Performance Evaluation of the Ginkgo Sparse Linear Solver Framework on Arm

May. 25th, 2023 – AHUG workshop at ISC conference, Hamburg, Germany

Luka STANISIC, Robert MIJAKOVIC, Matthias GRIES

HUAWEI

Scope

Huawei's investment in HPC

- Many products: storage (main focus of ISC 2023 booth E518), cloud, traditional HPC servers, AI accelerators, software, etc.
- Our team concentrates on Arm software ecosystem enablement for HPC & AI (Munich research center, funded by HiSilicon)
- Recognized current and future importance of sparse linear algebra for performance of top HPC applications

Ginkgo sparse linear solver framework (https://ginkgo-project.github.io/)

- Offering set of iterative solvers and preconditioners under BSD 3-clause permissive open-source license
- Comes with built-in benchmarks for spmv, matrix conversion and solver / preconditioner
- Recent, fresh approach, started in 2017 – already integrated with MFEM, deal.II, xSDK
- Main developers and maintainers located in Germany at Karlsruhe Institute of Technology (KIT)
- Most programming efforts spent on kernels for Nvidia, AMD and Intel accelerators
- Very competitive performance

Focus of the current research project

- Improving the OpenMP CPU execution target for Huawei Kunpeng 920 AArch64 chipset
- Relying on GNU GCC compilation
- First step: performance characterization of Ginkgo sparse linear solver on Arm & x86

Experimental Setup

Evaluated several thousand unique experimental setups

- Ginkgo options: 3 benchmarks (spmv, conversions, solver), 9 preconditioners (jacobi, paric, parict, parilu, parilut, paric-isai, parict-isai, parilu-isai, parilut-isai, or none), 5 sparse matrix formats (csr, coo, ell, hybrid, sellp), etc.
- Huawei Kunpeng 920 system options: GCC compilers (versions 10, 11, 12), OpenMP threads scaling (24, 48, 96), etc.
- Additional investigations (in backup slides): active Working Set Size (WSS), memory frequency scaling (1600/2133/2933 MHz), hardware prefetching (on/off), benchmark precision, I/O, C++ overhead, etc.

Evaluation with 10 real-world examples from SuiteSparse collection

- Ranked and selected matrices by studying ~200 sources (publications, whitepapers, products, frameworks) from industry and academia

Machines

- 96-cores 2x Huawei Kunpeng 920-4826 (launched in January 2019)
  - 128-cores version of Kunpeng 920 accessible via OEHI
- 64-cores Amazon Graviton 3 on AWS EC2 Ireland (remote access)
- 48-cores 2x Intel 3rd gen Xeon Scalable Processors Gold 6342 (IceLake)
- 48-cores 2x AMD 3rd gen EPYC 7413 (Milan)

Selected use cases from SuiteSparse matrix collection

Name	Group	Rows	Cols	Non-zeros	Sparsity	Kind	Date	Ref. count
thermal2	Schmid	1,228,045	1,228,045	8,580,313	0.0006%	Thermal Problem	2006	9
Serena	Janna	1,391,349	1,391,349	64,131,971	0.0033%	Structural Problem	2011	9
bone010	Oberwolfach	986,703	986,703	47,851,783	0.0049%	Model Reduction Problem	2006	7
atmosmodd	Bourchtein	1,270,432	1,270,432	8,814,880	0.0005%	Computational Fluid Dynamics	2009	5
offshore	Um	259,789	259,789	4,242,673	0.0063%	Electromagnetics Problem	2010	5
G3_circuit	AMD	1,585,478	1,585,478	7,660,826	0.0003%	Circuit Simulation	2006	5
consph	Williams	83,334	83,334	6,010,480	0.0865%	2D/3D Problem	2008	5
pdb1HYS	Williams	36,417	36,417	4,344,765	0.3276%	Undirected Graph	2008	5
Hardesty3	Hardesty	8,217,820	7,591,564	40,451,632	0.0001%	Computer Graphics/Vision	2015	0
analytics	Precima	303,813	303,813	2,006,126	0.0022%	Data Analytics Problem	2018	0

http://sparse.tamu.edu/

Overview

- **Ginkgo performance analysis**
  - Roofline
  - Instruction mix and top-down
  - Hot kernels
  - OpenMP imbalance
  - Scalability
  - Overall performance

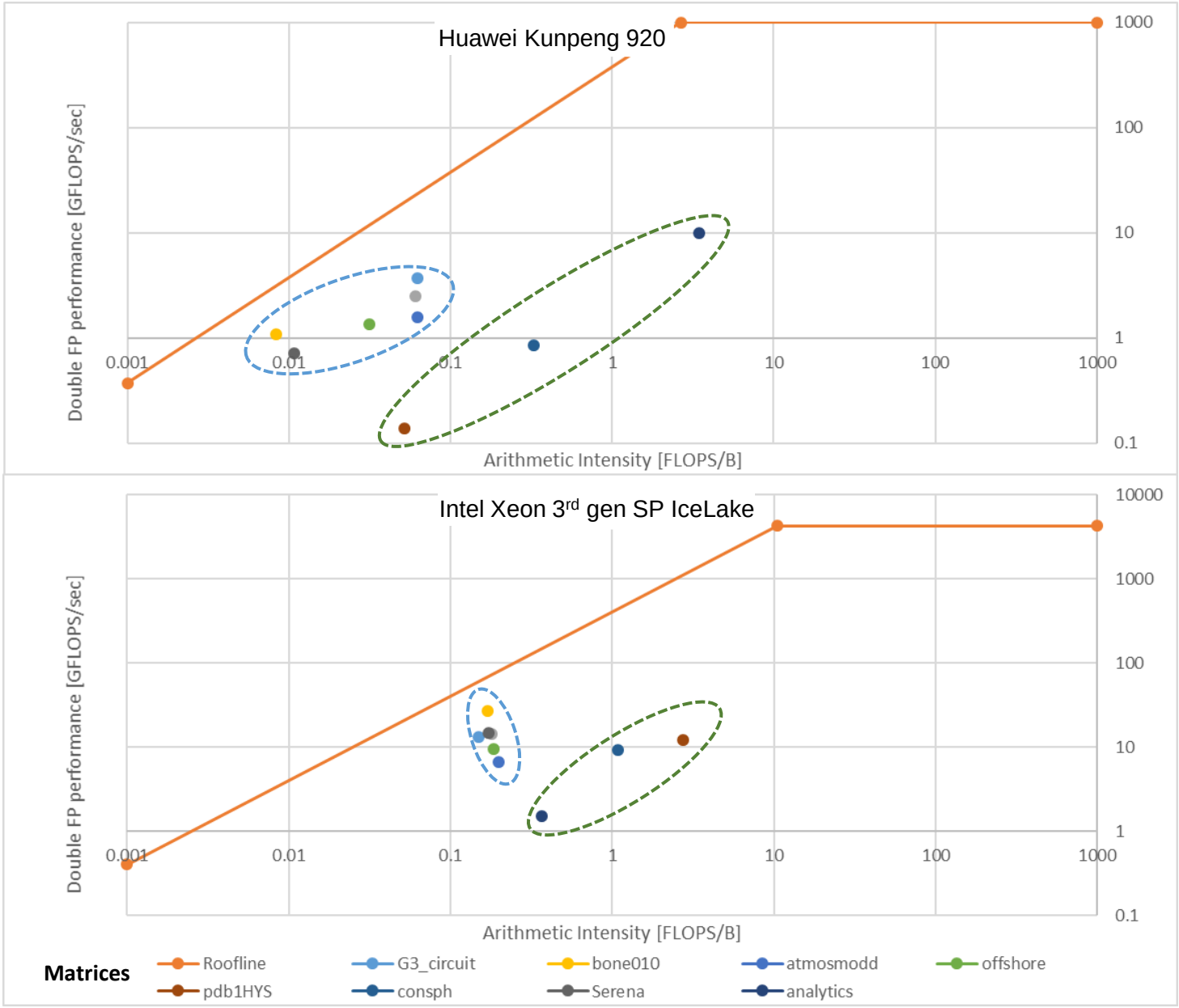
- **Concluding Remarks**
  - Pitfalls and next steps

Roofline Analysis

- Best preconditioner + solver combination per matrix applied, at double-precision

- In memory-bound region of the roof, with large dynamic range for arithmetic intensity
- Significant distance to the roof due to control flow, partial cacheline use, non-consecutive memory accesses, lack of SIMD use, etc.

- FLOPS efficiency thus below 1% (similar to, e.g., HPCG)
- Behavior depending on number of non-zeros (NNZ) and sparsity patterns of matrices



Instruction Mix and Top-down Analysis

Statistics for complete preconditioner + solver runs on Kunpeng 920 summarized for all matrices

- Low fraction of floating point and SIMD instructions
- High amount of integer operations
- Mainly limited by memory backend (LLC and main memory)
- Limitation by LLC or main memory strongly depends on matrix

Additional experiments across Arm & x86 machines confirmed minimal performance impact of advanced SIMD support coming from GCC 12 auto-vectorization

To be further analyzed and tuned

	Sub-category	AVERAGE	MIN	MAX
Instruction Mix	Memory (%)	21.84	16.72	24.99
	Integer (%)	48.38	45.25	53.27
	Floating Point (%)	2.65	0.42	4.33
	Advanced SIMD (%)	1.27	0.04	6.06
	Not Retired (%)	2.22	0.36	4.75
Top-down	Retiring (%)	30.83	16.46	49.60
	Backend Bound (%)	62.98	39.48	79.62
	-> Memory Bound (%)	40.46	15.66	62.05
	--> L1 Bound (%)	6.55	3.13	10.10
	--> L2 Bound (%)	0.72	0.16	1.56
	--> L3 or DRAM Bound (%)	33.13	9.68	54.94
	--> Store Bound (%)	0.06	0.02	0.15
	-> Core Bound (%)	22.51	17.57	27.41
	Frontend Bound (%)	5.41	2.50	9.32
	Bad Speculation (%)	0.79	0.09	1.99
Memory subsystem	Average DRAM Bandwidth (GB/s)	34.08	2.76	68.54
	-> Read (GB/s)	28.82	2.32	65.03
	-> Write (GB/s)	5.27	0.44	13.05
	L3 By-Pass ratio (%)	8.39	0.83	22.19
	L3 miss ratio (%)	62.13	24.29	82.98
	L3 Utilization Efficiency (%)	79.51	36.37	96.85
	Within Socket Bandwidth (GB/s)	1.07	0.25	3.35
	Inter Socket Bandwidth (GB/s)	1.79	0.33	5.73

Hot Compute Kernels on Kunpeng 920

Example: spmv kernel for csr format

- Iterating over rows
- Non-consecutive column accesses
- Multiply-add operation

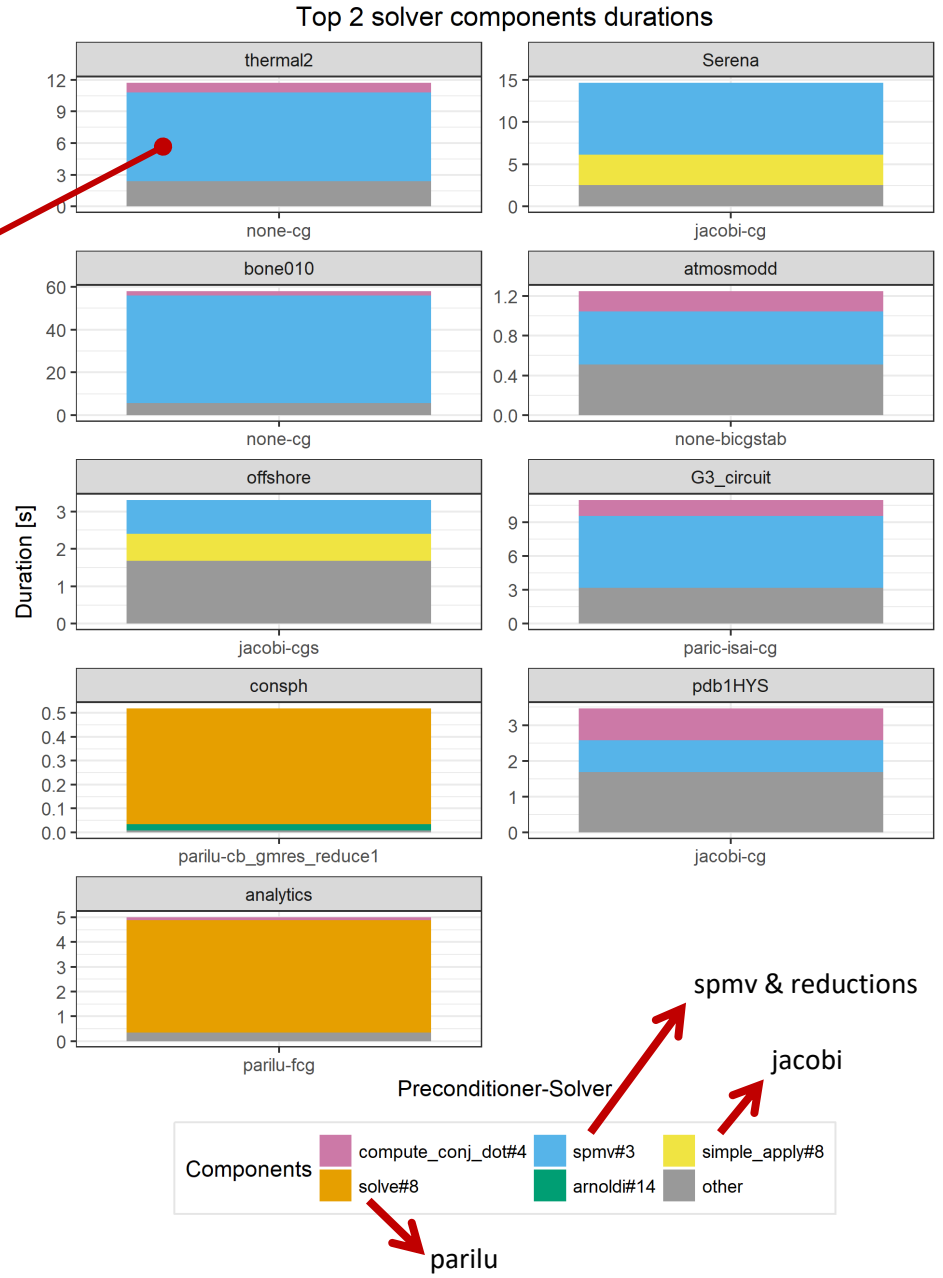
```
#pragma omp parallel for
    for (size_type row = 0; row < a->get_size()[0]; ++row) {
        for (size_type j = 0; j < c->get_size()[1]; ++j) {
            c->at(row, j) = zero<ValueType>();
        }
        for (size_type k = row_ptrs[row];
             k < static_cast<size_type>(row_ptrs[row + 1]); ++k) {
            auto val = vals[k];
            auto col = col_idxs[k];
            for (size_type j = 0; j < c->get_size()[1]; ++j) {
                c->at(row, j) += val * b->at(col, j);
            }
        }
    }
```

Limitations

- Cannot take advantage of SIMD for multiply-add
- Memory bound because of non-consecutive accesses

Further hotspots identified

- Column reduction operations
- Jacobi preconditioner
- ParILU (Parallel Incomplete LU) preconditioner



OpenMP Imbalance Examples: Kunpeng 920, Preconditioner + Solver

- Considerable parallel time, but with noticeable imbalance between 96 OpenMP threads

Matrix	Total execution (s)	Serial time (s)	Serial time (%)	Parallel time (s)	Parallel time (%)	Imbalance (s)	Imbalance (%)
bone010	27.98	6.32	22.61%	21.65	77.39%	9.08	41.96%
Top parallel region	Elapsed time (s)	Perc. of total (%)	Perc. of parallel (%)	Average (ms)	Count	Imbalance (s)	Imbalance (%)
csr_kernels.cpp:84	8.35	29.86%	38.58%	0.82	10200	1.90	22.78%
kernel_launch_reduction.hpp:335	2.42	8.66%	11.19%	0.12	20403	1.34	55.13%
kernel_launch.hpp:83	2.27	8.11%	10.48%	0.22	10200	0.66	29.18%
kernel_launch_reduction.hpp:353	1.64	5.85%	7.56%	0.08	20403	1.33	81.45%

- High serial time, but low imbalance in parallel regions

Matrix	Total execution (s)	Serial time (s)	Serial time (%)	Parallel time (s)	Parallel time (%)	Imbalance (s)	Imbalance (%)
Serena	42.54	26.10	61.35%	16.44	38.65%	1.64	9.99%
Top parallel region	Elapsed time (s)	Perc. of total (%)	Perc. of parallel (%)	Average (ms)	Count	Imbalance (s)	Imbalance (%)
sellp_kernels.cpp:67	9.63	22.65%	58.60%	5.65	1706	0.8117	8.43%
jacobi_kernels.cpp:564	4.09	9.61%	24.87%	2.39	1709	0.3982	9.74%
kernel_launch_reduction.hpp:335	0.62	1.47%	3.80%	0.18	3415	0.0856	13.71%

- Preconditioner dominating, high imbalance (kernels de facto sequential)

Matrix	Total execution (s)	Serial time (s)	Serial time (%)	Parallel time (s)	Parallel time (%)	Imbalance (s)	Imbalance (%)
consph	23.32	3.26	13.97%	20.06	86.03%	16.80	83.76%
Top parallel region	Elapsed time (s)	Perc. of total (%)	Perc. of parallel (%)	Average (ms)	Count	Imbalance (s)	Imbalance (%)
upper_trs_kernels.cpp:101	9.58	41.07%	47.74%	28.93	331	9.45	98.73%
lower_trs_kernels.cpp:101	6.36	27.27%	31.70%	19.21	331	6.28	98.85%

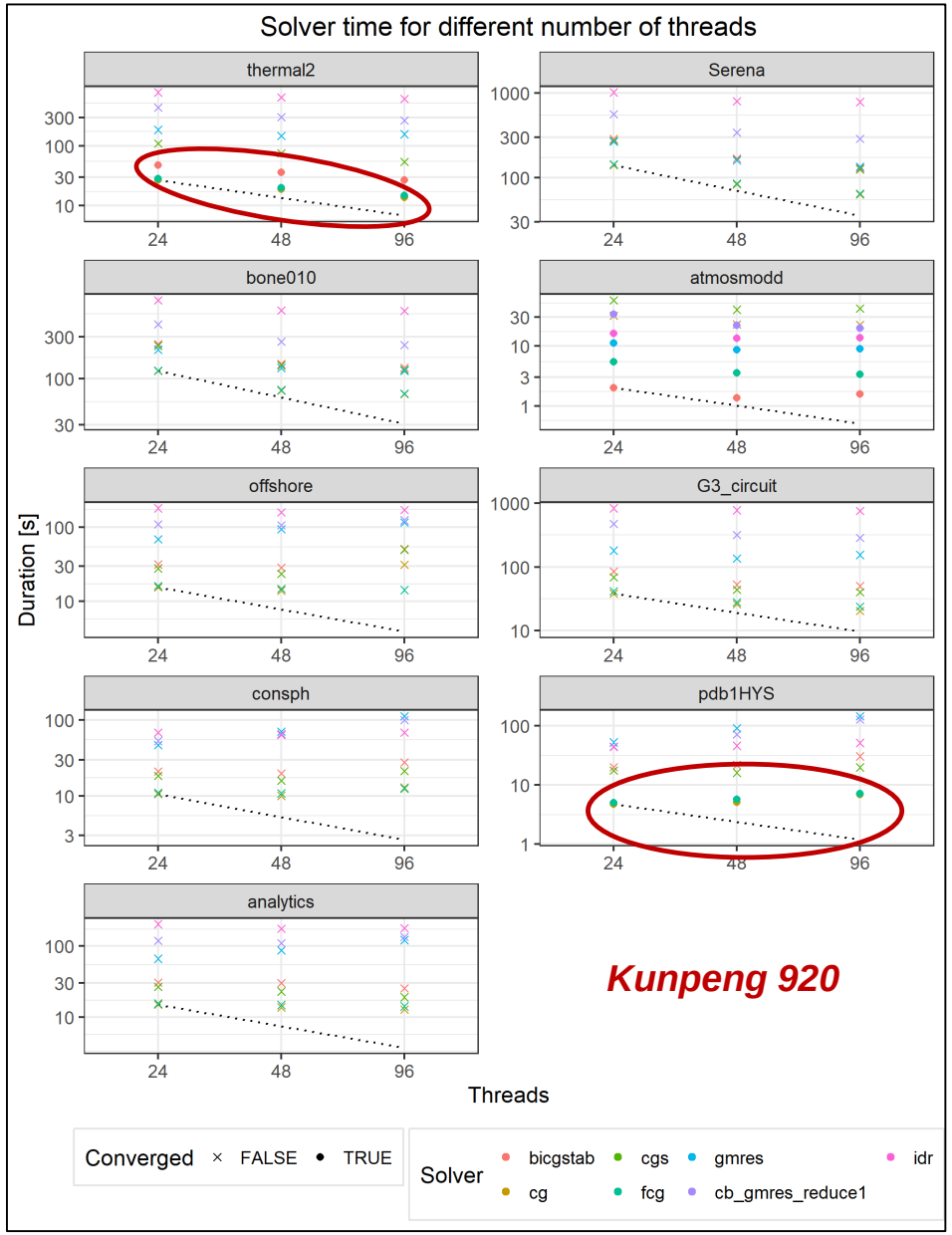
To be further analyzed and tuned

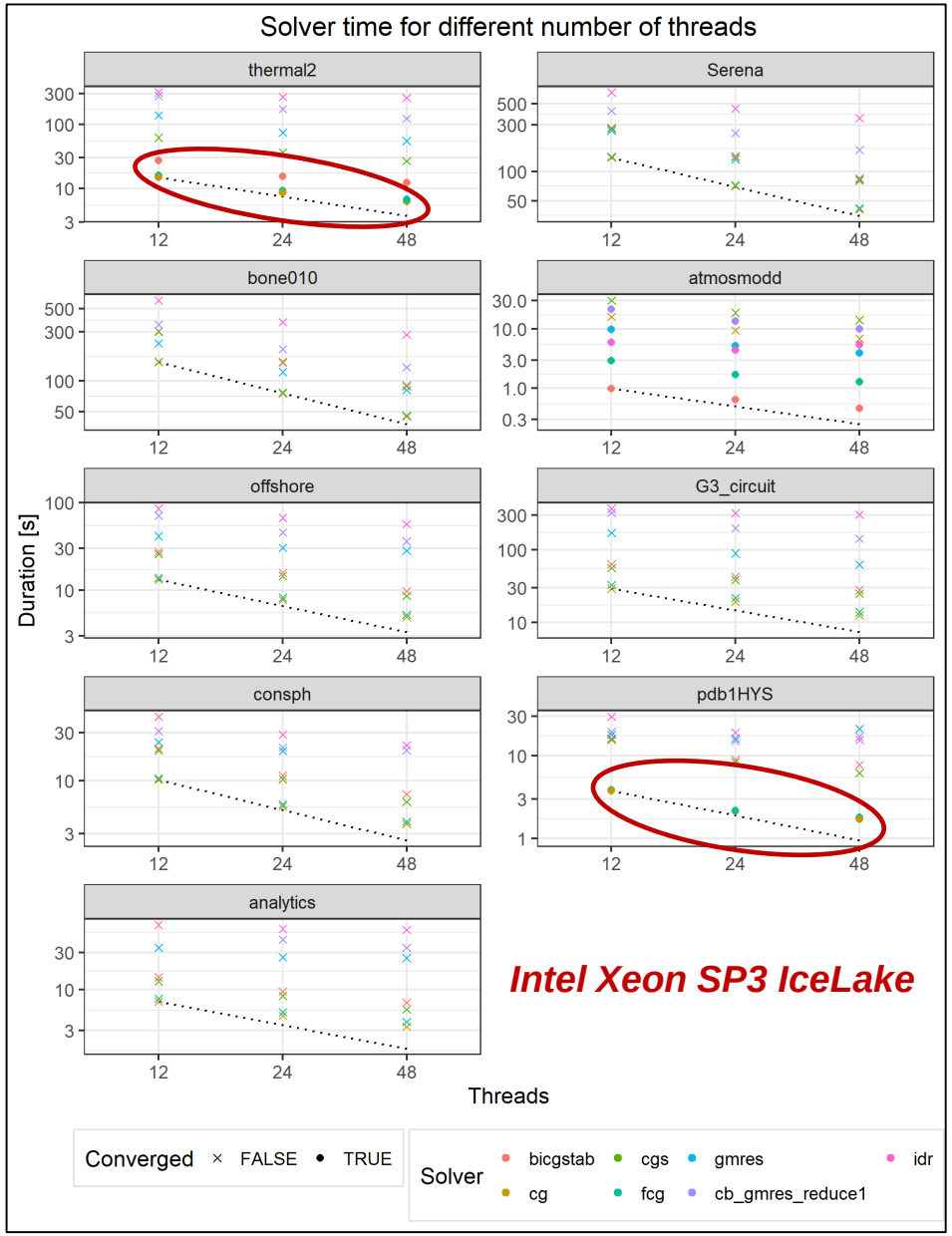
Scalability

- Solver benchmark
- Log vertical axis
- Results again very dependent on matrix

- Kunpeng 920 has 96 cores and often cannot take advantage of complete node

- Mediocre scaling on other machines (48 and 64 cores)
- No benefits from hyperthreading

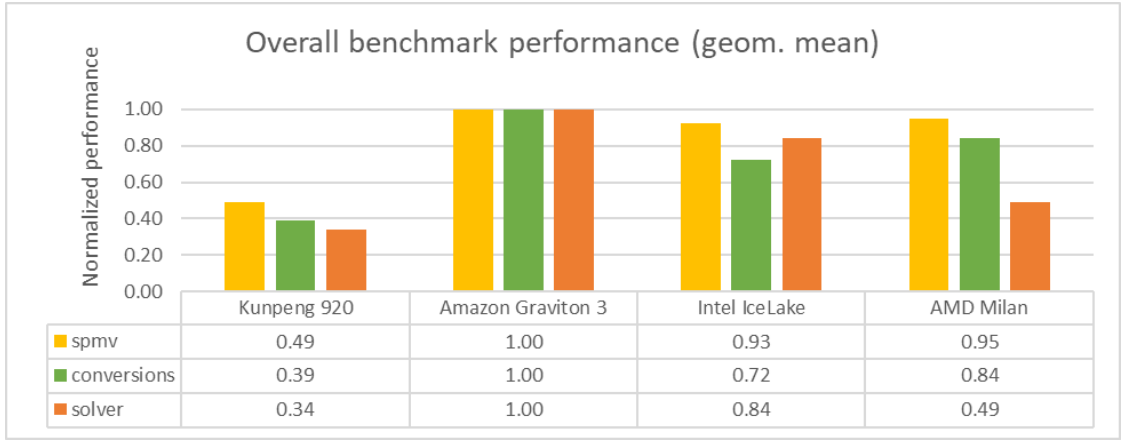




Overall Performance

Geometric mean across matrices, default config ranking:

1. 64-cores Amazon Graviton 3 on AWS EC2 - *2022*
2. 48-cores 2x Intel Xeon SP3 Gold 6342 (IceLake) - *2021*
3. 48-cores 2x AMD 3rd gen EPYC 7413 (Milan) - *2021*
4. 96-cores 2x Huawei Kunpeng 920-4826 - *2019*



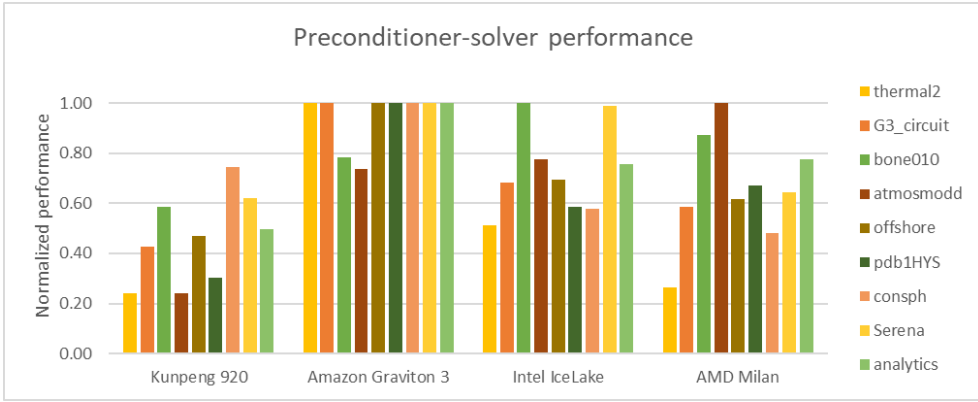
Preconditioner + solver selection

- Simple Jacobi preconditioner often the best, followed by not using a preconditioner at all
- Selected cases where one of the parallel incomplete Cholesky/LU variants allows much faster convergence
- Conjugate Gradient (CG) solver algorithm often the best

Matrix	Kunpeng 920		Amazon Graviton 3		Intel IceLake		AMD Milan	
	Precond.	Solver	Precond.	Solver	Precond.	Solver	Precond.	Solver
thermal2	none	cg	none	cg	none	cg	jacobi	cg
G3_circuit	paric-isai	cg	paric-isai	cg	paric-isai	cg	paric-isai	cg
bone010	none	cg	none	cg	none	cg	none	cg
atmosmodd	none	bicgstab	none	bicgstab	none	bicgstab	none	bicgstab
offshore	jacobi	cgs	parilu-isai	cgs	parilu-isai	cgs	jacobi	cgs
pdb1HYS	jacobi	cg	jacobi	cg	jacobi	cg	jacobi	cg
consph	parilu	cb_gmres	jacobi	cg	jacobi	cgs	jacobi	cg
Serena	jacobi	cg	jacobi	cg	jacobi	cg	jacobi	cg
analytics	parilu	fcg	parilu	gmres	parilu	cg	parilu	cb_gmres_re

Preconditioner + solver performance per matrix

- Graviton 3 mostly performs the best (7 out of 9 matrices)
- Performance on AMD Milan greatly depends on matrix



Overview

- **Ginkgo performance analysis**
  - Roofline
  - Instruction mix and top-down
  - Hot kernels
  - OpenMP imbalance
  - Scalability
  - Overall performance

- **Concluding Remarks**
  - Pitfalls and next steps

Concluding Remarks

Pitfalls

- Variability of repeated runs with the same settings can be significant
  - Mixture of OpenMP overhead, NUMA effects, startup calibrations by Ginkgo for selecting matrix format, nondeterministic preconditioner algorithms, responsiveness of memory subsystem under load

- Extra investigation of OpenMP parallelism
  - On Kunpeng 920, when a subset of cores is used: the best choice between “spread” and “close” mappings is matrix dependent, and can have a significant impact on the overall performance
  - Using dynamic and guided OpenMP scheduling strategies decreases performance, especially for smaller chunks

Next steps

- Performance analysis of direct sparse linear solvers (e.g., MUMPS framework)
- Ginkgo contributions related to micro-architecture level optimization (e.g., SVE) for selected algorithms

Thank you.

Bring digital to every person, home and organization for a fully connected, intelligent world.

References and Setups

References

- H. Anzt, T. Cojean, G. Flegar et al.: *Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing*, ACM Transactions on Mathematical Software, volume 48(1), 2022
  https://ginkgo-project.github.io/
- T.A. Davis, Yifan Hu: *The university of Florida sparse matrix collection*, ACM Transactions on Mathematical Software, Volume 38(1), 2011
  SuiteSparse matrix collection: http://sparse.tamu.edu/
- Jing Xia, Chuanning Cheng, Xiping Zhou et al.: *Kunpeng 920: The First 7-nm Chiplet-Based 64-Core ARM SoC for Cloud Services*, IEEE Micro, volume 41(5), 2021
- Brendan Gregg, *How To Measure the Working Set Size on Linux*, https://www.brendangregg.com/blog/2018-01-17/measure-working-set-size.html (accessed May 2023)

Machine setups

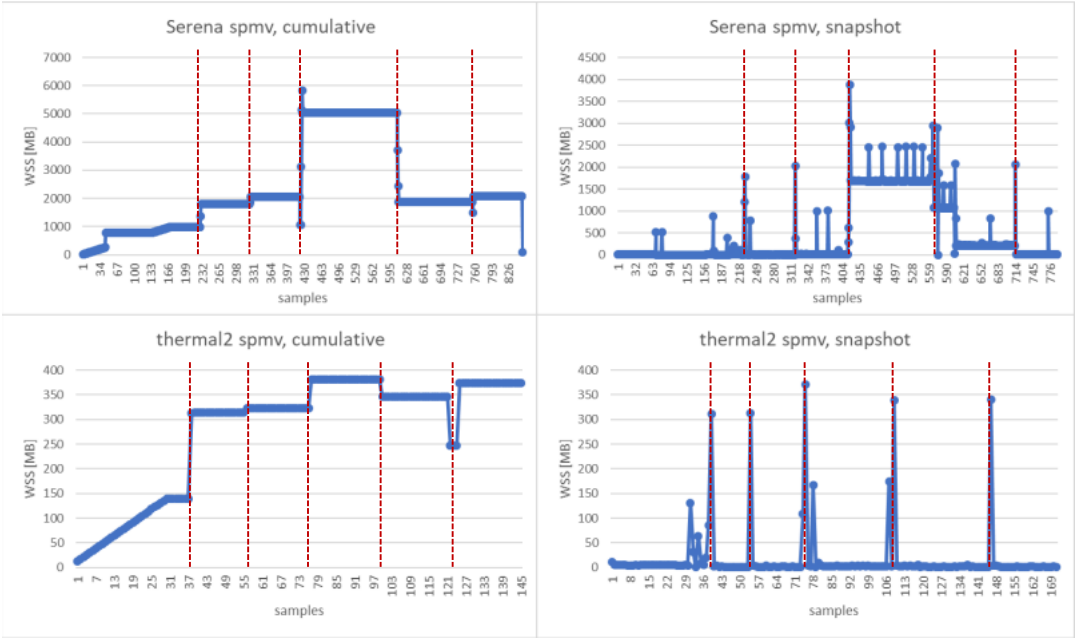
- 2x Kunpeng 920-4826 (48 cores) at 2.6 GHz, 48 MB LLC, 128 bit NEON vectorization support (one FMA unit enabled for double precision, two FMA units enabled for single precision), 16 x 32 GB DDR4-2933 memory DIMMs (single DIMM per channel), Ubuntu 18.04.6 LTS (Linux 5.4.0-136-generic aarch64)
- 2x AMD EPYC 7413 (24 cores) at 2.65 GHz base frequency, 128 MB LLC, AVX2 vectorization support (two FMA units), 16 x 16 GB DDR4-3200 memory DIMMs (single DIMM per channel), Ubuntu 18.04.6 LTS (Linux 5.4.0-90-generic x86_64)
- 2x Intel Xeon SP3 Gold 6342 (24 cores) at 2.8 GHz base frequency, 36 MB LLC, AVX512 vectorization support (two FMA units), 16 x 16 GB DDR4-3200 memory DIMMs (single DIMM per channel), Ubuntu 18.04.6 LTS (Linux 5.15.31-051531-generic x86_64)
- AWS EC2 c7g.16xlarge instance: 1x Amazon Graviton 3 (64 cores) at likely 2.6 GHz, likely 64 MB LLC, SVE vectorization support (two FMA units enabled for SVE256, 4 FMA units enabled for NEON), likely 8 channels at DDR5-4800 (128 GB total), Ubuntu 22.04.1 LTS (Linux 5.15.0-1027-aws aarch64)

Active Working Set Size (WSS)

- WSS depends on matrix characteristics
- 3 matrices need several GBs (served from main memory)
- 3 matrices take up to 0.5 GB
- 4 matrices take up to 0.25 GB and can leverage the LLC

Working set size [MB]	spmv	conversion	Best preconditioner + solver
thermal2	380.4	451.3	202.2
G3_circuit	359.6	398.1	279.7
bone010	2251.6	3143.4	3034.7
atmosmodd	354.2	415.4	291.5
offshore	198.5	244.0	237.5
pdb1HYS	188.5	249.8	117.4
consph	239.5	283.6	238.9
Serena	5839.3	5965.3	2637.1
Hardesty3	1717.1	1881.9	--
analytics	179.2	89348.9	88.1

- After initialization, WSS cumulative traces of spmv benchmark show 5 distinct phases for 5 evaluated matrix formats
- Corresponding WSS snapshot traces show peak memory requirements at the beginning of each phase



Memory Bandwidth Characteristics

Top-down analysis showed that Ginkgo solver execution is backend bounded mainly by memory

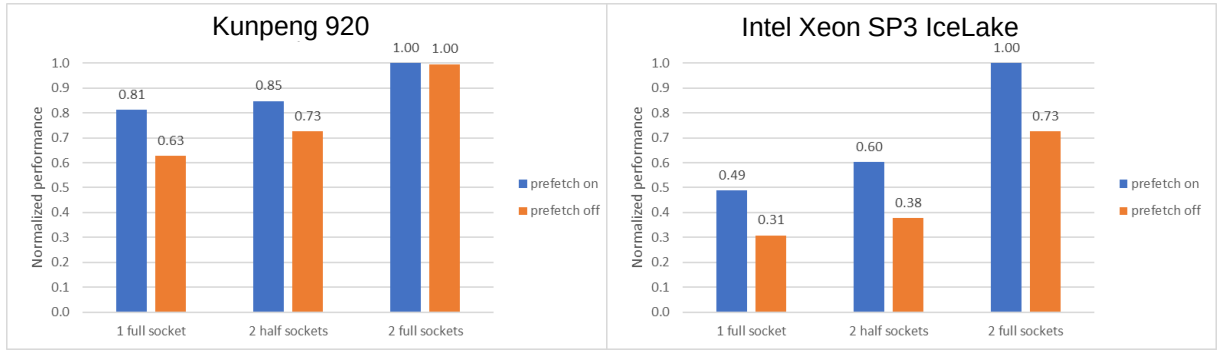
Hardware prefetcher (example: spmv benchmark)

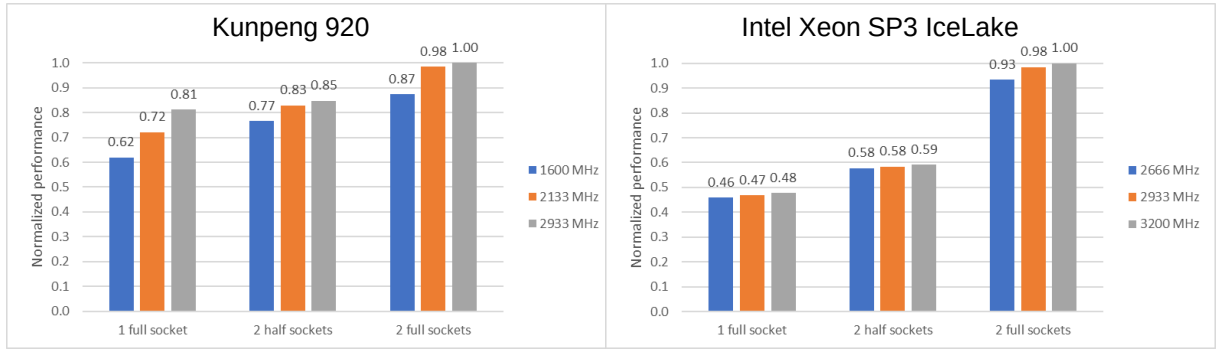
- Intel IceLake's prefetchers show the highest impact, reducing performance up to 38% by switching them off
- Followed by Kunpeng 920 (23%) and AMD Milan (13%)

Impact of DRAM DIMM speed (example: spmv benchmark)

- On Kunpeng 920, performance drops by up to 24% (mem frequency reduced by 45%, DDR4-2933 → 1600)
- On AMD & Intel, performance goes down by 3% and 7% (mem frequency reduced by 17%, DDR4-3200 → 2666)
- Overall, proportional dependency on memory bandwidth (performance partially limited by memory latency as well)

Normalized performance (higher is better)

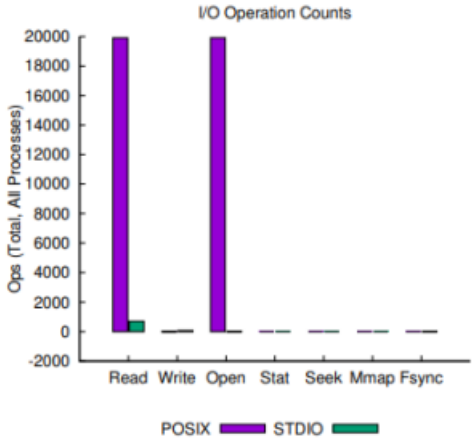




Additional Investigations

- In most cases lower precision single kernels are faster than double, and scomplex kernels faster than dcomplex
- For atmosmodd and pdb1HYS matrices higher precision is faster than lower due to faster convergence

- Darshan traces confirmed that Ginkgo executions have almost no I/O footprint (except initial reading of the matrix), similar to other sparse linear solvers



- Measured negligible overhead of C++ runtime polymorphism

```
Running 1000000 iterations of the CG solver took a total of 4.56531 seconds.
        Average library overhead:     4565.31 [nanoseconds / iteration]
```