HLRS

High-Performance
Computing Center
Stuttgart

Evaluating a Real-Time Lossy Array Compression Algorithm for Computer Simulations

Darjan Krijan

Me

Darjan Krijan M.Sc.

darjan.krijan@hlrs.de

- Mechanical Engineering background, focused on
  - Thermal Turbomachinery @ ITSM/Uni Stuttgart
  - Sealing Technology @ IMA/Uni Stuttgart
- At HLRS since 10/2019
  - In project SiVeGCS [2017-2025] funded by
    - BMBF
    - MWK BW

Bundesministerium für Bildung und Forschung

Baden-Württemberg
MINISTRY OF SCIENCE, RESEARCH AND ARTS

- Optimizing node-level performance of HPC applications
  - NS3Dneo
  - m-AIA
- Started PhD 04/2024

Overview

- Introduction: Memory Bandwidth Problems
- State of Technology – Roofline Model
- State of Technology – Remedies
- New Approach – Proof of Concept
- Outlook

Overview

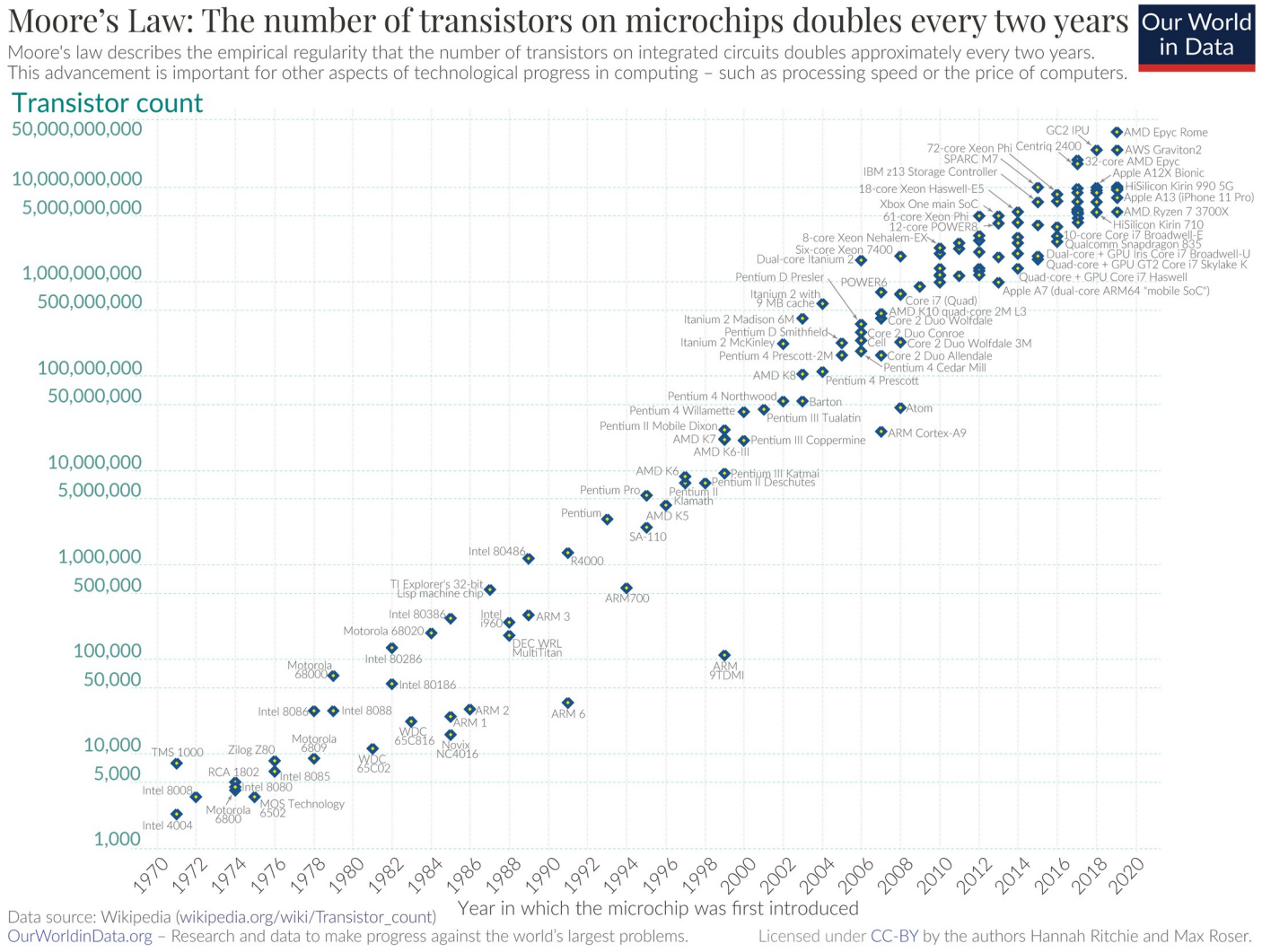
- Introduction: Memory Bandwidth Problems
- State of Technology – Roofline Model
- State of Technology – Remedies
- New Approach – Proof of Concept
- Outlook

Introduction: Memory Bandwidth Problems

Moore’s Law

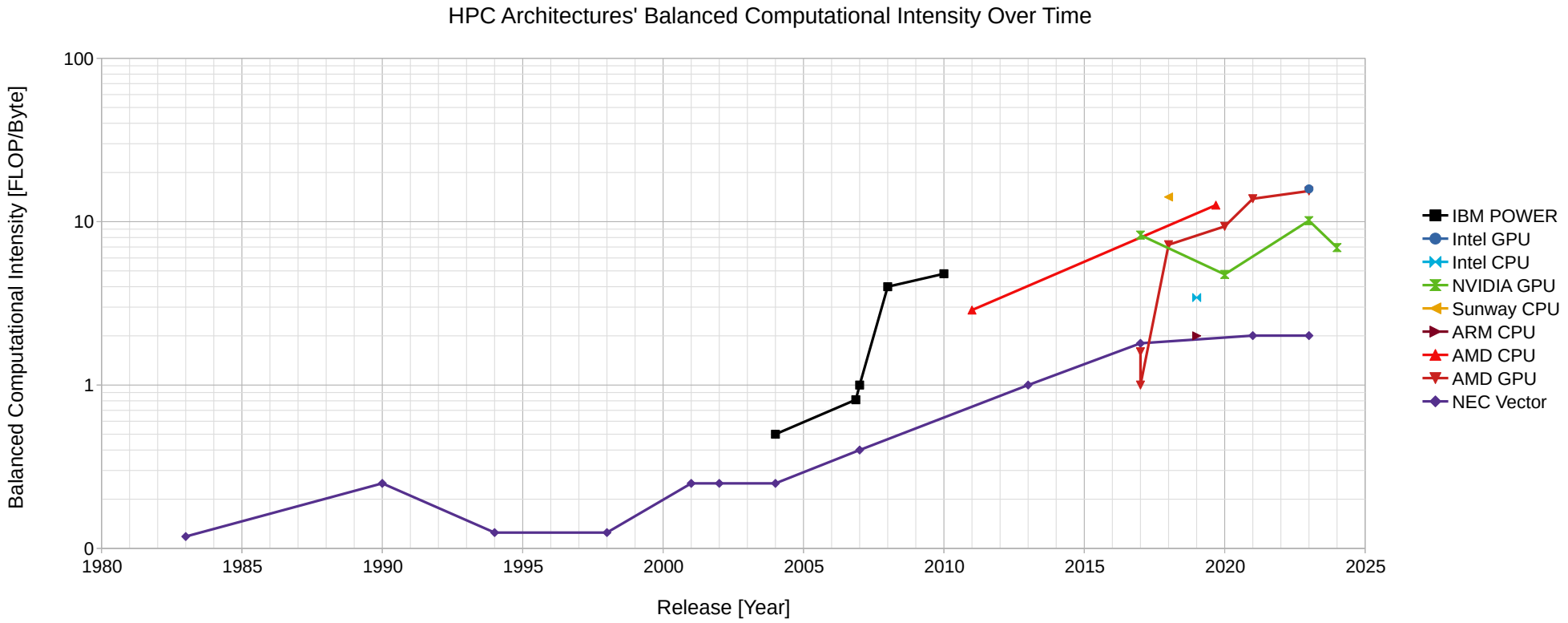
Moore’s Law

- Described by Gordon Moore, Intel founder (January 3, 1929 – March 24, 2023)
- Doubling complexity every ~2 years
  - ~ FLOP performance
- Does this apply to memory bandwidth as well?



FLOPS/Memory Bandwidth Progression

- Memory Bandwidth falls behind by a factor of ~5.1x/decade since the 1990s
- Processors being starved out of data, limiting performance
- Large fraction of HPC codes are memory bandwidth bound today due to this imbalance



Overview

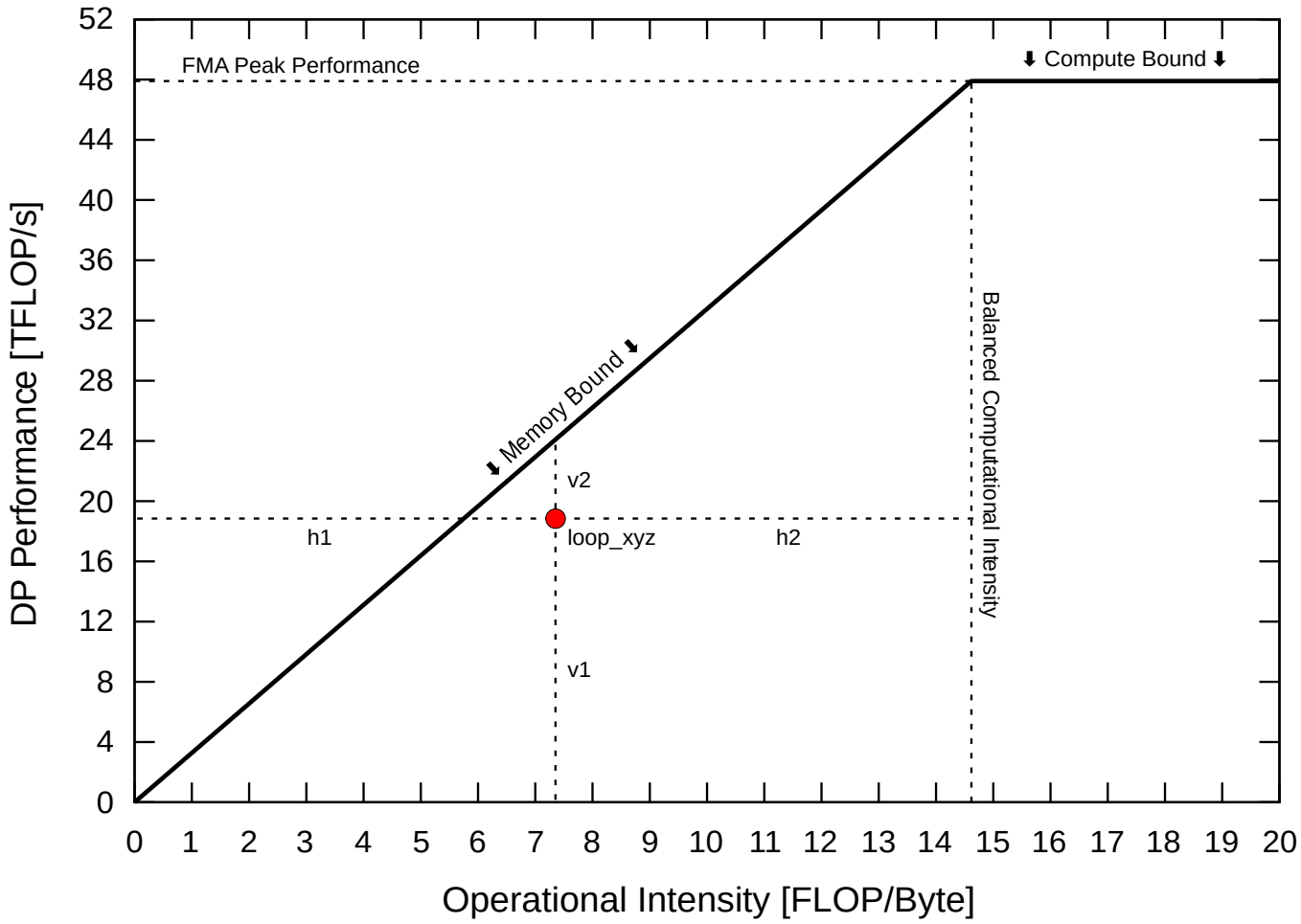
- Introduction: Memory Bandwidth Problems
- State of Technology – Roofline Model
- State of Technology – Remedies
- New Approach – Proof of Concept
- Outlook

State of Technology – Roofline Model

Memory vs. FLOPS

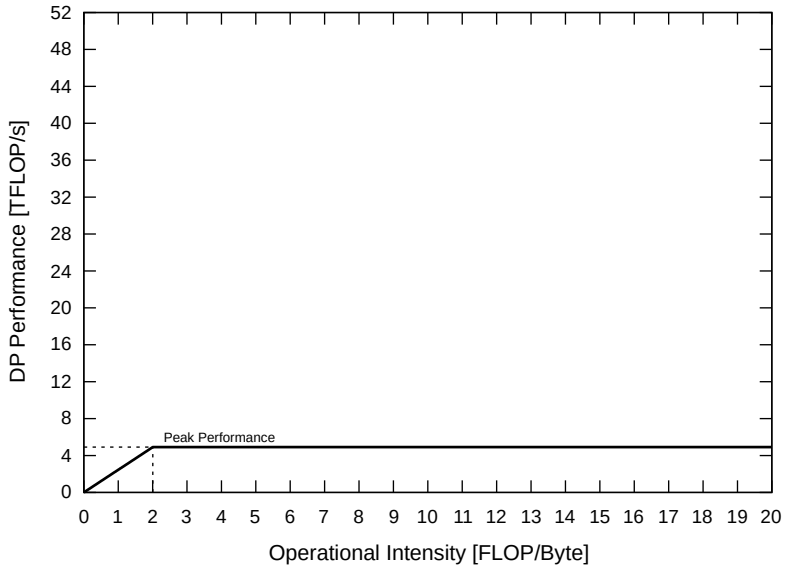
Roofline Model

- Maximum achievable performance as a function of Operational Intensity [FLOP/Byte]
- Operational Intensity for e.g. a loop: #FLOPs / data_moved_from_to_memory
- Idealized model with peak performance calculated from FMA units working at 100%
- h1/(h1+h2) = possible FPU usage [%]
- v1/(v1+v2) = efficiency of possible FPU usage [%]



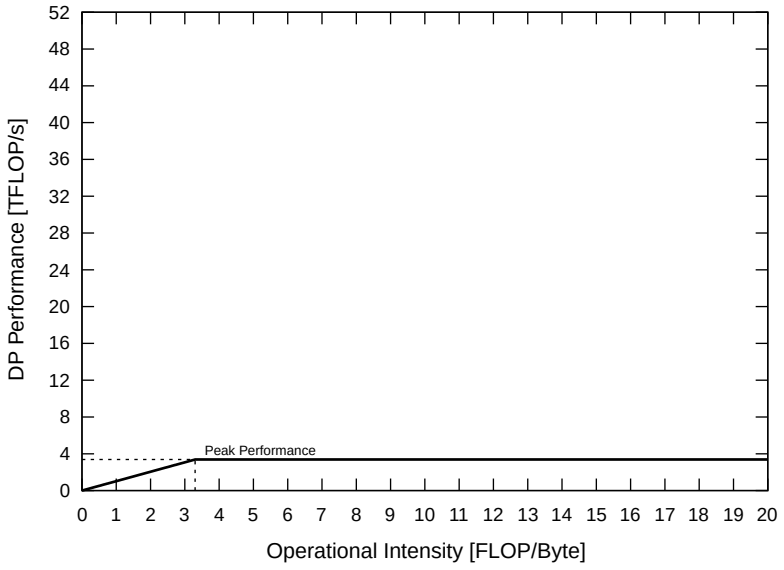
Roofline Model – Current Compute Architectures

Roofline model for NEC SX-Aurora TSUBASA 30A



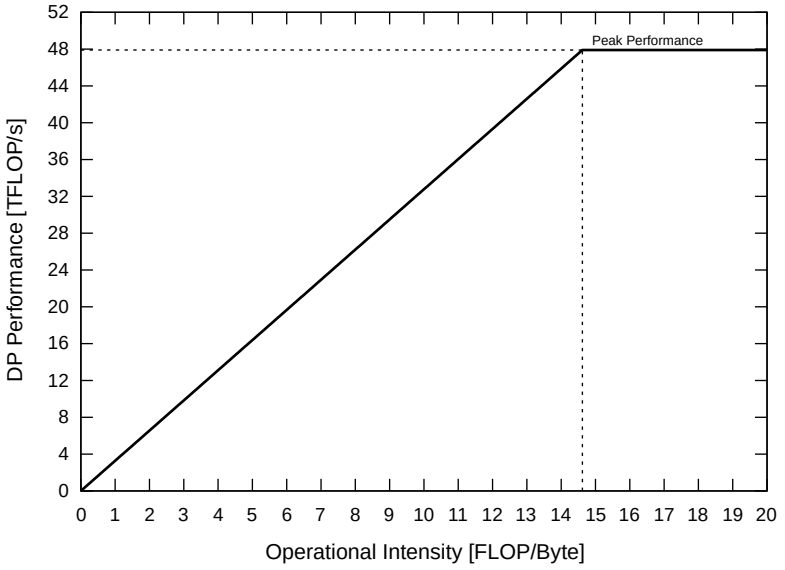
4.91 TFLOP/s
2.45 TB/s

Roofline model for Fujitsu A64FX



3.4 TFLOP/s
1.024 TB/s

Roofline model for AMD Instinct MI250X



47.9 TFLOP/s
3.2768 TB/s

Overview

- Introduction: Memory Bandwidth Problems
- State of Technology – Roofline Model
- State of Technology – Remedies
- New Approach – Proof of Concept
- Outlook

State of Technology – Remedies

Mixed Precision, Lossy Compression

State of Technology – Remedies

- Mixed-Precision approaches
  - Storing as single-precision → convert to double
    - Doubling throughput
    - Accuracy issues
  - Instructions taking single-precision as input, but output to double precision
  - Fixed-point
    - Expensive DIV when converting back to Floating-point
- (lossy) memory compression, ZFP from LLNL
  - Linkable library → function calls
  - Bandwidths of a couple ~100 MB/s, ~1 GB/s multi-threaded
    - New bottleneck…
  - Feasible for I/O
  - Basic concepts are used in new approach

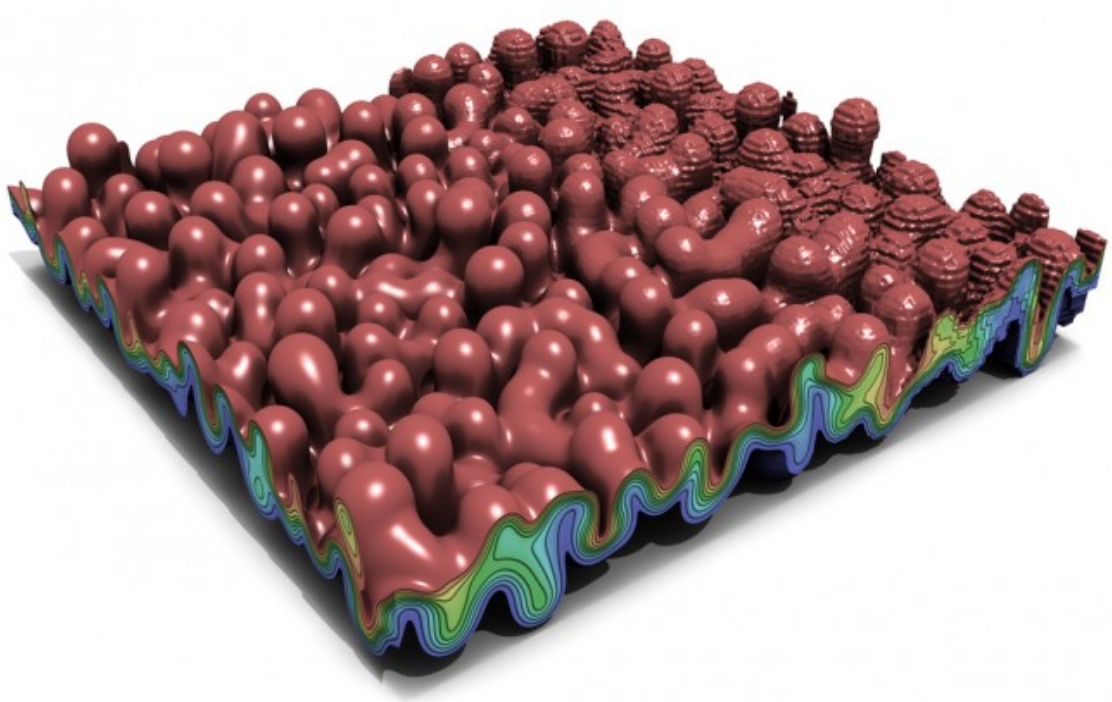


Illustration of zfp's ability to vary the compression ratio to any desired setting, from 10:1 on the left to 250:1 on the right, where the partitioning of the data set into small blocks is evident. The visualization shows the density field from a Rayleigh-Taylor instability simulation.

Overview

- Introduction: Memory Bandwidth Problems
- State of Technology – Roofline Model
- State of Technology – Remedies
- New Approach – Proof of Concept
- Outlook

New Approach – Proof of Concept

Real-Time Lossy Array Compression

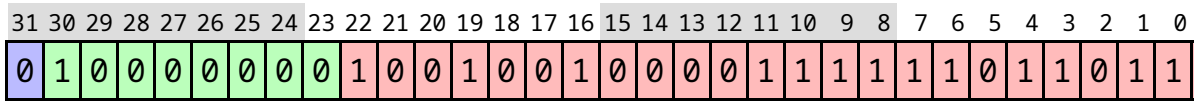
Real-Time Lossy Array Compression

- Computer simulations are simulating physical effects, thus have data arrays of physical properties
- Example physical properties for CFD simulatinos:
  - temperature, pressure, density, viscosity, enthalpy, entropy, etc.
- In a specific simulation, the upper and lower bounds of values of these arrays can be
  - determined by running a simulation
  - estimated from boundary conditions
- The **knowledge** of the array value bounds will be used at **compile-time** for a **real-time** compression algorithm
  - real-time: keep up with memory bandwidth and not take measurable time

Floating-Point Format – IEEE (32/64-bit)

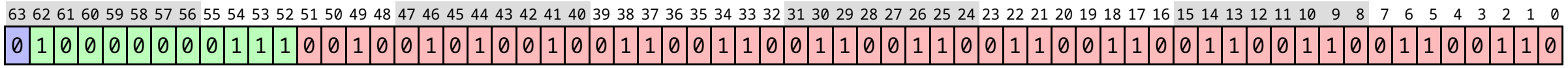
Components:

- Sign
- Exponent
- Significand



$1 \times 2^{1} \times 1.5707964 = 3.1415927$

Accuracy: ~7-8 digits

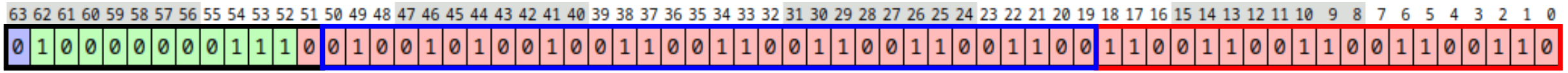


$1 \times 2^{8} \times 1.1451171875 = 293.15$

Accuracy: ~15-17 digits

Floating-Point Format – Example Array

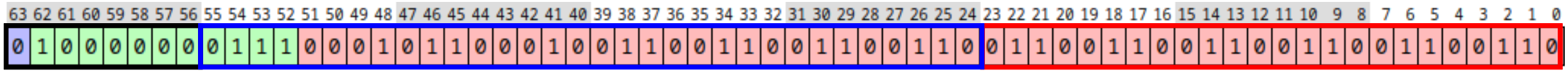
Temperature array: 293.15K...333.15K ≙20°C...60°C



$1 \times 2^{8} \times 1.145117187499999 = 293.15$

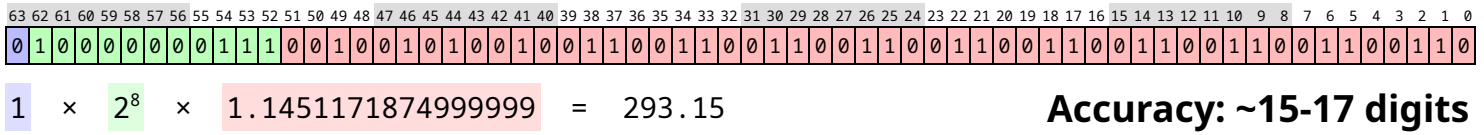
Floating-Point Format – Example Array 2

Temperature array: 278.15K...1823.15K ≙5°C...1550°C

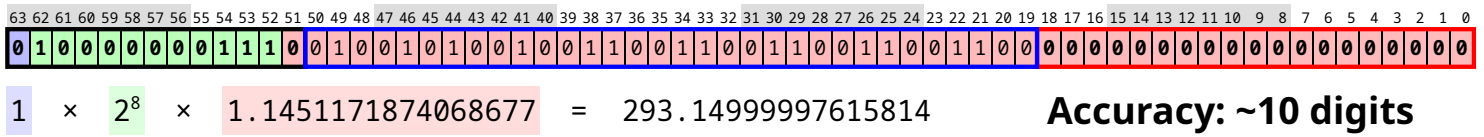


$1 \times 2^{8} \times 1.086523437499999 = 278.15$

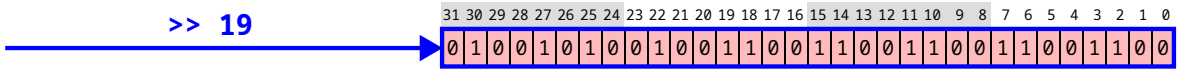
Lossy Array Compression Algorithm (32-bit)



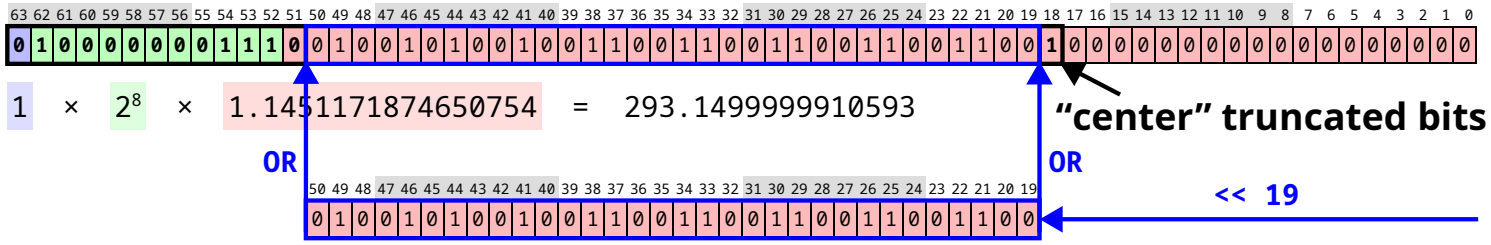
- e.g. a temperature value of 293.15K ≙ 20°C
- value range (+margin) for simulation: 293.15K...333.15K



Memorize:
- least significant 19 bits essentially truncated
- memorize sign and exponent (base value)



Store value:
- swap type to **`uint64_t`**
- shift 19 bits right
- cast to **`uint32_t`**
- store **`uint32_t`** value to memory



Reconstruct value:
- retrieve **`uint32_t`** value from memory
- cast to **`uint64_t`**
- shift 19 bits left
- bitwise **`OR`** with base value
- swap type to **`double`**
- number crunching in **`double`** precision

Proof of Concept – Equation of State Compute Kernel

$$p \cdot V = m \cdot R_s \cdot T$$

$$p = \rho \cdot R_s \cdot T$$

Is this bound by FLOP/s or memory bandwidth on recent hardware?

```
const uint64_t n = HUGE;
void calc_p(double *p, double *rho, double *Rs, double *T) {
  #pragma omp parallel for schedule(static)
  for (uint64_t i = 0; i < n; i++) {
    p[i] = rho[i]*Rs[i]*T[i];
  }
}
```

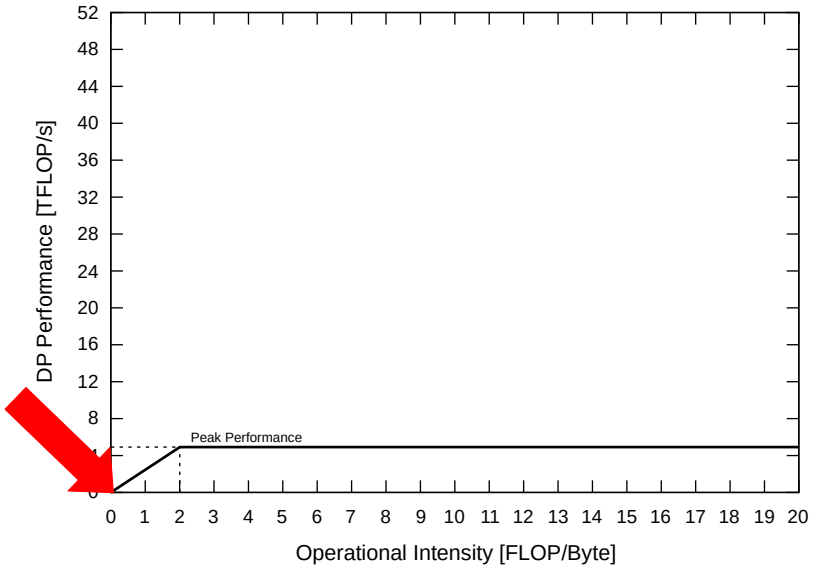
Proof of Concept – Equation of State Compute Kernel

```
const uint64_t n = HUGE;
void calc_p(double *p, double *rho, double *Rs, double *T) {
  #pragma omp parallel for schedule(static)
  for (uint64_t i = 0; i < n; i++) {
    p[i] = rho[i]*Rs[i]*T[i];
  }
}
```

- 2 multiplications → 2 FLOP
- Either:
  - 4 DP moves per iteration (3 loads, 1 store) → 4*sizeof(double) = 4*8 = 32 Byte
  - Operational Intensity:
    (2 FLOP)/(32 Byte) = 1/16 FLOP/Byte = **0.0625 FLOP/Byte**
- Or:
  - 5 DP moves per iteration (3 loads, 1 load+store) → 5*sizeof(double) = 5*8 = 40 Byte
  - Operational Intensity:
    (2 FLOP)/(40 Byte) = 1/20 FLOP/Byte = **0.05 FLOP/Byte**

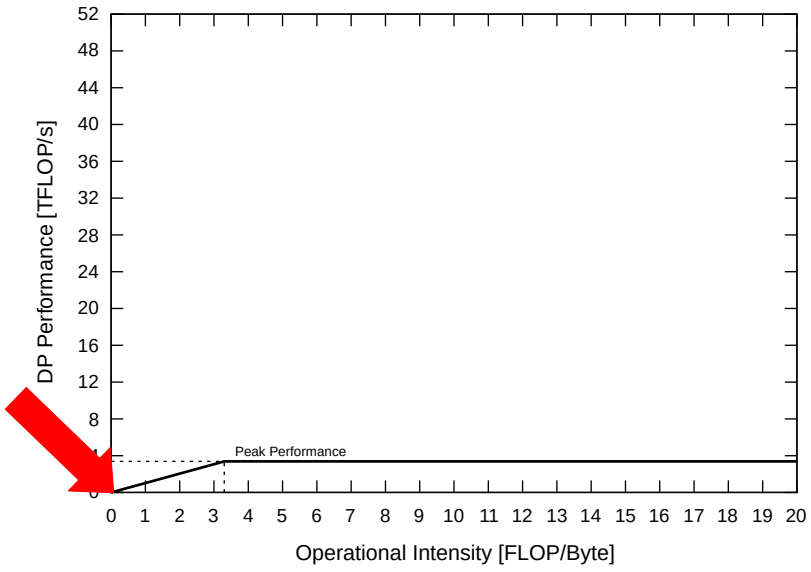
Equation of State in the Roofline Model Space

Roofline model for NEC SX-Aurora TSUBASA 30A



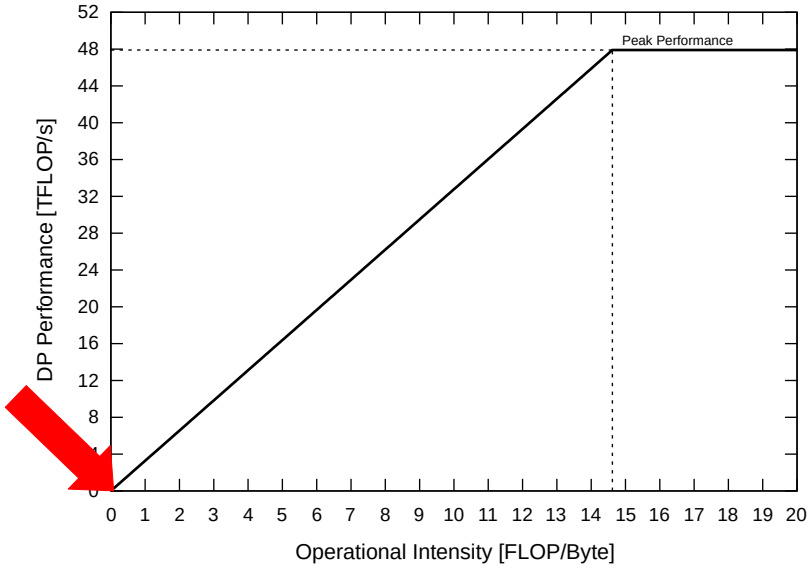
4.91 TFLOP/s
2.45 TB/s

Roofline model for Fujitsu A64FX



3.4 TFLOP/s
1.024 TB/s

Roofline model for AMD Instinct MI250X



47.9 TFLOP/s
3.2768 TB/s

Proof of Concept
Real-Time Lossy Array Compression Code (RTLAC)

H L R S

```
{
  "rtlac": [
    {
      "name": "rho",
      "type": "double",
      "bits":  32,
      "min":   1.155,
      "max":   1.204,
    },{
      "name": "Rs",
      "type": "double",
      "bits":  32,
      "min":   287.058,
      "max":   297.058,
    },{
      "name": "T",
      "type": "double",
      "bits":  32,
      "min":   293.15,
      "max":   333.15,
    },{
      "name": "p",
      "type": "double",
      "bits":  32,
      "min":    98000.0,
      "max":   120000.0,
    }
  ]
}
```

```
#include <stdint.h>

#ifndef RTLAC_DISABLE
[…] // RTLAC Code
#else
typedef double rho_ld_t;
#define rho_d2ld(v) (v)
#define rho_ld2d(v) (v)

typedef double Rs_ld_t;
#define Rs_d2ld(v) (v)
#define Rs_ld2d(v) (v)

typedef double T_ld_t;
#define T_d2ld(v) (v)
#define T_ld2d(v) (v)

typedef double p_ld_t;
#define p_d2ld(v) (v)
#define p_ld2d(v) (v)
#endif
```

Proof of Concept – Compute Kernel Code (Allocation)

```
printf("Allocating arrays...\n");
double   *rho    = (double*)malloc(n*sizeof(double));     // Density [kg/m^3]
float    *rho_sp = (float*)malloc(n*sizeof(float));       // Density [kg/m^3]
rho_ld_t *rho_ld = (rho_ld_t*)malloc(n*sizeof(rho_ld_t)); // Density [kg/m^3] (range 1.155...1.204)

double   *Rs     = (double*)malloc(n*sizeof(double));     // Ideal gas constant [J/(kg*K)]
float    *Rs_sp  = (float*)malloc(n*sizeof(float));       // Ideal gas constant [J/(kg*K)]
Rs_ld_t  *Rs_ld  = (Rs_ld_t*)malloc(n*sizeof(Rs_ld_t));   // Ideal gas constant [J/(kg*K)] (range 287.058...297.058)

double   *T      = (double*)malloc(n*sizeof(double));     // Temperature [K]
float    *T_sp   = (float*)malloc(n*sizeof(float));       // Temperature [K]
T_ld_t   *T_ld   = (T_ld_t*)malloc(n*sizeof(T_ld_t));     // Temperature [K] (range 293.15...333.15)

double   *p      = (double*)malloc(n*sizeof(double));     // Pressure [N/m^2]
float    *p_sp   = (float*)malloc(n*sizeof(float));       // Pressure [N/m^2]
p_ld_t   *p_ld   = (p_ld_t*)malloc(n*sizeof(p_ld_t));     // Pressure [N/m^2] (range 980000...1200000)
```

Proof of Concept – Compute Kernel Code (Initialization)

```
printf("Initializing arrays...\n");
// Populate and first-touch data
#pragma omp parallel for schedule(static)
for (uint64_t i = 0; i < n; i++) {
    rho[i]    = rho_max - (double)i/(double)n*(rho_max - rho_min);  //      1.204 ...      1.155 kg/m^3
    rho_sp[i] = (float)rho[i];
    rho_ld[i] = rho_d2ld(rho[i]);

    Rs[i]     = Rs_min + (double)i/(double)n*(Rs_max - Rs_min);     //    287.058 ...    297.058 J/(kg*K)
    Rs_sp[i]  = (float)Rs[i];
    Rs_ld[i]  = Rs_d2ld(Rs[i]);

    T[i]      = T_min + (double)i/(double)n*(T_max - T_min);        //    293.15  ...    333.15 °C
    T_sp[i]   = (float)T[i];
    T_ld[i]   = T_d2ld(T[i]);

    // output, just touch these up
    p[i]      = 0;
    p_sp[i]   = 0;
    p_ld[i]   = 0;                                                   // 98000.0   ... 120000.0
}
```

Proof of Concept – Compute Kernel Code (Calculation)

Reference (double)

```
void calc_p(double *rho_, double *Rs_, double *T_, double *p) {
#if defined(GPU)
  uint64_t i = GPU_IDX; if (i < n) {
#else
  #pragma omp parallel for schedule(static)
  for (uint64_t i = 0; i < n; i++) {
#endif
    double rho = rho_[i];
    double Rs  = Rs_[i];
    double T   = T_[i];
    double p_i;
    p_i = rho;
    p_i *= Rs;
    p_i *= T;
    p[i] = p_i;
  }
}
```

4 arrays replaced with **float**

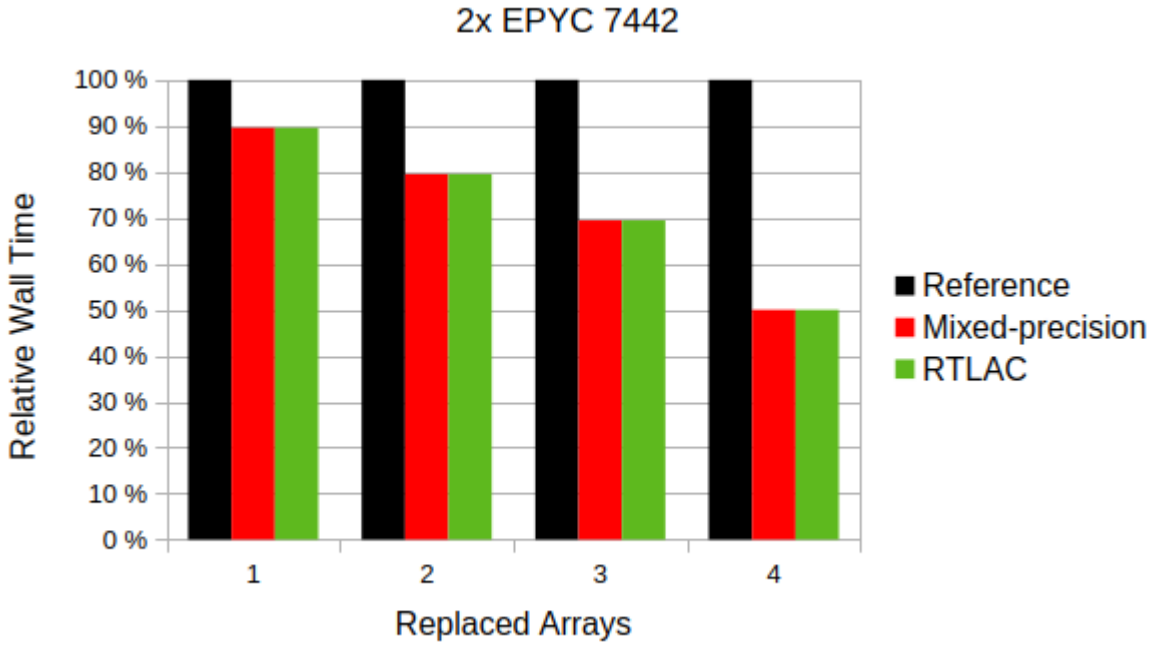
```
void calc_p_4sp(float *rho_sp, float *Rs_sp, float *T_sp, float *p_sp,
double *p) {
#if defined(GPU)
  uint64_t i = GPU_IDX; if (i < n) {
#else
  #pragma omp parallel for schedule(static) MAX_ERR_RED
  for (uint64_t i = 0; i < n; i++) {
#endif
    double rho = (double)rho_sp[i];
    double Rs  = (double)Rs_sp[i];
    double T   = (double)T_sp[i];
    double p_i;
    p_i  = rho;
    p_i *= Rs;
    p_i *= T;
    p_sp[i] = (float)p_i;
    MAX_ERR(p_i = (double)p_sp[i]; if (fabs(p[i] - p_i) > p_max_err)
p_max_err = fabs(p[i] - p_i));
  }
}
```

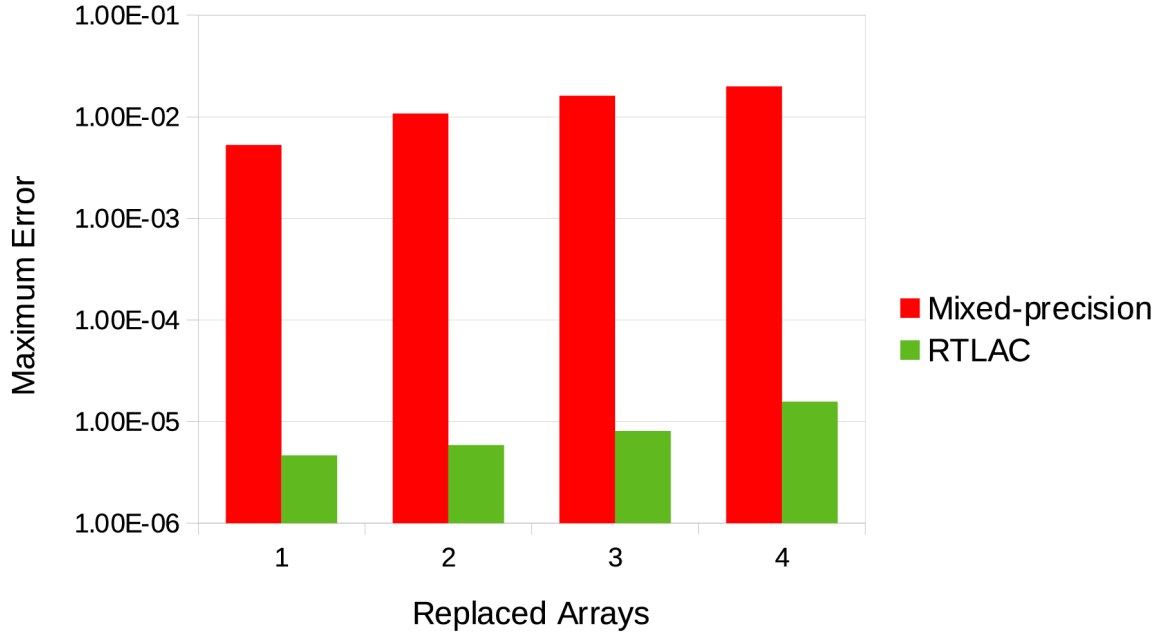
4 arrays replaced with **lossy double**

```
void calc_p_4ld(rho_ld_t *rho_ld, Rs_ld_t *Rs_ld, T_ld_t *T_ld, p_ld_t
*p_ld, double *p) {
#if defined(GPU)
  uint64_t i = GPU_IDX; if (i < n) {
#else
  #pragma omp parallel for schedule(static) MAX_ERR_RED
  for (uint64_t i = 0; i < n; i++) {
#endif
    double rho = rho_ld2d(rho_ld[i]);
    double Rs  = Rs_ld2d(Rs_ld[i]);
    double T   = T_ld2d(T_ld[i]);
    double p_i;
    p_i  = rho;
    p_i *= Rs;
    p_i *= T;
    p_ld[i] = p_d2ld(p_i);
    MAX_ERR(p_i = p_ld2d(p_ld[i]); if (fabs(p[i] - p_i) > p_max_err)
p_max_err = fabs(p[i] - p_i));
  }
}
```

Proof of Concept Compute Kernel Code w/ 128GB Hawk Node Results – Wall Time & Max. Error

```
OMP_NUM_THREADS=128
     OMP_PLACES=cores
Reference time (double): 0.278s
```

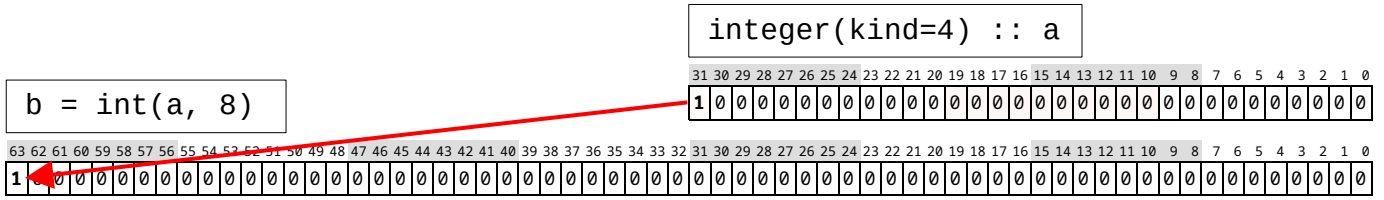




Fortran Port (Native Module + C Interface Module) Auto-generated

```
module rtlac
  implicit none
  contains
#if !defined(RTLAC_DISABLE)
  pure integer(kind=4) function rho_d2ld(v)
    implicit none
    real(kind=8), intent(in) :: v
    real(kind=8)             :: vr
    integer(kind=8)          :: vi
    equivalence(vr,vi)
    vr = v
    rho_d2ld = int(ishft(vi, -17), 4)
  end function rho_d2ld

  pure real(kind=8) function rho_ld2d(v)
    implicit none
    integer(kind=4), intent(in)   :: v
    integer(kind=4), dimension(2) :: v2
    integer(kind=8)               :: vi
    real(kind=8)                  :: rr
    integer(kind=8)               :: ri
    equivalence(v2,vi)
    equivalence(rr,ri)
    vi     = 0
    v2(1)  = v ! Little endian
    ri     = ior(ishft(vi, 17), 4607745368753504256_8)
    rho_ld2d = rr
  end function rho_ld2d
  […] ! More functions
#else
  […] ! Disable RTLAC Code
#endif
```

Technology for unsigned integers is just not there yet for Fortran:



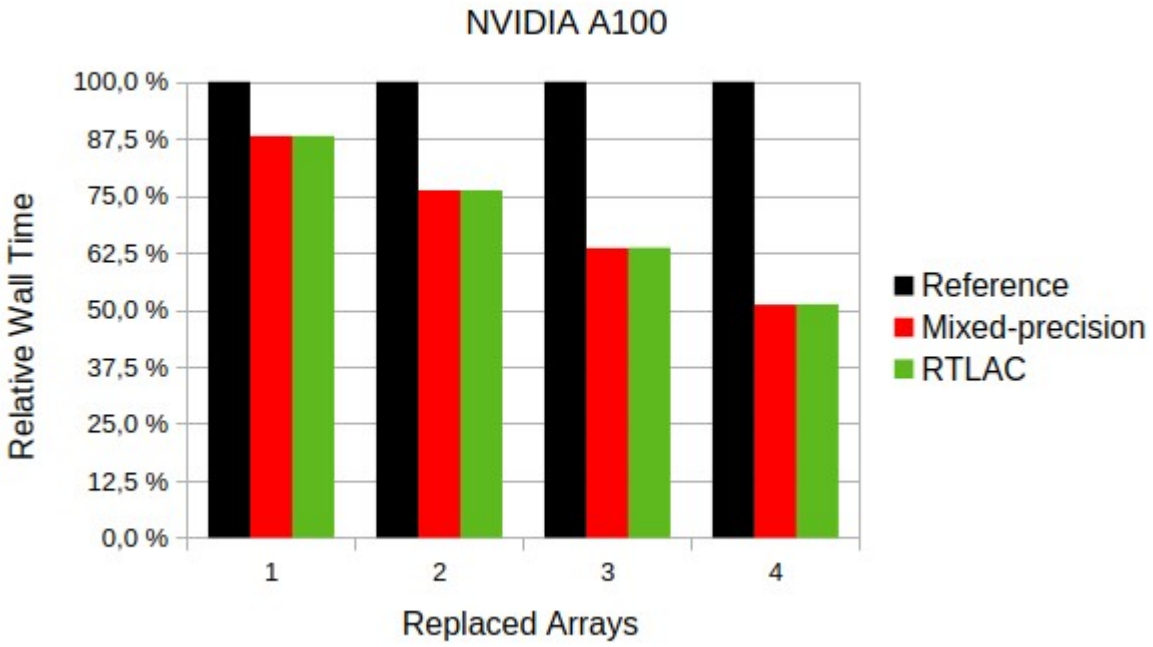

```
module rtlac_cif
  implicit none
  contains
#if !defined(RTLAC_DISABLE)
  integer(kind=4) function rho_d2ld(v) bind(C, name = 'rho_d2ld')
    use, intrinsic        :: iso_c_binding
    real(c_double), value :: v
  end function rho_d2ld

  real(kind=8) function rho_ld2d(v) bind(C, name = 'rho_ld2d')
    use, intrinsic         :: iso_c_binding
    integer(kind=4), value :: v
  end function rho_ld2d

  […] ! More functions
#else
  […] ! Disable RTLAC Code
#endif
```

Compiler Performance (Hawk x86_64 128GB)

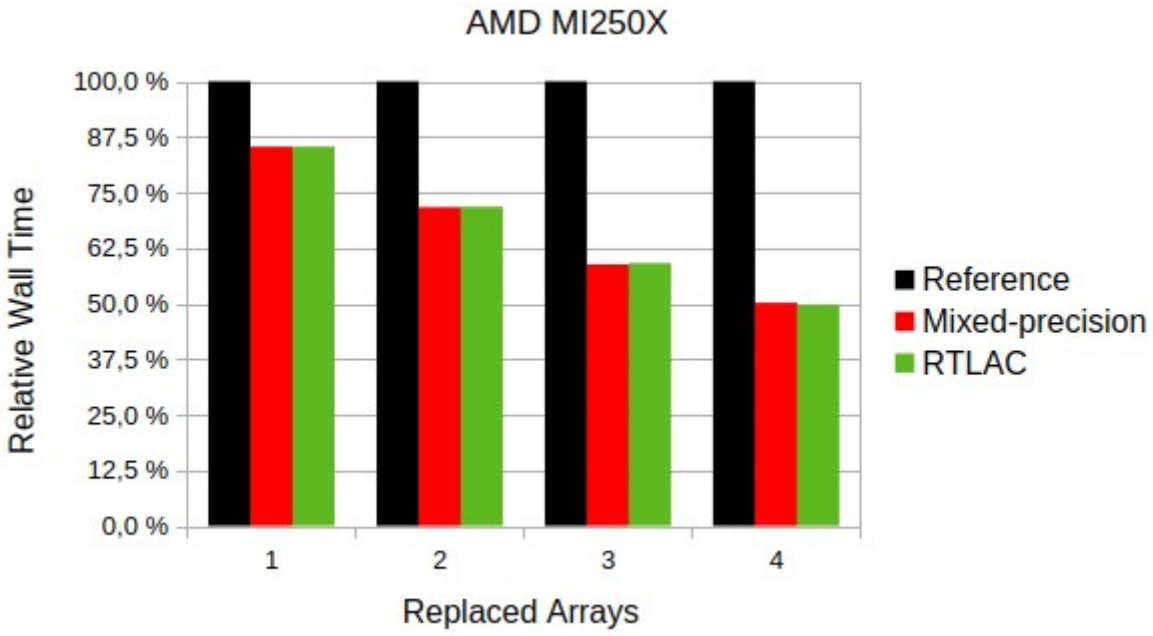
					Replaced Arrays				
			inlines	vectorizes	1	2	3	4	comment
LTO	C	gcc 10.2.0	yes	yes	0.249	0.222	0.193	0.139	
		aocc 4.1.0	yes	yes	0.249	0.222	0.194	0.139	
		icc 19.1.3	yes	no	0.250	0.221	0.193	0.139	
		cc 17.0.0	yes	yes	0.250	0.221	0.192	0.138	
	Fortran	gcc 10.2.0	yes	no	0.250	0.222	0.193	0.139	
		aocc 4.1.0	yes	yes	0.249	0.221	0.193	0.139	
		ifort 19.1.3	yes	no	0.250	0.221	0.192	0.139	
		ftn 17.0.0	yes	no	x	x	x	x	Did not finish (~1000x slower)
	Fortran (CIF)	gcc 10.2.0	yes	no	0.249	0.221	0.193	0.139	
		aocc 4.1.0	yes	yes	0.249	0.221	0.193	0.139	
		ifort 19.1.3	no	no	0.249	0.221	0.193	0.189	
		ftn 17.0.0	x	x	x	x	x	x	Did not compile (different LTO impl.)
!LTO	C	gcc 10.2.0	yes	yes	0.249	0.222	0.193	0.139	
		aocc 4.1.0	yes	yes	0.248	0.222	0.193	0.139	
		icc 19.1.3	yes	no	0.250	0.222	0.193	0.139	
		cc 17.0.0	yes	yes	0.250	0.221	0.192	0.138	
	Fortran	gcc 10.2.0	no	no	0.249	0.221	0.194	0.165	
		aocc 4.1.0	no	no	0.249	0.221	0.194	0.179	
		ifort 19.1.3	no	no	0.249	0.221	0.193	0.157	
		ftn 17.0.0	yes	no	x	x	x	x	Did not finish (~1000x slower)
	Fortran (CIF)	gcc 10.2.0	no	no	0.250	0.222	0.194	0.168	
		aocc 4.1.0	no	no	0.249	0.221	0.194	0.176	
		ifort 19.1.3	no	no	0.249	0.221	0.194	0.175	
		ftn 17.0.0	x	x	x	x	x	x	Did not compile (link error)

Performance on Accelerators
AMD/NVIDIA, HIP, 32GB Data

Reference time (double): 0.012s

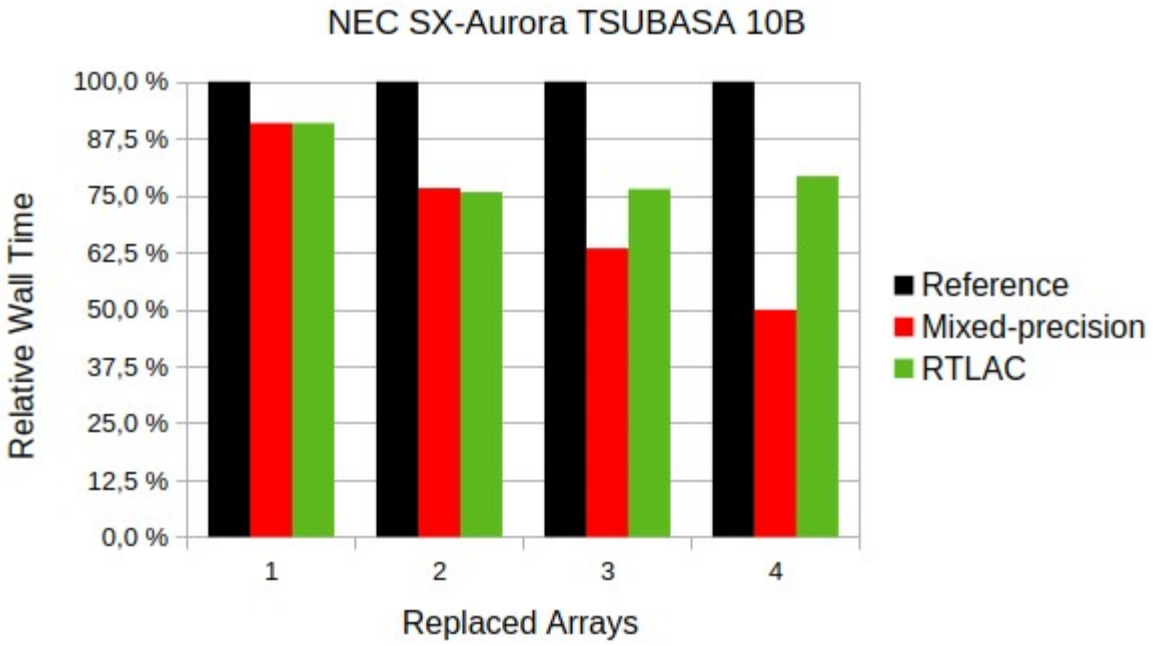


Reference time (double): 0.014s



Performance on Accelerators
NEC, 32GB Data

C + OpenMP
Reference time (double): 0.018s



Compiler generates instructions with too much overhead
→ Ticket open at NEC

Fortran (native) + OpenMP
Reference time (double): 0.018s

```
========================================
 lossy double precision
========================================

calc_p_1ld: 1.771s
calc_p_2ld: 3.349s
calc_p_3ld: 5.684s
calc_p_4ld: 3.504s
```

Compiler doesn't seem to work properly with how this is implemented

Overview

- Introduction: Memory Bandwidth Problems
- State of Technology – Roofline Model
- State of Technology – Remedies
- New Approach – Proof of Concept
- Outlook

Outlook

Outlook – Topics

- Proof of concept looks promising for C and Fortran
- Further developing a (header-only) library that provides the functionality:
  - Extend suppport for 24-/40-/48-/56-bit storage precisions for both AoS/SoA
  - Experiment with adaptive value ranges at run-time per subdomain with periodical updates
- Analyze feasibility with different (memory bandwidth bound) simulation codes for actual numerics:
  - m-AIA (formerly ZFS) C++ AIA RWTH Aachen
  - NS3d neo Fortran IAG Uni Stuttgart
- Bring the idea across to save resources limited by memory bandwidth
- Get compiler developers to optimize around this more (especially Fortran)

The End

Thank you for your attention.

Questions, Feedback