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

International Summer School on Energy Aware Transprecision Computing

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
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 through 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.

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 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
- [Image]

**Analyze and Classify**
- μController
- L2 Memory
- IOs

**Transmit**
- Short range, medium BW
- Long range, low BW

100 µW ÷ 2 mW

1 ÷ 2000 MOPS

1 ÷ 10 mW

Low rate (periodic) data

Idle: ~1µW
Active: ~ 50mW

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

Context: Distributed Embedded Computing

---

**Sense**

- MEMS IMU
- MEMS Microphone
- ULP Imager
- EMG/ECG/EIT

**Analyze and Classify**

- ULP Imager
- MEMS IMU
- MEMS Microphone

**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
- Long range, low BW

---

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]

---


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** $\leftrightarrow$ **dynamic range**
  - **mantissa** $\leftrightarrow$ **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** $\rightarrow$ 1 bit sign, $e$ bits exponent, $m$ bits mantissa

![FP16 format diagram](image)

- FP16 can represent 30,720 values $\rightarrow$ 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

```c
__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
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?

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 execution time and energy consumption
  - 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.

![Diagram of floating-point types and their characteristics]

---

[1] Giuseppe Tagliavini, Stefan Mach, Andrea Marongiu, Davide Rossi, Luca Benini
A Transprecision Floating-Point Platform for Ultra-Low Power Computing
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

- **Slice32**
  - FP32 ADD/SUB
  - FP32 MUL
  - FP16 ACC
  - FP16 -> int32

- **Slice16**
  - FP16 ADD/SUB
  - FP16 MUL
  - FP16alt MUL
  - FP16 -> int16

- **Slice8**
  - FP8 ADD/SUB
  - FP8 MUL
  - FP8 MUL

**Data Distribution and Operand Isolation**

**Operand Inputs**

**Output Data Selection**

**Result Output**

1) Supporting the SmallFloat data type extension
Energy consumption of SmallFloat operations

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

Average energy per operation (from post-layout simulations)

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

Idle System Energy per Cycle
Almost Identical
Energy decreases with fewer mantissa bits

95.2 pJ / 4 = 23.8 pJ
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

- C/C++ plus SmallFloat type system
- JAVA front-end
  - C++ front-end
  - C front-end
  - parse trees
- middle-end
  - generic trees
- back-end
  - RTL
- GCC Passes
  - generic trees
  - gimple trees
  - into SSA
  - SSA optimizations
  - out of SSA
  - gimple trees
  - generic trees

Generate code to use RISCV SmallFloat extensions
Compiler support to the SmallFloat data types

Front End
- SmallFloat types

Middle End
- Opt pass 1
- Opt pass 2
- Vectorizer

Back End
- RISCV specific
  - SmallFloat scalar types
  - SmallFloat vector types
  - SmallFloat casts
  - SmallFloat scalar ops
  - SmallFloat vector ops

FlexFloat / TransFloat support

RISCV Extension

ASM

BINUTILS
- RISCV ISA
- SmallFloat ISA

Assembler

Binary Tools

Binary

Execution on Final target
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...**

```c
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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Energy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flw fa5,0(s0)</td>
<td>472.0 pJ</td>
<td>LOAD/STORE half word operands does not reduce the energy consumption</td>
</tr>
<tr>
<td>flw fa3,0(s2)</td>
<td>425.6 pJ</td>
<td>ADD (integer)</td>
</tr>
<tr>
<td>flw fa4,0(s3)</td>
<td>213.6 pJ</td>
<td>ADD (float)</td>
</tr>
<tr>
<td>add s0,s0,4</td>
<td>472.0 pJ</td>
<td>LOAD/STORE</td>
</tr>
<tr>
<td>add s4,s4,4</td>
<td>425.6 pJ</td>
<td>ADD (integer)</td>
</tr>
<tr>
<td>add s2,s2,4</td>
<td>197.6 pJ</td>
<td>ADD (float16)</td>
</tr>
<tr>
<td>add s3,s3,4</td>
<td>74.7 pJ</td>
<td>CONV</td>
</tr>
<tr>
<td>fadd.s fa5,fa5,fa3</td>
<td>472.0 pJ</td>
<td>LOAD/STORE</td>
</tr>
<tr>
<td>fadd.s fa4,fa4,fa5</td>
<td>425.6 pJ</td>
<td>ADD (integer)</td>
</tr>
<tr>
<td>fsw fa5,-4(s0)</td>
<td>213.6 pJ</td>
<td>ADD (float)</td>
</tr>
<tr>
<td>flh a3,0(s1)</td>
<td>472.0 pJ</td>
<td>LOAD/STORE</td>
</tr>
<tr>
<td>flh a2,0(s3)</td>
<td>425.6 pJ</td>
<td>ADD (integer)</td>
</tr>
<tr>
<td>sh a3,-2(s1)</td>
<td>213.6 pJ</td>
<td>ADD (float)</td>
</tr>
<tr>
<td>sh a4,0(s4)</td>
<td>74.7 pJ</td>
<td>CONV</td>
</tr>
</tbody>
</table>

1169.9 pJ TOT

Additional cast operations are required.
The role of vectorization

How does automatic vectorization work?

Data in Memory:

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

Vector Registers

VF = 4

OP(a)

OP(b)

OP(c)

OP(d)

Vector operation

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?

- Loop based vectorization
  - No dependences between iterations

Automatic vectorization is the key compiler optimization to enable energy savings

- original serial loop:
  ```c
  for(i=0; i<N; i++){
    a[i] = a[i] + b[i];
  }
  ```

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

- loop in vector notation:
  ```c
  for (i=0; i<N; i+=VF){
    a[i:i+VF] = a[i:i+VF] + b[i:i+VF];
  }
  ```

- epilog loop:
  ```c
  for (; i < N; i++) {
    a[i] = a[i] + b[i];
  }
  ```
The role of vectorization

GCC Passes
- 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

parse trees
- middle-end
  - generic trees
- back-end
  - RTL
  - machine description

SmallFloat extensions

C front-end
- C++ front-end
  - Java front-end

misc opts

dependence analysis
The role of vectorization

<table>
<thead>
<tr>
<th>.L3:</th>
<th>.L3:</th>
<th>.L3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>flw  fa5,0(s0)</td>
<td>flw  fa5,0(s2)</td>
<td>lw    a0,0(s4)</td>
</tr>
<tr>
<td>flw  fa3,0(s2)</td>
<td>flh  a3,0(s1)</td>
<td>lw    a4,0(s6)</td>
</tr>
<tr>
<td>flw  fa4,0(s3)</td>
<td>flh  a2,0(s3)</td>
<td>flw   fa4,8(s5!)</td>
</tr>
<tr>
<td>add  s0,s0,4</td>
<td>add  s1,s1,2</td>
<td>flw   fa5,8(a1!)</td>
</tr>
<tr>
<td>add  s4,s4,4</td>
<td>add  s3,s3,2</td>
<td>add   s6,s6,4</td>
</tr>
<tr>
<td>add  s2,s2,4</td>
<td>add  s2,s2,4</td>
<td>add   a3,a3,4</td>
</tr>
<tr>
<td>add  s3,s3,4</td>
<td>add  s4,s4,2</td>
<td>add   s4,s4,4</td>
</tr>
<tr>
<td>fadd.s fa5,fa5,fa3</td>
<td>fcvt.h.s a4,fa5</td>
<td>vfcpka.h.s a5,fa4,fa5</td>
</tr>
<tr>
<td>fadd.s fa4,fa4,fa5</td>
<td>fadd.h a3,a3,a2</td>
<td>vfadd.h a4,a4,a0</td>
</tr>
<tr>
<td>fsw  fa5,-4(s0)</td>
<td>fadd.h a4,a4,a3</td>
<td>vfadd.h a5,a5,a4</td>
</tr>
<tr>
<td>fsw  fa5,-4(s4)</td>
<td>sh    a3,-2(s1)</td>
<td>sw    a4,-4(s4)</td>
</tr>
<tr>
<td>sh    a4,0(s4)</td>
<td>sh    a4,0(s4)</td>
<td>sw    a5,0(a3)</td>
</tr>
</tbody>
</table>

1169.9 pJ (iter) * 1024 iters = 1198 nJ

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)
Simplifying the deployment of *SmallFloat*-based applications

**Precision Tuning** of FP variables

- Programs are written using **standard FP formats**
  - C/C++ programs \( \rightarrow \) float and double variables

- **Precision tuning** \( \rightarrow \) transforming programs by changing default FP types to introduce smaller ones
  - Manually
  - Semi-automatically
  - Automatically

- Research papers and open source tools are available...
SOA of precision tuning

- Statistical methods
  - Source-to-source transform
  - Compiler IR language
  - Binary instrumentation

- Exact methods
  - Formal theorem proof
  - Branch and bound methods

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


© 2018 OPRECOMP – http://oprecomp.eu
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

- Upper Bound
- Solution
- Step1
- Lower Bound
### Experiments: Single-precision and Half-precision

#### Relative error on program results

<table>
<thead>
<tr>
<th>$\varepsilon$</th>
<th>Application</th>
<th>Precision (mantissa bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3-11</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>HOG</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>KNN</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>PCA</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>DWT</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>SVM</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>CONV</td>
<td>0%</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>HOG</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>KNN</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>PCA</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>DWT</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>SVM</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>CONV</td>
<td>0%</td>
</tr>
<tr>
<td>$10^{-1}$</td>
<td>HOG</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>KNN</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>PCA</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>DWT</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>SVM</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>CONV</td>
<td>0%</td>
</tr>
</tbody>
</table>

#### Percentage of variables after tuning

<table>
<thead>
<tr>
<th>$\varepsilon$</th>
<th>Application</th>
<th>Precision (mantissa bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3-11</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>HOG</td>
<td>50%</td>
</tr>
<tr>
<td></td>
<td>KNN</td>
<td>50%</td>
</tr>
<tr>
<td></td>
<td>PCA</td>
<td>91%</td>
</tr>
<tr>
<td></td>
<td>DWT</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>SVM</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>CONV</td>
<td>50%</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>HOG</td>
<td>50%</td>
</tr>
<tr>
<td></td>
<td>KNN</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>PCA</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>DWT</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>SVM</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>CONV</td>
<td>50%</td>
</tr>
<tr>
<td>$10^{-1}$</td>
<td>HOG</td>
<td>50%</td>
</tr>
<tr>
<td></td>
<td>KNN</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>PCA</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>DWT</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>SVM</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>CONV</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Single 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)
### Experiments: Single-precision, half-precision and quarter-precision

<table>
<thead>
<tr>
<th>$\varepsilon$</th>
<th>Application</th>
<th>Precision (mantissa bits)</th>
<th>3-11</th>
<th>12-23</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-6}$</td>
<td>HOG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>KNN</td>
<td>50%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCA</td>
<td>91%</td>
<td>9%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DWT</td>
<td>100%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SVM</td>
<td>100%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONV</td>
<td>50%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>HOG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>KNN</td>
<td>50%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCA</td>
<td>100%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DWT</td>
<td>100%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SVM</td>
<td>100%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONV</td>
<td>50%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td>$10^{-1}$</td>
<td>HOG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>KNN</td>
<td>50%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCA</td>
<td>100%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DWT</td>
<td>100%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SVM</td>
<td>100%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONV</td>
<td>100%</td>
<td>0%</td>
<td></td>
</tr>
</tbody>
</table>

#### Single precision + half precision

#### 100% half to quarter

#### 60% half to quarter (on avg)

### Single precision + half precision + quarter precision

<table>
<thead>
<tr>
<th>$\varepsilon$</th>
<th>Application</th>
<th>Precision (mantissa bits)</th>
<th>3</th>
<th>4-11</th>
<th>12-23</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-6}$</td>
<td>HOG</td>
<td></td>
<td>50%</td>
<td>0%</td>
<td>50%</td>
</tr>
<tr>
<td></td>
<td>KNN</td>
<td>0%</td>
<td>50%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCA</td>
<td>0%</td>
<td>91%</td>
<td>9%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DWT</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SVM</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONV</td>
<td>0%</td>
<td>50%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>HOG</td>
<td></td>
<td>0%</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td></td>
<td>KNN</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCA</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DWT</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SVM</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONV</td>
<td>0%</td>
<td>50%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td>$10^{-1}$</td>
<td>HOG</td>
<td></td>
<td>50%</td>
<td>0%</td>
<td>50%</td>
</tr>
<tr>
<td></td>
<td>KNN</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCA</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DWT</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SVM</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONV</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
</tbody>
</table>

#### QUARTER PRECISION

1 bit sign
5 bits exponent
2(+1) bits mantissa
Experiments: Single-precision, 2x half-precision and quarter-precision

<table>
<thead>
<tr>
<th>$\varepsilon$</th>
<th>Application</th>
<th>Precision (mantissa bits)</th>
<th>$10^{-6}$</th>
<th>$10^{-4}$</th>
<th>$10^{-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>4-11</td>
<td>12-23</td>
<td>3</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>HOG</td>
<td>50%</td>
<td>0%</td>
<td>50%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>KNN</td>
<td>0%</td>
<td>50%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>PCA</td>
<td>0%</td>
<td>91%</td>
<td>9%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>DWT</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>SVM</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>CONV</td>
<td>0%</td>
<td>50%</td>
<td>50%</td>
<td>0%</td>
</tr>
</tbody>
</table>

$10^{-4}$

| $10^{-4}$     | HOG         | 0%        | 50%      | 50%       | 0%        | 50%      | 0%       | 50%       |
|               | KNN         | 100%      | 0%       | 0%        | 100%      | 0%       | 0%       | 0%        |
|               | PCA         | 0%        | 100%     | 0%        | 0%        | 1%       | 91%      | 9%        |
|               | DWT         | 0%        | 100%     | 0%        | 0%        | 0%       | 100%     | 0%        |
|               | SVM         | 100%      | 0%       | 0%        | 0%        | 0%       | 0%       | 0%        |
|               | CONV        | 0%        | 50%      | 50%       | 0%        | 100%     | 0%       | 0%        |

$10^{-1}$

| $10^{-1}$     | HOG         | 50%       | 0%       | 50%       | 100%      | 0%       | 0%       | 50%       |
|               | KNN         | 100%      | 0%       | 0%        | 0%        | 0%       | 0%       | 0%        |
|               | PCA         | 0%        | 100%     | 0%        | 0%        | 0%       | 0%       | 0%        |
|               | DWT         | 0%        | 100%     | 0%        | 0%        | 0%       | 0%       | 0%        |
|               | SVM         | 100%      | 0%       | 0%        | 0%        | 0%       | 0%       | 0%        |
|               | CONV        | 100%      | 0%       | 0%        | 0%        | 0%       | 0%       | 0%        |

**ALTERNATIVE HALF PRECISION**

1 bit sign
8 bits exponent
7(+1) bits mantissa

Almost 100% single & half to alt half

100% single to alt half
**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));
```

**typedef struct**

```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
float a, b, c;
... = t1 + t2;

float a, b, c;
... = t1 + t2;

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

FlexFloat Pass
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

Diagram:
- GCC Passes
  - C front-end
  - C++ front-end
  - Java front-end
  - Parse trees
- Middle-end
  - Generic trees
- Back-end
  - RTL
- Machine description
- GCC passes:
  - Generic trees
  - Gimple trees
  - Into SSA
  - SSA optimizations
  - Out of SSA
  - Gimple trees
  - Generic trees
  - Misc opts
  - Flexfloat
  - Loop optimizations
    - Loop opts
    - Vectorization
    - Loop opts
    - Misc opts
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)

```plaintext
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)**
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;
```

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*

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

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 statement 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?

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

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

    return c;
}
```

...and its (GIMPLE) SSA representation

```c
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...

```c
int main ()
{
    double a1, b1, c1;
    double t1;
    flexfloat_t ff_a1;
    prec_t p_s1;
    a1 = 3.0;
    t1 = 2.0;
    b1 = a1 + t1;
    c1 = b1 * a1;
    return c1;
}
```

We start by walking BBs and statements therein

- **ff_a1**
- **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

```c
int main ()
{
    double a1, b1, c1;
    double t1;
    BB 1:
    
    al = 3.0;  // ff_a1
    t1 = 2.0;
    b1 = al + t1;
    c1 = b1 * al;
    return c1;
}
```
How does *FlexFloat* instrumentation work?

```c
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 = b1 * a;

    return c1;
}
```

We start by walking BBs and statements therein.

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

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

    return c1;
}
```
How does **FlexFloat** instrumentation work?

### A SIMPLE EXAMPLE...

```c
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;
}
```

**similar process**

```c
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. create precision variable for statement $S_j$ ($P_s$)
2. `ff_init` for `LHS-FF-alias`
3. `set-FF-alias` for `USEk`
4. `If (is-real-const (RHS))`
   - emit `FF-INIT<REAL-TYPE>` for `LHS-FF-alias`, `RHS`, $P_s$
How does **FlexFloat** instrumentation work?

```c
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;
}
```

**A SIMPLE EXAMPLE...**

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

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

    return c1;
}
```

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

FOR EACH USE of LHS
- set-FF-alias (USEk) // mark USE k with
  // defining FF alias
How does **FlexFloat** instrumentation work?

```c
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**

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

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

    return c1;
}
```

- **create FF alias for OP1 (OP1-FF-alias)**
- **create FF alias for OP2 (OP2-FF-alias)**
- **switch (EXPR (RHS))**
  - case BINARY-EXPR: **handle-binary-expr**
How does FlexFloat instrumentation work?

```c
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;
}
```

A SIMPLE EXAMPLE...

```c
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?

```c
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;
}
```

A SIMPLE EXAMPLE...

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

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

  return c1;
}
```

**handle BINOP expression**

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

```c
handle BINOP expression
ff_b1 = a1 + t1;
```

```c
switch (EXPR (RHS))
  case PLUS-EXPR:
  case MULT-EXPR:
```
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;
    similar

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;
}

/* A SIMPLE EXAMPLE... */

int main () {

double a1, b1, c1;
double t1;

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

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

        emit FF-<EXPR> (&LHS-FF-alias, &OP1-FF-alias, &OP2-FF-alias)

        return c1;
}

© 2018 OPRECOMP – http://oprecomp.eu
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;
}

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_cast_to_double (&c1, &ff_c1);
  return c1;
}

**FlexFloat instrumentation**

A SIMPLE EXAMPLE...

Precision variables are actually declared as globally visible, extern objects

```c
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 process:

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

```c
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);
```

```c
float a,b,c;
...
 c = a + b;
```

Precision tuning (INPUT: accuracy):

X86

```c
float a,b,c;
_sf8 t1;
_sf16 t2;
(...) = t1 + t2;
```

Target Platform

X86 Back End

Opt passes

Target Back End

Type instrument

Type transform

© 2018 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
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
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

Open Transprecision Computing
© 2017 OPRECOMP – http://oprecomp.eu