The JIL library (Justinline) for computations with SLPs

Joris van der Hoeven

Joint work with Albin Ahlbäck, Ricardo Buring, Grégoire Lecerf

CNRS, École polytechnique, France

Funded by the European Union

European Research Council
Established by the European Commission

ODELIX thematic program, Palaiseau **December 8, 2025**

Part I

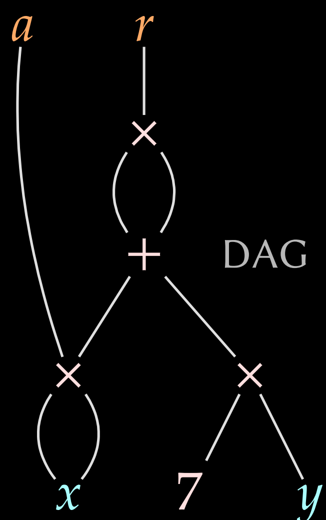
Introduction and motivation

Example of an SLP (Straight Line Program)

$$
\begin{aligned}
&\mathrm{in}(x, y)\\
&a := x \cdot x\\
&b := 7 \cdot y\\
&r := a + b\\
&r := r \cdot r\\
&\mathrm{out}(a, r)
\end{aligned}
$$

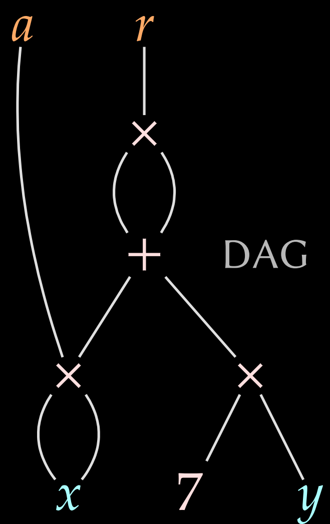
Example of an SLP (Straight Line Program)

$$
\begin{aligned}
&\text{in}(x, y)\\
&a := x \cdot x\\
&b := 7 \cdot y\\
&r := a + b\\
&r := r \cdot r\\
&\text{out}(a, r)
\end{aligned}
$$



Example of an SLP (Straight Line Program)

$$\begin{aligned} &\mathrm{in}(x, y) \\ &a := x \cdot x \\ &b := 7 \cdot y \\ &r := a + b \\ &r := r \cdot r \\ &\mathrm{out}(a, r) \end{aligned}$$



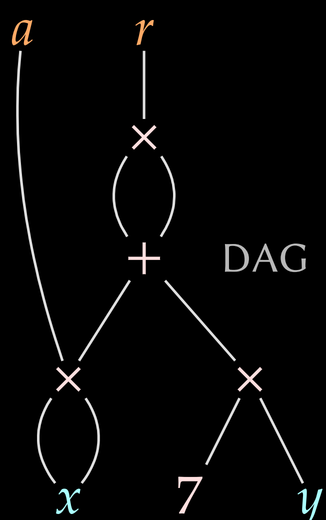
DAG

$$(x^2, (x^2 + 7y)^2)$$

Expression

Example of an SLP (Straight Line Program)

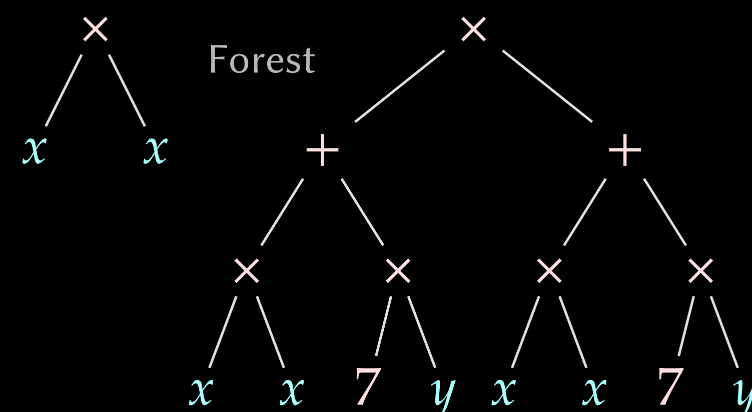
$$
\begin{aligned}
&\mathrm{in}(x, y)\\
&a := x \cdot x\\
&b := 7 \cdot y\\
&r := a + b\\
&r := r \cdot r\\
&\mathrm{out}(a, r)
\end{aligned}
$$



DAG

$$(x^2, (x^2 + 7y)^2)$$

Expression



Forest

Why SLPs ?

Theory

- Algebraic complexity analysis

Why SLPs ?

Theory

- Algebraic complexity analysis
- More general than it seems $\longrightarrow$ recording

Why SLPs ?

Theory

- Algebraic complexity analysis
- More general than it seems $\longrightarrow$ recording

Practice

- Evaluate same SLP many times $\longrightarrow$ allows for massive parallelism

Why SLPs ?

Theory

- Algebraic complexity analysis
- More general than it seems $\longrightarrow$ recording

Practice

- Evaluate same SLP many times $\longrightarrow$ allows for massive parallelism
- Trivial control $\longrightarrow$ easier to compile and optimize

Why SLPs ?

Theory

- Algebraic complexity analysis
- More general than it seems $\longrightarrow$ recording

Practice

- Evaluate same SLP many times $\longrightarrow$ allows for massive parallelism
- Trivial control $\longrightarrow$ easier to compile and optimize
- Optimization techniques from both compilers and symbolic computation

Why SLPs ?

Theory

- Algebraic complexity analysis
- More general than it seems $\longrightarrow$ recording

Practice

- Evaluate same SLP many times $\longrightarrow$ allows for massive parallelism
- Trivial control $\longrightarrow$ easier to compile and optimize
- Optimization techniques from both compilers and symbolic computation

Applications

- Homotopy continuation

Why SLPs ?

Theory

- Algebraic complexity analysis
- More general than it seems ⟶ recording

Practice

- Evaluate same SLP many times ⟶ allows for massive parallelism
- Trivial control ⟶ easier to compile and optimize
- Optimization techniques from both compilers and symbolic computation

Applications

- Homotopy continuation, geometric resolution

Why SLPs ?

Theory

- Algebraic complexity analysis
- More general than it seems ⟶ recording

Practice

- Evaluate same SLP many times ⟶ allows for massive parallelism
- Trivial control ⟶ easier to compile and optimize
- Optimization techniques from both compilers and symbolic computation

Applications

- Homotopy continuation, geometric resolution, gradient descent

Why SLPs ?

Theory

- Algebraic complexity analysis
- More general than it seems $\longrightarrow$ recording

Practice

- Evaluate same SLP many times $\longrightarrow$ allows for massive parallelism
- Trivial control $\longrightarrow$ easier to compile and optimize
- Optimization techniques from both compilers and symbolic computation

Applications

- Homotopy continuation, geometric resolution, gradient descent
- Discrete Fourier Transforms

Why SLPs ?

Theory

- Algebraic complexity analysis
- More general than it seems $\longrightarrow$ recording

Practice

- Evaluate same SLP many times $\longrightarrow$ allows for massive parallelism
- Trivial control $\longrightarrow$ easier to compile and optimize
- Optimization techniques from both compilers and symbolic computation

Applications

- Homotopy continuation, geometric resolution, gradient descent
- Discrete Fourier Transforms, multi-precision arithmetic

Why SLPs ?

Theory

- Algebraic complexity analysis
- More general than it seems $\longrightarrow$ recording

Practice

- Evaluate same SLP many times $\longrightarrow$ allows for massive parallelism
- Trivial control $\longrightarrow$ easier to compile and optimize
- Optimization techniques from both compilers and symbolic computation

Applications

- Homotopy continuation, geometric resolution, gradient descent
- Discrete Fourier Transforms, multi-precision arithmetic, other"codelets"

Why SLPs ?

Theory

- Algebraic complexity analysis
- More general than it seems $\longrightarrow$ recording

Practice

- Evaluate same SLP many times $\longrightarrow$ allows for massive parallelism
- Trivial control $\longrightarrow$ easier to compile and optimize
- Optimization techniques from both compilers and symbolic computation

Applications

- Homotopy continuation, geometric resolution, gradient descent
- Discrete Fourier Transforms, multi-precision arithmetic, other“codelets”
- Solving ODEs via relaxed power series computations

Some references

Theory

- Bürgisser, Clausen, Shokrollahi: *Algebraic complexity theory*, 1997

Some references

Theory

- Bürgisser, Clausen, Shokrollahi: *Algebraic complexity theory*, 1997

Software

- Díaz, Kaltofen: *FoxBox*, 1998

Some references

Theory

- Bürgisser, Clausen, Shokrollahi: *Algebraic complexity theory*, 1997

Software

- Díaz, Kaltofen: *FoxBox*, 1998

Many *ad hoc* software for specific applications

- Geometric resolution ⟶ Aldaz, Castaño, Llovet, Martìnez, Hägele, Bruno, Heintz, Matera, Giusti, Lecerf, Salvy, Durvye, …, 2000–2008
- Codelets, DFTs ⟶ Frigo-Johnson, *FFTW3*, 1999, Püschel et al., *Spiral*, 2005
- Automatic differentiation ⟶ *Autodiff*, *Torch.autograd*, *Juliadiff*, …

Some references

Theory

- Bürgisser, Clausen, Shokrollahi: *Algebraic complexity theory*, 1997

Software

- Díaz, Kaltofen: *FoxBox*, 1998

Many *ad hoc* software for specific applications

- Geometric resolution ⟶ Aldaz, Castaño, Llovet, Martìnez, Hägele, Bruno, Heintz, Matera, Giusti, Lecerf, Salvy, Durvye, ..., 2000–2008
- Codelets, DFTs ⟶ Frigo-Johnson, *FFTW3*, 1999, Püschel et al., *Spiral*, 2005
- Automatic differentiation ⟶ *Autodiff*, *Torch.autograd*, *Juliadiff*, ...

More references

- vdH, Lecerf: *Towards a library for straight-line programs*, AAECC, 2025

Towards a dedicated library

Justinline (in Mathemagix) **(2015–2024)**

Towards a dedicated library

Justinline (in Mathemagix) **(2015–2024)**

- High level source code and interface.

Towards a dedicated library

JUSTINLINE (in MATHEMAGIX) **(2015–2024)**

- High level source code and interface.
- Reasonably fast on-the-fly compilation of SLPs.

Towards a dedicated library

Justinline (in Mathemagix) **(2015–2024)**

- High level source code and interface.
- Reasonably fast on-the-fly compilation of SLPs.
- Resulting machine code is fast.

Towards a dedicated library

JUSTINLINE (in MATHEMAGIX) **(2015–2024)**

- High level source code and interface.
- Reasonably fast on-the-fly compilation of SLPs.
- Resulting machine code is fast.
- Suite of high level transformations and routines on SLPs.

Towards a dedicated library

Justinline (in Mathemagix) **(2015–2024)**

- High level source code and interface.
- Reasonably fast on-the-fly compilation of SLPs.
- Resulting machine code is fast.
- Suite of high level transformations and routines on SLPs.

JIL (in C++) **(2024–*)**

Towards a dedicated library

Justinline (in Mathemagix) **(2015–2024)**

- High level source code and interface.
- Reasonably fast on-the-fly compilation of SLPs.
- Resulting machine code is fast.
- Suite of high level transformations and routines on SLPs.

JIL (in C++) **(2024–*)**

- Stand-alone C++ library $\longrightarrow$ lower threshold for developers and users.

Towards a dedicated library

JUSTINLINE (in MATHEMAGIX) **(2015–2024)**

- High level source code and interface.
- Reasonably fast on-the-fly compilation of SLPs.
- Resulting machine code is fast.
- Suite of high level transformations and routines on SLPs.

JIL (in C++) **(2024–*)**

- Stand-alone C++ library $\longrightarrow$ lower threshold for developers and users.
- Low level source code, planned bindings for MATHEMAGIX, JULIA, FLINT, …

Towards a dedicated library

JUSTINLINE (in MATHEMAGIX) **(2015–2024)**

- High level source code and interface.
- Reasonably fast on-the-fly compilation of SLPs.
- Resulting machine code is fast.
- Suite of high level transformations and routines on SLPs.

JIL (in C++) **(2024–*)**

- Stand-alone C++ library $\longrightarrow$ lower threshold for developers and users.
- Low level source code, planned bindings for MATHEMAGIX, JULIA, FLINT, …
- Very fast on-the-fly compilation of SLPs.

Towards a dedicated library

JUSTINLINE (in MATHEMAGIX) **(2015–2024)**

- High level source code and interface.
- Reasonably fast on-the-fly compilation of SLPs.
- Resulting machine code is fast.
- Suite of high level transformations and routines on SLPs.

JIL (in C++) **(2024–*)**

- Stand-alone C++ library ⟶ lower threshold for developers and users.
- Low level source code, planned bindings for MATHEMAGIX, JULIA, FLINT, …
- Very fast on-the-fly compilation of SLPs.
- More architectures: X86, ARM, OPENCL, CUDA, SASS.

Towards a dedicated library

Justinline (in Mathemagix) **(2015–2024)**

- High level source code and interface.
- Reasonably fast on-the-fly compilation of SLPs.
- Resulting machine code is fast.
- Suite of high level transformations and routines on SLPs.

JIL (in C++) **(2024–*)**

- Stand-alone C++ library $\longrightarrow$ lower threshold for developers and users.
- Low level source code, planned bindings for Mathemagix, Julia, Flint, …
- Very fast on-the-fly compilation of SLPs.
- More architectures: X86, ARM, OpenCL, Cuda, Sass.
- Beyond SLPs…?

Motivation and spoiler

		JustInline		Jil		GPU	
	sols	jit	exe_8	jit	exe_8	jit	exe_{4096}
$Katsura_6$	64	1.11 s	3 ms	7.13 ms	39.1 ms	1.00 s	68.9 ms
$Katsura_8$	256	2.17 s	10 ms	13.6 ms	45.5 ms	3.21 s	0.42 s
$Katsura_{10}$	1024	3.74 s	37 ms	23.8 ms	0.20 s	7.58 s	0.72 s
$Katsura_{12}$	4096	6.16 s	0.23 s	42.4 ms	0.46 s	15.3 s	1.22 s
$Katsura_{14}$	16384	9.34 s	1.59 s	90.1 ms	1.94 s	29.4 s	5.53 s
$Katsura_{16}$	65536	13.4 s	11.3 s	0.27 s	11.9 s	49.6 s	29.1 s
$Katsura_{18}$	262144	21.3 s	145 s	0.95 s	83.2 s	82.3 s	157 s
$Katsura_{20}$	1048576	45.1 s	824 s	3.95 s	512 s	124 s	911 s
$Posso_{3,3}$	27	0.35 s	<1 ms	3.05 ms	0.26 ms	0.49 s	5.98 ms
$Posso_{4,4}$	256	1.50 s	3 ms	11.9 ms	1.5 ms	1.76 s	0.12 s
$Posso_{5,5}$	3125	8.22 s	78 ms	52.4 ms	0.31 s	19.1 s	0.96 s

Intel Xeon 3,2 GHz, AVX2 (JustInline), AVX512 (JIL), 8 threads

Nvidia GeForce RTX 4070 Super GPU, OpenCL, 4096 threads

2nd order stepper (JustInline), 1st order stepper (JIL)

Modern hardware

CPU

- Out of order execution
- Cache hierarchies
- Up to 8 or 16 wide SIMD parallelism
- Up to 100 cores

Modern hardware

CPU

- Out of order execution
- Cache hierarchies
- Up to 8 or 16 wide SIMD parallelism
- Up to 100 cores

GPU

- Up to 10 000 cores
- Often limited to 32 bits
- Large transfer times with CPU
- Hard to program and processors mostly undocumented

Modern hardware

CPU

- Out of order execution
- Cache hierarchies
- Up to 8 or 16 wide SIMD parallelism
- Up to 100 cores

GPU

- Up to 10 000 cores
- Often limited to 32 bits
- Large transfer times with CPU
- Hard to program and processors mostly undocumented

TPU

- Only matrix multiplication, but about 8 times faster than SIMD in theory

Part II

Representing SLPs in JIL

Low level representation of SLPs

$$
\begin{aligned}
&\mathrm{in}(x, y)\\
a &:= x \cdot x\\
b &:= 7 \cdot y\\
r &:= a + b\\
r &:= r \cdot r\\
&\mathrm{out}(a, r)
\end{aligned}
$$

Low level representation of SLPs

$$\begin{array}{lll} \mathrm{in}(x, y) & & \\ a & := & x \cdot x \\ b & := & 7 \cdot y \\ r & := & a + b \\ r & := & r \cdot r \\ \mathrm{out}(a, r) & & \end{array}$$

$$\begin{array}{lll} \mathrm{in}(x_0, x_1) & & \\ x_3 & := & x_0 \cdot x_0 \\ x_4 & := & x_2 \cdot x_1 \\ x_5 & := & x_3 + x_4 \\ x_5 & := & x_5 \cdot x_5 \\ \mathrm{out}(x_3, x_5) & & \end{array}$$

Low level representation of SLPs

$\text{in}(x, y)$	$\text{in}(x_0, x_1)$
$a := x \cdot x$	$x_3 := x_0 \cdot x_0$
$b := 7 \cdot y$	$x_4 := x_2 \cdot x_1$
$r := a + b$	$x_5 := x_3 + x_4$
$r := r \cdot r$	$x_5 := x_5 \cdot x_5$
$\text{out}(a, r)$	$\text{out}(x_3, x_5)$

in

0	1

out

3	5

prg

×	3	0	0	×	4	2	1
+	5	3	4	+	5	5	5

data

0	0	7	0	0	0

Low level representation of SLPs

$\mathrm{in}(x, y)$	$\mathrm{in}(x_0, x_1)$
$a := x \cdot x$	$x_3 := x_0 \cdot x_0$
$b := 7 \cdot y$	$x_4 := x_2 \cdot x_1$
$r := a + b$	$x_5 := x_3 + x_4$
$r := r \cdot r$	$x_5 := x_5 \cdot x_5$
$\mathrm{out}(a, r)$	$\mathrm{out}(x_3, x_5)$

in

0	1

out

3	5

prg

×	3	0	0	×	4	2	1
+	5	3	4	+	5	5	5

data

0	0	7	0	0	0

in, out, prg: arrays of 32 bit integers

Low level representation of SLPs

$$\begin{array}{lcl} \mathrm{in}(x,y) & & \\ a & := & x \cdot x \\ b & := & 7 \cdot y \\ r & := & a + b \\ r & := & r \cdot r \\ \mathrm{out}(a,r) & & \end{array}
\qquad
\begin{array}{lcl} \mathrm{in}(x_0,x_1) & & \\ x_3 & := & x_0 \cdot x_0 \\ x_4 & := & x_2 \cdot x_1 \\ x_5 & := & x_3 + x_4 \\ x_5 & := & x_5 \cdot x_5 \\ \mathrm{out}(x_3,x_5) & & \end{array}$$

in

0	1

out

3	5

prg

×	3	0	0	×	4	2	1
+	5	3	4	+	5	5	5

data

0	0	7	0	0	0

in, out, prg: arrays of 32 bit integers

$\Longrightarrow$ operations $+$, $\times$, … encoded as 32 bit integers

Low level representation of SLPs

in(x, y)	in(x_0, x_1)
$a := x \cdot x$	$x_3 := x_0 \cdot x_0$
$b := 7 \cdot y$	$x_4 := x_2 \cdot x_1$
$r := a + b$	$x_5 := x_3 + x_4$
$r := r \cdot r$	$x_5 := x_5 \cdot x_5$
out(a, r)	out(x_3, x_5)

in

0	1

out

3	5

prg

×	3	0	0	×	4	2	1
+	5	3	4	+	5	5	5

data

0	0	7	0	0	0

in, out, prg: arrays of 32 bit integers

$\Longrightarrow$ operations $+$, $\times$, … encoded as 32 bit integers

data: array of elements in some “scalar domain” $\mathbb{K}$

Domains

Hardware domains

- $\mathbb{Z}_8, \mathbb{Z}_{16}, \mathbb{Z}_{32}, \mathbb{Z}_{64}, \mathbb{N}_8, \mathbb{N}_{16}, \mathbb{N}_{32}, \mathbb{N}_{64}, \mathbb{R}_{32}, \mathbb{R}_{64}$

Domains

Hardware domains

- $\mathbb{Z}_8, \mathbb{Z}_{16}, \mathbb{Z}_{32}, \mathbb{Z}_{64}, \mathbb{N}_8, \mathbb{N}_{16}, \mathbb{N}_{32}, \mathbb{N}_{64}, \mathbb{R}_{32}, \mathbb{R}_{64}$
- SIMD variants, e.g. $\mathbb{Z}_8^{64}, \mathbb{Z}_{16}^{32}, \mathbb{Z}_{32}^{16}, \mathbb{Z}_{64}^{8}, \mathbb{N}_8^{64}, \mathbb{N}_{16}^{32}, \mathbb{N}_{32}^{16}, \mathbb{N}_{64}^{8}, \mathbb{R}_{32}^{16}, \mathbb{R}_{64}^{8}$

Hardware domains

- $\mathbb{Z}_8, \mathbb{Z}_{16}, \mathbb{Z}_{32}, \mathbb{Z}_{64}, \mathbb{N}_8, \mathbb{N}_{16}, \mathbb{N}_{32}, \mathbb{N}_{64}, \mathbb{R}_{32}, \mathbb{R}_{64}$
- SIMD variants, e.g. $\mathbb{Z}_8^{64}, \mathbb{Z}_{16}^{32}, \mathbb{Z}_{32}^{16}, \mathbb{Z}_{64}^{8}, \mathbb{N}_8^{64}, \mathbb{N}_{16}^{32}, \mathbb{N}_{32}^{16}, \mathbb{N}_{64}^{8}, \mathbb{R}_{32}^{16}, \mathbb{R}_{64}^{8}$

Software domains

- $\mathbb{K}[\mathrm{i}]$, $\mathrm{Ball}(\mathbb{K}_{\mathrm{cen}}, \mathbb{K}_{\mathrm{rad}})$, ...

Domains

Hardware domains

- $\mathbb{Z}_8, \mathbb{Z}_{16}, \mathbb{Z}_{32}, \mathbb{Z}_{64}, \mathbb{N}_8, \mathbb{N}_{16}, \mathbb{N}_{32}, \mathbb{N}_{64}, \mathbb{R}_{32}, \mathbb{R}_{64}$
- SIMD variants, e.g. $\mathbb{Z}_8^{64}, \mathbb{Z}_{16}^{32}, \mathbb{Z}_{32}^{16}, \mathbb{Z}_{64}^{8}, \mathbb{N}_8^{64}, \mathbb{N}_{16}^{32}, \mathbb{N}_{32}^{16}, \mathbb{N}_{64}^{8}, \mathbb{R}_{32}^{16}, \mathbb{R}_{64}^{8}$

Software domains

- $\mathbb{K}[\mathrm{i}]$, $\mathrm{Ball}(\mathbb{K}_{\mathrm{cen}}, \mathbb{K}_{\mathrm{rad}})$, …
- $\mathbb{K}^n$, $\mathbb{K}^{r \times c}$, $\mathbb{K}[x]/(x^n)$, …

Domains

Hardware domains

- $\mathbb{Z}_8, \mathbb{Z}_{16}, \mathbb{Z}_{32}, \mathbb{Z}_{64}, \mathbb{N}_8, \mathbb{N}_{16}, \mathbb{N}_{32}, \mathbb{N}_{64}, \mathbb{R}_{32}, \mathbb{R}_{64}$
- SIMD variants, e.g. $\mathbb{Z}_8^{64}, \mathbb{Z}_{16}^{32}, \mathbb{Z}_{32}^{16}, \mathbb{Z}_{64}^{8}, \mathbb{N}_8^{64}, \mathbb{N}_{16}^{32}, \mathbb{N}_{32}^{16}, \mathbb{N}_{64}^{8}, \mathbb{R}_{32}^{16}, \mathbb{R}_{64}^{8}$

Software domains

- $\mathbb{K}[\mathrm{i}]$, $\mathrm{Ball}(\mathbb{K}_{\mathrm{cen}}, \mathbb{K}_{\mathrm{rad}})$, …
- $\mathbb{K}^n$, $\mathbb{K}^{r \times c}$, $\mathbb{K}[x]/(x^n)$, …
- $\mathbb{Z}_{64}/(m\,\mathbb{Z}_{64})$, …

Hardware domains

- $\mathbb{Z}_8, \mathbb{Z}_{16}, \mathbb{Z}_{32}, \mathbb{Z}_{64}, \mathbb{N}_8, \mathbb{N}_{16}, \mathbb{N}_{32}, \mathbb{N}_{64}, \mathbb{R}_{32}, \mathbb{R}_{64}$
- SIMD variants, e.g. $\mathbb{Z}_8^{64}, \mathbb{Z}_{16}^{32}, \mathbb{Z}_{32}^{16}, \mathbb{Z}_{64}^{8}, \mathbb{N}_8^{64}, \mathbb{N}_{16}^{32}, \mathbb{N}_{32}^{16}, \mathbb{N}_{64}^{8}, \mathbb{R}_{32}^{16}, \mathbb{R}_{64}^{8}$

Software domains

- $\mathbb{K}[\mathrm{i}]$, $\mathrm{Ball}(\mathbb{K}_{\mathrm{cen}}, \mathbb{K}_{\mathrm{rad}})$, …
- $\mathbb{K}^n$, $\mathbb{K}^{r \times c}$, $\mathbb{K}[x]/(x^n)$, …
- $\mathbb{Z}_{64}/(m\,\mathbb{Z}_{64})$, …
- $\mathrm{Recorder}(\mathbb{K})$

Operations

Signature Σ: the supported operations $\sigma\colon \mathbb{K}^{|\sigma|} \to \mathbb{K}$

$$\Sigma \;:=\; \Sigma_{\text{basic}} \cup \Sigma_{\text{ext}}$$

Operations

Signature Σ: the supported operations $\sigma\colon \mathbb{K}^{|\sigma|} \to \mathbb{K}$

$$\Sigma \;:=\; \Sigma_{\text{basic}} \cup \Sigma_{\text{ext}}$$

Σ_{basic}: operations that most domains $\mathbb{K}$ supports

$$\Sigma_{\text{basic}} \;\supseteq\; \{\text{move}, \text{neg}, \text{add}, \text{sub}, \text{sqr}, \text{mul}, \text{nmul}, \text{fma}, \text{fnma}, \ldots\}$$

Operations

Signature Σ: the supported operations $\sigma\colon \mathbb{K}^{|\sigma|} \to \mathbb{K}$

$$\Sigma \;:=\; \Sigma_{\text{basic}} \cup \Sigma_{\text{ext}}$$

Σ_{basic}: operations that most domains $\mathbb{K}$ supports

$$\begin{aligned}\Sigma_{\text{basic}} \;\supseteq\; & \{\text{move}, \text{neg}, \text{add}, \text{sub}, \text{sqr}, \text{mul}, \text{nmul}, \text{fma}, \text{fnma}, \ldots\} \\ & \cup \{\text{inv}, \text{div}, \ldots\}\end{aligned}$$

Operations

Signature Σ: the supported operations $\sigma\colon \mathbb{K}^{|\sigma|} \to \mathbb{K}$

$$\Sigma \;:=\; \Sigma_{\text{basic}} \cup \Sigma_{\text{ext}}$$

Σ_{basic}: operations that most domains $\mathbb{K}$ supports

$$\begin{aligned}\Sigma_{\text{basic}} \;\supseteq\;& \{\text{move}, \text{neg}, \text{add}, \text{sub}, \text{sqr}, \text{mul}, \text{nmul}, \text{fma}, \text{fnma}, \ldots\}\\ &\cup \{\text{inv}, \text{div}, \ldots\}\\ &\cup \{\text{min}, \text{max}, \text{abs}, \ldots\} \cup \{\text{floor}, \text{ceil}, \text{round}, \text{trunc}\}\end{aligned}$$

Operations

Signature Σ: the supported operations $\sigma\colon \mathbb{K}^{|\sigma|} \to \mathbb{K}$

$$\Sigma \;:=\; \Sigma_{\text{basic}} \cup \Sigma_{\text{ext}}$$

Σ_{basic}: operations that most domains $\mathbb{K}$ supports

$$\begin{aligned}\Sigma_{\text{basic}} \;\supseteq\; & \{\text{move}, \text{neg}, \text{add}, \text{sub}, \text{sqr}, \text{mul}, \text{nmul}, \text{fma}, \text{fnma}, \ldots\} \\ & \cup \{\text{inv}, \text{div}, \ldots\} \\ & \cup \{\text{min}, \text{max}, \text{abs}, \ldots\} \cup \{\text{floor}, \text{ceil}, \text{round}, \text{trunc}\} \\ & \cup \{\text{not}, \text{or}, \text{xor}, \text{and}\} \cup \{\text{eq}, \text{neq}, \text{lt}, \text{le}, \text{gt}, \text{ge}\}\end{aligned}$$

Operations

Signature Σ: the supported operations $\sigma\colon \mathbb{K}^{|\sigma|} \to \mathbb{K}$

$$\Sigma \;:=\; \Sigma_{\text{basic}} \cup \Sigma_{\text{ext}}$$

Σ_{basic}: operations that most domains $\mathbb{K}$ supports

$$\begin{aligned}\Sigma_{\text{basic}} \;\supseteq\; & \{\text{move}, \text{neg}, \text{add}, \text{sub}, \text{sqr}, \text{mul}, \text{nmul}, \text{fma}, \text{fnma}, \ldots\} \\ & \cup \{\text{inv}, \text{div}, \ldots\} \\ & \cup \{\text{min}, \text{max}, \text{abs}, \ldots\} \cup \{\text{floor}, \text{ceil}, \text{round}, \text{trunc}\} \\ & \cup \{\text{not}, \text{or}, \text{xor}, \text{and}\} \cup \{\text{eq}, \text{neq}, \text{lt}, \text{le}, \text{gt}, \text{ge}\}\end{aligned}$$

Σ_{ext}: further operations for special domains $\mathbb{K}$ and user extensions

$$\begin{aligned}\Sigma_{\text{ext}} \;\supseteq\; & \{\text{duplicate}, \text{permute}\} \\ & \cup \{\text{shl}, \text{shr}, \ldots\} \\ & \cup \{\text{addc}, \text{subc}, \text{fmac}, \ldots\}\end{aligned}$$

Special operations

Embedding Boolean $\hookrightarrow \mathbb{K}$

- Needed for $\mathrm{eq}, \mathrm{neq}, \ldots, \mathrm{not}, \mathrm{or}, \ldots$

Special operations

Embedding Boolean $\hookrightarrow \mathbb{K}$

- Needed for $\mathrm{eq}, \mathrm{neq}, \ldots, \mathrm{not}, \mathrm{or}, \ldots$
- Usually, natural implementation: for $\mathbb{K} = \mathbb{Z}_{64}$, take $\mathrm{false} \mapsto 0$, $\mathrm{true} \mapsto -1$.

Special operations

Embedding Boolean $\hookrightarrow \mathbb{K}$

- Needed for $\mathrm{eq}, \mathrm{neq}, \ldots, \mathrm{not}, \mathrm{or}, \ldots$
- Usually, natural implementation: for $\mathbb{K} = \mathbb{Z}_{64}$, take $\mathrm{false} \mapsto 0$, $\mathrm{true} \mapsto -1$.

Embedding $\mathbb{Z} \hookrightarrow \mathbb{K}$ and $\mathbb{Z}^w \hookrightarrow \mathbb{K}^w$

- For $\mathrm{shl}, \mathrm{shr}, \ldots$, where $\mathrm{shl}(a, n) := a\,2^n$

Special operations

Embedding Boolean $\hookrightarrow \mathbb{K}$

- Needed for $\mathrm{eq}, \mathrm{neq}, \ldots, \mathrm{not}, \mathrm{or}, \ldots$
- Usually, natural implementation: for $\mathbb{K} = \mathbb{Z}_{64}$, take $\mathrm{false} \mapsto 0$, $\mathrm{true} \mapsto -1$.

Embedding $\mathbb{Z} \hookrightarrow \mathbb{K}$ and $\mathbb{Z}^w \hookrightarrow \mathbb{K}^w$

- For $\mathrm{shl}, \mathrm{shr}, \ldots$, where $\mathrm{shl}(a, n) := a\,2^n$
- For $\mathrm{permute}$ on SIMD types, with $\mathrm{permute}(a, \pi) := (a_{\pi(0)}, \ldots, a_{\pi(w-1)})$

Special operations

Embedding Boolean $\hookrightarrow \mathbb{K}$

- Needed for $\mathrm{eq}, \mathrm{neq}, \ldots, \mathrm{not}, \mathrm{or}, \ldots$
- Usually, natural implementation: for $\mathbb{K} = \mathbb{Z}_{64}$, take $\mathrm{false} \mapsto 0$, $\mathrm{true} \mapsto -1$.

Embedding $\mathbb{Z} \hookrightarrow \mathbb{K}$ and $\mathbb{Z}^w \hookrightarrow \mathbb{K}^w$

- For $\mathrm{shl}, \mathrm{shr}, \ldots$, where $\mathrm{shl}(a, n) := a\, 2^n$
- For $\mathrm{permute}$ on SIMD types, with $\mathrm{permute}(a, \pi) := (a_{\pi(0)}, \ldots, a_{\pi(w-1)})$

Multi-sorted signatures

- We can also create union domains like $\mathbb{K} \cup \mathbb{L}$

Special operations

Embedding Boolean $\hookrightarrow \mathbb{K}$

- Needed for $\mathrm{eq}, \mathrm{neq}, \ldots, \mathrm{not}, \mathrm{or}, \ldots$
- Usually, natural implementation: for $\mathbb{K} = \mathbb{Z}_{64}$, take $\mathrm{false} \mapsto 0$, $\mathrm{true} \mapsto -1$.

Embedding $\mathbb{Z} \hookrightarrow \mathbb{K}$ and $\mathbb{Z}^w \hookrightarrow \mathbb{K}^w$

- For $\mathrm{shl}, \mathrm{shr}, \ldots$, where $\mathrm{shl}(a, n) := a\,2^n$
- For $\mathrm{permute}$ on SIMD types, with $\mathrm{permute}(a, \pi) := (a_{\pi(0)}, \ldots, a_{\pi(w-1)})$

Multi-sorted signatures

- We can also create union domains like $\mathbb{K} \cup \mathbb{L}$
- And introduce variants $\sigma_{\mathbb{K}}, \sigma_{\mathbb{L}}, \ldots$ of $\sigma \in \Sigma$ depending on the sort $\mathbb{K}$, $\mathbb{L}$

Special operations

Embedding Boolean $\hookrightarrow \mathbb{K}$

- Needed for $\mathrm{eq}, \mathrm{neq}, \ldots, \mathrm{not}, \mathrm{or}, \ldots$
- Usually, natural implementation: for $\mathbb{K} = \mathbb{Z}_{64}$, take $\mathrm{false} \mapsto 0$, $\mathrm{true} \mapsto -1$.

Embedding $\mathbb{Z} \hookrightarrow \mathbb{K}$ and $\mathbb{Z}^w \hookrightarrow \mathbb{K}^w$

- For $\mathrm{shl}, \mathrm{shr}, \ldots$, where $\mathrm{shl}(a, n) := a\, 2^n$
- For $\mathrm{permute}$ on SIMD types, with $\mathrm{permute}(a, \pi) := (a_{\pi(0)}, \ldots, a_{\pi(w-1)})$

Multi-sorted signatures

- We can also create union domains like $\mathbb{K} \cup \mathbb{L}$
- And introduce variants $\sigma_{\mathbb{K}}, \sigma_{\mathbb{L}}, \ldots$ of $\sigma \in \Sigma$ depending on the sort $\mathbb{K}$, $\mathbb{L}$
- And versions $\sigma_{\mathrm{condional}}$ with an extra $\mathrm{Boolean}$ argument (if $\mathbb{L} = \mathrm{Boolean}$)

Why include an operation like fma in Σ_{basic} ?

Cons

- Need to implement fma for many software domains

Why include an operation like fma in Σ_{basic} ?

Cons

- Need to implement fma for many software domains
- We could write a "clever" routine to simplify $ab + c \longrightarrow \text{fma}(a, b, c)$

Why include an operation like fma in Σ_{basic} ?

Cons

- Need to implement fma for many software domains
- We could write a “clever” routine to simplify $ab + c \longrightarrow \text{fma}(a, b, c)$

Pros

- fma corresponds to an important instruction in hardware

Why include an operation like fma in Σ_{basic} ?

Cons

- Need to implement fma for many software domains
- We could write a "clever" routine to simplify $a\,b + c \longrightarrow \mathrm{fma}(a,b,c)$

Pros

- fma corresponds to an important instruction in hardware
- $\mathrm{fma}(a,b,c)$ does more than $a\,b + c$ for $\mathbb{R}_{32}$ and $\mathbb{R}_{64}$

Why include an operation like fma in Σ_{basic} ?

Cons

- Need to implement fma for many software domains
- We could write a “clever” routine to simplify $ab+c \longrightarrow \text{fma}(a,b,c)$

Pros

- fma corresponds to an important instruction in hardware
- $\text{fma}(a,b,c)$ does more than $ab+c$ for $\mathbb{R}_{32}$ and $\mathbb{R}_{64}$
- Systematic support of $\pm x$, $\pm xy$, $\pm xy \pm z$ tends to yield better simplifications

Why include an operation like fma in Σ_{basic} ?

Cons

- Need to implement fma for many software domains
- We could write a "clever" routine to simplify $ab+c \longrightarrow \text{fma}(a,b,c)$

Pros

- fma corresponds to an important instruction in hardware
- $\text{fma}(a,b,c)$ does more than $ab+c$ for $\mathbb{R}_{32}$ and $\mathbb{R}_{64}$
- Systematic support of $\pm x$, $\pm xy$, $\pm xy \pm z$ tends to yield better simplifications
- On $\mathbb{Z}/p\mathbb{Z}$ via $\mathbb{R}_{64}$, better implementation of $\text{fma}(a,b,c)$ than $ab+c$:

$\text{reduce}(a)$

$$\begin{aligned} q &:= a \cdot u \\ q &:= \text{round}(q) \\ r &:= \text{fnma}(p,q,r) \end{aligned}$$

$ab+c$

$$\begin{aligned} h &:= a \cdot b \\ l &:= \text{fms}(a,b,h) \\ r &:= \text{reduce}(h) \\ r &:= l+r \\ r &:= r+c \end{aligned}$$

$\text{fma}(a,b,c)$

$$\begin{aligned} h &:= \text{fma}(a,b,c) \\ l &:= \text{fms}(a,b,h) \\ l &:= l+c \\ r &:= \text{reduce}(h) \\ r &:= l+r \end{aligned}$$

Part III

Using JIL

Typical workflow

$$\xrightarrow{\text{initiate}} \text{SLP } f_1 \xrightarrow{\text{transforms}} \text{SLP } f_2 \xrightarrow{\text{optimize}} \text{SLP } f_3 \xrightarrow{\text{backend}} \text{machine code}$$

Example: fast code for the cofactor matrix of a 4×4 matrix

- **Record** a program to compute a generic 4×4 determinant $\longrightarrow$ Initial SLP f_1
- Compute gradient of $f_1 \longrightarrow$ **algebraically transformed** SLP f_2
- Simplify the result $\longrightarrow$ **optimized** SLP f_3
- **Compile** f_3 for a specific architecture $\longrightarrow$ binary code inside memory

Typical workflow

$$\xrightarrow{\text{initiate}} \text{SLP } f_1 \xrightarrow{\text{transforms}} \text{SLP } f_2 \xrightarrow{\text{optimize}} \text{SLP } f_3 \xrightarrow{\text{backend}} \text{machine code}$$

Example: fast code for the cofactor matrix of a 4×4 matrix

- **Record** a program to compute a generic 4×4 determinant $\longrightarrow$ Initial SLP f_1
- Compute gradient of $f_1 \longrightarrow$ **algebraically transformed** SLP f_2
- Simplify the result $\longrightarrow$ **optimized** SLP f_3
- **Compile** f_3 for a specific architecture $\longrightarrow$ binary code inside memory

Typical workflow

$$\xrightarrow{\text{initiate}} \text{SLP } f_1 \xrightarrow{\text{transforms}} \text{SLP } f_2 \xrightarrow{\text{optimize}} \text{SLP } f_3 \xrightarrow{\text{backend}} \text{machine code}$$

Example: fast code for the cofactor matrix of a 4×4 matrix

- **Record** a program to compute a generic 4×4 determinant $\longrightarrow$ Initial SLP f_1
- Compute gradient of f_1 $\longrightarrow$ **algebraically transformed** SLP f_2
- Simplify the result $\longrightarrow$ **optimized** SLP f_3
- **Compile** f_3 for a specific architecture $\longrightarrow$ binary code inside memory

Typical workflow

$$\xrightarrow{\text{initiate}} \text{SLP } f_1 \xrightarrow{\text{transforms}} \text{SLP } f_2 \xrightarrow{\text{optimize}} \text{SLP } f_3 \xrightarrow{\text{backend}} \text{machine code}$$

Example: fast code for the cofactor matrix of a 4×4 matrix

- **Record** a program to compute a generic 4×4 determinant $\longrightarrow$ Initial SLP f_1
- Compute gradient of $f_1 \longrightarrow$ **algebraically transformed** SLP f_2
- Simplify the result $\longrightarrow$ **optimized** SLP f_3
- **Compile** f_3 for a specific architecture $\longrightarrow$ binary code inside memory

Typical workflow

$$\xrightarrow{\text{initiate}} \text{SLP } f_1 \xrightarrow{\text{transforms}} \text{SLP } f_2 \xrightarrow{\text{optimize}} \text{SLP } f_3 \xrightarrow{\text{backend}} \text{machine code}$$

Example: fast code for the cofactor matrix of a 4×4 matrix

- **Record** a program to compute a generic 4×4 determinant $\longrightarrow$ Initial SLP f_1
- Compute gradient of $f_1 \longrightarrow$ **algebraically transformed** SLP f_2
- Simplify the result $\longrightarrow$ **optimized** SLP f_3
- **Compile** f_3 for a specific architecture $\longrightarrow$ binary code inside memory

Recording

Assume that we have a generic C++ function

```
template<typename C> C
f (const C& x) { return x * x + x - 3; }
```

Recording

Assume that we have a generic C++ function

```
template<typename C> C
f (const C& x) { return x * x + x - 3; }
```

We may record an SLP for f as follows:

```
slp
record_f (const domain& tp) {
  recorder_start (tp);
  slp_variable in = input_variable ();
  slp_variable out= f (in);
  specify_output (out);
  return recorder_end ();
}
```

Recording

Assume that we have a generic C++ function

```
template<typename C> C
f (const C& x) { return x * x + x - 3; }
```

We may record an SLP for `f` as follows:

```
slp
record_f (const domain& tp) {
  recorder_start (tp);
  slp_variable in = input_variable ();
  slp_variable out= f (in);
  specify_output (out);
  return recorder_end ();
}
```

Idea: an instance of `slp_variable` specifies a data field on which to operate
operations on `slp_variable` are recorded instead of being executed

Recording – example

Assume that we have a generic C++ function

```
template<typename C> C
f (const C& x) { return x * x + x - 3; }
```

Recording – example

Assume that we have a generic C++ function

```
template<typename C> C
f (const C& x) { return x * x + x - 3; }
```

Step by step execution

```
recorder_start (tp)
```

in

out

prg

data

Recording – example

Assume that we have a generic C++ function

```
template<typename C> C
f (const C& x) { return x * x + x - 3; }
```

Step by step execution

```
recorder_start (tp)
in = input_variable ()
```

in [0]

out

prg

data [0]

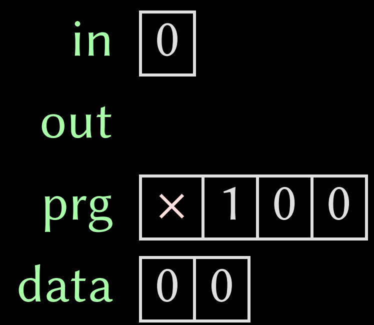
Recording – example

Assume that we have a generic C++ function

```
template<typename C> C
f (const C& x) { return x * x + x - 3; }
```

Step by step execution

```
recorder_start (tp)
in = input_variable ()
out= foo (in)
  x * x
```



Recording – example

Assume that we have a generic C++ function

```
template<typename C> C
f (const C& x) { return x * x + x - 3; }
```

Step by step execution

```
recorder_start (tp)
in = input_variable ()
out= foo (in)
  x * x
  x * x + x
```

in	0							
out								
prg	×	1	0	0	+	2	1	0
data	0	0	0					

Recording – example

Assume that we have a generic C++ function

```
template<typename C> C
f (const C& x) { return x * x + x - 3; }
```

Step by step execution

```
recorder_start (tp)
in = input_variable ()
out= foo (in)
  x * x
  x * x + x
  3
```

in | 0 |

out

prg | × | 1 | 0 | 0 | + | 2 | 1 | 0 |

data | 0 | 0 | 0 | 3 |

Recording – example

Assume that we have a generic C++ function

```
template<typename C> C
f (const C& x) { return x * x + x - 3; }
```

Step by step execution

```
recorder_start (tp)
in = input_variable ()
out= foo (in)
  x * x
  x * x + x
  3
  x * x + x - 3
```

in

0

out

prg

×	1	0	0	+	2	1	0	−	4	2	3

data

0	0	0	3	0

Recording – example

Assume that we have a generic C++ function

```
template<typename C> C
f (const C& x) { return x * x + x - 3; }
```

Step by step execution

```
recorder_start (tp)
in = input_variable ()
out= foo (in)
  x * x
  x * x + x
  3
  x * x + x - 3
output_variable (out)
```

in	0											
out	4											
prg	×	1	0	0	+	2	1	0	−	4	2	3
data	0	0	0	3	0							

Recording – example

Assume that we have a generic C++ function

```
template<typename C> C
f (const C& x) { return x * x + x - 3; }
```

Step by step execution

```
recorder_start (tp)
in = input_variable ()
out= foo (in)
  x * x
  x * x + x
  3
  x * x + x - 3
output_variable (out)
```

in	0											
out	4											
prg	×	1	0	0	+	2	1	0	−	4	2	3
data	0	0	0	3	0							

Note: clean execution trace more important than efficient execution of f

Traditional transformations

Important: by design, all transformations of SLP take linear time in JIL

Traditional transformations

Important: by design, all transformations of SLP take linear time in JIL

Simplification

- Common subexpression elimination
- CSE + algebraic simplifications ($0+x \to x$, etc.)
- Dead code elimination
- Rewrite $a\,b+c \longrightarrow \mathrm{fma}(a,b,c)$

Traditional transformations

Important: by design, all transformations of SLP take linear time in JIL

Simplification

- Common subexpression elimination
- CSE + algebraic simplifications ($0 + x \to x$, etc.)
- Dead code elimination
- Rewrite $a b + c \longrightarrow \mathrm{fma}(a, b, c)$

Backend

- Emulate missing instructions
- Optimizations for immediate arguments
- Rescheduling
- Register allocation

Algebraic transformations

General transformations

- Forward and backward differentiation: $f \longmapsto \mathrm{Jac}(f)$
- If f is linear (w.r.t. some of its inputs), then compute the transposed map
- $P \in \mathbb{K}[x_1, \ldots, x_n] \longmapsto$ homogeneous $\tilde{P} \in \mathbb{K}[x_1, \ldots, x_n, t]$ with $P(\boldsymbol{x}) = \tilde{P}(\boldsymbol{x}, 1)$
- Add just enough reductions to avoid overflows for redundant representations

Algebraic transformations

General transformations

- Forward and backward differentiation: $f \longmapsto \text{Jac}(f)$
- If f is linear (w.r.t. some of its inputs), then compute the transposed map
- $P \in \mathbb{K}[x_1, \ldots, x_n] \longmapsto$ homogeneous $\tilde{P} \in \mathbb{K}[x_1, \ldots, x_n, t]$ with $P(\boldsymbol{x}) = \tilde{P}(\boldsymbol{x}, 1)$
- Add just enough reductions to avoid overflows for redundant representations

Lifting and related transformations

- Lift SLP over $\mathbb{K}$ to SLP over $\mathbb{A}$ for $\mathbb{K}$-algebra $\mathbb{A}$
- Reinterpret SLP over $\mathbb{A}$ as SLP over $\mathbb{K}$
- Specific vector and ball lifts
- Reduce the number of divisions in SLPs

Example of complexification

$\mathbb{K}[i]$ is an **SLP algebra**: operations in Σ_{basic} can be implemented using SLPs:

add	subtract	multiply
$\text{in}(x_1, y_1, x_2, y_2)$	$\text{in}(x_1, y_1, x_2, y_2)$	$\text{in}(x_1, y_1, x_2, y_2)$
$x_3 := x_1 + x_2$	$x_3 := x_1 + x_2$	$x_3 := x_1 \cdot x_2$
$y_3 := y_1 + y_2$	$y_3 := y_1 + y_2$	$y_3 := x_1 \cdot y_2$
$\text{out}(x_3, y_3)$	$\text{out}(x_3, y_3)$	$x_3 := \text{fms}(y_1, y_2, x_3)$
		$y_3 := \text{fma}(x_2, y_1, y_3)$
		$\text{out}(x_3, y_3)$

Be careful with *aliasing* like in $z := u \cdot z$

Example of complexification

$\mathbb{K}[\mathrm{i}]$ is an **SLP algebra**: operations in Σ_{basic} can be implemented using SLPs:

add

$$\begin{aligned} x_3 &:= x_1 + x_2 \\ y_3 &:= y_1 + y_2 \end{aligned}$$

subtract

$$\begin{aligned} x_3 &:= x_1 + x_2 \\ y_3 &:= y_1 + y_2 \end{aligned}$$

multiply

$$\begin{aligned} x_3 &:= x_1 \cdot x_2 \\ y_3 &:= x_1 \cdot y_2 \\ x_3 &:= \mathrm{fms}(y_1, y_2, x_3) \\ y_3 &:= \mathrm{fma}(x_2, y_1, y_3) \end{aligned}$$

$$\begin{aligned} &\mathrm{in}(z_0) \\ z_1 &:= z_0 \cdot z_0 \\ z_2 &:= z_1 + z_0 \\ z_4 &:= z_2 - 7 \\ &\mathrm{out}(z_4) \end{aligned}$$

$\xrightarrow{\text{lift}}$

$$\begin{aligned} &\mathrm{in}(x_0, y_0) \\ x_1 &:= x_0 \cdot x_0 \\ y_1 &:= x_0 \cdot y_0 \\ x_1 &:= \mathrm{fms}(y_0, y_0, x_1) \\ y_1 &:= \mathrm{fma}(x_0, y_0, y_1) \\ x_2 &:= x_1 + x_0 \\ y_2 &:= y_1 + y_0 \\ x_4 &:= x_2 - 7 \\ y_4 &:= y_2 - 0 \\ &\mathrm{out}(x_4, y_4) \end{aligned}$$

$\xrightarrow{\text{simplify}}$

$$\begin{aligned} &\mathrm{in}(x_0, y_0) \\ x_1 &:= x_0 \cdot x_0 \\ y_1 &:= x_0 \cdot y_0 \\ x_1 &:= \mathrm{fms}(y_0, y_0, x_1) \\ y_1 &:= \mathrm{fma}(x_0, y_0, y_1) \\ x_2 &:= x_1 + x_0 \\ y_2 &:= y_1 + y_0 \\ x_4 &:= x_2 - 7 \\ &\mathrm{out}(x_4, y_2) \end{aligned}$$

Some timings

n	len	cse	sim	∇	lift	reg	jit	exe
2	2	1460	2687	1221	1796	1436	12784	10.00
3	9	543	825	407	614	562	2963	2.222
4	40	229	341	184	262	291	924	0.550
5	205	126	196	125	163	237	452	0.424
6	1236	97	149	112	138	221	455	0.422
7	8659	89	144	127	134	229	447	0.419
8	69280	94	171	125	160	261	470	0.424
9	623529	133	207	188	178	347	472	0.865
10	6235300	158	242	296	245	391	522	3.308

Table. Timings in cycles per instruction on INTEL XEON for (very) naive $n \times n$ determinants.

Part IV
Upcoming features

Integrated benching, selection, and caching

We can efficiently generated multiple implementations of a function and determine which one is best by benching each implementation

Integrated benching, selection, and caching

We can efficiently generated multiple implementations of a function and determine which one is best by benching each implementation

This requires

- Good random sample generators
- Mechanism to list alternative implementations and bench them
- Mechanism to cache results on disk
- Mechanism to predict timings without running any code

Beyond SLPs

Typical use cases:

- Run SLP $f: \mathbb{K}^m \rightarrow \mathbb{K}^n$ on vectors: $\tilde{f}: (\mathbb{K}^m)^N \longrightarrow (\mathbb{K}^n)^N$ with $N \gg 1$

Beyond SLPs

Typical use cases:

- Run SLP $f\colon \mathbb{K}^m \to \mathbb{K}^n$ on vectors: $\tilde{f}\colon (\mathbb{K}^m)^N \longrightarrow (\mathbb{K}^n)^N$ with $N \gg 1$
- Iterate SLP $f\colon \mathbb{K}^n \to \mathbb{K}^n$ until condition is met (e.g. homotopy continuation)

Beyond SLPs

Typical use cases:

- Run SLP $f: \mathbb{K}^m \rightarrow \mathbb{K}^n$ on vectors: $\tilde{f}: (\mathbb{K}^m)^N \longrightarrow (\mathbb{K}^n)^N$ with $N \gg 1$
- Iterate SLP $f: \mathbb{K}^n \rightarrow \mathbb{K}^n$ until condition is met (e.g. homotopy continuation)
- SLP over "big" SLP algebra $\mathbb{A}$ over $\mathbb{K}$ $\longrightarrow$ subroutines for operations in $\mathbb{A}$

Beyond SLPs

Typical use cases:

- Run SLP $f: \mathbb{K}^m \to \mathbb{K}^n$ on vectors: $\tilde{f}: (\mathbb{K}^m)^N \longrightarrow (\mathbb{K}^n)^N$ with $N \gg 1$
- Iterate SLP $f: \mathbb{K}^n \to \mathbb{K}^n$ until condition is met (e.g. homotopy continuation)
- SLP over "big" SLP algebra $\mathbb{A}$ over $\mathbb{K}$ $\longrightarrow$ subroutines for operations in $\mathbb{A}$

Need for *shallow* control structures

Beyond SLPs

Typical use cases:

- Run SLP $f: \mathbb{K}^m \to \mathbb{K}^n$ on vectors: $\tilde{f}: (\mathbb{K}^m)^N \longrightarrow (\mathbb{K}^n)^N$ with $N \gg 1$
- Iterate SLP $f: \mathbb{K}^n \to \mathbb{K}^n$ until condition is met (e.g. homotopy continuation)
- SLP over "big" SLP algebra $\mathbb{A}$ over $\mathbb{K}$ $\longrightarrow$ subroutines for operations in $\mathbb{A}$

Need for *shallow* control structures

Dedicated support for various frequent patters…
… or better to use general purpose techniques?

Example on AVX512 platform

Input: $N \in \mathbb{N}$ and vectors $u, v \in \mathbb{R}_{64}[\mathrm{i}]^N$

Output: $w \in \mathbb{R}_{64}[\mathrm{i}]^N$ with $w_i = u_i v_i$ for all $0 \leqslant i < N$

Example on AVX512 platform

Input: $N \in \mathbb{N}$ and vectors $u, v \in \mathbb{R}_{64}[\mathrm{i}]^N$

Output: $w \in \mathbb{R}_{64}[\mathrm{i}]^N$ with $w_i = u_i v_i$ for all $0 \leqslant i < N$

First assume that $N = 32$.

Example on AVX512 platform

Input: $N \in \mathbb{N}$ and vectors $u, v \in \mathbb{R}_{64}[\mathrm{i}]^N$

Output: $w \in \mathbb{R}_{64}[\mathrm{i}]^N$ with $w_i = u_i v_i$ for all $0 \leqslant i < N$

First assume that $N = 32$.

- Reinterpret u, v as $\tilde{u}, \tilde{v} \in (\mathbb{R}_{64}^8)^4[\mathrm{i}]$ (this requires SIMD matrix transposition)
- Lift complex multiplication over $\mathbb{R}_{64}$ to multiplication in $(\mathbb{R}_{64}^8)^4[\mathrm{i}]$ over $\mathbb{R}_{64}^8$
- Reinterpret $\tilde{w} := \tilde{u}\,\tilde{v}$ as an element of $\mathbb{R}_{64}[\mathrm{i}]$ (another SIMD transposition)

Example on AVX512 platform

Input: $N \in \mathbb{N}$ and vectors $u, v \in \mathbb{R}_{64}[\mathrm{i}]^N$

Output: $w \in \mathbb{R}_{64}[\mathrm{i}]^N$ with $w_i = u_i v_i$ for all $0 \leqslant i < N$

First assume that $N = 32$.

- Reinterpret u, v as $\tilde{u}, \tilde{v} \in (\mathbb{R}_{64}^8)^4[\mathrm{i}]$ (this requires SIMD matrix transposition)
- Lift complex multiplication over $\mathbb{R}_{64}$ to multiplication in $(\mathbb{R}_{64}^8)^4[\mathrm{i}]$ over $\mathbb{R}_{64}^8$
- Reinterpret $\tilde{w} := \tilde{u}\,\tilde{v}$ as an element of $\mathbb{R}_{64}[\mathrm{i}]$ (another SIMD transposition)

Create a loop which repeats this code until $N < 32$.

Example on AVX512 platform

Input: $N \in \mathbb{N}$ and vectors $u, v \in \mathbb{R}_{64}[\mathrm{i}]^N$

Output: $w \in \mathbb{R}_{64}[\mathrm{i}]^N$ with $w_i = u_i v_i$ for all $0 \leqslant i < N$

First assume that $N = 32$.

- Reinterpret u, v as $\tilde{u}, \tilde{v} \in (\mathbb{R}_{64}^8)^4[\mathrm{i}]$ (this requires SIMD matrix transposition)
- Lift complex multiplication over $\mathbb{R}_{64}$ to multiplication in $(\mathbb{R}_{64}^8)^4[\mathrm{i}]$ over $\mathbb{R}_{64}^8$
- Reinterpret $\tilde{w} := \tilde{u}\,\tilde{v}$ as an element of $\mathbb{R}_{64}[\mathrm{i}]$ (another SIMD transposition)

Create a loop which repeats this code until $N < 32$. Next

Example on AVX512 platform

Input: $N \in \mathbb{N}$ and vectors $u, v \in \mathbb{R}_{64}[\mathrm{i}]^N$
Output: $w \in \mathbb{R}_{64}[\mathrm{i}]^N$ with $w_i = u_i v_i$ for all $0 \leqslant i < N$

First assume that $N = 32$.

- Reinterpret u, v as $\tilde{u}, \tilde{v} \in (\mathbb{R}_{64}^8)^4[\mathrm{i}]$ (this requires SIMD matrix transposition)
- Lift complex multiplication over $\mathbb{R}_{64}$ to multiplication in $(\mathbb{R}_{64}^8)^4[\mathrm{i}]$ over $\mathbb{R}_{64}^8$
- Reinterpret $\tilde{w} := \tilde{u}\,\tilde{v}$ as an element of $\mathbb{R}_{64}[\mathrm{i}]$ (another SIMD transposition)

Create a loop which repeats this code until $N < 32$. Next

- Reduce further to the cases when $N < 16$ and then $N < 8$
- Reduce the case when $N < 8 \longrightarrow$ case $N = 8$ with an appropriate mask

Example on AVX512 platform

Input: $N \in \mathbb{N}$ and vectors $u, v \in \mathbb{R}_{64}[\mathrm{i}]^N$

Output: $w \in \mathbb{R}_{64}[\mathrm{i}]^N$ with $w_i = u_i v_i$ for all $0 \leqslant i < N$

First assume that $N = 32$.

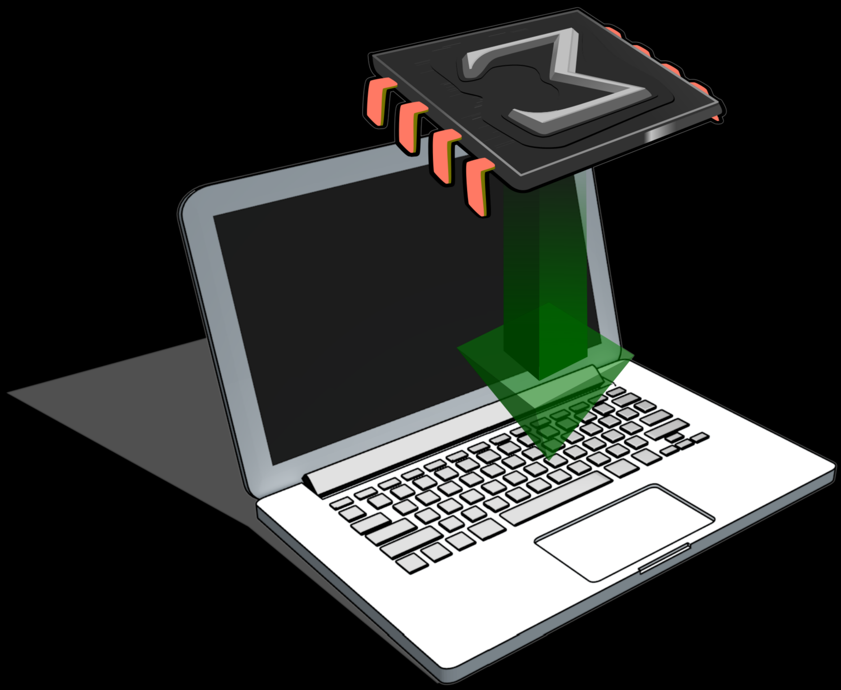
- Reinterpret u, v as $\tilde{u}, \tilde{v} \in (\mathbb{R}_{64}^8)^4[\mathrm{i}]$ (this requires SIMD matrix transposition)
- Lift complex multiplication over $\mathbb{R}_{64}$ to multiplication in $(\mathbb{R}_{64}^8)^4[\mathrm{i}]$ over $\mathbb{R}_{64}^8$
- Reinterpret $\tilde{w} := \tilde{u}\,\tilde{v}$ as an element of $\mathbb{R}_{64}[\mathrm{i}]$ (another SIMD transposition)

Create a loop which repeats this code until $N < 32$. Next

- Reduce further to the cases when $N < 16$ and then $N < 8$
- Reduce the case when $N < 8 \longrightarrow$ case $N = 8$ with an appropriate mask

This whole implementation can be run using only AVX512 vector instructions.

Thank you !



http://www.TeXmacs.org