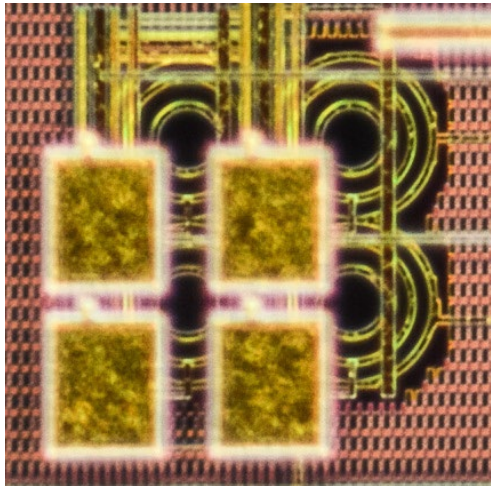
THE UNIVERSITY *of* EDINBURGH

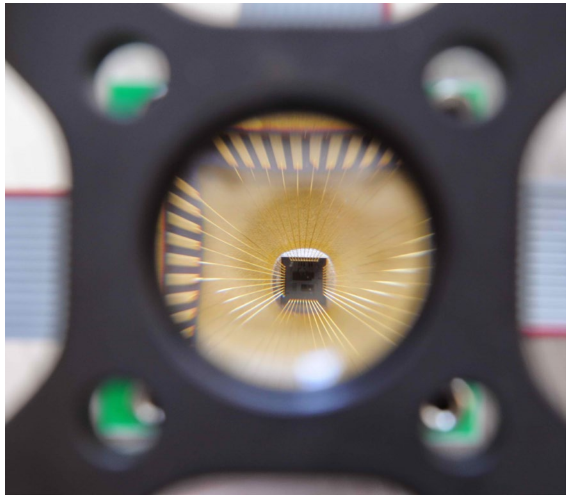
Acceleration of sparse matrix linear solvers for circuit simulation

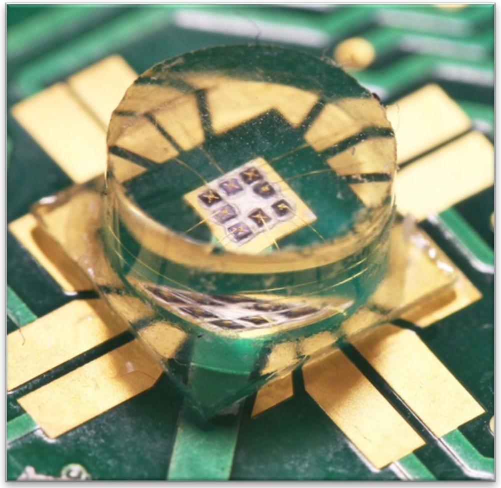
Yichen Zhang, Salman Saiful Redzuan, and **Danial Chitnis**

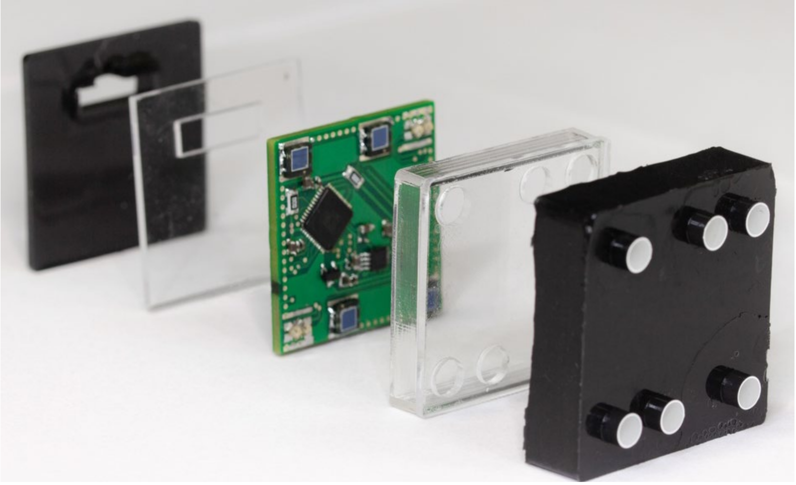
August 2022

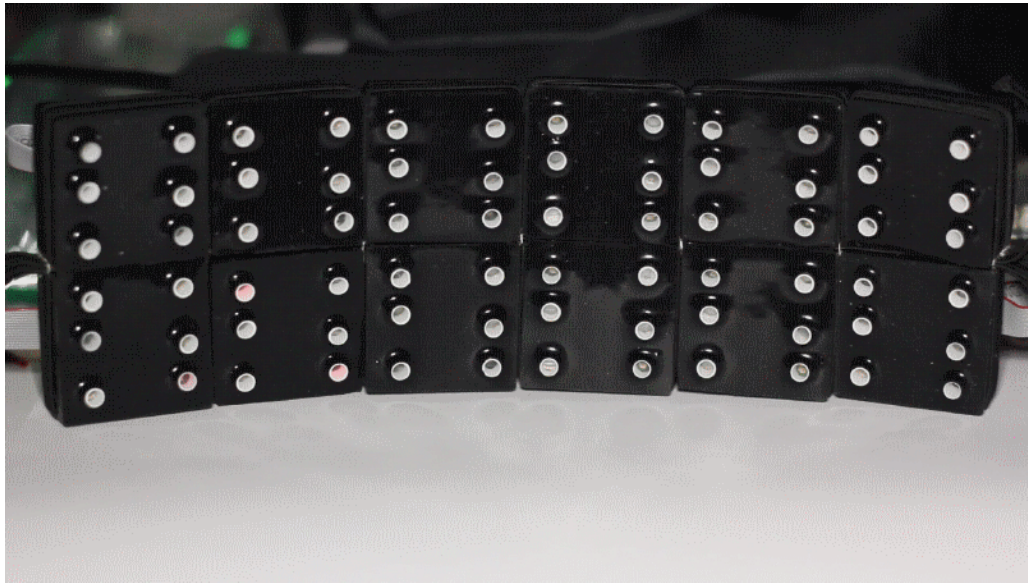
My Background

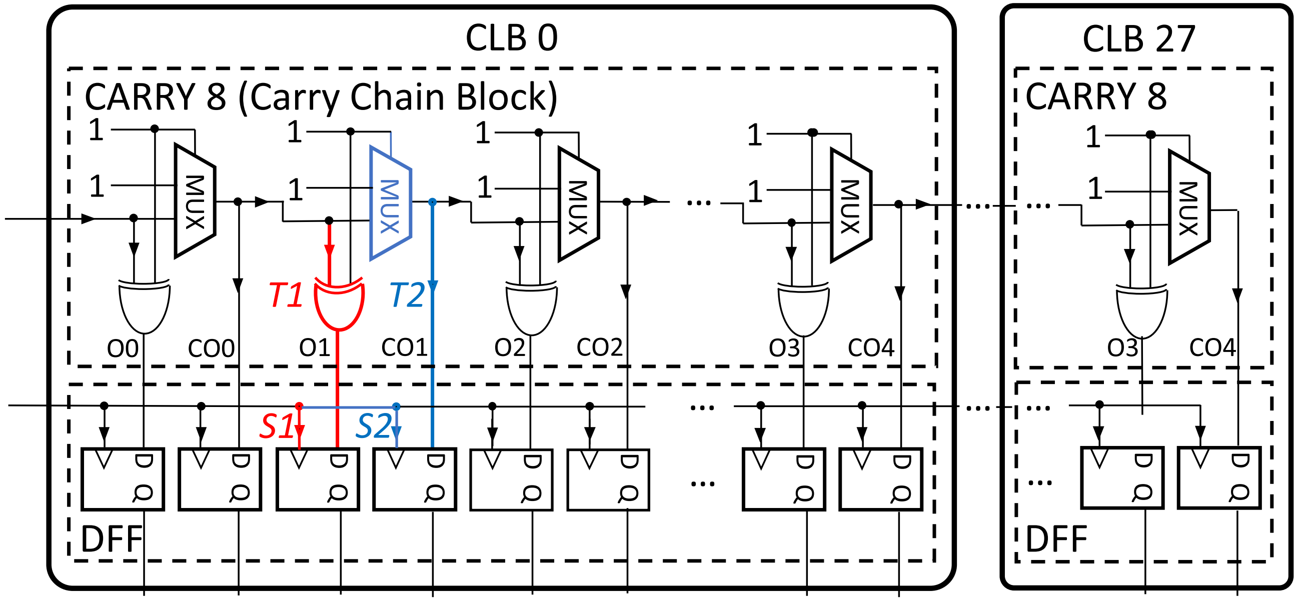






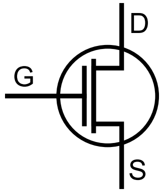


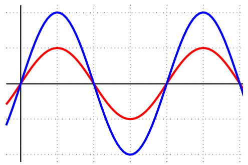


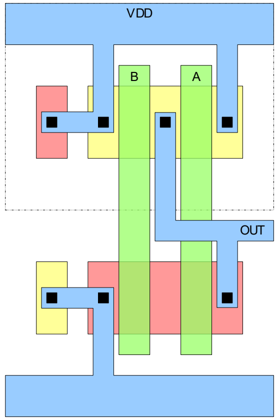


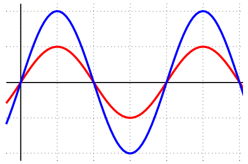
Simulation of Integrated Circuits

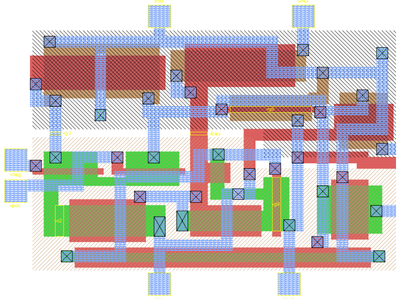
- Simulation of chips with billions of transistors
- Transistor level required for pushing the limits of transistors
- Simulation of a block could take from 1 ms to few days
- full-chip post-layout simulation near impossible

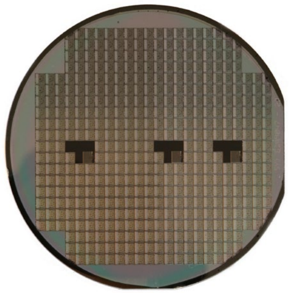


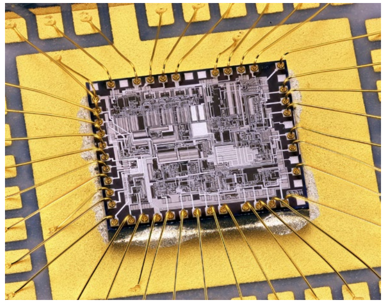












Simulation of Integrated Circuits

Statistical variation on matrix *G* (lhs matrix)

Examples:

Manufacturing tolerance on transistor length

Parametric optimization of transistor widths

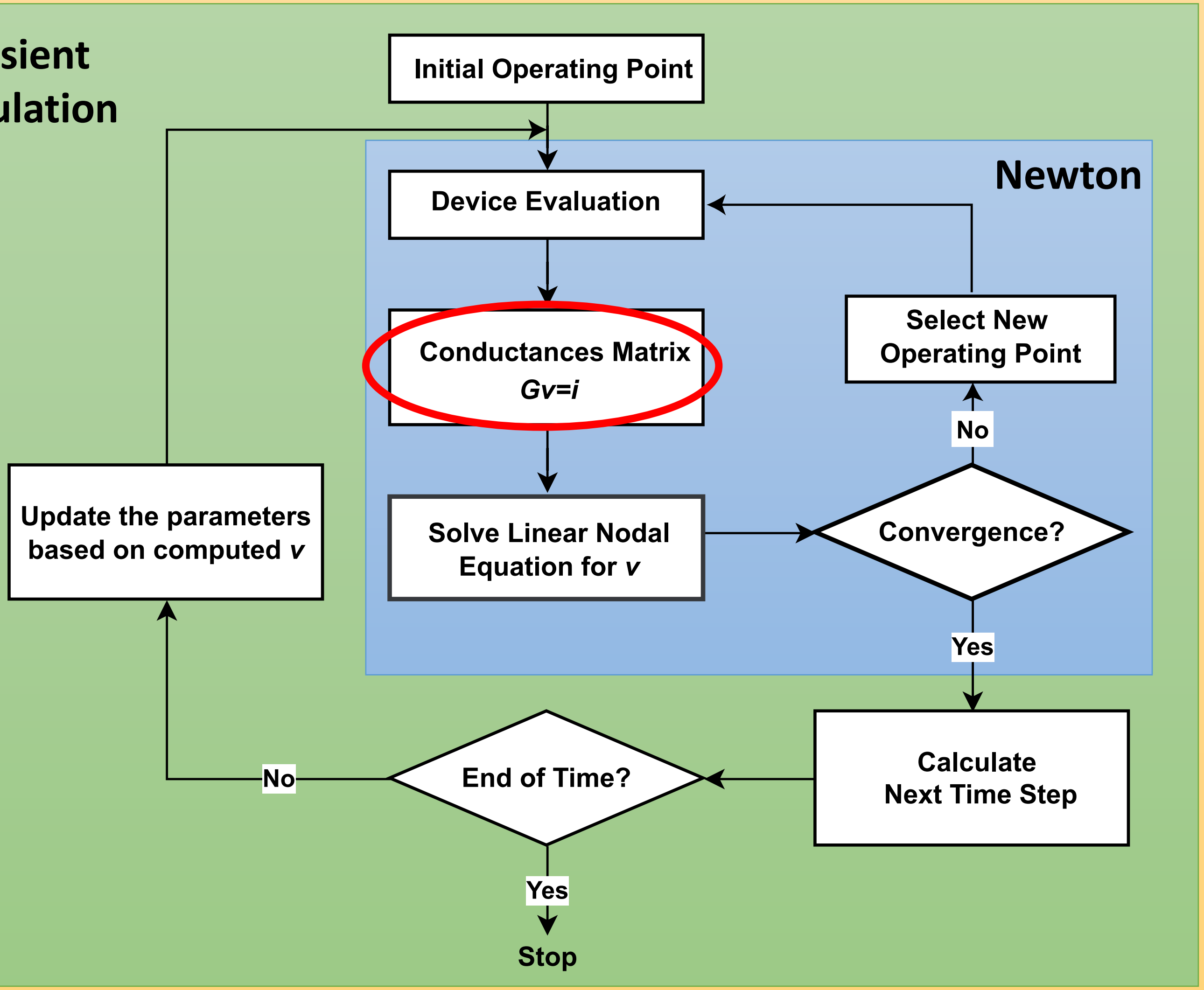
Statistical variation on matrix *i* (rhs matrix)

Examples:

Noise of power supply network

Input bias voltage for a PLL circuit

Transient Simulation



Initial Operating Point

Newton

Device Evaluation

Conductances Matrix
Gv=i

Select New Operating Point

No

Solve Linear Nodal Equation for *v*

Convergence?

Yes

Update the parameters based on computed *v*

Calculate Next Time Step

End of Time?

No

Yes

Stop

Linear Solvers - Current state-of-art

- Create an algorithm – CPU, GPU, or FPGA
- Marginal improvements ~2x
- Narrow usage (i.e. mathematician or chip designer?)
- Only few test runs, claim only large circuits need acceleration
- No source code available in some cases (only binaries)

Our Answer – speed ups

Marginal improvements ~2x

- Small speedups are not motivation for a mature community to make substantial changes to their workflow

Narrow usage (i.e. mathematician or chip designer?)

- In chip design statistical simulations are an essential part design:
  - device variations (transistor sizes)
  - Input parameter sensitivity (e.g. supply and bias voltages)
- These create “independent” simulations and inherently parallel
- **How many simulations can we bundle for maximum efficiency?**

Our Answer – test inputs

Only few test runs, claim only large circuits need acceleration

- We are running across ~110 circuit matrices with various sizes as part of Tim Davis collection of sparse matrices.
- This collection is the only available sparse matrix collection so far.
- Not easy to obtain meaningful circuit netlists due IP and legal limitations.
- **Small circuits are equally important because they are often more sensible to optimise and more often used**

Our Answer – portability

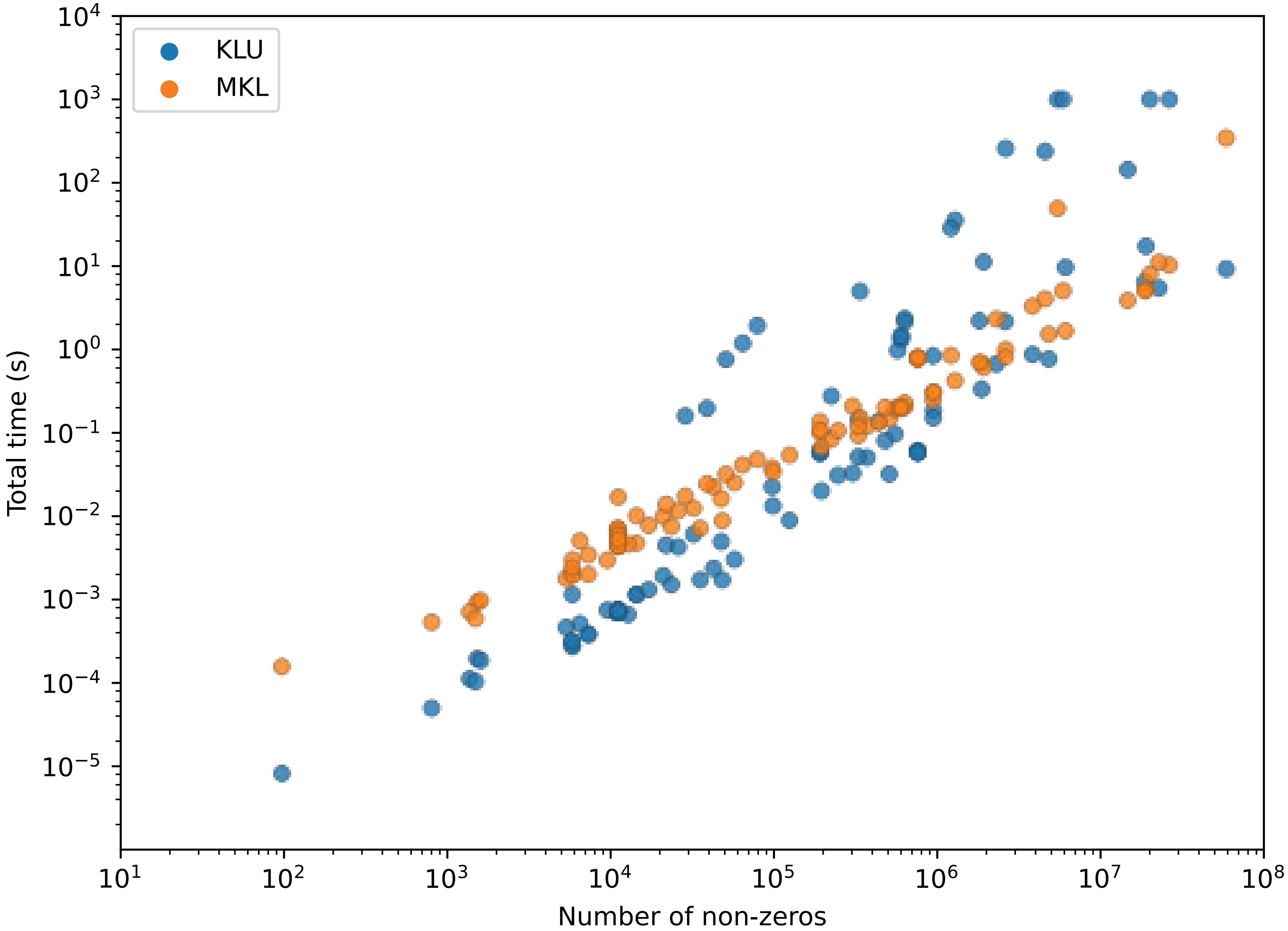
No source code available in some cases (only binaries)

- We use KLU linear solver for sparse circuit matrices (part of Tim Davis's SuiteSparse)
- Full KLU code available and continuously maintained
- Full KLU’s dependant libraries also available
- It has multiple open-source licenses
- **This is essential for long term development and custom implementations across heterogenous platform**
- (oneMKL – Pardiso is also a good royalty-free option but still no source code available and only on Intel x86 CPU)

KLU vs oneMKL

- Original KLU
- 110 matrices

- Core i7-9700
- GCC 11.2.0



Linear Solvers - KLU

Inner workings of KLU:

- **Symbolic**: computed only once per pattern of non-zero elements of ***A***
- **Numeric**: computed each time value of non-zero elements changes of ***A***
- **Solve**: computed each time value of right-hand side matrix changes of ***b***

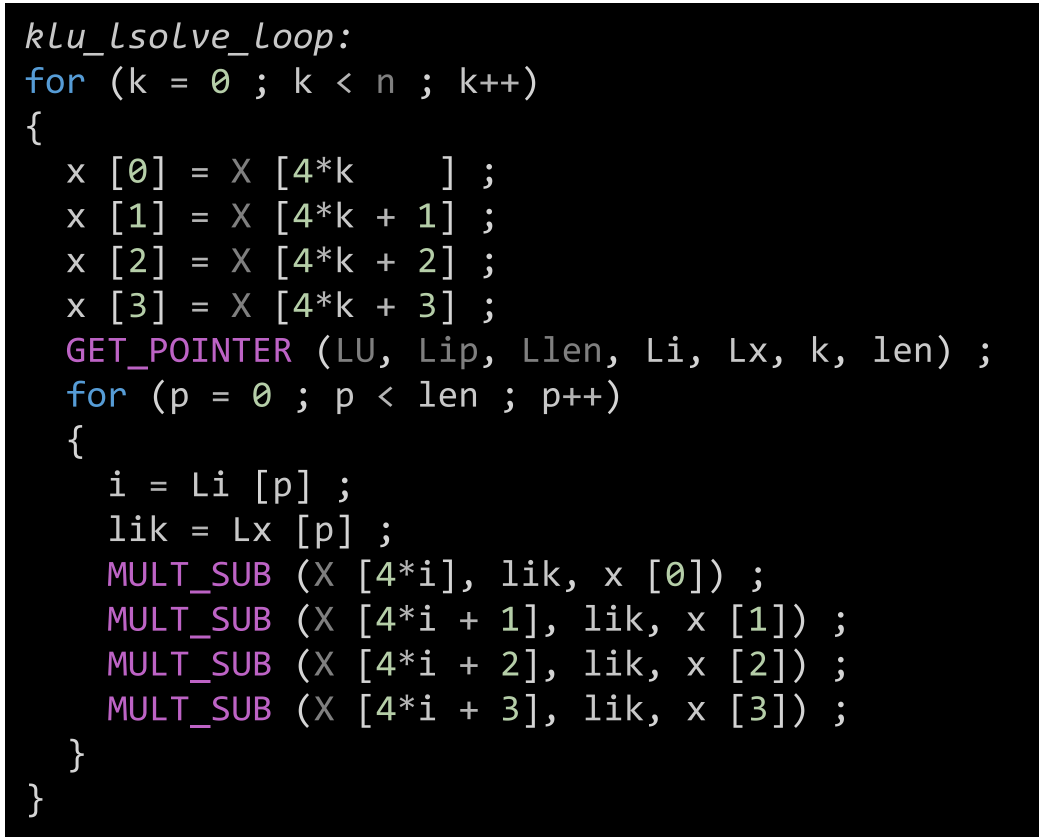
$$\text{lhs} \quad \boxed{\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \boxed{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}} \quad \text{rhs}$$

$$\boldsymbol{Ax} = \boldsymbol{b}$$

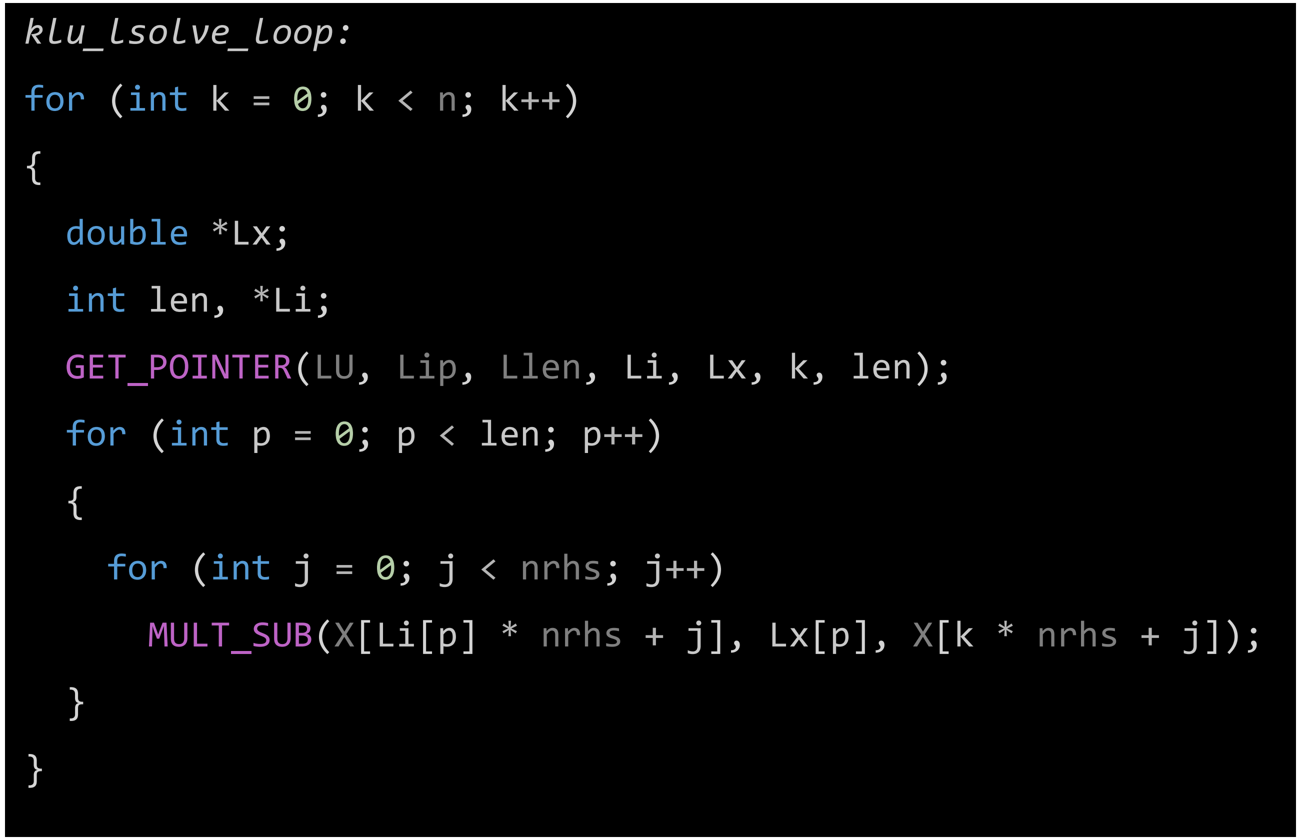
Introducing “nrhs”

Original KLU code

modified KLU code



```
klu_lsolve_loop:
for (k = 0 ; k < n ; k++)
{
  x [0] = X [4*k    ] ;
  x [1] = X [4*k + 1] ;
  x [2] = X [4*k + 2] ;
  x [3] = X [4*k + 3] ;
  GET_POINTER (LU, Lip, Llen, Li, Lx, k, len) ;
  for (p = 0 ; p < len ; p++)
  {
    i = Li [p] ;
    lik = Lx [p] ;
    MULT_SUB (X [4*i], lik, x [0]) ;
    MULT_SUB (X [4*i + 1], lik, x [1]) ;
    MULT_SUB (X [4*i + 2], lik, x [2]) ;
    MULT_SUB (X [4*i + 3], lik, x [3]) ;
  }
}
```



```
klu_lsolve_loop:
for (int k = 0; k < n; k++)
{
  double *Lx;
  int len, *Li;
  GET_POINTER(LU, Lip, Llen, Li, Lx, k, len);
  for (int p = 0; p < len; p++)
  {
    for (int j = 0; j < nrhs; j++)
      MULT_SUB(X[Li[p] * nrhs + j], Lx[p], X[k * nrhs + j]);
  }
}
```

Introducing Thread Pool

- Thread pool based on pthreads
- https://github.com/bshoshany/thread-pool
- Standard C++17 with no dependencies

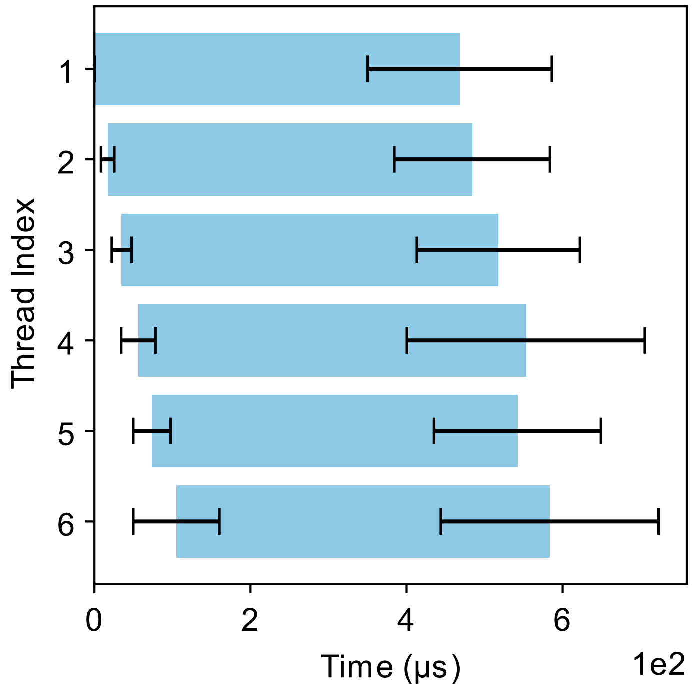
```
Create thread pool
Do a dummy task
Wait for all task to finish

for (reps) {
  reset matrices
  start timer_factor

  for (threads) {
     submit factorize(A) to the thread pool
  }

  stop timer_factor
  wait for all tasks to finish

  start timer_solve

  for (threads) {
     submit to the thread pool [](
        for (nrhs)
           solve(b)
      )
  }

  stop timer_solve
  wait for all tasks to finish
}
```

Threaded performance

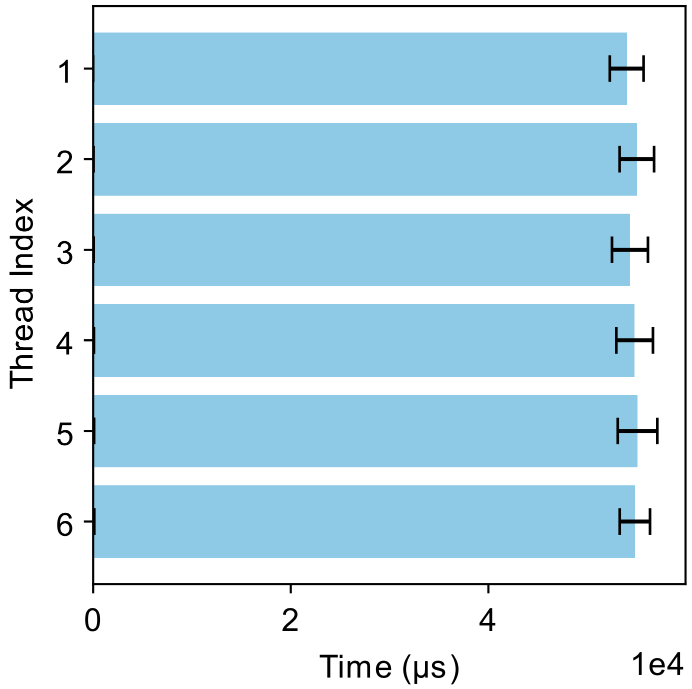
- Micro-benchmarking each thread from a reference origin (for “solve” only)

Intel Core i7-9700, 8-cores, 8-threads

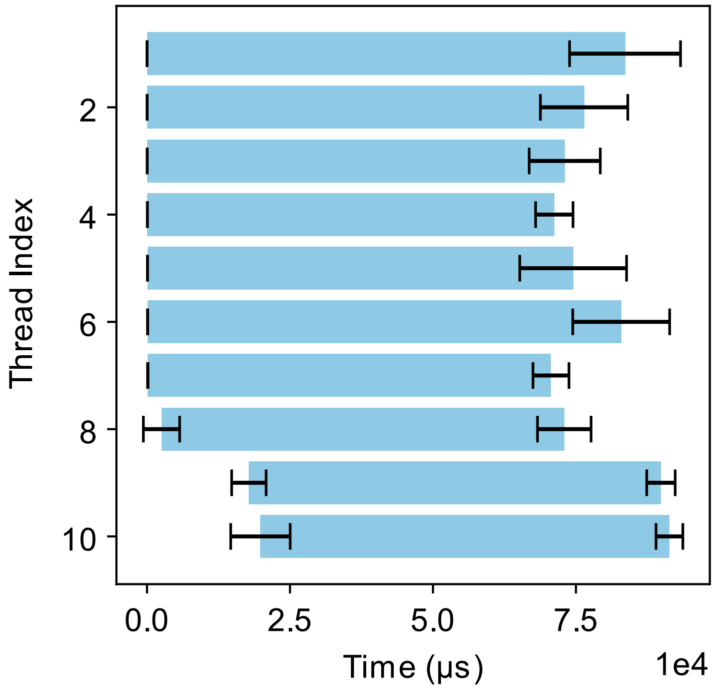
circuit_2 (nnz=21k)
6 thread / 8 core



scircuit (nnz=960k)
6 thread / 8 core



scircuit (nnz=960k)
10 thread / 8 core

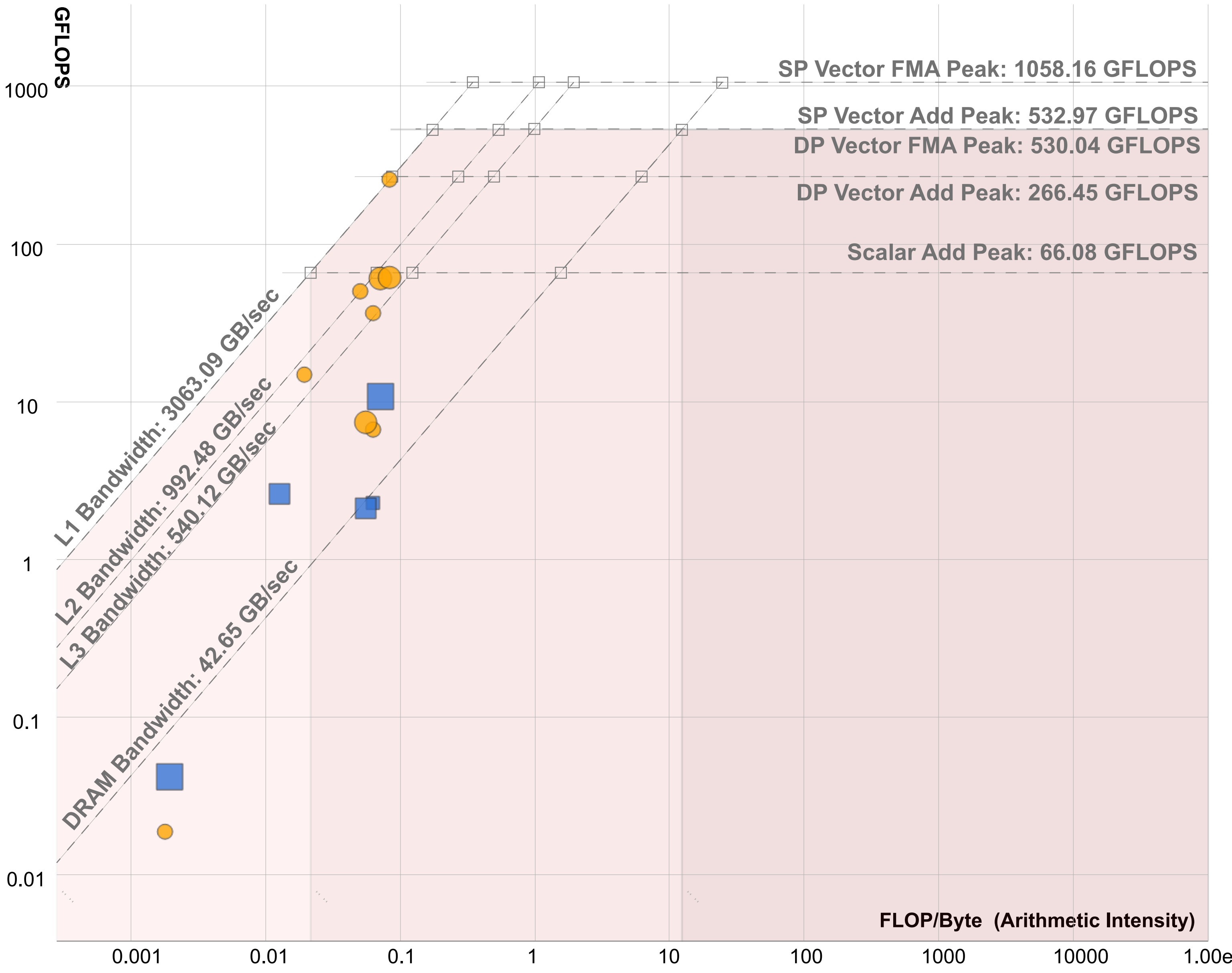


Roofline Model – Before & After

● After threading and nrhs

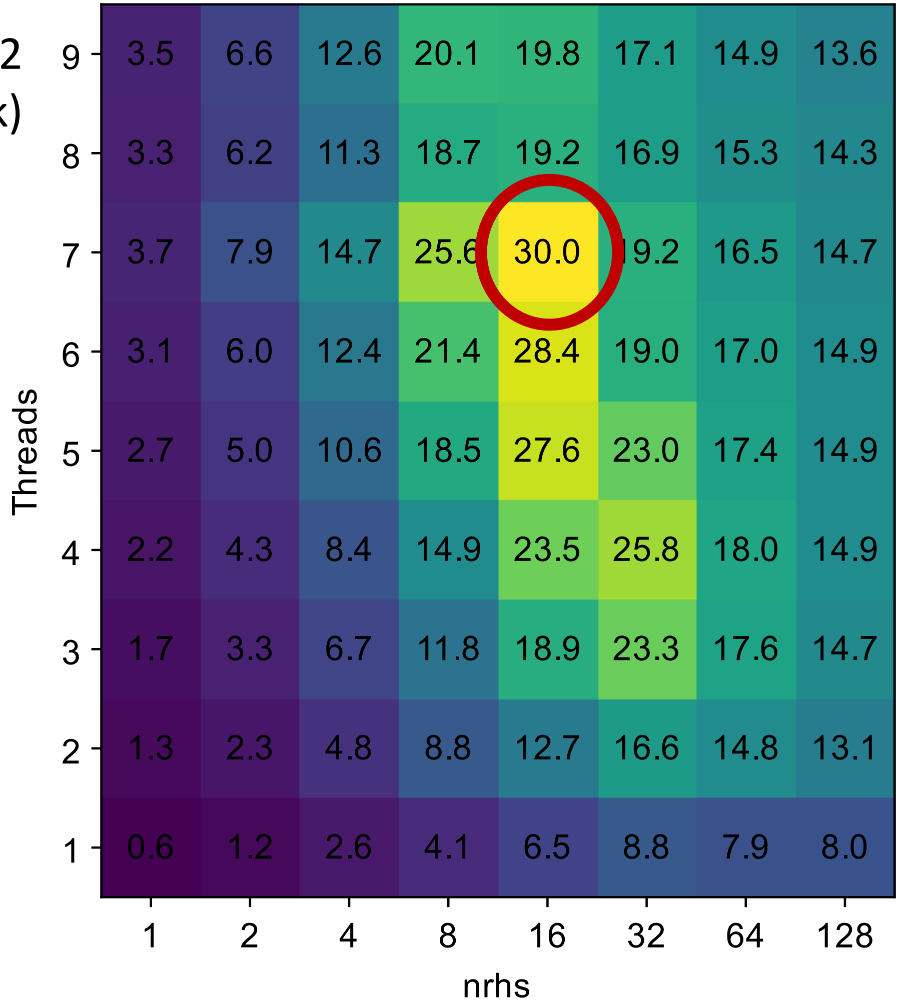
■ Single thread nrhs=1

- Significant improvement in memory efficiency reaching L1 speeds

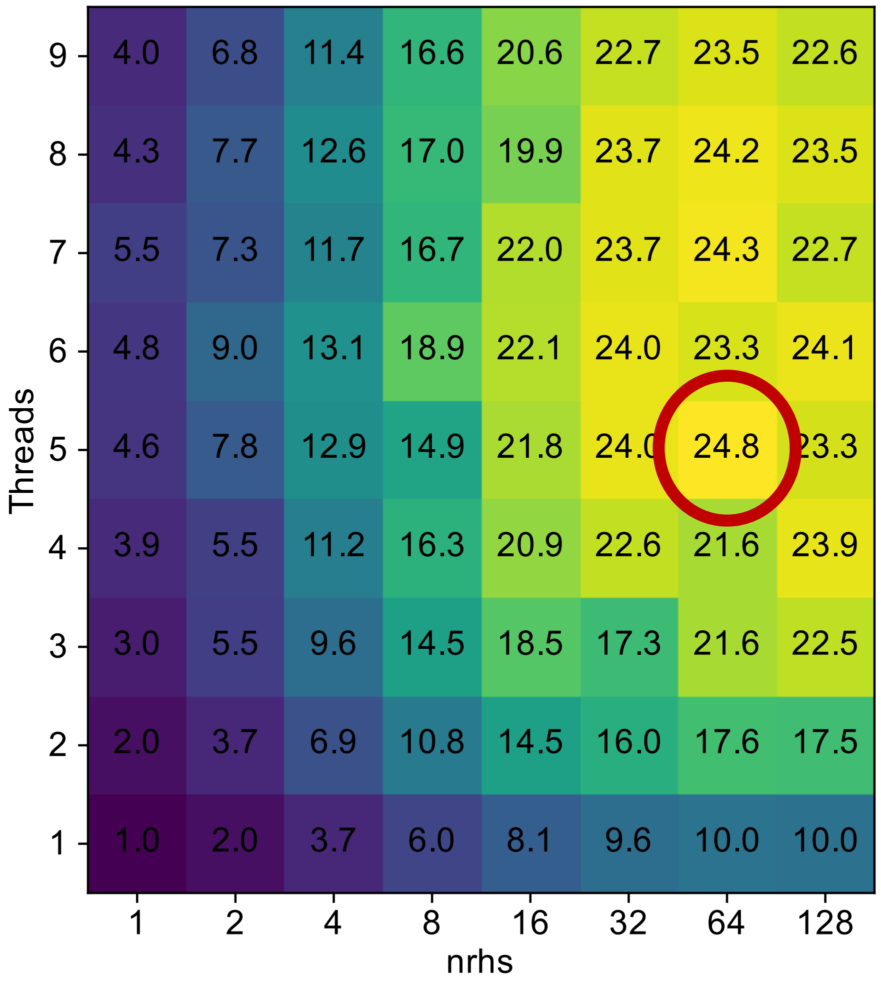


Relative to single threaded (without pool and nrhs=1)

circuit_2
(nnz=21k)



Threads \ nrhs	1	2	4	8	16	32	64	128
9	3.5	6.6	12.6	20.1	19.8	17.1	14.9	13.6
8	3.3	6.2	11.3	18.7	19.2	16.9	15.3	14.3
7	3.7	7.9	14.7	25.6	30.0	19.2	16.5	14.7
6	3.1	6.0	12.4	21.4	28.4	19.0	17.0	14.9
5	2.7	5.0	10.6	18.5	27.6	23.0	17.4	14.9
4	2.2	4.3	8.4	14.9	23.5	25.8	18.0	14.9
3	1.7	3.3	6.7	11.8	18.9	23.3	17.6	14.7
2	1.3	2.3	4.8	8.8	12.7	16.6	14.8	13.1
1	0.6	1.2	2.6	4.1	6.5	8.8	7.9	8.0



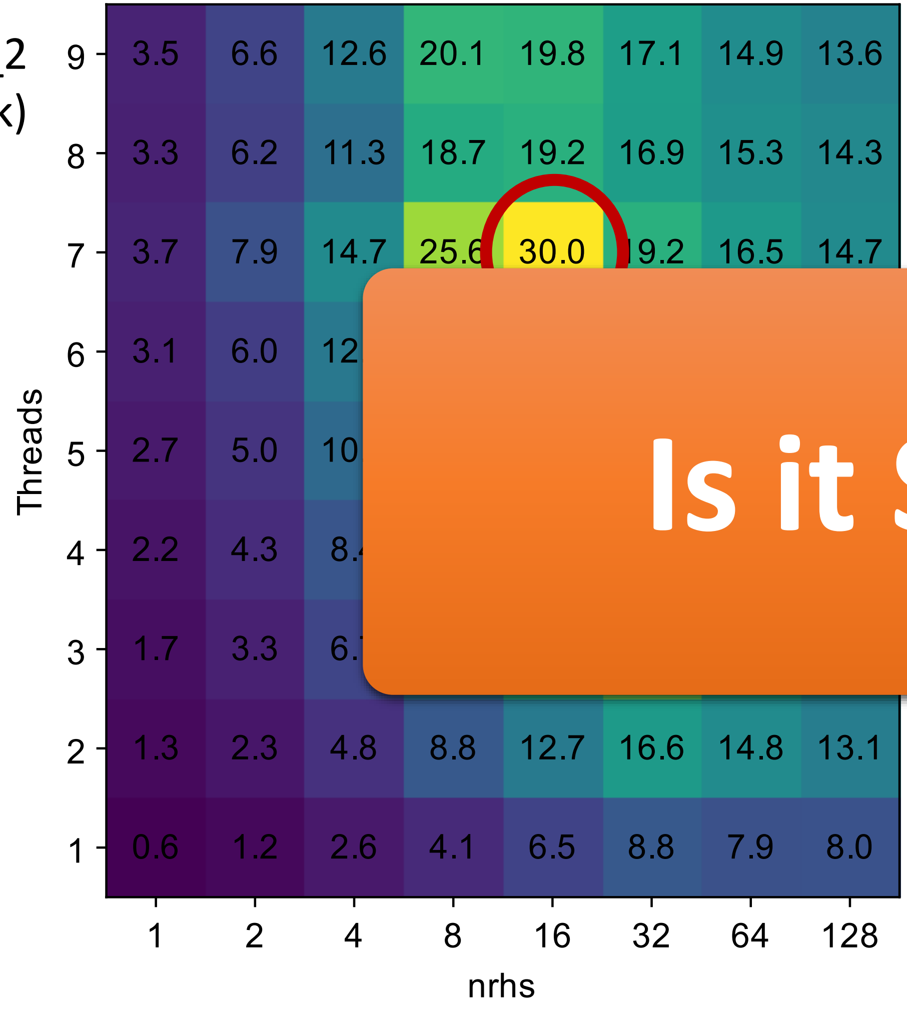
Threads \ nrhs	1	2	4	8	16	32	64	128
9	4.0	6.8	11.4	16.6	20.6	22.7	23.5	22.6
8	4.3	7.7	12.6	17.0	19.9	23.7	24.2	23.5
7	5.5	7.3	11.7	16.7	22.0	23.7	24.3	22.7
6	4.8	9.0	13.1	18.9	22.1	24.0	23.3	24.1
5	4.6	7.8	12.9	14.9	21.8	24.0	24.8	23.3
4	3.9	5.5	11.2	16.3	20.9	22.6	21.6	23.9
3	3.0	5.5	9.6	14.5	18.5	17.3	21.6	22.5
2	2.0	3.7	6.9	10.8	14.5	16.0	17.6	17.5
1	1.0	2.0	3.7	6.0	8.1	9.6	10.0	10.0

scircuit
(nnz=960k)

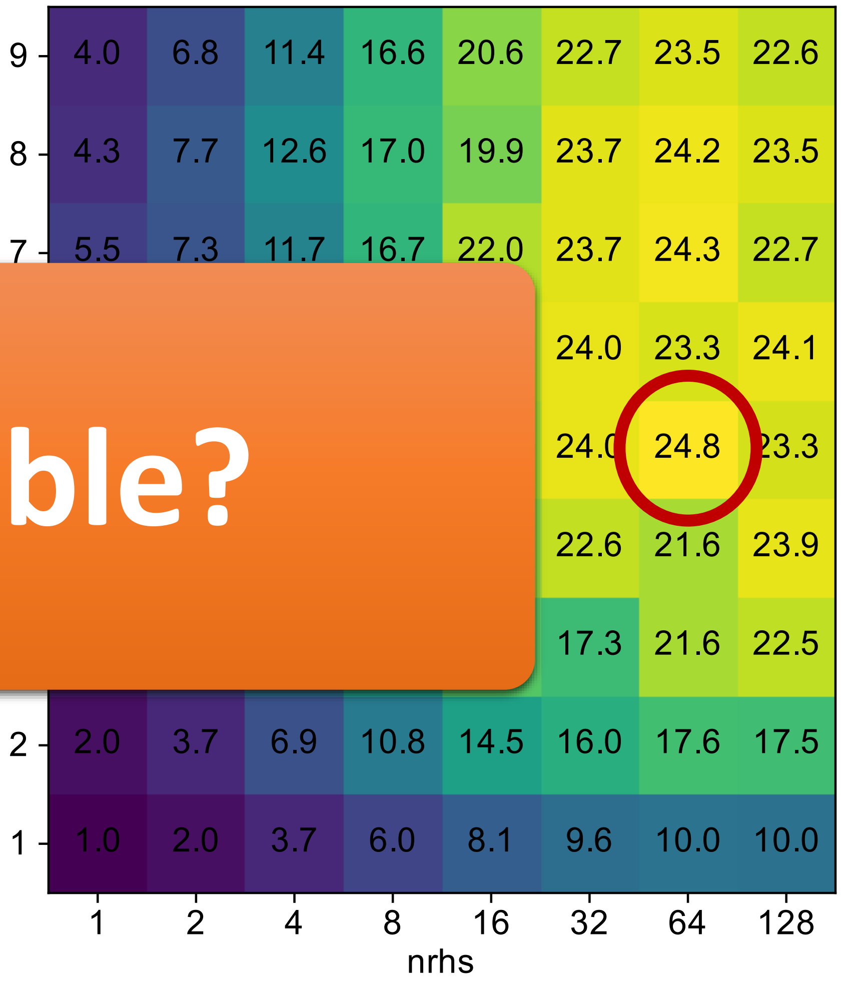
- Values are relative speed-up versus single threaded without pool and nrhs-1
- Core i7-9700 and GCC 11.2.0

Relative to single threaded (without pool and nrhs=1)

circuit_2
(nnz=21k)



Is it Scalable?

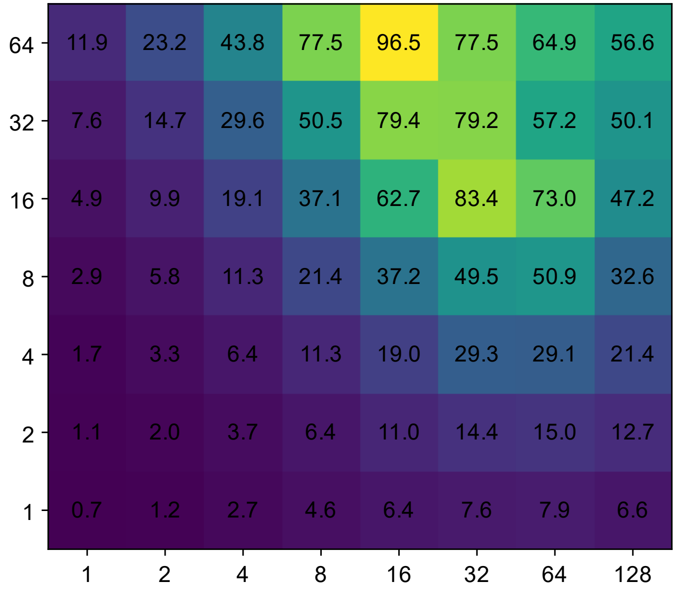


scircuit
(nnz=960k)

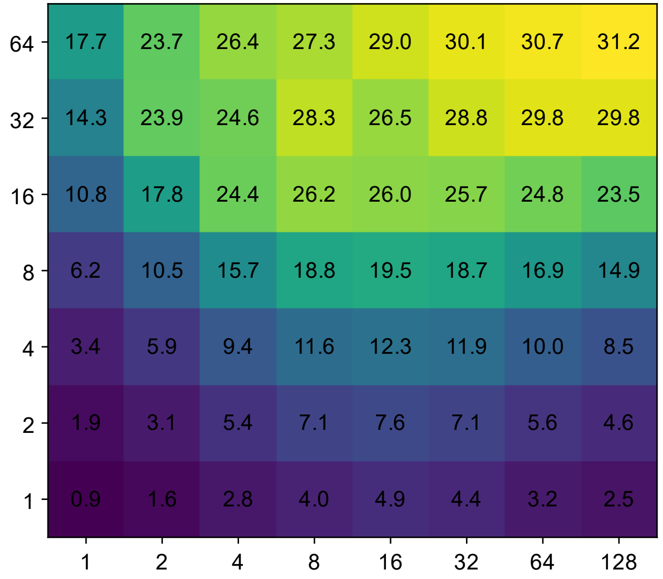
- Values are relative speed-up versus single threaded without pool and nrhs-1
- Core i7-9700 and GCC 11.2.0

Scalability vs Matrices (AWS c6i – Intel 3rd gen 64 vCPU)

circuit_2
(nnz=21k)



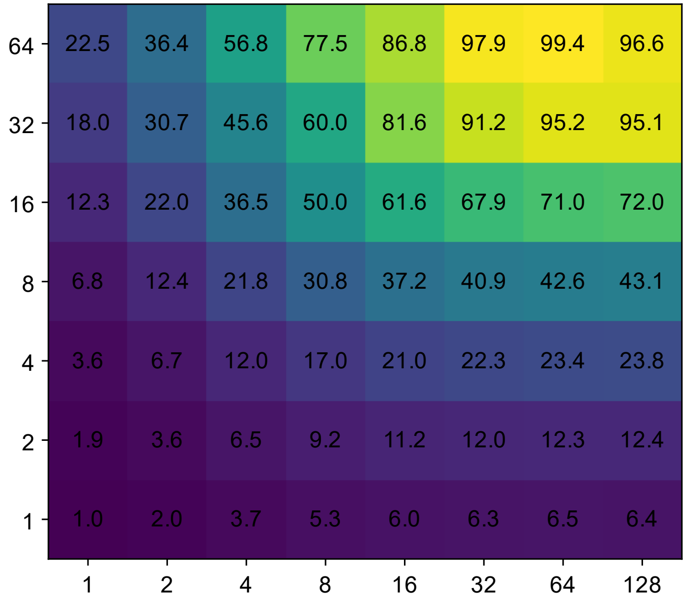
	1	2	4	8	16	32	64	128
64	11.9	23.2	43.8	77.5	96.5	77.5	64.9	56.6
32	7.6	14.7	29.6	50.5	79.4	79.2	57.2	50.1
16	4.9	9.9	19.1	37.1	62.7	83.4	73.0	47.2
8	2.9	5.8	11.3	21.4	37.2	49.5	50.9	32.6
4	1.7	3.3	6.4	11.3	19.0	29.3	29.1	21.4
2	1.1	2.0	3.7	6.4	11.0	14.4	15.0	12.7
1	0.7	1.2	2.7	4.6	6.4	7.6	7.9	6.6



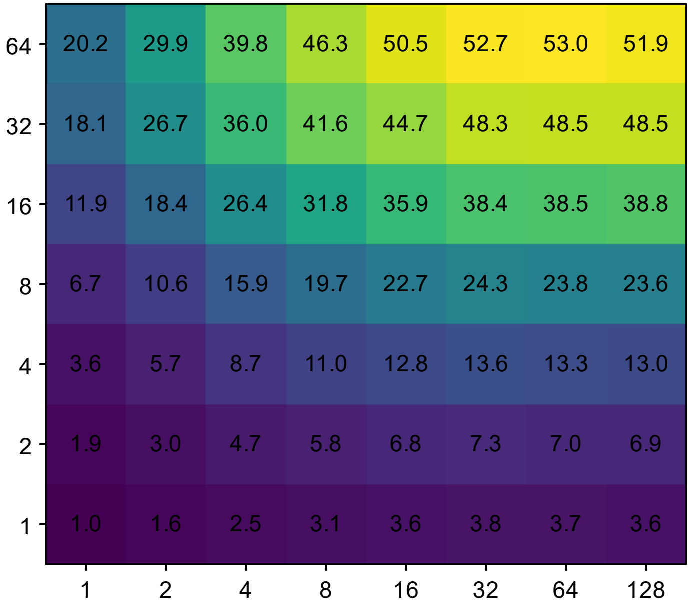
	1	2	4	8	16	32	64	128
64	17.7	23.7	26.4	27.3	29.0	30.1	30.7	31.2
32	14.3	23.9	24.6	28.3	26.5	28.8	29.8	29.8
16	10.8	17.8	24.4	26.2	26.0	25.7	24.8	23.5
8	6.2	10.5	15.7	18.8	19.5	18.7	16.9	14.9
4	3.4	5.9	9.4	11.6	12.3	11.9	10.0	8.5
2	1.9	3.1	5.4	7.1	7.6	7.1	5.6	4.6
1	0.9	1.6	2.8	4.0	4.9	4.4	3.2	2.5

circuit_4
(nnz=308k)

scircuit
(nnz=960k)



	1	2	4	8	16	32	64	128
64	22.5	36.4	56.8	77.5	86.8	97.9	99.4	96.6
32	18.0	30.7	45.6	60.0	81.6	91.2	95.2	95.1
16	12.3	22.0	36.5	50.0	61.6	67.9	71.0	72.0
8	6.8	12.4	21.8	30.8	37.2	40.9	42.6	43.1
4	3.6	6.7	12.0	17.0	21.0	22.3	23.4	23.8
2	1.9	3.6	6.5	9.2	11.2	12.0	12.3	12.4
1	1.0	2.0	3.7	5.3	6.0	6.3	6.5	6.4



	1	2	4	8	16	32	64	128
64	20.2	29.9	39.8	46.3	50.5	52.7	53.0	51.9
32	18.1	26.7	36.0	41.6	44.7	48.3	48.5	48.5
16	11.9	18.4	26.4	31.8	35.9	38.4	38.5	38.8
8	6.7	10.6	15.9	19.7	22.7	24.3	23.8	23.6
4	3.6	5.7	8.7	11.0	12.8	13.6	13.3	13.0
2	1.9	3.0	4.7	5.8	6.8	7.3	7.0	6.9
1	1.0	1.6	2.5	3.1	3.6	3.8	3.7	3.6

ASIC_680ks
(nnz=2330k)

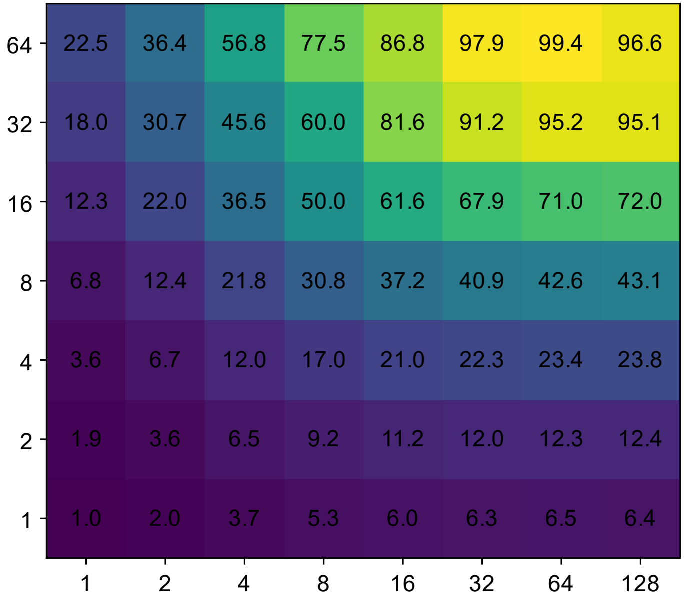
Scalable CPUs

- Fully bundled source code with no dependency
- GCC 11.2.0 (-O3 -march=native -mtune=native) & Ubuntu 22.0.4 LTS

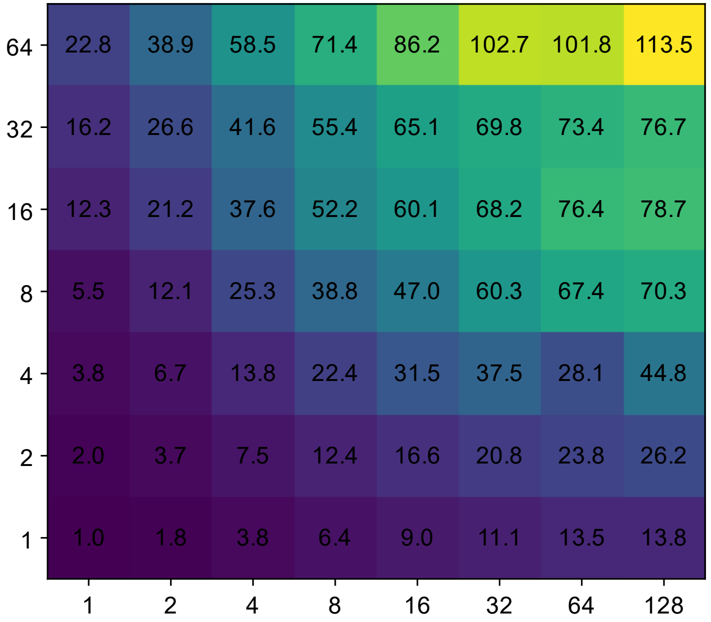
Instance	Provider	vCPU	cores	Type
c7g	AWS	64	64	Graviton3 (~ Neoverse V1)
c6g	AWS	64	64	Graviton2 (Neoverse N1)
c6i	AWS	64	32	Intel 3rd gen (Ice Lake)
c6a	AWS	64	32	AMD EPYC 3rd gen (Milan – 2x2x8)
T2A	GCP	48	48	Ampere Altra (Neoverse N1)
C2	GCP	60	30	Intel 2nd gen (Cascade Lake)
C2D	GCP	56	28	AMD EPYC 3rd gen (Milan – 1x7x4)

Scalability vs CPUs (solve only for scircuit)

c6i
(AWS) Intel 3rd gen



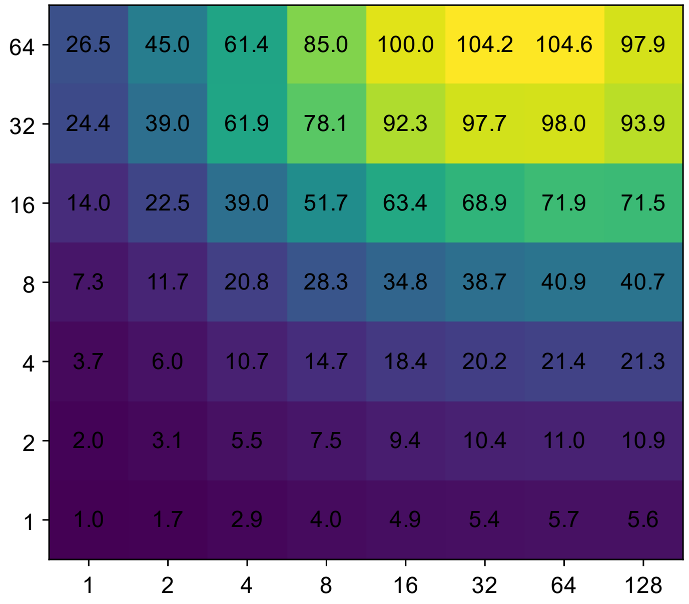
	1	2	4	8	16	32	64	128
64	22.5	36.4	56.8	77.5	86.8	97.9	99.4	96.6
32	18.0	30.7	45.6	60.0	81.6	91.2	95.2	95.1
16	12.3	22.0	36.5	50.0	61.6	67.9	71.0	72.0
8	6.8	12.4	21.8	30.8	37.2	40.9	42.6	43.1
4	3.6	6.7	12.0	17.0	21.0	22.3	23.4	23.8
2	1.9	3.6	6.5	9.2	11.2	12.0	12.3	12.4
1	1.0	2.0	3.7	5.3	6.0	6.3	6.5	6.4



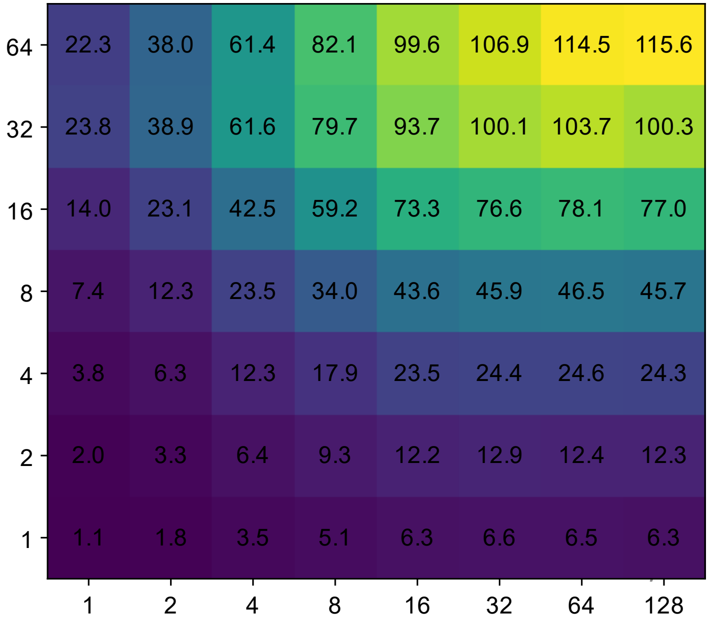
	1	2	4	8	16	32	64	128
64	22.8	38.9	58.5	71.4	86.2	102.7	101.8	113.5
32	16.2	26.6	41.6	55.4	65.1	69.8	73.4	76.7
16	12.3	21.2	37.6	52.2	60.1	68.2	76.4	78.7
8	5.5	12.1	25.3	38.8	47.0	60.3	67.4	70.3
4	3.8	6.7	13.8	22.4	31.5	37.5	28.1	44.8
2	2.0	3.7	7.5	12.4	16.6	20.8	23.8	26.2
1	1.0	1.8	3.8	6.4	9.0	11.1	13.5	13.8

c6a
(AWS) AMD EPYC Milan

c6g
(AWS) Graviton 2



	1	2	4	8	16	32	64	128
64	26.5	45.0	61.4	85.0	100.0	104.2	104.6	97.9
32	24.4	39.0	61.9	78.1	92.3	97.7	98.0	93.9
16	14.0	22.5	39.0	51.7	63.4	68.9	71.9	71.5
8	7.3	11.7	20.8	28.3	34.8	38.7	40.9	40.7
4	3.7	6.0	10.7	14.7	18.4	20.2	21.4	21.3
2	2.0	3.1	5.5	7.5	9.4	10.4	11.0	10.9
1	1.0	1.7	2.9	4.0	4.9	5.4	5.7	5.6

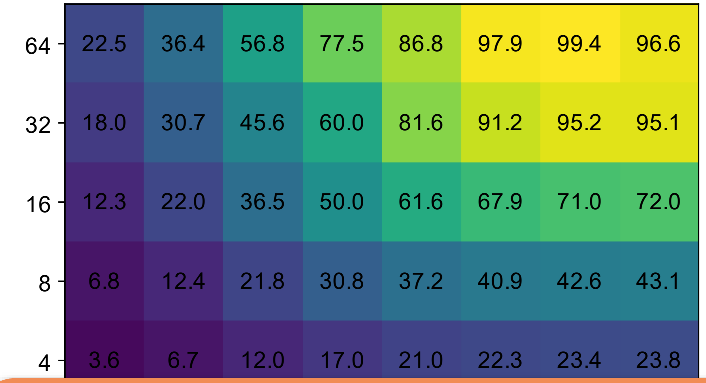


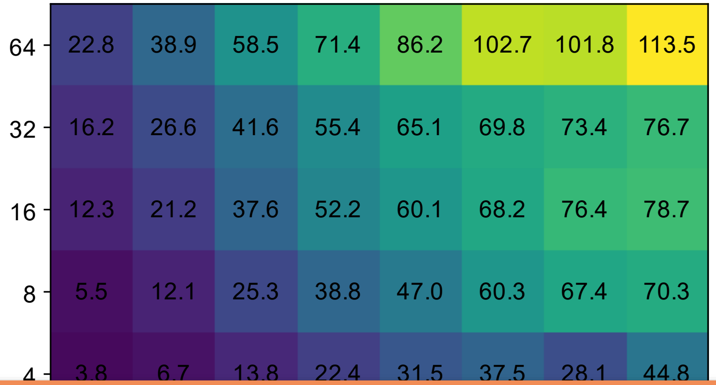
	1	2	4	8	16	32	64	128
64	22.3	38.0	61.4	82.1	99.6	106.9	114.5	115.6
32	23.8	38.9	61.6	79.7	93.7	100.1	103.7	100.3
16	14.0	23.1	42.5	59.2	73.3	76.6	78.1	77.0
8	7.4	12.3	23.5	34.0	43.6	45.9	46.5	45.7
4	3.8	6.3	12.3	17.9	23.5	24.4	24.6	24.3
2	2.0	3.3	6.4	9.3	12.2	12.9	12.4	12.3
1	1.1	1.8	3.5	5.1	6.3	6.6	6.5	6.3

c7g
(AWS) Graviton 3

Scalability vs CPUs (solve only for scircuit)

c6i
(AWS) Intel 3rd gen

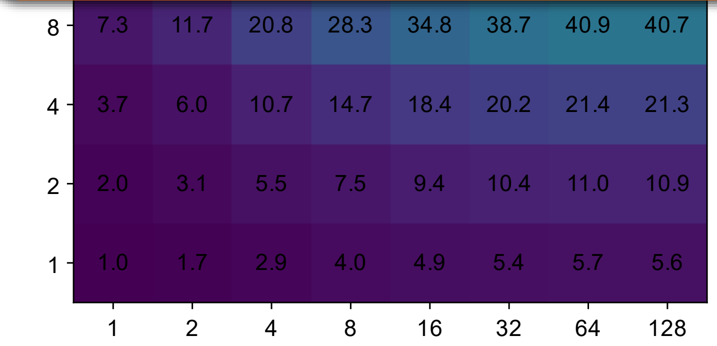




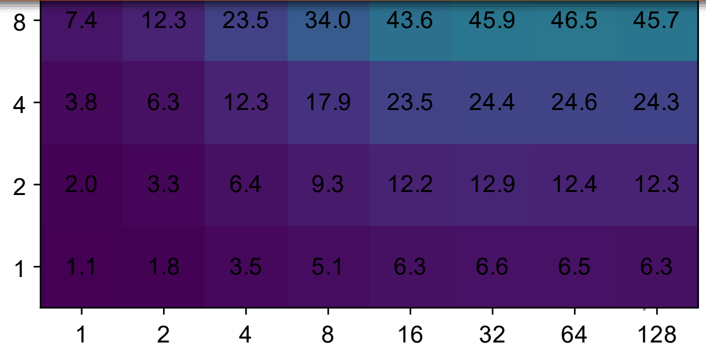
c6a
(AWS) AMD EPYC Milan

What does it means in absolute solve time?

c6g
(AWS) Graviton 2



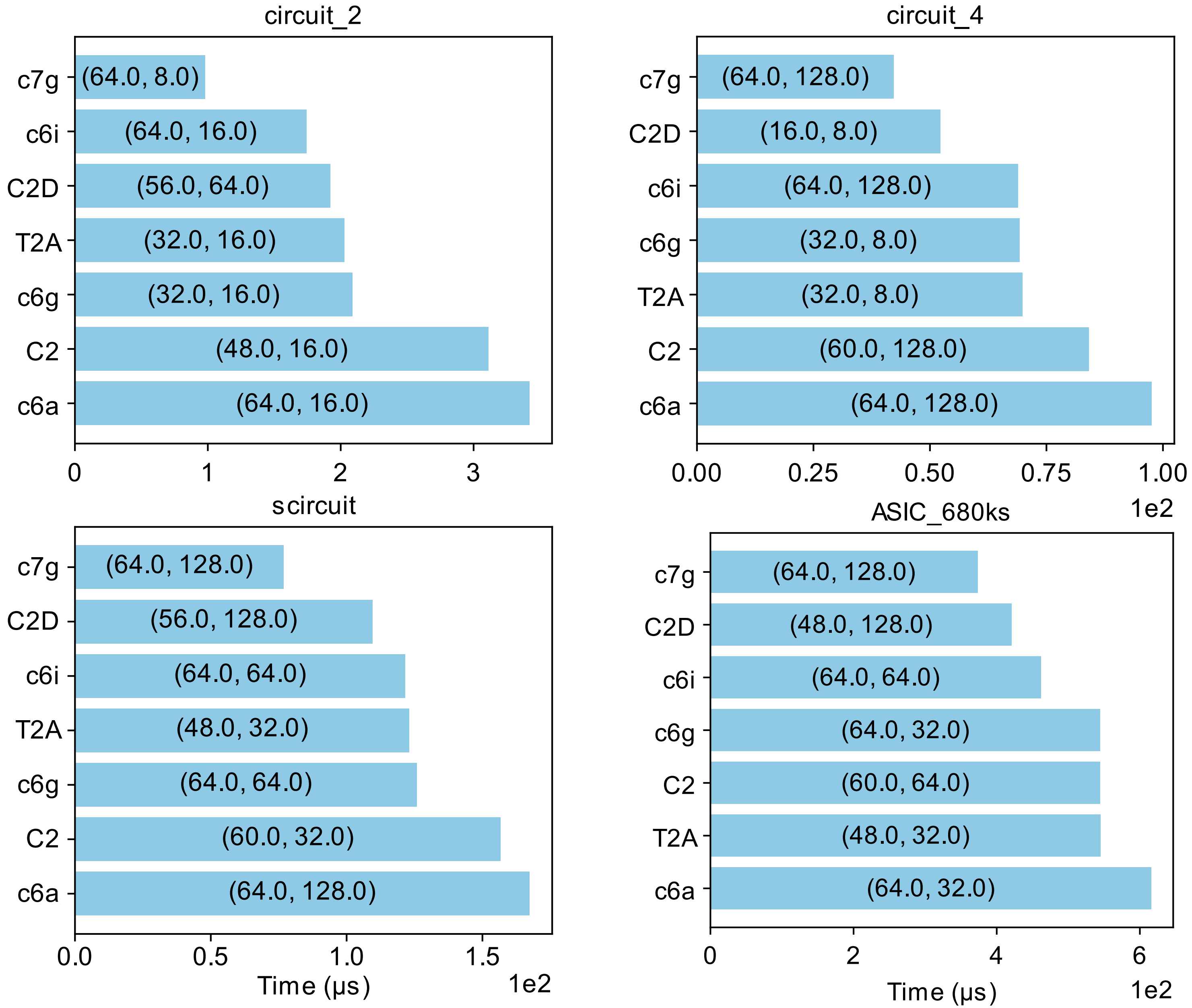
g
WS) Graviton 3



Benchmarks absolute time (Solve only)

(threads , nrhs)

Instance	Provider	vCPU	Type
c7g	AWS	64	Graviton3
c6g	AWS	64	Graviton2
c6i	AWS	64	Intel 3rd gen
c6a	AWS	64	AMD Milan
T2A	GCP	48	Ampere Altra
C2	GCP	60	Intel 2nd gen
C2D	GCP	56	AMD Milan



FoM among CPUs

- Real s$i\ \ total\ factor\ time\ nthreads + total\ solve\ time\ nrhs$
- $otal\ solve\ time\ nrhs\ \ solve\ time\ nrhs\ nrhs\ tal\ factor\ time\ nthreads\ tor\ time\ nthreads\ nthreads$ lation use case:

nthreads (device variation) x _nrhs_ (source variation)

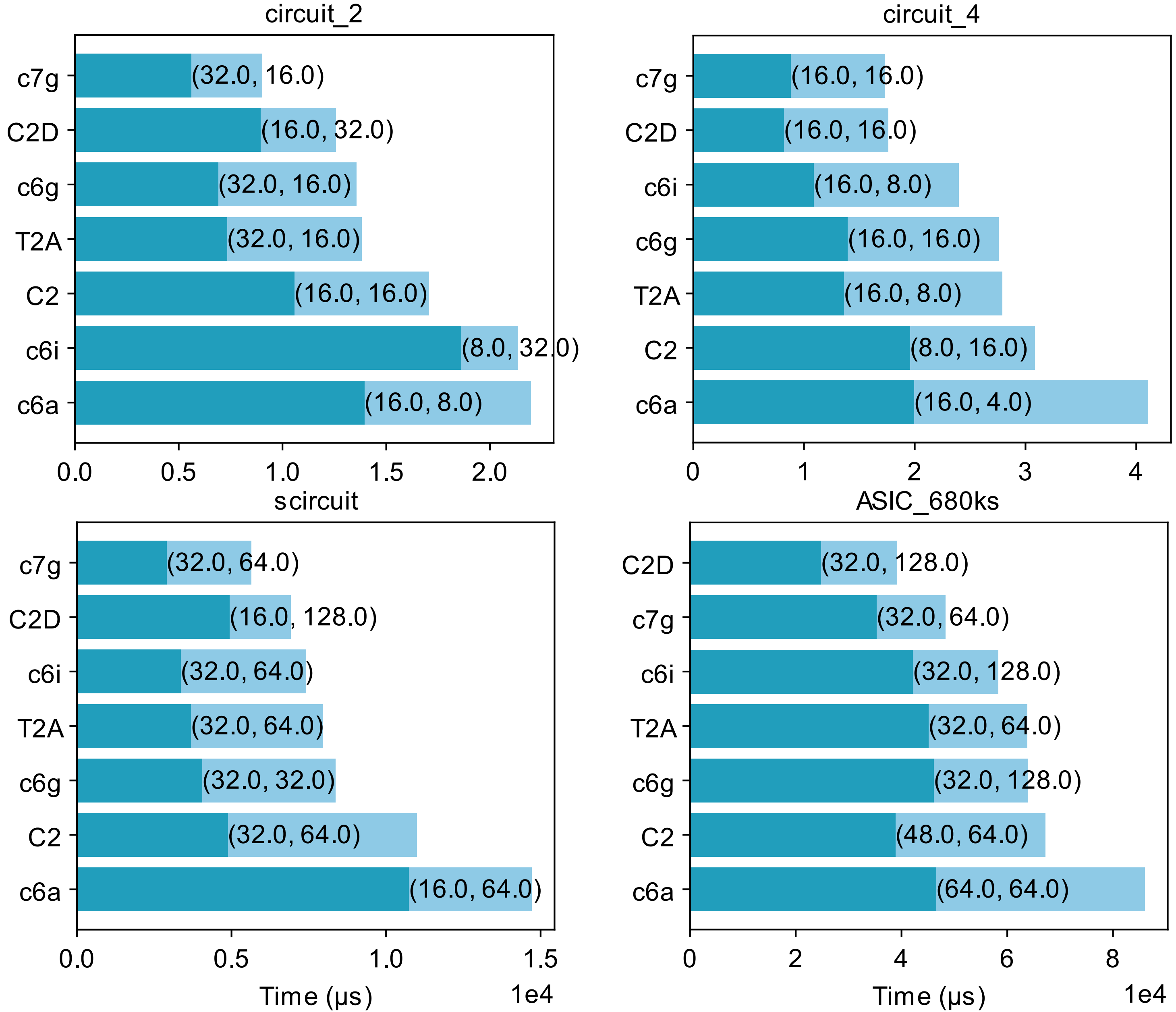
- nthreads→ Threads nrhs → SIMD
- We define a Figure-of-Merit as:

$$FOM = \frac{total\ factor\ time}{nthreads} + \frac{total\ solve\ time}{nrhs}$$

Benchmarks - FoM (factorisation + solve)

(threads , nrhs)

Instance	Provider	vCPU	Type
c7g	AWS	64	Graviton3
c6g	AWS	64	Graviton2
c6i	AWS	64	Intel 3rd gen
c6a	AWS	64	AMD Milan
T2A	GCP	48	Ampere Altra
C2	GCP	60	Intel 2nd gen
C2D	GCP	56	AMD Milan

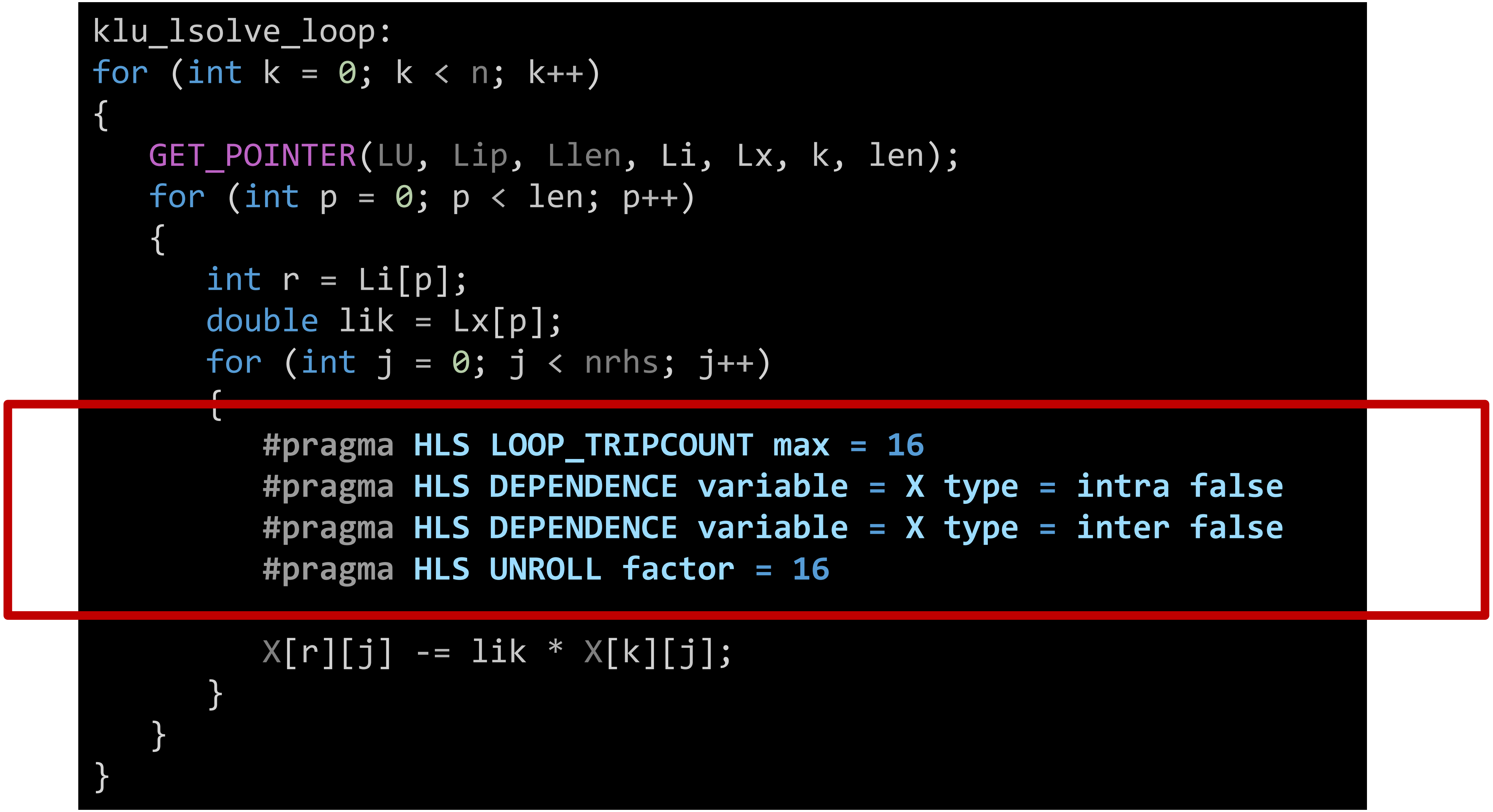


FPGA HLS Implementation

- Same code base as CPU only with added HLS pargmas
- Xilinx Alveo U280 on AMD/Xilinx’s Heterogeneous Accelerated Compute Clusters (HACC) and Vivado Vitis 2021

- FPGA specific optimisations:
  - Pipelining
  - Loop unrolling
  - Array partitioning

FPGA HLS Implementation



```
klu_lsolve_loop:
for (int k = 0; k < n; k++)
{
   GET_POINTER(LU, Lip, Llen, Li, Lx, k, len);
   for (int p = 0; p < len; p++)
   {
      int r = Li[p];
      double lik = Lx[p];
      for (int j = 0; j < nrhs; j++)
      {
         #pragma HLS LOOP_TRIPCOUNT max = 16
         #pragma HLS DEPENDENCE variable = X type = intra false
         #pragma HLS DEPENDENCE variable = X type = inter false
         #pragma HLS UNROLL factor = 16

         X[r][j] -= lik * X[k][j];
      }
   }
}
```

FPGA – optimisation methods

- FPGA (Alveo U280) utilisation for a single kernel

LUT	Registers	Block RAM	Ultra RAM	DSP
99k	120k	178	18	182
7.6%	4.6%	8.8%	1.9%	2.0%

- Benchmark vs optimisations for rajat11 matrix (nnz=812)

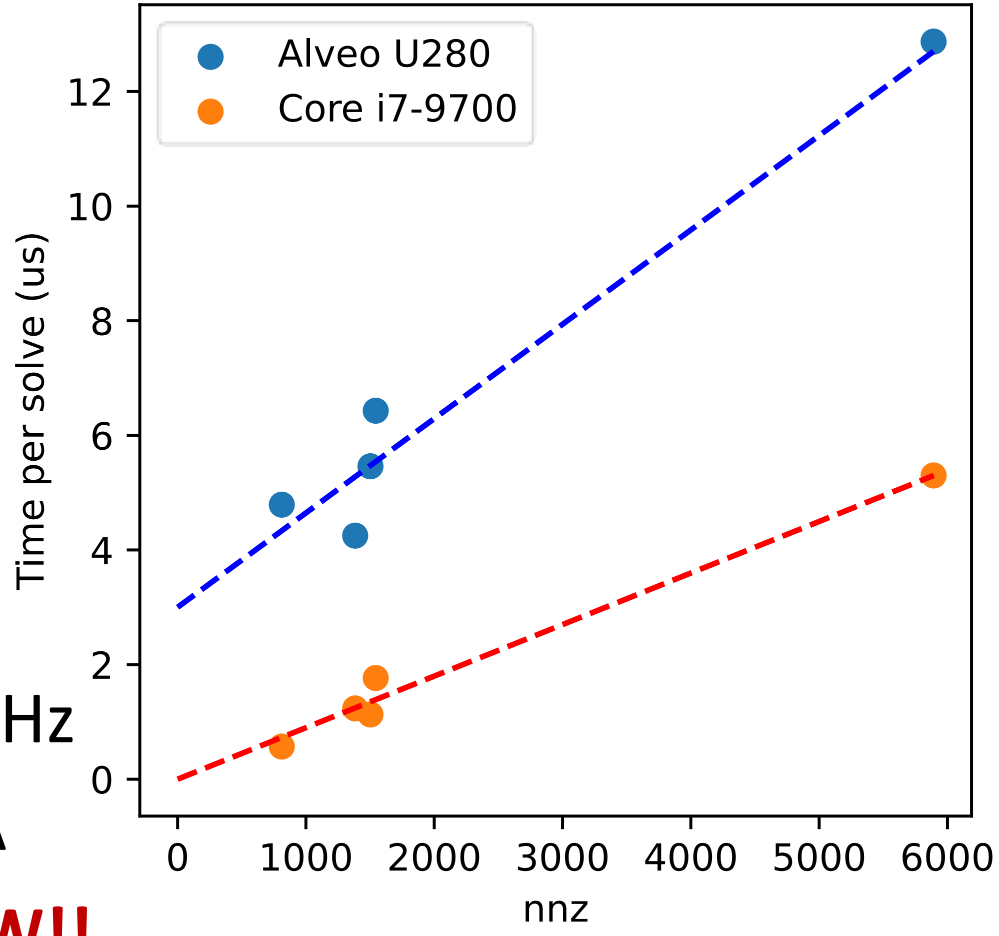
Optimisation	Factorisation (µs)	Solve 1 rhs (µs)	Solve 10 rhs (µs)	Speedup per rhs
Pipeline only	312	28	145	1.93
Pipeline & Unroll	320	31	332	**0.93**
Pipeline & Unroll & Array partition	297	29	36	**8.06**

- Array partitioning has a major impact in acceleration

FPGA - results

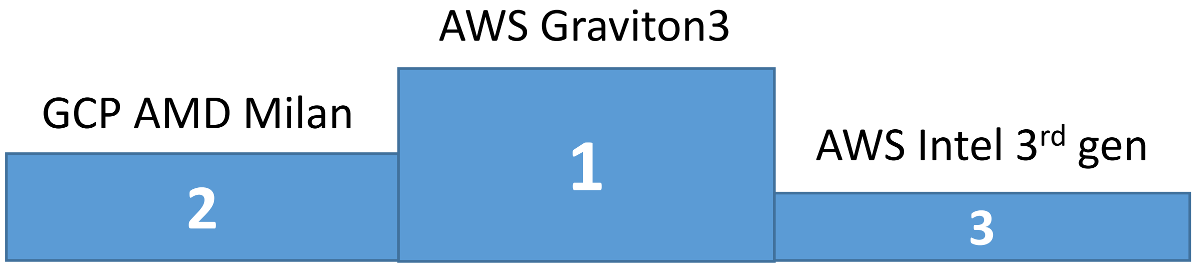
- Comparison with single threaded Corei7
- Both FPGA and CPU nrhs=10
- Similar trend with CPU but not as fast

- Only few small matrices available
- Large matrices give timing errors
- FPGA clocks is 250MHz and Core i7 at 4.5GHz
- Could try parallel kernels in the same FPGA
- **The FPGA development cycle is VERY SLOW!!**



Conclusions

- Modified KLU sparse linear solver using threading and SIMD
- Use inherit parallelism in chip design workflow
- Solve time is strongly related to the number of non-zero elements
- Each matrix has an optimum thread and nrhs
- Strong correlation with hardware and cache levels



AWS Graviton3

1

GCP AMD Milan

2

AWS Intel 3rd gen

3

- FPGA implementation still in early stages

Future work

- Further bring the codebase closer to C++17
  (and eventually SYCL)

- Benchmark more matrices for robust results

- Can this be implemented in GPU?
  (perhaps the solve part, but not factorisation)

Acknowledgments

- AMD/Xilinx
  - Ronan Keryell
  - Mario and Cathal (XUP Dublin)
- Intel Dev Cloud
- Google's Cloud Paltfrom
- AWS

Open Source Librariers:

- Tim Davis - ***SuiteSparse***
- Barak Shoshany - ***Threadpool***
- Killian Sheriff - ***Lovelyplots***