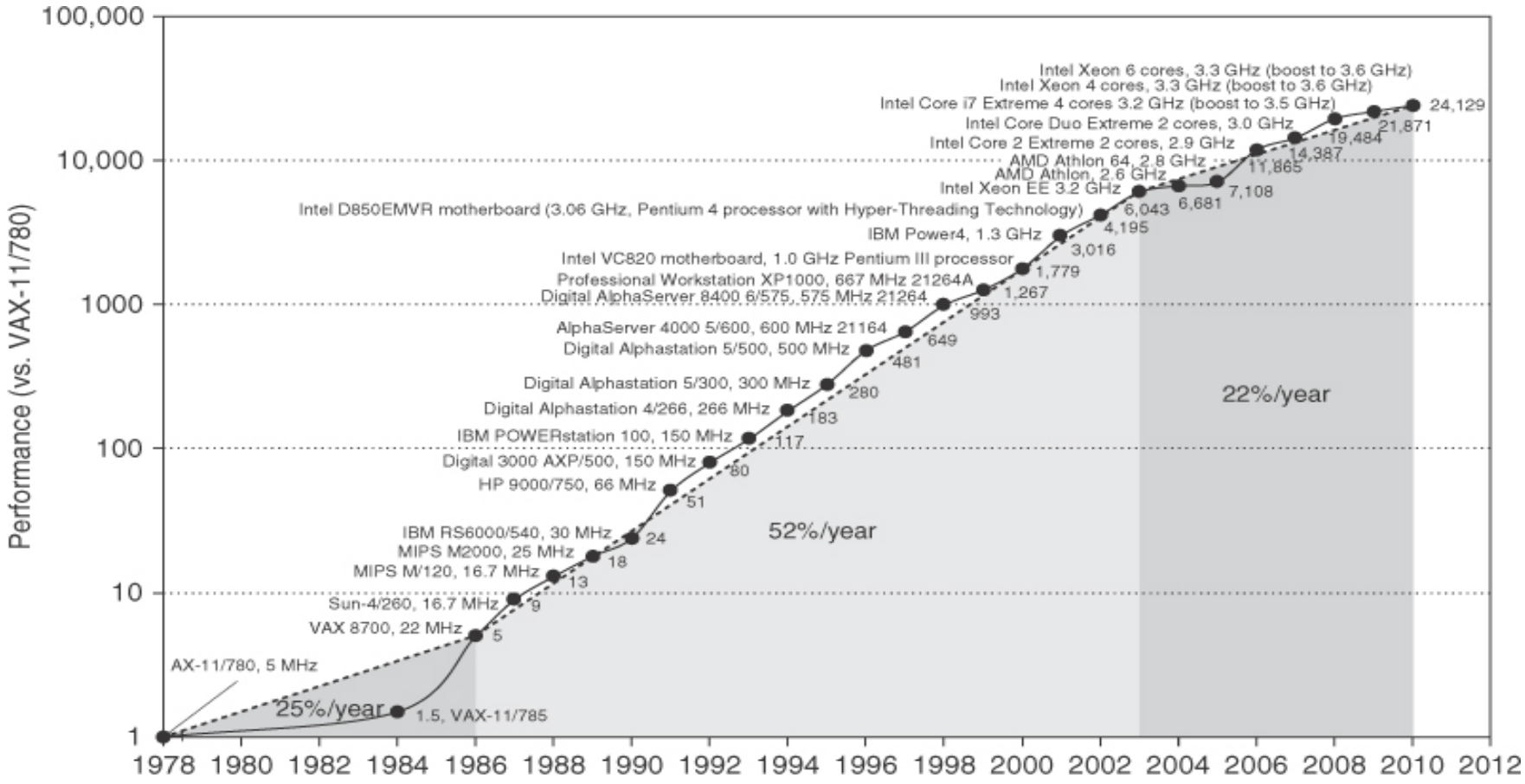
Adaptive-Precision Algorithms for Sparse Linear Algebra

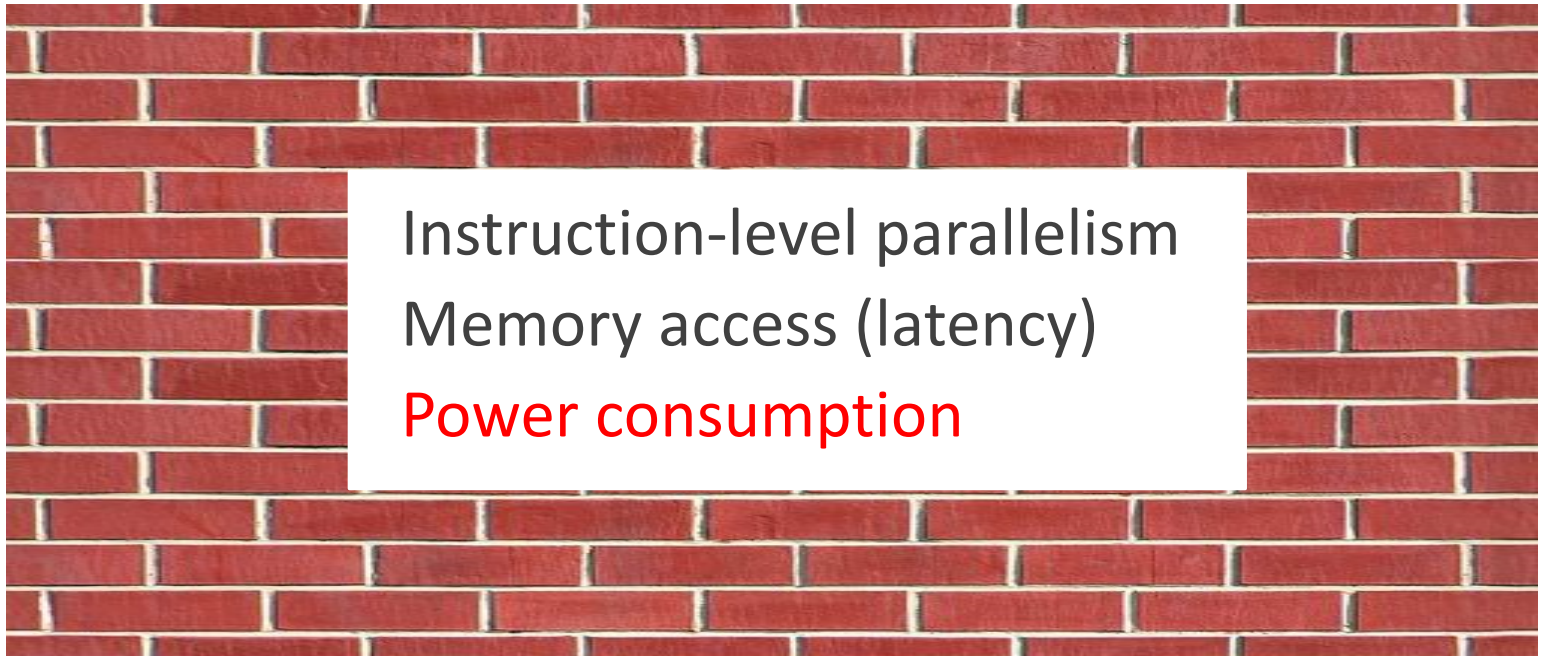
Enrique S. QUINTANA-ORTÍ
Professor of Computer Architecture
Group leader High Performance Computing & Architectures (HPC&A) group
http://www.uji.es/~quintana

Motivation



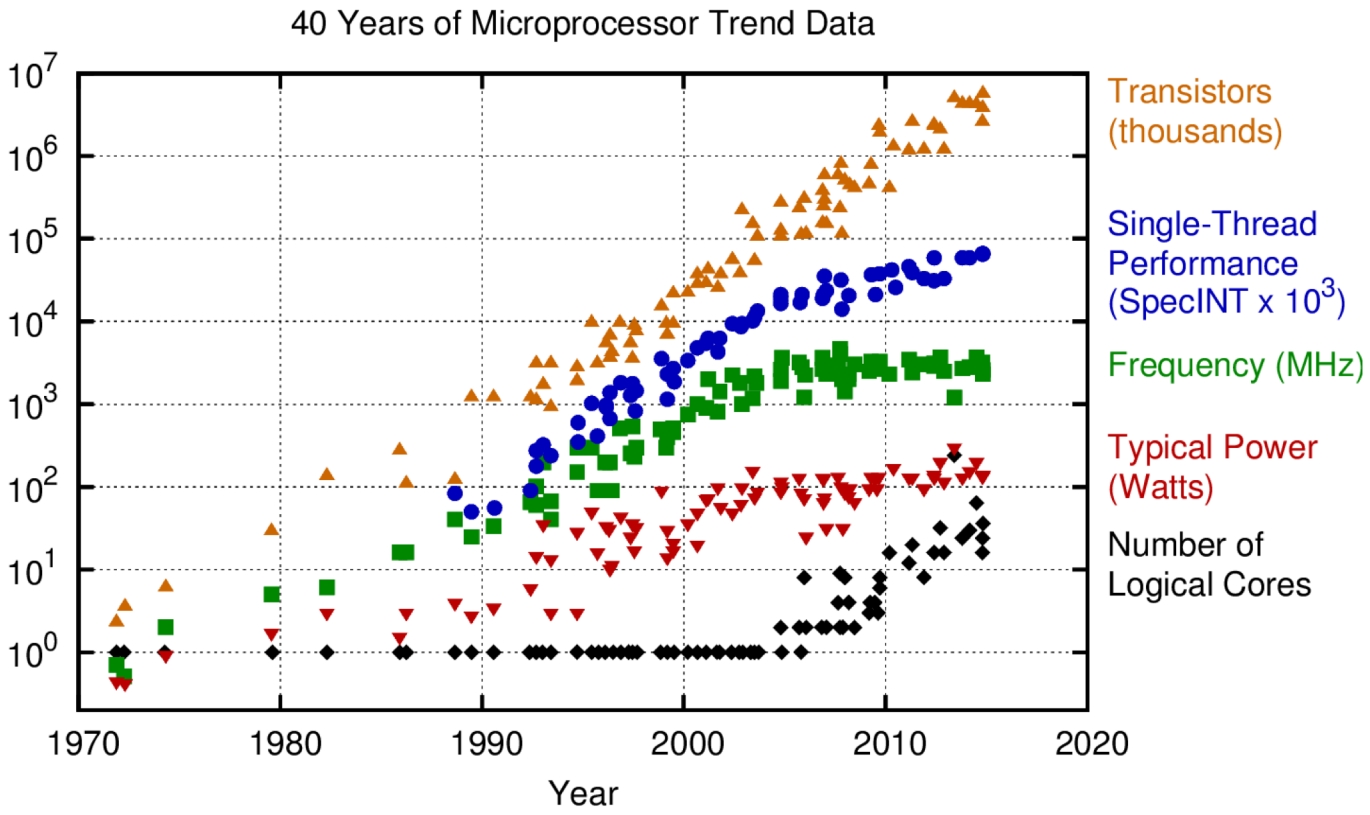
"Computer Organization and Design". D. A: Patterson, J. L. Hennessy, 2014

Motivation


Instruction-level parallelism
Memory access (latency)
Power consumption

Motivation

- Dennard's scaling vs Moore's Law



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Intel Xeon Platinum 8180

Q3'17

14 nm

2.5 GHz

TDP 205 W

28 cores/56 threads

Motivation

- Reduce energy consumption!
  - Costs over lifetime of an HPC facility often exceed acquisition costs
  - Hazard for health and environment
  - Heat reduces hardware reliability

- Personal view
  - Hardware features some power-saving mechanisms (from mobile/embedded to desktop/server)
  - Scientific apps. are in general energy-oblivious

Outline

- What can I do? … A recipe to saving energy:

  Optimize performance!!!

1. Choose the “right” hardware
2. Dynamic Voltage-Frequency Scaling (DVFS)
3. Dynamic Concurrency Throttling (DCT)
4. Avoid polling
5. Near Threshold Voltage Computing (NTVC)
6. Energy-proportional hardware
7. Virtualization of HPC resources
8. Approximate computing/adaptive precision

Outline

- DISCLAIMER
  - Sorry, most of the examples come from linear algebra:
    - Solution of dense/sparse linear systems via direct/iterative methods
    - Solution of eigenvalue problems
  - …but the message carries over to many other math kernels for scientific and engineering applications

1. Choose the right hardware

- The Conjugate Gradient (CG) method is representative of the performance/energy efficiency attained by real scientific applications (HPCCG benchmark)
- Performance depends on:
  - Target architecture: frequency-voltage setting, #cores, arithmetic floating-point precision, etc.
  - Compiler optimizations
  - Sparsity pattern
  - Storage format
  - Programmer's optimization effort

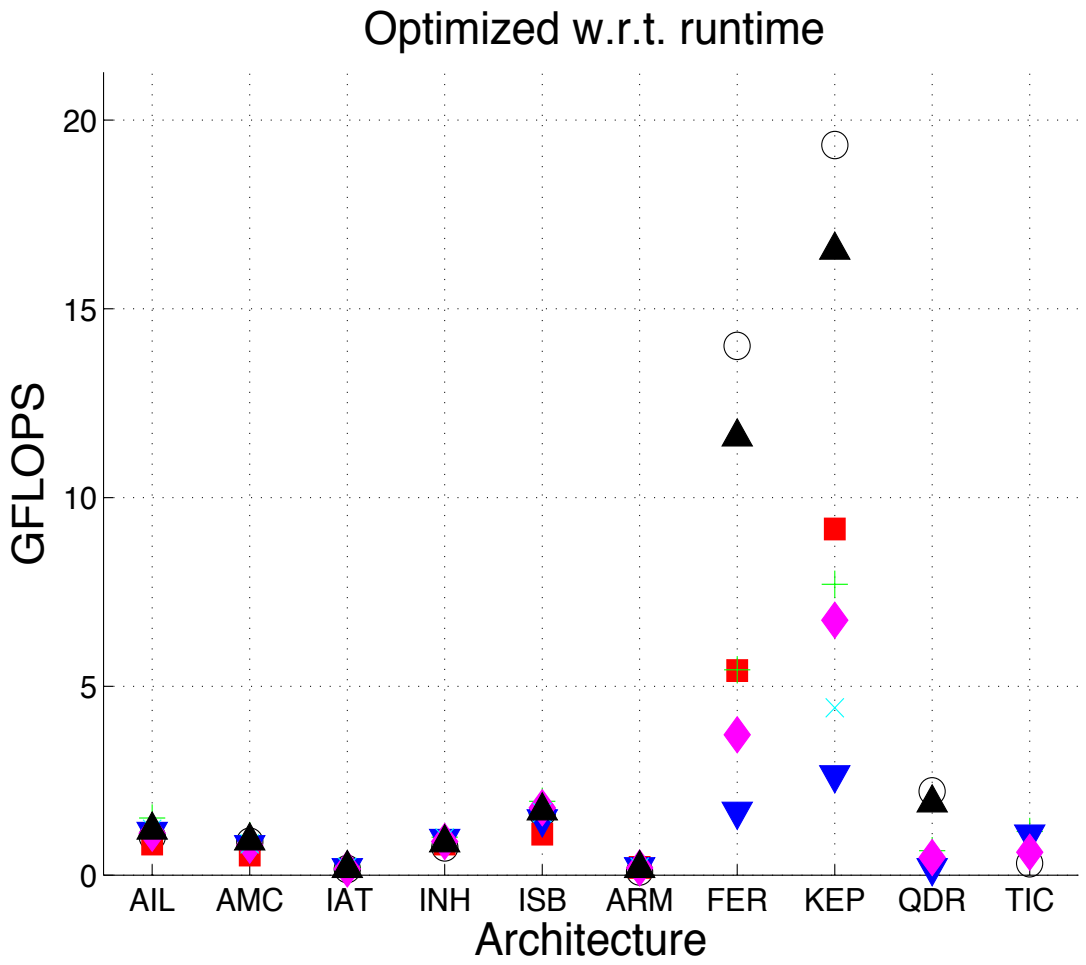
1. Choose the right hardware

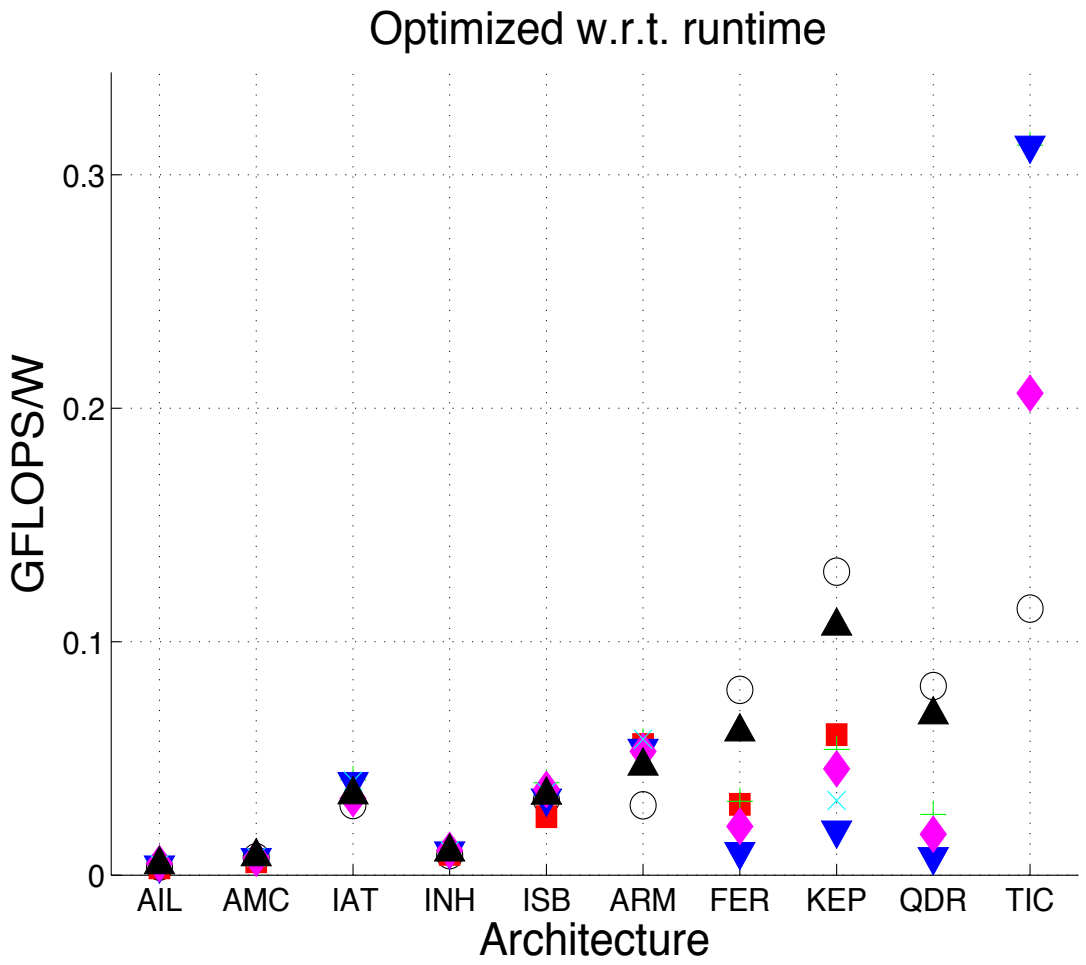
Acron.	Architecture	Total #cores	Frequency (GHz) – Idle power (W)	RAM size, type	Compiler
AIL	AMD Opteron 6276 (Interlagos)	8	1.4–167.29, 1.6–167.66 1.8–167.31, 2.1–167.17 2.3–168.90	64GB, DDR3 1.3GHz	gcc 4.4.6
AMC	AMD Opteron 6128 (Magny-Cours)	8	0.8–107.48, 1.0–109.75, 1.2–114.27, 1.5–121.15, 2.0–130.07	48GB, DDR3 1.3GHz	gcc 4.4.6
IAT	Intel Atom D510	2	0.8–11.82, 1.06–11.59, 1.33-11.51, 1.6–11.64	1GB, DDR2 533MHz	gcc 4.5.2
INH	Intel Xeon E5504 (Nehalem)	8	2.0–280.6, 2.33–281.48, 2.83–282.17	32GB, DDR3 800MHz	gcc 4.1.2
ISB	Intel E5-2620 (Sandy-Bridge)	6	1.2–93.35, 1.4–93.51, 1.6–93.69, 1.8–93.72, 2.0–93.5	32GB, DDR3 1.3GHz	gcc 4.1.2
ARM	ARM Cortex A9	4	0.62–11.7, 1.3–12.2	2GB, DDR3L	gcc 4.5.2
FER	Intel Xeon E5520 NVIDIA Tesla C2050 (Fermi)	8 448	1.6–222.0, 2.27–226.0 1.15	24GB, 3GB, GDDR5	gcc 4.4.6 nvcc 4.2
KEP	Intel Xeon i7-3930K NVIDIA Tesla K20 (Kepler)	6 2,496	1.2–106.30, 3.2–106.50 0.7	24GB, 5GB, GDDR5	gcc 4.4.6 nvcc 4.2
QDR	ARM Cortex A9 NVIDIA Quadro 1000M	4 96	0.102–11.2, 1.3–12.2 1.4	2GB, DDR3L 2GB, DDR3	gcc 4.5.2 nvcc 4.2
TIC	Texas Instruments C6678	8	1.0–18.0	512MB, DDR3	cl6x 7.4.1

1. Choose the right hardware

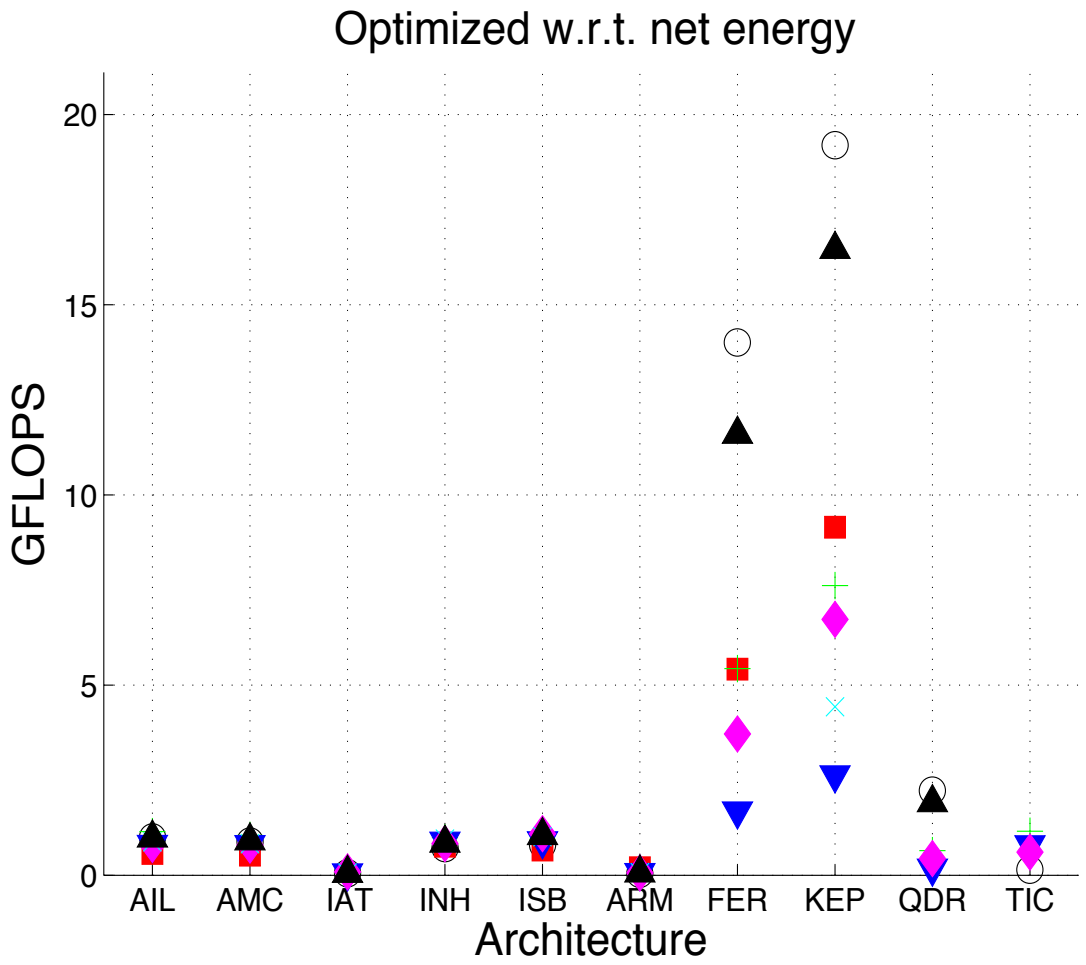
- Optimization effort:
  - Multicore x86-based: Intel MKL with CSR and BCSR, and CSB library
  - Other multicore: CSR+OpenMP
  - GPUs: ELLPACK & SELL-P, with further optimizations (described in last block)

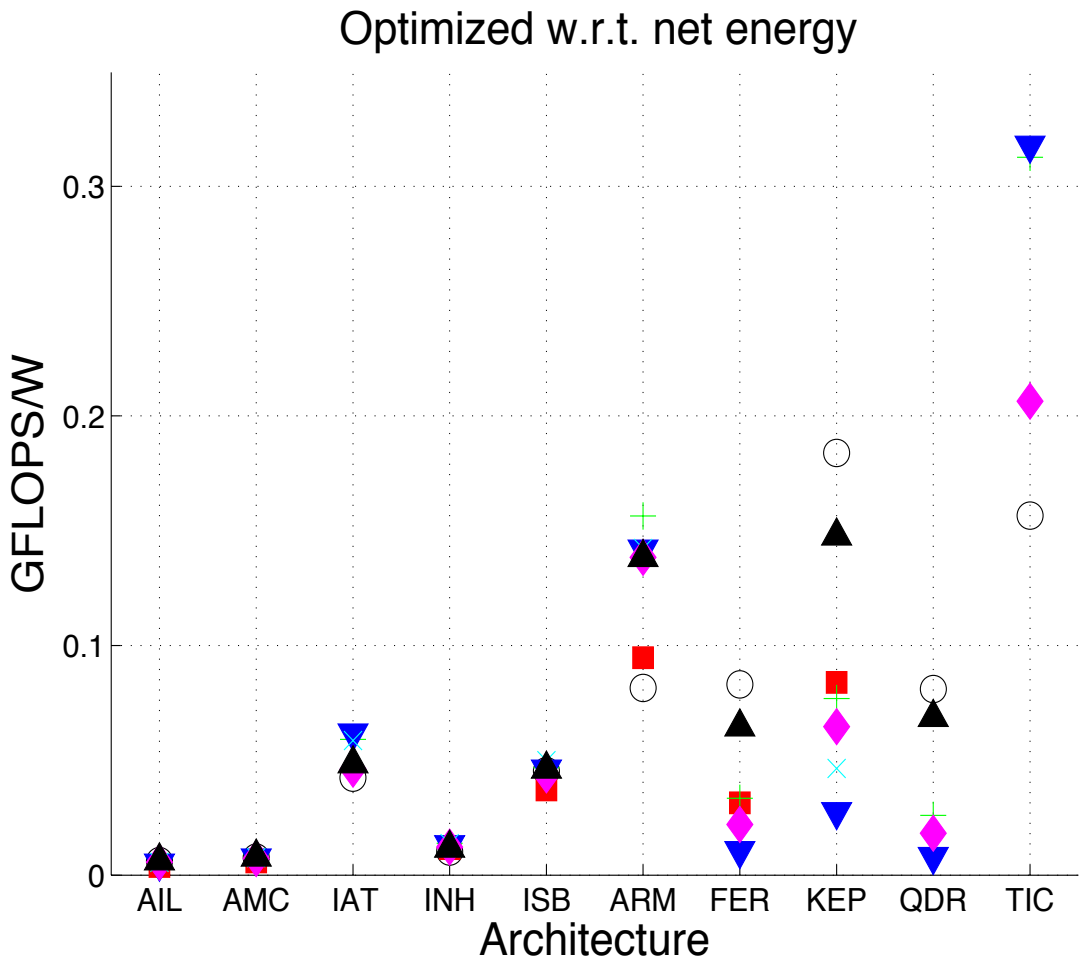
1. Choose the right hardware





1. Choose the right hardware





2. DVFS

- Dense linear algebra kernels are the building blocks for many scientific and engineering applications: _GEMV, _GEMM
- (Dense) LU factorization is the basis for the LINPACK benchmark (Top500/Green500 lists): _GETRF
- Routines are highly optimized as part of vendor implementations of BLAS/LAPACK (Intel MKL, AMD ACML, IBM ESSL, NVIDIA CuBLAS, etc.)

2. DVFS

- Current processors adhere to the ACPI (Advanced Configuration and Power Interface) standard:
  - P-states:
    - Adjust voltage-frequency to the workload in execution
    - Control by the Linux kernel or user
  - C-states:
    - Suspend processor components to save energy
    - Can waste energy if CPU needs to be activated soon
    - No control by the user

2. DVFS

- P-states

Conf.	ARM Cortex-A7		ARM Cortex-A15		ARM Cortex-A53		ARM Cortex-A57	
	Freq.	Voltage	Freq.	Voltage	Freq.	Voltage	Freq.	Voltage
C_1	0.200	0.913	0.200	0.912	0.450	0.820	0.450	0.810
C_2	0.400	0.913	0.400	0.912	0.575	0.860	0.625	0.850
C_3	0.600	0.951	0.600	0.912	0.700	0.910	0.800	0.900
C_4	0.800	1.026	0.800	0.925	0.775	0.960	0.950	0.950
C_5	1.000	1.101	1.000	0.973	0.850	1.010	1.100	1.000
C_6	1.200	1.176	1.200	1.023	–	–	–	–
C_7	1.400	1.273	1.400	1.062	–	–	–	–
C_8	–	–	1.600	1.115	–	–	–	–
C_9	–	–	1.800	1.191	–	–	–	–
C_{10}	–	–	2.000	1.318	–	–	–	–

2. DVFS

- P-states

Architecture	Configu-ration	GEMV			GEMM			GETRF		
		G	P	EE	G	P	EE	G	P	EE
ARM Cortex-A7	C_1	0.271	0.064	4.238	0.758	0.072	10.473	0.554	0.060	9.259
	C_2	0.501	0.115	4.360	1.573	0.140	11.204	1.193	0.119	9.984
	C_3	0.677	0.166	4.086	2.355	0.217	10.834	1.809	0.186	9.733
	C_4	0.802	0.233	3.440	3.258	0.328	9.937	2.398	0.279	8.603
	C_5	0.911	0.319	2.853	4.070	0.483	8.426	2.998	0.409	7.323
	C_6	0.999	0.417	2.395	4.893	0.672	7.278	3.536	0.561	6.303
	C_7	1.000	0.541	1.848	5.630	0.943	5.968	3.980	0.784	5.078
ARM Cortex-A15	C_1	0.381	0.188	2.028	3.502	0.496	7.067	2.205	0.388	5.679
	C_2	0.718	0.335	2.141	7.109	0.970	7.328	4.619	0.761	6.068
	C_3	0.997	0.471	2.115	10.652	1.436	7.418	7.067	1.132	6.245
	C_4	1.227	0.582	2.108	14.165	1.926	7.356	9.232	1.506	6.130
	C_5	1.396	0.768	1.817	17.757	2.686	6.611	11.702	2.120	5.519
	C_6	1.539	0.981	1.568	21.145	3.632	5.822	13.700	2.793	4.905
	C_7	1.648	1.182	1.394	24.344	4.562	5.336	15.856	3.532	4.489
	C_8	1.756	1.489	1.179	27.710	5.978	4.635	17.152	4.539	3.779
	C_9	1.728	1.855	0.931	–	–	–	–	–	–
	C_{10}	1.744	2.569	0.679	–	–	–	–	–	–
ARM Cortex-A53	C_1	0.877	0.198	4.425	7.461	0.359	20.787	4.901	0.272	18.148
	C_2	1.008	0.259	3.887	9.488	0.510	18.594	6.221	0.374	16.818
	C_3	1.106	0.327	3.387	11.271	0.685	16.447	7.453	0.494	15.204
	C_4	1.148	0.399	2.880	12.533	0.855	14.658	8.164	0.620	13.161
	C_5	1.191	0.470	2.536	13.629	1.045	13.040	8.883	0.727	12.333
ARM Cortex-A57	C_1	0.733	0.270	2.715	6.159	0.536	11.482	4.189	0.509	8.222
	C_2	0.972	0.404	2.408	8.491	0.843	10.072	5.801	0.781	7.429
	C_3	1.163	0.560	2.077	10.812	1.214	8.903	7.375	1.157	6.373
	C_4	1.286	0.709	1.814	12.698	1.586	8.006	8.702	1.490	5.839
	C_5	1.375	0.858	1.603	14.538	2.059	7.059	9.982	1.952	5.115

2. DVFS

- Control of P-states by user possible via `cpufreq`, but too slow:
  - 225 µseconds in Intel E5-2620
  - Directly writing in MSR (in µseconds):

f_T \ f_S	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0
1.2	–	28.98	29.11	29.07	28.98	29.14	29.57	29.08	29.32
1.3	34.10	–	29.14	28.90	28.99	29.23	28.98	29.13	28.78
1.4	35.33	34.14	–	28.96	29.00	28.65	29.04	29.26	28.90
1.5	35.60	34.68	33.29	–	28.39	28.57	28.83	28.99	28.68
1.6	35.61	35.09	34.16	33.63	–	28.08	28.12	28.34	28.50
1.7	35.63	35.41	34.74	34.30	32.35	–	28.27	28.33	28.21
1.8	36.69	36.30	35.40	35.05	34.50	33.70	–	28.36	28.23
1.9	36.89	36.21	36.02	35.29	34.63	34.19	32.52	–	27.70
2.0	37.22	36.65	36.27	35.49	35.24	34.31	33.54	32.54	–

3. DCT

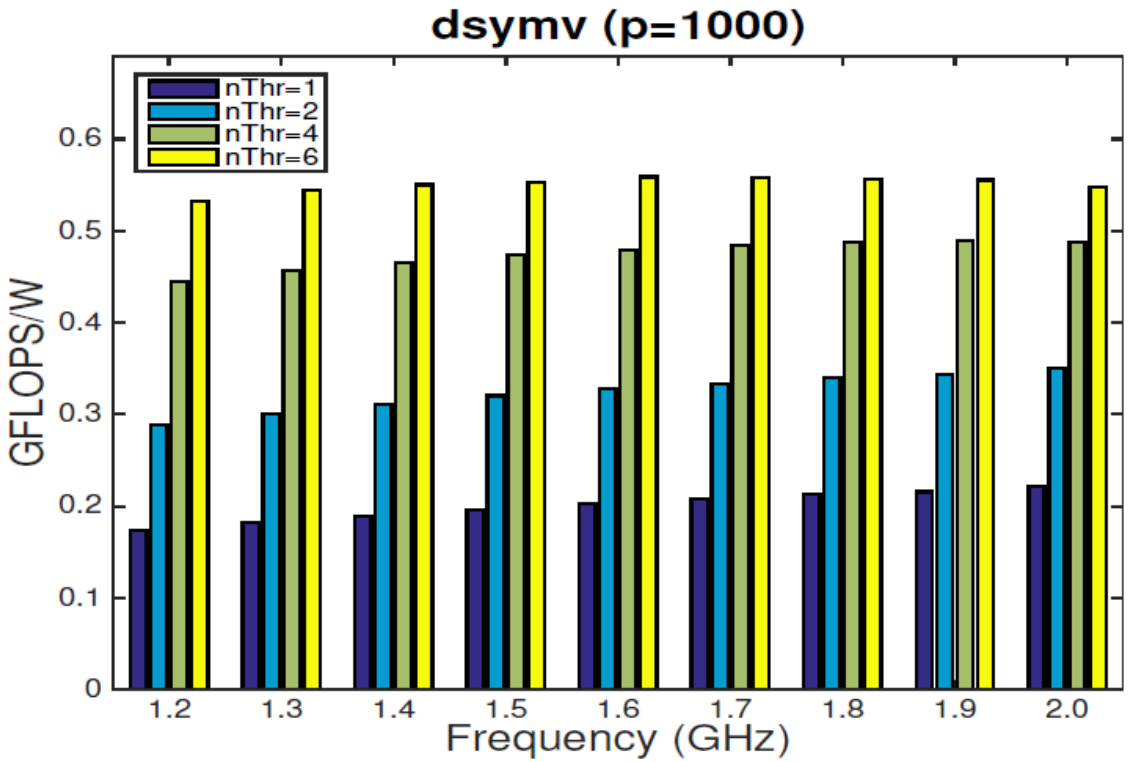
- Solution of eigenvalue problems is one of the cornerstones for scientific/engineering applications
- In many cases, the problem is dense and presents a symmetric structure

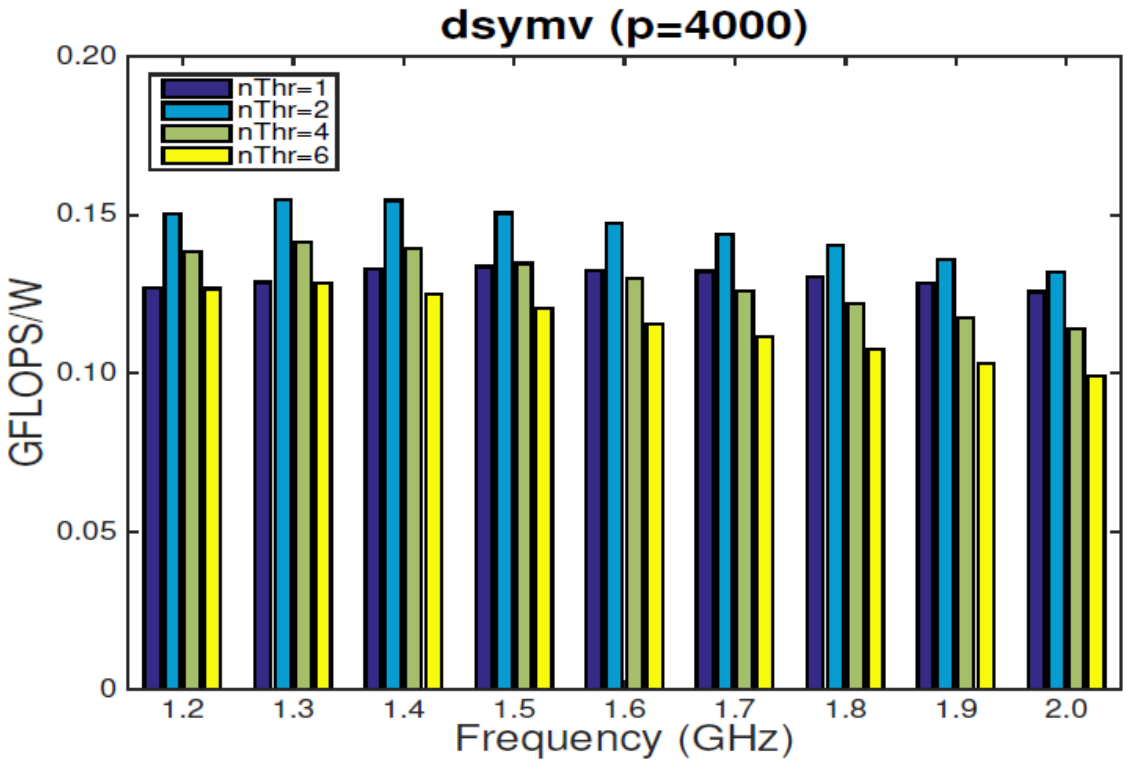
3. DCT

- Control the number of threads in execution
  - More threads does not necessarily mean faster
  - Even if slightly faster, (or at least not slower,) it may not be more energy efficient

3. DCT

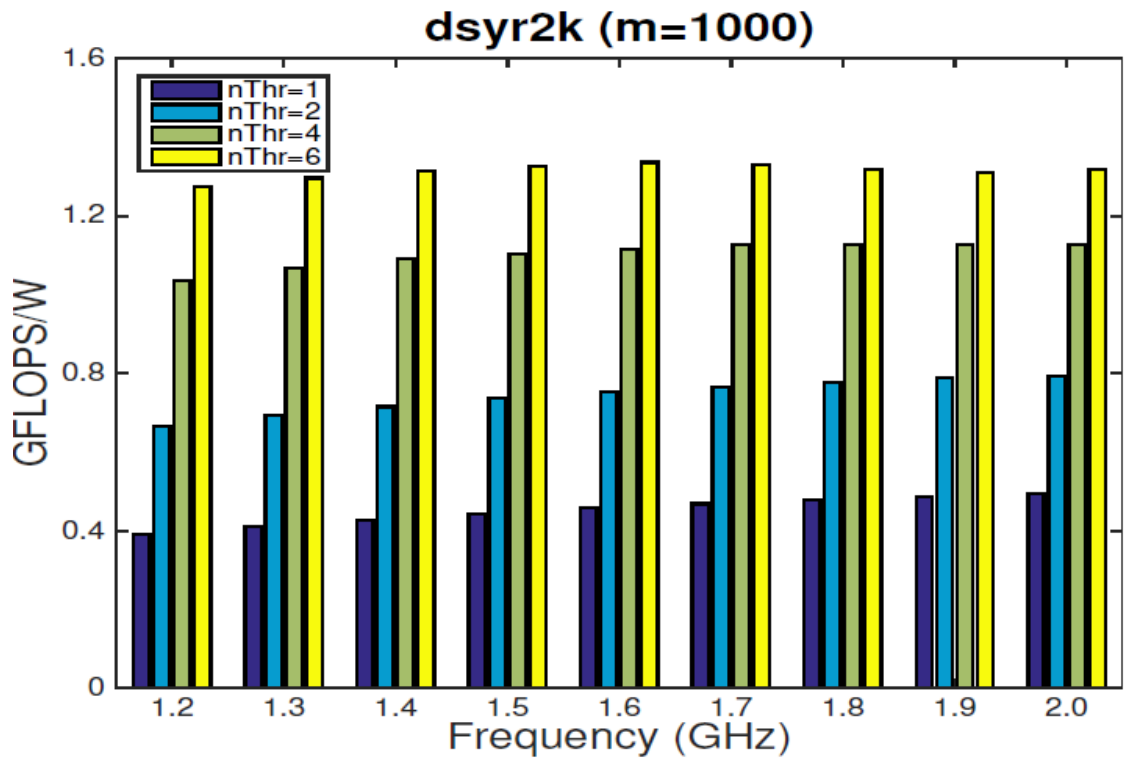
- Intel E5-2620. DSYMV:

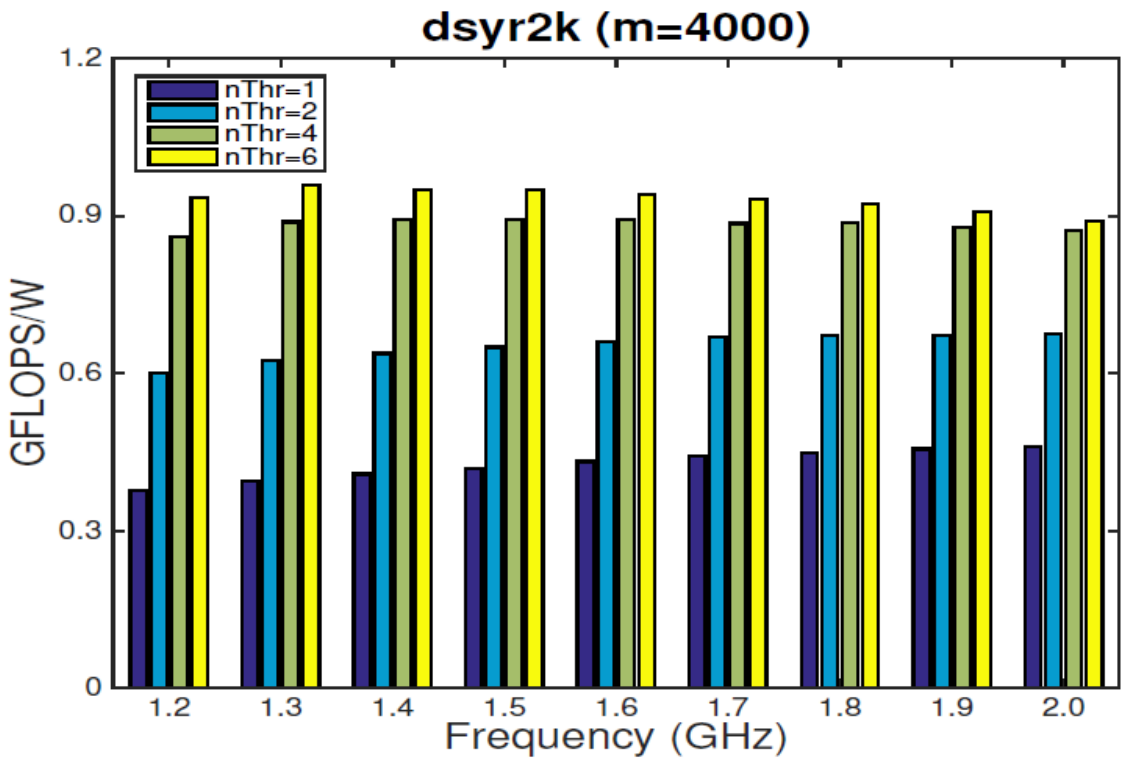




3. DCT

- Intel E5-2620. DSYR2K:





3. DCT

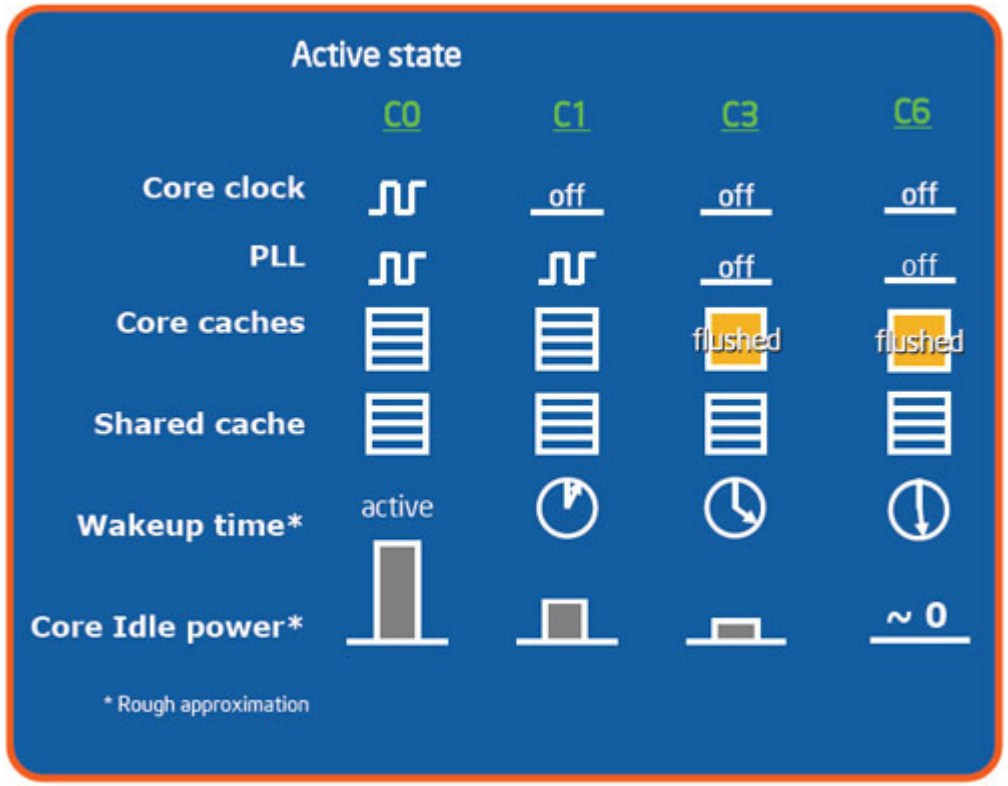
- Control with fine granularity may be necessary:
  - Reduction to tridiagonal from via _SYTRD (key for the solution of dense eigenvalue problems) spends half of its flops in _SYMV and the other half in _SYR2K
  - Subproblems become progressively smaller, till they fit into the cache

4. Avoid polling

- Do nothing well!
- Polling ensures a rapid reaction of CPU to status changes, but prevents it from entering energy-saving C-states
  - Wait for other tasks to complete (task-parallelism, synchronization)
  - CPU-GPU execution
  - MPI blocking routines

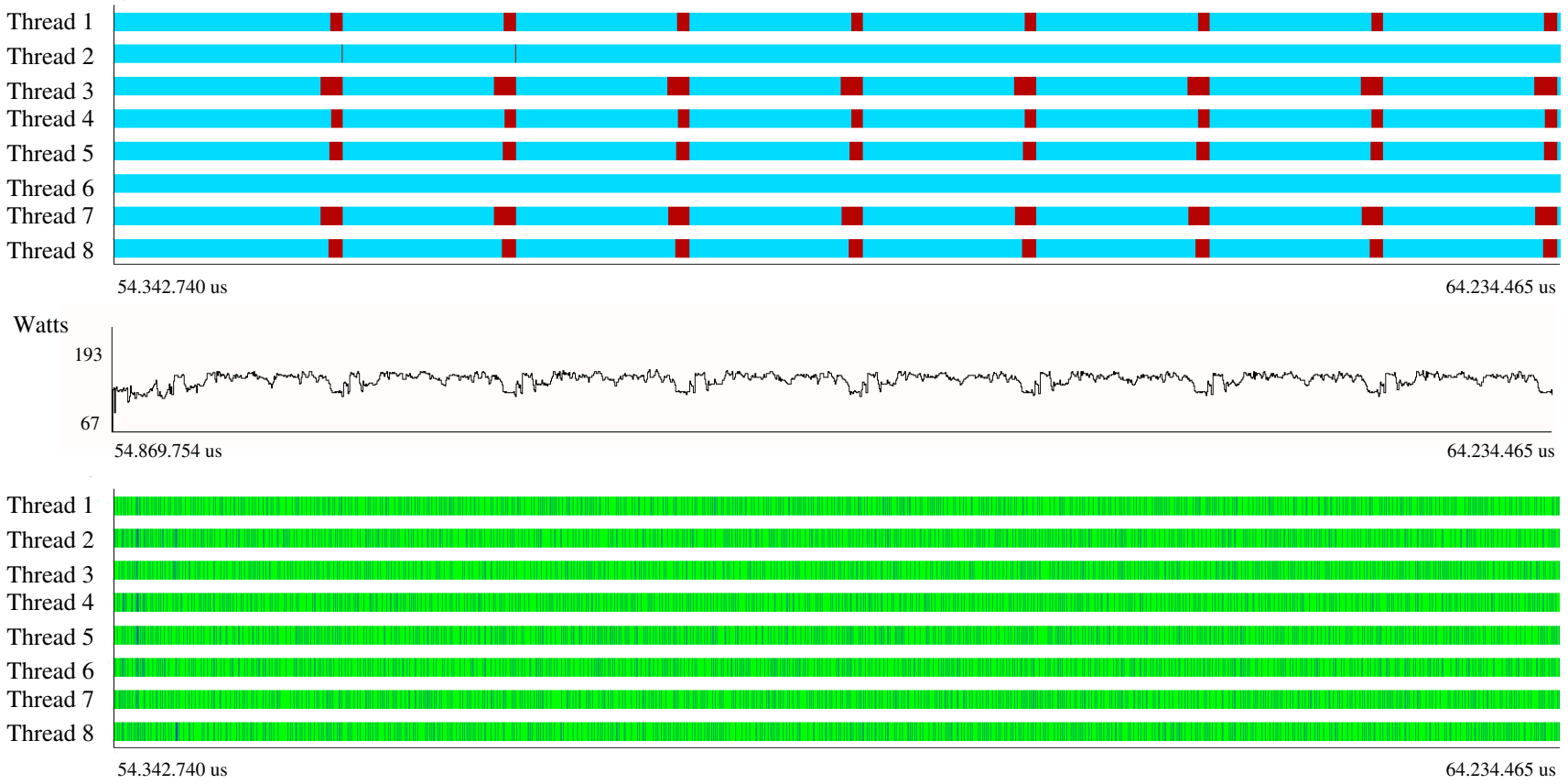
4. Avoid polling

- C-states (Core i7-Nehalem, similar for others)



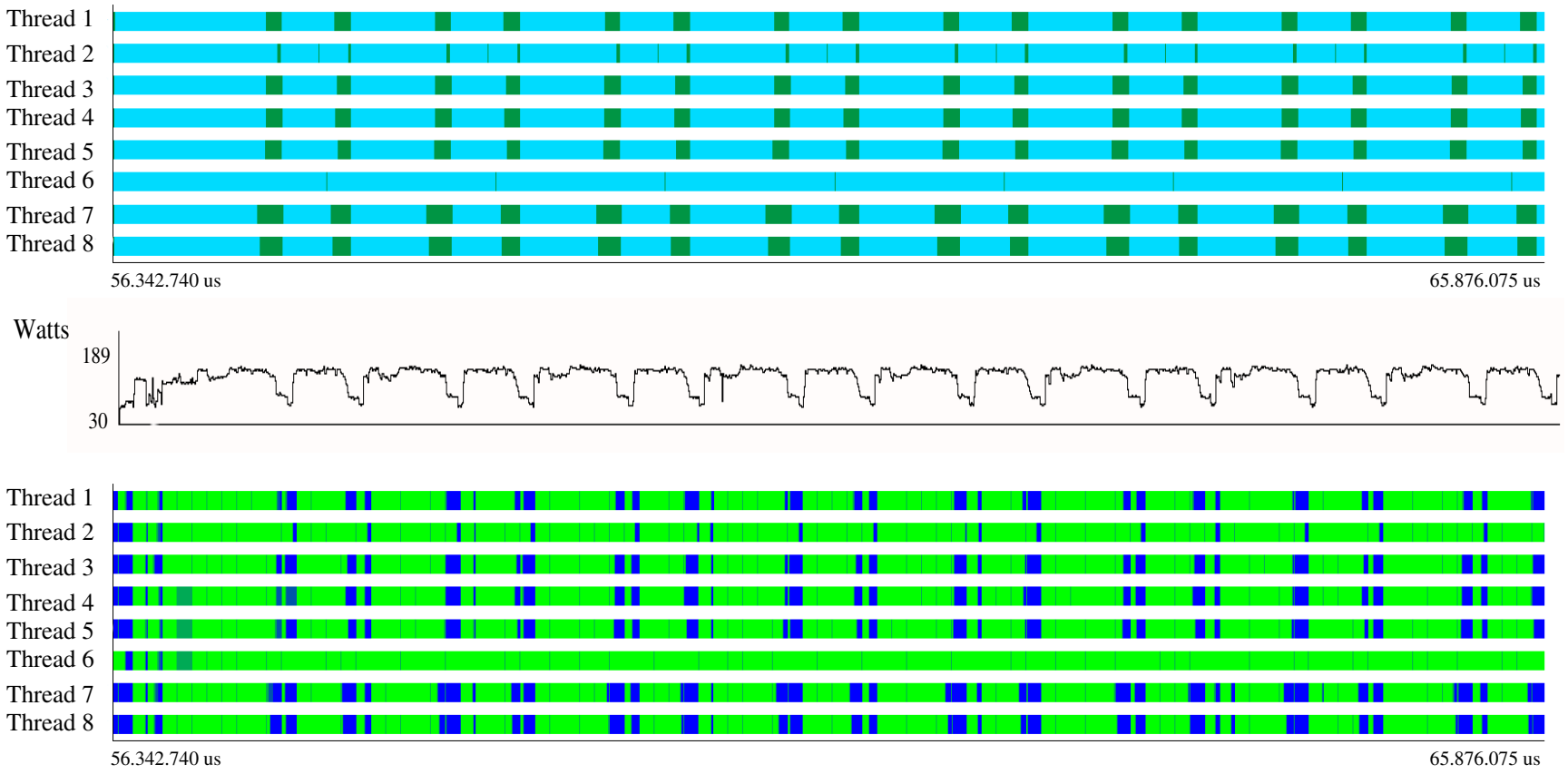
4. Avoid polling

- ILUPACK's PCG on Intel Xeon E5504 (2x4 cores)



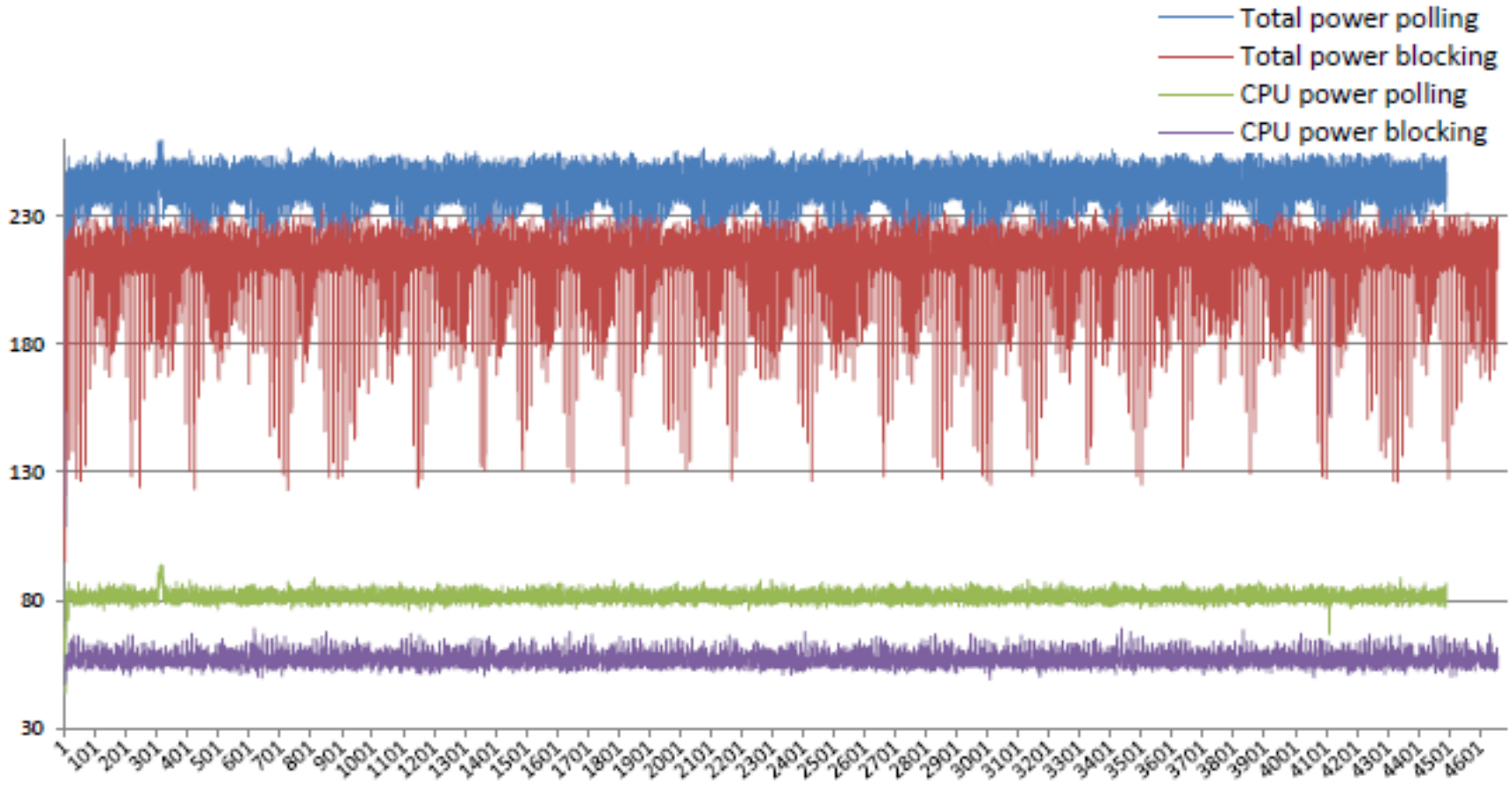
4. Avoid polling

- ILUPACK's PCG on Intel Xeon E5504 (2x4 cores)



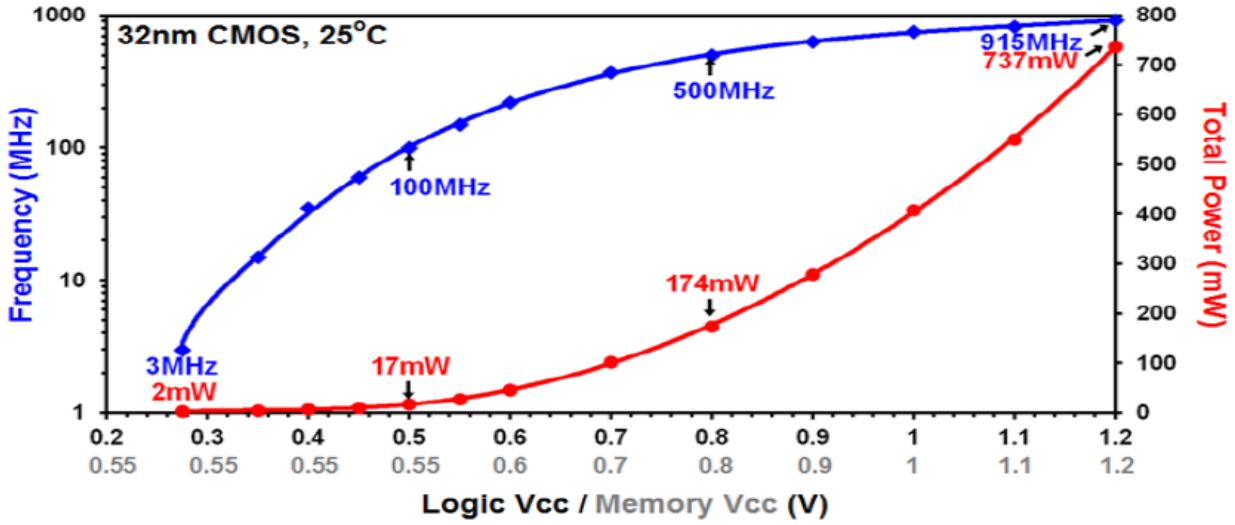
4. Avoid polling

- CG on GPU: Intel Core i7-3770K + NVIDIA GeForce GTX480



5. NTVC

- Dynamic power is proportional to *V^2 f*



- Undervolting: Reduce *V*, but maintain *f*
- NTVC: Reduce *(V,f)* in the same proportion
  - For some applications, reducing *f* does not impact performance
  - For others, a linear decrease in performance is expected

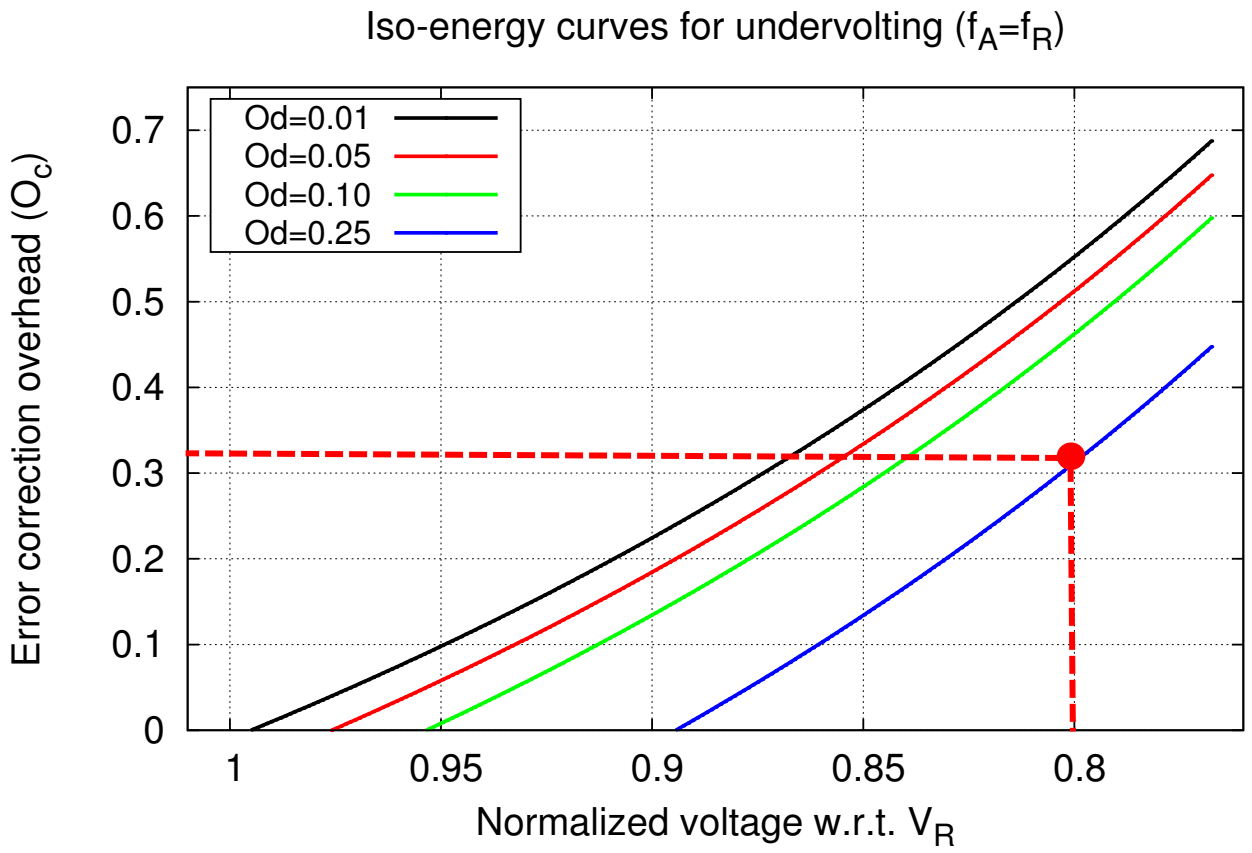
5. NTVC

- Operating near the voltage threshold may introduce errors
- Integrate fault-tolerance into software (applications)
  - Check-point + restart
  - Modular redundancy
  - Algorithmic-based fault tolerance (ABFT)
- What is the energy trade-off?
  - Move from error-free (V_R, f_R) → *error-prone* (V_A, f_A)
  - Detection overhead $\mathcal{O}_d$ (even if no errors occur)
  - Correction overhead $\mathcal{O}_c$ (proportional to error rate)

5. NTVC

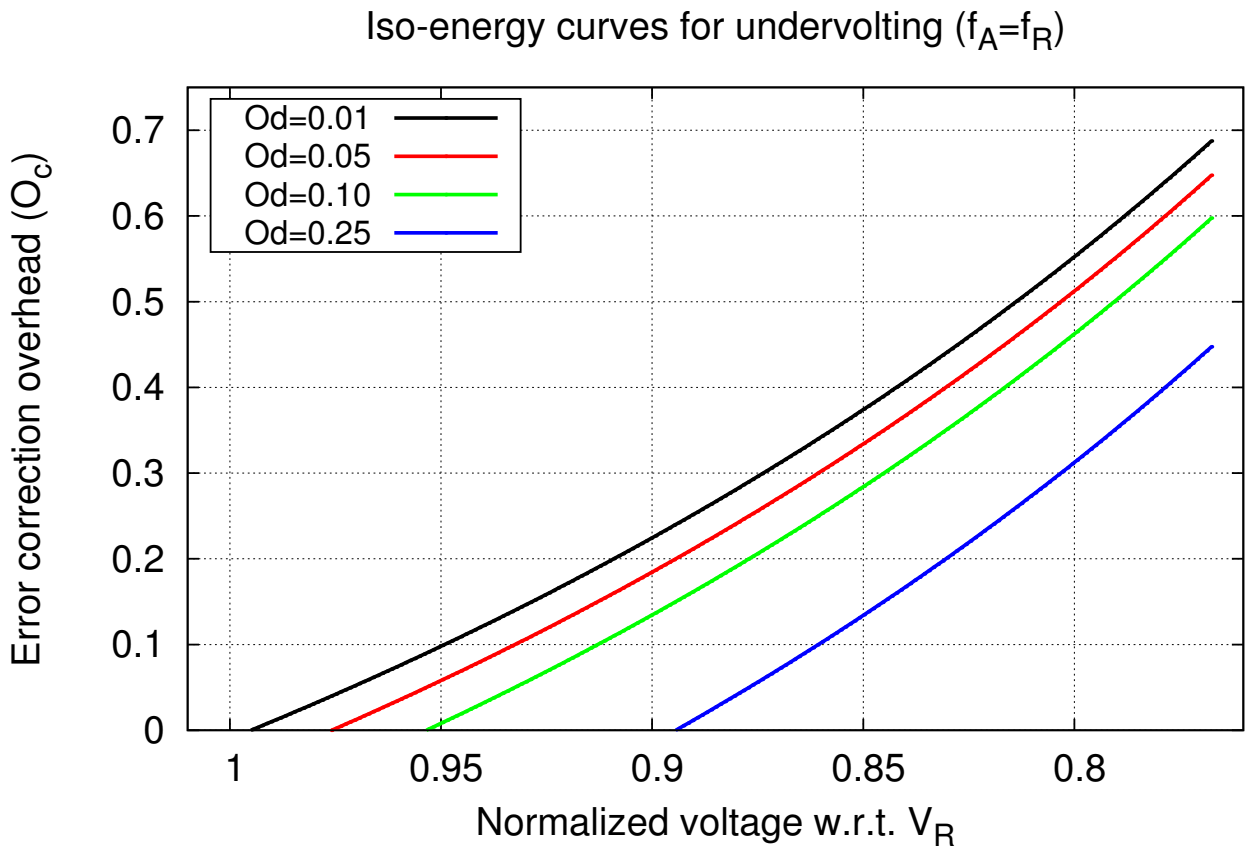
- For undervolting, any DLA and architecture:

$$\mathcal{O}_c^{iso} = \left(\frac{V_R}{V_A}\right)^2 - (1 + \mathcal{O}_d)$$



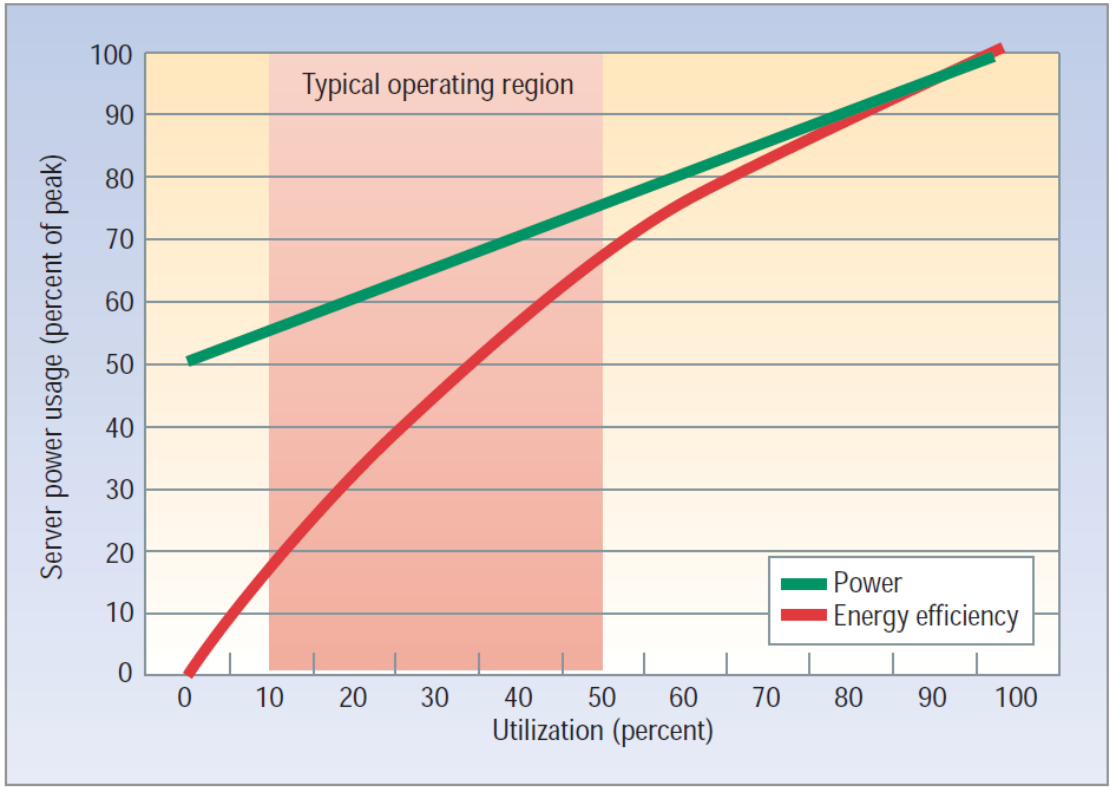
5. NTVC

- For NTVC, assuming linear relation between performance and *f*:
  - Same relation for compute-bound kernels
  - …but iso-energy more difficult for memory-bound kernels



6. Energy-proportional hardware

- Power consumption should be proportional to use of resources



"The case for energy-proportional computing". L. A. Barroso, U. Hölzle, IEEE Computer, 2007

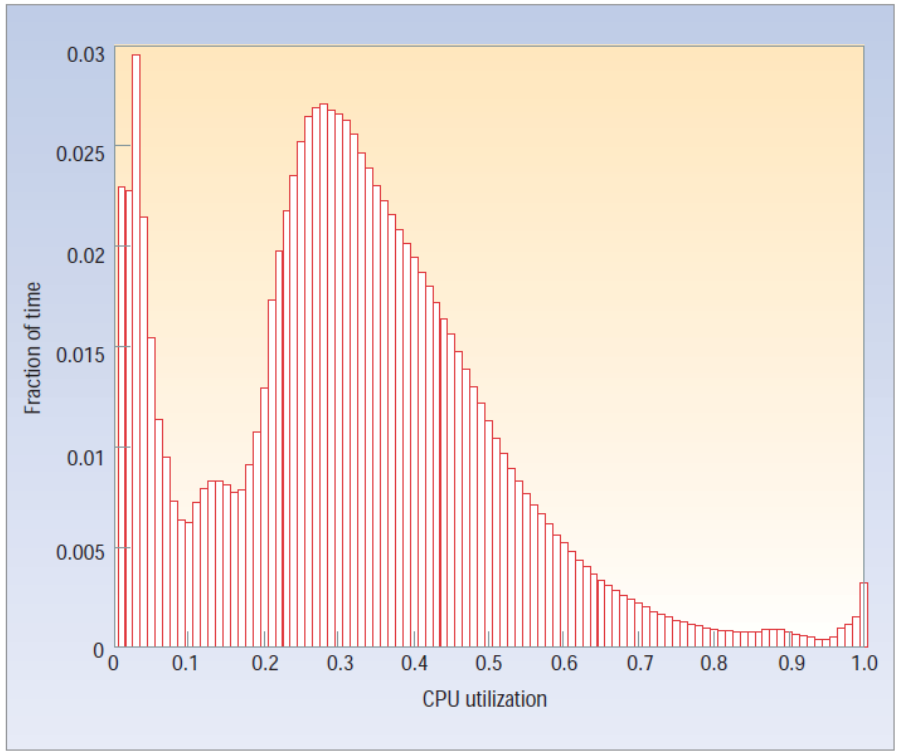
6. Energy-proportional hardware

- Stencil computation on Intel Xeon E5-2620 (2x6 cores)

#cores	Power (W)
1	135.1
2	144.1
3	150.2
4	155.9
5	162.8
6	169.4
7	171.4
8	180.0
9	186.0
10	188.4
11	190.6
12	195.1

7. Virtualization of HPC resources

- Servers seldom operate at 100% of their maximum utilization level

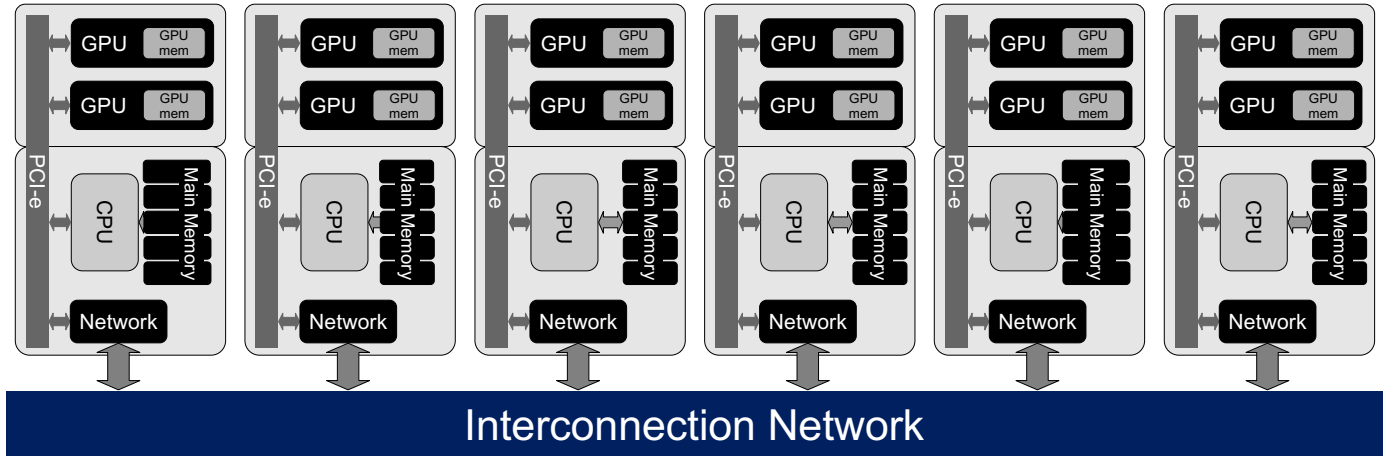


Average CPU utilization of more than 5,000 servers during a six-month period

"The case for energy-proportional computing". L. A. Barroso, U. Hölzle, IEEE Computer, 2007

7. Virtualization of HPC resources

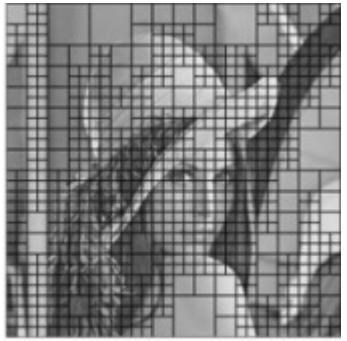
- Same for GPUs in a cluster:
  - Not all applications can run on a GPU
  - Not all parts of application's code benefit from a GPU
- Virtualization of accelerators



8. Approximate computing

- Some applications do not need a “fully accurate” answer:
  - Signal & video processing
  - Probabilistic inference
  - Service profiling
  - Monte Carlo simulation
  - Machine learning

(a)

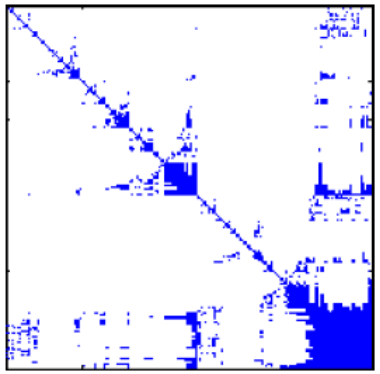


(b)

- Trade off accuracy for energy

Adaptive-precision algorithms for sparse linear algebra

- Numerical linear algebra for scientific computing?
  - Tiny errors (round-off) can rapidly “aggregate”
  - Double precision is the standard



- Can we work in reduced precision (most of the time), but still compute a full-precision solution?
  - Adaptive precision

Adaptive-precision algorithms for sparse linear algebra

- Preconditioned Conjugate Gradient (PCG)

$r_0 := b - Ax_0,\ z_0 := M^{-1}r_0,\ d_0 := z_0,\ \beta_0 := r_0^T z_0,$	Initializations
$\tau_0 :=\| r_0 \|_2,\ j := 0$	
while $(\tau_j > \tau_{\max})$	**Iterative PCG solve**
$w_j := Ad_j$	SPMV
$\rho_j := \beta_j / d_j^T w_j$	DOT product
$x_{j+1} := x_j + \rho_j d_j$	AXPY
$r_{j+1} := r_j - \rho_j w_j$	AXPY
$z_{j+1} := M^{-1} r_{j+1}$	Preconditioning
$\beta_{j+1} := r_{j+1}^T z_{j+1}$	DOT product
$\alpha_j := \beta_{j+1} / \beta_j$	
$d_{j+1} := z_{j+1} + \alpha_j d_j$	AXPY-like
$\tau_{j+1} :=\| r_{j+1} \|_2$	2-norm
$j := j + 1$	
endwhile	

Adaptive-precision algorithms for sparse linear algebra

- Customize precision:
  - CG is a memory-bounded algorithm: Cost comes from moving data, not arithmetic

Integer	
Add	
8 bit	0.03pJ
32 bit	0.1pJ
Mult	
8 bit	0.2pJ
32 bit	3 pJ

FP	
FAdd	
16 bit	0.4pJ
32 bit	0.9pJ
FMult	
16 bit	1pJ
32 bit	4pJ

Memory	
Cache	(64bit)
8KB	10pJ
32KB	20pJ
1MB	100pJ
DRAM	1.3-2.6nJ

"Computing's energy problem". M. Horowitz, 2014

  - Storage for CG iteration variables in MPIR: matrix, recurrence vectors
  - Storage for block-Jacobi preconditioner

Adaptive-precision algorithms for sparse linear algebra

- Customize precision:
  - CG is a memory-bounded algorithm: Cost comes from moving data, not arithmetic

Integer	
Add	
8 bit	0.03pJ
32 bit	0.1pJ
Mult	
8 bit	0.2pJ
32 bit	3 pJ

FP	
FAdd	
16 bit	0.4pJ
32 bit	0.9pJ
FMult	
16 bit	1pJ
32 bit	4pJ

Memory	
Cache	(64bit)
8KB	10pJ
32KB	20pJ
1MB	100pJ
DRAM	1.3-2.6nJ

"Computing's energy problem". M. Horowitz, 2014

  - **Storage for CG iteration variables in MPIR: matrix, recurrence vectors**
  - Storage for block-Jacobi preconditioner

Mixed precision iterative refinement

- Iterative refinement (IR) is a technique to improve the accuracy of an initial solution x_0:

$$
\begin{aligned}
&\textbf{for } k := 0, 1, 2, \ldots \\
&\qquad r := \hat{b} - A\hat{x}_k && \text{Residual calculation} \\
&\text{Solve } Ay = r \text{ for } y && \text{Inner solver} \\
&\qquad \hat{x}_{k+1} := \hat{x}_k + y && \text{Solution update}
\end{aligned}
$$

- Any inner solver: dense/sparse factorization…
  even an iterative Krylov(-type) solver
- In machine precision u, provided $uk(A) \leq 1$, IR eventually produces an accurate solution to full precision u

Mixed precision iterative refinement

- On many architectures, IR can be efficiently combined with a mixed precision (single-double, half-double, half-single) scheme

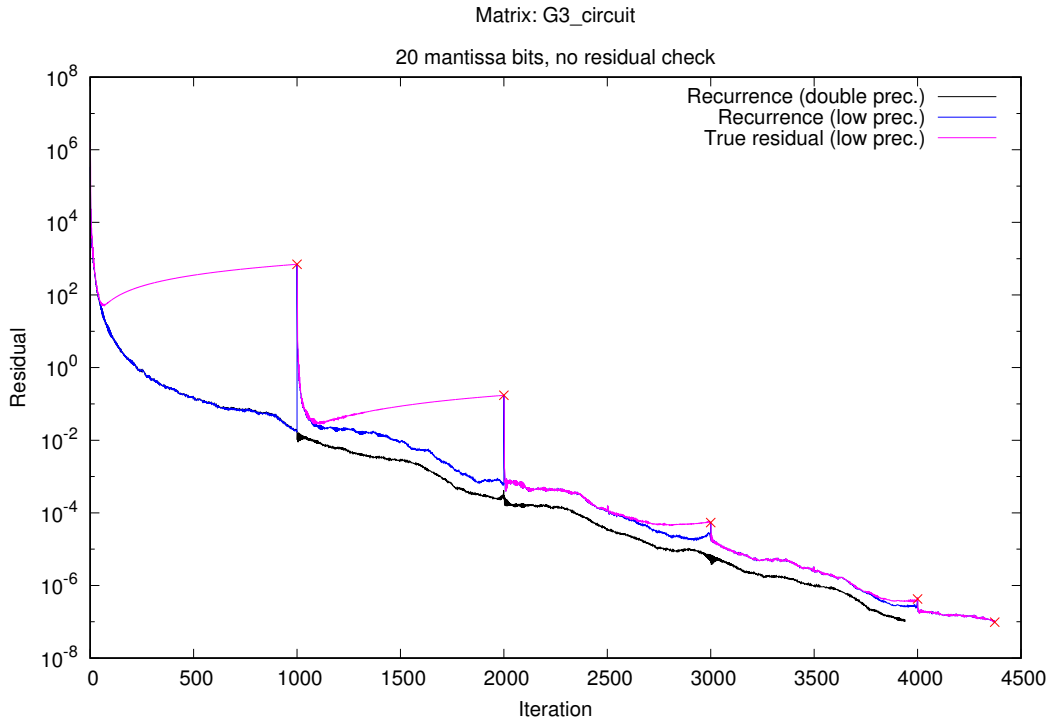
for $k := 0, 1, 2, \ldots$

$r := \hat{b} - A\hat{x}_k$	Residual calculation	Extended precision
Solve $Ay = r$ for y	Inner solver	Reduced precision
$\hat{x}_{k+1} := \hat{x}_k + y$	Solution update	Extended precision

- Most of the cost is in the inner solver
- Accuracy is improved by the outer refinement process

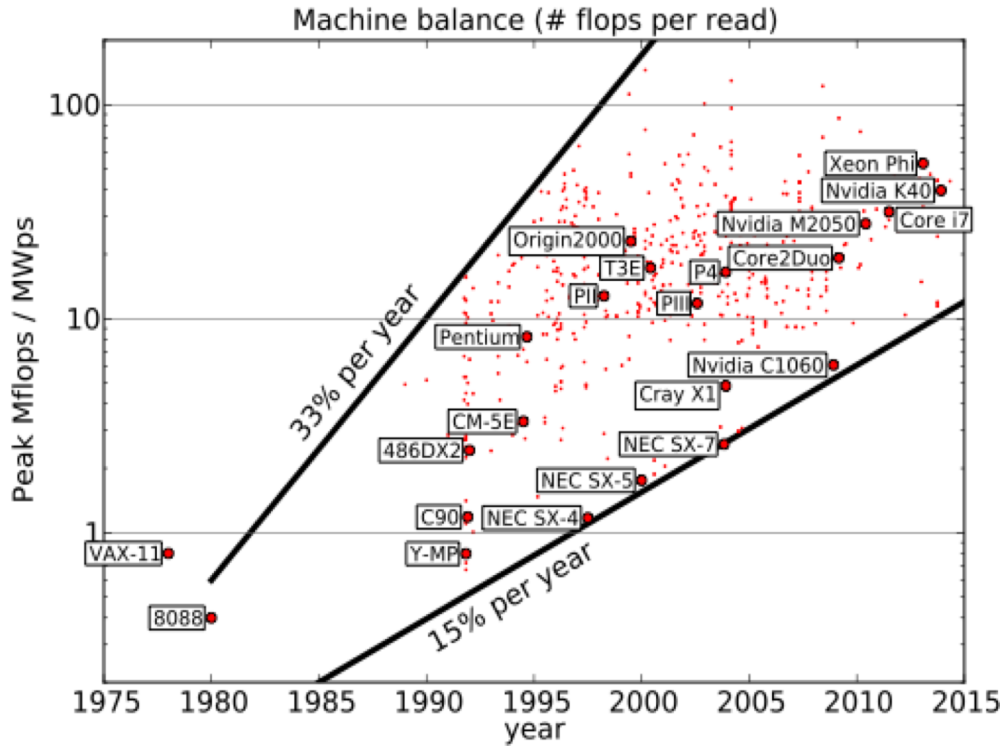
Mixed precision iterative refinement

- MPIR be **efficiently** combined with an iterative Krylov inner solver
  - Maintain **convergence rate** by avoiding numerical pitfalls in the recurrence residual due to finite precision



Mixed precision iterative refinement

- For Krylov solvers applied to sparse linear systems, **the theoretical cost/energy/time is in moving data, not in arithmetic**



Mixed precision iterative refinement
Outline

- Residual replacement for Krylov solvers
- Cost for Krylov solvers
- Modular precision format

Residual replacement for Krylov solvers

- PCG

$r_0 := b - Ax_0,\ z_0 := M^{-1}r_0,\ d_0 := z_0,\ \beta_0 := r_0^T z_0,$	Initializations
$\tau_0 :=\| r_0 \|_2,\ j := 0$	
while $(\tau_j > \tau_{\max})$	**Iterative PCG solve**
$w_j := Ad_j$	SPMV
$\rho_j := \beta_j / d_j^T w_j$	DOT product
$x_{j+1} := x_j + \rho_j d_j$	AXPY
$r_{j+1} := r_j - \rho_j w_j$	AXPY
$z_{j+1} := M^{-1} r_{j+1}$	Preconditioning
$\beta_{j+1} := r_{j+1}^T z_{j+1}$	DOT product
$\alpha_j := \beta_{j+1} / \beta_j$	
$d_{j+1} := z_{j+1} + \alpha_j d_j$	AXPY-like
$\tau_{j+1} :=\| r_{j+1} \|_2$	2-norm
$j := j + 1$	
endwhile	

Residual replacement for Krylov solvers

- PCG

$\tau_0 := \| r_0 \|_2$, $j := 0$
while $(\tau_j > \tau_{\max})$

$x_{j+1} := x_j + \rho_j d_j$
$r_{j+1} := r_j - \rho_j w_j$ Recurrence residual

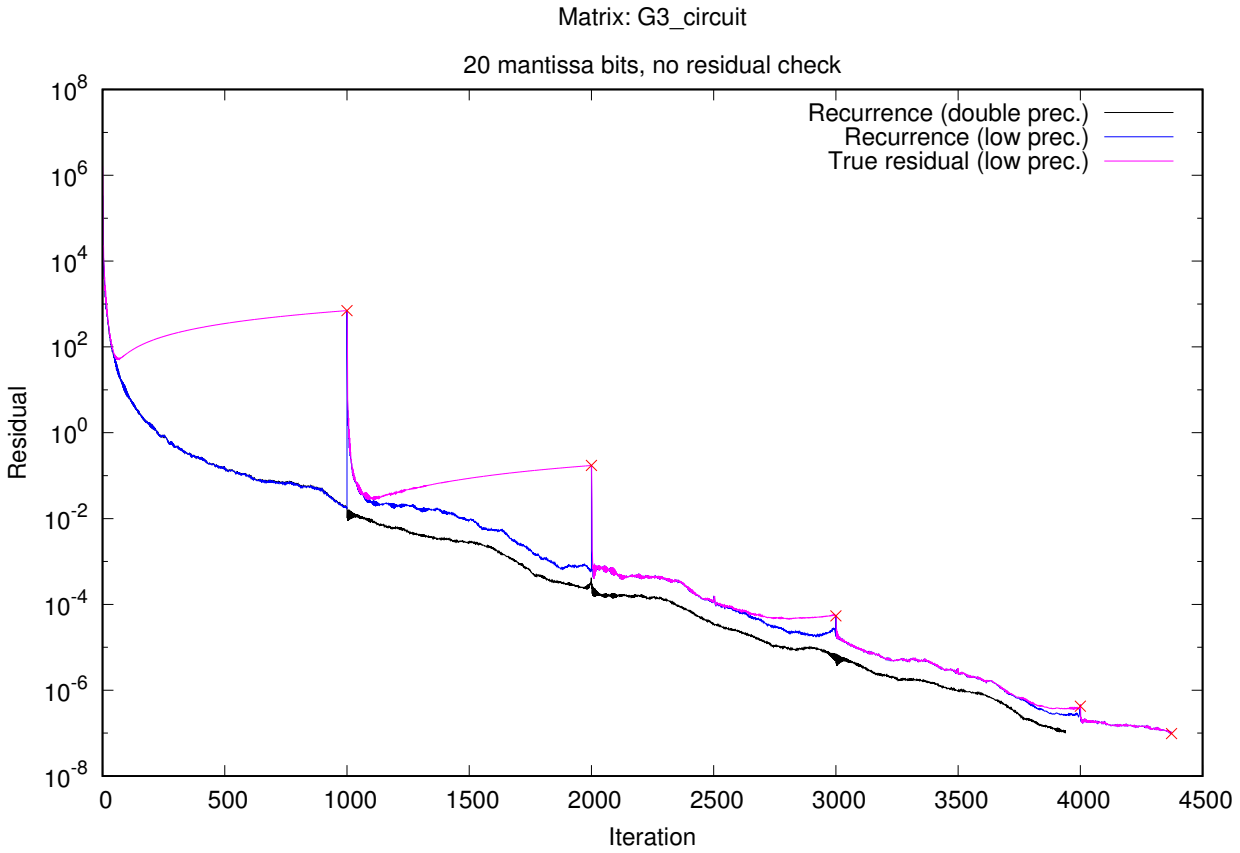
$\tau_{j+1} := \| r_{j+1} \|_2$
$j := j + 1$
endwhile

True residual

$$b - Ax_{j+1}$$

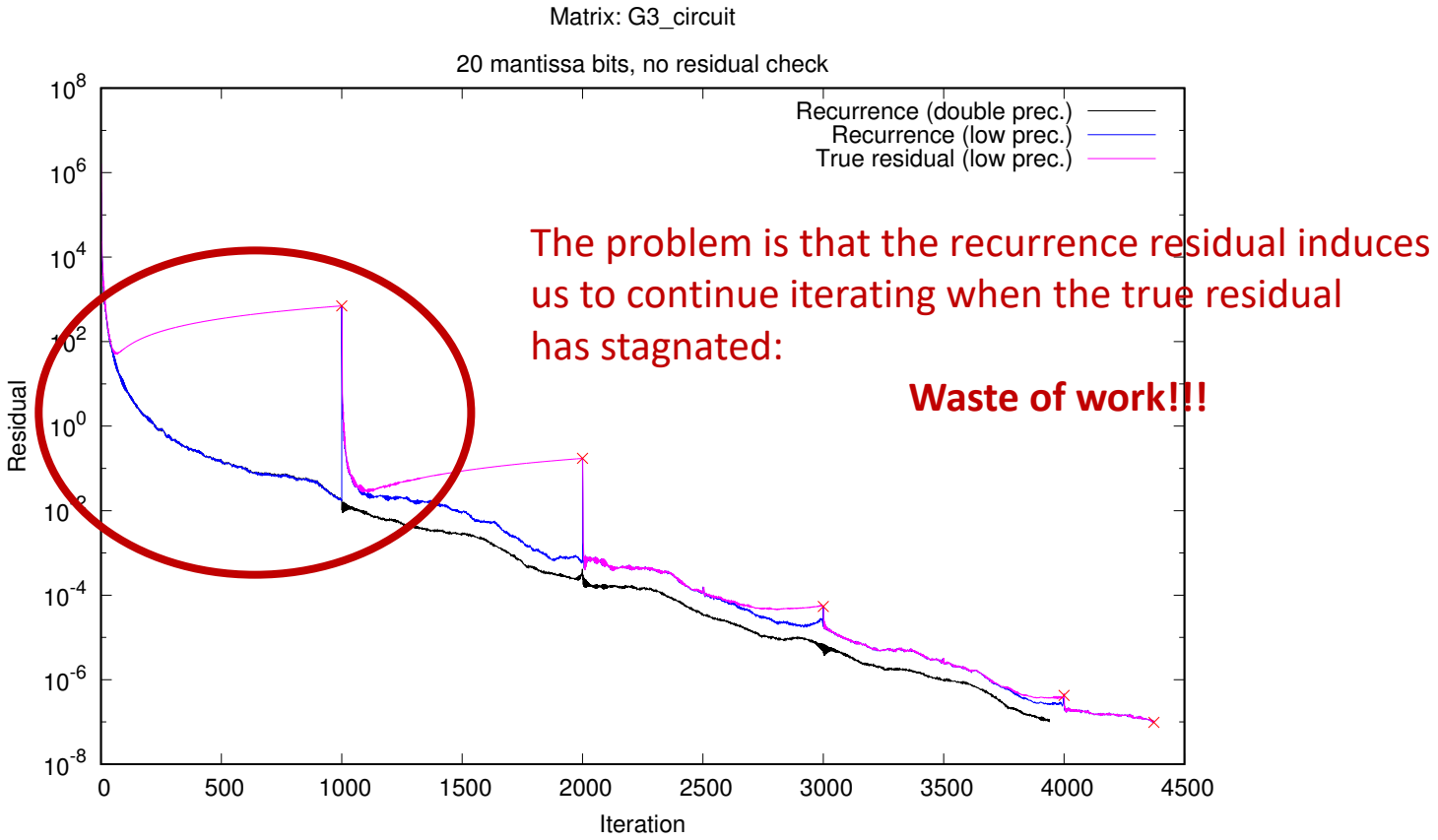
Residual replacement for Krylov solvers

- Finite precision causes divergence between recurrence vs true residuals



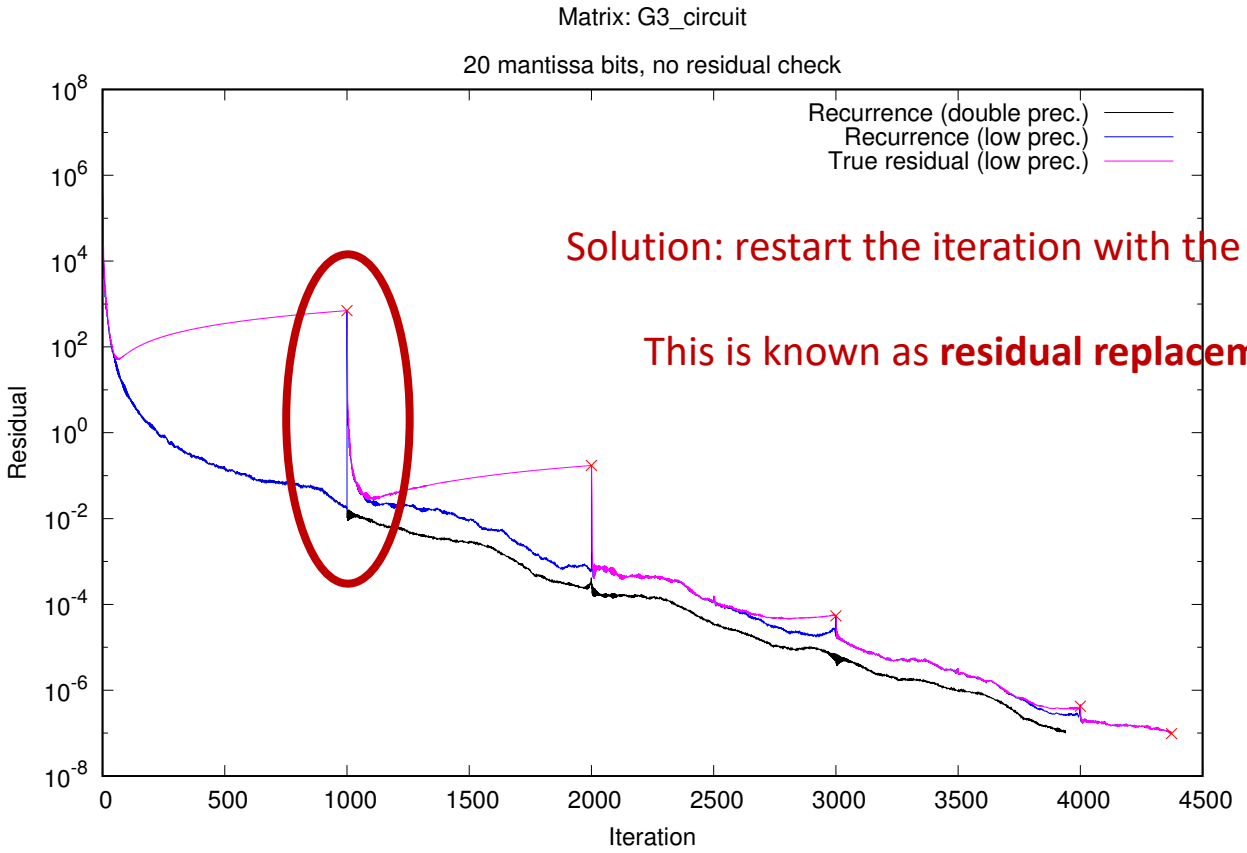
Residual replacement for Krylov solvers

- Finite precision causes divergence between recurrence vs true residuals



Residual replacement for Krylov solvers

- Finite precision causes divergence between recurrence vs true residuals



Solution: restart the iteration with the true residual

This is known as **residual replacement (RR)**

Residual replacement for Krylov solvers

- Divergence problem can be tackled via RR:
  - **Replace always** (at every iteration): doubles the cost per iteration and may deteriorate the convergence of the iteration
  - **Replace periodically** (every t iterations): may deteriorate the convergence of the iteration
  - **Compute explicit deviation at every iteration and replace if needed:** doubles the cost per iteration
  - **Estimate deviation and replace if needed**

H. A. Van der Vorst, Q. Ye. "Residual replacement strategies for Krylov subspace iterative methods for the convergence of true residuals." SIAM J. Sci. Comput., 22(3), 2000

Residual replacement for Krylov solvers

H. A. Van der Vorst, Q. Ye. “Residual replacement strategies for Krylov subspace iterative methods for the convergence of true residuals.” SIAM J. Sci. Comput., 22(3), 2000

- **Van der Vorst and Ye** (VY), 2000. Keep track of accumulated deviation:

$$d_0 = d_{init} := u_r(||r_0|| + N||A||||x_0||),$$
$$d_{j+1} := d_j + u_r(||r_j|| + N||A||||\tilde{x}_j||), \quad j = 0, 1, 2, \ldots,$$

  - Then, perform RR if the following three conditions hold

$$d_j \leq \epsilon||r_j||, \qquad d_{j+1} > \epsilon||r_{j+1}||, \qquad d_{j+1} \geq 1.1\, d_{init}$$

- Compared with others, VY’s RR technique:
  - Preserves convergence mechanism of the iteration
  - Ensures sufficiently small deviations between recurrence/true residuals
  - It is cheap and easy to add to existing Krylov implementations

Residual replacement for Krylov solvers

H. Anzt et al. “Residual replacement in mixed precision iterative refinement for sparse linear systems.” 1st ATCET Workshop, 2018

- **Explicit residual deviation** (ERD): Test periodically (i.e., every t iterations)

$$\|r_{k+1}^{true}\|_2 / \|r_{k+1}^{rec}\|_2 \geq \tau$$

  - Computing the residual explicitly is expensive (SpMV), but it can be done **in reduced precision**
  - Cost can be further reduced by performing the residual calculation together with SpMV for inner solver

- If deviation exceeds the threshold, stop the inner solver and start a new iteration of refinement (outer level) → enforces a residual replacement **in extended precision**

Cost for Krylov solvers

- Premises of cost model:
  - For a memory-bound algorithm, such as PCG applied to a sparse linear system, the "cost" is dominated by data movement while floating-point arithmetic is irrelevant
    - If cost = execution time, arithmetic cost is minor (memory wall) and can be overlapped with communication
    - If cost = energy, accesses to main memory are much more expensive than arithmetic

Integer	
Add	
8 bit	0.03pJ
32 bit	0.1pJ
Mult	
8 bit	0.2pJ
32 bit	3 pJ

FP	
FAdd	
16 bit	0.4pJ
32 bit	0.9pJ
FMult	
16 bit	1pJ
32 bit	4pJ

Memory	
Cache	(64bit)
8KB	10pJ
32KB	20pJ
1MB	100pJ
DRAM	1.3-2.6nJ

"Computing's energy problem". M. Horowitz, 2014

Cost for Krylov solvers

- Premises (cont'd):
  - After each particular operation, data does not remain in cache (reasonable if vectors are long enough)
  - Costs are linearly dependent on the bit-length of data
  - Problem of size n, with sparse matrix stored in CSR format consisting of n_z nonzero entries
  - Simple Jacobi preconditioner for CG

Cost for Krylov solvers

- Cost of sparse matrix-vector product (SpMV, CSR format) using data with xx bits in cost-units (cus) in terms of bit transfers:

$$\mathcal{C}_{\text{SPMV}}(\text{xx}) = \underbrace{(n+2n_z)\cdot \text{fpxx}}_{\text{Vector, matrix entries}} + \underbrace{(n+n_z)\cdot \text{int32}}_{\text{indices}} \quad \text{cus}$$

```
for (i=0; i<m; i++) {
  tmp = 0;
  for (j=row_ptr[i]; j<row_ptr[i+1]; j++)
    tmp += val[j] * x[col_idx[j]];
  y[i] = tmp;
}
```

- PCG solver operating with xx bits:

$$\mathcal{C}^{\text{iter}}_{\text{PCG}}(\text{xx}) = \underbrace{14n\cdot \text{fpxx}}_{\text{vector ops.}} + \mathcal{C}_{\text{SPMV}}(\text{xx}) + \underbrace{3n\cdot \text{fpxx}}_{\text{preconditioner appl.}} \quad \text{cus}$$

Cost for Krylov solvers

- For MPIR-VY, cost depends on:
  - #IS: number of iterations of inner solver
  - #RR: total number of residual replacements
  - #RS: number of iterative refinement steps
- For example, using (32,64) mixed precision:

$$\mathcal{C}^{\mathrm{VY}}_{\mathrm{MPIR}}(32{,}64) = \underbrace{\mathcal{C}^{\mathrm{iter}}_{\mathrm{PCG}}(32) \cdot \#\mathrm{IS}}_{\text{Plain inner PCG solver}} + \underbrace{n \cdot \mathsf{fp32} \cdot \#\mathrm{IS}}_{\text{Replacement condition test}}$$
$$+ \underbrace{(4n \cdot \mathsf{fp32} + \mathcal{C}_{\mathrm{SPMV}}(32)) \cdot \#\mathrm{RR}}_{\text{RRs in inner PCG solver}}$$
$$+ \underbrace{(6n \cdot \mathsf{fp64} + n \cdot \mathsf{fp32} + \mathcal{C}_{\mathrm{SPMV}}(64)) \cdot \#\mathrm{RS}}_{\text{Refinement steps}} \quad \text{cus}$$

Cost for Krylov solvers

- Cost of ERD:

$$\mathcal{C}^{\text{ERD}}_{\text{MPIR}}(32{,}64) = \underbrace{\mathcal{C}^{\text{iter}}_{\text{PCG}}(32) \cdot \#\text{IS}}_{\text{Plain inner PCG solver}} + \underbrace{(4n \cdot \text{fp32} + \mathcal{C}_{\text{SPMV}}(32)) \cdot \#\text{IS}/t}_{\text{Residual tests in inner PCG solver}} + \underbrace{(6n \cdot \text{fp64} + n \cdot \text{fp32} + \mathcal{C}_{\text{SPMV}}(64)) \cdot \#\text{RS}}_{\text{Refinement steps}} \quad \text{cus}$$

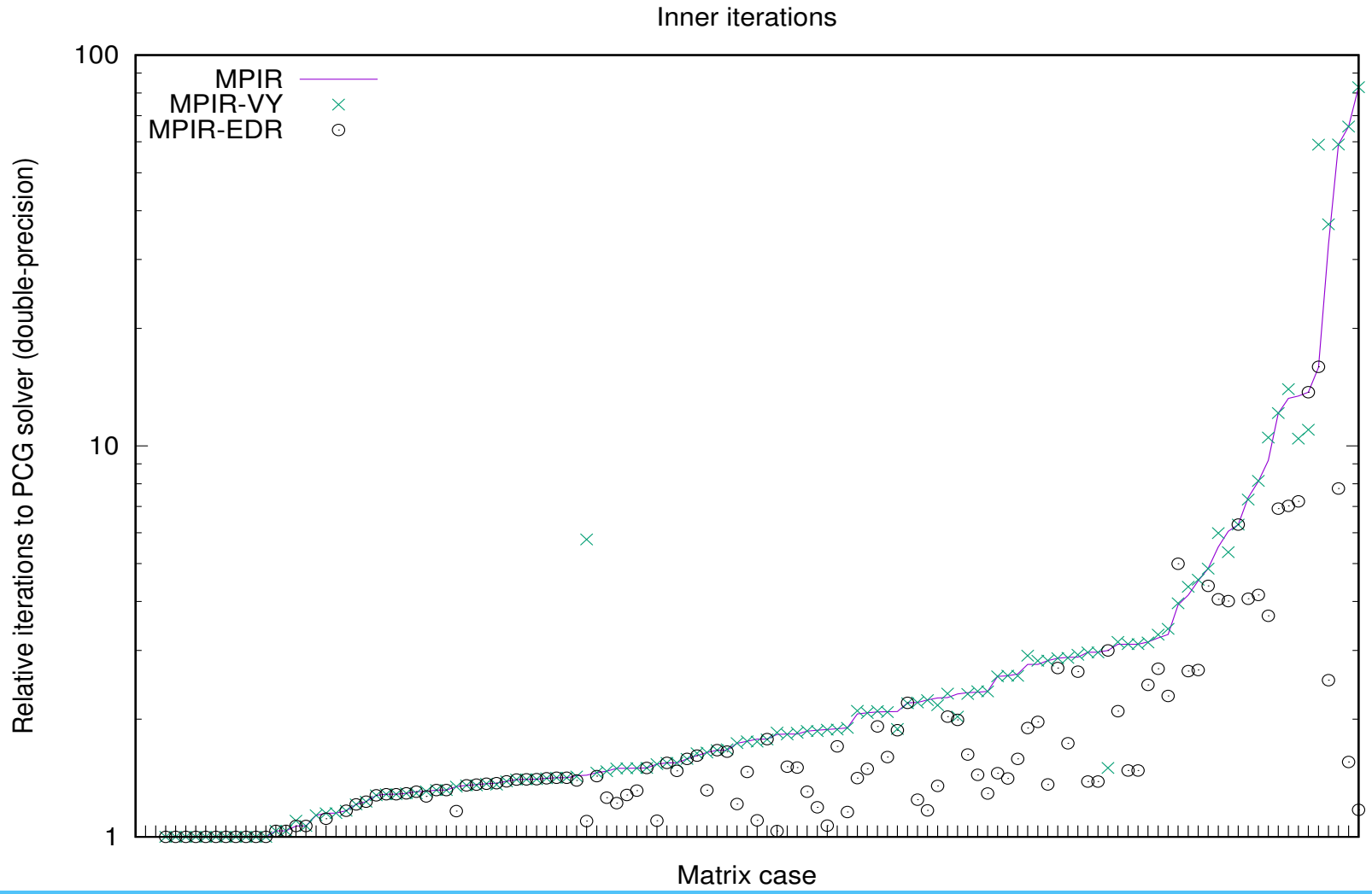
Cost for Krylov solvers

- VY vs EDR:
  - VR-RR incurs detection overhead at each iteration (test replacement condition) and pays correction overhead in case RR is necessary
  - EDR-RR incurs detection overhead only every t iterations (periodicity of the test), risking to waste work in case of stagnation from last test
  - Detection techniques are different and, therefore, also are numerical effects and overhead

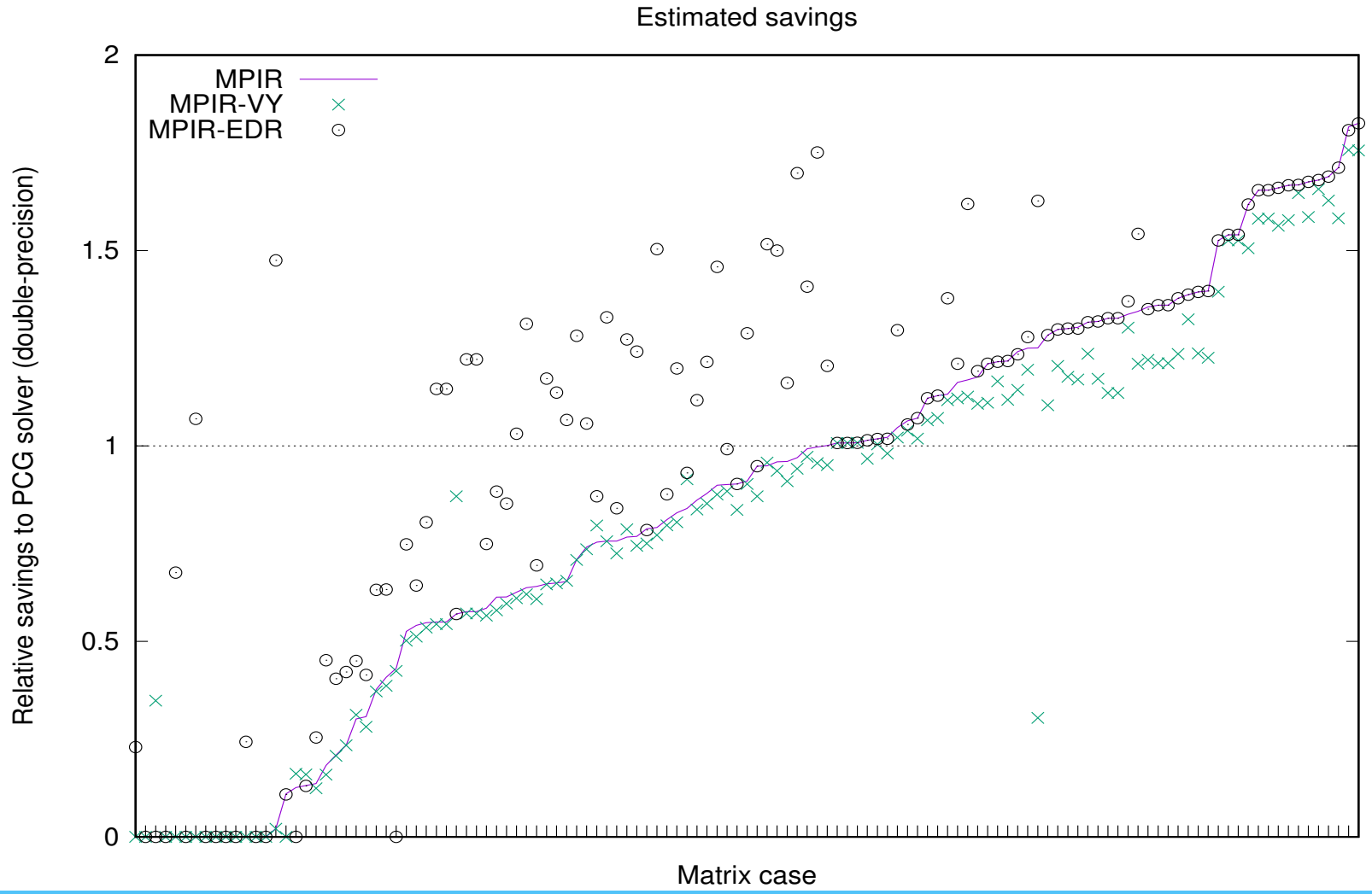
Cost for Krylov solvers

- Setup:
  - 123 symmetric positive definite matrices from SuiteSparse Matrix Collection (formerly UFMC)
  - Baseline solver: PCG in double precision
  - All arithmetic done in double precision
  - For MPIR variants, the coefficient matrix, the preconditioner and all iteration vectors used in the inner solver are stored in single precision: **reduced transfer cost!**
  - For EDR-RR, the test is performed every t = 100 iterations, and the maximum number of RR is set to 10
  - Cost take into account the actual number of iterations to obtain an absolute residual error below 10^{-7}

Cost for Krylov solvers



Cost for Krylov solvers



Modular precision format

- Decouple arithmetic from storage formats:
  - FPUs only support a limited number of IEEE 754 formats (single, double and, in some architectures, half)
  - … but we are free to store the data in memory in any **customized format**

- Remember: As a memory-bound algorithm, PCG is limited by memory bandwidth (i.e., how many bit are used to store the data)
  - Extended can be double
  - Reduced can be, e.g., 16, 24, 32, 40, 48, 56 bits
  - Maintain a single copy of the matrix with “multiple precisions” via **segments**

T. Grützmacher, H:. Anzt. “A modular precision format for decoupling arithmetic format and storage format. To appear in HeteroPar 2018

Adaptive-precision algorithms for sparse linear algebra

- Customize precision:
  - CG is a memory-bounded algorithm: Cost comes from moving data, not arithmetic

Integer	
Add	
8 bit	0.03pJ
32 bit	0.1pJ
Mult	
8 bit	0.2pJ
32 bit	3 pJ

FP	
FAdd	
16 bit	0.4pJ
32 bit	0.9pJ
FMult	
16 bit	1pJ
32 bit	4pJ

Memory	
Cache	(64bit)
8KB	10pJ
32KB	20pJ
1MB	100pJ
DRAM	1.3-2.6nJ

"Computing's energy problem". M. Horowitz, 2014

  - Storage for CG iteration variables in MPIR: matrix, recurrence vectors
  - **Storage for block-Jacobi preconditioner**

Block-Jacobi preconditioning

- **Jacobi method** based on **diagonal scaling:** $P = diag(A)$
  - Can be used as iterative solver:

$$x^{(k+1)} = x^{(k)} + P^{-1}b - P^{-1}Ax^{(k)}$$

  - Can be used as preconditioner: $\tilde{A} = P^{-1}A$, $\tilde{b} = P^{-1}b$

$$Ax = b \Leftrightarrow \tilde{A}x = \tilde{b}$$

Block-Jacobi preconditioning

- **Jacobi method** based on **diagonal scaling:** $P = diag(A)$
  - Can be used as iterative solver:

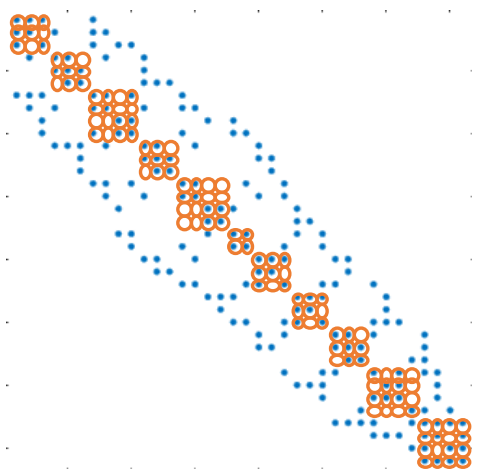
$$x^{(k+1)} = x^{(k)} + P^{-1}b - P^{-1}Ax^{(k)}$$

  - Can be used as preconditioner: $\tilde{A} = P^{-1}A \quad \tilde{b} = P^{-1}b$

$$Ax = b \Leftrightarrow \tilde{A}x = \tilde{b}$$

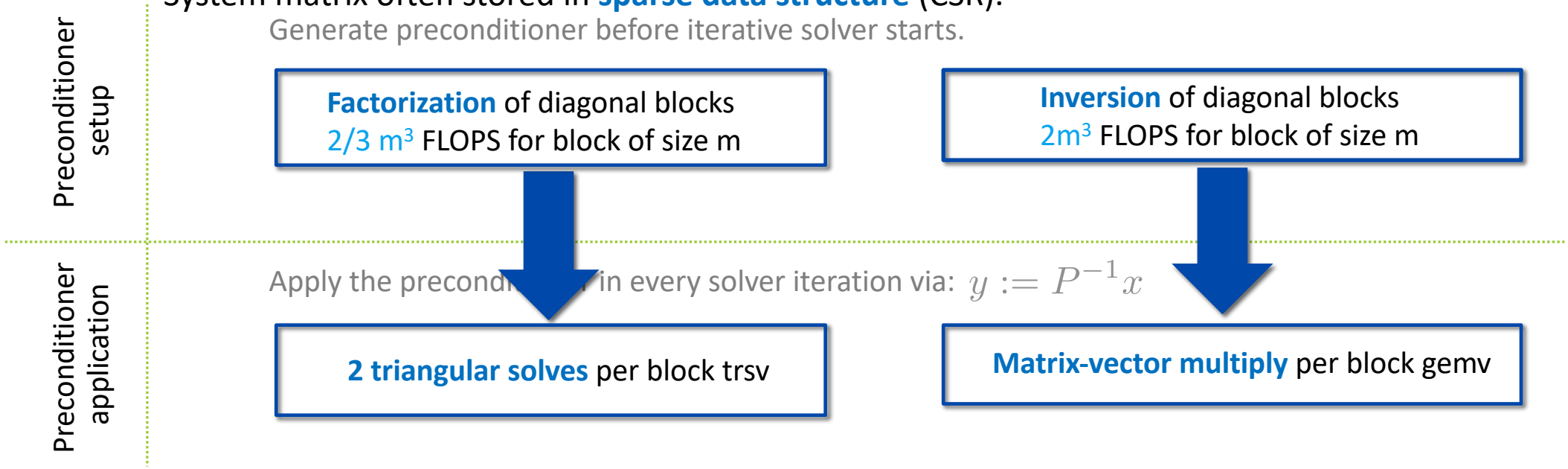
- **Block-Jacobi** is based on **block-diagonal scaling**:
  - Large set of small diagonal blocks.
  - Each block corresponds to one (small) linear system.
    - *Larger* blocks typically **improve convergence**.
    - *Larger* blocks make block-Jacobi **more expensive**.

      Extreme case: one block of matrix size.

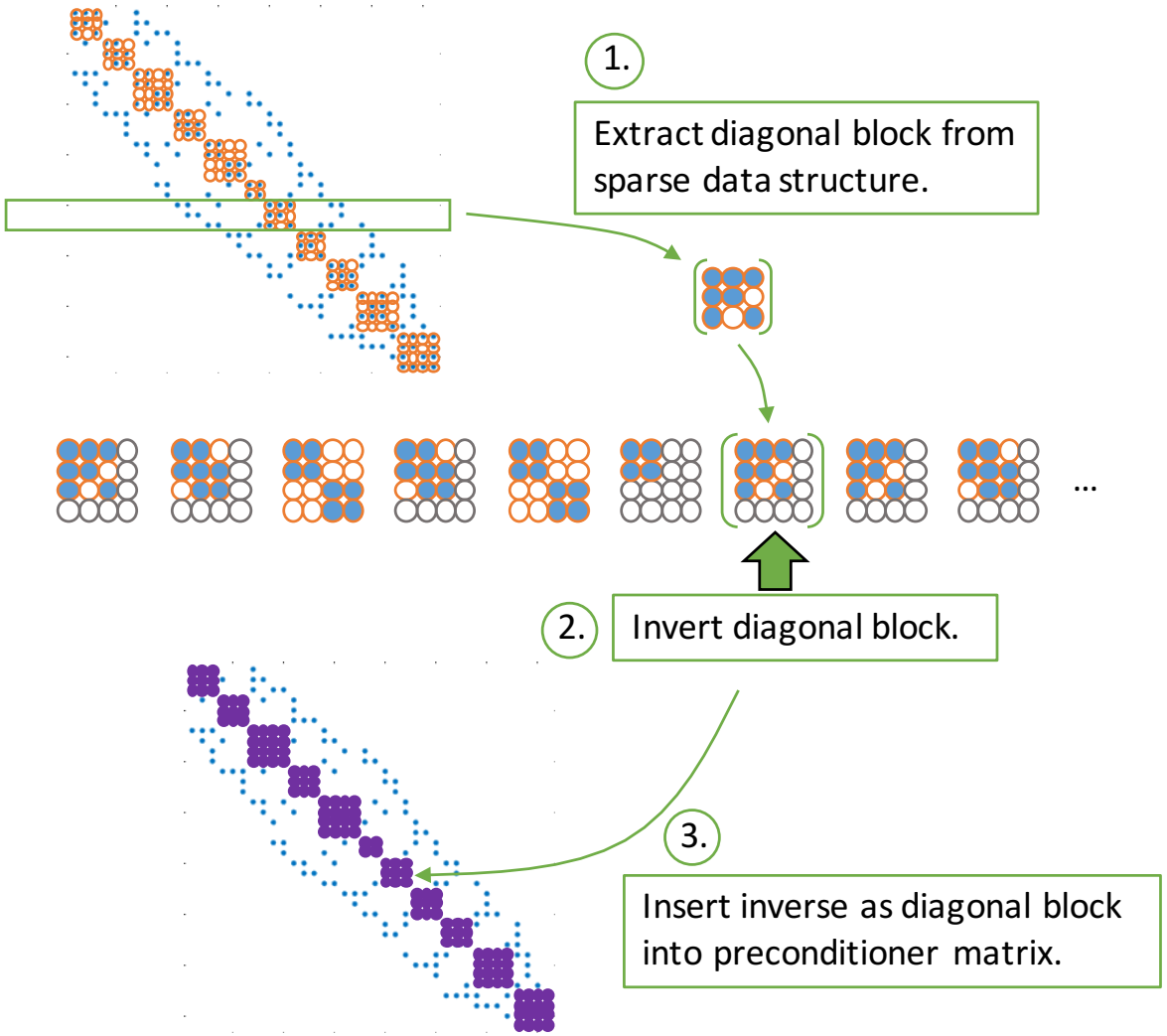


Block-Jacobi preconditioning

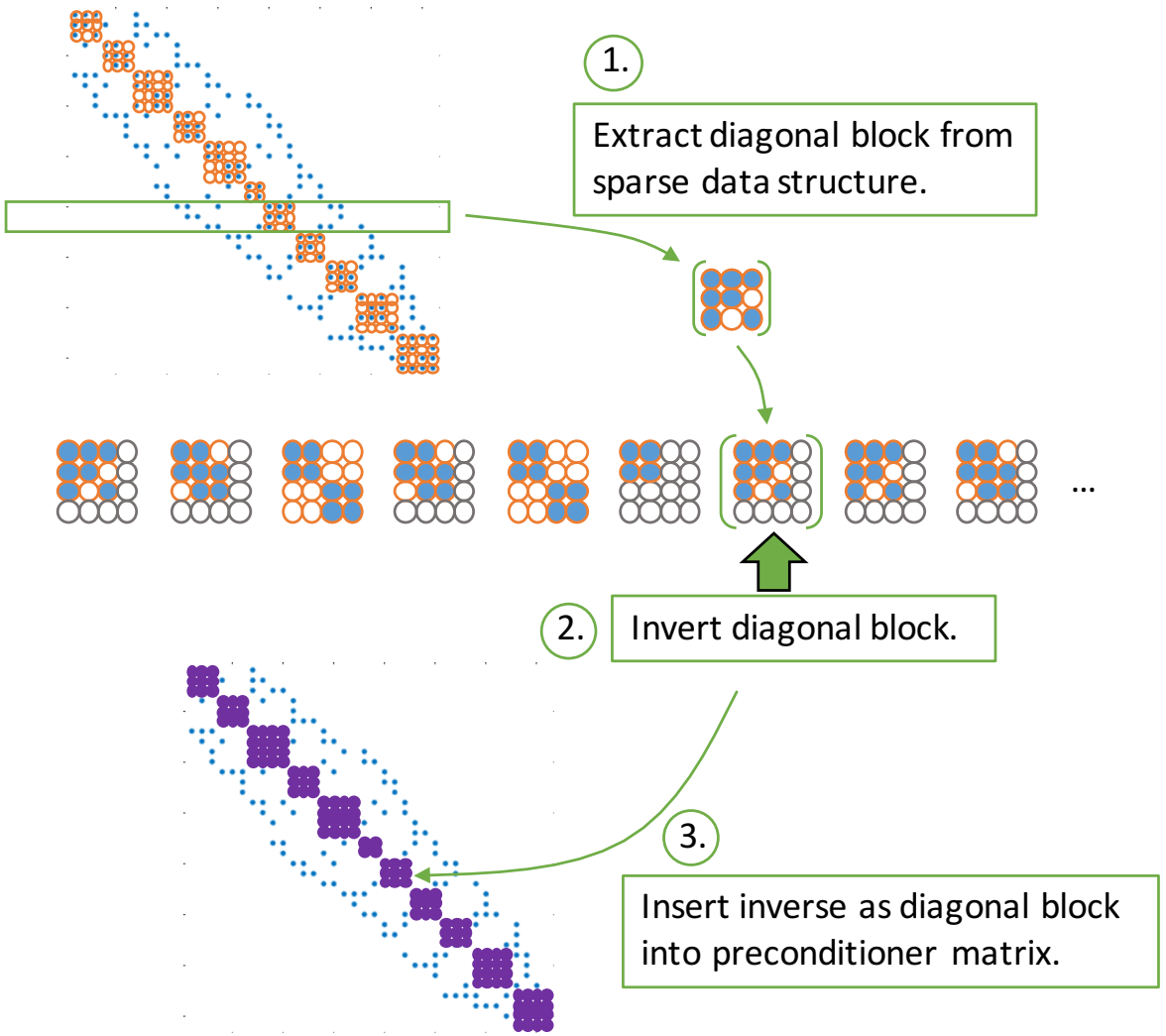
- **Block-Jacobi** method typically used as **preconditioner** inside Krylov solver.
- Target: large, sparse linear systems.
- **FEM** discretizations often carry a **block-structure** (multiple variables per node).
- "Natural blocks" of **small size** (8, 12,...).
- System matrix often stored in **sparse data structure** (CSR).



Block-Jacobi preconditioning



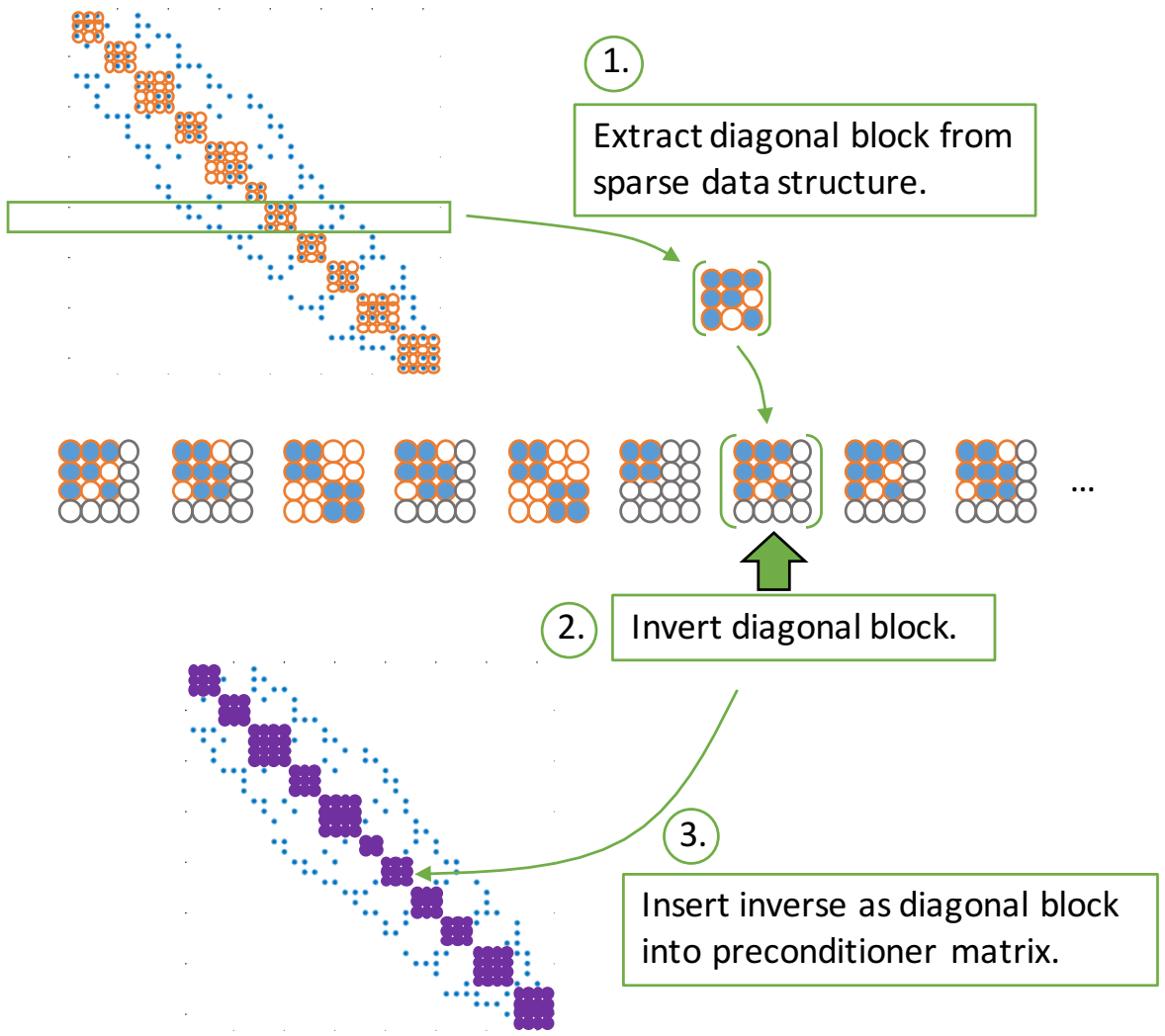
Block-Jacobi preconditioning



- Cost of **Inversion:**
  $2m_i^3$ FLOPs for block of size m_i.

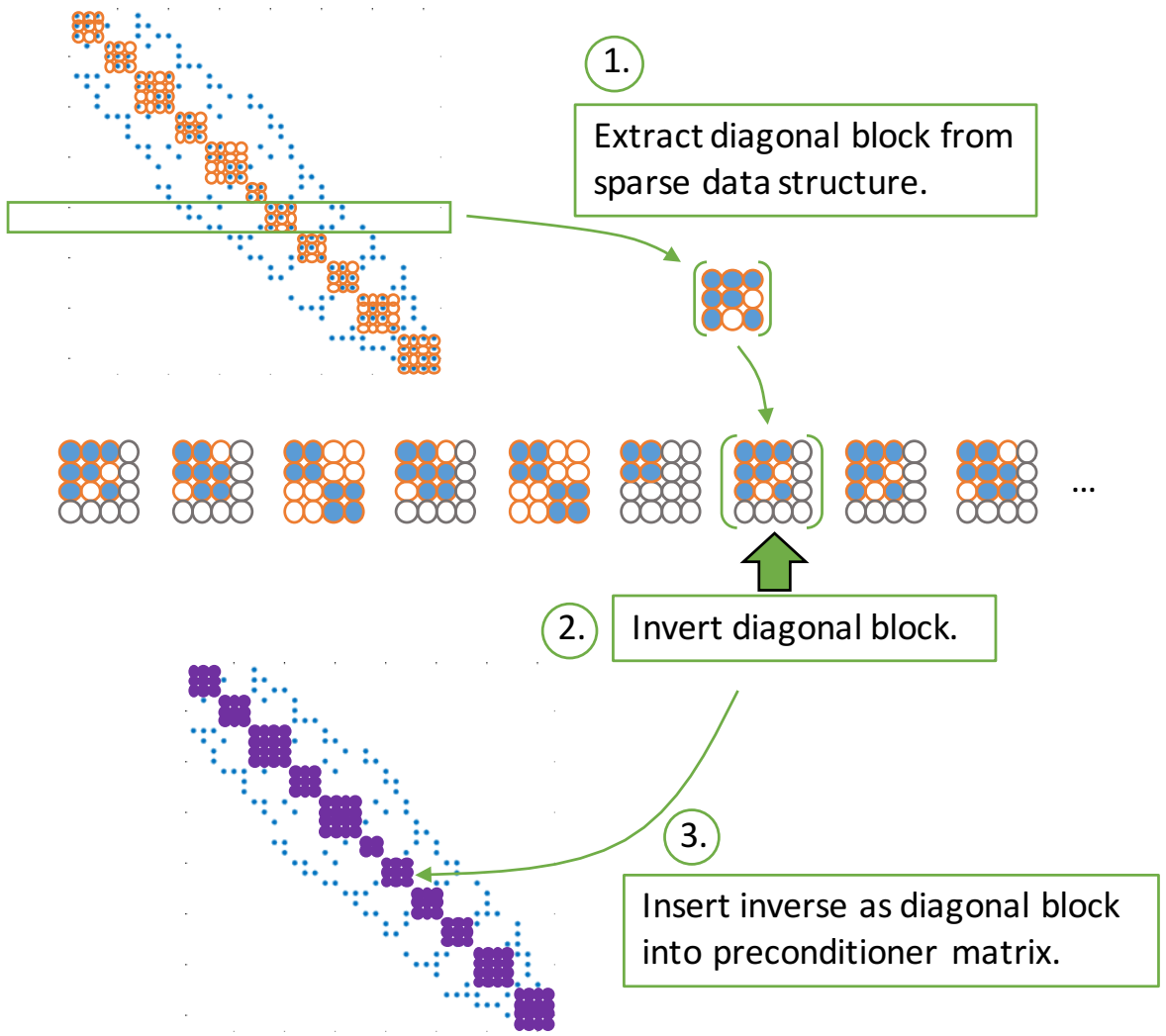
$$\sum_{blocks} 2m_i^3$$

Block-Jacobi preconditioning



- Cost of **Inversion:**
  $2m_i^3$ FLOPs for block of size m_i.
  $$\sum_{blocks} 2m_i^3$$
- Cost of **Preconditioner application**:
  m_i^2 FLOPs for block of size m_i :
  $$\sum_{blocks} m_i^2$$
- Total memory consumption:
  $$\sum_{blocks} m_i^2$$

Block-Jacobi preconditioning



- Cost of **Inversion:**
  $2m_i^3$ FLOPs for block of size m_i.

$$\sum_{blocks} 2m_i^3$$

- Cost of **Preconditioner application**:
  m_i^2 FLOPs for block of size m_i :

$$\sum_{blocks} m_i^2$$

- Total memory consumption:

$$\sum_{blocks} m_i^2$$

- **Energy balance** for one **preconditioner application** (DP)*:

$$\sum_{blocks} m_i^2 \cdot \underbrace{(100}_{\text{Computation/2}} + \underbrace{4800)}_{\text{data read}}$$

*John Shalf (LBNL)

Block-Jacobi preconditioning

Mixed Precision Idea:

- Do **all calculations in working precision**
- **Store** the block-Jacobi matrix in **reduced precision**
  - Benefit from **faster data access**
  - Benefit from **reduced data read cost**

Implications:

- Reduced preconditioner quality
  - Need for **more Krylov solver iterations**
  - Potential loss of regularity (breakdown)

- Cost of **Inversion:**
  $2m_i^3$ FLOPs for block of size m_i.
  $$\sum_{blocks} 2m_i^3$$
- Cost of **Preconditioner application**:
  m_i^2 FLOPs for block of size m_i :
  $$\sum_{blocks} m_i^2$$
- Total memory consumption:
  $$\sum_{blocks} m_i^2$$

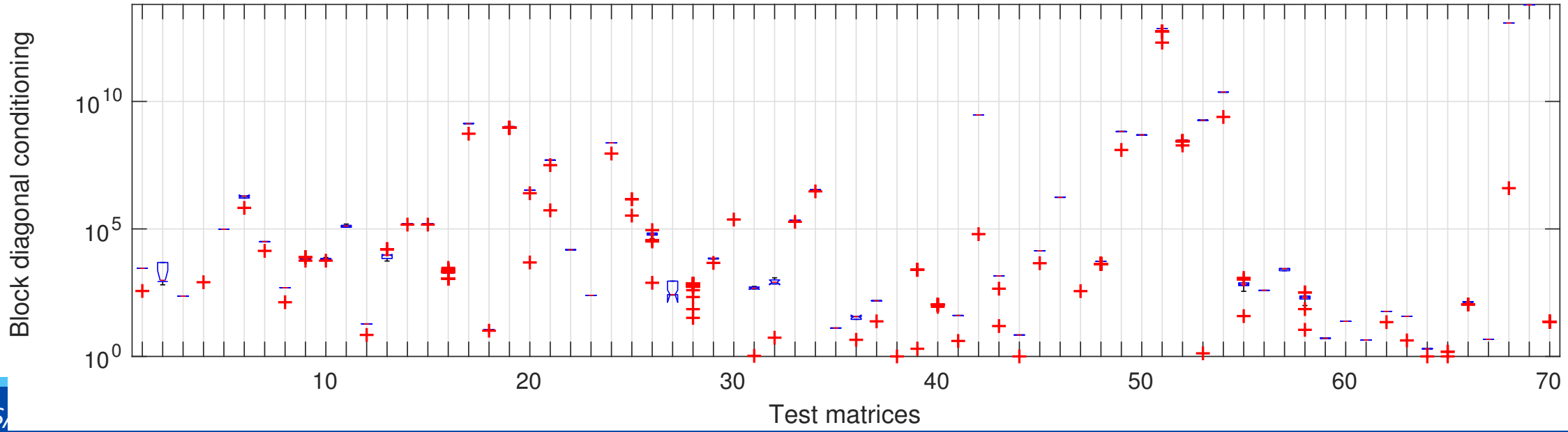
- **Energy balance** for one **preconditioner application** (DP)*:

$$\sum_{blocks} m_i^2 \cdot \underbrace{(100}_{\text{Computation/2}} + \underbrace{4800)}_{\text{data read}}$$

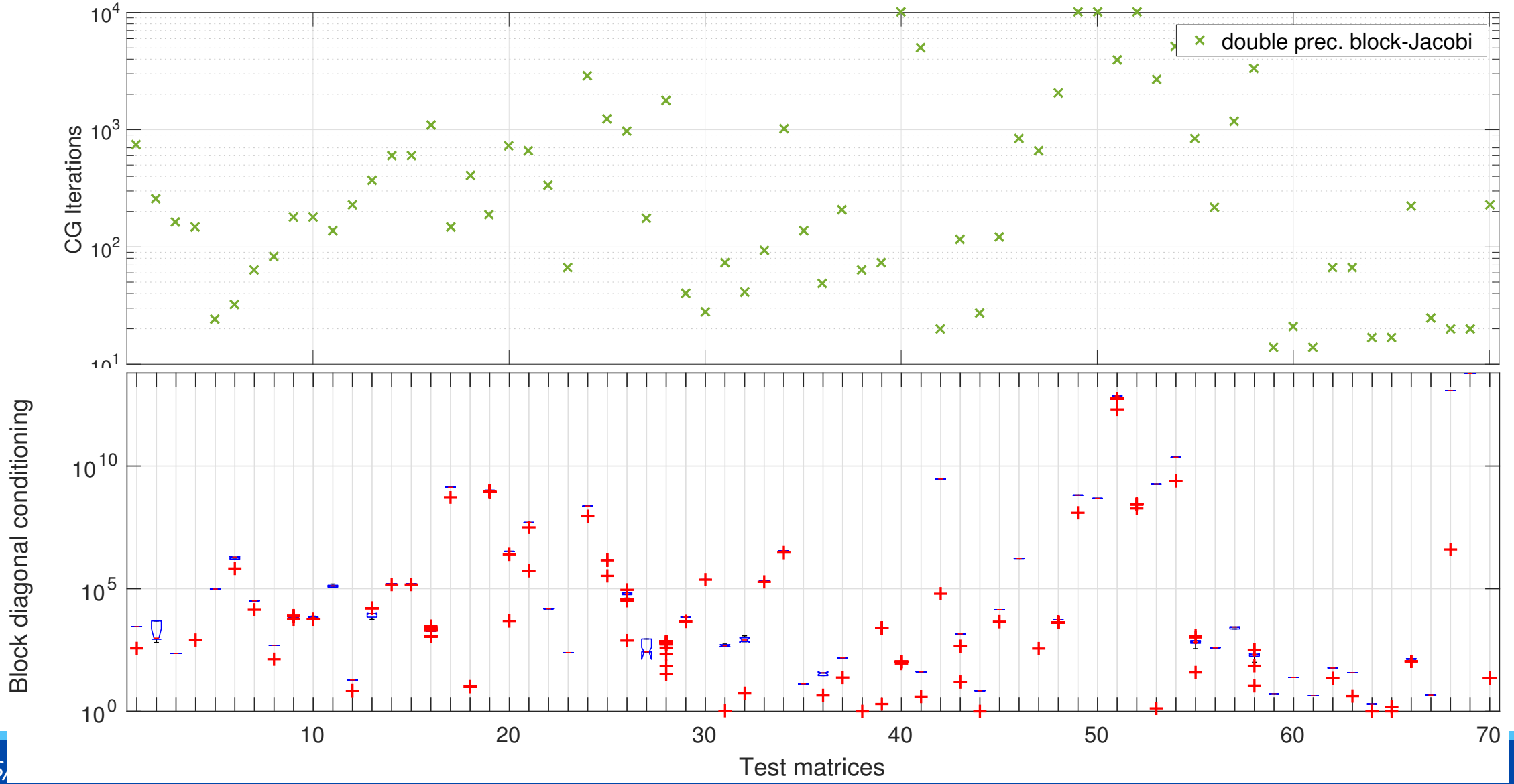
*John Shalf (LBNL)

Block-Jacobi preconditioning

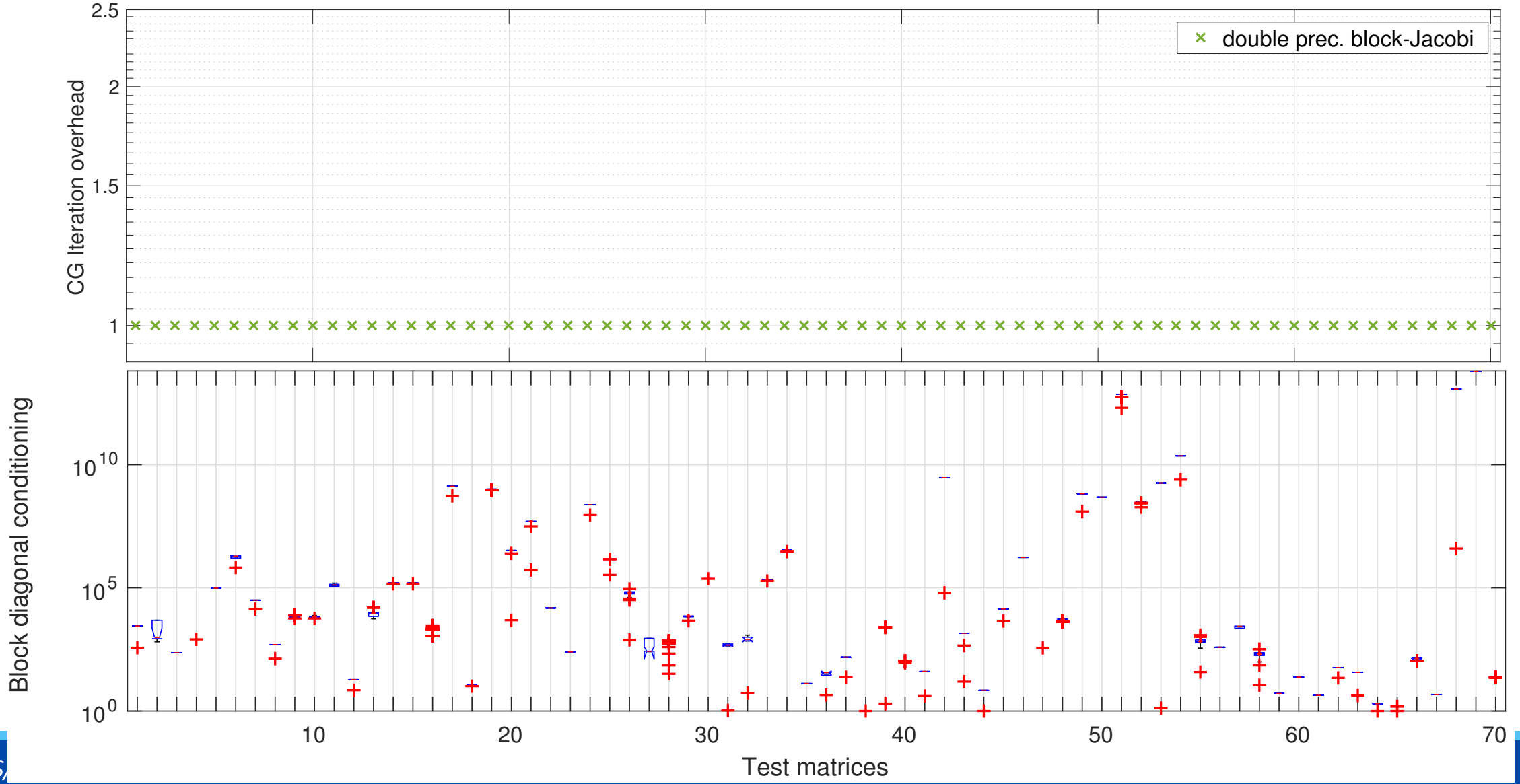
- 70 matrices from the SuiteSparse Matrix Collection
- Use block-size 24 with Super-Variable agglomeration (24 is upper bound for size of blocks)
- Report conditioning of all arising diagonal blocks



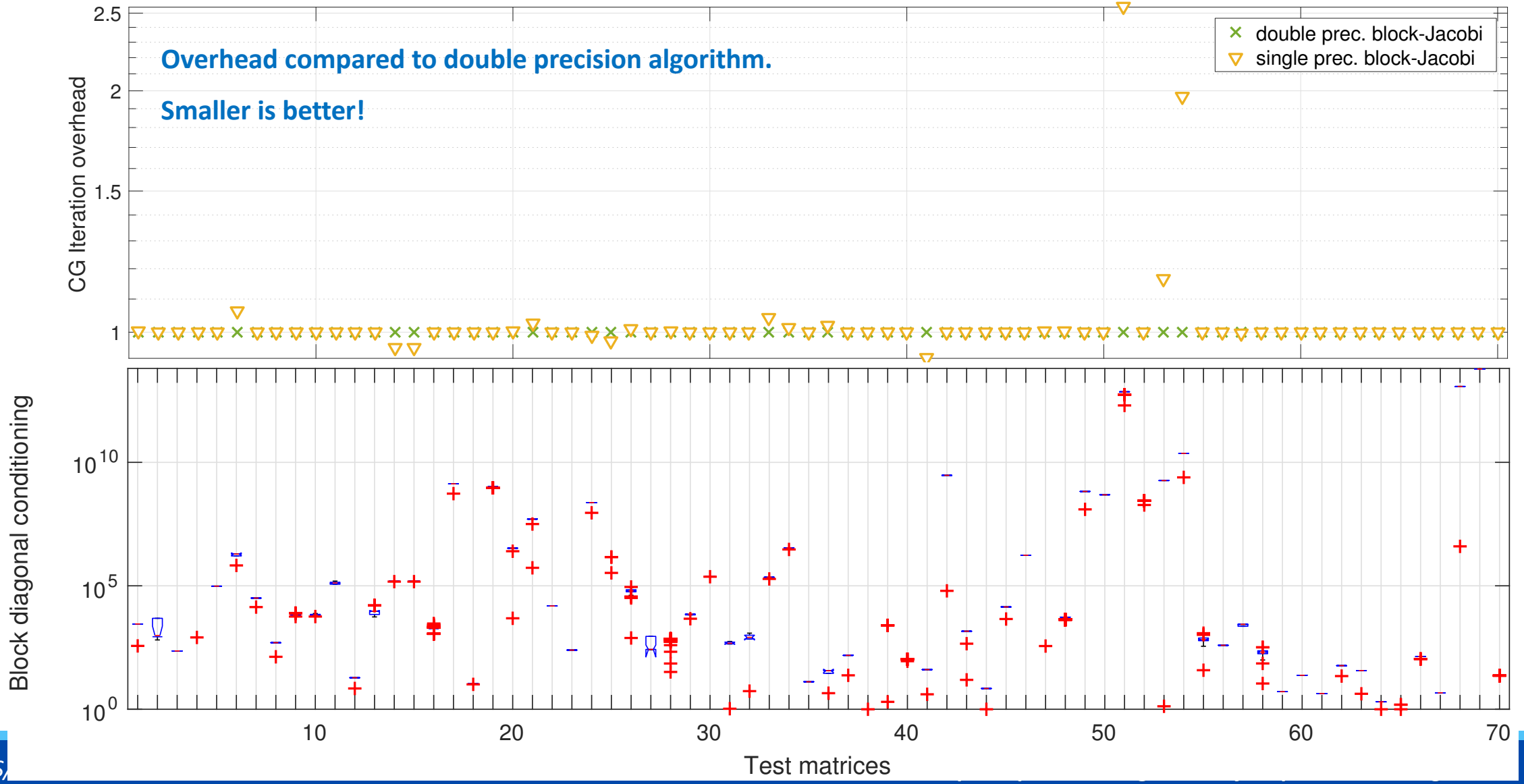
Block-Jacobi preconditioning



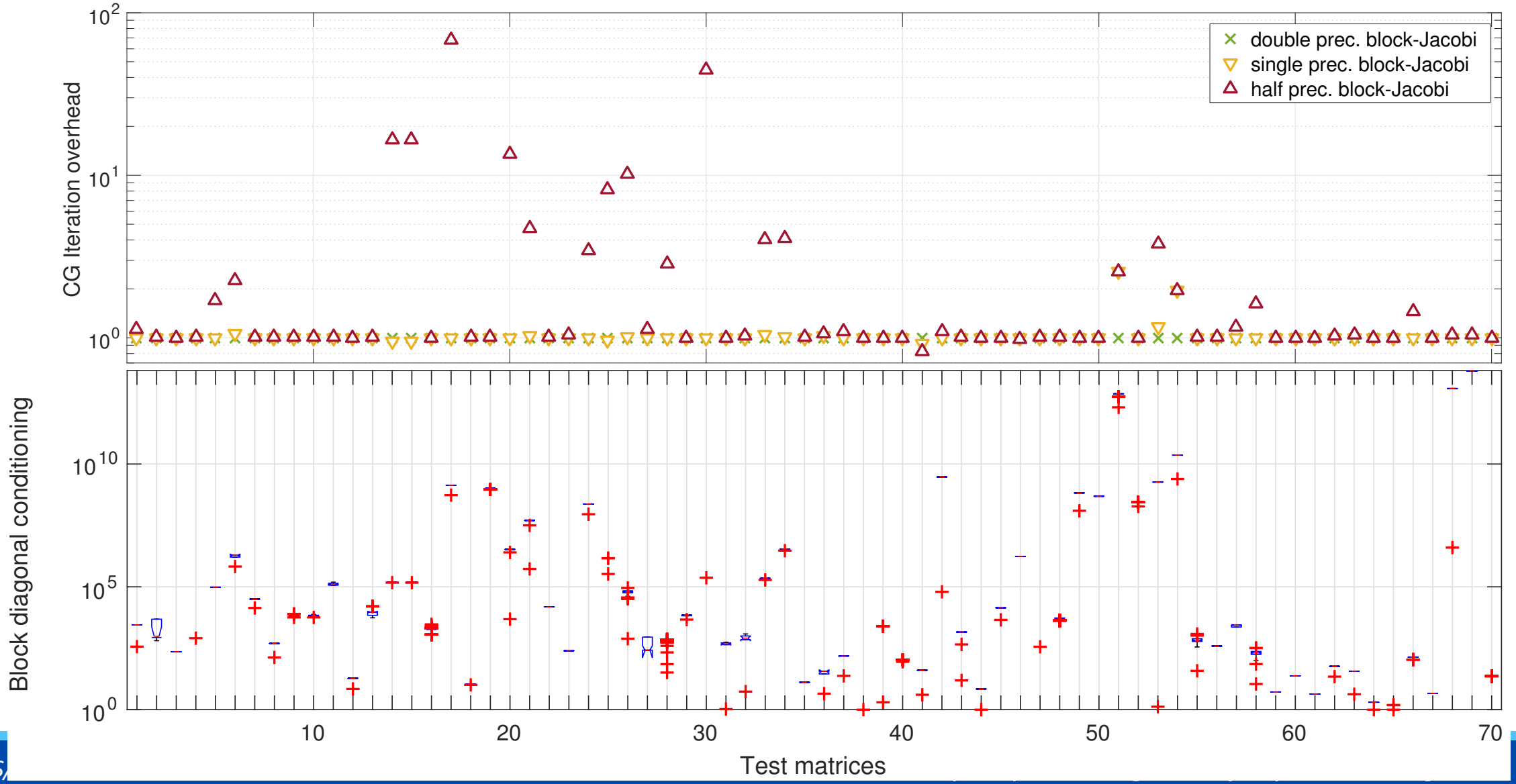
Block-Jacobi preconditioning



Block-Jacobi preconditioning



Block-Jacobi preconditioning

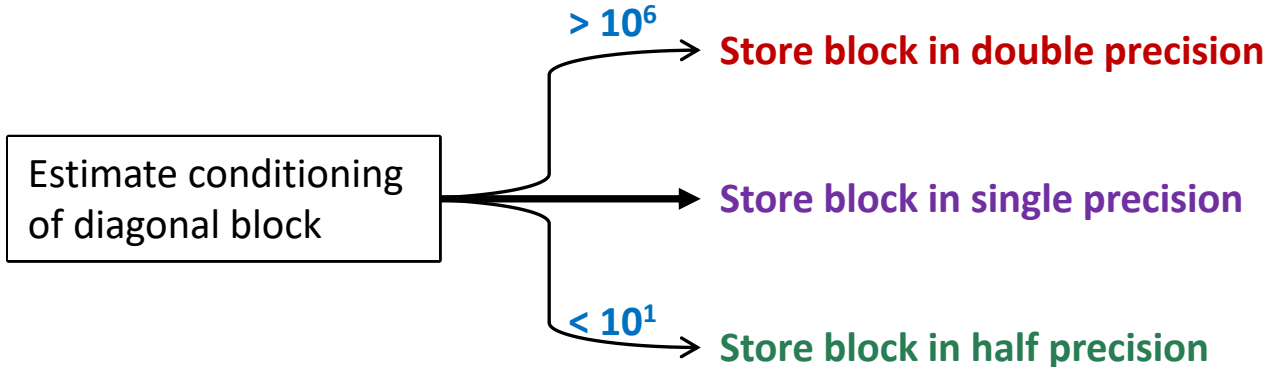


Adaptive-precision Block-Jacobi preconditioning

Adaptive Precision Idea:

- **All computations use double precision!**

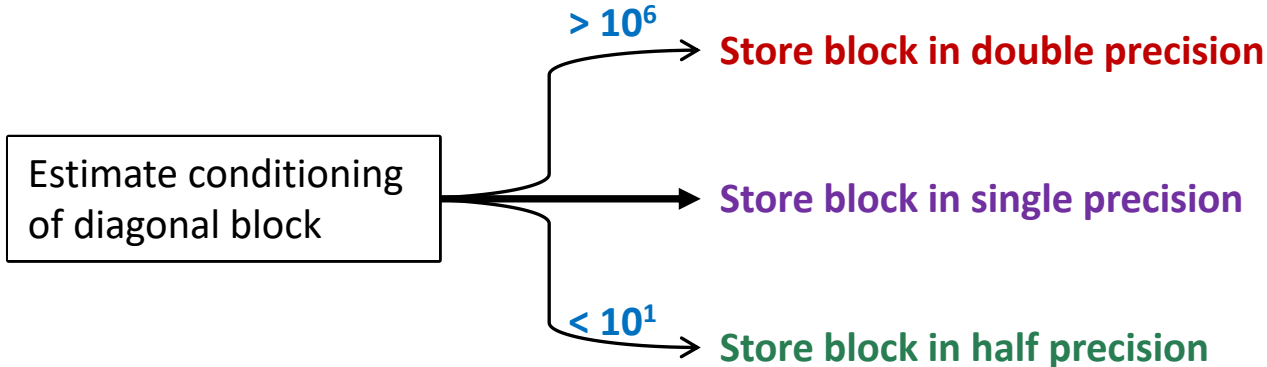
- Store distinct blocks in different formats
- Use **single precision as standard** storage format
- Where **necessary:** switch to **double**
- For well-conditioned blocks use **half precision**



Adaptive-precision Block-Jacobi preconditioning

Adaptive Precision Idea:

- **All computations use double precision!**
- Store distinct blocks in different formats
- Use **single precision as standard** storage format
- Where **necessary:** switch to **double**
- For well-conditioned blocks use **half precision**



Operation	approximate energy cost
DP floating point multiply-add	100 pJ
DP DRAM read-to-register	4800 pJ
DP word transmit-to-neighbor	7500 pJ
DP word transmit-across-system	9000 pJ

*John Shalf (LBNL)

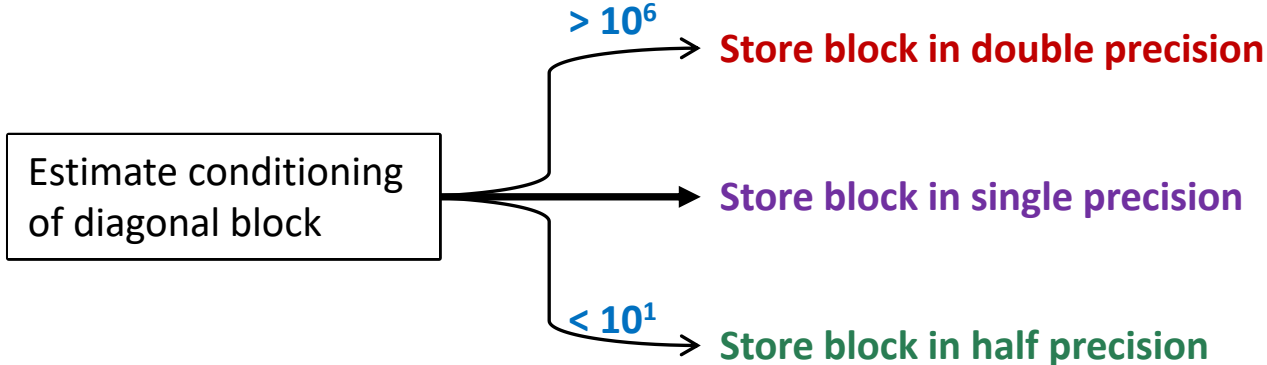
Energy model:

- 4800 pJ for double precision (64 bit)
- 2400 pJ for single precision / integers (32 bit)
- 1200 pJ for half precision (16 bit)

Adaptive-precision Block-Jacobi preconditioning

Adaptive Precision Idea:

- **All computations use double precision!**
- Store distinct blocks in different formats
- Use **single precision as standard** storage format
- Where **necessary:** switch to **double**
- For well-conditioned blocks use **half precision**



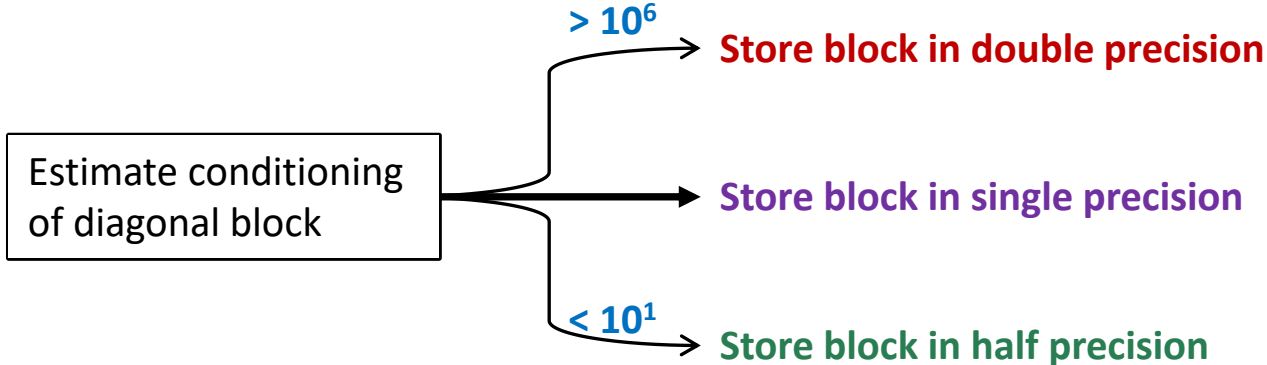
How much data we need to read/write in a Conjugate Gradient (CG) loop:

Operation	Memory volume	# per CG loop	Energy est.
CSR-SpMV	nz double + nz int + n int + 2n double	1	(nz + 2n) * 4800 pJ + (nz + n) * 2400 pJ
axpy	3n double	3	9n * 4800 pJ
dot	2n double	2	4n * 4800 pJ
preconditioner	[used format]	1	* ?

Adaptive-precision Block-Jacobi preconditioning

Adaptive Precision Idea:

- **All computations use double precision!**
- Store distinct blocks in different formats
- Use **single precision as standard** storage format
- Where **necessary:** switch to **double**
- For well-conditioned blocks use **half precision**



Energy model for block-Jacobi-CG

- We ignore computational cost, only memory access
- No data (matrix / vector) is cached, only DRAM reads
- CG outer solver (in DP)
- Adaptive precision for the block-Jacobi preconditioner

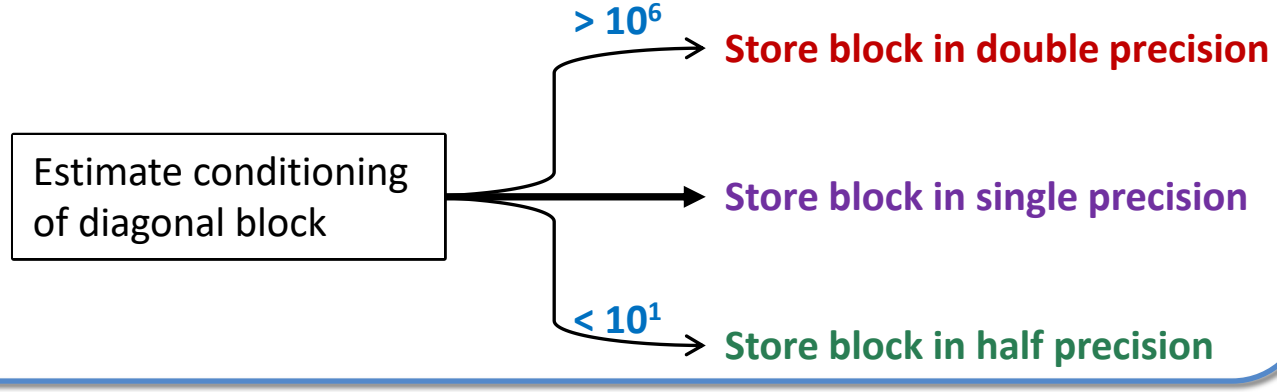
Block-Jacobi CG

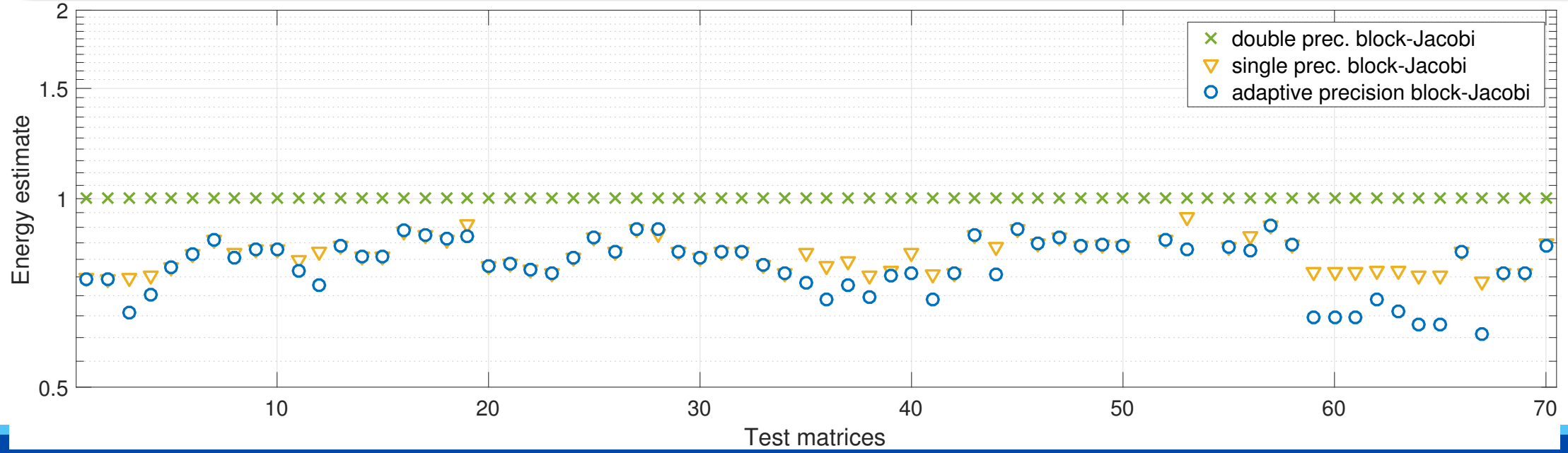
- Ax=b with x:=0 and b:=A*1
- Relative residual stopping crit. 1e-9

Adaptive-precision Block-Jacobi preconditioning

Adaptive Precision Idea:

- **All computations use double precision!**
- Store distinct blocks in different formats
- Use **single precision as standard** storage format
- Where **necessary:** switch to **double**
- For well-conditioned blocks use **half precision**





Outline

- A recipe to saving energy: Optimize performance!!!

    1. Choose the “right” hardware
    2. Dynamic Voltage-Frequency Scaling (DVFS)
    3. Dynamic Concurrency Throttling (DCT)
    4. Avoid polling
    5. Near Threshold Voltage Computing (NTVC)
    6. Energy-proportional hardware
    7. Virtualization of HPC resources
    8. Approximate computing/adaptive precision

… but other “chefs” may propose a different recipe

Adaptive-Precision Algorithms for Sparse Linear Algebra

Enrique S. QUINTANA-ORTÍ
Professor of Computer Architecture
Group leader High Performance Computing & Architectures (HPC&A) group
http://www.uji.es/~quintana