When zero-cost abstraction fails

How to fix your compiler?

Adrien Guinet

adrien@guinet.me
adriengnt

C++Frug / 2018-10-16

Table of Contents

Whoami?

Adrien Guinet

- Work at Quarkslab as Product Manager on an LLVM-based obfuscating compiler
- On my free time, work on open source projects:
  - DragonFFI (https://github.com/aguinet/dragonffi): seamlessly call C functions from Python (using Clang/LLVM)
  - Pythran (https://github.com/serge-sans-paille/pythran): *a claim-less Python to c++ converter*
- Also enjoy' cryptography and reverse engineering!

Contact

adrien@guinet.me
https://github.com/aguinet
adriengnt

The story of a compiler bug

The story

What I'm going to talk about is the story of a compiler bug affecting the performances of Pythran generated code.

The story of a compiler bug

The story

What I'm going to talk about is the story of a compiler bug affecting the performances of Pythran generated code.

It's an excuse to

- Introduce the LLVM IR
- Show how such a bug can be understood and potentially fixed
- Give insight to C++ developers on what's happening in their compiler!

The story of a compiler bug

The story

What I'm going to talk about is the story of a compiler bug affecting the performances of Pythran generated code.

It's an excuse to

- Introduce the LLVM IR
- Show how such a bug can be understood and potentially fixed
- Give insight to C++ developers on what's happening in their compiler!

The ultimate goal

Make you feel comfortable digging into your compiler!

The story of a compiler bug

The story

What I'm going to talk about is the story of a compiler bug affecting the performances of Pythran generated code.

It's an excuse to

- Introduce the LLVM IR
- Show how such a bug can be understood and potentially fixed
- Give insight to C++ developers on what's happening in their compiler!

The ultimate goal

Make you feel comfortable digging into your compiler!

After all, it's just another C++ project :)

Table of Contents

Vector instructions (SIMD)

Definition

SIMD = Single Instruction Multiple Data

- Perform multiple operations within about the same number of cycles as the associated serial instruction.
- Instructions SSE wth 128-bit registers (`XMM*`)
- Instructions `AVX(2)` wth 256-bit registers (`YMM*`)

Vector instructions (SIMD)

Definition

SIMD = Single Instruction Multiple Data

- Perform multiple operations within about the same number of cycles as the associated serial instruction.
- Instructions SSE wth 128-bit registers (`XMM*`)
- Instructions AVX(2) wth 256-bit registers (`YMM*`)

Example: addition of four 32-bit integer (128-bit register)

Vector instructions (SIMD)

Definition

SIMD = Single Instruction Multiple Data

- Perform multiple operations within about the same number of cycles as the associated serial instruction.
- Instructions SSE wth 128-bit registers (XMM$*$)
- Instructions AVX(2) wth 256-bit registers (YMM$*$)

Example: addition of four 32-bit integer (128-bit register)

XMM_a =

a_3	a_2	a_1	a_0
127	95	63	31 — 0

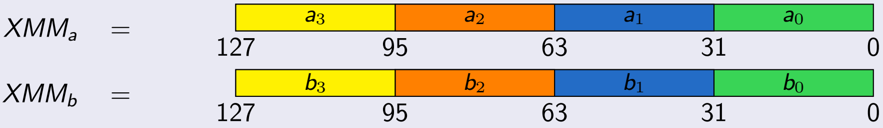
Vector instructions (SIMD)

Definition

SIMD = Single Instruction Multiple Data

- Perform multiple operations within about the same number of cycles as the associated serial instruction.
- Instructions SSE wth 128-bit registers (XMM$*$)
- Instructions AVX(2) wth 256-bit registers (YMM$*$)

Example: addition of four 32-bit integer (128-bit register)



Vector instructions (SIMD)

Definition

SIMD = Single Instruction Multiple Data

- Perform multiple operations within about the same number of cycles as the associated serial instruction.
- Instructions SSE wth 128-bit registers (XMM∗)
- Instructions `AVX(2)` wth 256-bit registers (YMM∗)

Example: addition of four 32-bit integer (128-bit register)

$XMM_a \quad =$

a_3	a_2	a_1	a_0

127 95 63 31 0

$XMM_b \quad =$

b_3	b_2	b_1	b_0

127 95 63 31 0

$XMM_r \quad = \quad add\_epi32(XMM_a, XMM_b)$ (instruction `PADDD`)

Vector instructions (SIMD)

Definition

SIMD = Single Instruction Multiple Data

- Perform multiple operations within about the same number of cycles as the associated serial instruction.
- Instructions SSE wth 128-bit registers (XMM*)
- Instructions AVX(2) wth 256-bit registers (YMM*)

Example: addition of four 32-bit integer (128-bit register)

$XMM_a =$

a_3	a_2	a_1	a_0

127 95 63 31 0

$XMM_b =$

b_3	b_2	b_1	b_0

127 95 63 31 0

$XMM_r = add\_epi32(XMM_a, XMM_b)$ (instruction PADDD)

$=$

$a_3 + b_3$	$a_2 + b_2$	$a_1 + b_1$	$a_0 + b_0$

127 95 63 31 0

LaTeX module to show vectors (c) Serge Guelton.

Table of Contents

Introduction to Pythran

(Subset of) Python to C++ converter

- Compiler that converts a subset of Python to C++
- Generated code uses a C++ backend (*pythonic*) to implement the behavior of some Python modules (like numpy)
- Aimed at scientific Python
- Generally compared to Cython/Pypy and alike

Introduction to Pythran

(Subset of) Python to C++ converter

- Compiler that converts a subset of Python to C++
- Generated code uses a C++ backend (*pythonic*) to implement the behavior of some Python modules (like numpy)
- Aimed at scientific Python
- Generally compared to Cython/Pypy and alike

Example

```python
#pythran export add(uint32[],uint32[])
def add(a, b):
    return a+b
```

Let's benchmark this versus a hand-written C loop!

Discovering the bug

Benchmarking the array addition

For 200000 elements, using Clang 4.0 and AVX2:

- Pure C: 59.6 us (+/- 28 us)
- Pythran: 126 us (+/- 23.2 us)

Discovering the bug

Benchmarking the array addition

For 200000 elements, using Clang 4.0 and AVX2:

- Pure C: 59.6 us (+/- 28 us)
- Pythran: 126 us (+/- 23.2 us)

Where's the difference?

- Clang verbose mode for vectorization:
  - `-Rpass=loop-vectorize`: show loops that have been vectorized
  - `-Rpass-missed=loop-vectorize`: show loops that haven't been vectorized
  - `-Rpass-analysis=loop-vectorize`: show why they haven't been vectorized
- Look at the generated ASM code using IDA [a]

[a] https://www.hex-rays.com/products/ida/support/download_freeware.shtml

Fixing the issue

Solutions to vectorize the code

- Write intrinsic-based/asm code
  - Tedious work (support every operations, ...)
  - Not portable across architectures

Fixing the issue

Solutions to vectorize the code

- Write intrinsic-based/asm code
  - Tedious work (support every operations, ...)
  - Not portable across architectures
- Use a high-level C++ library
  - Boost.SIMD (NumScale)
  - xSIMD (QuantStack)
  - Pythran already does this (demo)

Fixing the issue

Solutions to vectorize the code

- Write intrinsic-based/asm code
  - Tedious work (support every operations, ...)
  - Not portable across architectures
- Use a high-level C++ library
  - Boost.SIMD (NumScale)
  - xSIMD (QuantStack)
  - Pythran already does this (demo)

What about fixing it for everyone else?

Fixing the issue

Solutions to vectorize the code

- Write intrinsic-based/asm code
  - Tedious work (support every operations, ...)
  - Not portable across architectures
- Use a high-level C++ library
  - Boost.SIMD (NumScale)
  - xSIMD (QuantStack)
  - Pythran already does this (demo)

What about fixing it for everyone else?

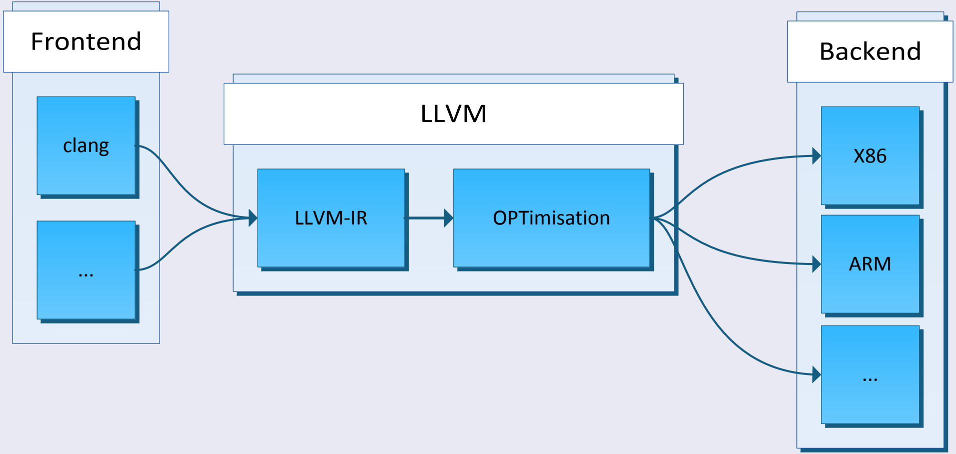
Fix the compiler!

- Reduce the test case
- Figure out what's happening
- Try to fix and report it

Table of Contents

Clang/LLVM: compilation flow

Classical compilation flow



LLVM IR introduction

Some definitions

- Kind of structured typed assembly
- Module
  - Global variables
  - Functions $\Rightarrow$ Basic blocks $\Rightarrow$ Instructions
- Single Static Assignment (SSA)
- Can be serialized/deserialized into/from a textual form

LLVM IR introduction

Some definitions

- Kind of structured typed assembly
- Module
  - Global variables
  - Functions ⇒ Basic blocks ⇒ Instructions
- Single Static Assignment (SSA)
- Can be serialized/deserialized into/from a textual form

Remarks

- C ABI is partially resolved
- **Not** a portable virtual machine
- Clang generates LLVM IR code according to the target architecture/OS
- More on all of that later...

LLVM IR: examples

C code

```
#include <stdint.h>
uint16_t add(uint16_t a, uint16_t b) {
  return a+b;
}
```

LLVM IR for amd64/Linux with -O1

```
$ clang-6.0 -S -emit-llvm -O1 -o - add.c
source_filename = "add.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

define zeroext i16 @add(i16 zeroext %a, i16 zeroext %b) {
  %1 = add i16 %b, %a
  ret i16 %1
}
```

LLVM IR: examples

C code

```c
#include <stdint.h>
uint16_t add(uint16_t a, uint16_t b) {
  return a+b;
}
```

LLVM IR for amd64/Linux w/o opt

```llvm
define zeroext i16 @add(i16 zeroext %a, i16 zeroext %b) {
  %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, i16* %1, align 2
  %4 = zext i16 %3 to i32
  %5 = load i16, i16* %2, align 2
  %6 = zext i16 %5 to i32
  %7 = add nsw i32 %4, %6
  %8 = trunc i32 %7 to i16
  ret i16 %8
}
```

LLVM IR: optimisations

Applying passes one by one

- `opt` applies a list of pass on the LLVM IR
- Useful to debug / understand passes

Non-optimized IR

```
define zeroext i16 @add(i16 zeroext %a, i16 ↩
    zeroext %b) {
  %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, i16* %1, align 2
  %4 = zext i16 %3 to i32
  %5 = load i16, i16* %2, align 2
  %6 = zext i16 %5 to i32
  %7 = add nsw i32 %4, %6
  %8 = trunc i32 %7 to i16
  ret i16 %8
}
```

`opt -mem2reg -S`

```
define zeroext i16 @add(i16 zeroext %a, i16 ↩
    zeroext %b) #0 {
  %1 = zext i16 %a to i32
  %2 = zext i16 %b to i32
  %3 = add nsw i32 %1, %2
  %4 = trunc i32 %3 to i16
  ret i16 %4
}
```

LLVM IR: optimisations

Applying passes one by one

- `opt` applies a list of pass on the LLVM IR
- Useful to debug / understand passes

Non-optimized IR

```
define zeroext i16 @add(i16 zeroext %a, i16 ↩
    zeroext %b) {
  %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, i16* %1, align 2
  %4 = zext i16 %3 to i32
  %5 = load i16, i16* %2, align 2
  %6 = zext i16 %5 to i32
  %7 = add nsw i32 %4, %6
  %8 = trunc i32 %7 to i16
  ret i16 %8
}
```

`opt -mem2reg -S`

```
define zeroext i16 @add(i16 zeroext %a, i16 ↩
    zeroext %b) #0 {
  %1 = zext i16 %a to i32
  %2 = zext i16 %b to i32
  %3 = add nsw i32 %1, %2
  %4 = trunc i32 %3 to i16
  ret i16 %4
}
```

`opt -mem2reg -instcombine -S`

```
define zeroext i16 @add(i16 zeroext %a, i16 ↩
    zeroext %b) #0 {
  %1 = add i16 %a, %b
  ret i16 %1
}
```

LLVM IR: control flow and vectorization

Demo

- Simple control flow
- Loop w/o vectorization
- Loop with vectorization
- Examples of non-portability

Table of Contents

Minimal reproducer

Find a minimal C++ file with the bug

- Hints given by `-Rpass-analysis=loop-vectorize`
- Start from there and understand what's going on
- Other hint: Pythran uses expression templates

Reduced test case

https://godbolt.org/z/val3Xm

Gathering clues

Diffing LLVM IRs

- We know the C version is vectorized
- Let's compare the two non-vectorized version of the LLVM IR
- It might help us understand where something fails

https://godbolt.org/z/VV6S-h

Gathering clues

Diffing LLVM IRs

- We know the C version is vectorized
- Let's compare the two non-vectorized version of the LLVM IR
- It might help us understand where something fails

https://godbolt.org/z/VV6S-h

Have a little talk with the vectorizer (notebook)

```
$ clang++-4.0 -O2 -std=c++12 op_zip_iterator.cpp -Rpass-missed=loop-vectorize -Rpass-analysis=loop-↩
    vectorize -c -o /dev/null
/usr/include/c++/8/bits/stl_algobase.h:322:4: remark: loop not vectorized:  value that could not be
    identified as reduction is used outside the loop [-Rpass-analysis=loop-vectorize]
        for(_Distance __n = __last - __first; __n > 0; --__n)
```

Checking the LLVM source code

rgrep 'reduction is used outside the loop' llvm/lib

```
// For each block in the loop.
for (BasicBlock *BB : TheLoop->blocks()) {
  // Scan the instructions in the block and look for hazards.
  for (Instruction &I : *BB) {
    if (auto *Phi = dyn_cast<PHINode>(&I)) {
      // [...]
      InductionDescriptor ID;
      if (InductionDescriptor::isInductionPHI(Phi, TheLoop, PSE, ID)) {
        addInductionPhi(Phi, ID, AllowedExit);
        // [...]
        continue;
      }
      // [...]
      // As a last resort, coerce the PHI to a AddRec expression
      // and re-try classifying it a an induction PHI.
      if (InductionDescriptor::isInductionPHI(Phi, TheLoop, PSE, ID, true)) {
        addInductionPhi(Phi, ID, AllowedExit);
        continue;
      }
      ORE->emit(createMissedAnalysis("NonReductionValueUsedOutsideLoop", Phi)
                << "value that could not be identified as "
                   "reduction is used outside the loop");
      DEBUG(dbgs() << "LV: Found an unidentified PHI." << *Phi << "\n");
      return false;
```

Getting the problematic PHI node

LLVM in debug mode

- The `DEBUG` macro can give us the PHI node that seems to cause trouble
- `DEBUG` macros are deactivated in release builds
- We need to compile LLVM in debug mode and print debug informations

Getting the problematic PHI node

LLVM in debug mode

- The `DEBUG` macro can give us the PHI node that seems to cause trouble
- `DEBUG` macros are deactivated in release builds
- We need to compile LLVM in debug mode and print debug informations

Compilation

- Lots of information here: https://llvm.org/docs/CMake.html
- Download LLVM and Clang 4.0 sources from http://releases.llvm.org, then:
- `cmake -DCMAKE_BUILD_TYPE=Debug -DBUILD_SHARED_LIBS=ON -DLLVM_OPTIMIZED_TABLEGEN=ON -G ninja .. && ninja`
- Grab a coffee, go for a run, this can take some time depending on your hardware (~20min on mine)

Getting the problematic PHI node

Show debug informations

```
$ /path/to/build/bin/clang++ -mllvm -debug -mllvm -debug-only=loop-vectorize
     -Rpass-analysis=vectorize -O2  code/op_zip_iterator.cpp -Xclang -discard-value-names -c -o /dev/↩
     null  -std=c++11
LV: Checking a loop in "_Z2opPj16add_zip_iteratorS0_" from /usr/lib/gcc/x86_64-linux-gnu/8/../../../../↩
     include/c++/8/bits/stl_algobase.h:322:4
LV: Loop hints: force=? width=0 unroll=0
LV: Found a loop:  for.body.i.i.i.i
LV: Found an induction variable.
LV: Found an induction variable.
LV: Found an induction variable.
In file included from /home/aguinet/confs/18-10-16-cppfrug-llvm/code/op_zip_iterator.cpp:1:
In file included from /usr/lib/gcc/x86_64-linux-gnu/8/../../../../include/c++/8/algorithm:61:
/usr/lib/gcc/x86_64-linux-gnu/8/../../../../include/c++/8/bits/stl_algobase.h:322:4: remark: loop not ↩
     vectorized: value that could not be identified as reduction is used outside the loop
      [-Rpass-analysis=loop-vectorize]
          for(_Distance __n = __last - __first; __n > 0; --__n)
          ^
LV: Found an unidentified PHI. %16 = phi i64 [ %22, %12 ], [ %6, %10 ]
LV: Can't vectorize the instructions or CFG
LV: Not vectorizing: Cannot prove legality.
```

Putting the pieces together

```
define void @op_distance(i32* %res, i32* %it.a, i32* %it.b, i32* %it_end.a, i32* ↩
      %it_end.b) {
  %it.a_int = ptrtoint i32* %it.a to i64
  %it_end.a_int = ptrtoint i32* %it_end.a to i64
  %8 = sub i64 %it.a_int, %it_end.a_int
  %9 = icmp sgt i64 %8, 0
  br i1 %9, label %loop_header, label %ret

; <label>:loop_header:                                ; preds = %loop, %entry
  %count = lshr exact i64 %8, 2
  br label %loop

; <label>:loop:                                       ; preds = %loop, %loop_header
  %cur_count = phi i64 [ %next_count, %loop ], [ %count, %loop_header ]
  %cur_res = phi i32* [ %next_res, %loop ], [ %res, %loop_header ]
  %cur_it.b = phi i32* [ %next_it.b, %loop ], [ %it.b, %loop_header ]
  %cur_it.a_int = phi i64 [ %next_it.a_int, %loop ], [ %it.a_int, %loop_header ]
  %17 = inttoptr i64 %16 to i32*
  %18 = load i32, i32* %17, align 4, !tbaa !1
  %19 = load i32, i32* %15, align 4, !tbaa !1
  %20 = add i32 %19, %18
  store i32 %20, i32* %14, align 4, !tbaa !1
  %21 = getelementptr inbounds i32, i32* %17, i64 1
  %next_it.a_int = ptrtoint i32* %21 to i64
  %next_it.b = getelementptr inbounds i32, i32* %15, i64 1
  %next_res = getelementptr inbounds i32, i32* %14, i64 1
  %next_count = add nsw i64 %cur_count, -1
  %26 = icmp sgt i64 %cur_count, 1
  br i1 %26, label %loop, label %loop_end
; <label>:loop_end:                                       ; preds = %loop
  br label %ret

; <label>:ret:                                        ; preds = %loop_end, %entry
  ret void
}
```

What's happening here?

- The loop vectorizer checks that everything in the loop can be vectorized
- It relies on an analysis that finds induction variables
- PHI node `%cur_it.a_int` isn't considered as one (and should be)
- Because of the `inttoptr` <=> `ptrtoint` round-trip?

Further analyses

The "Scalar evolution" analysis in LLVM

- In one sentence: "Change in the Value of Scalar Variables Over Iterations of the Loop"
- Sentence from this nice talk about it in EuroLLVM 2018: https://www.youtube.com/watch?v=AmjliNp0_00
- Used by the "is an induction variable?" analysis

Further analyses

The "Scalar evolution" analysis in LLVM

- In one sentence: "Change in the Value of Scalar Variables Over Iterations of the Loop"
- Sentence from this nice talk about it in EuroLLVM 2018: https://www.youtube.com/watch?v=AmjliNp0_00
- Used by the "is an induction variable?" analysis

In its source code

```
// It's tempting to handle inttoptr and ptrtoint as no-ops, however this can
// lead to pointer expressions which cannot safely be expanded to GEPs,
// because ScalarEvolution doesn't respect the GEP aliasing rules when
// simplifying integer expressions.
```

- Looks like these `inttoptr <=> ptrtoint` conversions are the problem!

One "fix"

Remove the `inttoptr <=> ptrtoint` round-trip

- Out-of-tree pass that does this: https://github.com/aguinet/llvm-intptrcleanup
- Not sure this is semantically valid, but it will show that our theories are valid

One "fix"

Remove the `inttoptr <=> ptrtoint` round-trip

- Out-of-tree pass that does this: https://github.com/aguinet/llvm-intptrcleanup
- Not sure this is semantically valid, but it will show that our theories are valid

Result

- Pass is registered before vectorization (in the pipeline)
- `inttoptr <=> ptrtoint` round-trips are removed
- Vectorisation happens!

http://localhost:10240/z/aWxbpR

Going further

Which pass creates this?

- Using clang in debug mode, we see

```
__first.sroa.0.012.i.i.i.i = phi i64 [%5, %for.body.i.i.i.i], [%0, %for.body.preheader.i.i.i.i]
```

- SROA = Scalar Replacement of Aggregates

Scalar Replacement of Aggregates

- "This transform breaks up alloca instructions of aggregate type (structure or array) into individual alloca instructions for each member if possible. Then, if possible, it transforms the individual alloca instructions into nice clean scalar SSA form." [a]

[a] https://llvm.org/docs/Passes.html#sroa-scalar-replacement-of-aggregates

The proper fix

- Properly fixing probably means fixing SROA

Reporting the bug

Bug reporting

- LLVM bug tracker: https://bugs.llvm.org/. Registration is closed, an email needs to be sent to get an account.
- `llvm-dev` mailing list: to ask for help on how to really fix the issue. People are generally reactive!

Related links

- Bug tracker: https://bugs.llvm.org/show_bug.cgi?id=33532
- Mailing-list: http://lists.llvm.org/pipermail/llvm-dev/2017-June/114300.html

Table of Contents

Conclusion

Did we fix the Pythran code vectorization?

- Let's try it...

Conclusion

Did we fix the Pythran code vectorization?

- Let's try it...
- ...somehow we fixed our test case, but not the whole Pythran case
- clang 7 still fails at this!
- So we need to re-do this again :)

Conclusion

Did we fix the Pythran code vectorization?

- Let's try it...
- ...somehow we fixed our test case, but not the whole Pythran case
- clang 7 still fails at this!
- So we need to re-do this again :)

What did we learn? (I hope)

- Compilers have lots of ways to debug their analyses
- How to dig into them to understand what's happening
- Compiling communities are receptives to this kind of issues

Discussion

Zero-cost abstractions?

- Zero-cost abstractions are often presented as an advantage of C++
- But **nothing guarantees** it in the **standard**
- It works because we have good **optimising compilers**!

Discussion

Zero-cost abstractions?

- Zero-cost abstractions are often presented as an advantage of C++
- But **nothing guarantees** it in the **standard**
- It works because we have good **optimising compilers**!

Issue

When performance is a user contract, there's no way **standard** way to guarantee many of it!

Guaranteed optimisations

Guaranteed optimisations?

- `pragmas` to trigger an error if an optimisation fails?
- User-driver optimisation flow?
- Level of optimisations guaranteed by the C++ standard?

Guaranteed optimisations

Guaranteed optimisations?

- `pragmas` to trigger an error if an optimisation fails?
- User-driver optimisation flow?
- Level of optimisations guaranteed by the C++ standard?

Ideas

- Use `FileCheck`
  - Check the generated assembly (problem: this is platform-dependant)
  - Check the generated LLVM IR (still platform-dependant, but more generic)
  - Demo!
- C backend for the LLVM IR for an easier-to-read representation?

Table of Contents

How does this compile? (Linux/x86-64)

XOR two buffer of 16 bytes

```
void xorBlocks(uint8_t* Out, uint8_t const* In) {
  for (size_t I = 0; I < 16; ++I) {
    Out[I] ^= In[I];
  }
}
```

https://godbolt.org/z/MpmttL

Questions?

Thanks for your attention!

Questions?

Acknowledgment

Serge Guelton for the Pythran project and the discussions regarding this talk!

Contact

adrien@guinet.me
https://github.com/aguinet
adriengnt