July 18th 2018, NIPS Lab (Perugia, IT)

International Summer School on Energy Aware Transprecision Computing

Funded by the H2020 Framework
Programme of the European Union

SW and TOOLS

Overview of integrated support for Transprecision Computing

Andrea Marongiu (*a.marongiu@unibo.it*)
Giuseppe Tagliavini (*giuseppe.tagliavini@unibo.it*)

DISI - Department of Computer Science and Engineering
DEI – Department of Electronic Engineering
University of Bologna
Bologna, Italy

OPRECOMP
Open Transprecision Computing

© 2017 OPRECOMP – *http://oprecomp.eu*

Agenda

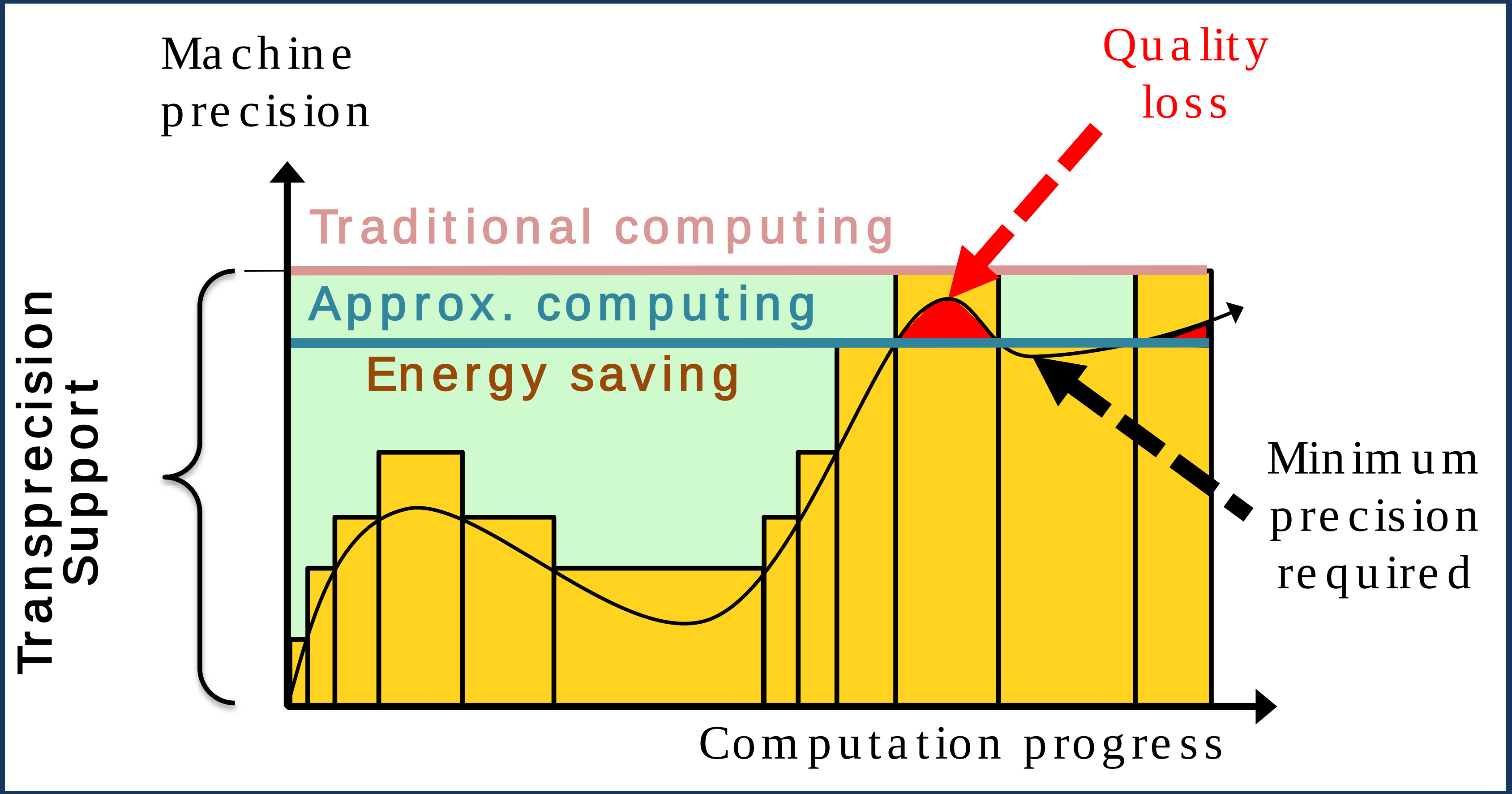
- ☐ **Introduction – Transprecision Computing**
- ☐ *Smaller-than-32-bit* floating point types
- ☐ Implementing the *smallFloat* extension
  - HW support
  - Compiler support
- ☐ Simplifying the deployment of *SmallFloat-based* applications
- ☐ Conclusion

Towards a new computing paradigm: Transprecision Computing

Beyond approximate computing!
A transprecision computing framework:

- ❑ controls approximation in space and time (when and where) at a fine grain though multiple hardware and software feedback control loops.
- ❑ does not imply reduced precision at the application level
  - ▪ it is still possible to soften precision requirements for extra benefits.
- ❑ defines computing architectures that operate with a smooth and wide range of precision vs. cost trade-off curve.

OPRECOMP
Open Transprecision Computing [5]



[5]: Malossi et al.: The Transprecision Computing Paradigm: Concept, Design, and Applications. DATE 2018, 2018.

Towards a new computing paradigm: **Transprecision Computing**

- ❑ lack of an *application-to-hardware* framework for managing precision without compromising application quality.
  - key barrier to a widespread adoption of classic approximate computing

- ❑ in a *transprecision computing framework* this limit is overcome via fine-grained and distributed control of hardware operation coupled with static and dynamic software control
  - Compiler support to extended floating-point data types
  - feedback based programming model enabling on-line tracking of error metrics and modulation of operating parameters

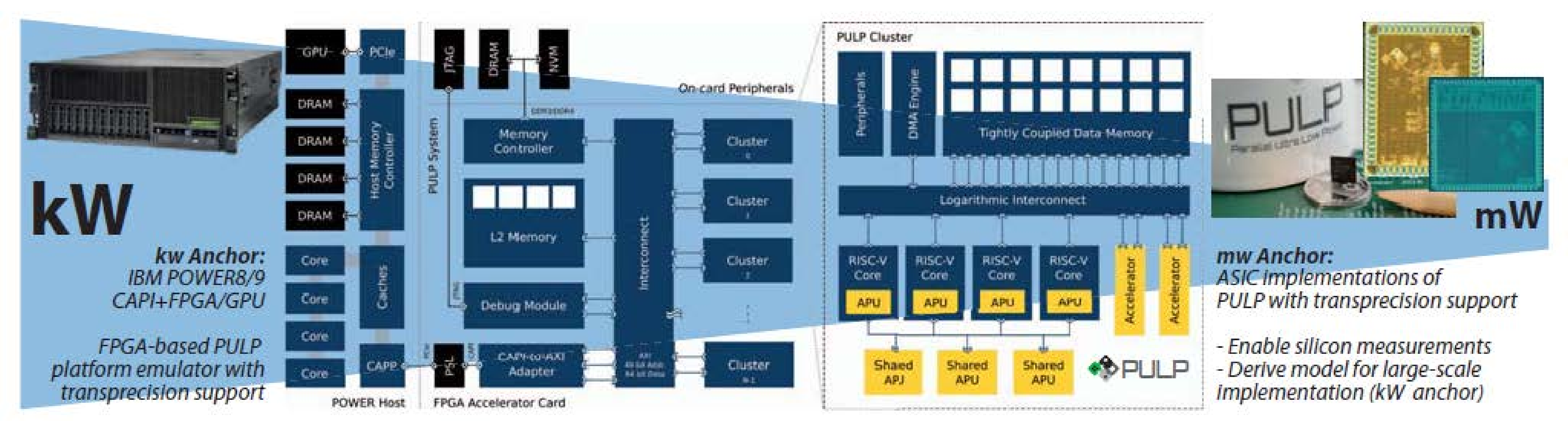
Towards a new computing paradigm: **Transprecision Computing**

- In practice, there are several different approaches taken to achieve this goal within the project

- The focus of this talk is on floating-point computation
  - Methodologies to discipline the use of reduced precision computation in applications (e.g., explore minimum precision requirements in applications)
  - Use of such methodologies in an integrated framework
  - Automation of manual procedures from state-of-the-art approaches

Towards a new computing paradigm: Transprecision Computing

The PULP platform



kw Anchor:
IBM POWER8/9
CAPI+FPGA/GPU

FPGA-based PULP platform emulator with transprecision support

mw Anchor:
ASIC implementations of PULP with transprecision support

- Enable silicon measurements
- Derive model for large-scale implementation (kW anchor)

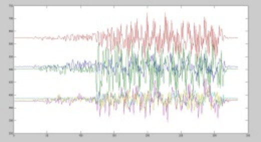
- ❑ The key focus of this talk is on the mW anchor
  - but the techniques apply to large-scale, high-performance targets as well

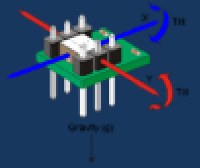
Towards a new computing paradigm: Transprecision Computing

Context: Distributed Embedded Computing

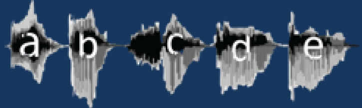
Sense

MEMS IMU

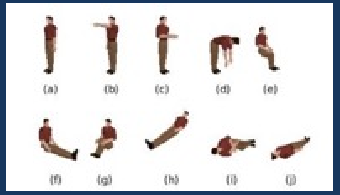




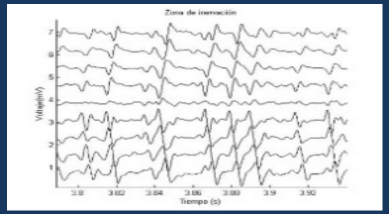
MEMS Microphone



ULP Imager

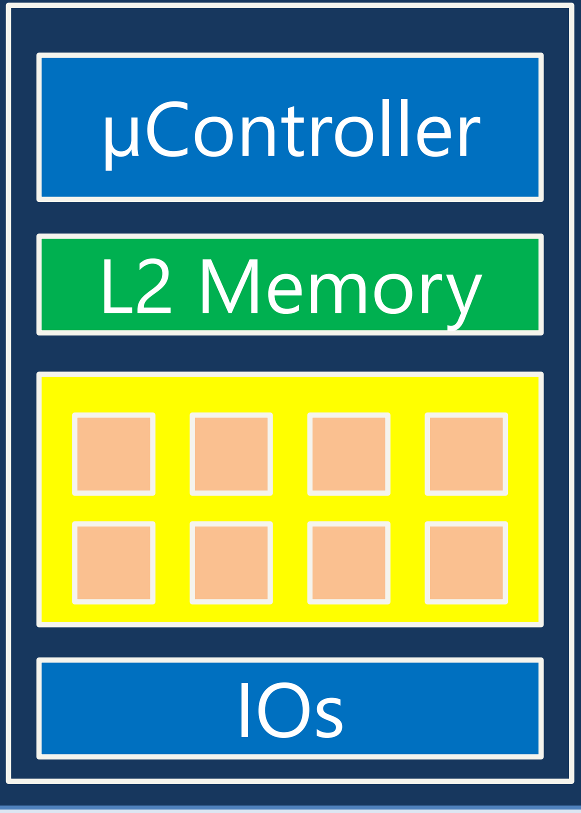


EMG/ECG/EIT



100 µW ÷ 2 mW

Analyze and Classify



µController

L2 Memory

IOs

1 ÷ 2000 MOPS
1 ÷ 10 mW

Transmit

Short range, medium BW

LoRa SEMTECH

Long range, low BW

Low rate (periodic) data

Idle: ~1µW
Active:~ 50mW

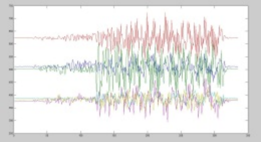
[1]: Tagliavini et al.: A Transprecision Floating-Point Platform for Ultra-Low Power Computing. DATE 2018, 2018.

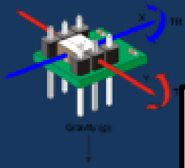
Towards a new computing paradigm: Transprecision Computing

Context: Distributed Embedded Computing

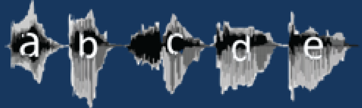
Sense

MEMS IMU

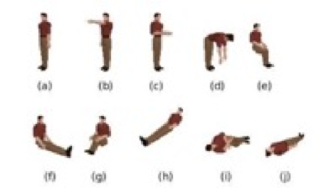


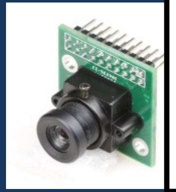


MEMS Microphone

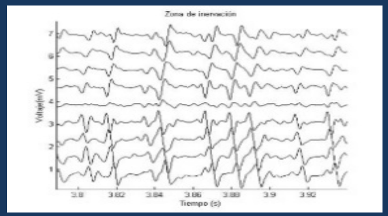


ULP Imager





EMG/ECG/EIT



100 µW ÷ 2 mW

Analyze and Classify

IOs

1 ÷ 2000 MOPS
1 ÷ 10 mW

Low Power, High Performance

- Data processing usually requires FP support
- HW support needed for performance (speed)
- Up to 50% of processor power for FP-related operations. [1]

→ Make processing more **energy efficient** on a **system level**

Transmit

Short range, medium BW

LoRa SEMTECH

Long range, low BW

Low rate (periodic) data

Idle: ~1µW
Active:~ 50mW

[1]: Tagliavini et al.: A Transprecision Floating-Point Platform for Ultra-Low Power Computing. DATE 2018, 2018.

Agenda

- ❑ Introduction – Transprecision Computing
- ❑ ***Smaller-than-32-bit*** **floating point types**
- ❑ Implementing the *smallFloat* extension
  - HW support
  - Compiler support
- ❑ Simplifying the deployment of *SmallFloat-based* applications
- ❑ Conclusion

The Need for Floating-Point Arithmetic

Do we need **floating-point** at all?

- ❑ **Fixed-Point?**
  - Not enough flexibility (dynamic range)
  - Manual tuning required
- ❑ **Logarithmic Number Systems (LNS)?**
  - Add/Subtract very expensive. [1]
- ❑ **UNUM?**
  - Unwieldy for LP HW implementation. [2]

[1] Gautschi et al.: An Extended Shared Logarithmic Unit for Nonlinear Function Kernel Acceleration in a 65-nm CMOS Multicore Cluster. IEEE Journal of Solid-State Circuits, 52(1):98–112, 2017.

[2] Glaser et al.: An 826 MOPS, 210 uW/MHz Unum ALU in 65 nm. ISCAS 2018

The Need for Floating-Point Arithmetic

Floating point formats

- ❑ Floating-point (FP) formats are widely adopted to design applications characterized by a **large dynamic range**

- ❑ IEEE 754 specification defines an encoding format that breaks a FP number into 3 parts:
  a ***sign***, a ***mantissa***, and an ***exponent***
  - **exponent** ⇔ ***dynamic range***
  - **mantissa** ⇔ ***precision***

The Need for Floating-Point Arithmetic

❑ IEEE 754-2008 standard types

- *binary16* (half precision)
- *binary32* (single precision)
- *binary64* (double precision)
- *binary128* (quadruple precision)

Mostly used by programmers (so far…)

Available in embedded systems

FP16 support on NVidia GPUs

- **IEEE 754 formats** → 1 bit sign, *e* bits exponent, *m* bits mantissa

FP16																
	s	e	e	e	e	e	m	m	m	m	m	m	m	m	m	m

- FP16 can represent 30,720 values → 1024 values between 2^{-14} and 2^{15}
- NVIDIA Tesla P100 and newer GPUs support a 2-way vector half-precision unit

- Support in CUDA in **cuda_fp16.h**
  - **half** and **half2** data types
  - **intrinsic functions** for operating on data types
  - **2x faster than FP32**

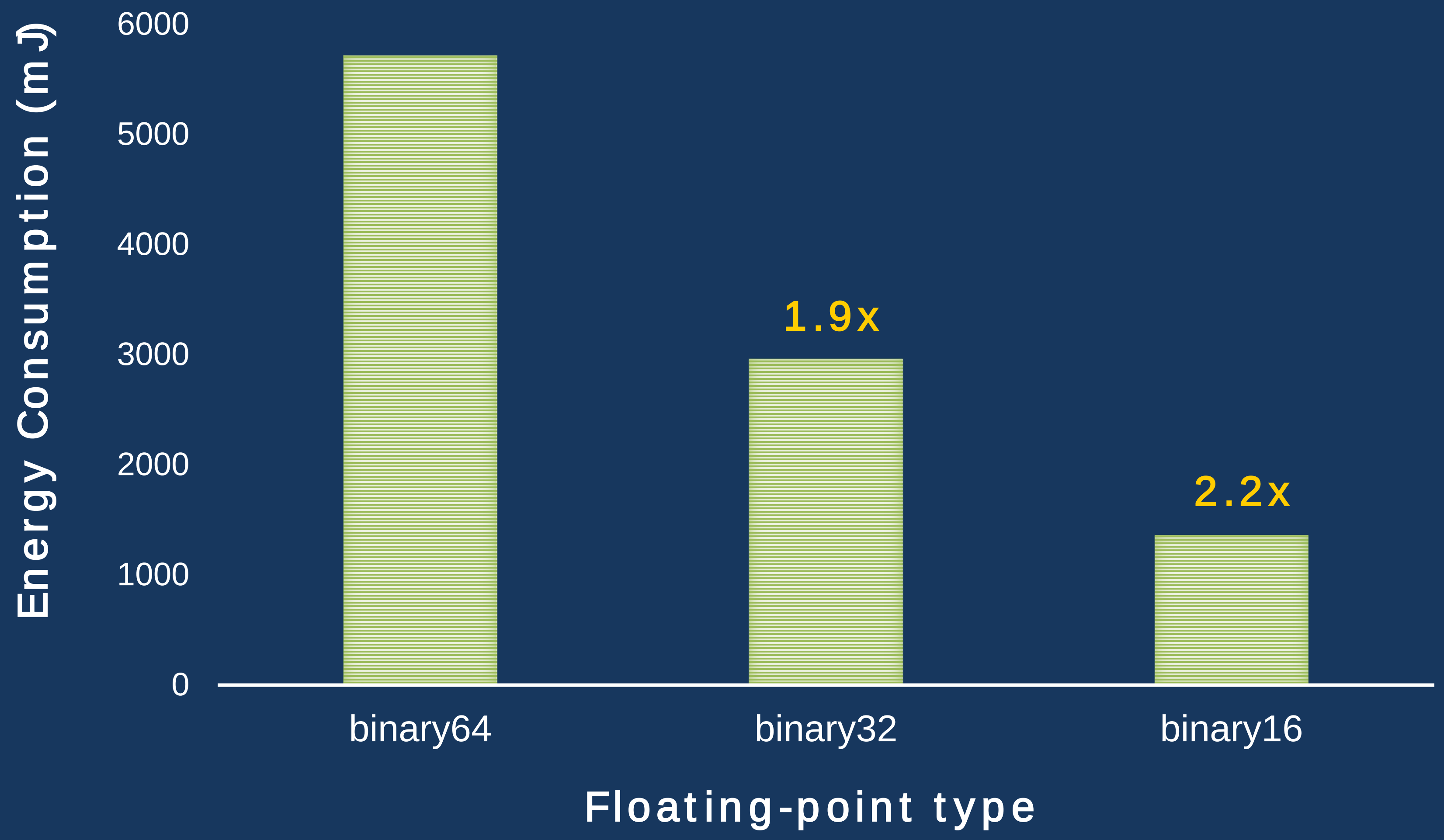
- Mixed-precision programming is integrated in CUDA libraries
  - *cuDNN, TensorRT, cuBLAS, cuFFT, cuSPARSE*

saxpy CUDA kernel using half arithmetic

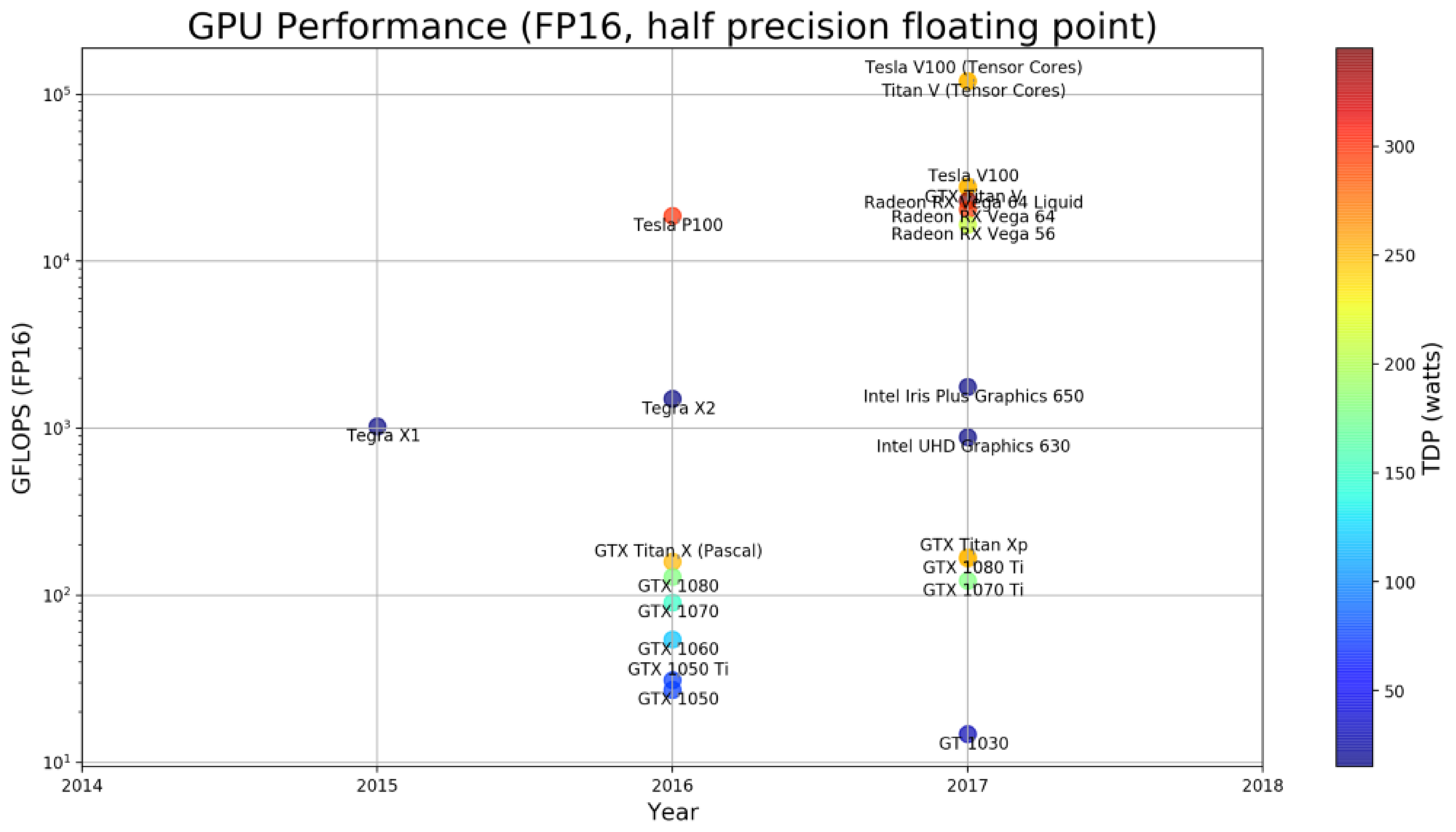
```
__global__
void saxpy(int n, half a, const half *x, half *y) {
  int start = threadIdx.x + blockDim.x * blockIdx.x;
  int stride = blockDim.x * gridDim.x;

  int n2 = n/2;
  half2  a2 = __halves2half2(a, a);
  half2 *x2 = (half2*)x
  half2 *y2 = (half2*)y;

  for (int i = start; i < n2; i+= stride)
    y2[i] = __hfma2(a2, x2[i], y2[i]);

  if (start == 0 && (n%2))
    y[n-1] = __hfma(a, x[n-1], y[n-1]);
}
```

Compiler intrinsics to program operation of non-standard types

Energy consumption of saxpy (NVidia Tegra X2 GPU)



FP16 on modern GPUs: the full picture



Source: https://blog.inten.to

Smaller-than-32bit floating point types one step further

1) How much precision do we actually need?

- ❑ **Only two levels of precision are quite limited**
  - Why stop there?
  - Which ones are useful? [3]

2) How to simplify deployment of applications with *smaller-than-32-bit* floats?

[3]: Tagliavini et al.: A Transprecision Floating-Point Platform for Ultra-Low Power Computing. DATE 2018, 2018.

Smaller-than-32bit floating point types one step further

SmallFloat formats for transprecision computing

- ❑ **Trans-precision computing**
  1. strong focus on the precision of **intermediate computations**
  2. exploiting *application-level softening of* ***precision requirements*** for extra benefits (e.g., **energy saving**)

- ❑ *Smaller-than-32-bit* FP formats (**smallFloats** can reduce <u>execution time</u> and <u>energy consumption</u>
  - **Simpler logic in arithmetic units**
  - **Vectorization**
  - **Bandwidth reduction**

SmallFloat **extension of a standard FP type system**

- Need architecture support
- Need compiler support (language frontend, machine backend)

How to address the two key goals?

1. Supporting the *SmallFloat* data type extension
    - Hardware Support
    - Compiler Support

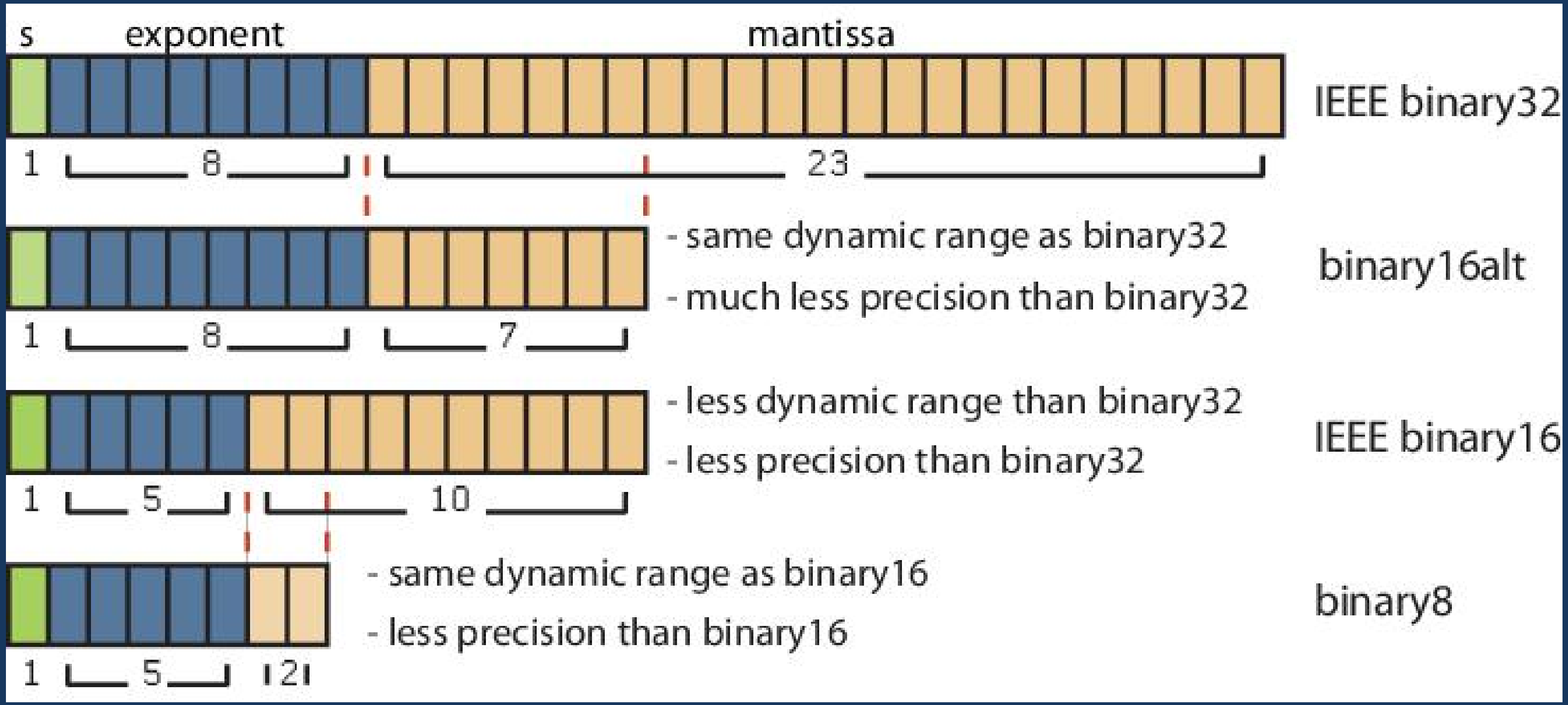
2. Simplifying the deployment of *SmallFloat-based* applications
    - SmallFloat emulation
    - Precision Tuning
    - Automation (compiler support)

Agenda

- ☐ Introduction – Transprecision Computing
- ☐ *Smaller-than-32-bit* floating point types
- ☐ **Implementing the *smallFloat* extension**
  - HW support
  - Compiler support
- ☐ Simplifying the deployment of *SmallFloat-based* applications
- ☐ Conclusion

smallFloat type system

- Preliminary experiments [1] motivate *smaller-than-32-bit* FP types
- Several alternatives are possible. A few useful ones have been defined already.



Some applications require large dynamic range…

…some others require higher precision

[1] Giuseppe Tagliavini, Stefan Mach, Andrea Marongiu, Davide Rossi, Luca Benini
A Transprecision Floating-Point Platform for Ultra-Low Power Computing
In Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1051-1056. IEEE, 2018.

1) Supporting the SmallFloat data type extension

Hardware Support (1): The **PULP** Platform

- Open-source *ultra-low-power* computing platform by **ETH Zürich** and **University of Bologna**
- Based on the open-source **RISC-V** instruction set architecture
  - extensible without breaking official RISC-V support

pulp-platform.org

1) Supporting the SmallFloat data type extension

Hardware Support (2): **Goals for SmallFloat HW**

- ☐ Provide ***smallFloat*** formats in RISCV core
  - Computational operations (ADD, SUB, MUL)
  - Conversions between integers and FP formats, and among FP formats

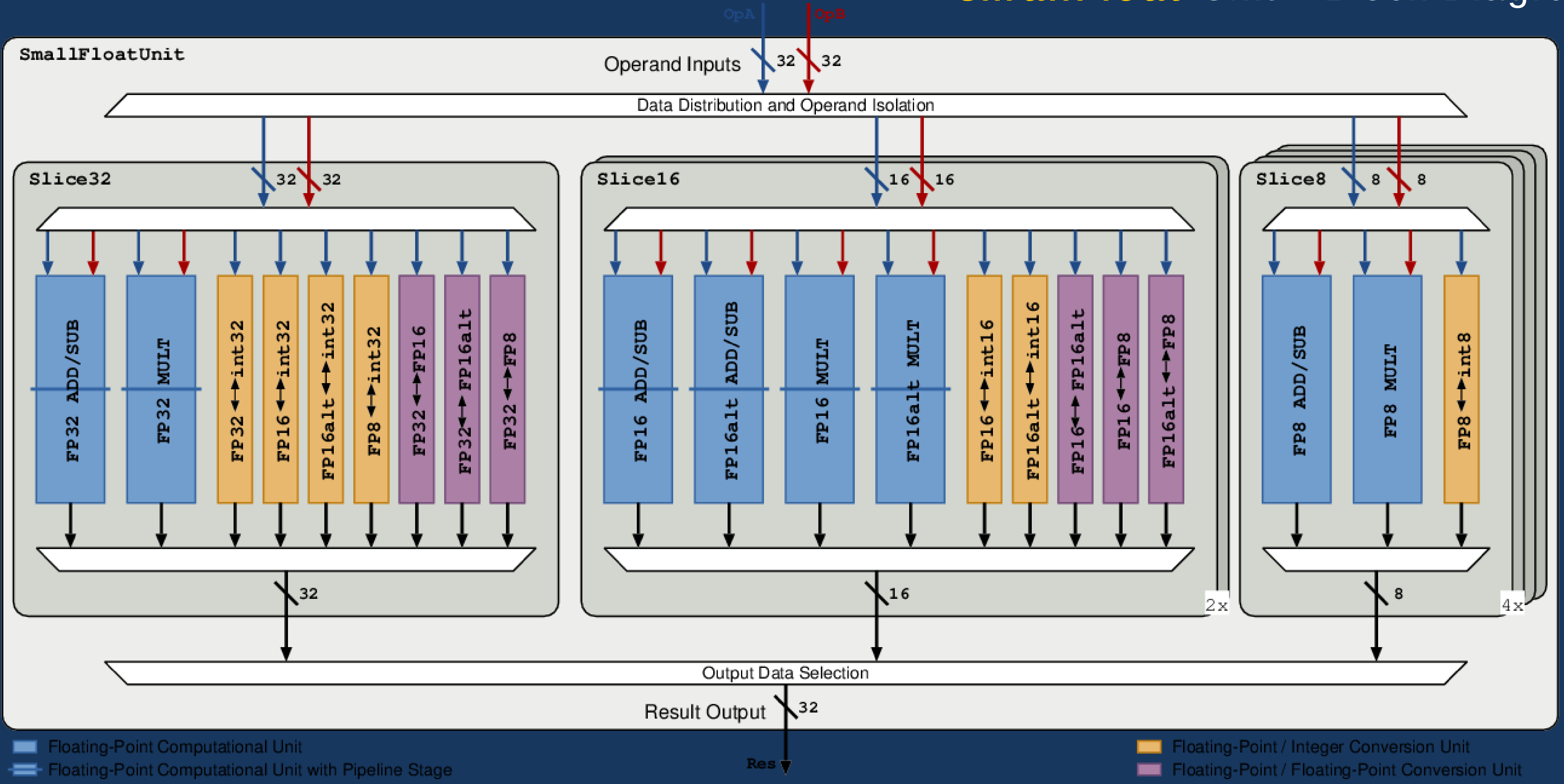
- ☐ **Vectorize** reduced-precision operations – 2x 16bit or 4x 8bit

- ☐ smallFloat operations (16bit, 8bit) and conversions in **single cycle**

- ☐ RISC-V **ISA extensions** to handle new formats/instructions

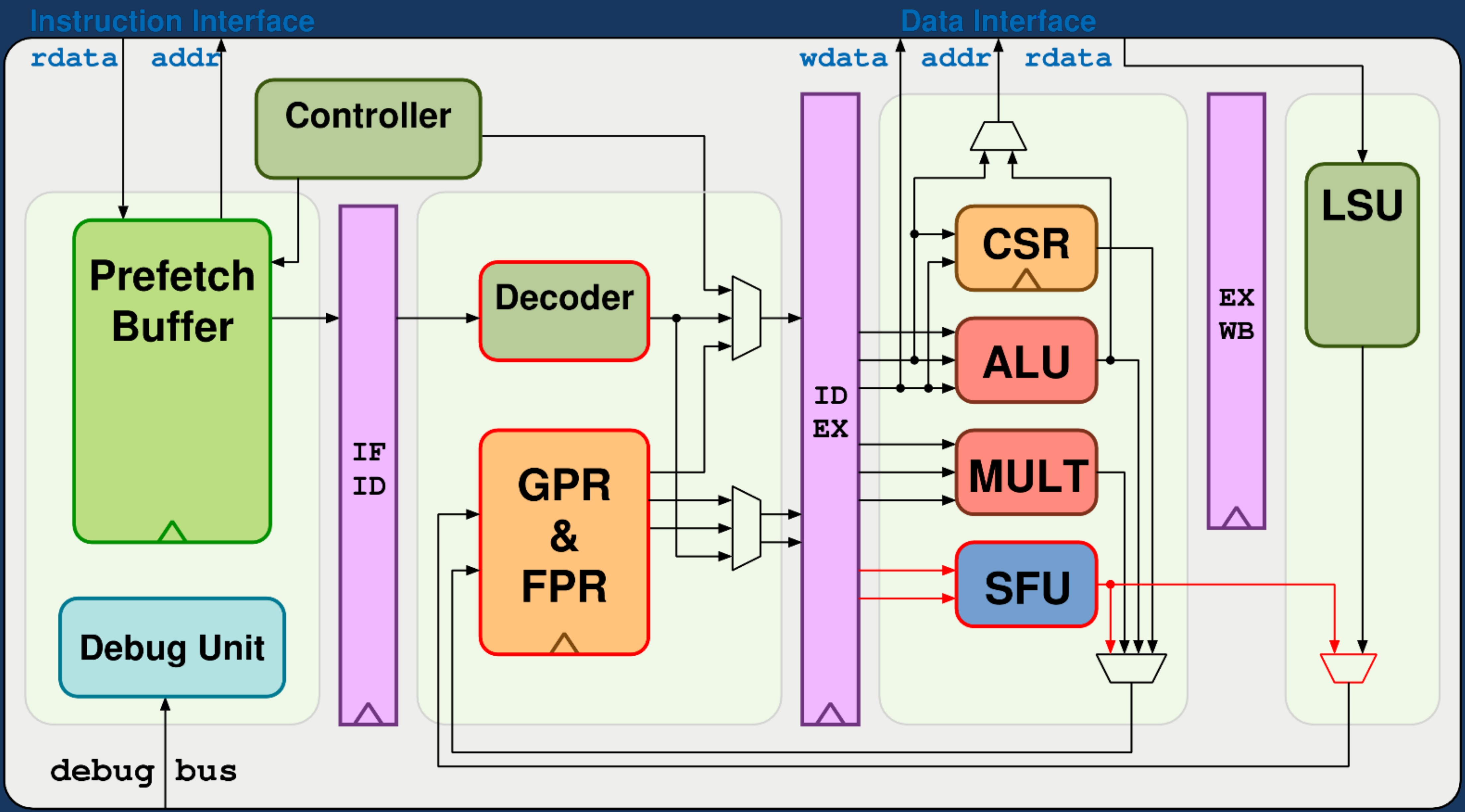
1) Supporting the SmallFloat data type extension

smallFloat Unit – Block Diagram



1) Supporting the SmallFloat data type extension

smallFloat Unit – Core integration



Energy consumption of SmallFloat operations

Format	Operation	Instruction (smallFloat ISA extension)	Energy
	Idle Cycle	nop	62.2 pJ
int32	Data movement	lw,sw	94.4 pJ
	Arithmetic	add,mul	106.4 pJ
float32	Arithmetic	f{add,mul}.s	106.8 pJ
	Conversions	fcvt.s.X	79.7 pJ
float16	Arithmetic	f{add,mul}.h	98.8 pJ
	Conversions	fcvt.h.X	74.7 pJ
	Vector Arithmetic	vf{add,mul}.h	132.6 pJ
	Vector Conversions	vfcvt.h.X	86.4 pJ
float16alt	Arithmetic	f{add,mul}.ah	87.2 pJ
	Conversions	fcvt.ah.x	73.5 pJ
	Vector Arithmetic	vf{add,mul}.ah	108.9 pJ
	Vector Conversions	vfcvt.ah.X	79.5 pJ
float8	Arithmetic	f{add,mul}.b	74.0 pJ
	Conversions	fcvt.b.x	72.5 pJ
	Vector Arithmetic	vf{add,mul}.b	95.2 pJ
	Vector Conversions	vfcvt.b.X	77.8 pJ

Average energy per operation (from post-layout simulations)

Idle System Energy per Cycle

Almost Identical

Energy decreases with fewer mantissa bits

95.2 pJ/4 = 23.8 pJ

UMC 65nm, target @350MHz
Worst-case libraries (1.08V, 125℃)

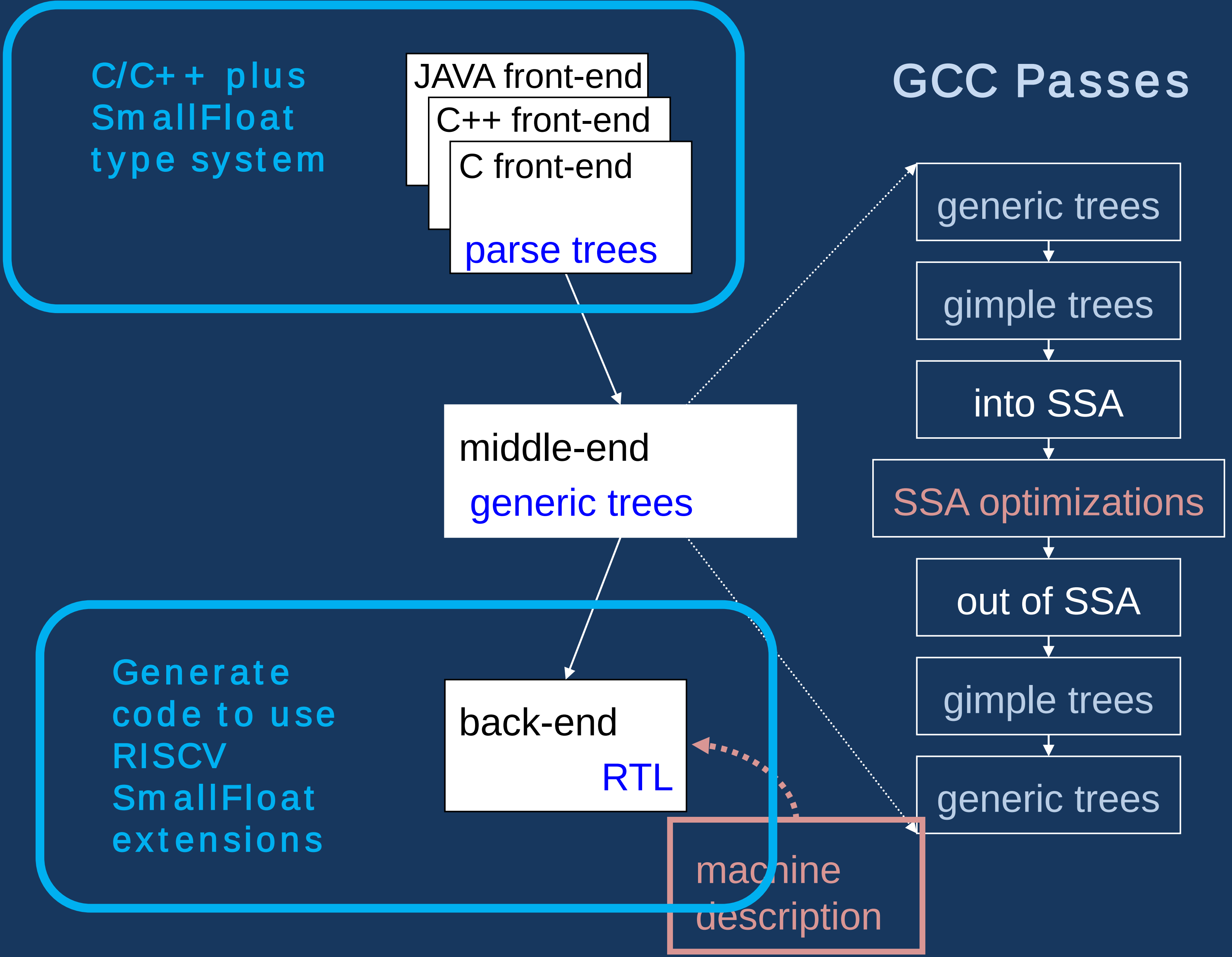
© 2018 OPRECOMP – *http://oprecomp.eu*

1) Supporting the SmallFloat data type extension

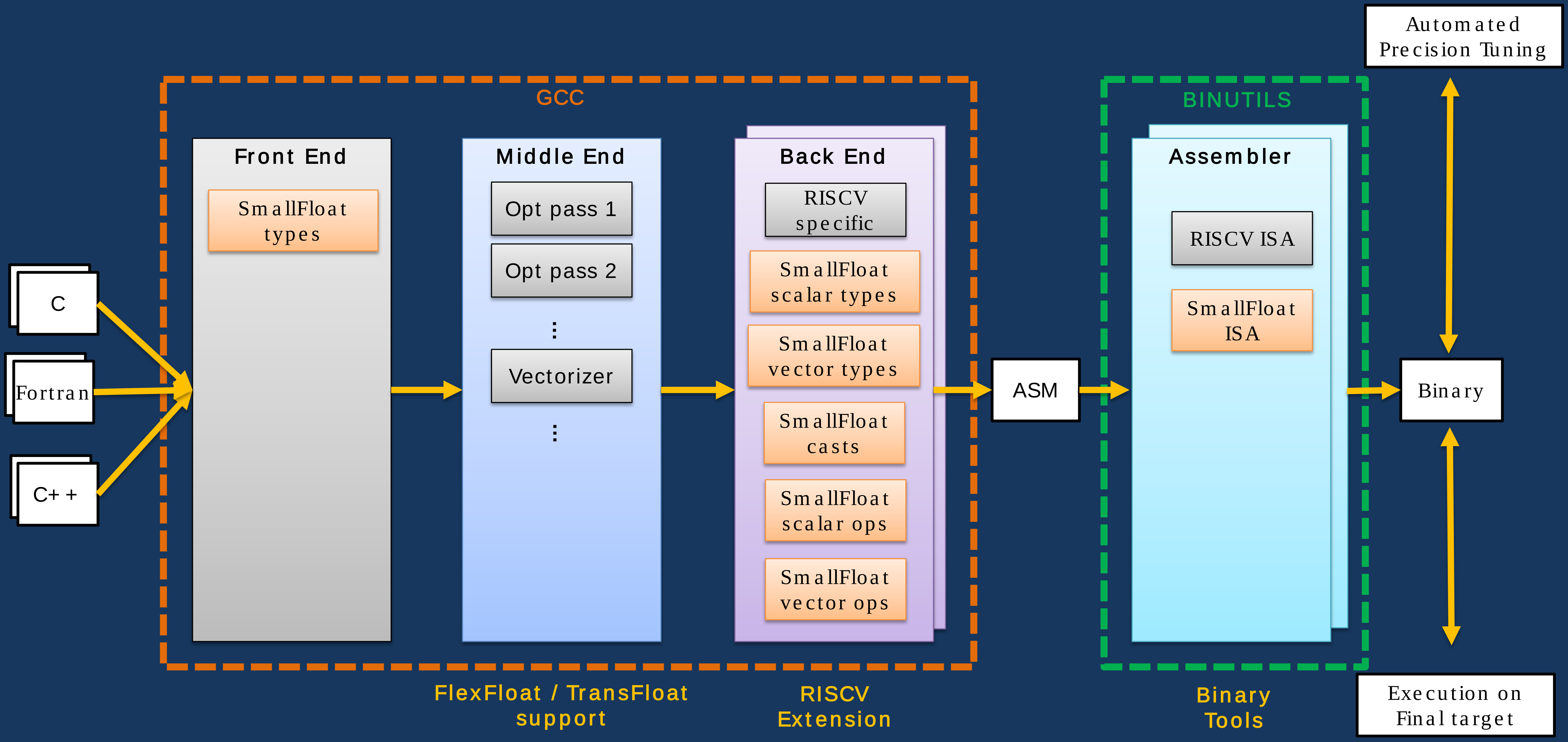
Compiler Support

- ❑ Language type system extension (front-end)
- ❑ ISA extension (back-end)
- ❑ The role of vectorization

Compiler support to the SmallFloat data types



Compiler support to the SmallFloat data types



Compiler support to the SmallFloat data types

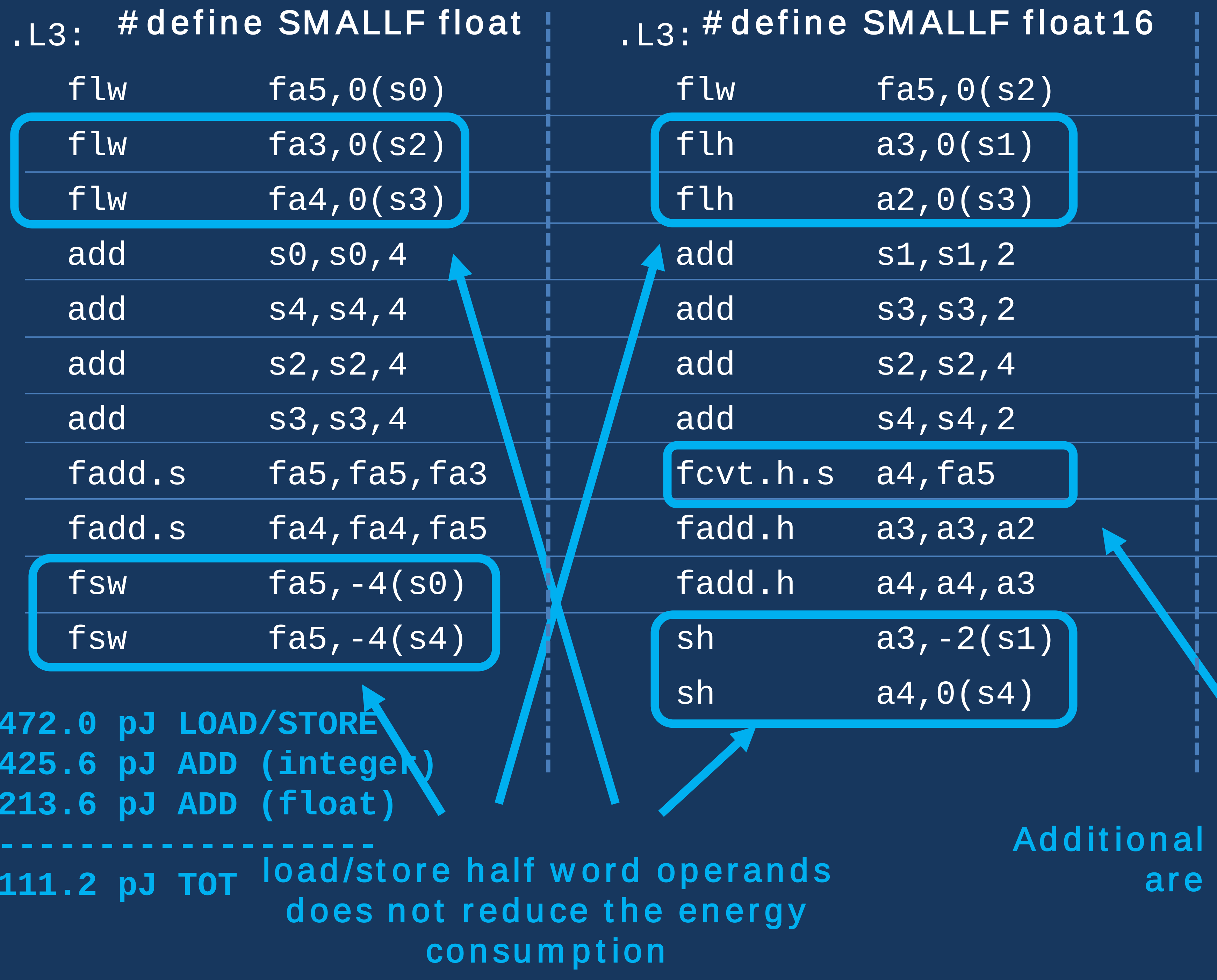
- ❑ Ok, now our compiler understands and handles smallFloat types.
- ❑ Is this sufficient to enable the expected energy savings?

The role of vectorization

LET’S CONSIDER THIS SIMPLE EXAMPLE…

```
int main ()
{
  int i;
  float a[SIZE];
  SMALLF b[SIZE], c[SIZE] d[SIZE];

  for(i = 0; i < SIZE; i++)
  {
     b[i] = b[i] + c[i];
     d[i] = b[i] +  (SMALLF) a[i];
  }
}
```

The role of vectorization



```
.L3: #define SMALLF float
    flw      fa5,0(s0)
    flw      fa3,0(s2)
    flw      fa4,0(s3)
    add      s0,s0,4
    add      s4,s4,4
    add      s2,s2,4
    add      s3,s3,4
    fadd.s   fa5,fa5,fa3
    fadd.s   fa4,fa4,fa5
    fsw      fa5,-4(s0)
    fsw      fa5,-4(s4)
```

```
.L3: #define SMALLF float16
    flw      fa5,0(s2)
    flh      a3,0(s1)
    flh      a2,0(s3)
    add      s1,s1,2
    add      s3,s3,2
    add      s2,s2,4
    add      s4,s4,2
    fcvt.h.s a4,fa5
    fadd.h   a3,a3,a2
    fadd.h   a4,a4,a3
    sh       a3,-2(s1)
    sh       a4,0(s4)
```

```
472.0 pJ LOAD/STORE
425.6 pJ ADD (integer)
213.6 pJ ADD (float)
--------------------
1111.2 pJ TOT
```

load/store half word operands does not reduce the energy consumption

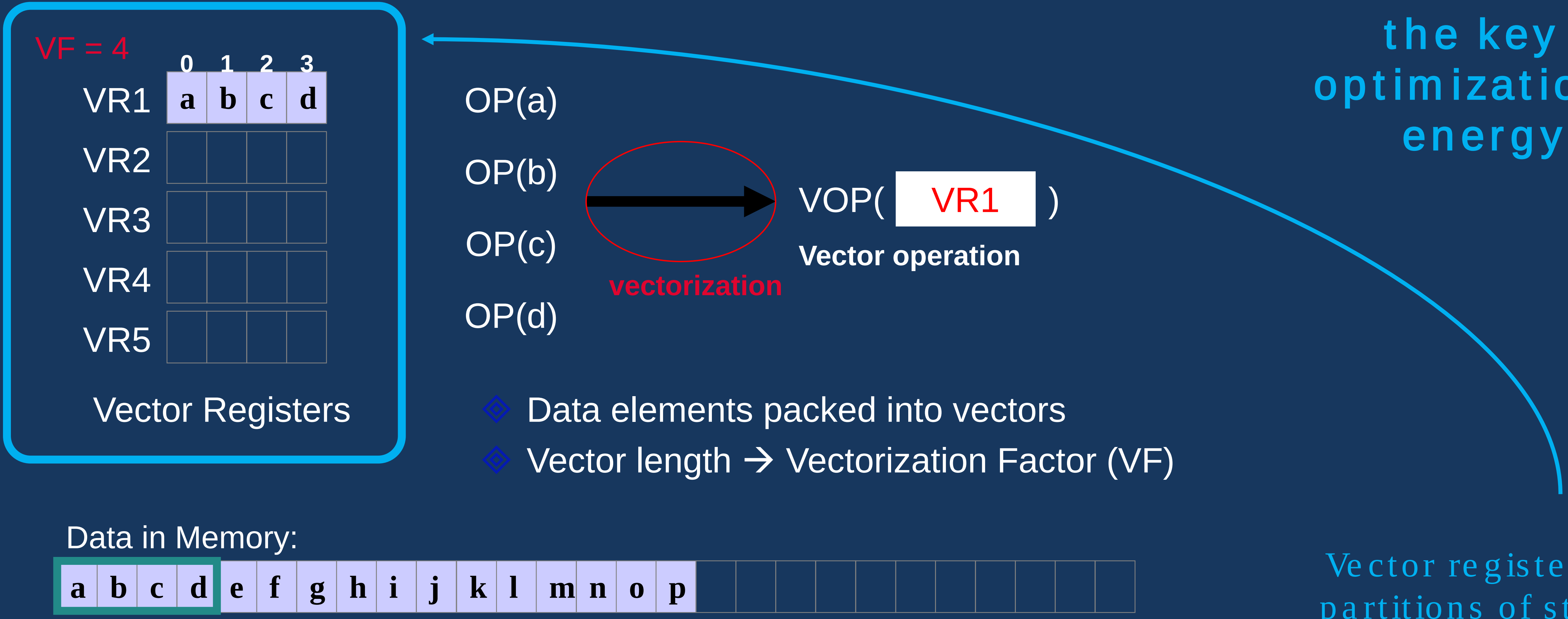
Additional cast operations are required

```
472.0 pJ LOAD/STORE
425.6 pJ ADD (integer)
197.6 pJ ADD (float16)
 74.7 pJ CONV
--------------------
1169.9 pJ TOT
```

Format	Operation	Instruction (smallFloat ISA extension)	Energy
	Idle Cycle	nop	62.2 pJ
int32	Data movement Arithmetic	lw,sw add,mul	94.4 pJ 106.4 pJ
float32	Arithmetic Conversions	f{add,mul}.s fcvt.s.X	106.8 pJ 79.7 pJ
float16	Arithmetic Conversions Vector Arithmetic Vector Conversions	f{add,mul}.h fcvt.h.X vf{add,mul}.h vfcvt.h.X	98.8 pJ 74.7 pJ 132.6 pJ 86.4 pJ
float16alt	Arithmetic Conversions Vector Arithmetic Vector Conversions	f{add,mul}.ah fcvt.ah.x vf{add,mul}.ah vfcvt.ah.X	87.2 pJ 73.5 pJ 108.9 pJ 79.5 pJ
float8	Arithmetic Conversions Vector Arithmetic Vector Conversions	f{add,mul}.b fcvt.b.x vf{add,mul}.b vfcvt.b.X	74.0 pJ 72.5 pJ 95.2 pJ 77.8 pJ

The role of vectorization

How does automatic vectorization work?



VF = 4

0 1 2 3

VR1 a b c d

VR2

VR3

VR4

VR5

Vector Registers

OP(a)

OP(b)

OP(c)

OP(d)

vectorization

VOP(VR1)

Vector operation

Data in Memory:

a b c d e f g h i j k l m n o p

- Data elements packed into vectors
- Vector length → Vectorization Factor (VF)

Automatic vectorization is the key compiler optimization to enable energy savings

Vector registers are logical partitions of standard 32bit registers in the ***smallFloat*** extension

The role of vectorization

How does automatic vectorization work?

Automatic vectorization is the key compiler optimization to enable energy savings

- original serial loop:

```
for(i=0; i<N; i++){
   a[i] = a[i] + b[i];
}
```

- loop in vector notation:

```
for (i=0; i<N; i+=VF){
   a[i:i+VF] = a[i:i+VF] + b[i:i+VF];
}
```

vectorization

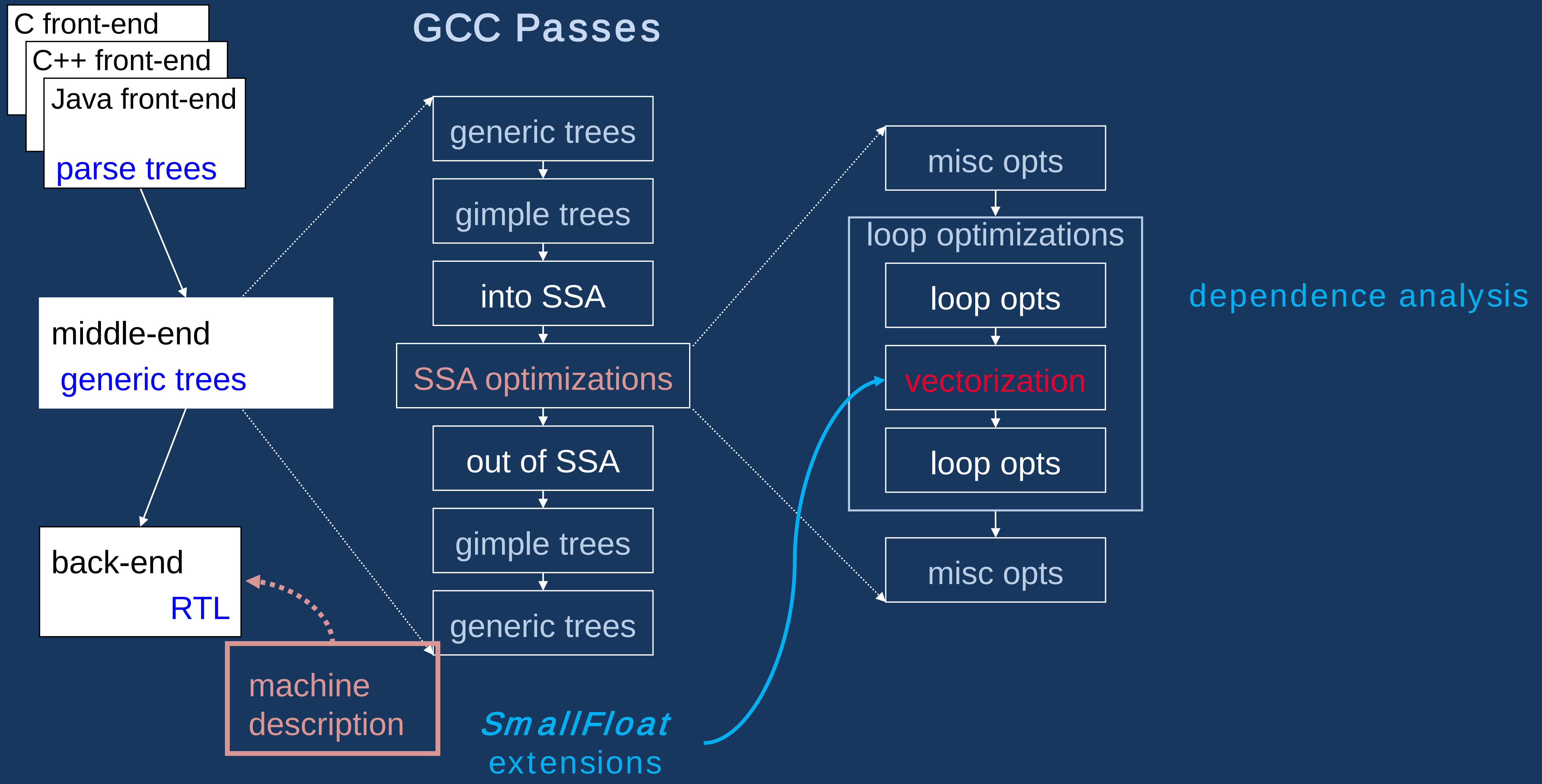
- loop in vector notation:

```
for (i=0; i<(N-N%VF); i+=VF){
   a[i:i+VF] = a[i:i+VF] + b[i:i+VF];
}  vectorized loop

for ( ; i < N; i++) {
   a[i] = a[i] + b[i];
}   epilog loop
```

- Loop based vectorization
- **No dependences between iterations**

© 2018 OPRECOMP – *http://oprecomp.eu*

The role of vectorization



GCC Passes

C front-end
C++ front-end
Java front-end
parse trees

middle-end
generic trees

back-end
RTL

machine
description

generic trees
gimple trees
into SSA
SSA optimizations
out of SSA
gimple trees
generic trees

misc opts
loop optimizations
loop opts
vectorization
loop opts
misc opts

dependence analysis

SmallFloat
extensions

The role of vectorization

```
.L3:
    flw       fa5,0(s0)
    flw       fa3,0(s2)
    flw       fa4,0(s3)
    add       s0,s0,4
    add       s4,s4,4
    add       s2,s2,4
    add       s3,s3,4
    fadd.s    fa5,fa5,fa3
    fadd.s    fa4,fa4,fa5
    fsw       fa5,-4(s0)
    fsw       fa5,-4(s4)
```

1111.2 pJ (iter) *
1024 iters = 1138 nJ

```
.L3:
    flw       fa5,0(s2)
    flh       a3,0(s1)
    flh       a2,0(s3)
    add       s1,s1,2
    add       s3,s3,2
    add       s2,s2,4
    add       s4,s4,2
    fcvt.h.s  a4,fa5
    fadd.h    a3,a3,a2
    fadd.h    a4,a4,a3
    sh        a3,-2(s1)
    sh        a4,0(s4)
```

1169.9 pJ (iter) *
1024 iters = 1198 nJ

```
.L3:
    lw          a0,0(s4)
    lw          a4,0(s6)
    flw         fa4,8(s5!)
    flw         fa5,8(a1!)
    add         s6,s6,4
    add         a3,a3,4
    add         s4,s4,4
    vfcpka.h.s  a5,fa4,fa5
    vfadd.h     a4,a4,a0
    vfadd.h     a5,a5,a4
    sw          a4,-4(s4)
    sw          a5,0(a3)
```

```
 566.4 pJ LOAD/STORE
 319.2 pJ ADD (integer)
 265.2 pJ vADD (float16)
  86.4 pJ CONV
---------------------
1237.2 pJ (iteration) *
 512 iterations = 633.5 nJ
```

Agenda

- ☐ Introduction – Transprecision Computing
- ☐ *Smaller-than-32-bit* floating point types
- ☐ Implementing the *smallFloat* extension
  - HW support
  - Compiler support
- ☐ **Simplifying the deployment of *SmallFloat-based* applications**
- ☐ Conclusion

Simplifying the deployment of *SmallFloat*-based applications

2) How to simplify deployment of applications with *smaller-than-32-bit* floats?

❑ **Fine-grained tuning of FP types for program variables**
- to enable exploration of precision requirements in applications (*)

❑ **Emulation of arbitrary FP types (*SmallFloat*)**
- to enable exploration of precision requirements in applications (*)

❑ **Automation**
- Compilation toolchain for transprecision computing

(*) also to steer the definition of HW extensions (in early stages)

Precision Tuning of FP variables

- ❑ Programs are written using **standard FP formats**
  - ❑ C/C++ programs → **float** and **double** variables

- ❑ **Precision tuning** → transforming programs by changing default FP types to introduce smaller ones
  - Manually
  - Semi-automaticaly
  - Automatically

- ❑ Research papers and open source tools are available…

SOA of precision tuning

- ☐ Statistical methods
  - Source-to-source transform [1]
  - Compiler IR language [2]
  - Binary instrumentation [3]

Dynamic analysis

- ☐ Exact methods
  - Formal theorem proof [4]
  - Branch and bound methods [5]

Static analysis

- ☐ Exact methods have a severe limitation → **applied to a single expression, not to a whole program**

[1] *Ho, Nhut-Minh, Elavarasi Manogaran, Weng-Fai Wong, and Asha Anoosheh. "Efficient floating point precision tuning for approximate computing." In ASP-DAC 2017, pp. 63-68. IEEE*

[2] *Rubio-González, Cindy, et al.. "Precimonious: Tuning assistant for floating-point precision." In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 27. ACM, 2013*

[3] *Lam, Michael O., and Barry L. Rountree. "Floating-point shadow value analysis." In Proceedings of the 5th Workshop on Extreme-Scale Programming Tools, pp. 18-25. IEEE Press, 2016*

[4] *Website: http://precisa.nianet.org/*

[5] *Chiang, Wei-Fan, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić.*

"Rigorous floating-point mixed-precision tuning." In SIGPLAN 2017, pp. 300-315. ACM

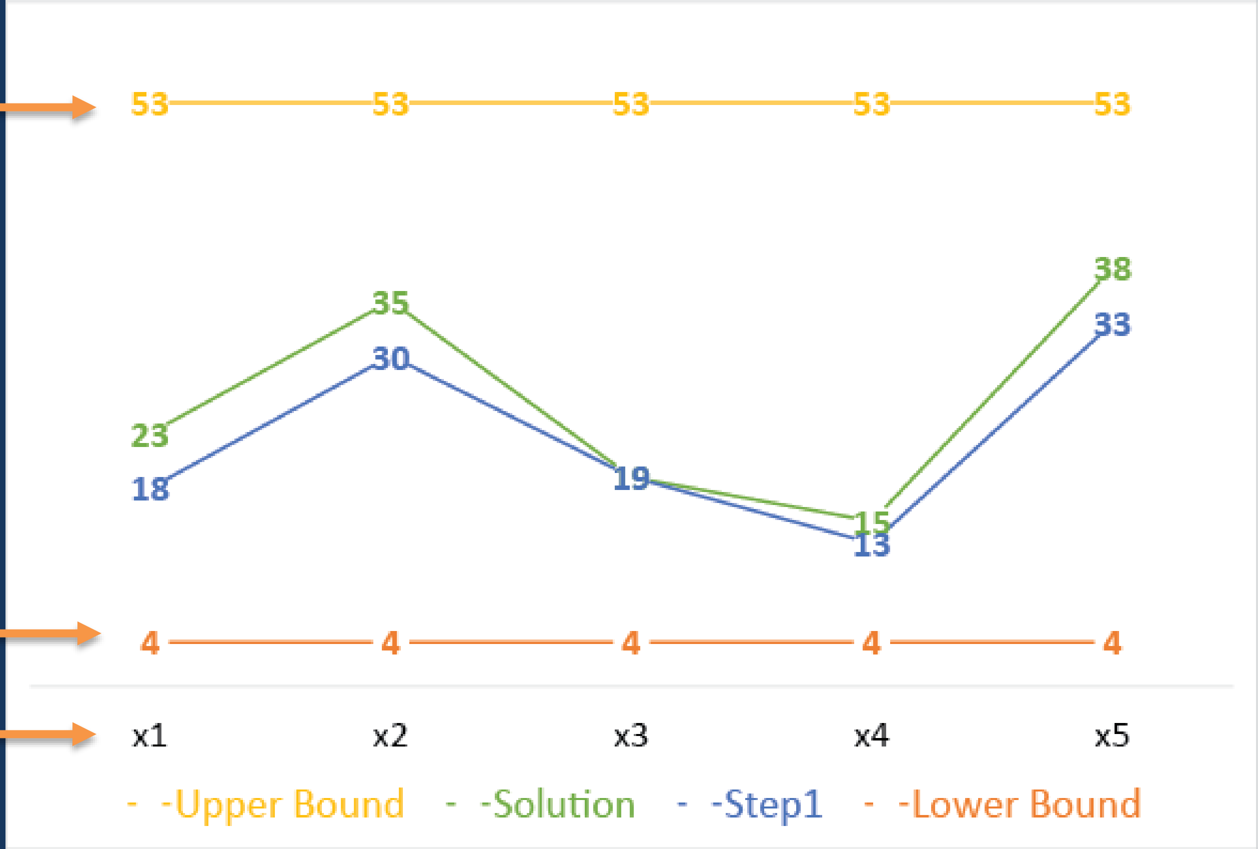
Precision tuning of FP variables

- ❑ Preliminary experiments using a statistical method → **fpPrecisionTuning**
  - FP types/operators instrumented to GNU MPFR structs/functions
  - **Tuning process:** heuristic search in **P^n** space (**n** is the **number of variables**, **P** is the set of available **precisions**) → Multiple executions with different values of precision associated to variables
    → Iterative refinement of the solution for different values of input variables

Double precision (53 mantissa bits)

Minimum precision for exploration

Variables



Experiments: Single-precision and half-precision

Relative error on program results

ε	Application	Precision (mantissa bits)	
		3-11	12-23
10^{-6}	HOG	0%	100%
	KNN	0%	100%
	PCA	0%	100%
	DWT	0%	100%
	SVM	0%	100%
	CONV	0%	100%
10^{-4}	HOG	0%	100%
	KNN	0%	100%
	PCA	0%	100%
	DWT	0%	100%
	SVM	0%	100%
	CONV	0%	100%
10^{-1}	HOG	0%	100%
	KNN	0%	100%
	PCA	0%	100%
	DWT	0%	100%
	SVM	0%	100%
	CONV	0%	100%

Single precision

ε	Application	Precision (mantissa bits)	
		3-11	12-23
10^{-6}	HOG	50%	50%
	KNN	50%	50%
	PCA	91%	9%
	DWT	100%	0%
	SVM	100%	0%
	CONV	50%	50%
10^{-4}	HOG	50%	50%
	KNN	100%	0%
	PCA	100%	0%
	DWT	100%	0%
	SVM	100%	0%
	CONV	50%	50%
10^{-1}	HOG	50%	50%
	KNN	100%	0%
	PCA	100%	0%
	DWT	100%	0%
	SVM	100%	0%
	CONV	100%	0%

Single precision + half precision

50% single to half

100% single to half

HALF PRECISION
1 bit sign
5 bits exponent
10(+1) bits mantissa

90% single to half (on average)

Percentage of variables after tuning

Experiments: Single-precision, half-precision and quarter-precision

ε	Application	Precision (mantissa bits)	
		3-11	12-23
10^{-6}	HOG	50%	50%
	KNN	50%	50%
	PCA	91%	9%
	DWT	100%	0%
	SVM	100%	0%
	CONV	50%	50%
10^{-4}	HOG	50%	50%
	KNN	100%	0%
	PCA	100%	0%
	DWT	100%	0%
	SVM	100%	0%
	CONV	50%	50%
10^{-1}	HOG	50%	50%
	KNN	100%	0%
	PCA	100%	0%
	DWT	100%	0%
	SVM	100%	0%
	CONV	100%	0%

Single precision + half precision

100% half to quarter

60% half to quarter (on avg)

ε	Application	Precision (mantissa bits)		
		3	4-11	12-23
10^{-6}	HOG	50%	0%	50%
	KNN	0%	50%	50%
	PCA	0%	91%	9%
	DWT	0%	100%	0%
	SVM	100%	0%	0%
	CONV	0%	50%	50%
10^{-4}	HOG	0%	50%	50%
	KNN	100%	0%	0%
	PCA	0%	100%	0%
	DWT	0%	100%	0%
	SVM	100%	0%	0%
	CONV	0%	50%	50%
10^{-1}	HOG	50%	0%	50%
	KNN	100%	0%	0%
	PCA	0%	100%	0%
	DWT	0%	100%	0%
	SVM	100%	0%	0%
	CONV	100%	0%	0%

Single precision + half precision + quarter precision

QUARTER PRECISION
1 bit sign
5 bits exponent
2(+1) bits mantissa

Experiments: Single-precision, 2x half-precision and quarter-precision

ε	Application	Precision (mantissa bits)		
		3	4-11	12-23
10^{-6}	HOG	50%	0%	50%
	KNN	0%	50%	50%
	PCA	0%	91%	9%
	DWT	0%	100%	0%
	SVM	100%	0%	0%
	CONV	0%	50%	50%
10^{-4}	HOG	0%	50%	50%
	KNN	100%	0%	0%
	PCA	0%	100%	0%
	DWT	0%	100%	0%
	SVM	100%	0%	0%
	CONV	0%	50%	50%
10^{-1}	HOG	50%	0%	50%
	KNN	100%	0%	0%
	PCA	0%	100%	0%
	DWT	0%	100%	0%
	SVM	100%	0%	0%
	CONV	100%	0%	0%

Single precision + half precision + quarter precision

100%
single
to alt half

Almost 100%
single & half
to alt half

ε	Application	Precision (mantissa bits)			
		3	4-8	9-11	12-23
10^{-6}	HOG	50%	0%	0%	50%
	KNN	0%	100%	0%	0%
	PCA	0%	1%	91%	9%
	DWT	0%	0%	100%	0%
	SVM	100%	0%	0%	0%
	CONV	0%	100%	0%	0%
10^{-4}	HOG	0%	50%	0%	50%
	KNN	100%	0%	0%	0%
	PCA	0%	100%	0%	0%
	DWT	0%	0%	100%	0%
	SVM	100%	0%	0%	0%
	CONV	0%	100%	0%	0%
10^{-1}	HOG	50%	0%	0%	50%
	KNN	100%	0%	0%	0%
	PCA	0%	100%	0%	0%
	DWT	0%	100%	0%	0%
	SVM	100%	0%	0%	0%
	CONV	100%	0%	0%	0%

Single precision + half precision + quarter precision + alternative half precision

ALTERNATIVE HALF PRECISION
1 bit sign
8 bits exponent
7(+1) bits mantissa

FlexFloat: Fast emulation of *SmallFloat* types

- Emulation library to test *less-than-32-bit* types - flexible, but performance-efficient, too
- Low-level interface (e.g., explicit casts, only binary operations)
- Full support to IEEE 754 concepts
- Intended for **integration within automatic tools**

Reference C code

```c
double a, b, c;
a = 10.4;
b = 11.5;

c = a + b;
printf("[result] c = %f\n", c);
```

FlexFloat C transformed code

```c
flexfloat_t a, b, c;
ff_init_double(&a, 10.4, (prec_t) {11, 52});
ff_init_double(&b, 11.5, (prec_t) {11, 52});
ff_init(&c, (prec_t) {11, 52});
ff_add(&c, &a, &b);
printf("[printf] c = %f\n", ff_get_double(&c));
```

`flexfloat_t` → FlexFloat type

`prec_t` → Format descriptor

```c
typedef struct
{
  unsigned int mant_bw;
  unsigned int exp_bw;
}
prec_t;
```

```c
typedef struct
{
  prec_t prec;
  double value;
}
flexfloat_t;
```

FlexFloat: Fast emulation of *SmallFloat* types

- Emulation library to test *less-than-32-bit* types - flexible, but performance-efficient, too
- Low-level interface (e.g., explicit casts, only binary operations)
- Full support to IEEE 754 concepts
- Intended for **integration within automatic tools**

Reference C code

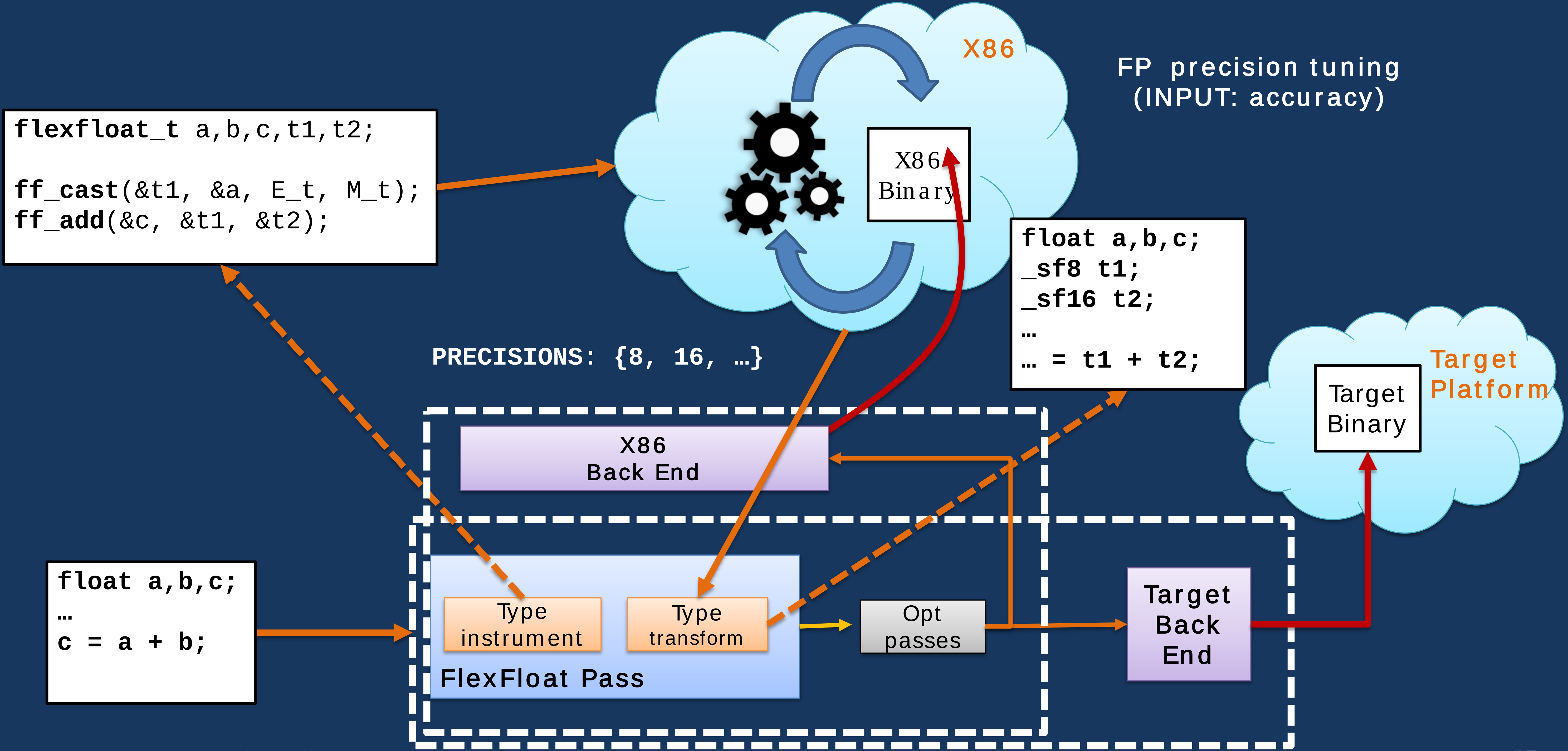
```c
double a, b, c;
a = 10.4;
b = 11.5;

c = a + b;
printf("[result] c = %f\n", c);
```

FlexFloat C transformed code

```c
flexfloat_t a, b, c;
ff_init_double(&a, 10.4, (prec_t) {11, 52});
ff_init_double(&b, 11.5, (prec_t) {11, 52});
ff_init(&c, (prec_t) {11, 52});
ff_add(&c, &a, &b);
printf("[printf] c = %f\n", ff_get_double(&c));
```

`flexfloat_t` → FlexFloat type
`prec_t` → Format descriptor
`ff_init` → Initialize a FlexFloat variable with a specific format
`ff_init_<float/double>` → Initialize a FlexFloat variable with a format and a float/double value
`ff_add, ff_sub,…` → Perform arithmetic operations
`ff_get_<float/double>` → Convert to standard FP types

Automation: integration with compilation toolchain



```
flexfloat_t a,b,c,t1,t2;

ff_cast(&t1, &a, E_t, M_t);
ff_add(&c, &t1, &t2);
```

X86

X86 Binary

FP precision tuning
(INPUT: accuracy)

```
float a,b,c;
_sf8 t1;
_sf16 t2;
…
… = t1 + t2;
```

PRECISIONS: {8, 16, …}

Target Platform

Target Binary

X86 Back End

```
float a,b,c;
…
c = a + b;
```

Type instrument

Type transform

Opt passes

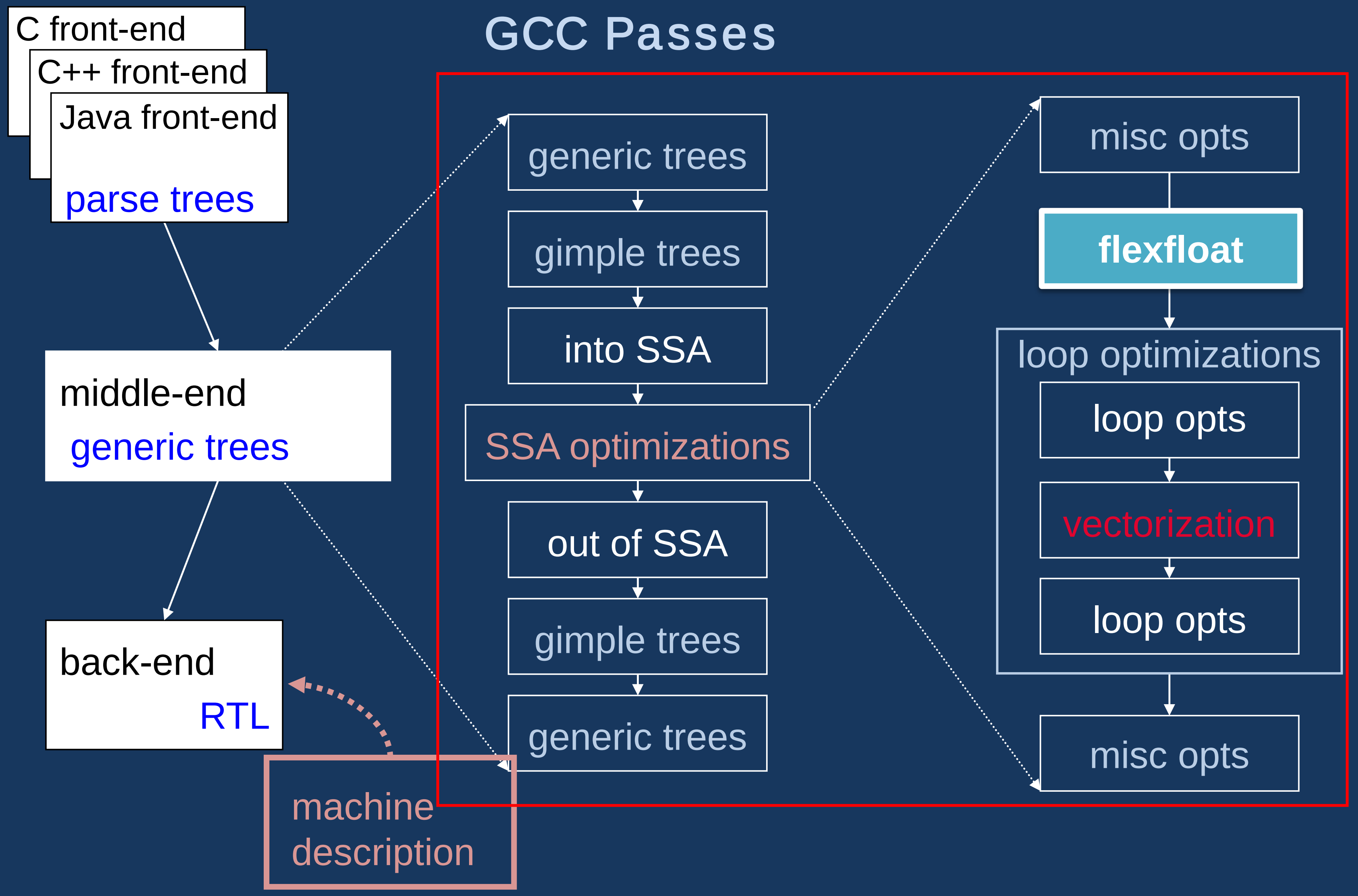
FlexFloat Pass

Target Back End

© 2018 OPRECOMP – *http://oprecomp.eu*

Automation: integration with compilation toolchain

- ❑ Manual program instrumentation with FlexFloat primitives can be a tedious and error-prone task
- ❑ Might want to hide the process as part of the compilation toolchain

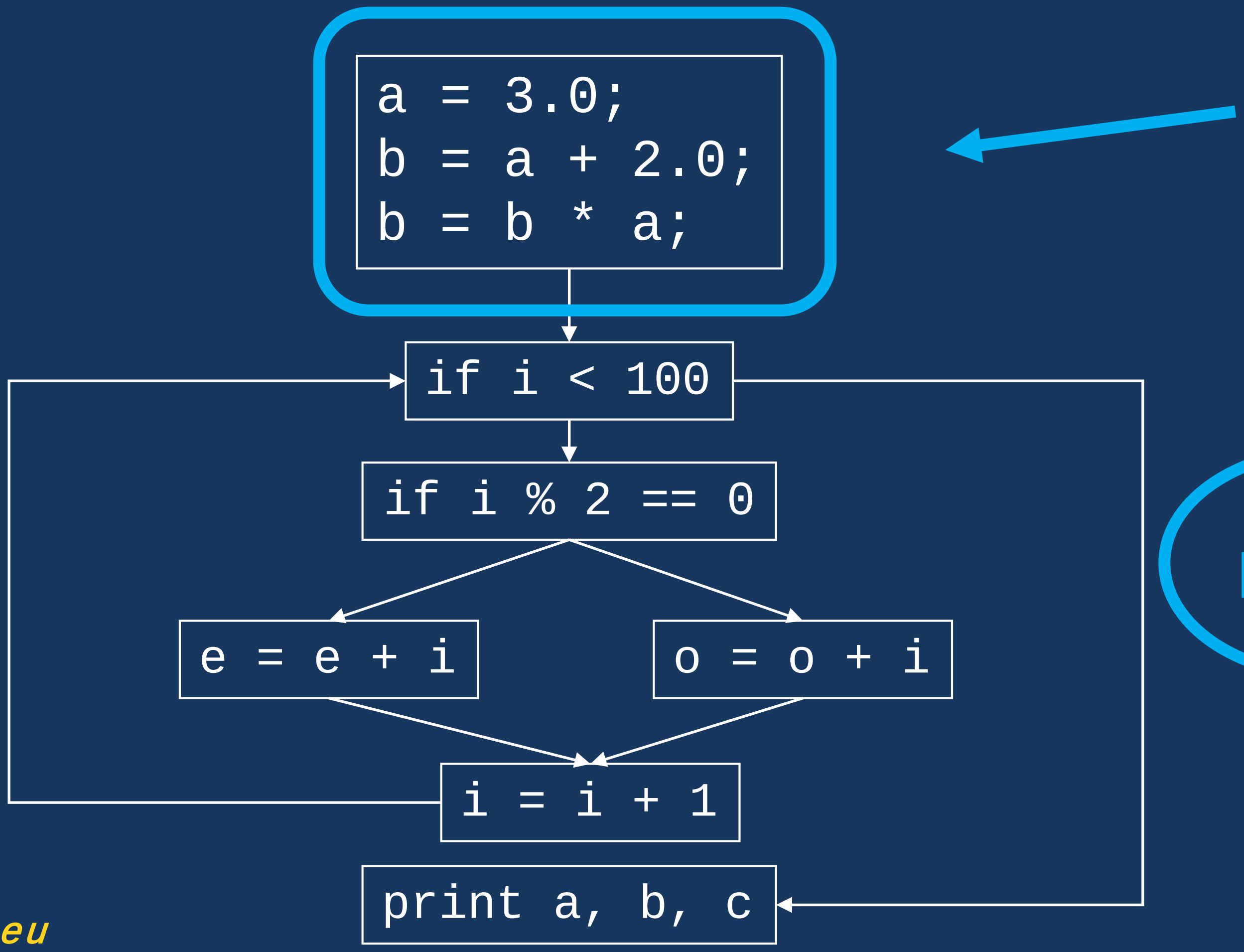


How does *FlexFloat* instrumentation work?

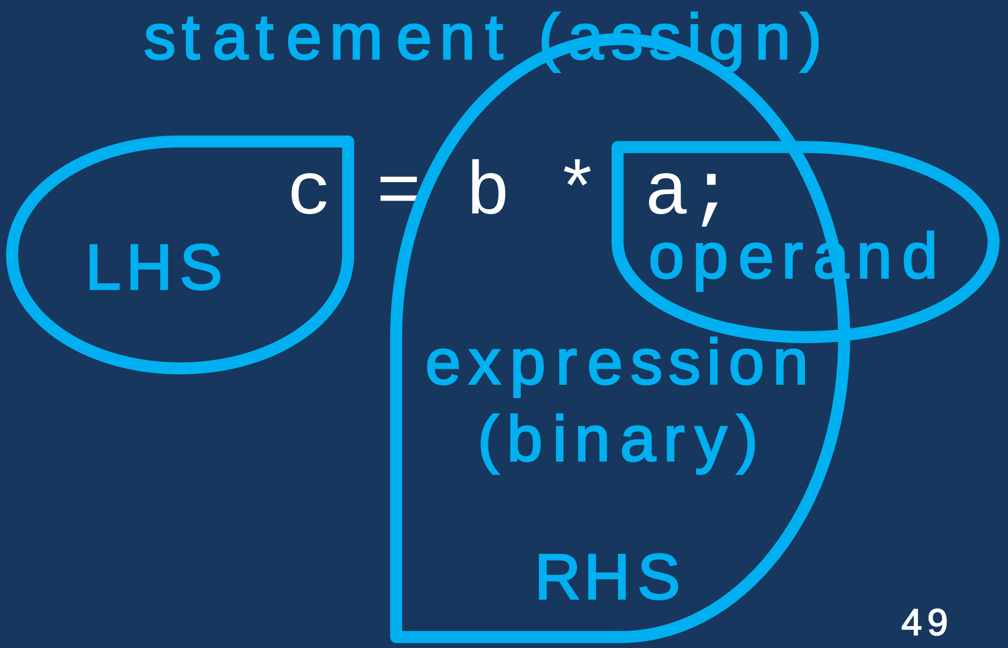
1. Implemented on top of the Single Static Assignment (SSA) form
   - allows to reason at fine granularity (GIMPLE statements only allow simple expressions with up to 3 operands)

```
a = 3.0;
b = a + 2.0;
b = b * a;
i = 0;

while (i<100)
    if (i % 2 == 0)
        c = a + b;
    else
        c = a – b;
    i = i + 1;

print a, b, c;
```

CONTROL FLOW GRAPH (CFG)



Basic block



How does *FlexFloat* instrumentation work?

1. Implemented on top of the Single Static Assignment (SSA) form
   - allows to reason at fine granularity (GIMPLE statements only allow simple expressions with up to 3 operands)

```
a = 3.0;
b = a + 2.0;
b = b * a;
```

```
a1 = 3.0;
b1 = a1 + 2.0;
b2 = b1 * a1;
```

SSA is a transformed program
- Whose variables are renamed
  - *e.g. x --> x_i*
- Having only one definition for each variable
- Without changing the semantics of the original program
  - *i.e. every renamed variable x_i of x must have the same value for every possible control flow path*

More compact def-use chain
Control flow becomes explicit on variable names
Improves performance of many data-flow analyses

How does *FlexFloat* instrumentation work?

2. Statements are walked and uses of REAL-TYPE variables are instrumented

```
FOR EACH BASIC BLOCK BBi
    FOR EACH STATEMENT Sj
        IF (Sj contains REAL-TYPE operands)
            create FF alias for LHS (LHS-FF-alias)
            create precision variable for statment Sj (Psj)

            FOR EACH USE of LHS
                set-FF-alias (USEk)  // mark USE k with
                                     // defining FF alias

            switch (EXPR (RHS))
                case UNARY-EXPR:
                        handle-unary-expr
                case BINARY-EXPR:
                        handle-binary-expr
                case TERNARY-EXPR:

            remove Sj

        ← ENDIF
```

handle-unary-expr

```
If (is-real-const (RHS)
    emit FF-INIT< REAL-TYPE>  (&LHS-FF-alias, RHS, Psj)

else
    create FF alias for RHS (RHS-FF-alias)
    emit FF-CAST (&LHS-FF-alias, &RHS-FF-alias, Psj)
```

handle-binary-expr

```
create FF alias for OP1 (OP1-FF-alias)
create FF alias for OP2 (OP2-FF-alias)

emit FF-CAST (&OP1-FF-alias, get-FF-alias (OP1), Psj)
emit FF-CAST (&OP2-FF-alias, get-FF-alias (OP2), Psj)


switch (EXPR (RHS))
    case PLUS-EXPR:
    case MULT-EXPR:
emit FF-< EXPR>  (&LHS-FF-alias, &OP1-FF-alias, &OP2-FF-alias)
```

How does *FlexFloat* instrumentation work?

A SIMPLE EXAMPLE…

```
int main ()
{
  double a, b, c;

  a = 3.0;
  b = a + 2.0;
  c = b * a;

  return c;
}
```

…and its (GIMPLE) SSA representation

```
int main ()
{
  double a1, b1, c1;
  double t1;

BB 1:
  a1 = 3.0;
  t1 = 2.0;
  b1 = a1 + t1;
  c1 = b1 * a1;

  return c1;
}
```

How does *FlexFloat* instrumentation work?

A SIMPLE EXAMPLE…

```
int main ()
{
  double a1, b1, c1;
  double t1;
  flexfloat_t ff_a1;
  prec_t p_s1;

  a1 = 3.0;
  t1 = 2.0;
  b1 = a1 + 2.0;
  c1 = b1 * a1;

  return c1;
}
```

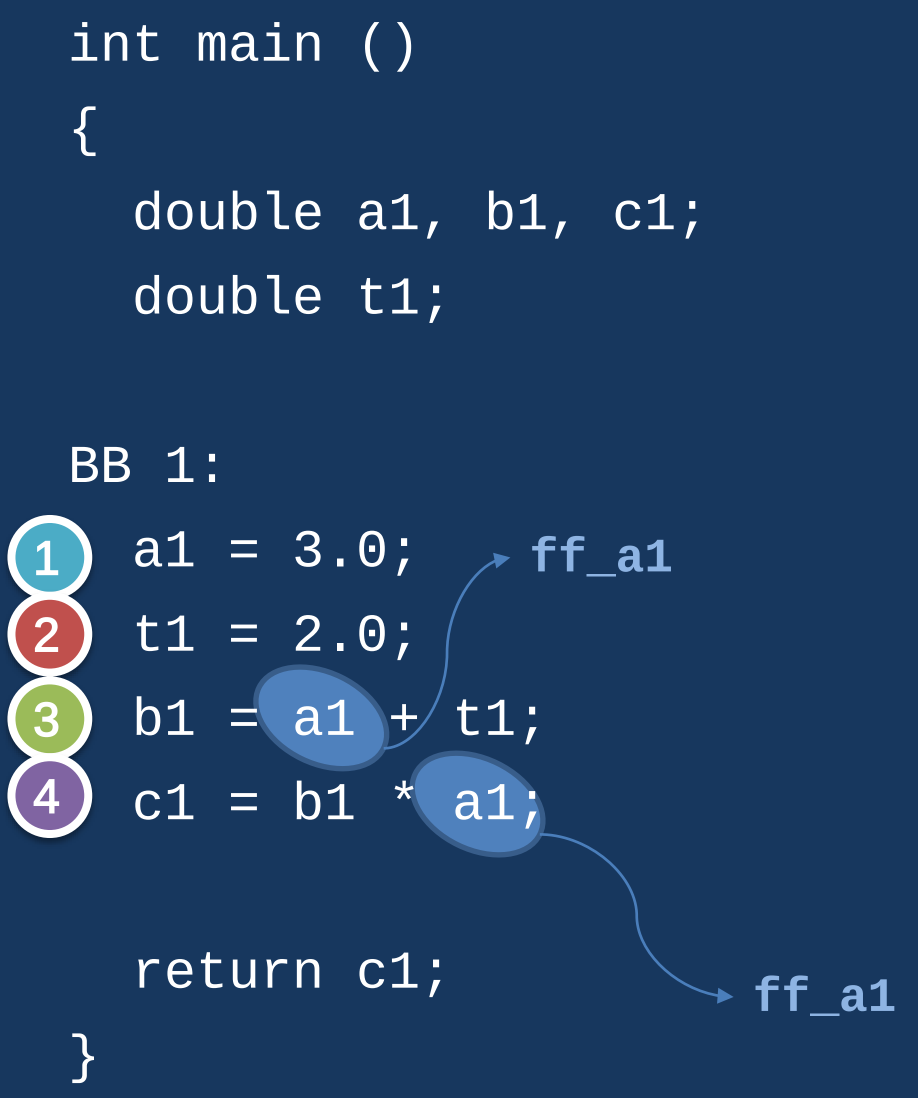
We start by walking BBs and statements therein

ff_a1

① a1 = 3.0;

```
IF (Sj contains REAL-TYPE operands)
        create FF alias for LHS (LHS-FF-alias)
        create precision variable for statment Sj (Psj)
```

```
FOR EACH USE of LHS
        set-FF-alias (USEk)  // mark USE  k with
                             // defining FF alias
```



```
int main ()
{
  double a1, b1, c1;
  double t1;

BB 1:
① a1 = 3.0;
② t1 = 2.0;
③ b1 = a1 + t1;
④ c1 = b1 * a1;

  return c1;
}
```

How does *FlexFloat* instrumentation work?

A SIMPLE EXAMPLE…

```
int main ()
{
  double ~~a1~~, b1, c1;
  double t1;
  flexfloat_t ff_a1;
  prec_t p_s1;

~~a1 = 3.0;~~
  ff_init (&ff_a1, 3.0, p_s1);
  t1 = 2.0;
  b1 = a1 + 2.0;
  c = b * a;

  return c1;
}
```

We start by walking BBs and statements therein

ff_a1

① a1 = 3.0;

```
switch (EXPR (RHS))
        case UNARY-EXPR:
                handle-unary-expr
```

```
If (is-real-const (RHS)
    emit FF-INIT< REAL-TYPE>  (&LHS-FF-alias, RHS, Psj)

                remove Sj
```

```
int main ()
{
  double a1, b1, c1;
  double t1;

BB 1:
  a1 = 3.0;
  t1 = 2.0;
  b1 = a1 + t1;
  c1 = b1 * a1;

  return c1;
}
```

① ② ③ ④

ff_a1

ff_a1

How does *FlexFloat* instrumentation work?

A SIMPLE EXAMPLE...

similar process

```
int main ()
{
  double b1, c1;
  double ~~t1~~;
  flexfloat_t ff_a1, ff_t1;
  prec_t p_s1, p_s2;

  ff_init (&ff_a1, 3.0, p_s1);
~~  t1 = 2.0;~~
  ff_init (&ff_t1, 2.0, p_s2);
  b1 = a1 + 2.0;
  c = b * a;

  return c1;
}
```

ff_t1

② t1 = 2.0;

```
create FF alias for LHS (LHS-FF-alias)
create precision variable for statment Sj (Psj)

FOR EACH USE of LHS
    set-FF-alias (USEk)  // mark USE  k with
                         // defining FF alias

If (is-real-const (RHS)
    emit FF-INIT< REAL-TYPE>  (&LHS-FF-alias, RHS, Psj)

remove Sj
```

```
int main ()
{
  double a1, b1, c1;
  double t1;

BB 1:
  a1 = 3.0;
  t1 = 2.0;
  b1 = a1 + t1;
  c1 = b1 * a1;

  return c1;
}
```

① ② ③ ④

ff_t1

How does *FlexFloat* instrumentation work?

A SIMPLE EXAMPLE...

```
int main ()
{
  double b1, c1;
  flexfloat_t ff_a1, ff_t1, ff_b1;

  prec_t p_s1, p_s2, p_s3;

  ff_init (&ff_a1, 3.0, p_s1);
  ff_init (&ff_t1, 2.0, p_s2);
  b1 = a1 + 2.0;
  c = b * a;

  return c1;
}
```

```
ff_b1
③ b1 = a1 + t1;
```

```
IF (Sj contains REAL-TYPE operands)
        create FF alias for LHS (LHS-FF-alias)
        create precision variable for statment Sj (Psj)

FOR EACH USE of LHS
        set-FF-alias (USEk)  // mark USE k with
                             // defining FF alias
```

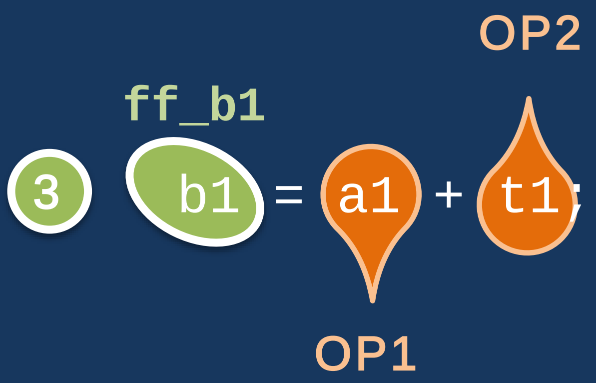
```
int main ()
{
  double a1, b1, c1;
  double t1;

BB 1:
① a1 = 3.0;
② t1 = 2.0;
③ b1 = a1 + t1;
④ c1 = b1 * a1;          ff_b1

  return c1;
}
```

How does *FlexFloat* instrumentation work?

A SIMPLE EXAMPLE…

```
int main ()
{
  double b1, c1;
  flexfloat_t ff_a1, ff_t1, ff_b1;
  flexfloat_t ff_a1_1, ff_t1_1;
  prec_t p_s1, p_s2, p_s3;

  ff_init (&ff_a1, 3.0, p_s1);
  ff_init (&ff_t1, 2.0, p_s2);
  b1 = a1 + 2.0;
  c = b * a;

  return c1;
}
```

handle BINOP expression



switch (EXPR (RHS))
case BINARY-EXPR:
handle-binary-expr

create FF alias for OP1 (OP1-FF-alias)
create FF alias for OP2 (OP2-FF-alias)

```
int main ()
{
  double a1, b1, c1;
  double t1;

BB 1:
①  a1 = 3.0;
②  t1 = 2.0;
③  b1 = a1 + t1;
④  c1 = b1 * a1;

  return c1;
}
```

ff_b1

How does *FlexFloat* instrumentation work?

```
int main ()
{
  double b1, c1;
  flexfloat_t ff_a1, ff_t1, ff_b1;
  flexfloat_t ff_a1_1, ff_t1_1;
  prec_t p_s1, p_s2, p_s3;

  ff_init (&ff_a1, 3.0, p_s1);
  ff_init (&ff_t1, 2.0, p_s2);
  ff_cast (&ff_a1_1, &ff_a1, p_s3);
  ff_cast (&ff_t1_1, &ff_t1, p_s3);
  b1 = a1 + 2.0;
  c = b * a;

  return c1;
}
```

handle BINOP expression

`ff_b1` `ff_a1` `ff_t1`

③ `b1 = a1 + t1;`

switch (EXPR (RHS))
case BINARY-EXPR:
handle-binary-expr

emit FF-CAST (&OP1-FF-alias, get-FF-alias (OP1), Psj)
emit FF-CAST (&OP2-FF-alias, get-FF-alias (OP2), Psj)

A SIMPLE EXAMPLE...

```
int main ()
{
  double a1, b1, c1;
  double t1;

BB 1:
① a1 = 3.0;
② t1 = 2.0;
③ b1 = a1 + t1;
④ c1 = b1 * a1;

  return c1;
}
```

`ff_b1`

How does *FlexFloat* instrumentation work?

```
int main ()
{
  double ~~b1~~, c1;
  flexfloat_t ff_a1, ff_t1, ff_b1;
  flexfloat_t ff_a1_1, ff_t1_1;
  prec_t p_s1, p_s2, p_s3;

  ff_init (&ff_a1, 3.0, p_s1);
  ff_init (&ff_t1, 2.0, p_s2);
  ff_cast (&ff_a1_1, &ff_a1, p_s3);
  ff_cast (&ff_t1_1, &ff_t1, p_s3);
  ff_add (&ff_b1, &ff_a1_1, &ff_t1_1);
  ~~b1 = a1 + 2.0;~~
  c = b * a;

  return c1;
}
```

handle BINOP expression

ff_b1 ff_a1 ff_t1

③ b1 = a1 + t1;

switch (EXPR (RHS))
case BINARY-EXPR:
handle-binary-expr

remove Sj

switch (EXPR (RHS))
case PLUS-EXPR:
case MULT-EXPR:
emit FF-<EXPR> (&LHS-FF-alias, &OP1-FF-alias, &OP2-FF-alias)

A SIMPLE EXAMPLE...

```
int main ()
{
  double a1, b1, c1;
  double t1;

BB 1:
① a1 = 3.0;
② t1 = 2.0;
③ b1 = a1 + t1;
④ c1 = b1 * a1;

  return c1;
}
```

ff_b1

FlexFloat instrumentation

A SIMPLE EXAMPLE…

```
int main ()
{
  double c1;
  flexfloat_t ff_a1, ff_t1, ff_b1, ff_c1;
  flexfloat_t ff_a1_1, ff_t1_1, ff_b1_1, ff_a1_2;
  prec_t p_s1, p_s2, p_s3, p_s4;

  ff_init (&ff_a1, 3.0, p_s1);
  ff_init (&ff_t1, 2.0, p_s2);
  ff_cast (&ff_a1_1, &ff_a1, p_s3);
  ff_cast (&ff_t1_1, &ff_t1, p_s3);
  ff_add (&ff_b1, &ff_a1_1, &ff_t1_1);
  ff_cast (&ff_b1_1, &ff_b1, p_s4);
  ff_cast (&ff_a1_2, &ff_a1, p_s4);
  ff_mul (&ff_c1, &ff_b1_1, &ff_a1_2);
```

~~c = b * a;~~

```
  return c1;
}
```

similar

remove Sj

ff_c1 ff_b1 ff_a1

③ `c1 = b1 * a1;`

```
switch (EXPR (RHS))
    case BINARY-EXPR:
        handle-binary-expr
```

```
switch (EXPR (RHS))
    case PLUS-EXPR:
    case MULT-EXPR:
emit FF-<EXPR> (&LHS-FF-alias, &OP1-FF-alias, &OP2-FF-alias)
```

```
int main ()
{
  double a1, b1, c1;
  double t1;

BB 1:
  a1 = 3.0;
  t1 = 2.0;
  b1 = a1 + t1;
  c1 = b1 * a1;

  return c1;
}
```

① ② ③ ④

ff_c1

FlexFloat instrumentation

A SIMPLE EXAMPLE…

```
int main ()
{
  double a1, b1, c1;
  double t1;

BB 1:
① a1 = 3.0;
② t1 = 2.0;
③ b1 = a1 + t1;
④ c1 = b1 * a1;

  return c1;
}
```

```
int main ()
{
  double c1;
  flexfloat_t ff_a1, ff_t1, ff_b1, ff_c1;
  flexfloat_t ff_a1_1, ff_t1_1, ff_b1_1, ff_a1_2;
  prec_t p_s1, p_s2, p_s3, p_s4;

  ff_init (&ff_a1, 3.0, p_s1);
  ff_init (&ff_t1, 2.0, p_s2);
  ff_cast (&ff_a1_1, &ff_a1, p_s3);
  ff_cast (&ff_t1_1, &ff_t1, p_s3);
  ff_add (&ff_b1, &ff_a1_1, &ff_t1_1);
  ff_cast (&ff_b1_1, &ff_b1, p_s4);
  ff_cast (&ff_a1_2, &ff_a1, p_s4);
  ff_add (&ff_c1, &ff_b1_1, &ff_a1_2);

  ff_get_double (&c1, &ff_c1);
  return c1;
}
```

similar

`ff_c1`

`return c1;`

FlexFloat instrumentation

A SIMPLE EXAMPLE…

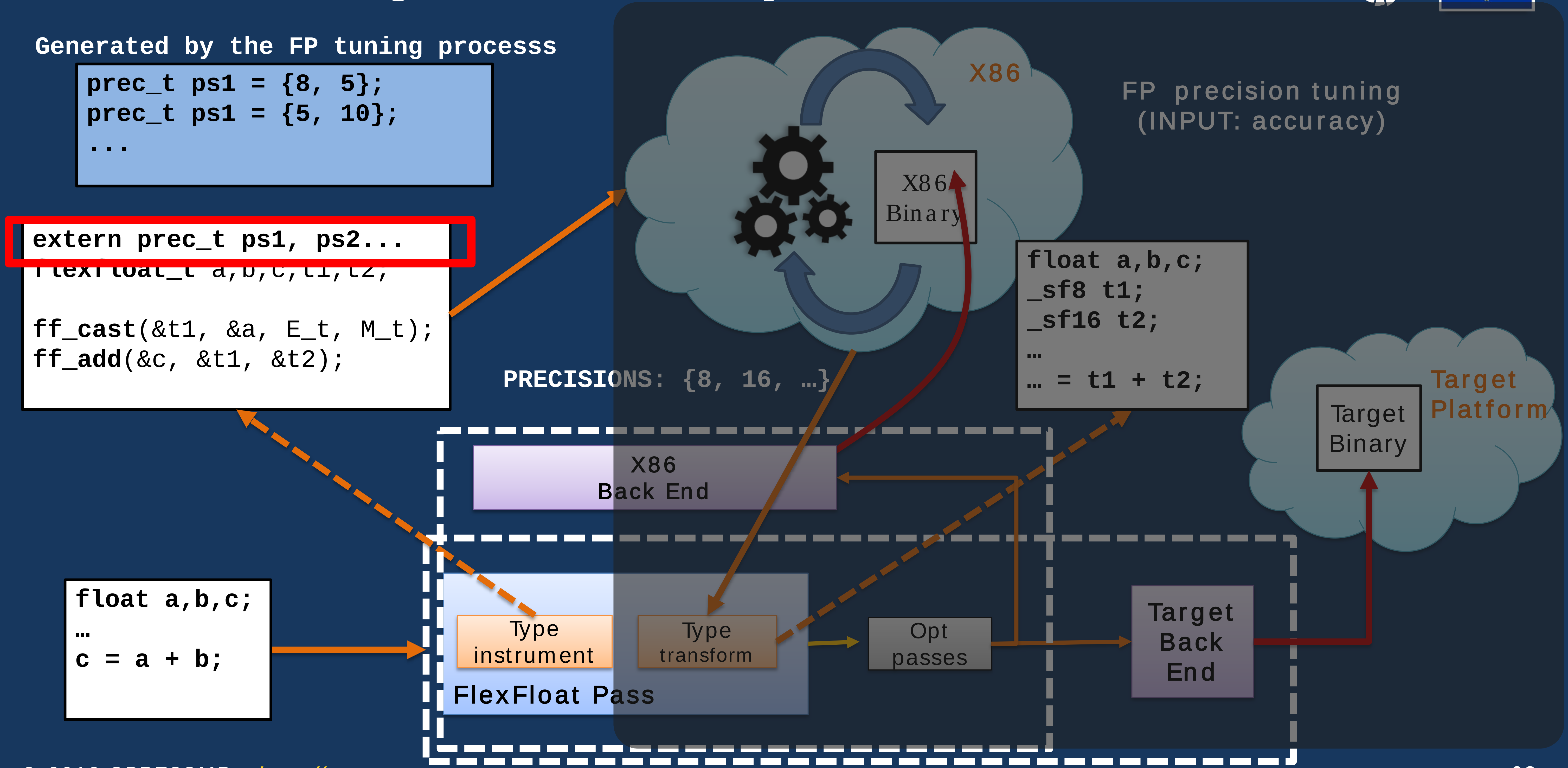
```
int main ()
{
  double c1;
  flexfloat_t ff_a1, ff_t1, ff_b1, ff_c1;
  flexfloat_t ff_a1_1, ff_t1_1, ff_b1_1, ff_a1_2;
  prec_t p_s1, p_s2, p_s3, p_s4;

  ff_init (&ff_a1, 3.0, p_s1);
  ff_init (&ff_t1, 2.0, p_s2);
  ff_cast (&ff_a1_1, &ff_a1, p_s3);
  ff_cast (&ff_t1_1, &ff_t1, p_s3);
  ff_add (&ff_b1, &ff_a1_1, &ff_t1_1);
  ff_cast (&ff_b1_1, &ff_b1, p_s4);
  ff_cast (&ff_a1_2, &ff_a1, p_s4);
  ff_add (&ff_c1, &ff_b1_1, &ff_a1_2);

  ff_cast_to double (&c1, &ff_c1);
  return c1;
}
```

Precision variables are actually declared as globally visible, extern objects

```
extern prec_t p_s1, p_s2, p_s3, p_s4;
```

…as this is an input from the precision tuning flow

Automation: integration with compilation toolchain



Generated by the FP tuning processs

```
prec_t ps1 = {8, 5};
prec_t ps1 = {5, 10};
...
```

```
extern prec_t ps1, ps2...
flexfloat_t a,b,c,t1,t2;

ff_cast(&t1, &a, E_t, M_t);
ff_add(&c, &t1, &t2);
```

```
float a,b,c;
…
c = a + b;
```

```
float a,b,c;
_sf8 t1;
_sf16 t2;
…
… = t1 + t2;
```

Agenda

- ❑ Introduction – Transprecision Computing
- ❑ *Smaller-than-32-bit* floating point types
- ❑ Implementing the *smallFloat* extension
  - HW support
  - Compiler support
- ❑ Simplifying the deployment of *SmallFloat-based* applications
- ❑ **Conclusion**

Conclusion

- ☐ *Less-than-32-bit* floating point types are beneficial to reduce execution time/energy consumption
- ☐ Support is required at HW level and compiler level to implement SmallFloat types
- ☐ A compilation toolchain can provide automatic tuning
    - In the best case, programmers use float/double variables as usual and do not care about auxiliary FP types

July 18th 2018, NIPS Lab (Perugia, IT)

International Summer School on Energy Aware Transprecision Computing

Funded by the H2020 Framework
Programme of the European Union

SW and TOOLS

Overview of integrated support for Transprecision Computing

Andrea Marongiu (*a.marongiu@unibo.it*)
Giuseppe Tagliavini (*giuseppe.tagliavini@unibo.it*)

DISI - Department of Computer Science and Engineering
DEI – Department of Electronic Engineering
University of Bologna
Bologna, Italy

OPRECOMP
Open Transprecision Computing

© 2017 OPRECOMP – *http://oprecomp.eu*