Andrea Arteaga

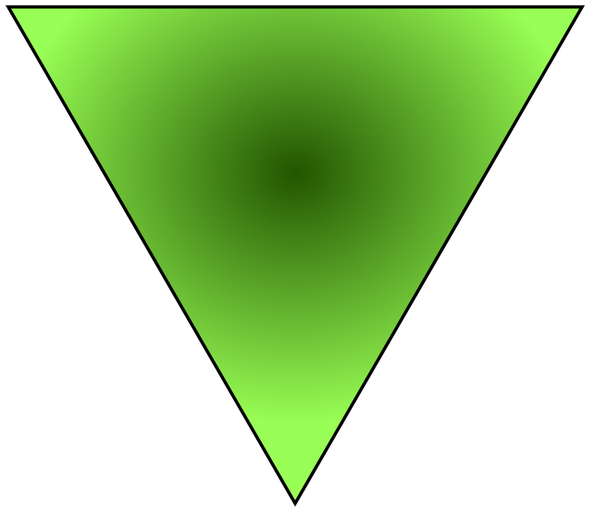
Bit-Reproducibility in HPC Applications

About Me

BSc and MSc in Computational Science and Engineering at ETH (2008-2013)

About Me

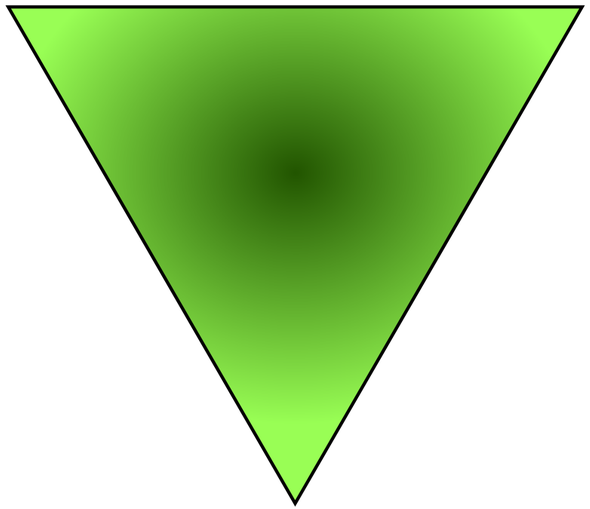
BSc and MSc in Computational Science and Engineering at ETH (2008-2013)



About Me

BSc and MSc in Computational Science and Engineering at ETH (2008-2013)

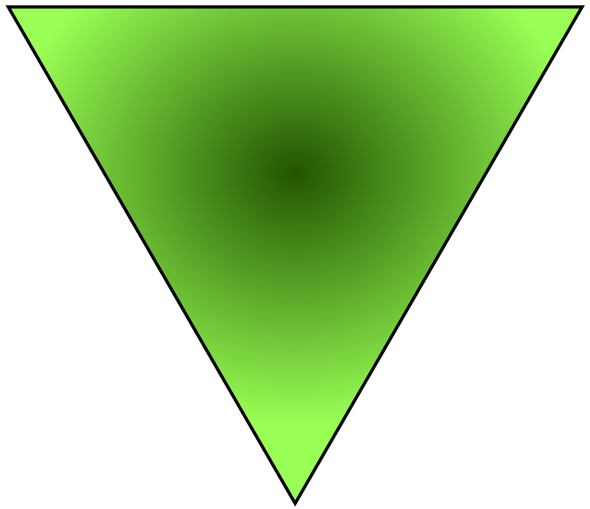
Oliver Fuhrer, MeteoSwiss



About Me

BSc and MSc in Computational Science and Engineering at ETH (2008-2013)

Oliver Fuhrer, MeteoSwiss



Torsten Höfler, ETH

About Me

BSc and MSc in Computational Science and Engineering at ETH (2008-2013)

Oliver Fuhrer, MeteoSwiss

Torsten Höfler, ETH

Piz Daint, CSCS

Floating-Point Algebra

Floating-Point Algebra

Floating-Point Operations

$$
\begin{aligned}
a \oplus b &:= rd(a + b) \\
a \ominus b &:= rd(a - b) \\
a \otimes b &:= rd(a \times b) \\
a \oslash b &:= rd(a \div b)
\end{aligned}
$$

$$
sqrt(a) := rd(\sqrt{a})
$$

Floating-Point Algebra

```
      a   :=   0.11100010100
      b   :=   0.11100111011
      c   :=   0.01100000101
-------------------------------
  exact: 10.001010100011
rounded: 10.001010100
```

Floating-Point Algebra

```
    a   :=   0.11100010100
    b   :=   0.11100111011
    c   :=   0.01100000101
-------------------------------
exact:  10.001010100011
rounded: 10.001010100
```

(a ⊕ b) ⊕ c

```
a + b :=   1.11001001111
rounded:   1.1100101000

  + c :=  10.001010100101
rounded:  10.001010101
```

Floating-Point Algebra

```
a   :=   0.11100010100
b   :=   0.11100111011
c   :=   0.01100000101
```

(a ⊕ b) ⊕ c

a ⊕ (b ⊕ c)

```
exact: 10.001010100011
rounded: 10.001010100
```

```
a + b :=   1.11001001111
rounded:   1.1100101000

  + c := 10.001010100101
rounded: 10.001010101
```

```
b + c :=   1.01000111101 1
rounded:   1.0100011111

  + a := 10.00101010010
rounded: 10.001010100
```

Questions

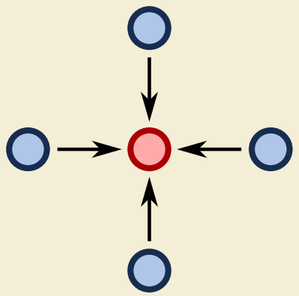
1. How to implement bit-reproducible scalar functions?
2. How to deterministicly reduce large arrays of numbers?

Questions

1. How to implement bit-reproducible scalar functions?
2. How to deterministicly reduce large arrays of numbers?

Stencil functions

$$b_{i,j} = -4 \cdot a_{i,j} + a_{i-1,j} + a_{i+1,j} + a_{i,j-1} + a_{i,j+1}$$



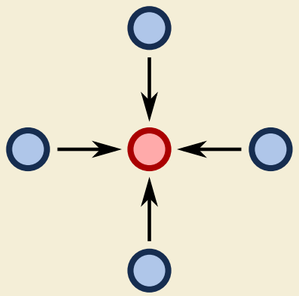
```
for i = 1..N-1
  for j = 1..N-1
    b[i,j] = f(a, i, j)
  endfor
endfor
```

Questions

1. How to implement bit-reproducible scalar functions?
2. How to deterministicly reduce large arrays of numbers?

Stencil functions

$$b_{i,j} = -4 \cdot a_{i,j} + a_{i-1,j} + a_{i+1,j} + a_{i,j-1} + a_{i,j+1}$$



```
for i = 1..N-1
  for j = 1..N-1
    b[i,j] = f(a, i, j)
  endfor
endfor
```

Reproducible summation

$$b = \sum_{i=0}^{N-1} a_i$$

```
float a[N];
float b_local = sum(a, N), b_global;
MPI_Reduce(b_local, b_global, 1, ...)
```

Reproducible Scalar Functions

$$b_{i,j} = -4 \cdot a_{i,j} + a_{i-1,j} + a_{i+1,j} + a_{i,j-1} + a_{i,j+1}$$

$$\theta_{i,j} = T_{i,j} \cdot \exp\left(-rocvp \cdot \log(p_{i,j} \cdot 10^{-5})\right)$$

Reproducible Scalar Functions

$$b_{i,j} = -4 \cdot a_{i,j} + a_{i-1,j} + a_{i+1,j} + a_{i,j-1} + a_{i,j+1}$$

$$b_{i,j} = (((-4 \cdot a_{i,j} + a_{i-1,j}) + a_{i+1,j}) + a_{i,j-1}) + a_{i,j+1}$$

Impose a fixed execution order (e.g. bracketing)

$$\theta_{i,j} = T_{i,j} \cdot \exp\left(-rocvp \cdot \log(p_{i,j} \cdot 10^{-5})\right)$$

Reproducible Scalar Functions

$$b_{i,j} = -4 \cdot a_{i,j} + a_{i-1,j} + a_{i+1,j} + a_{i,j-1} + a_{i,j+1}$$

$$b_{i,j} = (((-4 \cdot a_{i,j} + a_{i-1,j}) + a_{i+1,j}) + a_{i,j-1}) + a_{i,j+1}$$

Impose a fixed execution order (e.g. bracketing)

$$\theta_{i,j} = T_{i,j} \cdot \exp\left(-rocvp \cdot \log(p_{i,j} \cdot 10^{-5})\right)$$

Reproducible Scalar Functions

$$b_{i,j} = -4 \cdot a_{i,j} + a_{i-1,j} + a_{i+1,j} + a_{i,j-1} + a_{i,j+1}$$

$$b_{i,j} = (((-4 \cdot a_{i,j} + a_{i-1,j}) + a_{i+1,j}) + a_{i,j-1}) + a_{i,j+1}$$

Impose a fixed execution order (e.g. bracketing)

Fix the instruction set (e.g. FMA)

$$\theta_{i,j} = T_{i,j} \cdot \exp\left(-rocvp \cdot \log(p_{i,j} \cdot 10^{-5})\right)$$

Reproducible Scalar Functions

$$b_{i,j} = -4 \cdot a_{i,j} + a_{i-1,j} + a_{i+1,j} + a_{i,j-1} + a_{i,j+1}$$

$$b_{i,j} = (((-4 \cdot a_{i,j} + a_{i-1,j}) + a_{i+1,j}) + a_{i,j-1}) + a_{i,j+1}$$

Impose a fixed execution order (e.g. bracketing)

Fix the instruction set (e.g. FMA)

$$\theta_{i,j} = T_{i,j} \cdot \exp\left(-rocvp \cdot \log(p_{i,j} \cdot 10^{-5})\right)$$

Use reproducible transcendntal functions

Reproducible Summation: Error-Free Transform

```
a  :=    10.110110011
```

Value to extract

Reproducible Summation: Error-Free Transform

Extractor

```
a  :=    10.110110011
M  := 1000.0000000
```

Value to extract

Reproducible Summation: Error-Free Transform

Extractor

```
a  :=    10.110110011
M  := 1000.0000000

a+M   = 1010.110110011
      = 1010.1101101
```

Value to extract

Reproducible Summation: Error-Free Transform

Extractor

```
a  :=    10.110110011
M  := 1000.0000000

a+M   = 1010.110110011
      = 1010.1101101

(a+M)-M =   10.1101101
```

Value to extract

Rounded value

Reproducible Summation: Error-Free Transform

Extractor

Value to extract

```
a  :=    10.110110011
M  := 1000.0000000

a+M   = 1010.110110011
      = 1010.1101101

(a+M)-M   =    10.1101101

a-((a+M)-M) =   - 0.000000001
```

Rounded value

Remainder

Reproducible Summation: Algorithm

```
a[0]        10.110110011
a[1]           .10010010101
```

Reproducible Summation: Algorithm

```
M        10000.0000000


a[0]          10.110110011

a[1]             .10010010101
```

Reproducible Summation: Algorithm

```
M        10000.0000000

a[0]       10.11011001 1

a[1]         .10010010101
```

Reproducible Summation: Algorithm

```
M         10000.0000000

a[0]        10.11011011      →      10.1101101  -  0.000000001
a[1]          .10010010101   →        .1001001  +  0.0000000101
```

Reproducible Summation: Algorithm

```
M          10000.0000000

a[0]         10.1101100 11      →        10.1101101   -   0.000000001
a[1]            .1001001 0101   →           .1001001  +   0.00000000101
a[2]          1.1100101 1       →         1.1100110   -   0.00000001
```

Reproducible Summation: Algorithm

```
M          10000.0000000

a[0]         10.1101100 11      →        10.1101101   -   0.000000001
a[1]            .1001001 0101   →           .1001001   +   0.00000000101
a[2]          1.1100101 1       →         1.1100110    -   0.00000001
a[3]        110.1110110 1       →       110.1110110    +   0.00000001
```

Reproducible Summation: Algorithm

```
M          10000.0000000

a[0]         10.1101100 11      →      10.1101101   -   0.000000001
a[1]            .1001001 0101   →         .1001001  +   0.00000000101
a[2]          1.1100101 1       →       1.1100110   -   0.00000001
a[3]        110.1110110 1       →     110.1110110   +   0.00000001
                                  ---------------------------------
                                    1100.0010010   +   0.00000000001
```

Reproducible Summation: Algorithm

```
M          10000.0000000

a[0]         10.1101100 11      →      10.1101101   -   0.000000001
a[1]            .1001001 0101   →         .1001001  +   0.00000000101
a[2]          1.1100101 1       →       1.1100110   -   0.00000001
a[3]        110.1110110 1       →     110.1110110   +   0.00000001
```

```
1100.0010010   +   0.00000000001
```

First-level result

Second-level result

Reproducible Summation: Performance

	Memory	Flops	Communication	Values per communication
Traditional	N · sizeof(Type)	N	log(p)	1
Reproducible				

Reproducible Summation: Performance

	Memory	Flops	Communication	Values per communication
Traditional	N · sizeof(Type)	N	log(p)	1
Reproducible	N · sizeof(Type)			

Reproducible Summation: Performance

	Memory	Flops	Communication	Values per communication
Traditional	N · sizeof(Type)	N	log(p)	1
Reproducible	N · sizeof(Type)	16 · N		

Reproducible Summation: Performance

	Memory	Flops	Communication	Values per communication
Traditional	N · sizeof(Type)	N	log(p)	1
Reproducible	N · sizeof(Type)	16 · N	log(p)	

Reproducible Summation: Performance

	Memory	Flops	Communication	Values per communication
Traditional	N · sizeof(Type)	N	log(p)	1
Reproducible	N · sizeof(Type)	16 · N	log(p)	3 - 4

Reproducible Summation: Performance

	Memory	Flops	Communication	Values per communication
Traditional	N · sizeof(Type)	N	log(p)	1
Reproducible	N · sizeof(Type)	16 · N	log(p)	3 - 4

Memory-bound case: Good performance

Computation-bound case: Bad performance

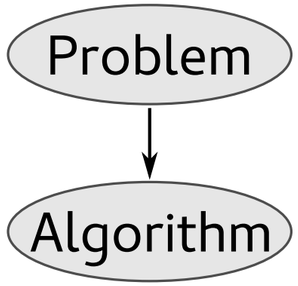
Communication-latency-bound case: Good performance

Thesis Project

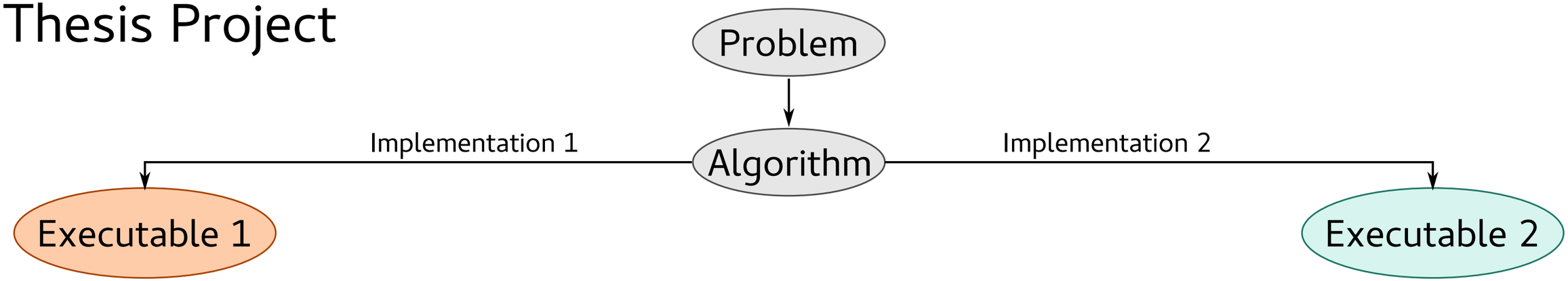
Thesis Project

Problem

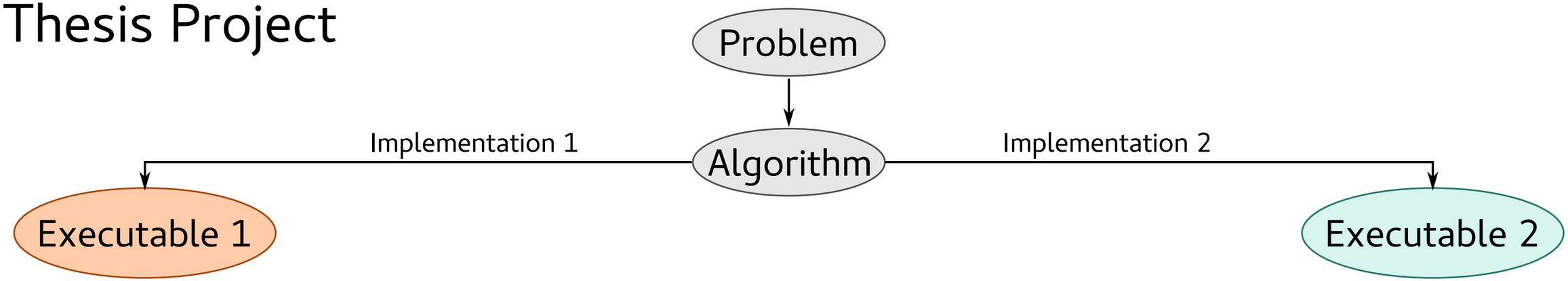
Thesis Project



Thesis Project



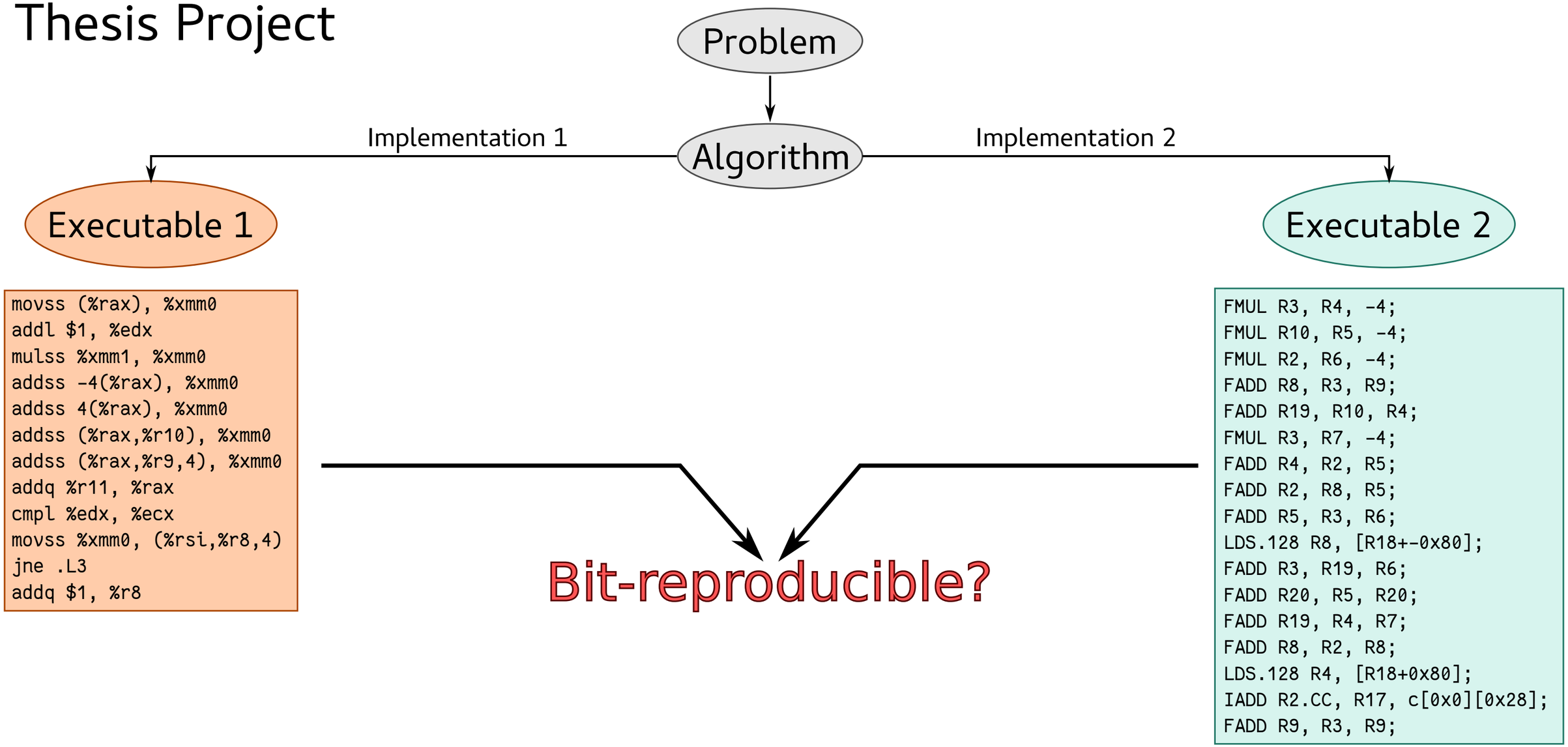
Thesis Project



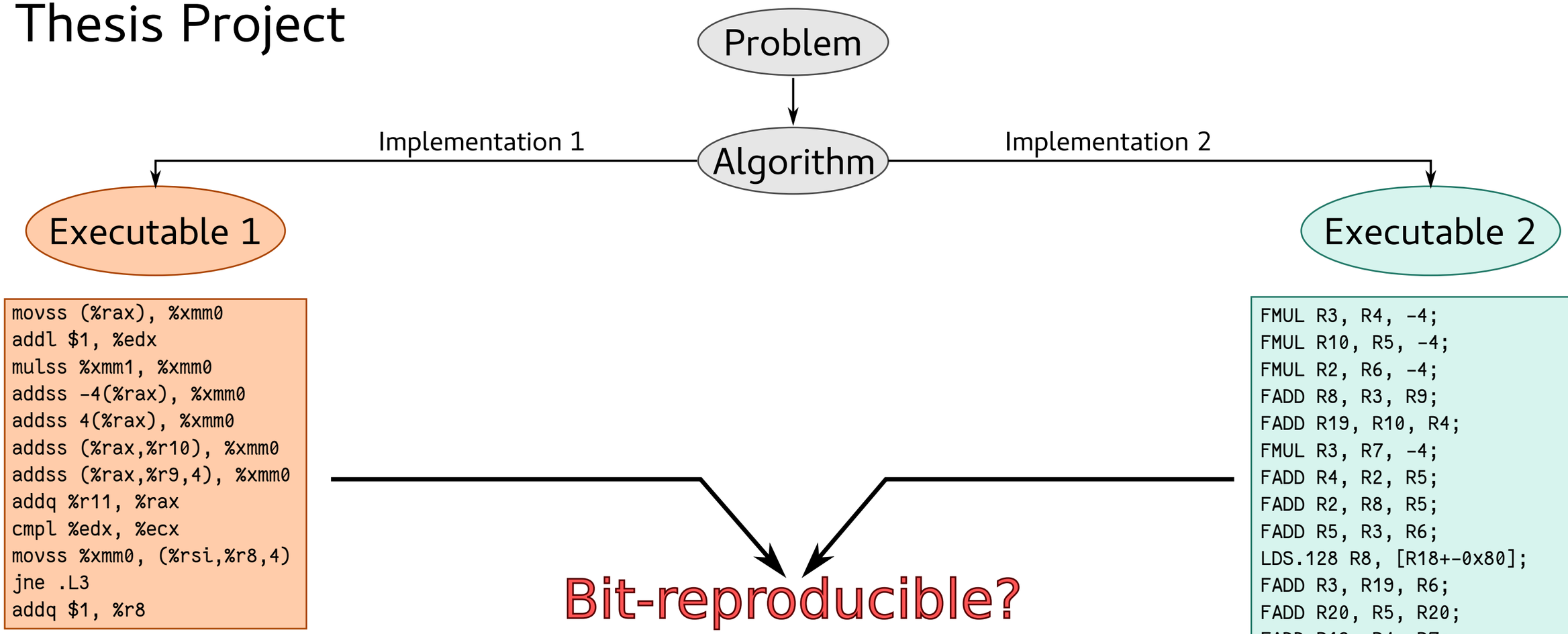

```
movss (%rax), %xmm0
addl $1, %edx
mulss %xmm1, %xmm0
addss -4(%rax), %xmm0
addss 4(%rax), %xmm0
addss (%rax,%r10), %xmm0
addss (%rax,%r9,4), %xmm0
addq %r11, %rax
cmpl %edx, %ecx
movss %xmm0, (%rsi,%r8,4)
jne .L3
addq $1, %r8
```

```
FMUL R3, R4, -4;
FMUL R10, R5, -4;
FMUL R2, R6, -4;
FADD R8, R3, R9;
FADD R19, R10, R4;
FMUL R3, R7, -4;
FADD R4, R2, R5;
FADD R2, R8, R5;
FADD R5, R3, R6;
LDS.128 R8, [R18+-0x80];
FADD R3, R19, R6;
FADD R20, R5, R20;
FADD R19, R4, R7;
FADD R8, R2, R8;
LDS.128 R4, [R18+0x80];
IADD R2.CC, R17, c[0x0][0x28];
FADD R9, R3, R9;
```

Thesis Project



Thesis Project



Contact: andrea.arteaga@env.ethz.ch