INTERNATIONAL
SUPERCOMPUTING CONFERENCE

LINPACK Benchmark with Time Limits on Multicore & GPU Based Accelerators

Jack Dongarra
University of Tennessee & Oak Ridge National Laboratory, USA

What Is LINPACK?

- LINPACK is a package of mathematical software for solving problems in linear algebra, mainly dense linear systems of linear equations.
- LINPACK: "LINear algebra PACKage"
  - Written in Fortran 66
- The project had its origins in 1974
- The project had four primary contributors: myself when I was at Argonne National Lab, Jim Bunch from the University of California-San Diego, Cleve Moler who was at New Mexico at that time, and Pete Stewart from the University of Maryland.
- LINPACK as a software package has been largely superseded by LAPACK, which has been designed to run efficiently on shared-memory, vector supercomputers.

Computing in 1974

- Fortran 66
- High Performance Computers:
  - IBM 370/195, CDC 7600, Univac 1110, DEC PDP-10, Honeywell 6030
- Trying to achieve software portability
- Run efficiently
- BLAS (Level 1)
  - Vector operations
- Software released in 1979
  - About the time of the Cray 1

LINPACK Benchmark?

- The Linpack Benchmark is a measure of a computer's floating-point rate of execution.
  - It is determined by running a computer program that solves a dense system of linear equations.
- Over the years the characteristics of the benchmark has changed a bit.
  - In fact, there are three benchmarks included in the Linpack Benchmark report.
- LINPACK Benchmark
  - Dense linear system solve with LU factorization using partial pivoting
  - Operation count is: $2/3\ n^3 + O(n^2)$
  - Benchmark Measure: MFlop/s
  - Original benchmark measures the execution rate for a Fortran program on a matrix of size 100x100.

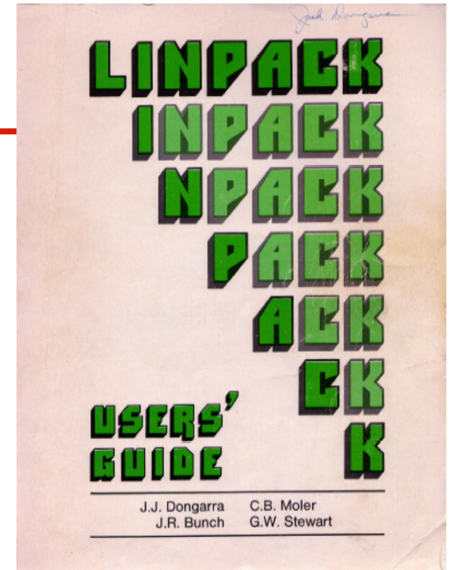
Accidental Benchmarker

- Appendix B of the Linpack Users' Guide
  - Designed to help users extrapolate execution time for Linpack software package
- First benchmark report from 1977;
  - Cray 1 to DEC PDP-10

UNIT = 10**6 TIME/(1/3 100**3 + 100**2)

Facility	TIME N=100 secs.	UNIT micro-secs.	Computer	Type	Compiler
NCAR	.049	0.14	CRAY-1	S	CFT, Assembly BLAS
LASL	.148	0.43	CDC 7600	S	FTN, Assembly BLAS
NCAR	.192	0.56	CRAY-1	S	CFT
LASL	.210	0.61	CDC 7600	S	FTN
Argonne	.297	0.86	IBM 370/195	D	H
NCAR	.359	1.05	CDC 7600	S	Local
Argonne	.388	1.33	IBM 3033	D	H
NASA Langley	.489	1.42	CDC Cyber 175	S	FTN
U. Ill. Urbana	.506	1.47	CDC Cyber 175	S	Ext. 4.6
LLL	.554	1.61	CDC 7600	S	CHAT, No optimize
SLAC	.579	1.69	IBM 370/168	D	H Ext., Fast mult.
Michigan	.631	1.84	Amdahl 470/V6	D	H
Toronto	.890	2.59	IBM 370/165	D	H Ext., Fast mult.
Northwestern	1.44	4.20	CDC 6600	S	FTN
Texas	1.93*	5.63	CDC 6600	S	RUN
China Lake	1.95*	5.69	Univac 1110	S	V
Yale	2.59	7.53	DEC KL-20	S	F20
Bell Labs	3.46	10.1	Honeywell 6080	S	Y
Wisconsin	3.49	10.1	Univac 1110	S	V
Iowa State	3.54	10.2	Itel AS/5 mod3	D	H
U. Ill. Chicago	4.10	11.9	IBM 370/158	D	G1
Purdue	5.69	16.6	CDC 6500	S	FUN
U. C. San Diego	13.1	38.2	Burroughs 6700	S	H
Yale	17.1*	49.9	DEC KA-10	S	F40

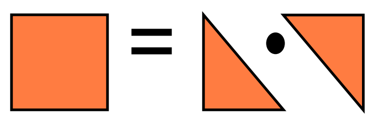
* TIME(100) = (100/75)**3 SGEFA(75) + (100/75)**2 SGESL(75)



Linpack 100

- Use the LINPACK software DGEFA and DGESL to solve a system of linear equations.
- DGEFA factors a matrix
- DGESL solve a system of equations based on the factorization.

<u>Step 1</u> $A = L\,U$



<u>Step 2</u> Forward Elimination

Solve $L\,y = b$

<u>Step 3</u> Backward Substitution

Solve $U\,x = y$

DGEFA and DGESL

```
c
c     gaussian elimination with partial pivoting
c
      info = 0
      nm1 = n - 1
      if (nm1 .lt. 1) go to 70
      do 60 k = 1, nm1
         kp1 = k + 1
c
c        find l = pivot index
c
         l = idamax(n-k+1,a(k,k),1) + k - 1
         ipvt(k) = l
c
c        zero pivot implies this column already triangularized
c
         if (a(l,k) .eq. 0.0d0) go to 40
c
c           interchange if necessary
c
            if (l .eq. k) go to 10
               t = a(l,k)
               a(l,k) = a(k,k)
               a(k,k) = t
   10       continue
c
c           compute multipliers
c
            t = -1.0d0/a(k,k)
            call dscal(n-k,t,a(k+1,k),1)
c
c           row elimination with column indexing
c
            do 30 j = kp1, n
               t = a(l,j)
               if (l .eq. k) go to 20
                  a(l,j) = a(k,j)
                  a(k,j) = t
   20          continue
               call daxpy(n-k,t,a(k+1,k),1,a(k+1,j),1)
   30       continue
         go to 50
   40    continue
            info = k
   50    continue
   60 continue
   70 continue
```

Most of the work is done Here: $O(n^3)$

```
c        first solve  l*y = b
c
         if (nm1 .lt. 1) go to 30
         do 20 k = 1, nm1
            l = ipvt(k)
            t = b(l)
            if (l .eq. k) go to 10
               b(l) = b(k)
               b(k) = t
   10       continue
            call daxpy(n-k,t,a(k+1,k),1,b(k+1),1)
   20    continue
   30    continue
c
c        now solve  u*x = y
c
         do 40 kb = 1, n
            k = n + 1 - kb
            b(k) = b(k)/a(k,k)
            t = -b(k)
            call daxpy(k-1,t,a(1,k),1,b(1),1)
   40    continue
      go to 100
   50 continue
```

Operation type	Operation count
addition	328350
multiplication	333300
reciprocal	99
absolute value	5364
comparison	4950
comparison with zero	5247

For Linpack with n = 100

- Not allowed to touch the code.
- Only set the optimization in the compiler and run.
- Table 1 of the report
  - http://www.netlib.org/benchmark/performance.pdf

5/6/2010 7

Table 1: **Performance in Solving a System of Linear Equations**

Computer	"LINPACK Benchmark" OS/Compiler	n=100 Mflop/s	"TPP" Best Effort n=1000 Mflop/s	"Theoritical Peak" Mflop/s
Intel Pentium Woodcrest (1 core, 3 GHz)	ifort -parallel -xT -O3 -ipo -mP2OPT_hlo_loop_unroll_factor=2	3018	6542	12000
Intel Pentium Woodcrest (1 core, 2.67 GHz)	ifort -O3 -ipo -xT -r8 -i8	2636		10680
Intel Core 2 Q6600 Kensfield) (4 core, 2.4 GHz)			13130	38400
Intel Core 2 Q6600 Kensfield) (3 core, 2.4 GHz)			11980	28800
Intel Core 2 Q6600 Kensfield) (2 core, 2.4 GHz)			9669	19200
Intel Core 2 Q6600 Kensfield) (1 core, 2.4 GHz)	ifort -O3 -xT -ipo -static -i8 -mP2OPT_hlo_loop_unroll_factor=2	2426	7519	9600
NEC SX-8/8 (8proc. 2 GHz)			75140	128000
NEC SX-8/4 (4proc. 2 GHz)			[illegible]	[illegible]

Linpack Benchmark Over Time

- In the beginning there was the Linpack 100 Benchmark (1977)
  - n=100 (80KB); size that would fit in all the machines
  - Fortran; 64 bit floating point arithmetic
  - No hand optimization (only compiler options)
- Linpack 1000 (1986)
  - n=1000 (8MB); wanted to see higher performance levels
  - Any language; 64 bit floating point arithmetic
  - Hand optimization OK
- Linpack HPL (1991) (Top500; 1993)
  - Any size (n as large as you can);
  - Any language; 64 bit floating point arithmetic
  - Hand optimization OK
    - Strassen's method not allowed (confuses the op count and rate)
  - Reference implementation available (HPL)
- In all cases results are verified by looking at: $\frac{\| Ax - b \|}{\| A \| \|x\| n \varepsilon} = O(1)$
- Operations count for factorization $\frac{2}{3}n^3 - \frac{1}{2}n^2$; solve $2n^2$

High Performance Linpack (HPL)

Benchmark Name	Matrix dimension	Optimizations allowed	Parallel Processing
Linpack 100	100	compiler	–[a]
Linpack 1000[b]	1000	hand, code replacement	–[c]
Linpack Parallel	1000	hand, code replacement	Yes
HPLinpack[d]	Arbitrary (usually as large as possible)	hand, code replacement	Yes

[a] Compiler parallelization possible.

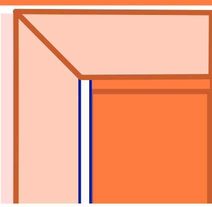
[b] Also known as TPP (Toward Peak Performance) or Best Effort

[c] Multiprocessor implementations allowed.

[d] Highly-Parallel LINPACK Benchmark is also known as NxN Linpack Benchmark or High Parallel Computing (HPC).

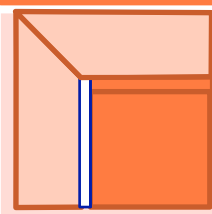
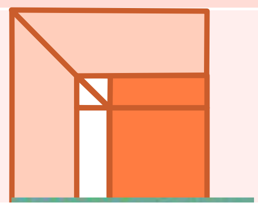
A New Generation of Software:

Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations

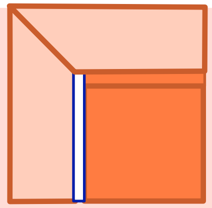
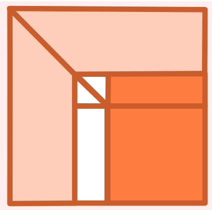
A New Generation of Software:

Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations

A New Generation of Software:

Every 10 Years or So.

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing

HPL Code is Based on ScaLAPACK

- Uses a form of look ahead to overlap communication and computation
- Uses MPI directly avoiding the overhead of BLASC communication layer.
- HPL doesn't form L (pivoting is only applied forward)
- HPL doesn't return pivots (they are applied as LU progresses)
  - LU is applied on [A, b] so HPL does one less triangular solve(HPL: triangular solve with U; ScaLAPACK: triangular solve with Land then U)
- HPL uses recursion to factorize the panel, ScaLAPACK uses rank-1 updates
- HPL has many variants for communication and computation: people write papers how to tune it; ScaLAPACK gives you a lot of defaults that are overall OK
- HPL combines pivoting with update: coalescing messages usually helps with performance

Communication and Computation Differences

- ScaLAPACK
- Communication layer
  - BLACS on top of:
    - MPI, PVM, vendor lib
- Communication variants
  - Only one pivot finding
  - BLACS broadcast topologies
- Rank-k panel factorization
- Separate pivot and panel data
  - Larger message count
- Lock-step operation
  - Extra synchronization points

- HPL
- Communication layer
  - MPI
    - Vendor MPI
- Communication variants
  - Pivot finding reductions
  - Update broadcasts
- Recursive panel factorization
- Coalescing of pivot and panel data
  - Smaller message count
- Look-ahead panel factorization
  - Critical path optimization

Differences in Formulation

- ScaLAPACK
  Ax=b
  AX=B (multiple RHS)
- First step: pivot and factorize
  PA = LU
- Second step: apply pivot to b
  b' = Pb
- Third step: back-solve with L
  Ly = b'
- Fourth step: back-solve with U
  Ux = y
- Result: L, U, P, x

- HPL
  Ax=b
- First step:pivot,factorize,apply L
  A,b = L'U,y
- Second step: back-solve with U
  Ux = y
-
- Result: U, x, scrambled L

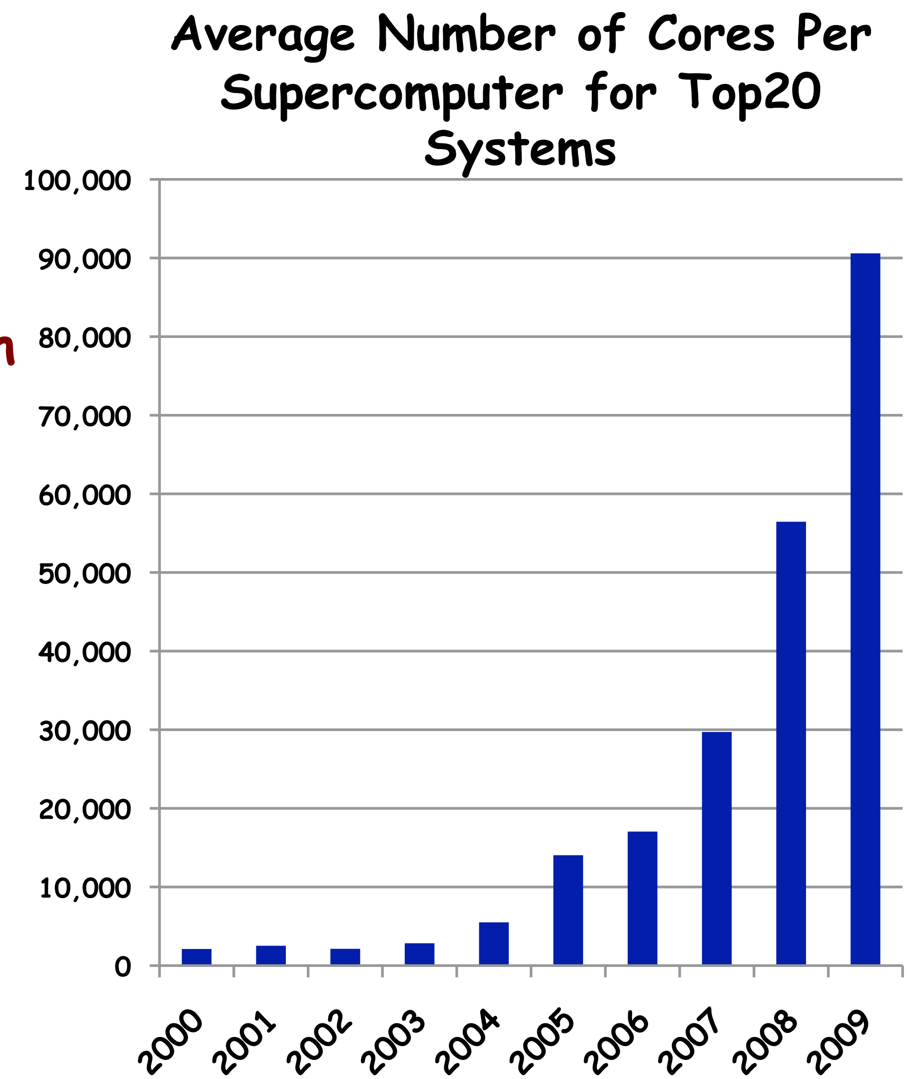
Other Differences

- ScaLAPACK
- Multiple precisions
  - 32-bit/64-bit/real /complex
- Random number generation
  - 32-bit
- Supported linear algebra libraries
  - BLAS

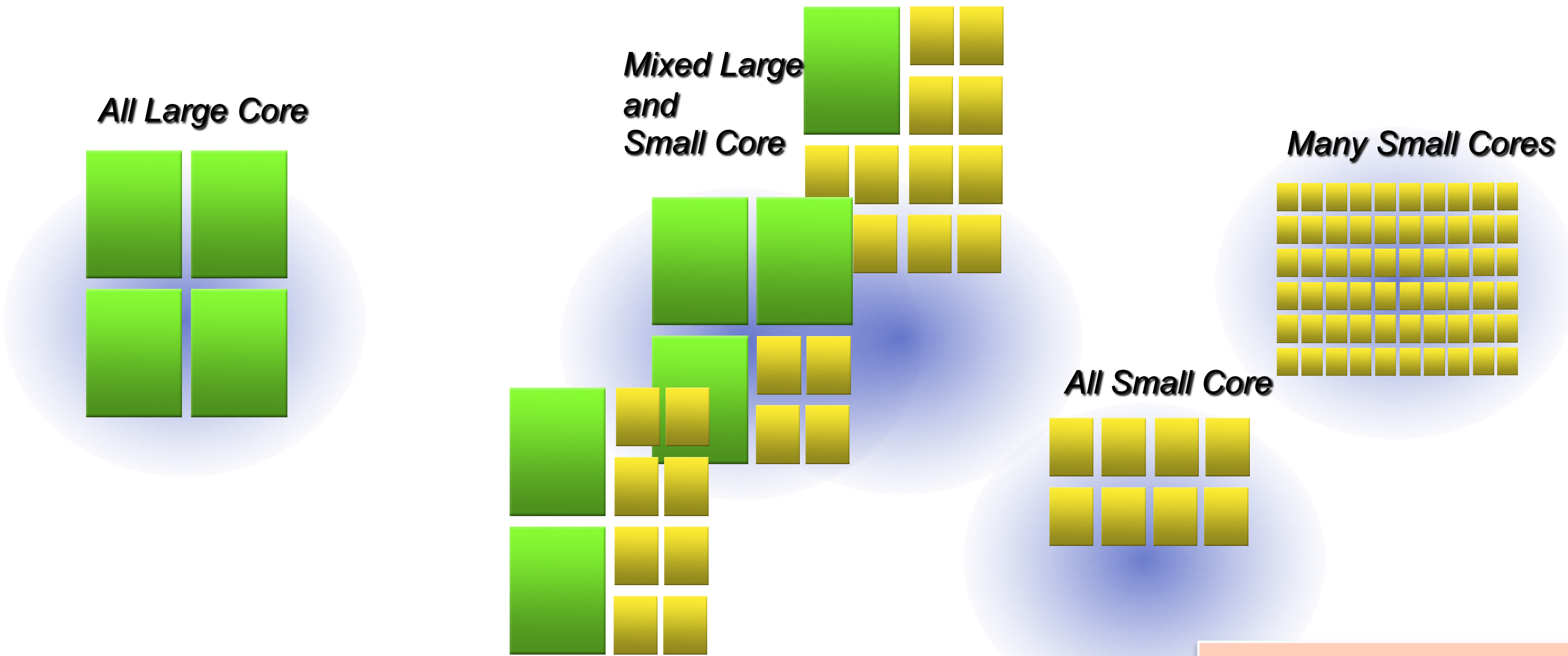
- HPL
- One precision
  - 64-bit real
- Random number generation
  - 64-bit
- Supported linear algebra libraries
  - BLAS, VSIPL

Moore's Law Reinterpreted

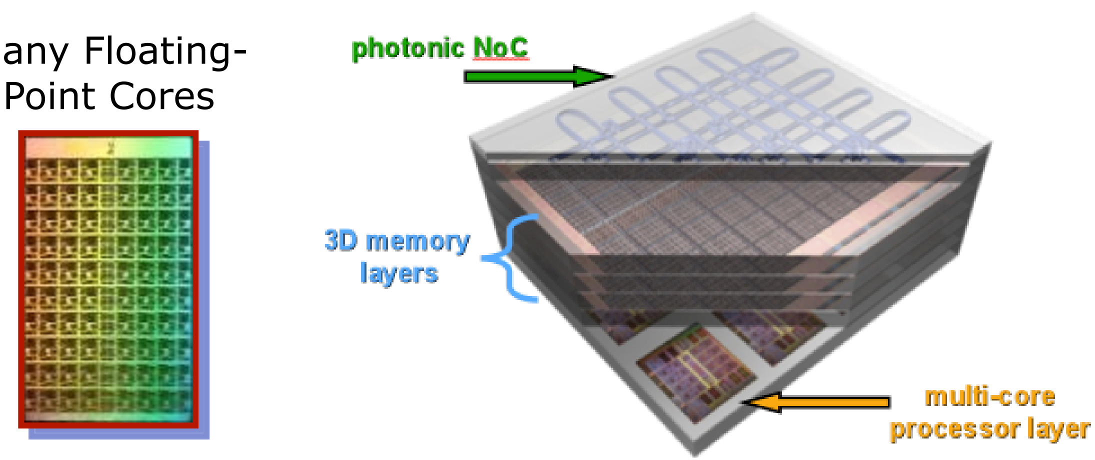
- Number of cores per chip doubles every 2 year, while clock speed decreases (not increases).
  - Need to deal with systems with millions of concurrent threads
    - Future generation will have billions of threads!
  - Need to be able to easily replace inter-chip parallelism with intro-chip parallelism
- Number of threads of execution doubles every 2 year



What’s Next?



Many Floating-Point Cores



+ 3D Stacked Memory

Different Classes of Chips
- Home
- Games / Graphics
- Business
- Scientific

Future Computer Systems

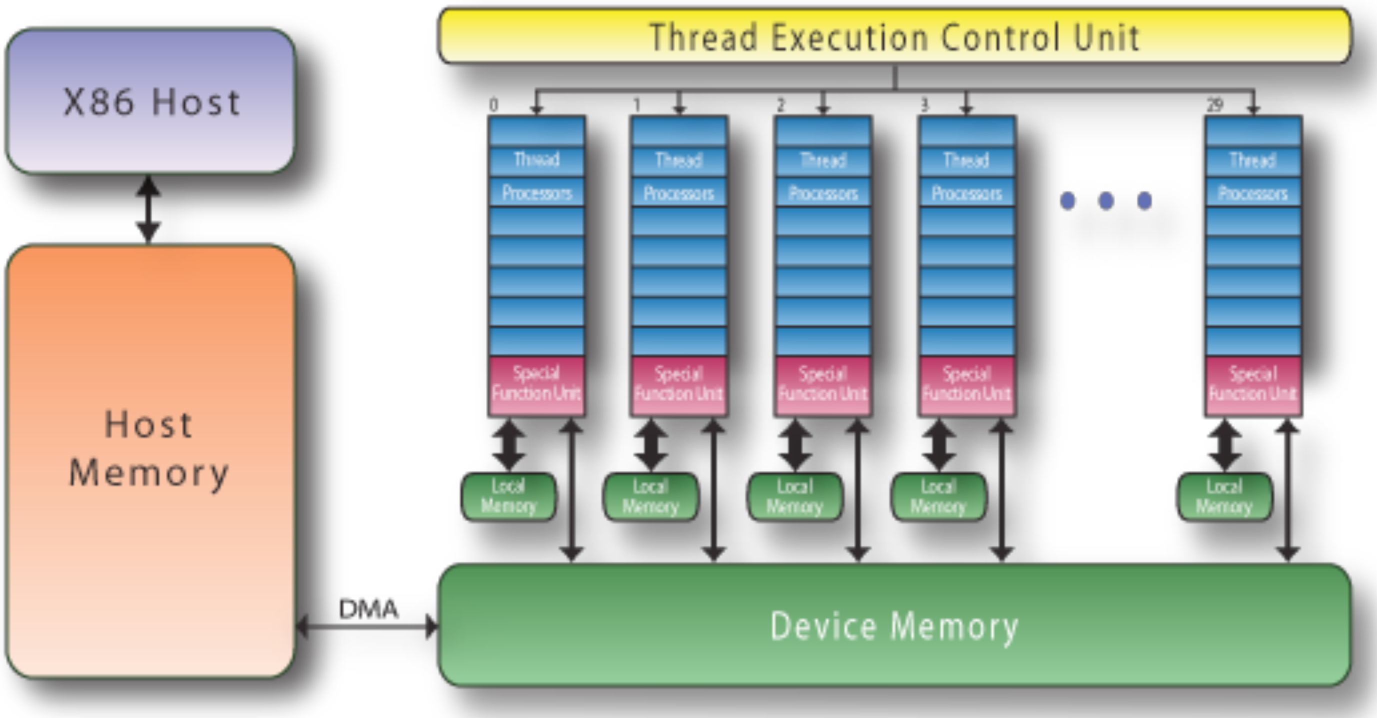
- Most likely be a hybrid design
- Think standard multicore chips and accelerator (GPUs)
- Today accelerators are attached
- Next generation more integrated
- Intel's Larrabee? Now called "Knights Corner" and "Knights Ferry" to come.
  - 48 x86 cores
- AMD's Fusion in 2011 - 2013
  - Multicore with embedded graphics ATI
- Nvidia's plans?

Exascale Systems: Two possible paths

- Light weight processors (think BG/P)
  - ~1 GHz processor (10^9)
  - ~1 Kilo cores/socket (10^3)
  - ~1 Mega sockets/system (10^6)

- Hybrid system (think GPU based)
  - ~1 GHz processor (10^9)
  - ~10 Kilo FPUs/socket (10^4)
  - ~100 Kilo sockets/system (10^5)

Commodity plus GPU Today



From: Michael Wolfe, PGI

Challenges of using GPUs

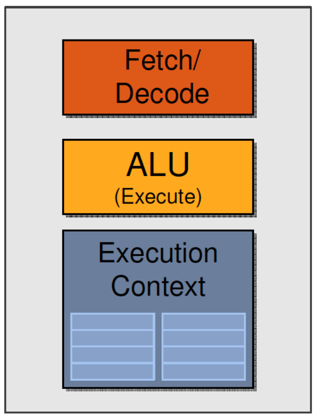
- **High levels of parallelism**
  Many GPU cores, serial kernel execution
  [e.g. 240 in the Nvidia Tesla; up to 512 in *Fermi* – to have concurrent kernel execution]

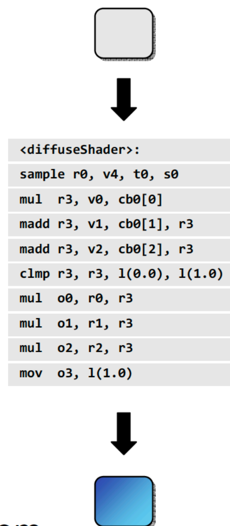
- **Hybrid/heterogeneous architectures**
  Match algorithmic requirements to architectural strengths
  [e.g. small, non-parallelizable tasks to run on CPU, large and parallelizable on GPU]

- **Compute *vs* communication gap**
  Exponentially growing gap; persistent challenge
  [Processor speed improves 59%, memory bandwidth 23%, latency 5.5%]
  [on all levels, e.g. a GPU Tesla C1070 (4 x C1060) has compute power of O(1,000) Gflop/s but GPUs communicate through the CPU using O(1) GB/s connection]

How to Count Cores?

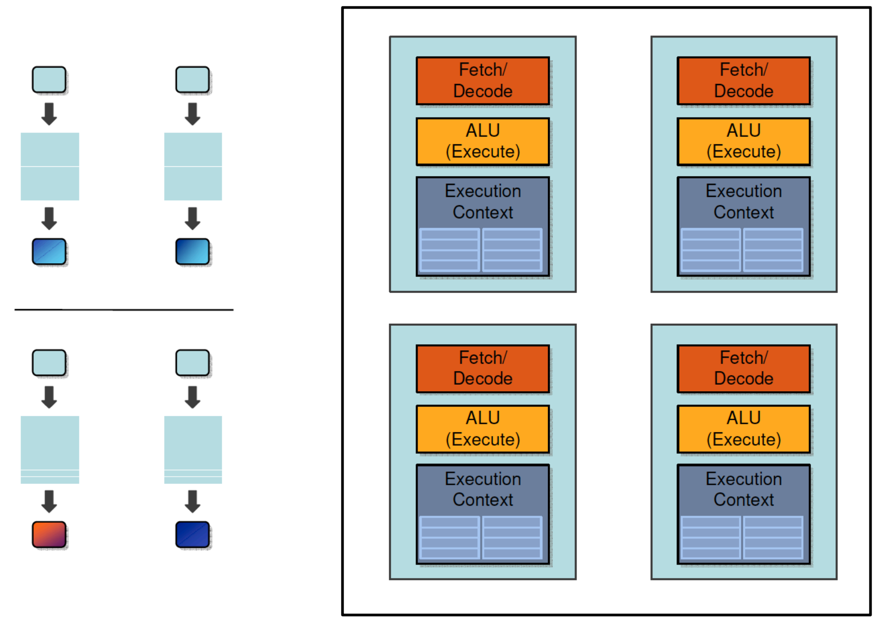
CPU Conventional Core





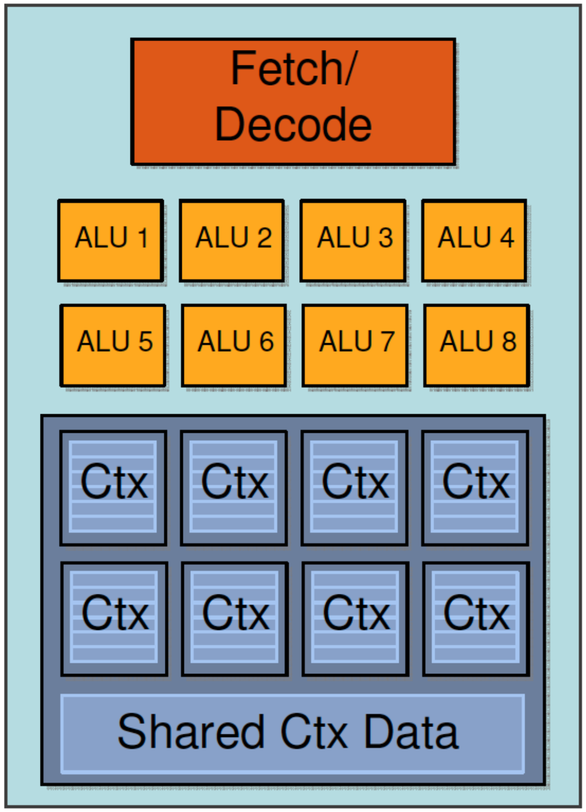
One instruction stream per work-item

Quad Core

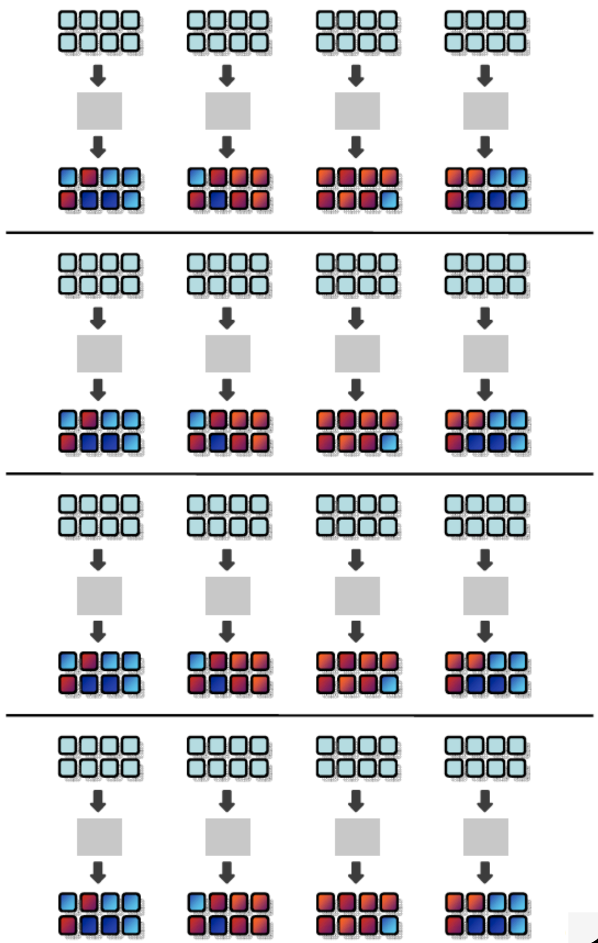


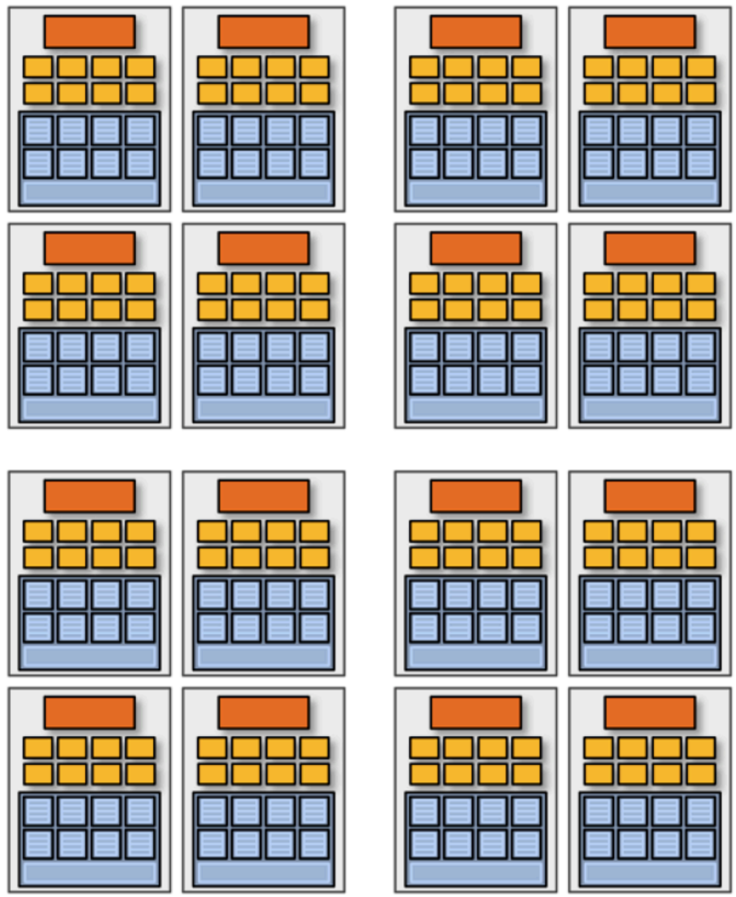
In GPUs - Add ALUs

- SIMD Processing
- Amortize cost/complexity of managing an instruction stream across many ALUs.
- NVIDIA refers to these ALUs as “CUDA Cores” (also streaming processors)



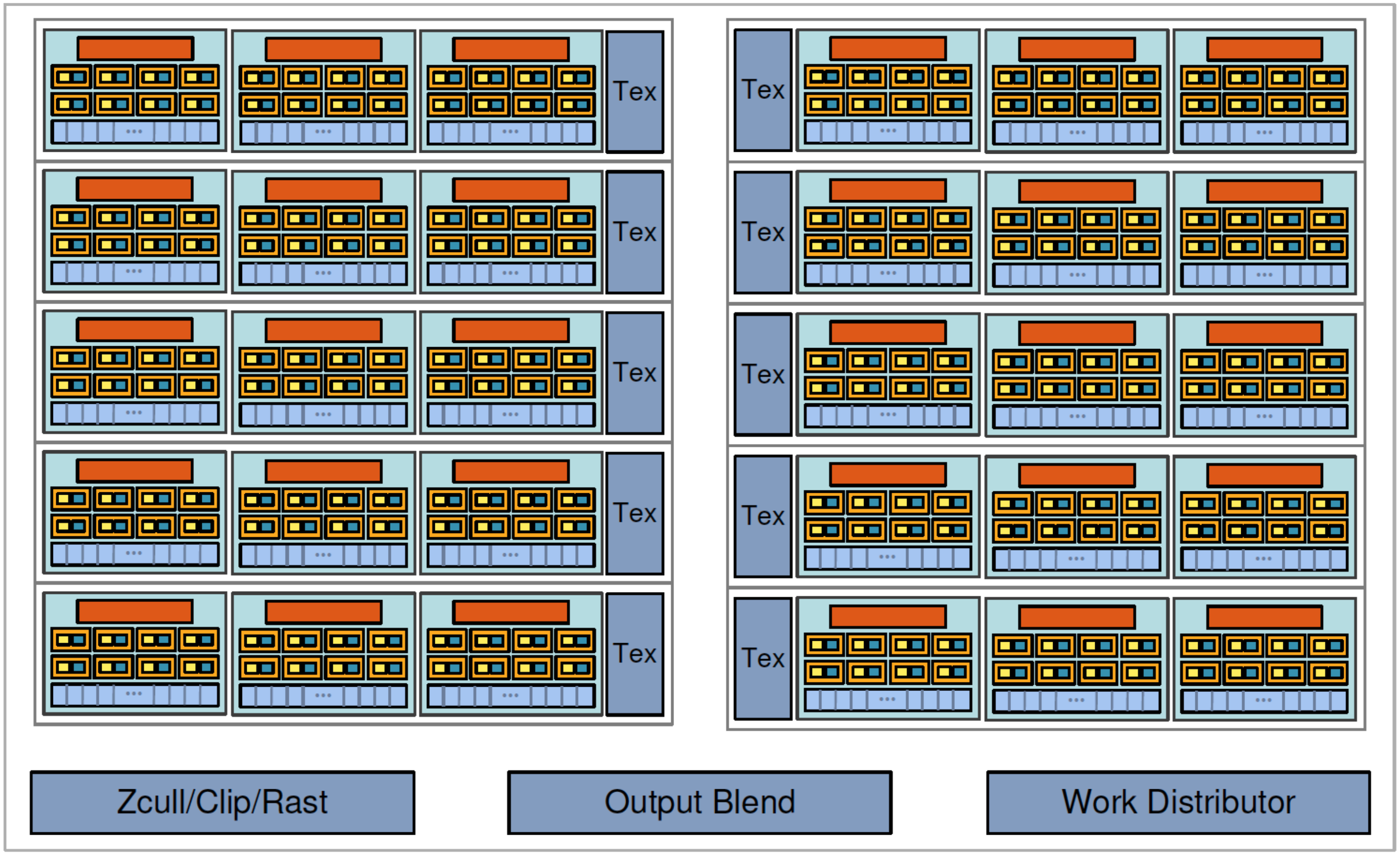
128 Elements in Parallel





16 cores each with 8 ALUs (CUDA Cores)
Total of 16 simultaneous instruction streams with 128 ALUs (CUDA Cores)

NVIDIA GT280 “old Telsa”

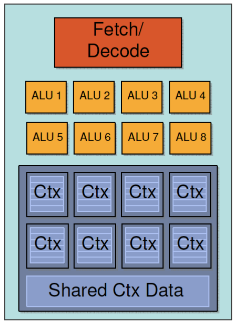


- 240 streaming processors (CUDA Cores) (ALUs)
- Equivalent to 30 processing cores, each with 8 “CUDA cores”

NVIDIA GeForce GTX 280 (Tesla)

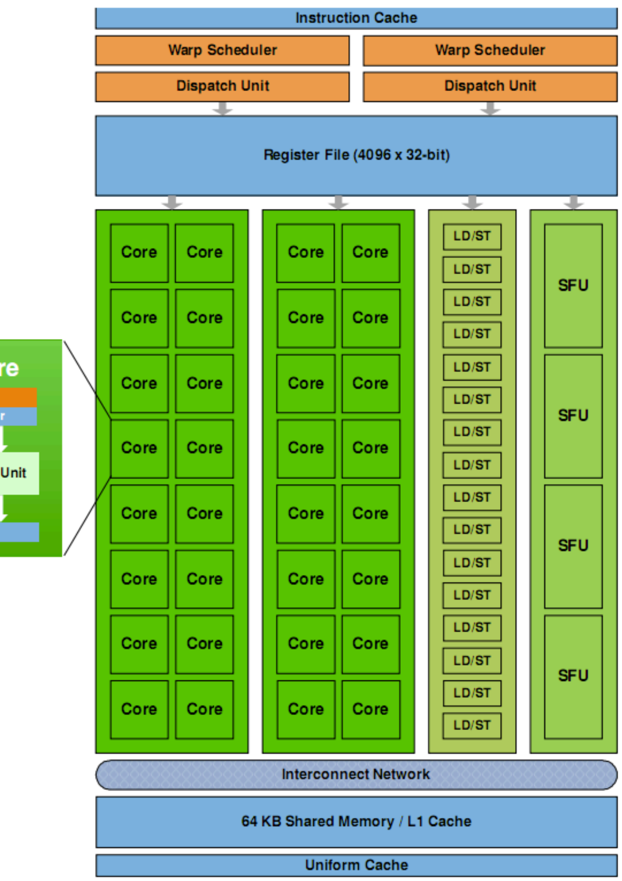
- NVIDIA-Speak
  - 240 CUDA cores (ALUs)
- Generic speak
  - 30 processing cores
    - 8 CUDA Cores (SIMD functional units) per core
  - 1 mul-add (2 flops) + 1 mul per functional unit (3 flops/cycle)
  - Best case theoretically: 240 mul-adds + 240 muls per cycle
    - 1.3 GHz clock
    - 30 * 8 * (2 + 1) * 1.33 = 933 Gflop/s peak
  - Best case reality: 240 mul-adds per clock
    - Just able to do the mul-add so 2/3 or 624 Gflop/s
  - All this is single precision
    - Double precision is 78 Gflop/s peak (Factor of 8 from SP; exploit mixed prec)
  - 141 GB/s bus, 1 GB memory
  - 4 GB/s via PCIe (we see: T = 11 us + Bytes/3.3 GB/s)
  - In SP SGEMM performance 375 Gflop/s

Processing Core



NVIDIA Fermi (GTX 480)

- **Fermi GTX 480 has 480 CUDA cores (ALUs)**
- **32 CUDA Cores (ALUs) in each of the 15 processing Cores**



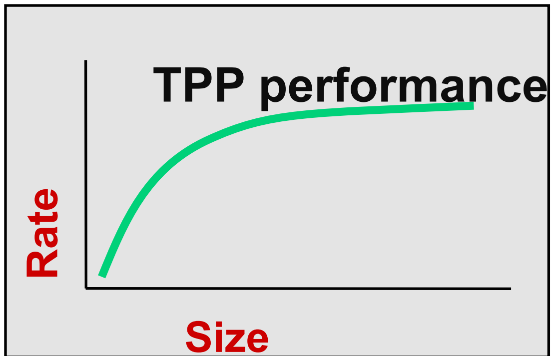
NVIDIA Tesla C2050 (Fermi), GF100 Chip

- NVIDIA-Speak
  - 448 CUDA cores (ALUs)
- Generic speak
  - 14 processing cores
    - 32 CUDA Cores (SIMD functional units) per core
  - 1 mul-add (2 flops) per ALU (2 flops/cycle)
  - Best case theoretically: 448 mul-adds
    - 1.15 GHz clock
    - 14 * 32 * 2 * 1.15 = 1.03 Tflop/s peak
  - All this is single precision
    - Double precision is half this rate, 515 Gflop/s
  - 144 GB/s bus, 3 GB memory
  - In SP SGEMM performance 580 Gflop/s
  - In DP DGEMM performance 300 Gflop/s
  - Power: 247 W
  - Interface PCIex16

Processing Core

High Performance Linpack

- Linpack benchmark (solve $Ax = b$, A is dense general matrix) uses $O(n^2)$ data and $O(n^3)$ operations.
- If we look at the performance as a function of size we see something like this.



- So you want to run a large a problem as you can on your machine to get the most performance.

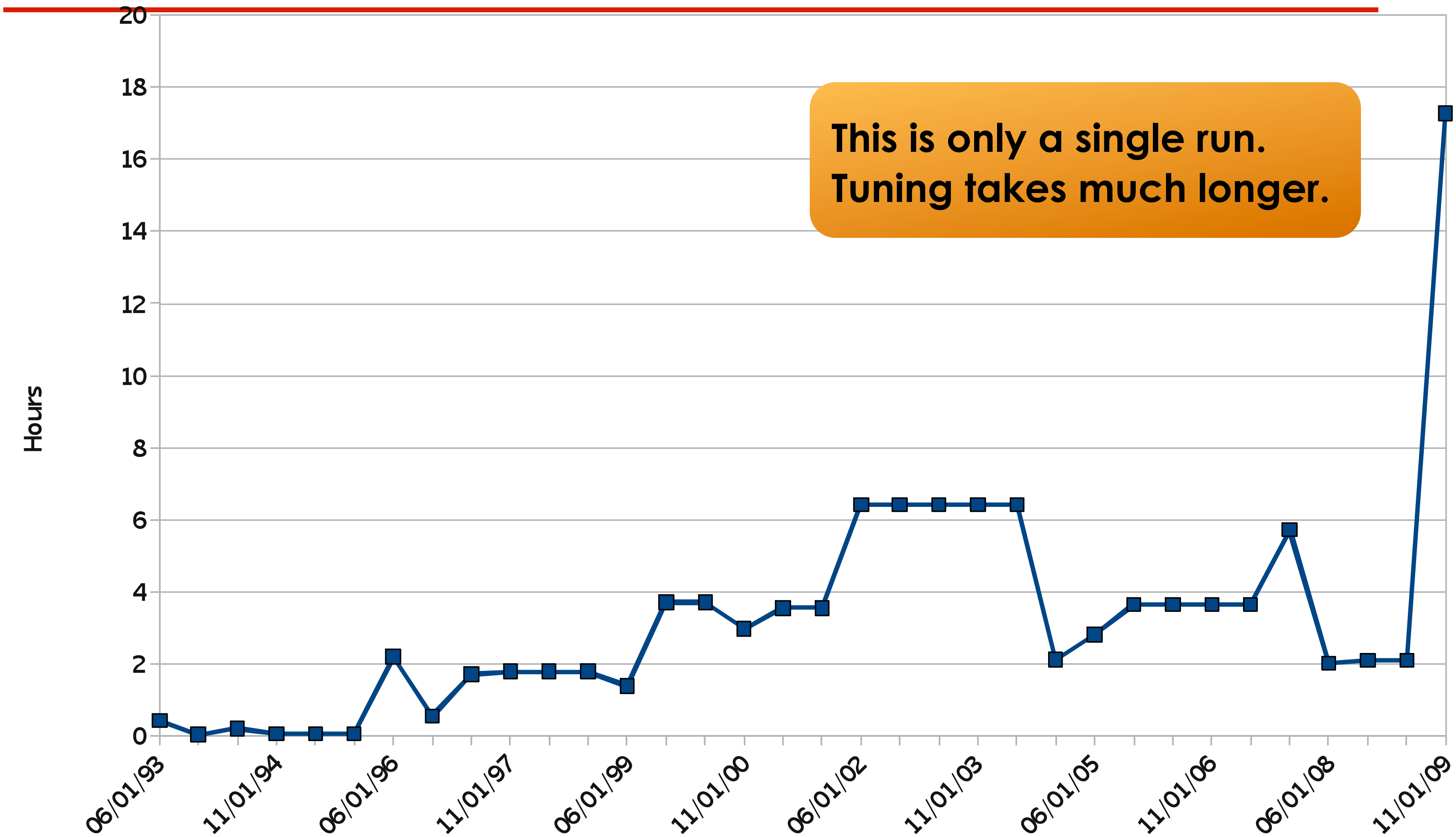
Benchmark Rules and Requirements

- Precision
  - 64-bit floating point
  - 32-bit not allowed
  - No Mixed precision
- Algorithm
  - Partial pivoting
  - No fast matrix-matrix multiply (i.e. Strassen's method)
  - No triangular matrix inversion on diagonal
- Data/Storage
  - Matrix generator must be used.
  - Initially: Data in main memory
  - During computation: arbitrary
  - At finish: Data in main memory
- Computation
  - Arbitrary: any device can compute
- Timing and performance
  - Clock is started and stopped with data in main memory.
  - All computation and data transfers are included in total time
  - Standard formula for performance $2/3 * n^3$ / time
- Verification
  - $|| Ax-b || / (||A|| ||x|| - ||b|| \, n \, e) = O(10)$

"How Long Will This HPL Thing Run?"

- The LANL RoadRunner HPL run took about 2 hours.
  - They ran a size of $n=2.3 \times 10^6$
- At ORNL they have more memory, 300 TB, and they wanted to run a problem which used most of it. They ran a matrix of size $n = 4.7 \times 10^6$
  - This run took about 18 hours!!
- JAXA Fujitsu system (slower than ORNL's system) ran a matrix of size 3.3×10^6
  - That took over 60 hours!!!!

Time to Run for #1 Entry on TOP500



In a Few Years …

- Have a 5 Pflop/s system
- If memory goes up by a factor of 5 we will be able to do a problem of size $n = 33.5 \times 10^6$
- Running at 5 Pflop/s the benchmark run will take 2.5 days to complete
- Clearly we have a issue here

We Have to Do Something

- One of the positive aspects of the Linpack Benchmark is that it stresses the system.
- Run only a portion of the run.
- But for how long?
  - 4 hours? 6 hours? 8 hours? 12 hours? 24 hours?
- Have to check the results for numerical accuracy.

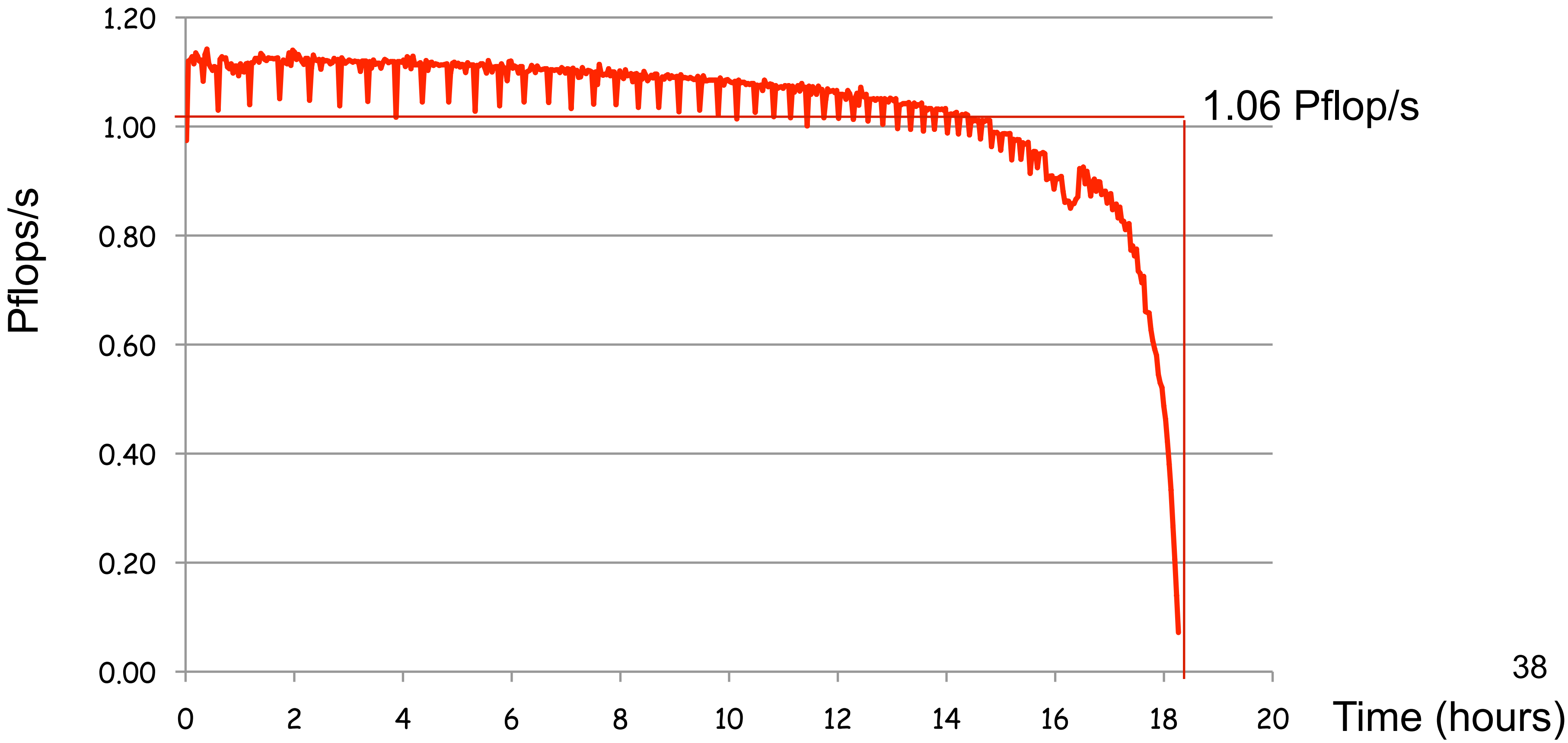
$$\frac{\| Ax - b \|}{(\| A \| \|x\| + \| b \|) n \varepsilon} \approx O(1)$$

Preliminary set of "Ground Rules"

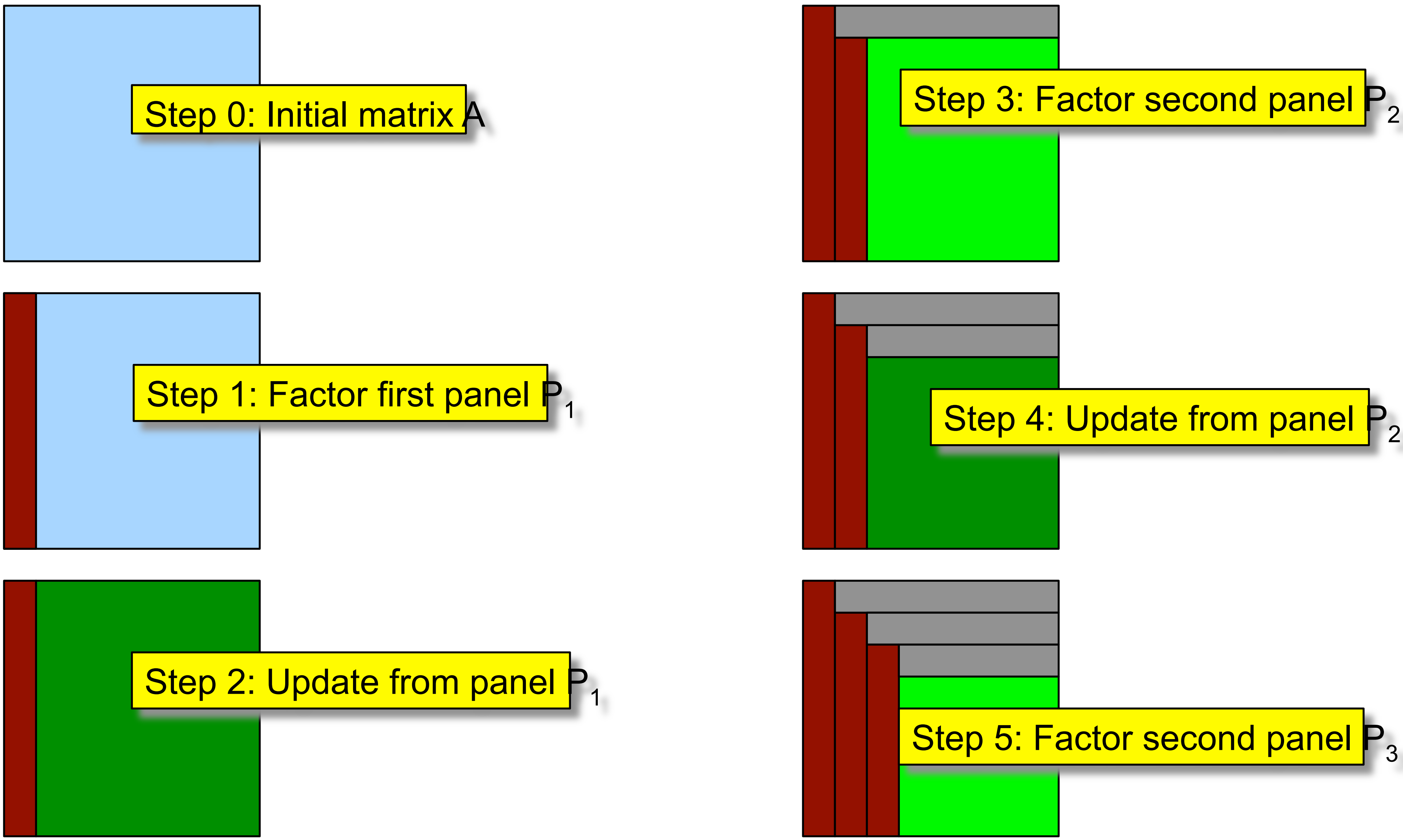
- Whatever is done should be simple to explain and implement.
- The time should still present some challenges, say 12 hours.
  - Stability test
- The results have to be verifiable.
  - Accuracy test
- Even if doing a partial run the full matrix has to be used.
- The rate of execution from the shorten run can never be more than the rate from a complete run.
  - Avoid gaming the benchmark

Over the Course of the Run

- Can't just start the run and stop it after a set amount of time.
- The performance will vary over the course of the run.

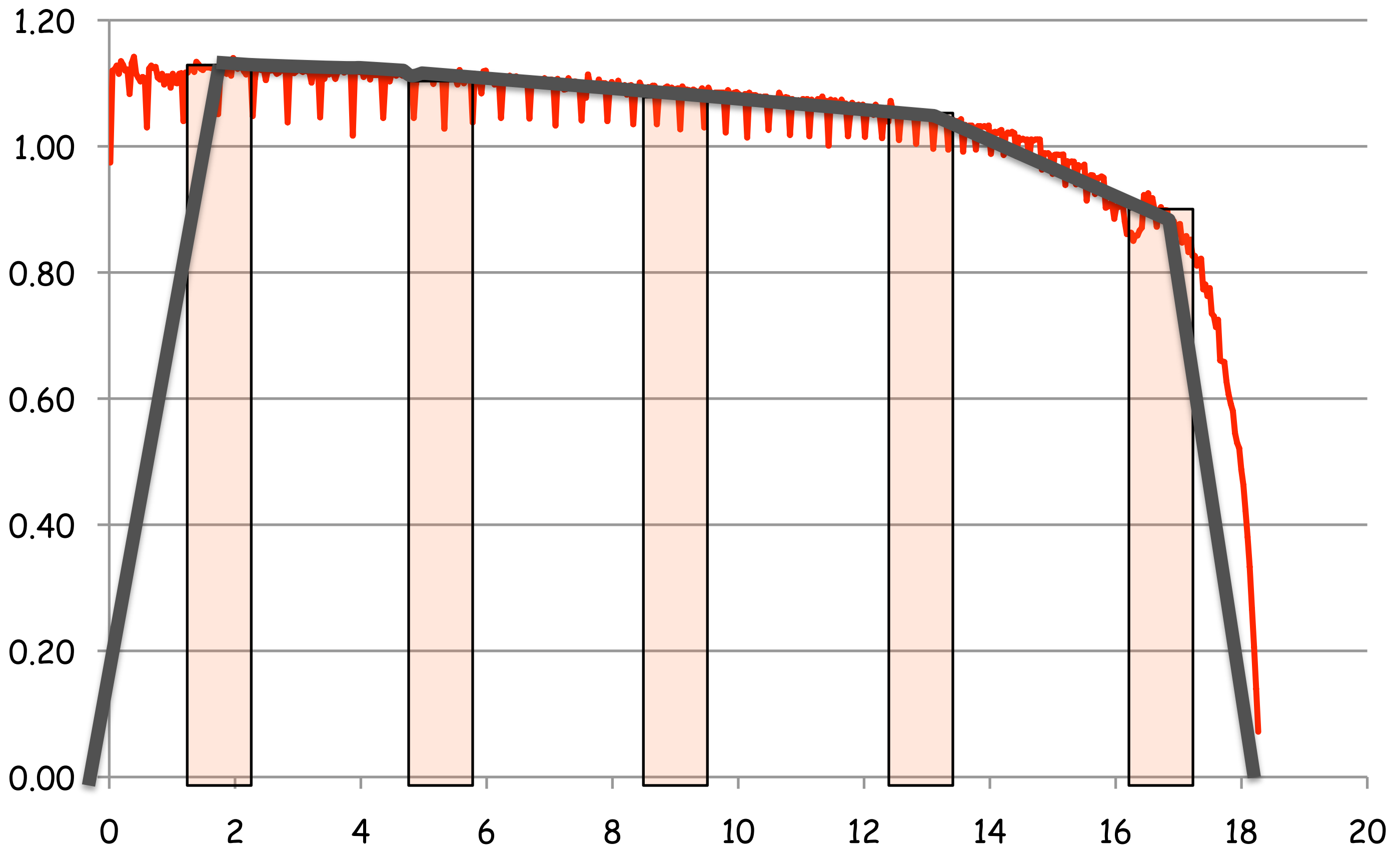


First 5 Steps of LU Factorization



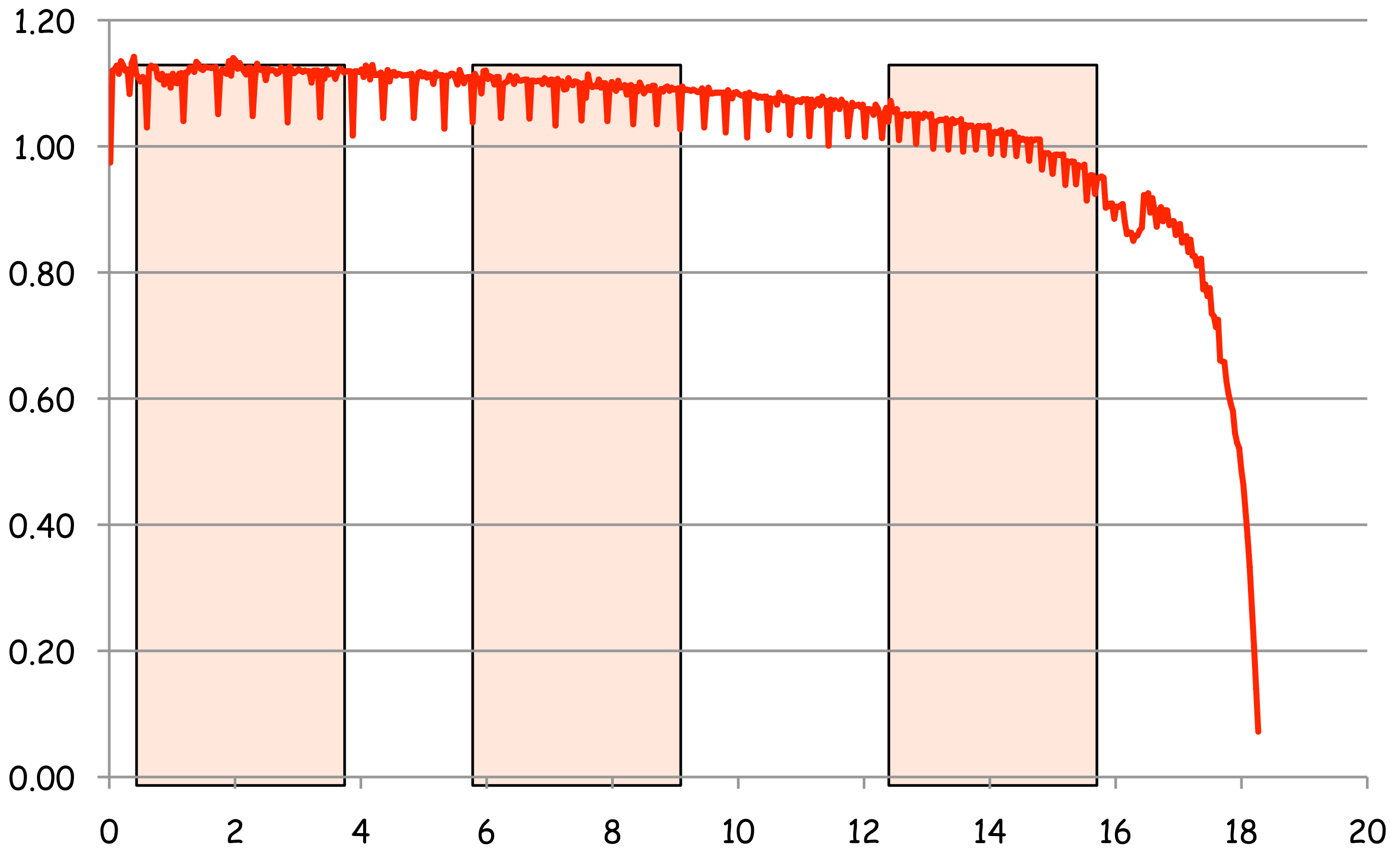
How to Capture Performance?

- **Should we do sampling, and apply quadrature?**



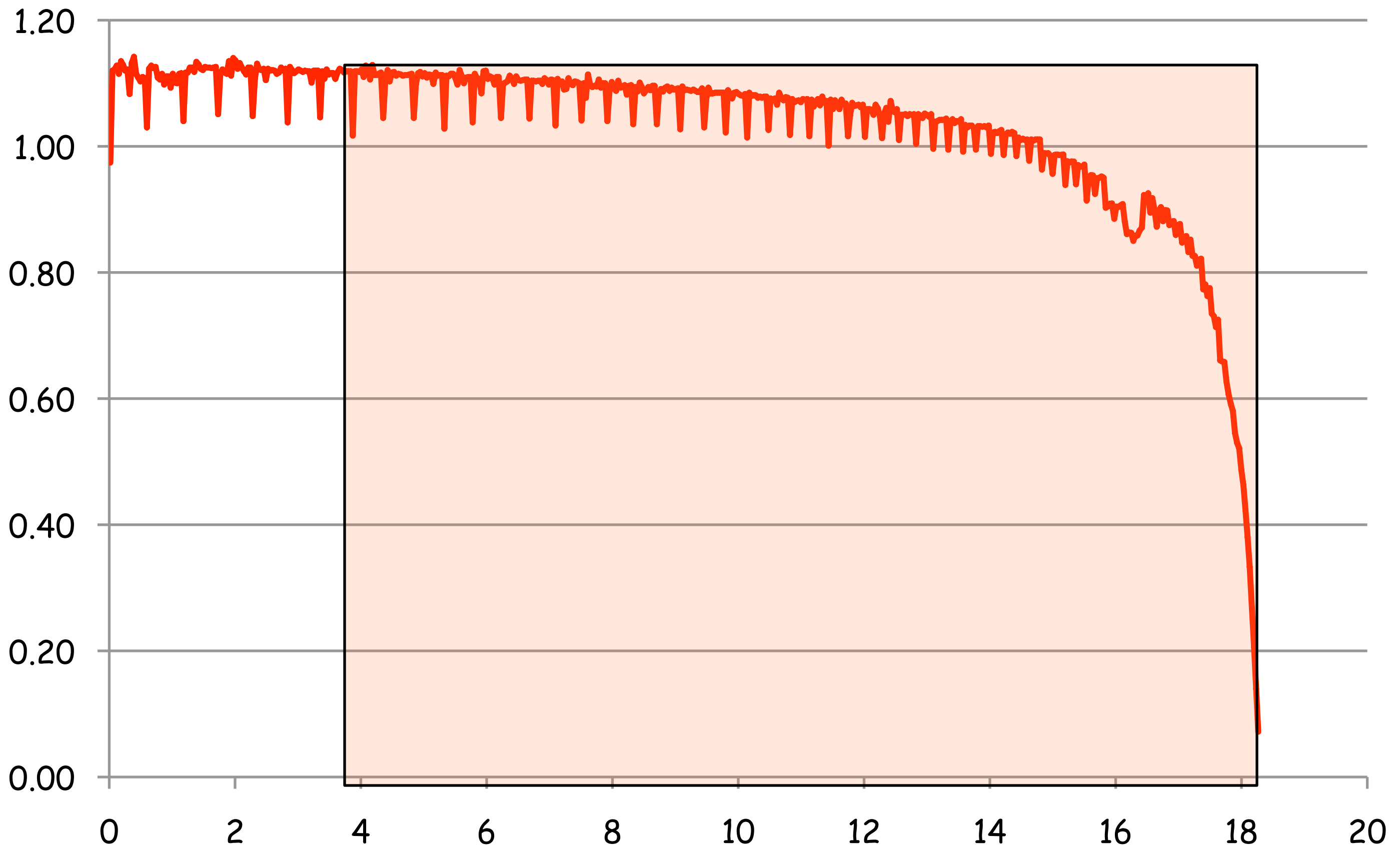
How to Capture Performance?

- Take a window of performance and use it.
- But what window?



How to Capture Performance?

- Figure out the point to start (say what would have been 12 hours into the run) and begin the timing there going to the end.

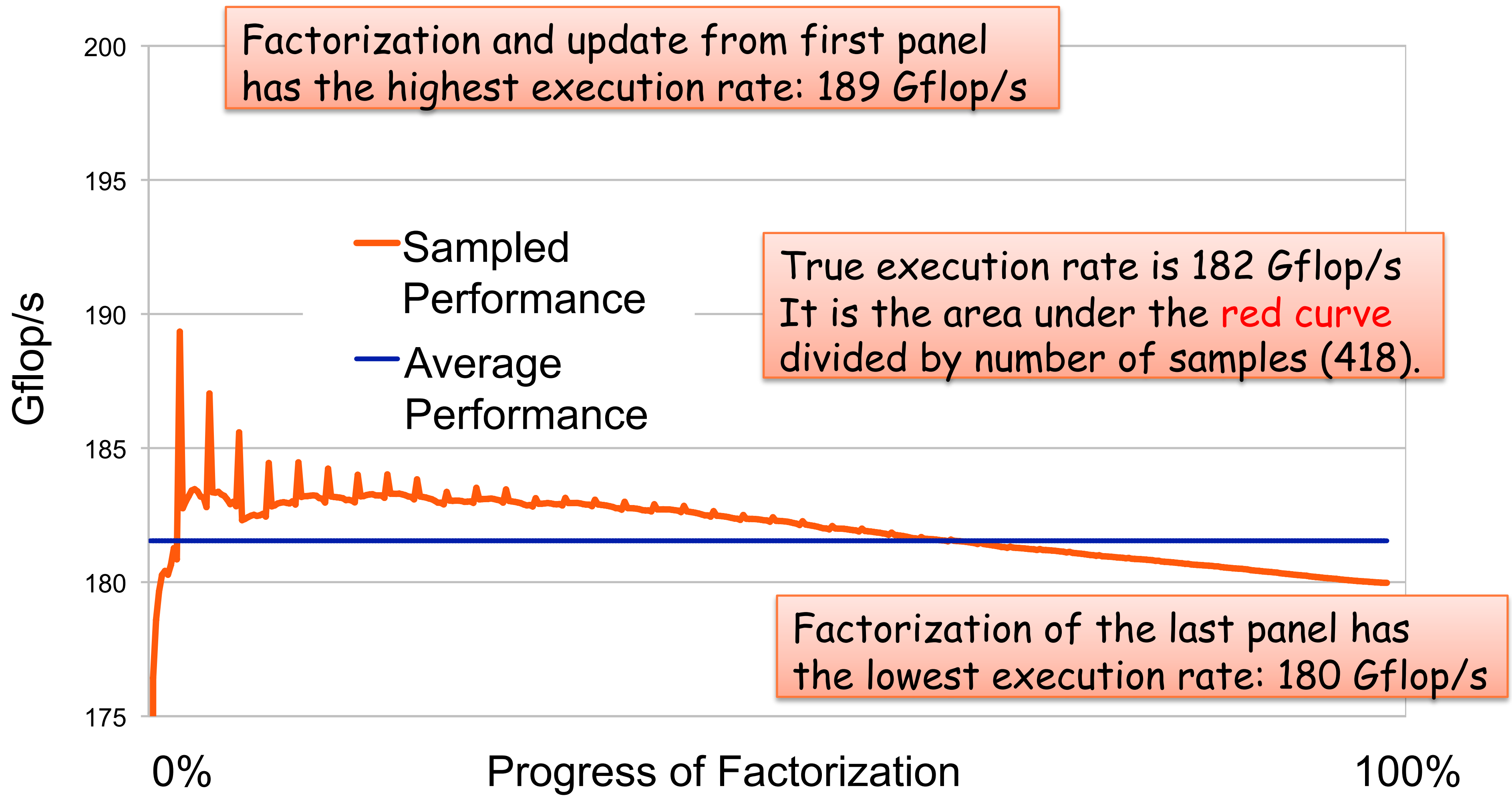


Real Example

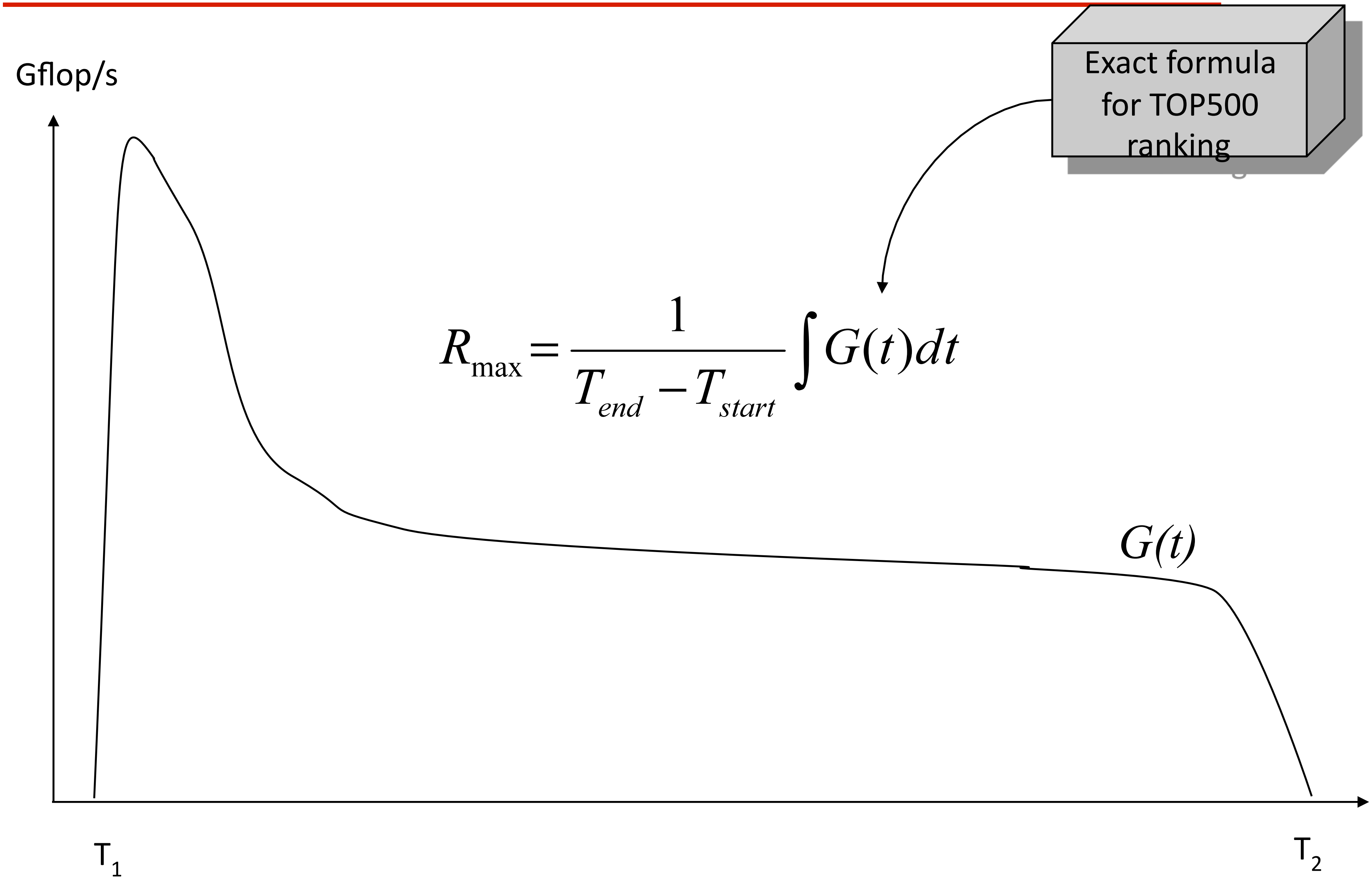
- Matrix size: $N = 50160$
- Block size: $N_B = 120$
- Performance: $^2/_3\ N^3$ / time = 182.135 Gflop/s
- Process grid: 10 by 10
- No. of panels: $N/N_B = 418$
- No. of samples: $N/N_B = 418$

- Each sample is a Gflop/s rate to perform a panel factorization and update
  - for j = 1, 2, 3, ..., 418
    - t = clock()
    - factor_panel(j)
    - update_from_panel(j)
    - t = clock() - t
    - $C = (N-j*N_B+N_B)^3-(N-j*N_B)^3$
    - gflops = $2/3 * C / t * 10^{-9}$
    - print gflops
  - end

LU Factorization Performance over Time

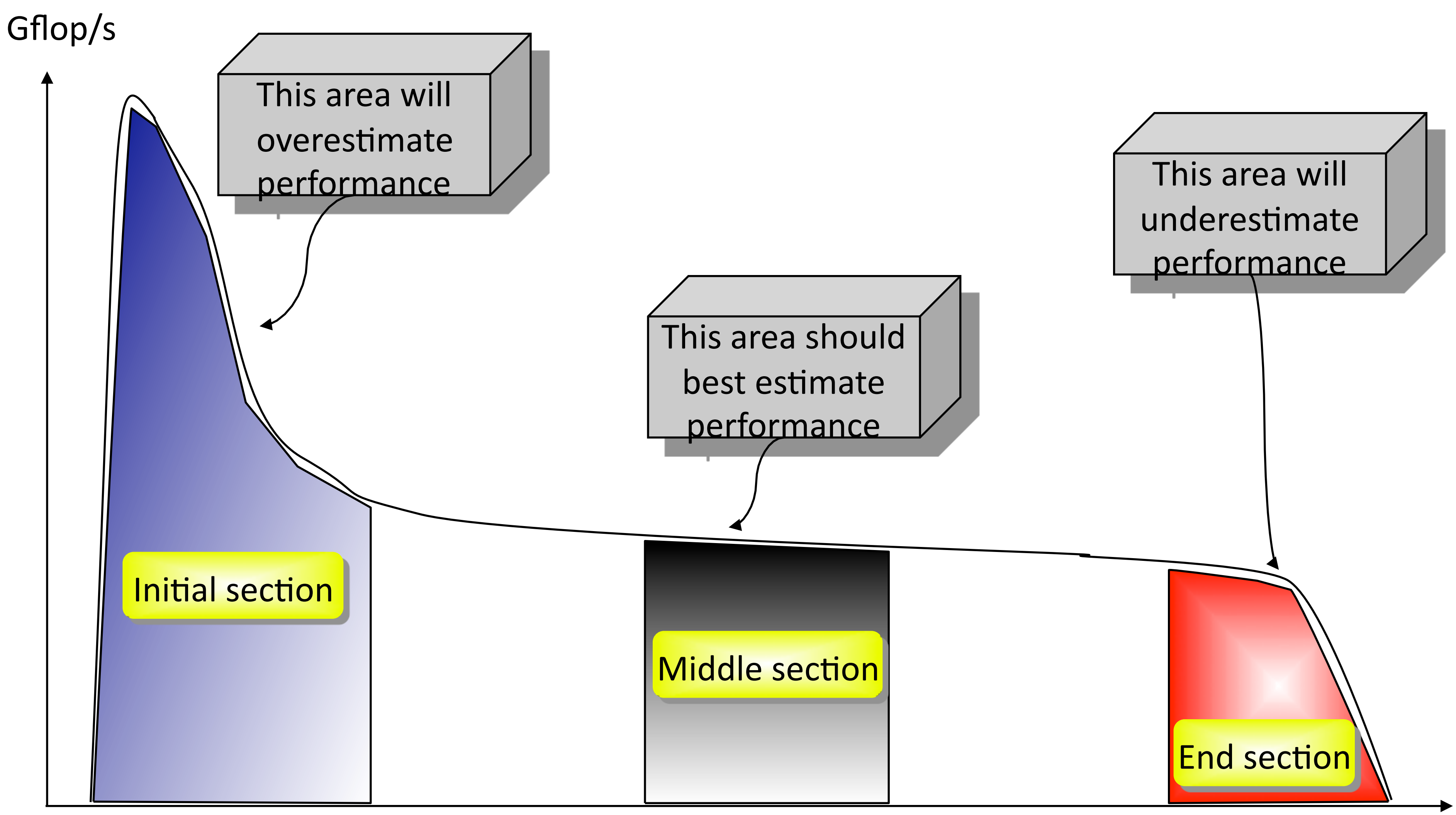


Performance of LINPACK Benchmark Run

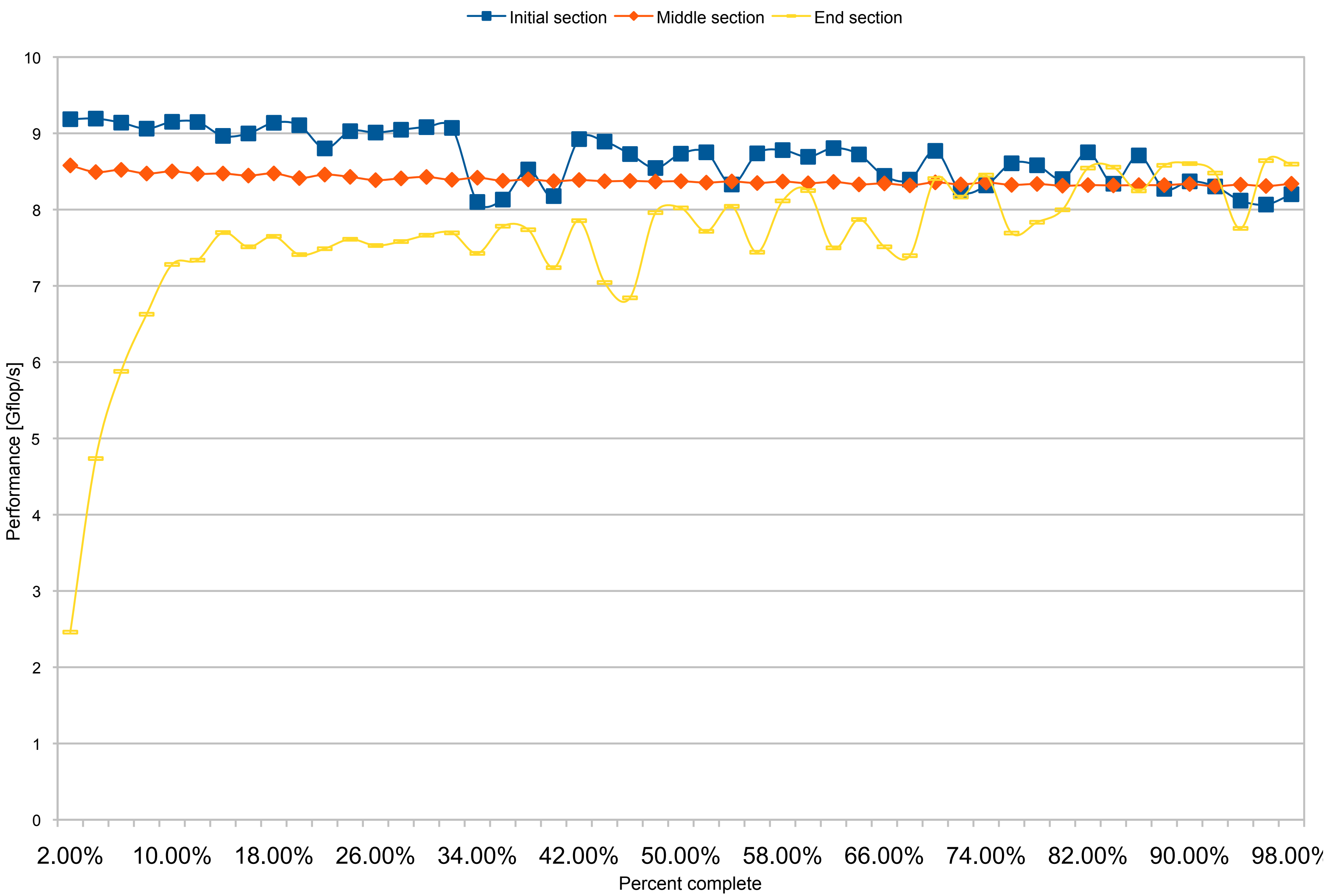


$$R_{\max} = \frac{1}{T_{end} - T_{start}} \int G(t)\,dt$$

Estimating Performance from a Shorter Run



All 3 Sections Compared

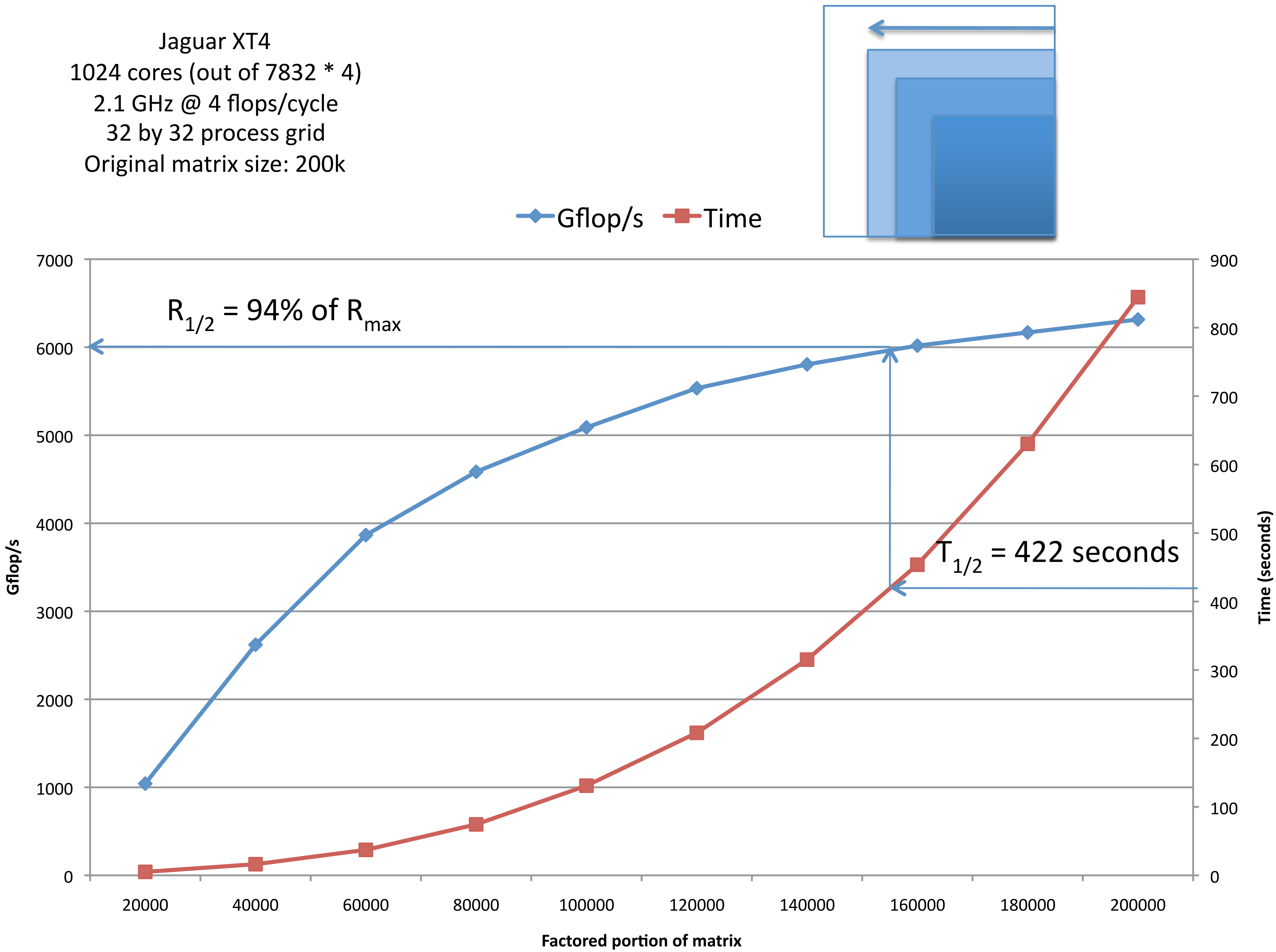


Limited Benchmark Run

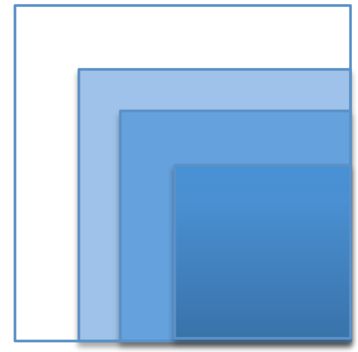
- Start the computation in at some and running to completion.
- Simplified the job of checking the solution.

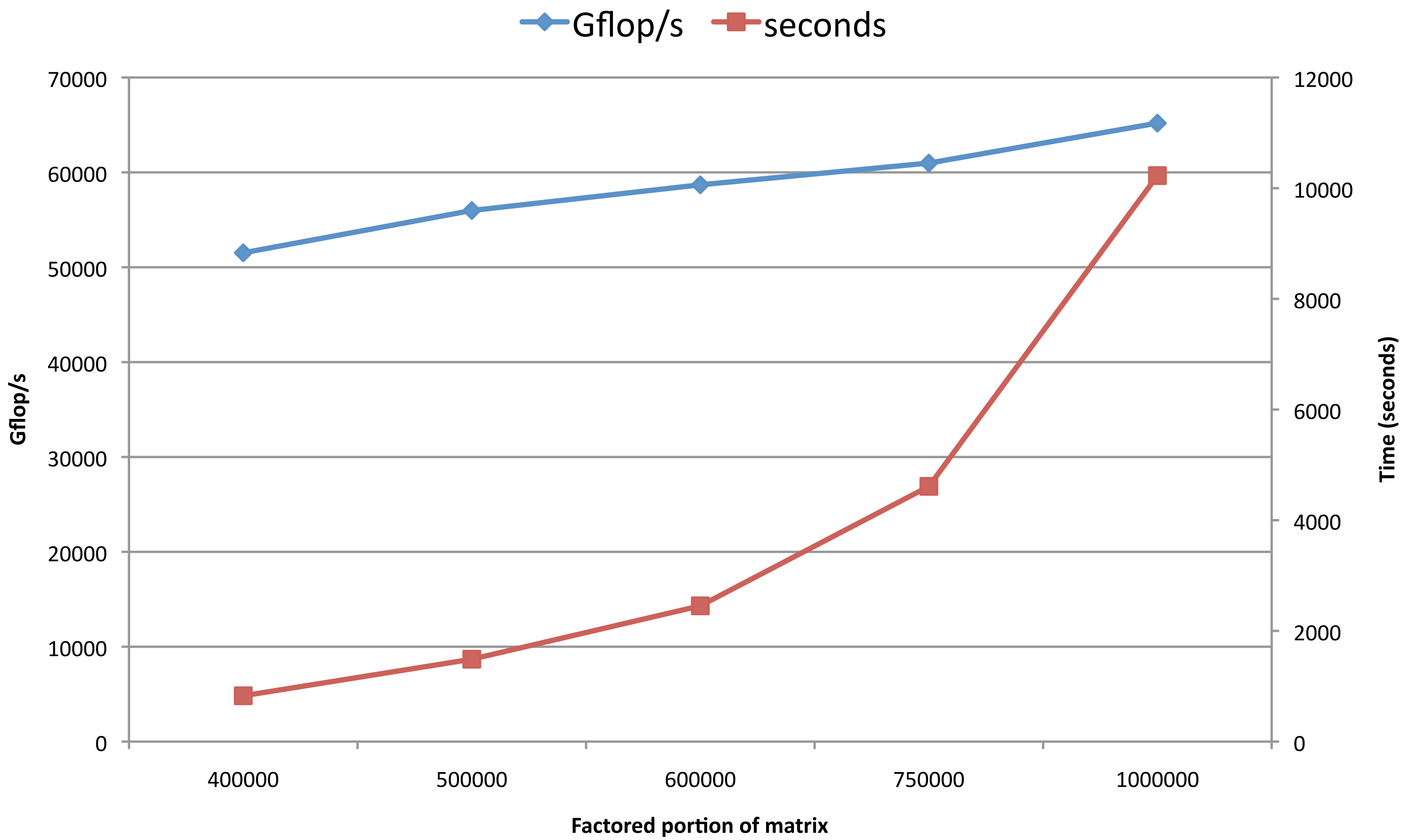
$$A = \begin{bmatrix} I & 0 \\ 0 & A' \end{bmatrix}$$

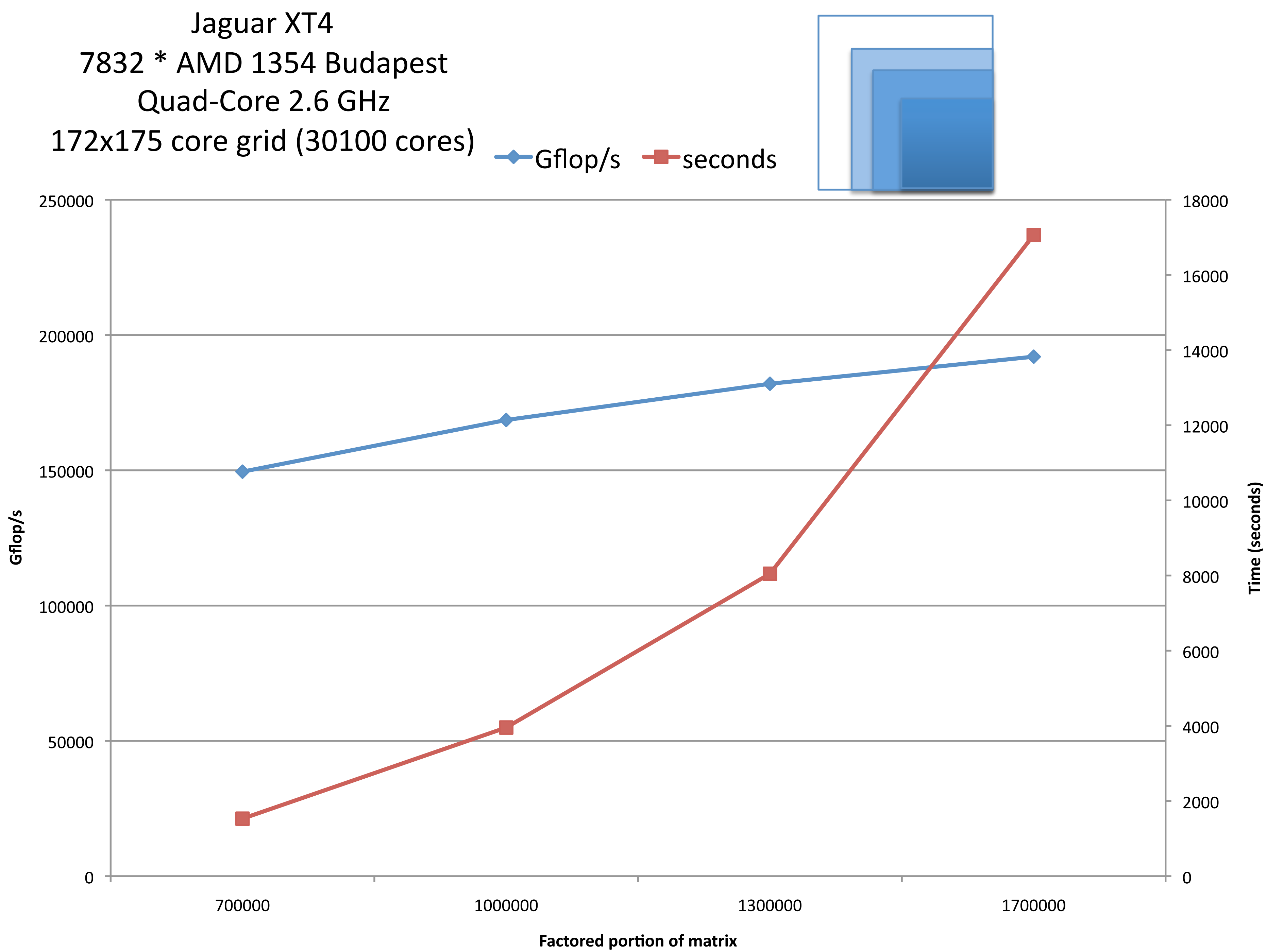
- Easy to Understand and implement.


Jaguar XT4
1024 cores (out of 7832 * 4)
2.1 GHz @ 4 flops/cycle
32 by 32 process grid
Original matrix size: 200k
Gflop/s
Time
$R_{1/2}$ = 94% of R_{max}
$T_{1/2}$ = 422 seconds
Gflop/s
Time (seconds)
Factored portion of matrix
0
1000
2000
3000
4000
5000
6000
7000
0
100
200
300
400
500
600
700
800
900
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000

Jaguar XT4
7832 * AMD 1354 Budapest
Quad-Core 2.6 GHz
100x100 core grid
10,000 cores







Jaguar XT4
7832 * AMD 1354 Budapest
Quad-Core 2.6 GHz
172x175 core grid (30100 cores)
Gflop/s
seconds
250000
200000
150000
100000
50000
0
Gflop/s
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
Time (seconds)
700000
1000000
1300000
1700000
Factored portion of matrix

HPL Summary

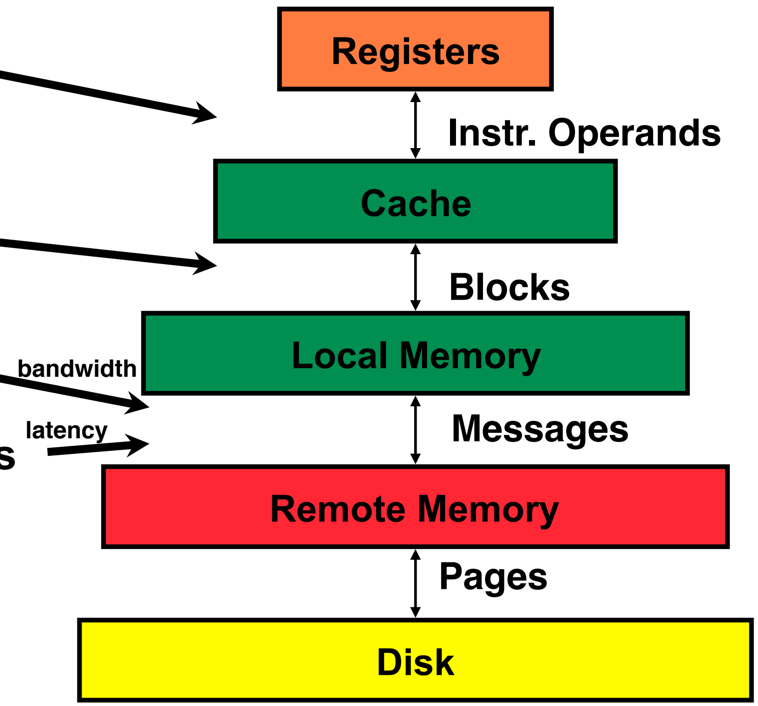
- Making changes to the benchmark should be done very carefully, hard to undo.
- Will continue to experiment with the approximate run.
- Provide a way to estimate time and size.
- Perhaps role this out as beta for November
- Plan for 12 hour max run
  - If your run would be less than 12 hours, then run on the whole matrix.
- Verify the computation
- Approximation rate will be an under approximation
- The longer the testing the more accurate the performance estimate

HPC Challenge Benchmarks for GPUs Next

HPC Challenge Benchmark

- HPL: solves a system
  Ax = b
- STREAM: vector operations
  A = B + s x C
- FFT: 1D Fast Fourier Transform
  Z = FFT(X)
- RandomAccess: random updates
  T(i) = XOR(T(i), r)

Corresponding Memory Hierarchy



- **HPC Challenge measures this hierarchy**