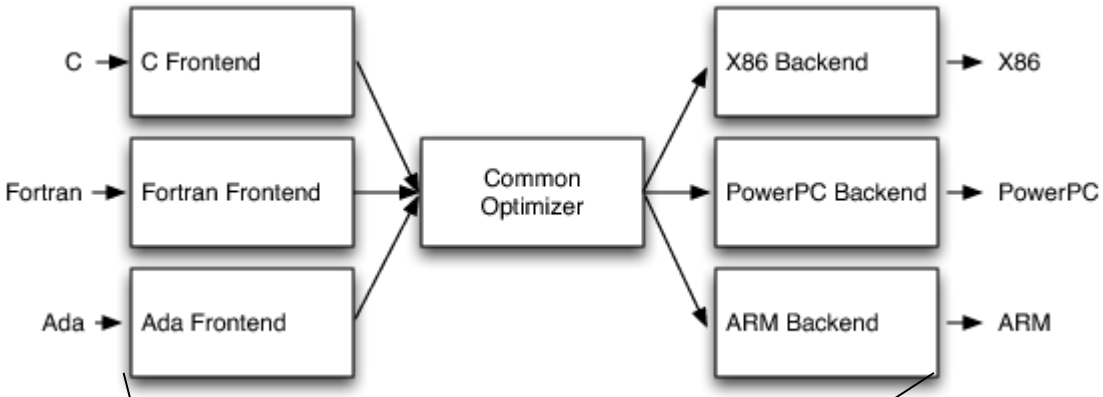

preprocessor
compiler
assembler
Linker and Locator

Preprocessor

- Adds some really useful extra features
  - #include
  - #define
  - #ifdef
- Also useful for other languages
  - Like the assembler

The compiler

- Compilation in three steps
  - AST, IR, Code generation



An example from C

```
int foo(int a, int b) { return a + b; }
```

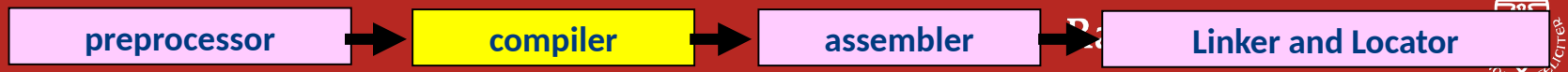
```
int foo(int a, int b) { return a + b; }
```

Abstract Syntax Tree (AST)

```
$ clang -Xclang -ast-dump -fsyntax-only f.c

TranslationUnitDecl ...
|-TypedefDecl ...
`-FunctionDecl ... <f.c:1:1, line:3:1> line:1:5 foo 'int(int,int)'
  |-ParmVarDecl ... <col:9, col:13> col:13 used a 'int'
  |-ParmVarDecl ... <col:16, col:20> col:20 used b 'int'
  `-CompoundStmt ... <col:23, line:3:1>
    `-ReturnStmt ... <line:2:2, col:11>
      `-BinaryOperator ... 'int' '+'
        |-ImplicitCastExpr ... 'int' <LValueToRValue>
        | `-DeclRefExpr ... 'int' lvalue ParmVar ... 'a' 'int'
        `-ImplicitCastExpr ... 'int' <LValueToRValue>
          `-DeclRefExpr ... 'int' lvalue ParmVar ... 'b' 'int'
```

AST Tools

- Direct translation back to C
  - Code formatting with clang-format
- AST analysis
  - clang-check
    - Buffer checking C code
  - clang-tidy
    - Cleans up common programming mistakes
- Transforming AST
  - Source-to-source transformations
  - clang-modernize
    - Update code to use newer language features



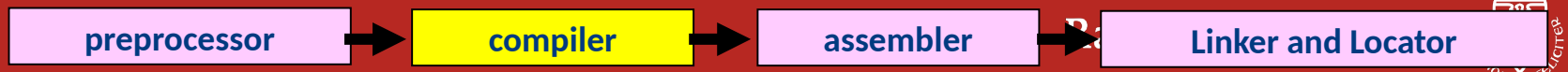
```
int foo(int a, int b) { return a + b; }
```

Intermediate Representation (IR)

```
$ clang --target=avr -emit-llvm -S f.c
```

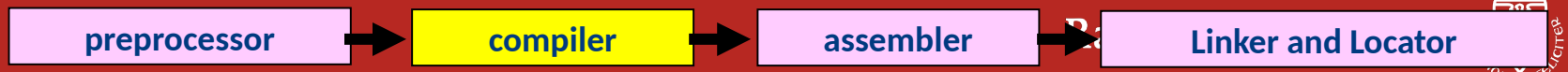
```
; ModuleID = 'f.c'
target datalayout = "..."
target triple = "avr-unknown-unknown"

; Function Attrs: nounwind
define i16 @foo(i16 %a, i16 %b) #0 {
  %1 = alloca i16, align 2
  %2 = alloca i16, align 2
  store i16 %a, i16* %1, align 2
  store i16 %b, i16* %2, align 2
  %3 = load i16* %1, align 2
  %4 = load i16* %2, align 2
  %5 = add nsw i16 %3, %4
  ret i16 %5
}
```

A kind of high-level assembly language



IR optimizations

- Code simplification
  - Dead-code elimination (DCE)
  - Common sub-expression elimination (CSE)
- Code analysis
  - Alias analysis
  - Control-flow analysis
- Code transformation
  - Loop transformation
  - Auto vectorization



```
int foo(int a, int b) { return a + b; }
```

Enabling IR optimizations

```
$ opt -S -O1 f.ll -o f.opt.ll
```

```
; ModuleID = 'f.ll'
target datalayout = "..."
target triple = "avr-unknown-unknown"

; Function Attrs: nounwind
define i16 @foo(i16 %a, i16 %b) #0 {
  %add = add nsw i16 %a, %b
  ret i16 %add
}
```

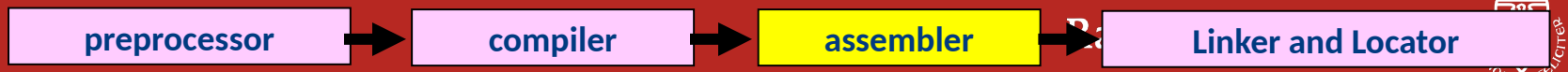
```
int foo(int a, int b) { return a + b; }
```

CodeGen

```
$ llc f.opt.ll
        .text
        .file   "f.opt.ll"
        .globl  foo
        .align  2
        .type   foo,@function
foo:                                    ; @foo
; BB#0:                                 ; %entry
        add     r24, r22
        adc     r25, r23
        ret
.Lfunc_end0:
        .size   foo, .Lfunc_end0-foo
```

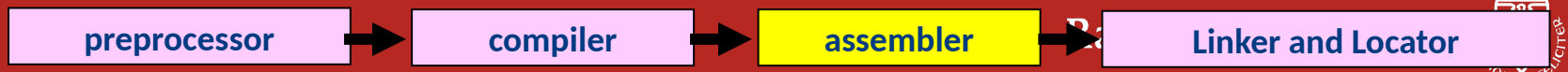
The assembler

- From human readable to executable
- Produces *object files*
  - Usually in ELF, COFF, or Mach-O format



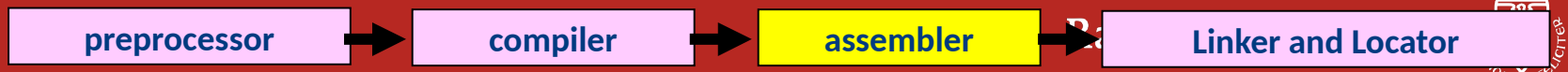
Executable and Linkable Format

- Contains the binary data for the processor
  - Code & Data
- Optionally contains extra information
  - Debug information
  - Symbol tables
  - Relocation information

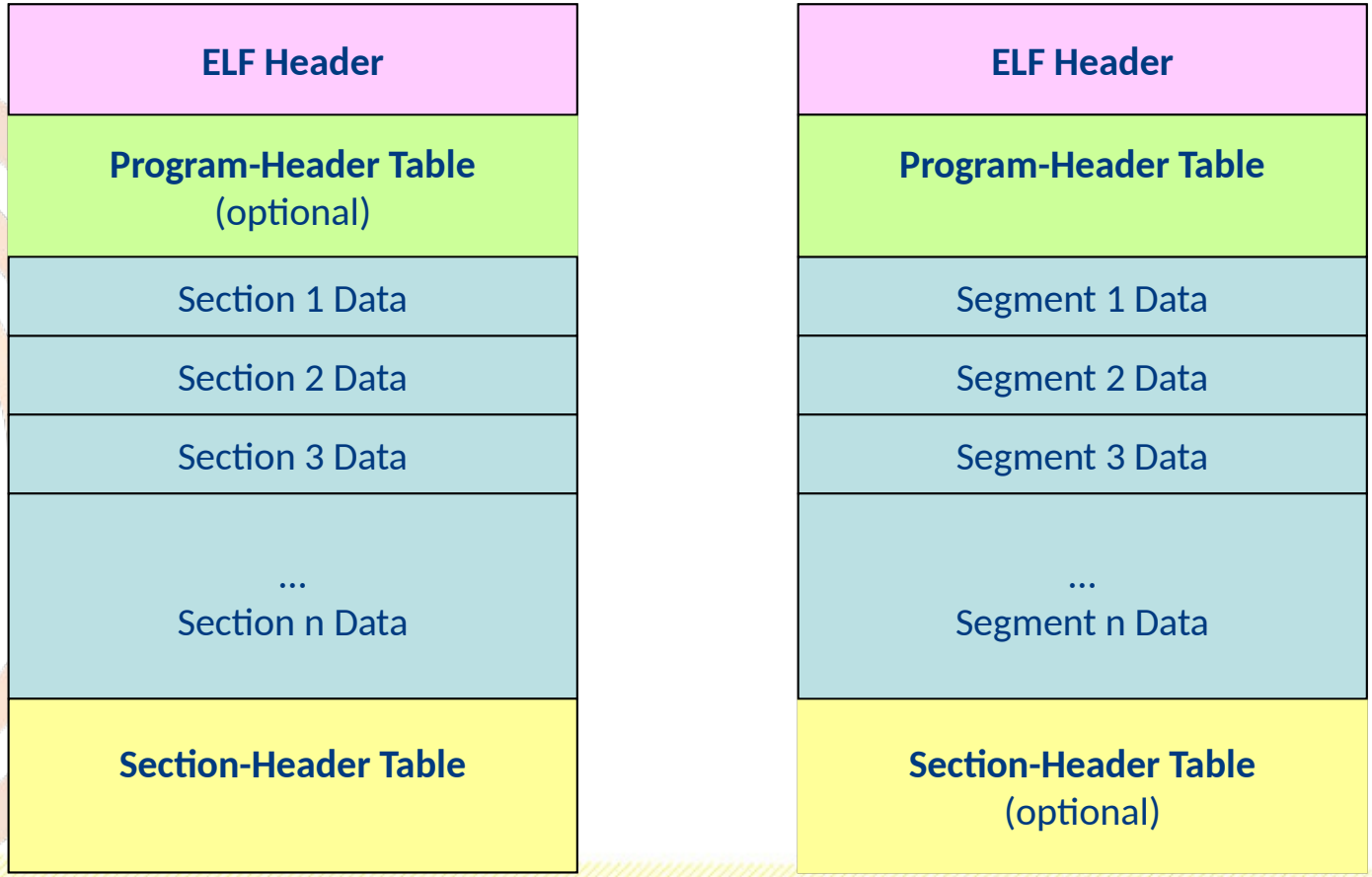


Linux ‘Executable’ ELF files

- The Executable ELF files produced by the Linux linker are configured for execution in a private ‘virtual’ address space, whereby every program gets loaded at the identical virtual memory-address



Executable versus Linkable

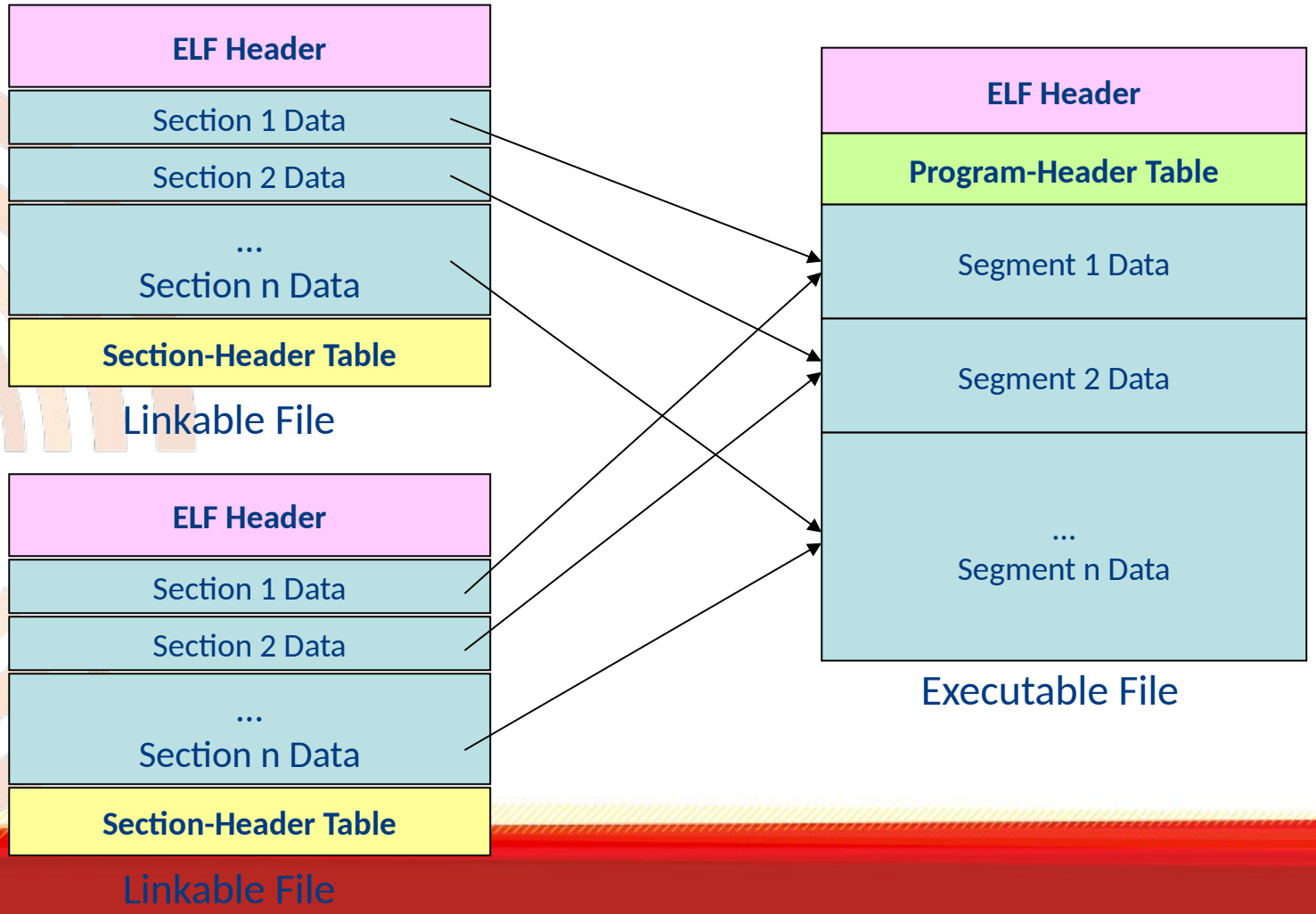


Linkable File

Executable File

preprocessor → compiler → assembler → Linker and Locator

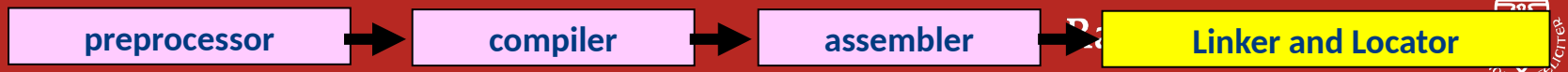
Role of the Linker



preprocessor → compiler → assembler → Linker and Locator

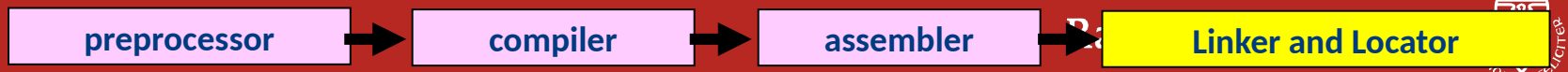
The linker

- Controllable by a linker script
  - Default provided for simple architectures
- Required when the mapping of sections to memories becomes more complex
  - Multiple data memories
  - Moving code around at runtime
  - Fat binaries: supporting multiple architectures
  - Encrypted binaries

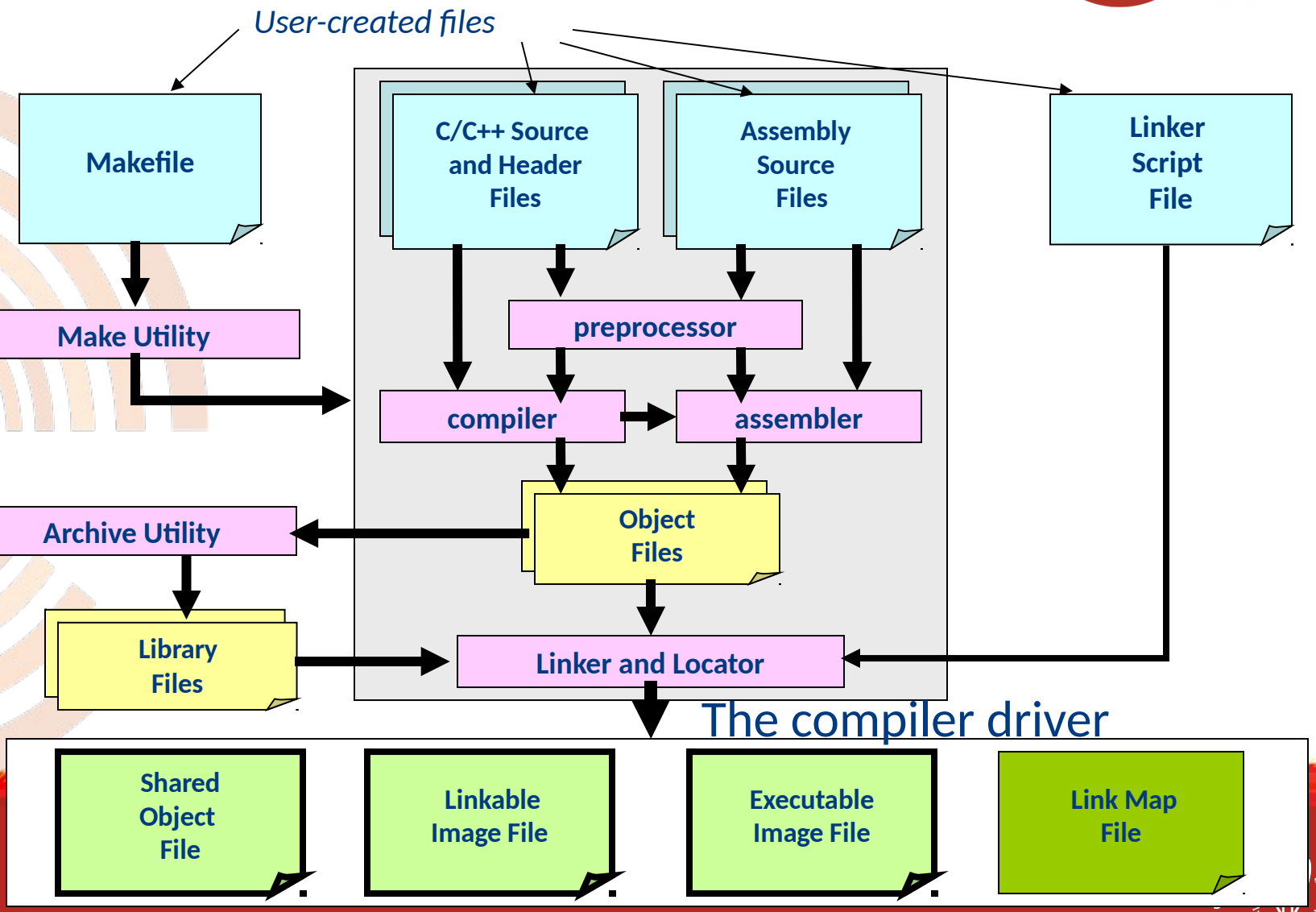


C runtime

- Initializing the processor
- What happens before and after main
  - Setting up the stack
  - Calling main
  - Stopping execution



To summarize



The compiler driver

- Many different tools in the compilation process
  - Mostly useful when debugging the compiler
- Needs simple interface

- The frontend program (clang) can actually call the right programs for you!

After the break chapter 2

- In Alice in Wonderland this chapter would be
  "The pool of tears"…
- For us, code optimizations using the compiler

RADBOUD RADIO LAB

Radboud University

RADBOUD RADIO LAB

Code optimization by example: Automatic vectorization

Roel Jordans
Radboud University & Eindhoven University of Technology

Radboud Universiteit Nijmegen

Welcome back

- Compiler optimization steps
- SIMD and Vectorization
  - Vector Machines
    - Vector MIPS
  - x86: MMX – SSE – AVX
  - ARM: NEON
    - We do not discuss others: e.g. PowerPC Altivec
- Auto-vectorization: an example

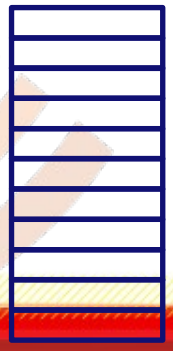
Compiler optimization steps

- Three important steps
  - Legality → Are we allowed to apply our transformation?
  - Cost → Do we gain anything by applying the transformation?
  - Transform → Restructure the code according to our findings

Data Parallelism

- Vector operations
- Multiple data elements per operation, e.g.
  - ADD**V** V1, V2, V3 // forall i V1[i] = V2[i]+V3[i]

- Executed using either
  - **Vector archtitecture:** highly pipelined (fast clocked) FU (function unit)
  - **SIMD:** multiple FUs acting in parallel



or

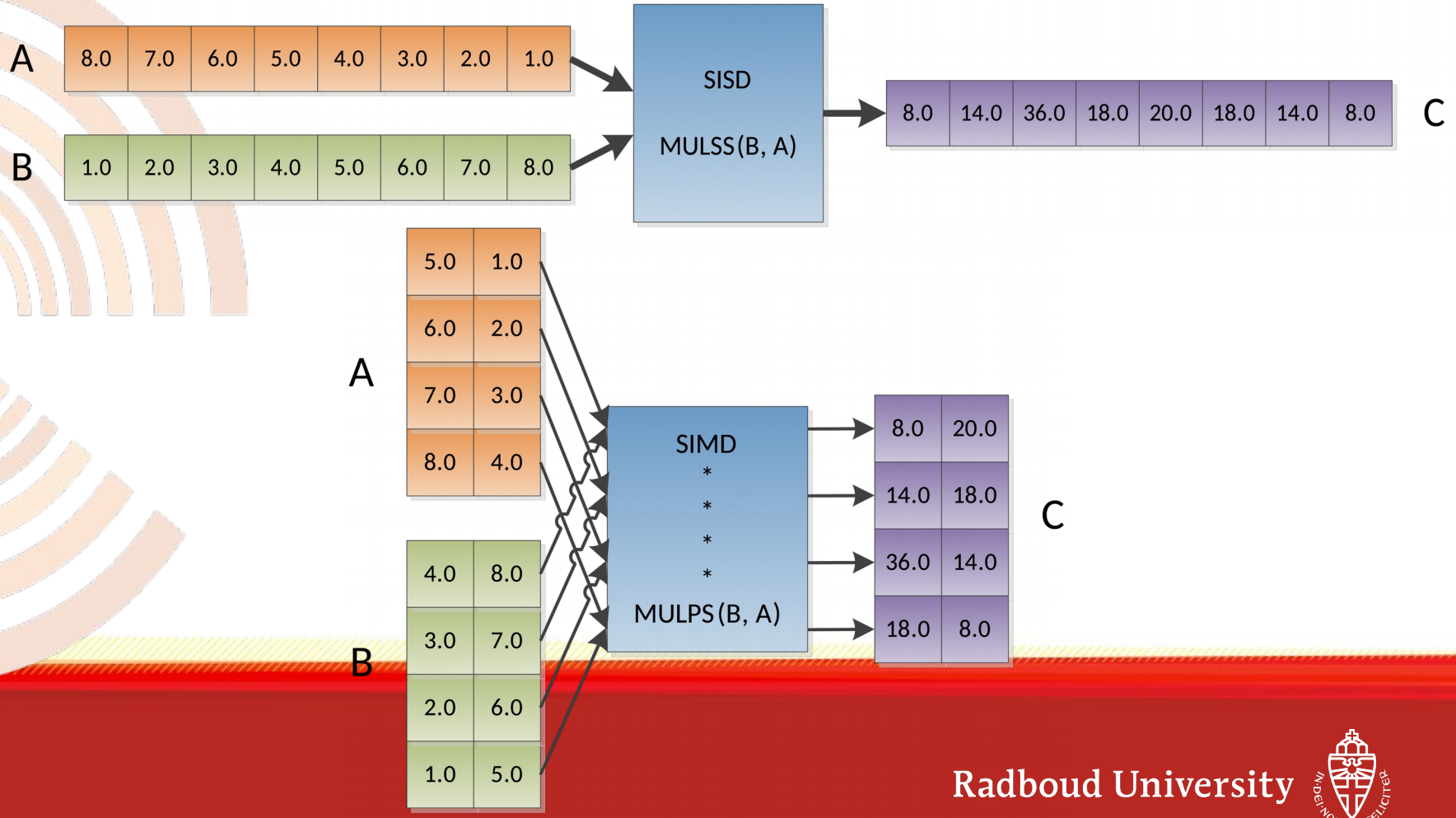
time

SIMD architecture

Vector architecture

SISD and SIMD vector operations

- C = A * B
  - `for(i=0; i<n; i++) c[i]=a[i]*b[i];`

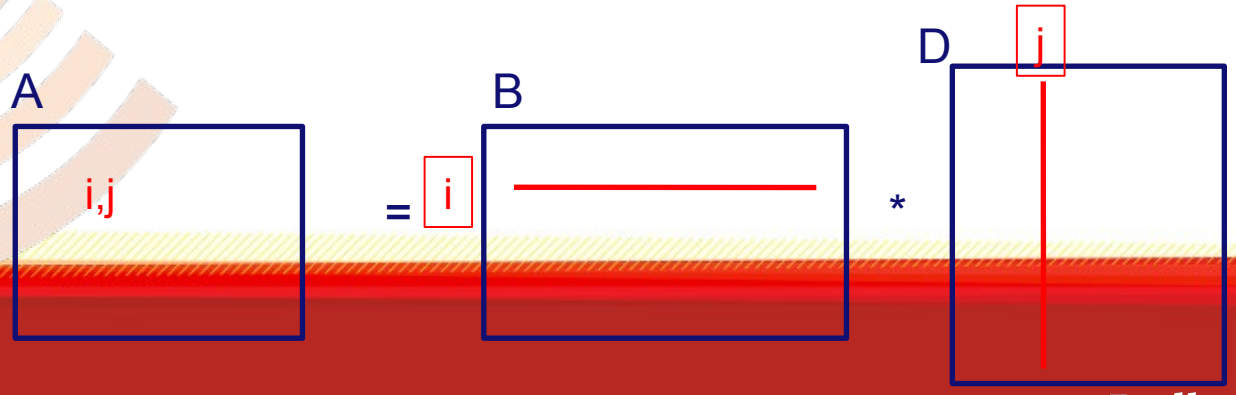


Stride & alignment

- Consider matrix multiplication: A=B*D

```
for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1) {
        A[i][j] = 0.0;
        for (k = 0; k < 100; k=k+1)
            A[i][j] += B[i][k] * D[k][j];
    }
```

- Must vectorize multiplication of rows of B with columns of D
- Use *non-unit stride*



Scatter-Gather

- Consider indirect vector access:

```
for (i = 0; i < n; i=i+1)
    A[K[i]] = A[K[i]] + C[M[i]];
```

- Use index vector to load e.g. only the non-zero elements of A into vector Va:

```
LV     Vk, Rk           ;load K
LVI       Va, (Ra+Vk)     ;load A[K[]]
LV     Vm, Rm           ;load M
LVI       Vc, (Rc+Vm)     ;load C[M[]]
ADDVV.D   Va, Va, Vc        ;add them
SVI       (Ra+Vk), Va     ;store A[K[]]
```

Vector support on x86

- Let's see how Intel and AMD support subword / SIMD parallelism

x86 architecture SIMD support

- ISA SIMD support
  - MMX, 3DNow!, SSE, SSE2, SSSE3, SSE4, AVX
  - Streaming SIMD Extensions (SSE)
  - SIMD instruction set extension to the x86 architecture
- Micro architecture support
  - Many functional units (for vector operations)
  - Multiple 128-bit vector registers, XMM0, …, XMM15

Performance difference

- C = A * B
  - `for(i=0; i<n; i++) c[i]=a[i]*b[i];`

Scalar loop:

```
L1:
   movss   xmm0, [rdx+r13*4]
   mulss   xmm0, [r8+r13*4]
   movss   [rcx+r13*4], xmm0
   add     r13, 1
   cmp     r13, r9
   jl      L1
```

6 instructions / 1 element

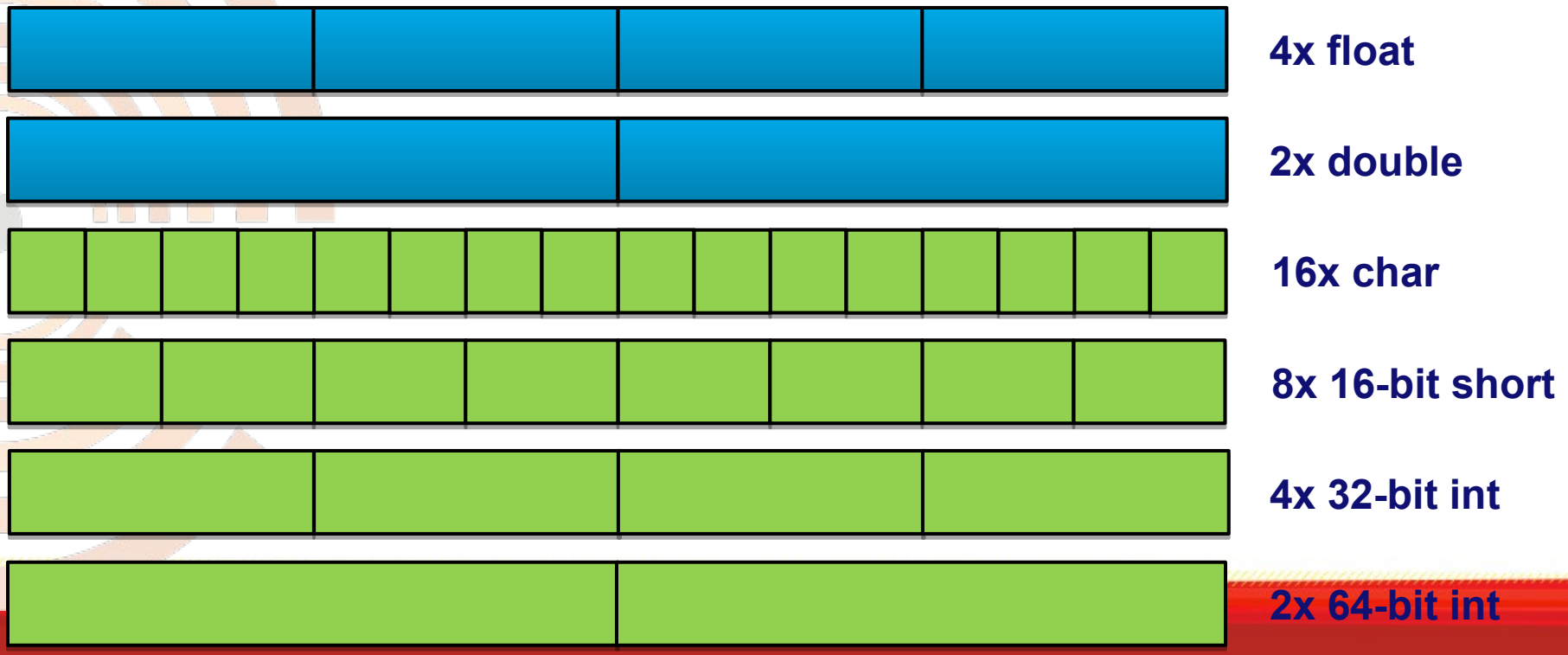
Vector loop:

```
L1:
   movups   xmm1, [rdx+r9*4]
   movups   xmm0, [r8+r9*4]
   mulps    xmm1, xmm0
   movaps   [rcx+r9*4], xmm1
   add      r9, 4
   cmp      r9, rax
   jl       L1
```

7 instructions / 4 elements 1.75 instr./element

Registers and data types (assuming 128-bit)

- Subword parallelism
- XMM registers 128-bit wide
- XMM0 to XMM15



How should we vectorize code?

- Automatic? Different compilers can help
  - Go to https://godbolt.org to test with many of them

- **GCC**
  - CFLAGS= -msse -msse2 -mssse3 -mfpmath=sse -O3 -ftree-vectorize -ftree-vectorizer-verbose=2
  - At least specify which vector types to support
- **Clang**
  - Just -O3
  - By default supports features of host architecture

Auto-vectorization not mature ?

- Only reasonably simple loops supported

```
void histogram(int * hist_out, unsigned char * img_in, int img_size, int nbr_bin){
    int i;

    for ( i = 0; i < nbr_bin; i ++){
        hist_out[i] = 0;
    }

    for ( i = 0; i < img_size; i ++){
        hist_out[img_in[i]] ++;
    }
}
for(i = 0; i < size; i ++){
    r = img_in.img[i*3];
    g = img_in.img[i*3+1];
    b = img_in.img[i*3+2];

    y  = (unsigned char)( 16 + 0.257f*r + 0.504f*g + 0.098f*b);
    cb = (unsigned char)(128 - 0.148f*r - 0.291f*g + 0.439f*b);
    cr = (unsigned char)(128 + 0.439f*r - 0.368f*g - 0.071f*b);

    img_out.img_y[i] = y;
    img_out.img_u[i] = cb;
    img_out.img_v[i] = cr;
}
```

Replaced by memset

img_size = 4288*2848
But too complex for auto vectorization

OK for current compilers, but only recent ones

What about ARM?

- ARM supports NEON SIMD instruction set

Auto vectorization (status 2009)

RADBOUD RADIO LAB

RGB to Grayscale
Optimize for NEON

In 2009: compilers do not produce NEON instructions
=> use intrinsics

(Example from http://hilbert-space.de/?p=22)

```
void reference_convert (uint8_t * restrict dest,
                        uint8_t * restrict src, int n)
{
    int i;
    for (i=0; i<n; i++) {
        int r = *src++;  // load red
        int g = *src++;  // load green
        int b = *src++; // load blue
        // build weighted average:
        int y = (r*77)+(g*151)+(b*28);
        // undo the scale by 256
        //and write to memory:
        *dest++ = (y>>8);
    }
}
```

Auto vectorization

Version on the right is the assembly produced:

Still a lot of non vector operations and storing to stack.

Compiler did a very poor job!

```
160: f46a040f vld3.8 {d16-d18}, [sl]
164: e1a0c005 mov ip, r5
168: ecc80b06 vstmia r8, {d16-d18}
16c: e1a04007 mov r4, r7
170: e2866001 add r6, r6, #1 ; 0x1
174: e28aa018 add sl, sl, #24 ; 0x18
178: e8bc000f ldm ip!, {r0, r1, r2, r3}
17c: e15b0006 cmp fp, r6
180: e1a08005 mov r8, r5
184: e8a4000f stmia r4!, {r0, r1, r2, r3}
188: eddd0b06 vldr d16, [sp, #24]
18c: e89c0003 ldm ip, {r0, r1}
190: eddd2b08 vldr d18, [sp, #32]
194: f3c00ca6 vmull.u8 q8, d16, d22
198: f3c208a5 vmlal.u8 q8, d18, d21
19c: e8840003 stm r4, {r0, r1}
1a0: eddd3b0a vldr d19, [sp, #40]
1a4: f3c308a4 vmlal.u8 q8, d19, d20
1a8: f2c80830 vshrn.i16 d16, q8, #8
1ac: f449070f vst1.8 {d16}, [r9]
1b0: e2899008 add r9, r9, #8 ; 0x8
1b4: caffffe9 bgt 160
```

Auto vectorization

To get the performance we would manually write the code

See right side:

- 7.5x faster than plain C

But a lot of work!

```
convert_asm_neon:
    # r0: Ptr to destination data
    # r1: Ptr to source data
    # r2: Iteration count:
     push {r4-r5,lr}
    lsr r2, r2, #3
    # build the three constants:
    mov r3, #77
    mov r4, #151
    mov r5, #28
    vdup.8 d3, r3
    vdup.8 d4, r4
    vdup.8 d5, r5
. loop:
    # load 8 pixels:
    vld3.8 {d0-d2}, [r1]!
    # do the weight average:
    vmull.u8 q3, d0, d3
    vmlal.u8 q3, d1, d4
    vmlal.u8 q3, d2, d5
    # shift and store:
    vshrn.u16 d6, q3, #8
    vst1.8 {d6}, [r0]!

    subs r2, r2, #1
    bne .loop
    pop { r4-r5, pc }
```

Auto vectorization (status 2017)

We wanted to optimize the RGB to Grayscale code on the right using NEON.

Lets try it with the latest LLVM

```
void reference_convert (uint8_t * restrict dest,
                        uint8_t * restrict src, int n)
{
    int i;
    for (i=0; i<n; i++) {
        int r = *src++;  // load red
        int g = *src++;  // load green
        int b = *src++; // load blue
        // build weighted average:
        int y = (r*77)+(g*151)+(b*28);
        // undo the scale by 256
        //and write to memory:
        *dest++ = (y>>8);
    }
}
```

Auto vectorization (status 2017)

That vectorized nicely!

- Who can spot the loop bound check now?
- What else changed from our manual version?

```
.LBB0_5:
    vld3.8  {d20, d22, d24}, [r1]!
    subs    r5, r5, #16
    vld3.8  {d21, d23, d25}, [r1]!
    vmull.u8    q14, d23, d16
    vorr    d19, d25, d25
    vmull.u8    q15, d22, d16
    vmull.u8    q0, d19, d17
    vorr    d19, d24, d24
    vmlal.u8    q14, d21, d18
    vmull.u8    q1, d19, d17
    vmlal.u8    q15, d20, d18
    vaddhn.i16  d21, q14, q0
    vaddhn.i16  d20, q15, q1
    vst1.8  {d20, d21}, [r0]!
    bne .LBB0_5
```

Manual vectorization

- Convert performance bottlenecks to Assembly

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

Vector loop:

```
L1:
  movups   xmm1, [rdx+r9*4]
  movups   xmm0, [r8+r9*4]
  mulps    xmm1, xmm0
  movaps   [rcx+r9*4], xmm1
  add      r9, 4
  cmp      r9, rax
  jl       L1
```

Complex!

- Utilize SSE with intrinsics
  - Similar to a function call
  - Translates to one or more lines of assembly
  - Supported in modern compilers, GCC, ICC, LLVM

Intrinsics: How to enable them?

- Header files to access SSE intrinsics
  - #include <mmintrin.h> // MMX
  - #include <xmmintrin.h> // SSE
  - #include <emmintrin.h> // SSE2
  - #include <pmmintrin.h> // SSE3
  - #include <tmmintrin.h> // SSSE3
  - #include <smmintrin.h> // SSE4

- ARM Intrinsics
  - #include <arm_neon.h>
  - Need to enable Neon through -mfpu=neon or -mfloat-abi=hard

https://software.intel.com/sites/landingpage/IntrinsicsGuide/
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0002a/ch01s04s02.html

Simple example

Plain C/C++

```
int i, n=100;

for(i=0; i<n; i++)
  p[i] += q[i]
```

SSE intrinsics

```
int i, n=100;
_m128 a,b;

for(i=0; i<n; i+=4){
  _mm_loadu_ps(a,p);
  _mm_loadu_ps(b,q);

  a = _mm_add_ps(a,b);

  _mm_store_ps(p,a);
}
```

Careful aligned store only works if 16-byte aligned

For recent architectures penalty for unaligned small

SSE intrinsics: different kinds (1)

- Data types (mapped to an XMM register)
  - __m128 float
  - __m128 double
  - __m128i integer
- Data movement and initialization

Load unaligned data

```
__m128 _mm_load_ps(float * p);
```

- _mm_load_ps/pd/si128, _mm_loadu_ps/pd/si128

```
__m128 _mm_set_ps(float z,float y,float x,float w);
```

- _mm_set_ps/pd/si128

```
void _mm_store_ps(float *p, __m128 a);
```

- _mm_store_ps/pd/si128, _mm_storeu_ps/pd/si128

ps=float, pd=double, si128=int

SSE intrinsics: different kinds (2)

- Arithmetic intrinsics

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

  - _mm_add_ps/pd/si128
  - _mm_mul_ps/pd/si128
  - _mm_div_ps/pd/si128
  - _mm_max_ps/pd/si128

- Logical operations

```
__m128i _mm_or_si128(__m128i a, __m128i b);
```

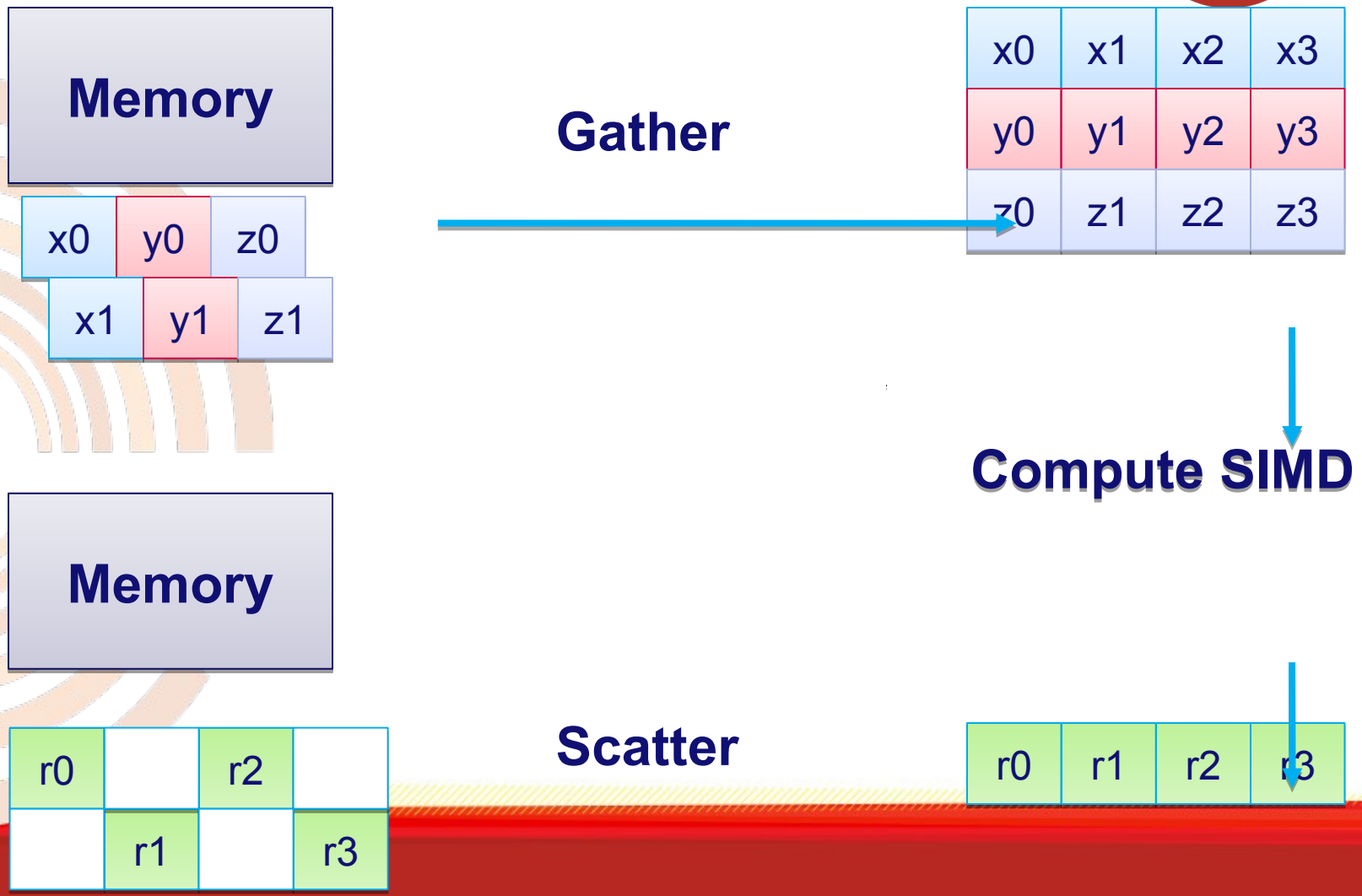
- Type conversion

```
__m128 _mm_cvtepi32_ps(__m128i a);
```

- Details of the intrinsics are on the MSDN website

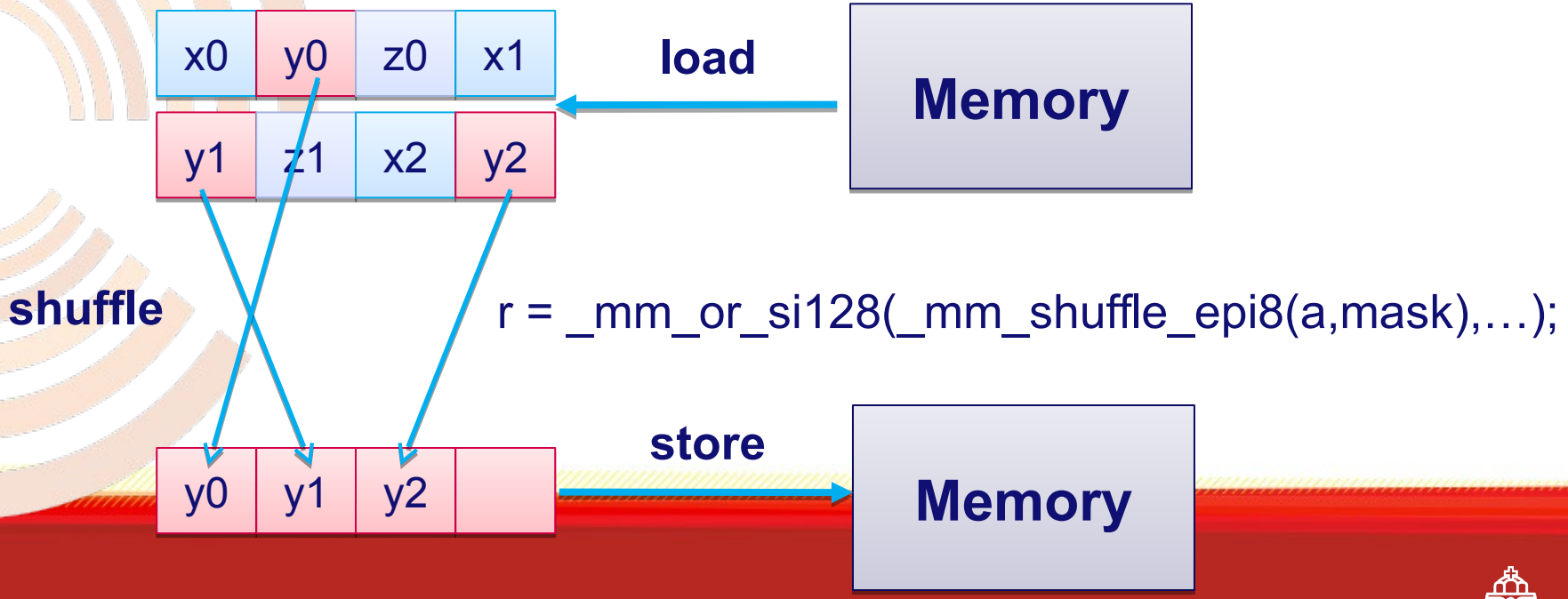
http://msdn.microsoft.com/en-us/library/y0dh78ez(v=vs.100).aspx

Vectorize a practical application



Gather / Scatter

- Single most difficult part of using SSE instructions
  - Key for good performance
  - Can be avoided with the set intrinsic, but slow
- The vector arithmetic operations are not that complex



Remove blue by channel scaling C

```
void removeblue(PPM_IMG input){
    int i;
    int size = input.w*input.h;

    for (i=0; i<size; i++){
        input.img[i*3+0]= (float)input.img[i*3+0]*1.1f;
        input.img[i*3+1]= (float)input.img[i*3+1]*0.7f;
        input.img[i*3+2]= (float)input.img[i*3+2]*0.45f;
    }
}
```

Processing time
187 ms

Remove blue SSE intrinsics

Processing time
62 ms

```
void removeblue(PPM_IMG input){
    int i;
    int size = input.w*input.h;
    unsigned char *img=input.img;

    __m128i din,dout;
    __m128  fin, coef, fout;

    coef = _mm_set_ps(0.0f , 0.45f , 0.7f , 1.1f);

    for (i=0; i<size; i++){
        din= _mm_set_epi32 (0, img[i*3+2],img[i*3+1],img[i*3]);

        fin= _mm_cvtepi32_ps (din);

        fout = _mm_mul_ps(fin , coef );

        dout = _mm_cvtps_epi32 (fout);

        img[i*3] = (unsigned char )_mm_cvtsi128_si32(dout);
        img[i*3+1] = (unsigned char )_mm_cvtsi128_si32 ( _mm_srli_si128(dout, 4) );
        img[i*3+2] = (unsigned char )_mm_cvtsi128_si32 ( _mm_srli_si128(dout, 8) );
    }
}
```

Load coefficients

Gather Data

Compute

Scatter

Coarse estimate
~15 instructions
for 3 pixels

Looking at the assembler output actually 21 instructions

Is this a fair comparison?

```
void removeblue(PPM_IMG input){
    int i;
    int size = input.w*input.h;

    for (i=0; i<size; i++){
        input.img[i*3+0]= (float)input.img[i*3+0]*1.1f;
        input.img[i*3+1]= (float)input.img[i*3+1]*0.7f;
        input.img[i*3+2]= (float)input.img[i*3+2]*0.45f;
    }
}
```

Processing time
187 ms

- So you mean to say we need 3*15 (or even 3*21) instructions for this scalar code?
  - Of course not!
- This experiment was done with an older compiler
  - These compilers are known to have difficulties with handling structs
  - Currently the -O3 generated scalar code is as fast as the vectorized code (using Clang/LLVM)

Optimization reports

- So why didn't it vectorize yet?
- Turn on optimization reports!
  - **-Rpass-missed=vectorize** tells you which loops failed to vectorize
  - **-Rpass-analysis=vectorize** tells you why

```
removeblue.c:13:42: remark: the cost-model indicates that
vectorization is not beneficial [-Rpass-analysis=loop-vectorize]
  input.img[i*3+0] = (float)input.img[i*3+0] * 1.1f;
                                ^
removeblue.c:13:42: remark: the cost-model indicates that interleaving
is not beneficial [-Rpass-analysis=loop-vectorize]
```

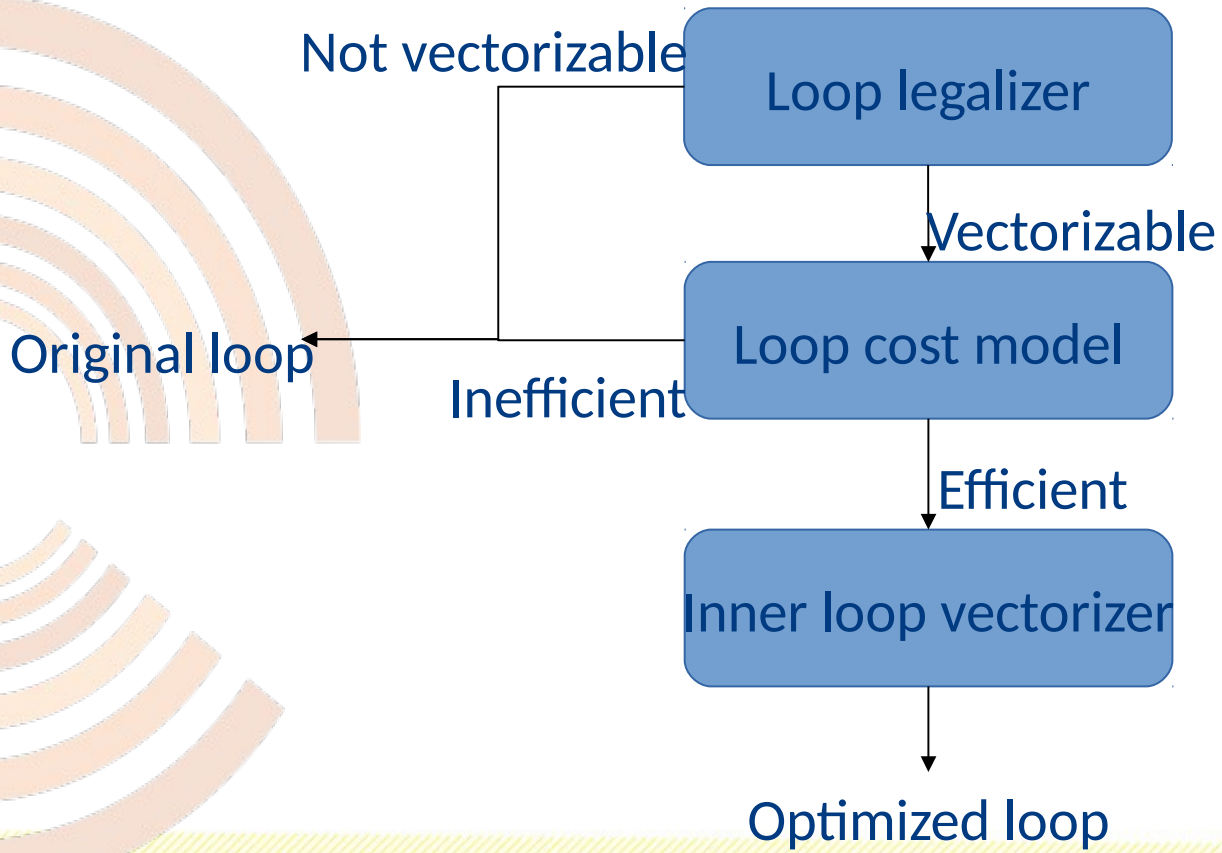
How does LLVM vectorize today?

RADBOUD RADIO LAB

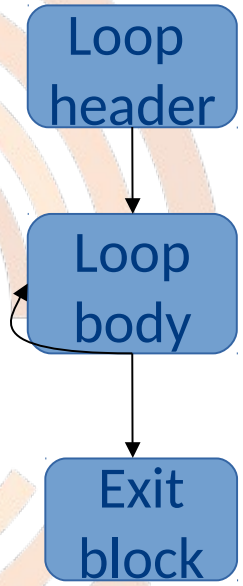
- Study http://llvm.org/docs/Vectorizers.html

- LLVM contains 2 vectorizers:
  - Loop vectorizer
    - checks loops and combines iterations into SIMD instructions
  - SLP vectorizer
    - merges multiple scalar calculations into SIMD instructions

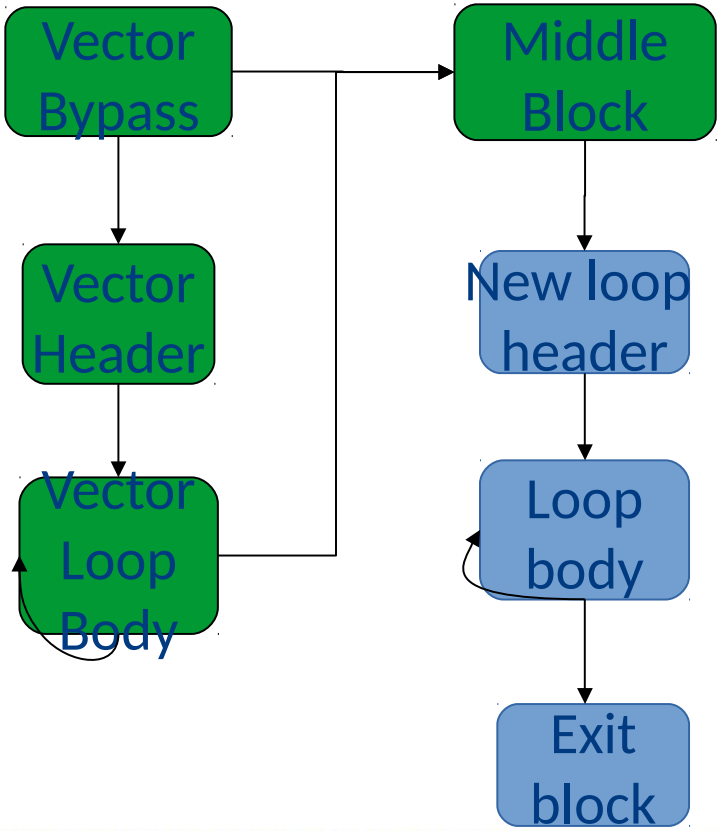
Vectorization steps



Loop vectorization



Scalar loop



Vector version

Unknown tripcount / pointer check

```
void bar(float *A, float* B, float K, int start, int end) {
  for (int i = start; i < end; ++i)
    A[i] *= B[i] + K;
}
```

Issues:

- unknown tripcount can be handled
- A and B may overlap !!!
  - LLVM adds runtime check
  - you could also uses the keyword 'restrict'

Reductions are ok

```
int foo(int *A, int *B, int n) {
  unsigned sum = 0;
  for (int i = 0; i < n; ++i)
    sum += A[i] + 5;
  return sum;
}
```

Function call in body

```
void foo(float *f) {
  for (int i = 0; i != 1024; ++i)
    f[i] = floorf(f[i]);
}
```

Many math function are allowed

Your own function could have 'side-effects'

Scatter – Gather support

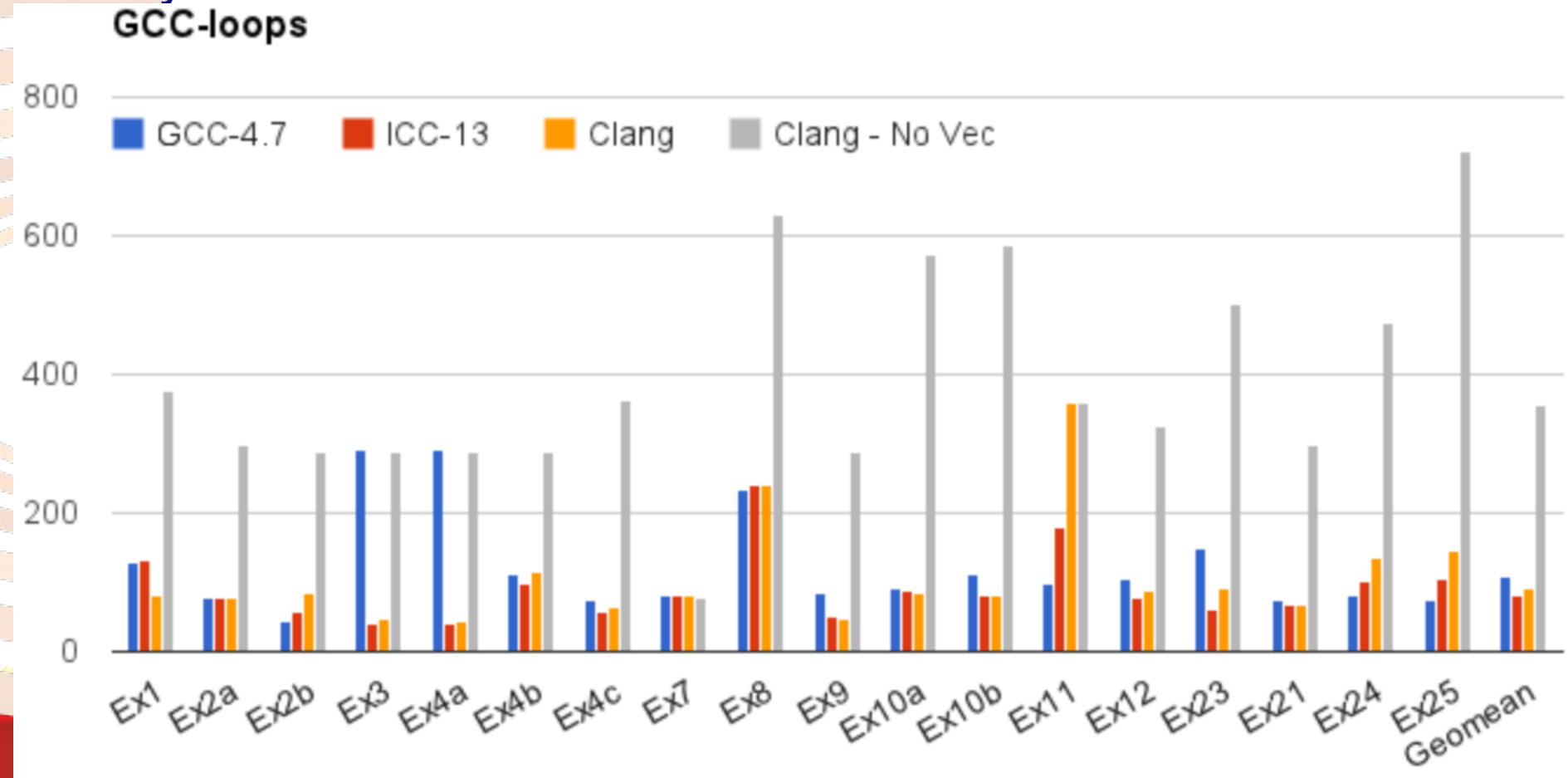
```
int foo(int * A, int * B, int n) {
  for (intptr_t i = 0; i < n; ++i)
    A[i] += B[i * 4];
}
```

- many other loops can be handled
  - reversed iterators
  - mixed vector types in same loop

- How good is LLVM “today”?

Execution time for a number of loops

- target: core i7, sandybridge
- y-axis: time in msec



Recap

- Autovectorization can help a lot
- But may need some help from you
- Many loop structures are OK
  - Scatter / gatter difficult
- If all else fails we can vectorize manually
  - Makes code very target specific

Material

- Check my website for our full set of compiler slides
  - http://www.es.ele.tue.nl/~rjordans/5LIM0.php
- Compiler explorer: https://godbolt.org
- LLVM auto-vectorizers page: http://llvm.org/docs/Vectorizers.html
- Intel Intrinsics overview: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

RADBOUD RADIO LAB

Radboud University