Working with Proxy-Applications: *Interesting Findings, Lessons Learned, and Future Directions*

BID WS @PPoPP'22 – 02. Apr. 2022

Jens Domke, Dr. rer. nat. < jens.domke@riken.jp >
High Performance Big Data Research Team, RIKEN R-CCS, Kobe, Japan

Computer simulations create the future

Disclaimer

- No in-depth presentation into single scientific topic
- High-level examples to encourage out-of-the-box thinking and questioning “always done this way” and revisiting old papers

- Intension:

  Strengthen collaboration between domain scientists and HPC architects to develop better Supercomputers

Motivation – Heterogeneous Future

- The “Cambrian explosion” was just the start! (50+ in this fig.)

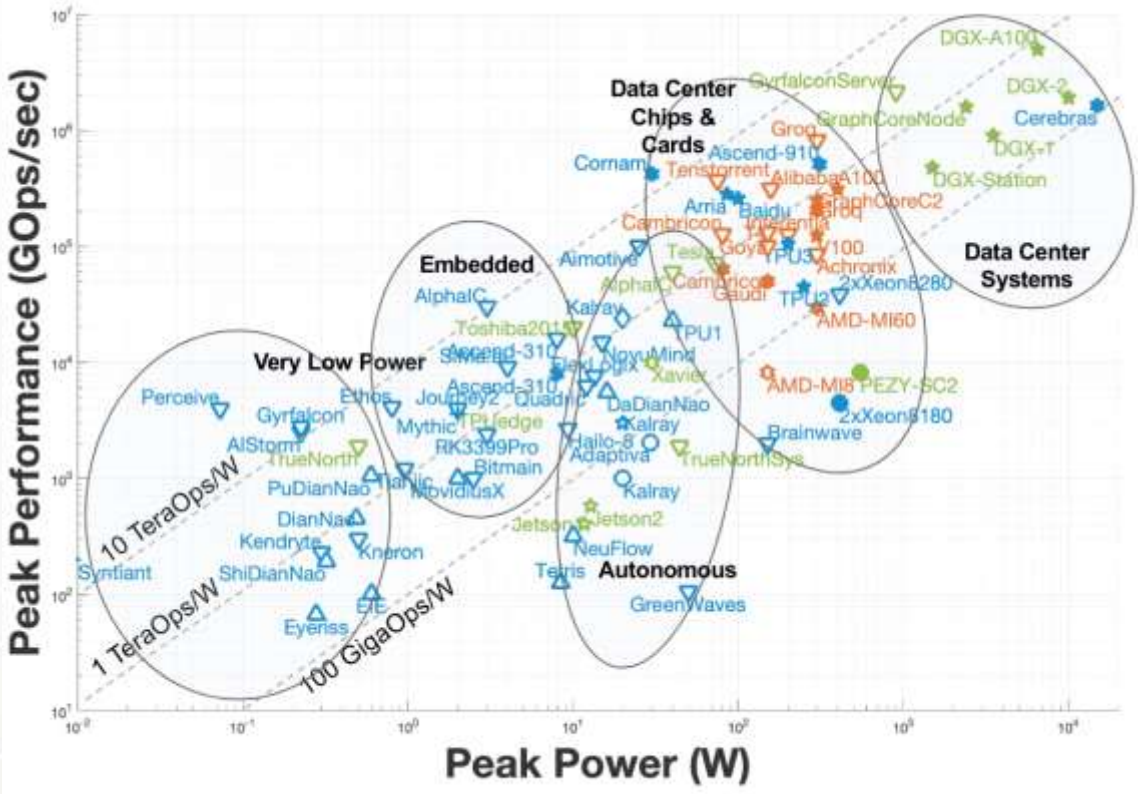


Fig. 2. Peak performance vs. power scatter plot of publicly announced AI accelerators and processors.

Source:
A. Reuther, et al.
“Survey of Machine Learning Accelerators”, 2020

Motivation – Heterogeneous Future

- Increase on almost all fronts:
  - Specialized chips in phones, FPGAs in Intel/AMD CPUs, …
  - Number of programming languages (Rust, Julia, …) and paradigms (CUDA, HIP, oneAPI, Kokkos, RAJA, …)
  - New workloads: containerization, DL/ML, big data, workload-chaining, etc.
  - (except for networks luckily and also unfortunately ☹)

➔ How to make sense of it all? (and fast)

➔ How to determine the best architecture per site?

Challenges – Future Scale-out & Diversification

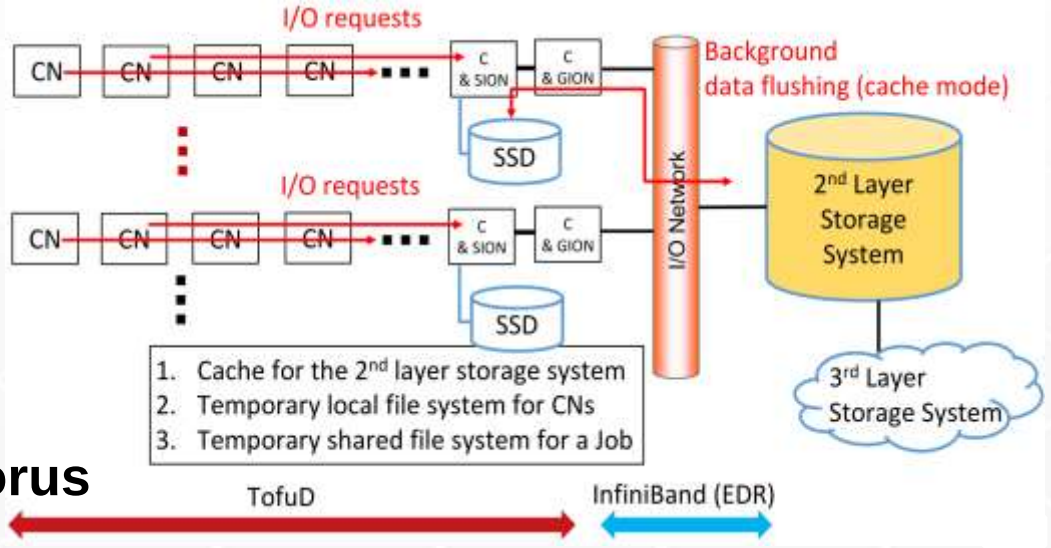
- No more gains from Moore's law

  ➔ Bigger HPC systems w/ more nodes (maybe island design for specialization)

  ➔ Larger interconn. networks

Supercomputer Fugaku

- >158k compute nodes
- 3 networks / topologies
  - CN: 24x23x24 **TofuD** (w/ 2x3x2 subgr.) as **6D torus**
  - Storage: EDR **InfiniBand** for every 16th CN (with fat-tree? topology)
  - Management + outside world: **Ethernet**



Source: Y. Tsujita et al. "Status of Lustre-Based Filesystem at the Supercomputer Fugaku"

Proxy-Apps: Traditional approach…

Computer simulations create the future

“Use benchmark X and run workload Y and report back.”
--HPC procurement

Opportunity for new topologies – HyperX

➔ **First large-scale 2.7 Pflop/s (DP) HyperX installation in the world!**

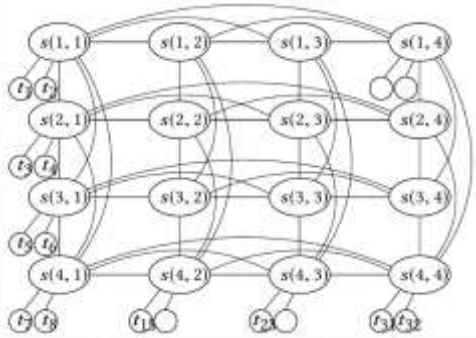


Fig.1: HyperX with n-dim. integer lattice (d_1,…,d_n) base structure fully connected in each dim.

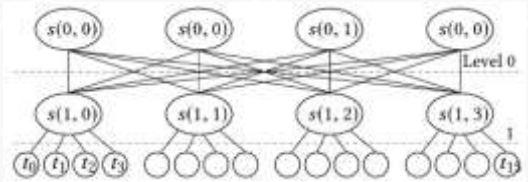
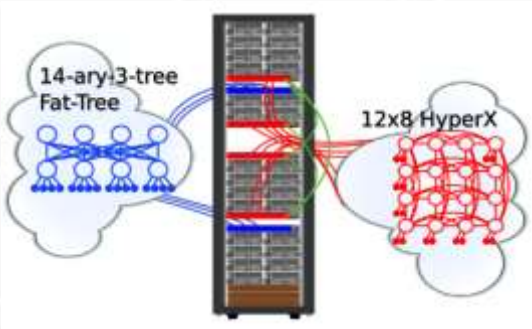


Fig.2: Indirect 2-level Fat-Tree

TokyTech's 2D HyperX:

- **24 racks** (of 42 T2 racks)
- 96 QDR switches (+ 1st rail) **without adaptive routing**
- 1536 IB cables (720 AOC)
- **672 compute nodes**
- **57% bisection bandwidth**



Theoretical Advantages (over Fat-Tree)

- **Reduced HW cost** (less AOC / SW)
- **Only needs 50% bisection BW**
- **Lower latency** (less hops)
- **Fits** rack-based **packaging**

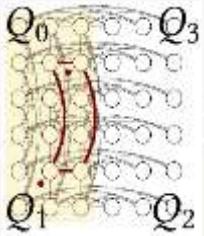
J. Domke et al. "HyperX Topology: First at-scale Implementation and Comparison to the Fat-Tree"

Opportunity for new topologies – HyperX

- TSUBAME2's older gen. of **QDR IB** hardware has **no adaptive routing** ☹
- HyperX with **static/minimum routing** suffers from **limited path diversity** per dimension
  ➔ results in high congestion and low (effective) bisection BW

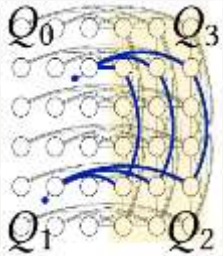
Mitigation Strategies???

- Option 1: Alternative Job Allocation
- **Option 2: Non-minimal, Pattern-aware Routing (PARX)**

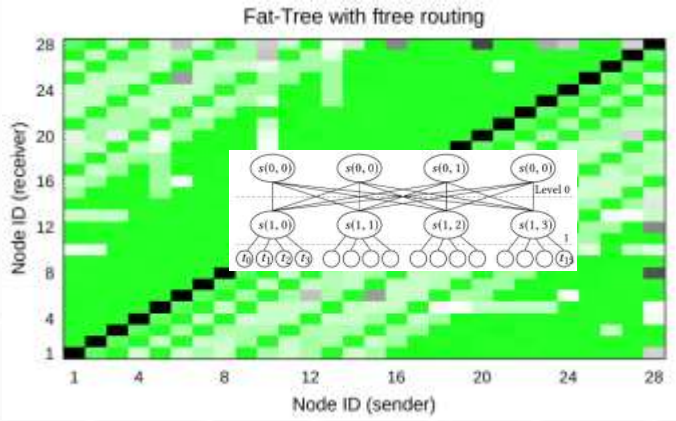


Minimum paths

➔ *combine* ➔

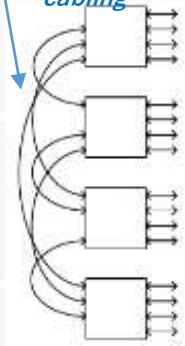


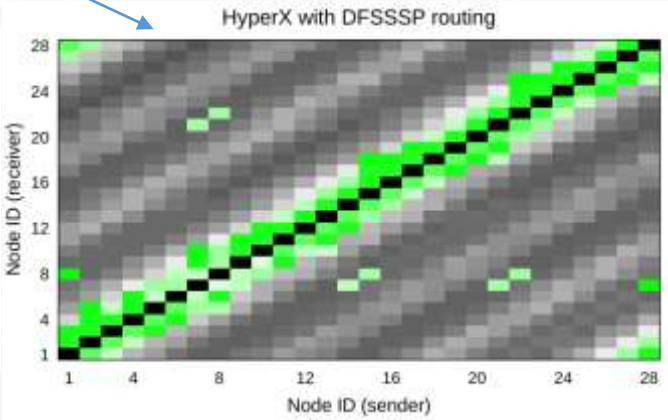
Forced detours



Measured BW in mpiGraph for 28 Nodes

HyperX intra-rack cabling





Opportunity for new topologies – HyperX

1:1 comparison (as fair as possible) of 672-node 3-level Fat-Tree and 12x8 2D HyperX

- NICs of 1st and 2nd rail even on same CPU socket
- Given our HW limitations (few "bad" links disabled)

Wide **variety of benchmarks** and configurations

- **3x Pure MPI benchmarks**
- **9x HPC proxy-apps**
- **3x Top500 benchmarks**
- 4x routing algorithms (incl. PARX)
- 3x rank-2-node mappings
- 2x execution modes

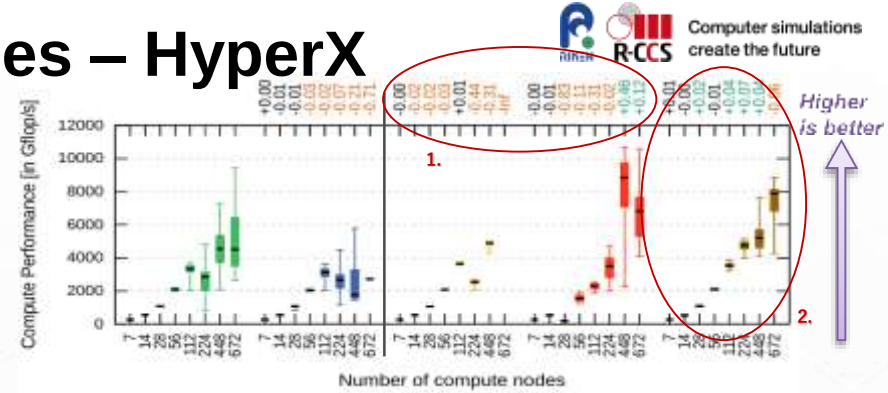


Fig.3: HPL (1GB pp, and 1ppn); scaled 7→672 cn

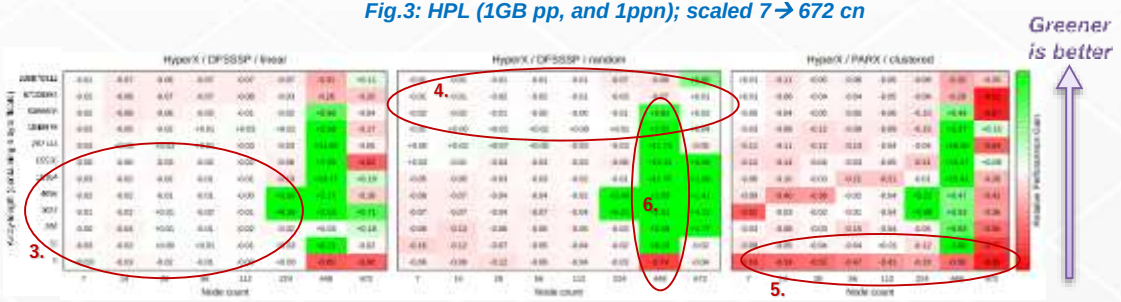


Fig.4: Baidu's (DeepBench) Allreduce (4-byte float) scaled 7→672 cn (vs. "Fat-tree / ftree / linear" baseline)

Primary research questions

Q1: Will reduced bisection BW (57% for HX vs. ≥100% for FT) impede performance?

Q2: Two mitigation strategies against lack of AR? (→ e.g. placement vs. "smart" routing)

1. Placement mitigation can alleviate bottleneck
2. HyperX w/ PARX routing outperforms FT in HPL
3. *Linear* good for small node counts/msg. size
4. *Random* good for DL-relevant msg. size (+/- 1%)
5. "Smart" routing suffered SW stack issues
6. FT + ftree had bad 448-node corner case

Conclusion

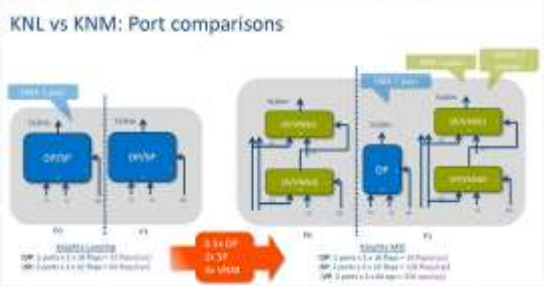
HyperX topology is promising and cheaper alternative to Fat-Trees (even w/o adaptive *R*) !

Proxy-Apps: Motivating vendors / domain scientists…

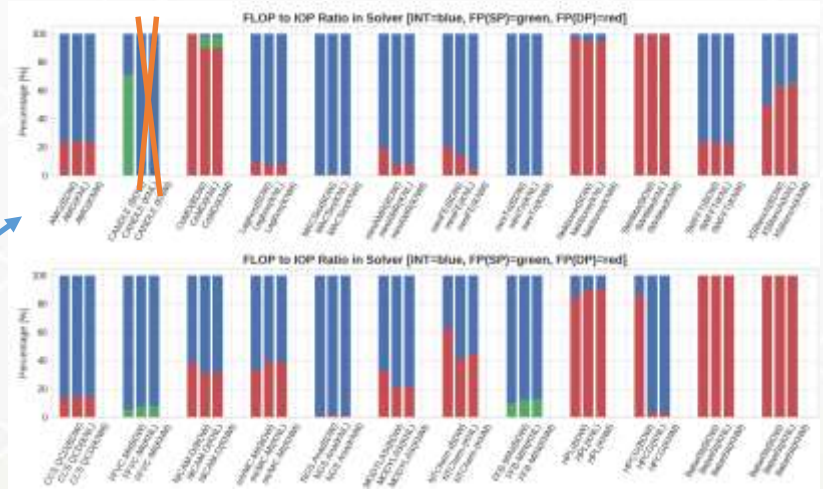
“Wanna do HPC? Then you need many and fast FP64 Units”

--most HPC beginner classes

More Flop/s ➔ more science?!

- Thanks to the (curse of) the TOP500 list, the **HPC community (and vendors) are chasing higher FP64 performance**, thru frequency, SIMD, more FP units, …
- **Motivation:**
  - **Less FP64 units**
    - **Saves power**
    - **Free chip area (ex: FP16)**
    - **Less divergence of "HPC-capable" CPUs from mainstream processors**
- Investigating many proxy-applications:
  - 

  - %FP32 vs. %FP64 vs. %Integer
    ➔ Integer (+DP) heavy (>50%; 16 of 22)
    ➔ Only 4 w/ FP32
    ➔ Only 1 mixed precision



J. Domke et al. "Double-precision FPUs in High-Performance Computing: an Embarrassment of Riches?"

Compare Time-to-Solution in Solver

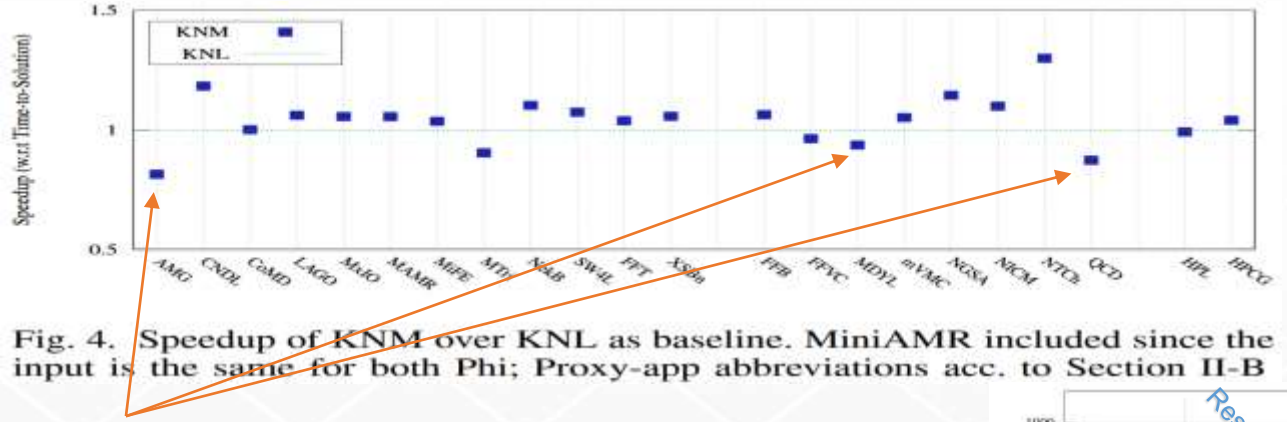
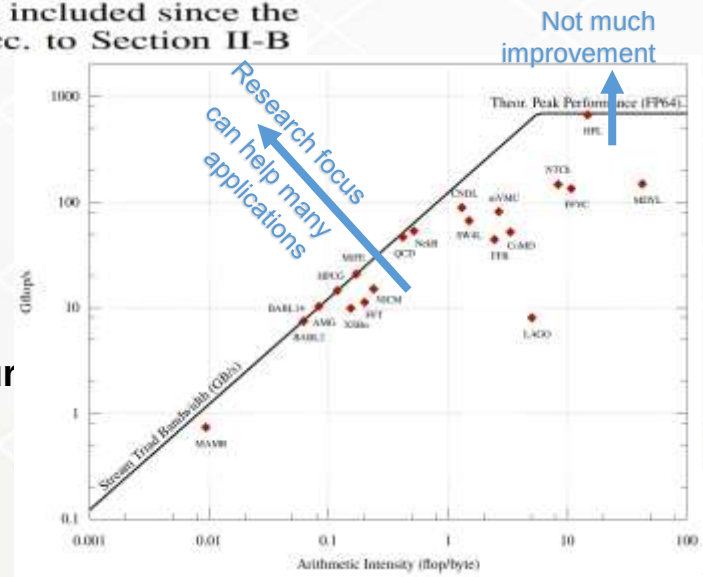


Fig. 4. Speedup of KNM over KNL as baseline. MiniAMR included since the input is the same for both Phi; Proxy-app abbreviations acc. to Section II-B

- **Only 3 apps seem to suffer from missing FP64 unit** (MiniTri: no FP; FFVC: only int+FP32)
- Options for **memory-bound** applications (almost all):
  - **Invest in memory-/data-centric architectures**
  - **Move to FP32/mixed precision ➔ less memory pressur**
- Options for **compute-bound** applications:
  - **Brace for less FP64 units** (driven by market forces) and less “free” performance (10nm, 7nm, 3nm, …then?)



Proxy-Apps: Influencing architecture…

Computer simulations create the future

“Wanna do HPC? Then you need fast [S/D]GEMM.”
--every HPC beginner class

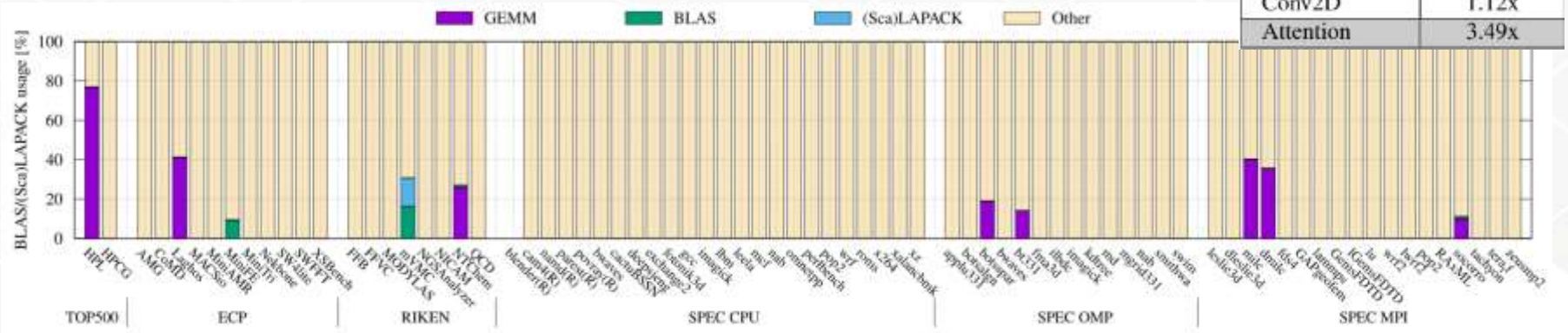
BLAS / GEMM utilization in HPC Applications

Computer simulations create the future

J. Domke et al. "Matrix Engines for High Performance Computing: A Paragon of Performance or Grasping at Straws?"

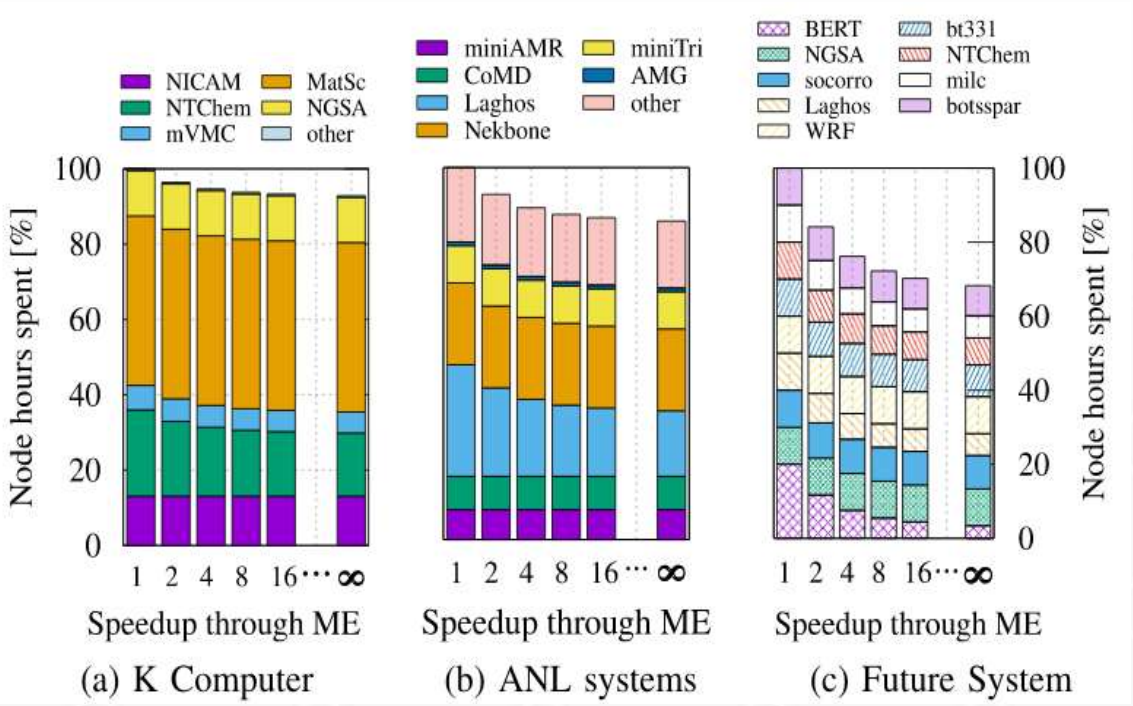
- Analyzed various data sources:
  - **Historical data from K computer**: only 53,4% of node-hours (in FY18) were consumed by applications which had GEMM functions in the symbol table
  - **Library dependencies**: only 9% of Spack packages have *direct* BLAS lib dependency (51.5% have indirect dependency)
  - **TensorCore benefit for DL**: up to 7.6x speedup for MLperf kernels
  - **GEMM utilization in HPC**: sampled across 77 HPC benchmarks (ECP proxy, RIKEN fiber, TOP500, SPEC CPU/OMP/MPI) and measured/profiled via Score-P and Vtune

Benchmark	Speedup
BERT	3.39x
Cosmoflow	1.16x
VGG16	1.71x
Resnet50	1.97x
DeepLabV3	1.75x
SSD300	1.78x
NCF	0.97x
GEMM	7.59x
GRU	3.67x
LSTM	5.69x
Conv2D	1.12x
Attention	3.49x



Estimated Benefit by MEs for HPC Centers

- Thought experiment: Assume we have/had GEMM units in past or future systems.
  - Known: node-hour by domain
  - Sample application with highest BLAS utilization
  - Estimate the node-hour reduction assuming different speedup by ME (2x–8x is realistic dep. on precision)
  - Future system includes 20% DL workloads, other science domains ~10% each



- Results w/ ideal conditions + 4x ME speedup: 5.3% less on K, 10.8% @ANL, 23.8% future system

➔ **HPC can utilize MEs when they come for free, but it's no magic bullet as for DL workloads**

➔ Explore more/other alternatives for Fugaku-next!

Proxy-Apps: Functionality and Regression Testing…

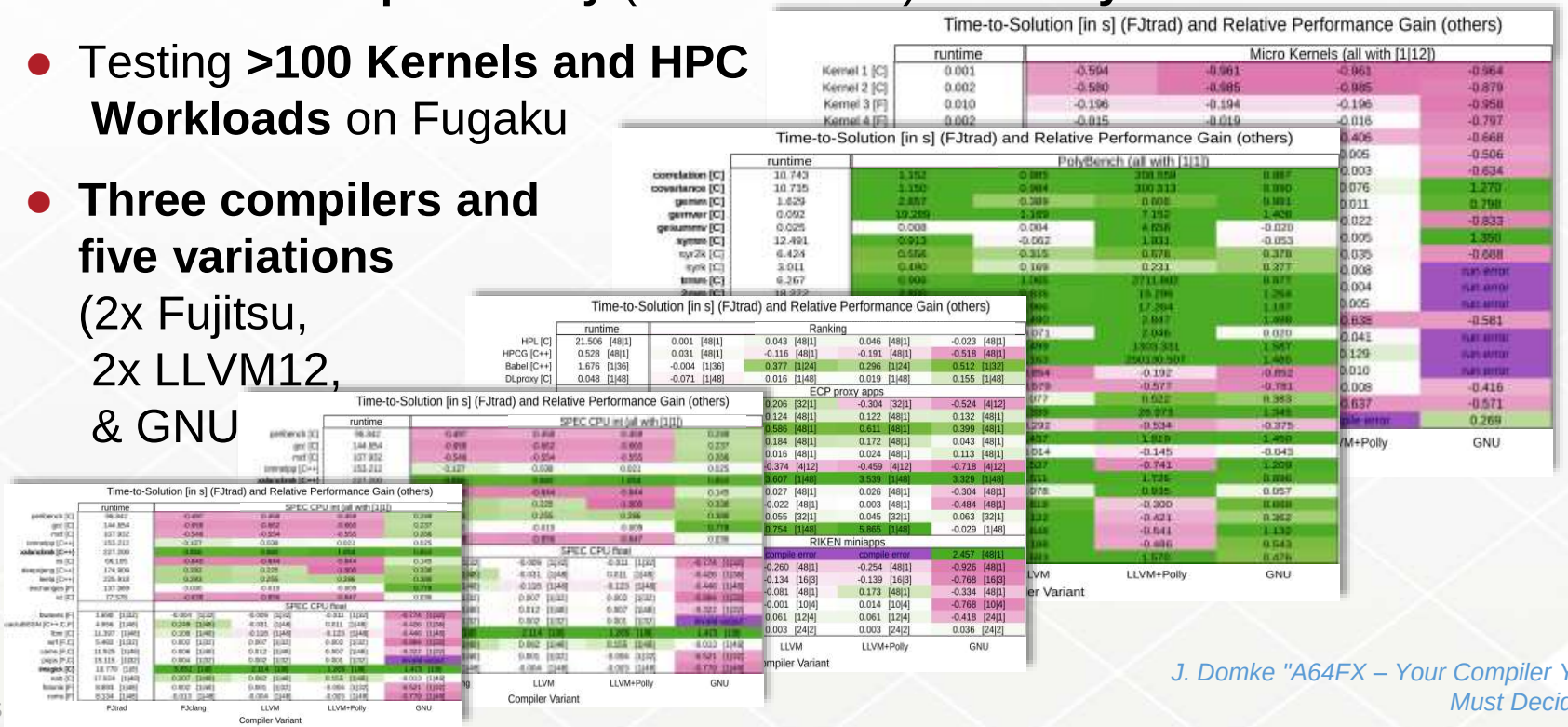
Computer simulations create the future

“Porting an application to A64FX? Just use fcc and –Kfast.”

--Fujitsu

“Silver bullet” compiler choice for A64FX?

- Issue: unexpected advantage of Xeon vs. A64FX in PolyBench
- **Performance portability** (x86→A64FX) **not easy** to achieve
- Testing **>100 Kernels and HPC Workloads** on Fugaku
- **Three compilers and five variations** (2x Fujitsu, 2x LLVM12, & GNU



J. Domke "A64FX – Your Compiler You Must Decide!"

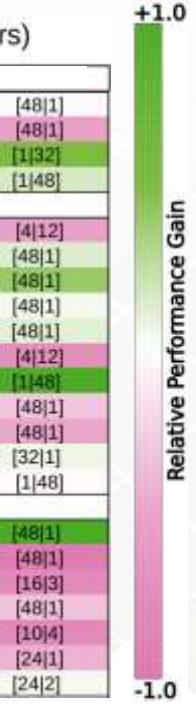
Results for x500, Babel, ECP, Fiber

- Surprising **≈5% gain for HPL** (LLVM or FJclang) despite main time in SSL2
- Same for DLproxy (matmul convolution; SSL2) but even higher gain w/ GNU
- **GNU: 51%** runtime reduction in **stream** (➔ eq. to higher GB/s)
- For **ECP apps use LLVM** or GNU and for **Fiber apps use Fujitsu**'s compiler
- **Avg. speedup: 1.65x** (median 1.09x) with **max. 6.7x** in XSBench
- **[4 | 12] rarely best option**

Time-to-Solution [in s] (FJtrad) and Relative Performance Gain (others)

	runtime	Ranking			
	FJtrad	FJclang	LLVM	LLVM+Polly	GNU
HPL [C]	21.506 [48|1]	0.001 [48|1]	0.043 [48|1]	0.046 [48|1]	-0.023 [48|1]
HPCG [C++]	0.528 [48|1]	0.031 [48|1]	-0.116 [48|1]	-0.191 [48|1]	-0.518 [48|1]
Babel [C++]	1.676 [1|36]	-0.004 [1|36]	0.377 [1|24]	0.296 [1|24]	0.512 [1|32]
DLproxy [C]	0.048 [1|48]	-0.071 [1|48]	0.016 [1|48]	0.019 [1|48]	0.155 [1|48]
ECP proxy apps					
AMG [C]	5.042 [4|12]	0.058 [8|6]	0.206 [32|1]	-0.304 [32|1]	-0.524 [4|12]
CoMD [C]	4.460 [48|1]	0.086 [48|1]	0.124 [48|1]	0.122 [48|1]	0.132 [48|1]
Laghos	45.436 [48|1]	run error	0.586 [48|1]	0.611 [48|1]	0.399 [48|1]
MACSio [C,C++]	24.343 [48|1]	-0.067 [48|1]	0.184 [48|1]	0.172 [48|1]	0.043 [48|1]
miniAMR [C]	18.857 [48|1]	-0.045 [48|1]	0.016 [48|1]	0.024 [48|1]	0.113 [48|1]
miniFE [C++]	0.373 [4|12]	-0.097 [4|12]	-0.374 [4|12]	-0.459 [4|12]	-0.718 [4|12]
miniTRI [C++]	29.182 [32|1]	0.217 [32|1]	3.607 [1|48]	3.539 [1|48]	3.329 [1|48]
Nekbone [F]	1.656 [48|1]	run error	0.027 [48|1]	0.026 [48|1]	-0.304 [48|1]
SW4lite [F,C++]	0.853 [48|1]	0.011 [48|1]	-0.022 [48|1]	0.003 [48|1]	-0.484 [48|1]
SWFFT [F,C]	1.194 [32|1]	0.035 [32|1]	0.055 [32|1]	0.045 [32|1]	0.063 [32|1]
XSBench [C]	1.649 [1|48]	0.323 [1|48]	0.754 [1|48]	5.865 [1|48]	-0.029 [1|48]
RIKEN miniapps					
FFB [F,C,C++]	29.877 [4|1]	compile error	compile error	compile error	2.457 [48|1]
FFVC [C++,F]	11.558 [1|36]	-0.026 [1|36]	-0.260 [48|1]	-0.254 [48|1]	-0.926 [48|1]
MODYLAS [F]	29.262 [16|3]	0.001 [16|3]	-0.134 [16|3]	-0.139 [16|3]	-0.768 [16|3]
mVMC [C,F]	15.026 [24|2]	-0.002 [24|2]	-0.081 [48|1]	0.173 [48|1]	-0.334 [48|1]
NICAM [F]	7.645 [10|4]	-0.003 [10|4]	-0.001 [10|4]	0.014 [10|4]	-0.768 [10|4]
NTChem [F]	9.440 [12|4]	-0.001 [12|4]	0.061 [12|4]	0.061 [12|4]	-0.418 [24|1]
QCD [F]	8.048 [24|2]	0.002 [24|2]	0.003 [24|2]	0.003 [24|2]	0.036 [24|2]

Compiler Variant



“Silver bullet” compiler choice for A64FX?

Conclusions:

- **C1**: recomm. usage model of **4 ranks and 12 threads often suboptimal**
- **C2**: **no “silver bullet” compiler** for A64FX (yet)
  - Dep. on situation, but some hint: **Fujitsu for Fortran codes,** and **GNU for integer-intensive apps,** and **any clang-based compilers for C/C++**
- **C3**: Twitter summary: “if Xeon is 70x faster than A64fx, suspect the compiler”

➔ Test all available compilers, and explore other rank/thread mappings!

Implications for Fugaku Users:

- LLVM 13 incl. “classic” flang (source /home/apps/oss/llvm-v13.0.0/init.sh)
  - SVE support still alpha ➔ expect even more performance with v14
  - Using Fujitsu’s SSL2 not always trivial or error-free

Proxy-Apps: Many years later …

“Uff... (to say it politely).”
--me

Drowning in overlapping Proxy-Apps (>>100)

- HPC challenge benchmark (HPCC)
- Exascale Computing Project (**ECP**) Proxy Applications
- Center for Efficient Exascale Discretizations (**CEED**) Miniapps
- DOE's **CORAL-2** Benchmarks and RIKEN's **Fiber** / TAPP
- European Union's PRACE Unified European Application Benchmark Suite (**UEABS**)
- **SPEC**, BenchCouncil, Intel's **HiBench**, DeathStarBench, Baidu's DeepBench, and MLCommons' **MLPerf**
- **Traditional**: HPL, HPCG, stream, Graph, and Intel's IMB, …

cos θ-similarity for **cache**-related perf. counters

	ExaMiniMD	LAMMPS	MiniQMC	QMCPack	sw4lite	sw4	SWFFT	HACC	pennant	snap
ExaMiniMD	0.00	5.02	54.54	38.73	11.70	12.49	6.58	6.38	13.21	7.13
LAMMPS	5.02	0.00	54.69	38.62	15.66	16.27	4.87	6.38	13.60	10.88
MiniQMC	54.54	54.69	0.00	17.15	47.12	46.08	50.02	48.98	42.16	49.15
QMCPack	38.73	38.62	17.15	0.00	32.64	31.67	33.92	32.94	26.29	33.78
sw4lite	11.70	15.66	47.12	32.64	0.00	1.15	13.41	11.40	11.15	5.07
sw4	12.49	16.27	46.08	31.67	1.15	0.00	13.74	11.70	10.69	5.69
SWFFT	6.58	4.87	50.02	33.92	13.41	13.74	0.00	2.24	9.09	8.80
HACC	6.38	6.38	48.98	32.94	11.40	11.70	2.24	0.00	7.86	6.87
pennant	13.21	13.60	42.16	26.29	11.15	10.69	9.09	7.86	0.00	9.37
snap	7.13	10.88	49.15	33.78	5.07	5.69	8.80	6.87	9.37	0.00

Source: D. Richards et al. "Best Practices for Using Proxy Apps as Benchmarks"

Lessons-Learned from using Proxy-Apps

- Inherently **complex** and **implementation biases**
  - Mostly implemented in Fortran and C[/C++], and tuned over the years
  - Highly tuned CUDA (and “legacy” CPU) from ECP efforts
- Huge porting & **maintenance overhead**
  - Macros, separate code paths, hand-written makefiles, …
  - Costly refactoring for: data layouts, parallelization strategies, accelerators
- Insufficient **inputs, testing, documentation**
  - Issues with non-std compilers and applying perf. analysis tools
  - Strong- and weak-scaling, varying input sizes, independent of #MPI/#OMP
- Lack of efficiency reporting
  - *How much performance achieved vs. the peak theoretical performance?*

Let’s clean up this mess …

Computer simulations create the future

“Octopodes to the rescue.”
--RIKEN & DOE

Fugaku Enhancement & Co-Design for Future

- Superseding current proxy-apps: ***Octopodes***
  - Downsides w/ Fiber/proxy-apps (s. Fugaku R&D)
  - On-going collaboration / brainstorming phase with DOE labs (position paper release in Apr.'22)
  - Set of highly-parameterizable, easily-amendable, DWARF-like problem representations
  - **Common "language" between HPC users, system operators, co-designers, and vendors** to describe the to-be-solved scientific problems: What needs to be computed, and how it can be computed?

https://www.pinterest.fr/pin/145170787976811341/

➔ Apply ML to identify, parameterize, and categorize compute phases

S. Matsuoka, J. Domke, M. Wahib, A. Drozd, A. Chien, R. Bair, J. S. Vetter, J. Shalf
"Preparing for the Future –Rethinking Proxy Applications"
to appear in Computing in Science & Engineering

Example of one Octopode: Matmul

- Input **shapes**: such as squared, rectangular, and tall/skinny
- Various numerical **precisions** (i.e., from fp128 to bfloat16, etc)
- **Batched** and non-batched executions modes
- **Dense** matrix-matrix operations, matrix-vector, sparse matrices
- **Sparse** matrix: random, realistic blocks, *Matrix Market*

➔ Use C++ templating to generate as many variants as possible to train ML models

➔ One Octopode for each distinct compute phase or math kernel

➔ In-between Berkeley DWARFs and Proxy-Apps

Usage of Octopodes for Co-Design

- "What needs & how can it be computed" not "Here is how you have to do it"
- For **performance modeling** of real workloads: identify compute phases which can be mapped to one or more Octopodes ➔ combine perf. model of the 'easier to understand' Octopodes ➔ approx. perf. model of full workloads
- For **vendors**:
  - Allowed tuning freedom for the Octopodes, i.e., changes of algo., implementation, integer/float. precision, data layout, etc., *as long as* intended result is the same
  - Accurately model consumer workloads ➔ Less over/under-selling of hardware
- **Porting** of user codes to new system:
  - Act as demonstrator for users to show how to port
  - ML/AI to identify phases can be used as helper for porting of real codes
- **Better suited for co-design tools**, e.g. compiler tests, regression testing, simulators (gem5/SST/CODES/…), quick "What-If" tools, etc.

Summary and Call for Co-Design Collaboration

- **Better & versatile simulators**
  - Consolidating and enhancing existing infrastructure
- More **testbeds of diff. scales**
  - Some netw. issues manifest at scale
  - Repurpose decommissioned system
  - Shared access
- **Better performance metrics** reflecting real-life **and tools** to collect them
  - Focus on more async., automated, easy-to-use for admins
- Cleaner **build environments** and user training
- Extensive **network co-design**
  - More insight into apps/workloads
- More **bottleneck analysis**
  - Fix the lack for memory-centric tools
- **1!** (better) **community test suite**
  - Easy migration, inter-op. testing, regression testing, etc.
- **Octopodes for co-design ☺**

Lots of exciting R&D challenges ahead!

Job & Collaboration Opportunities

- Collaborations and job opportunities:
  - **We are hiring!** Check out our research teams and open positions: https://www.riken.jp/en/research/labs/r-ccs/ and https://bit.ly/3faax8v
- Internship/fellowship for students (Bachelor→PhD):
  - Fellowship: https://www.riken.jp/en/careers/programs/index.html
  - Internship: https://www.r-ccs.riken.jp/en/about/careers/internship/
- Supercomputer **Fugaku**:
  - Apply for node-hours: https://www.r-ccs.riken.jp/en/fugaku/user-guide/
  - Interactive, virtual tour: https://www.r-ccs.riken.jp/en/fugaku/3d-models/ and https://www.youtube.com/watch?v=f3cx4PGDGmg