ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

DAY 3: Cray Math Software

Multi-threaded Programming, Tuning and Optimization on Multi-core MPP Platforms

15-17 February 2011

CSCS, Manno

Cray Scientific Libraries :
Usage, hybrid modes, advanced performance

Adrian Tate
Technical Lead of Scientific Libraires
Cray Inc. and Cray Switzerland

CRAY THE SUPERCOMPUTER COMPANY

CSCS Swiss National Supercomputing Centre

HP2C

Goals of this talk

1. Introduce libsci and explain usage
2. Work through the threaded and hybrid library model
3. Show XT5, XT6 and XE6 performance results
4. Hints on obtaining best performance for all libraries
5. Show you how you can get your cases specialized
6. Describe the future of libraries

Structure of the Talk

- Overview
  - Libsci overview history
  - Auto-tuning framework
  - Contents
  - General usage
- Threaded BLAS implementation
- Threaded LAPACK implementation
- ScaLAPACK and hybrid mode Scalapack.
- CASK – tuning PETSc and Trilinos.
- CASE – Cray Adaptive Simplified Eigensolver
- CRAFFT – Cray Adaptive FFT
- The future of libraries

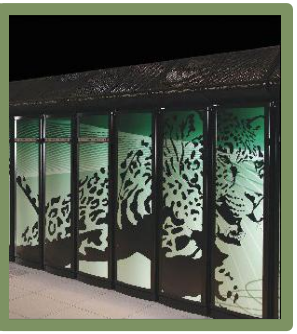
Most descriptions will refer to an example program we can try in the lab

Historical Perspective on Libraries

- Scientific libraries were the first ever productivity feature
- Popular code regions were encapsulated in subroutines
- Programmers of early machine did not need to waste time
- An advantage was that the routine could be tuned heavily
- The performance advantages became increasingly important
- Standards were written for the simplest operations (BLAS1, BLAS2, BLAS3).

History of Cray Supercomputers

1976	1982	1985	1988	1991	1993
					
Cray-1	Cray-XMP	Cray-2	Cray-YMP	Cray-C90	Cray-T3D

1994	1995	2001	2003	2005	2008
					
Cray-T90	Cray-T3E	Cray-SV1	Cray-X1	Cray-XT3	Cray-XT5

CRAY THE SUPERCOMPUTER COMPANY

CSCS Swiss National Supercomputing Centre

HP2C

hardware trends

hardware trends

BLAS1 LINPACK BLAS2 BLAS3 LAPACK

1976 1982 1985 1988 1991 1993

Cray-1 Cray-XMP Cray-2 Cray-YMP Cray-C90 Cray-T90

ScaLAPACK PETSc ATLAS FFTW Trilinos

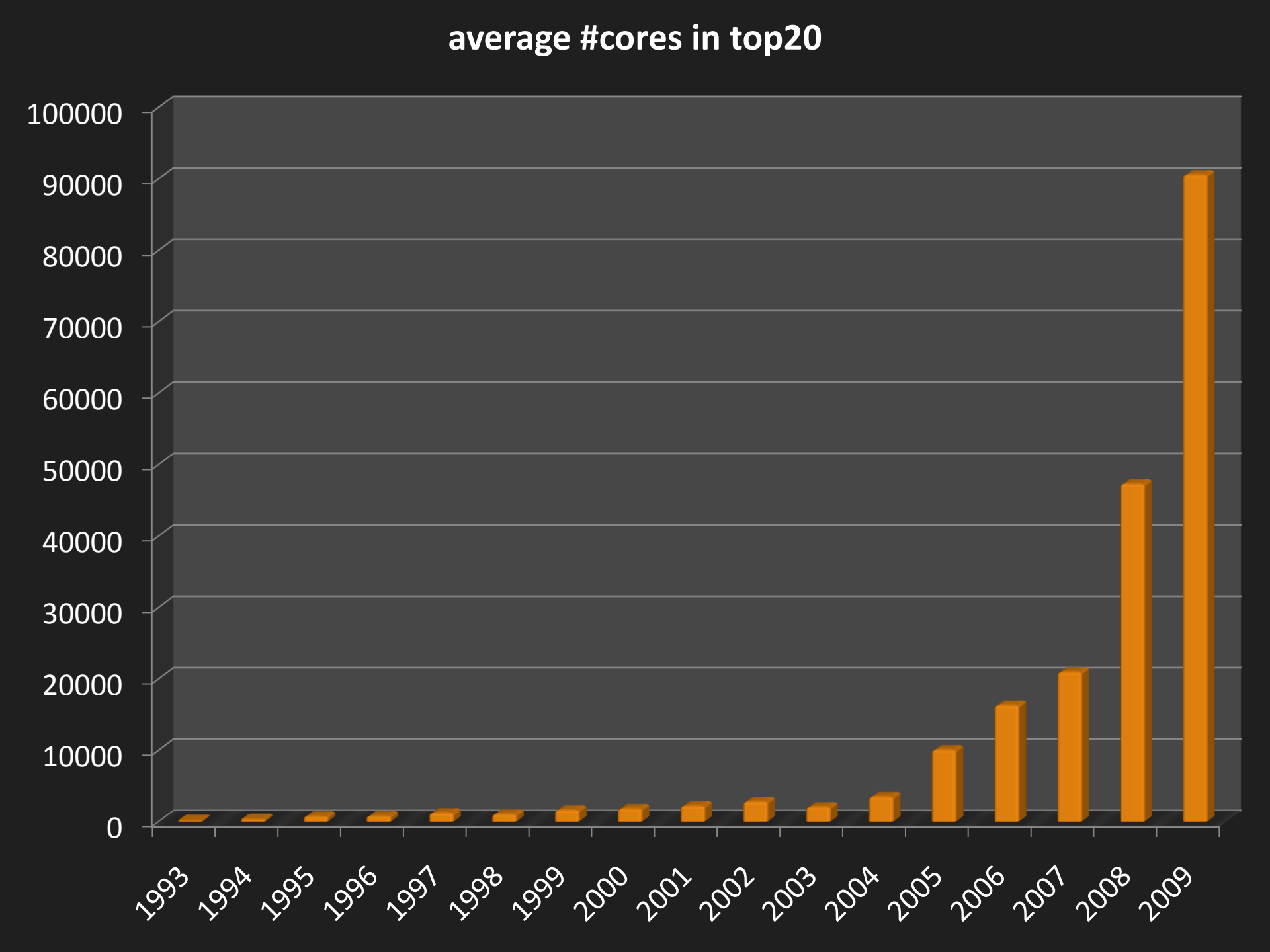
1994 1995 2001 2003 2005 2008

Cray-T3D Cray-T3E Cray-SV1 Cray-X1 Cray-XT3 Cray-XT5

CRAY THE SUPERCOMPUTER COMPANY

CSCS Swiss National Supercomputing Centre

HP2C


average #cores in top20
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009

Clearly…

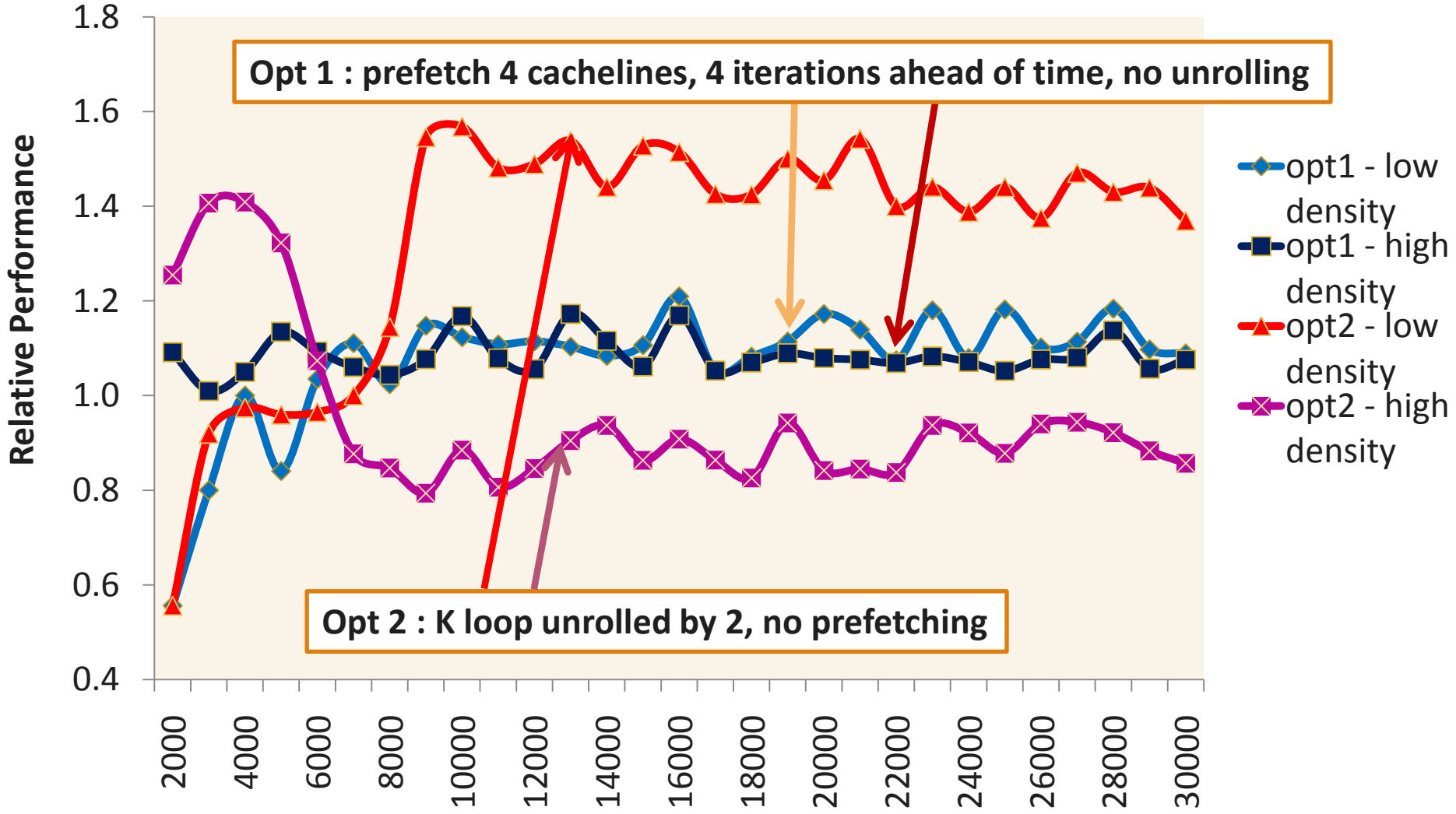
1. Libraries must exhibit multiple layers of parallelism
    1. MPP parallelism
    2. On-node threading parallelism
    3. SIMD vectorization
    4. Accelerator parallelism

2. Despite that libraries must hide the complexity of the system

Less obvious :

1. Libraries must be adaptive and / or auto-tuned
2. The definitions of library APIs must be extended

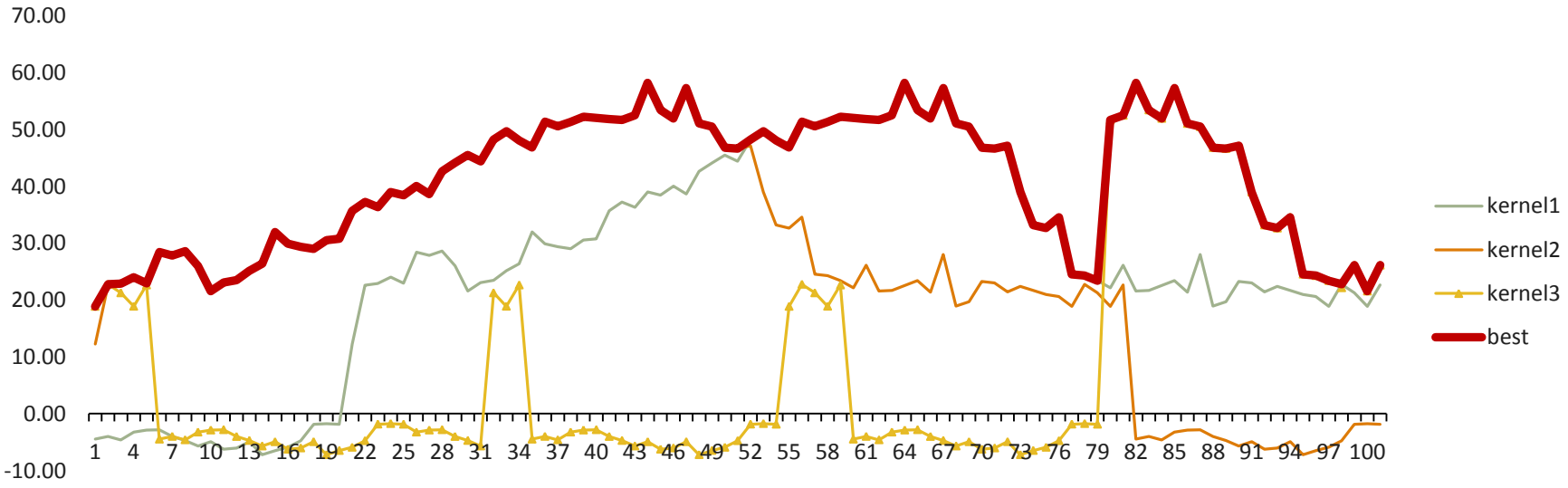
Performance of 2 tuned SpMV kernels relative to BASE case



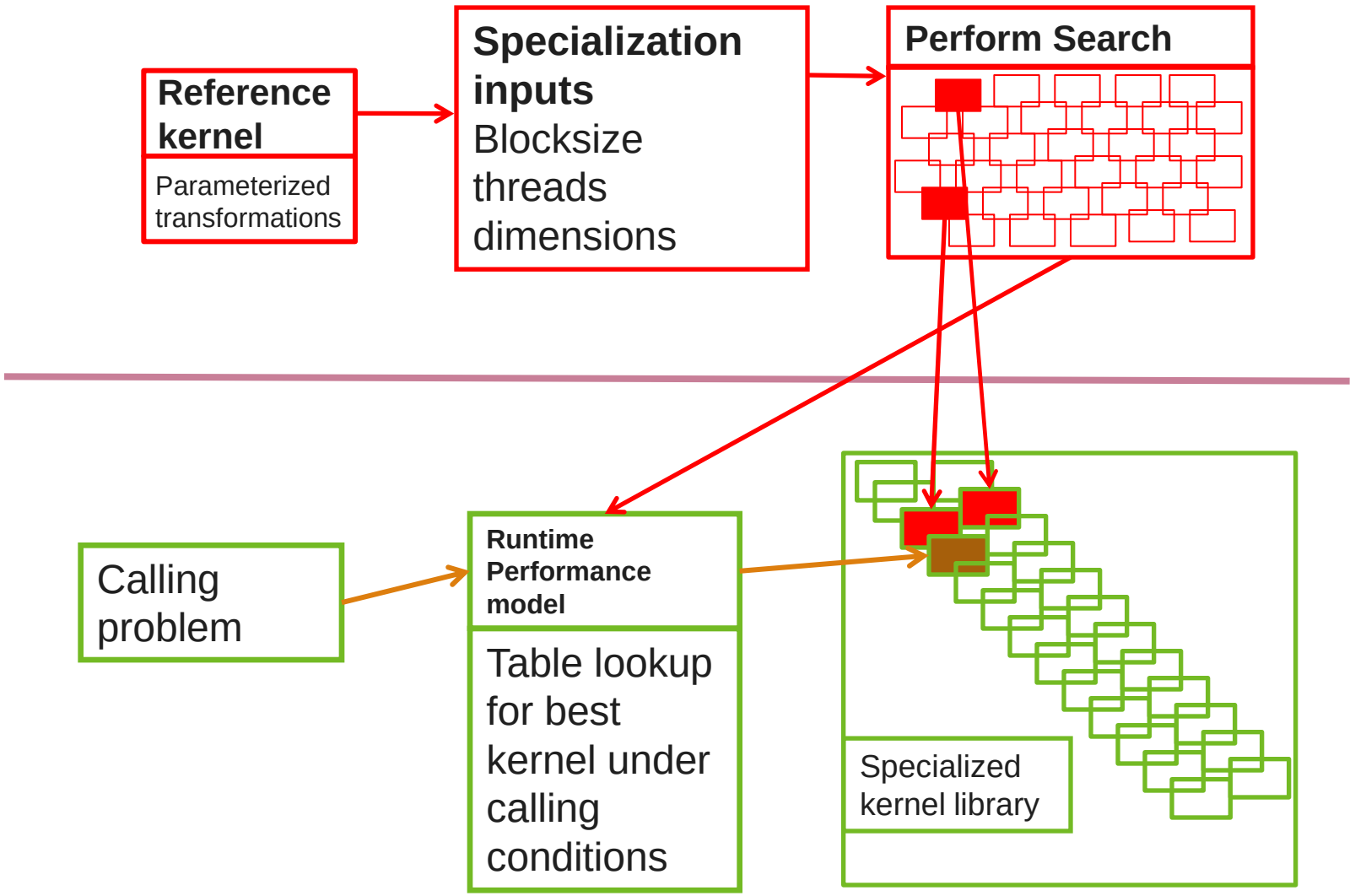
Adaptation

- Adaptation is corresponding concept to auto-tuning
- Given a set of good kernels and a range of input values
  - Map the best or very good kernel onto each combination of input values
  - Need a switching function such that

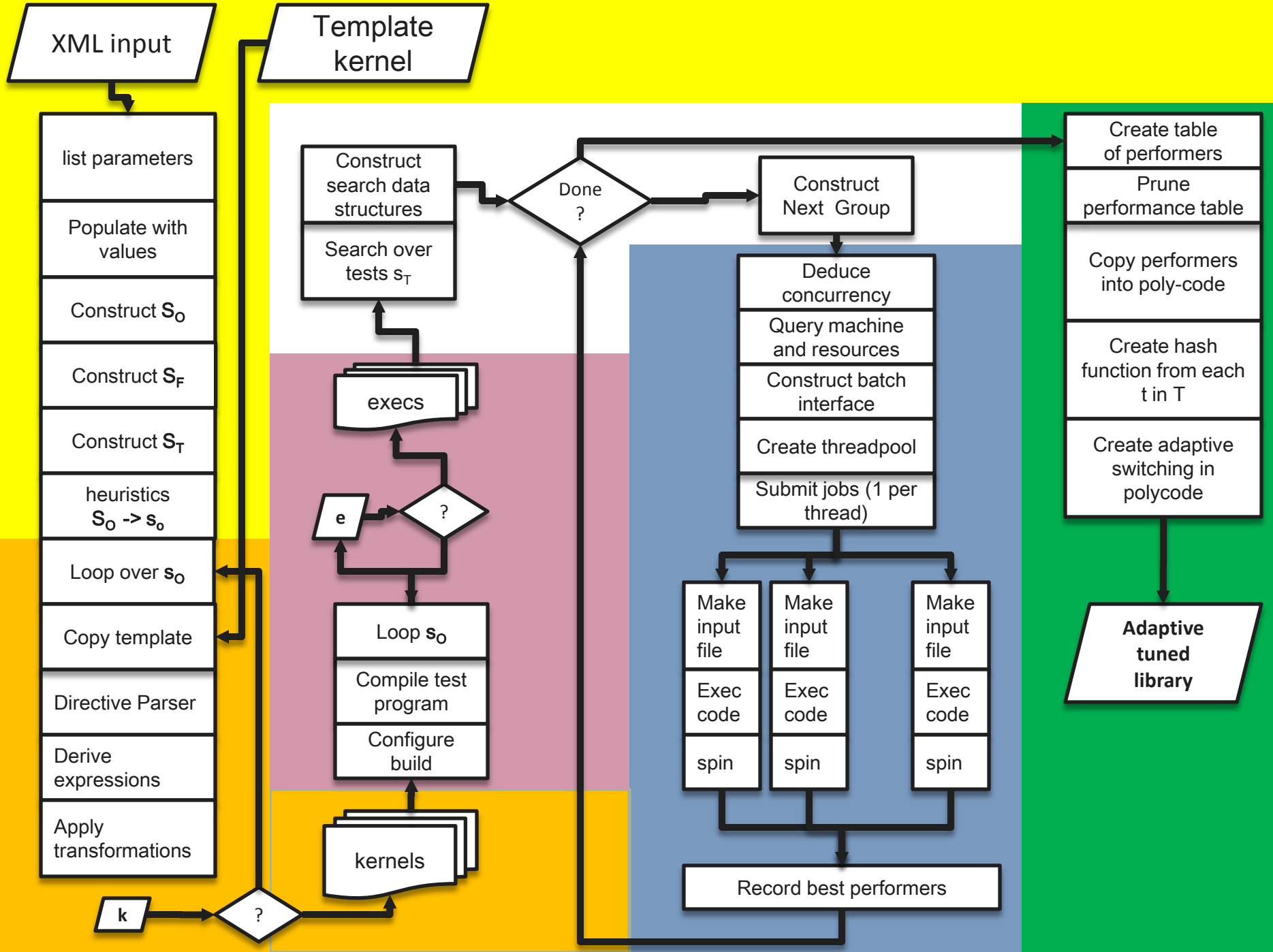
    for any set of testing parameters we obtain the best kernel



Adaptation, Auto-tuning and Specialization



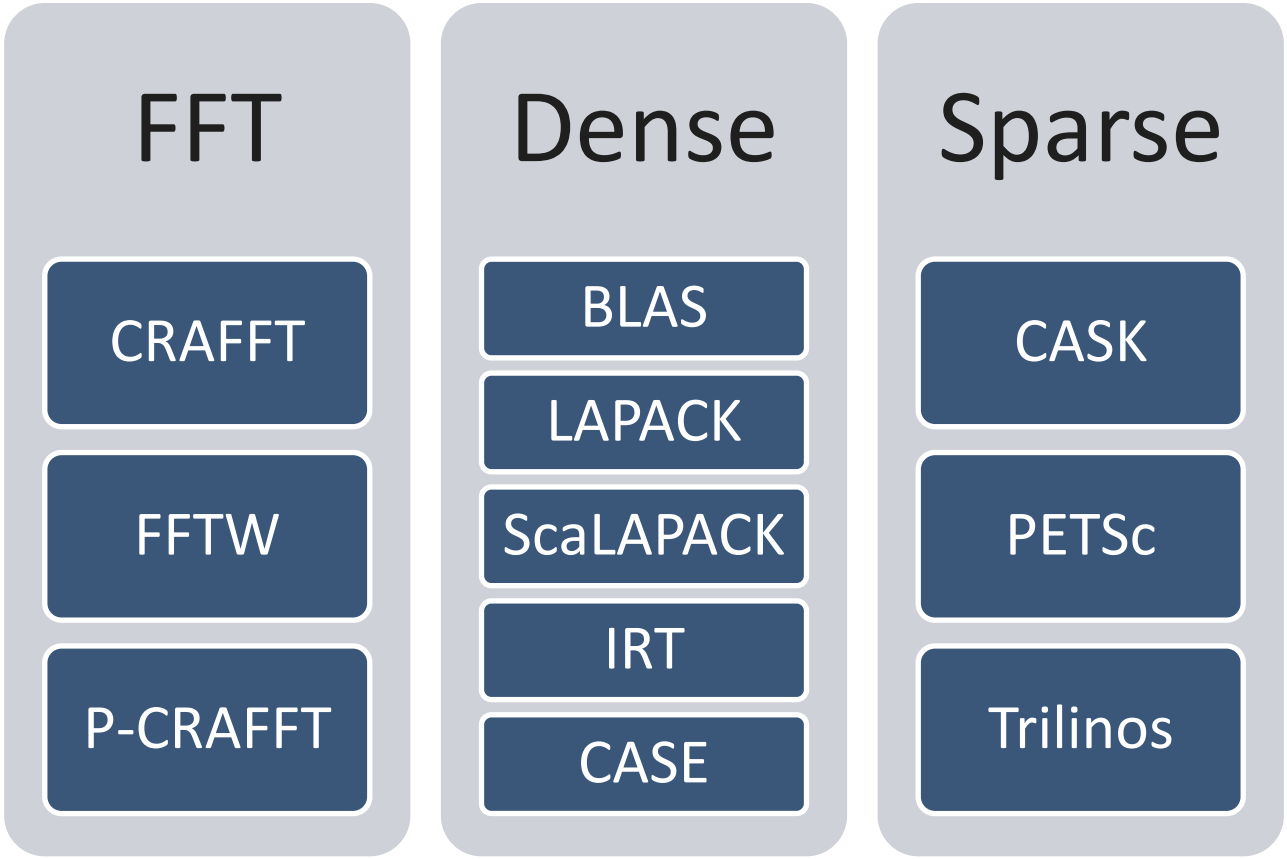
CrayATF is the world's first generalized tuning framework


XML input
Template kernel
list parameters
Populate with values
Construct S_O
Construct S_F
Construct S_T
heuristics S_O -> s_o
Loop over s_O
Copy template
Directive Parser
Derive expressions
Apply transformations
k
?
kernels
Compile test program
Configure build
Loop s_O
execs
e
?
Construct search data structures
Search over tests s_T
Done ?
Construct Next Group
Deduce concurrency
Query machine and resources
Construct batch interface
Create threadpool
Submit jobs (1 per thread)
Make input file
Exec code
spin
Make input file
Exec code
spin
Make input file
Exec code
spin
Record best performers
Create table of performers
Prune performance table
Copy performers into poly-code
Create hash function from each t in T
Create adaptive switching in polycode
Adaptive tuned library

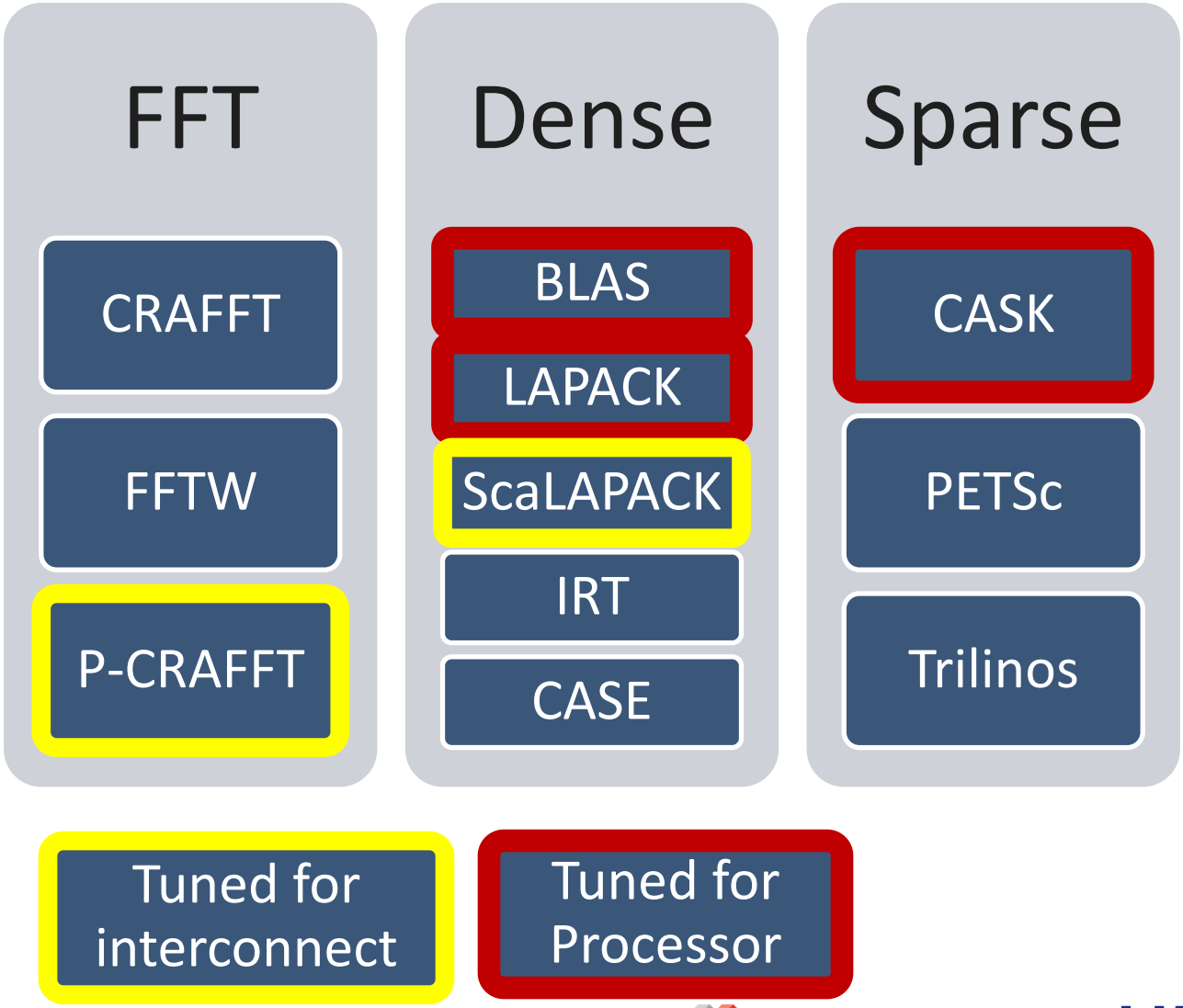
What this framework allows

- Better optimizations
  - CASK, BLAS
- Better flexibility
  - We can tune for highly specialized cases and adapt to them in the libraries
- Easier transitions to new architectures
  - We can re-run the framework when a new platform arrives
- Potential for more powerful tuning tools to users (later)

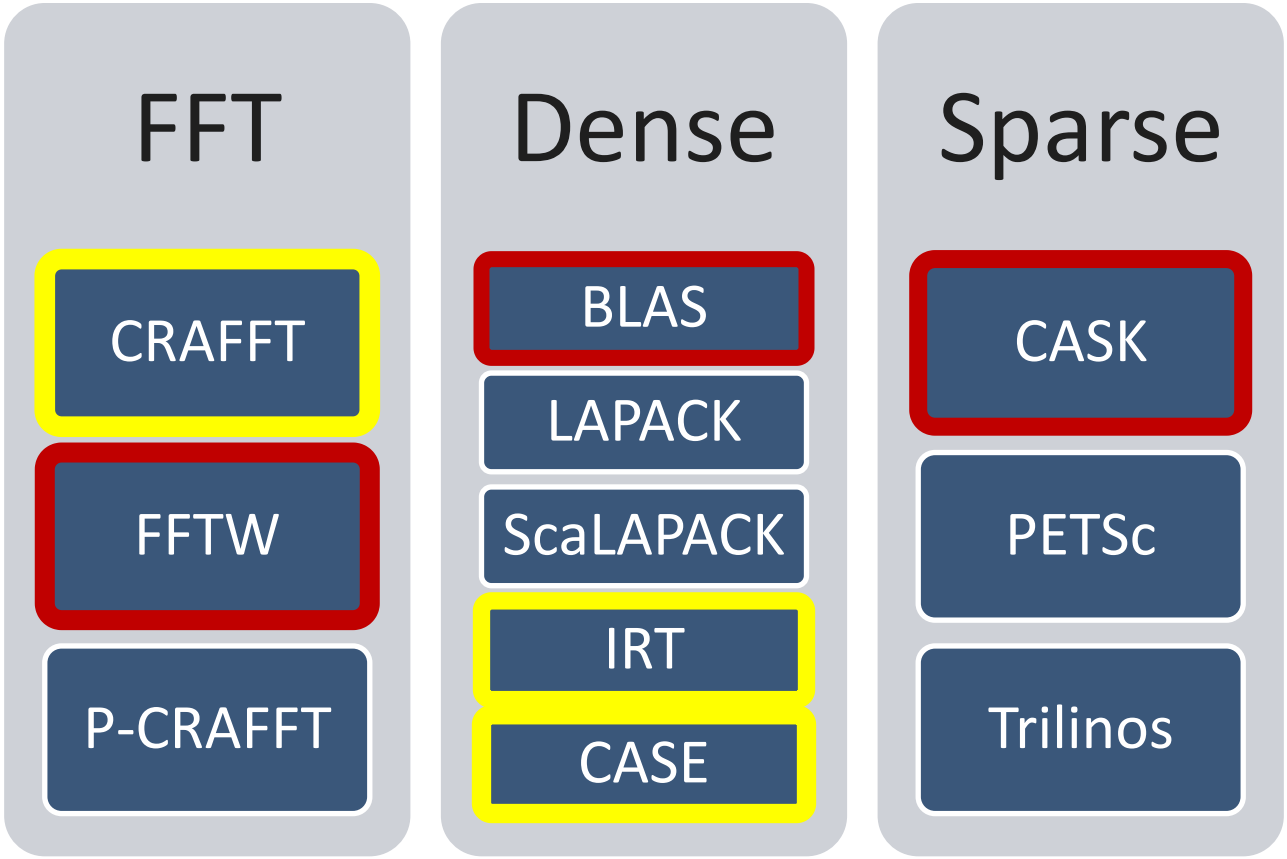
Cray Scientific Libraries



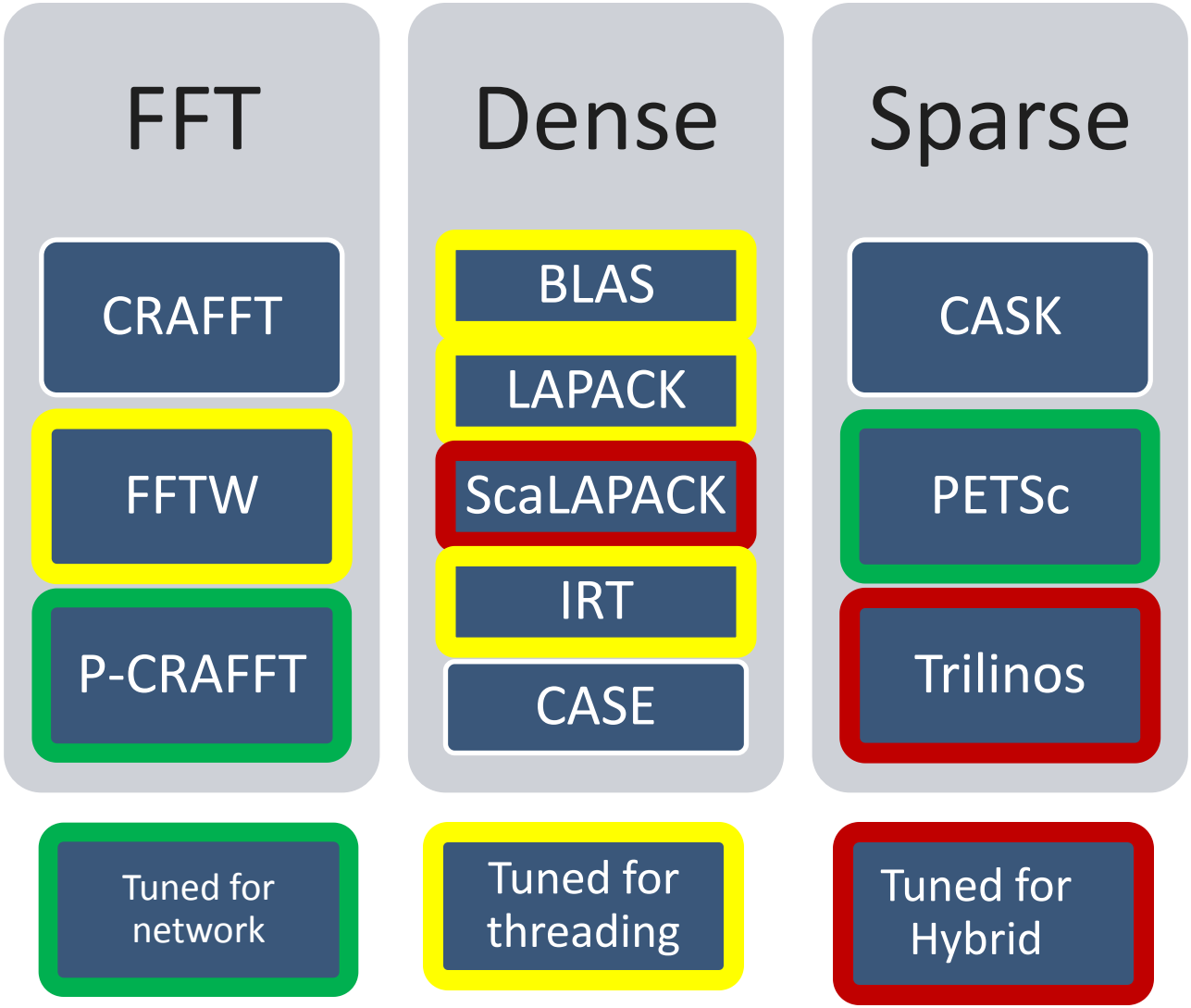
Cray Scientific Libraries - Tunings



Cray Scientific Libraries – autotuning focus



Cray Scientific Libraries – tuning focus



General usage information

- There are many libsci libraries on the systems
- One for each of
  - Compiler (intel, cray, gnu, pathscale, pgi)
  - Single thread, multiple thread
  - Target (istanbul, mc12)
- Best way to use libsci is to ignore all of this
- Load the xtpe-module
  - module load xtpe-mc12 / xtpe-istanbul / xtpe-mc8
- Cray's compiler drivers will link the library automatically
- PETSc, Trilinos, fftw, acml all have their own module

Adding another library

- Perhaps you want to link another library such as ACML

- This can be done. If the library is provided by Cray, then load the module. The link will be performed with the libraries in the correct order.
- If the library is not provided by Cray and has no module, add it to the link line.
  - Items you add to the explicit link will be in the correct place
- Note, to get explicit BLAS from ACML but scalapack from libsci
  - Load acml module. Explicit calls to BLAS in code resolve from ACML
  - BLAS calls from the scalapack code will be resolved from libsci (no way around this)

Making sure you have the right library

- I recommend adding options to the linker to make sure you have the correct library loaded.

- -Wl adds a command to the linker from the driver

- You can ask for the linker to tell you where an object was resolved from using the –y option.

- E.g. –Wl, -ydgemm_
- Will return :

  cc -L./ -o mmulator blas_test.o netlib_dgemm.o -Wl,-ydgemm_

  blas_test.o: reference to dgemm_

  /opt/xt-libsci/10.4.9/cray/lib/libsci.a(dgemm.o): definition of dgemm_

OpenMP BLAS

- Threading capabilities in previous libsci versions were poor
  - Used PTHREADS (more explicit affinity etc)
  - Required explicit linking to a _mp version of libsci
  - Was a source of concern for some applications that need hybrid performance and interoperability with openMP
- LibSci 10.4.2 February 2010
  - OpenMP-aware LibSci
  - Allows calling of BLAS inside or outside parallel region
  - Single library supported (there is still a single thread lib)
- Usage – load the xtpe module for your system (mc12)

GOTO_NUM_THREADS outmoded – use OMP_NUM_THREADS

OpenMP LibSci

- Allows seamless calling of the BLAS within or without a parallel region

e.g. OMP_NUM_THREADS = 12

call dgemm(…) threaded dgemm is used with 12 threads

!$OMP PARALLEL DO

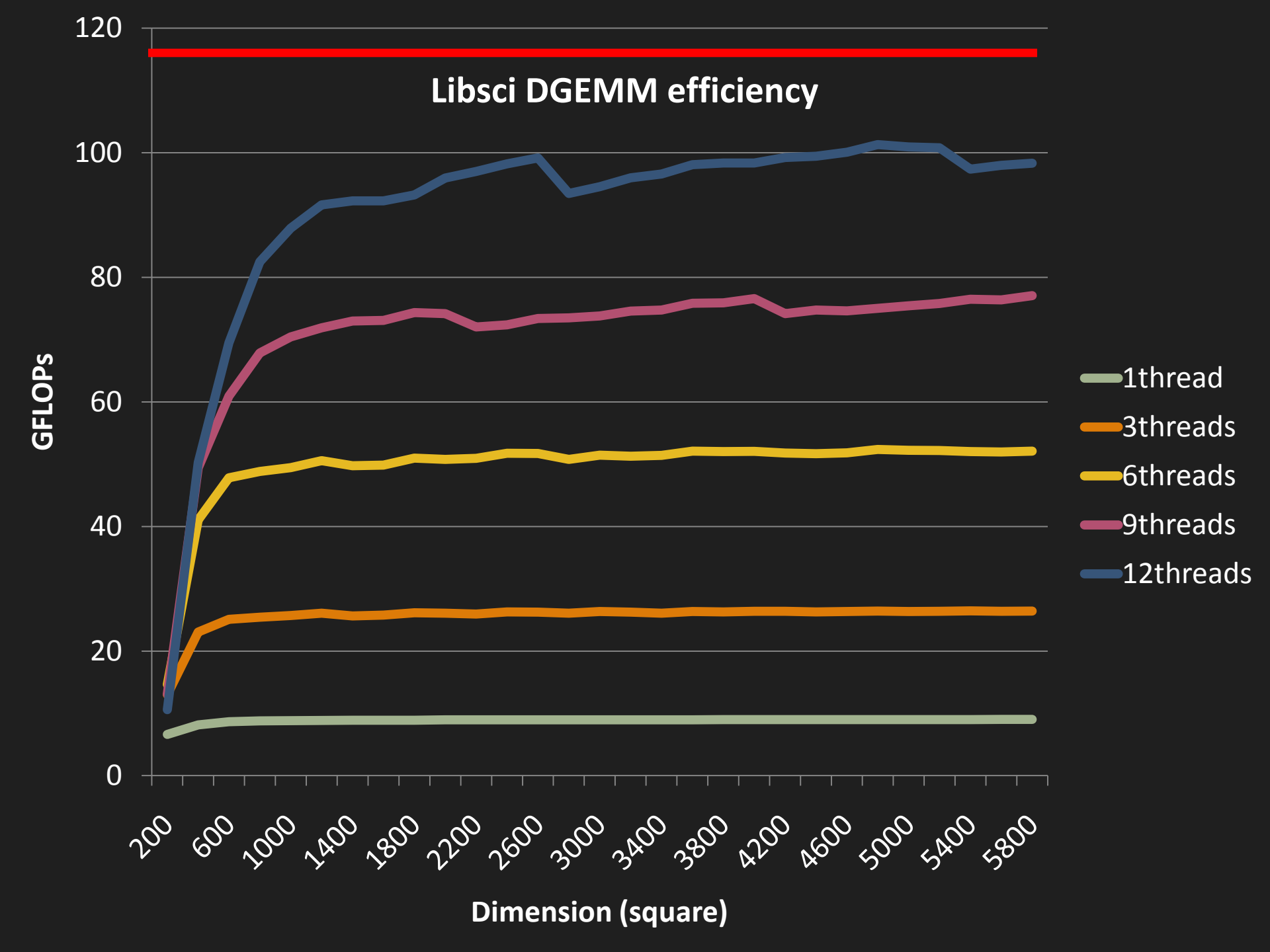
do

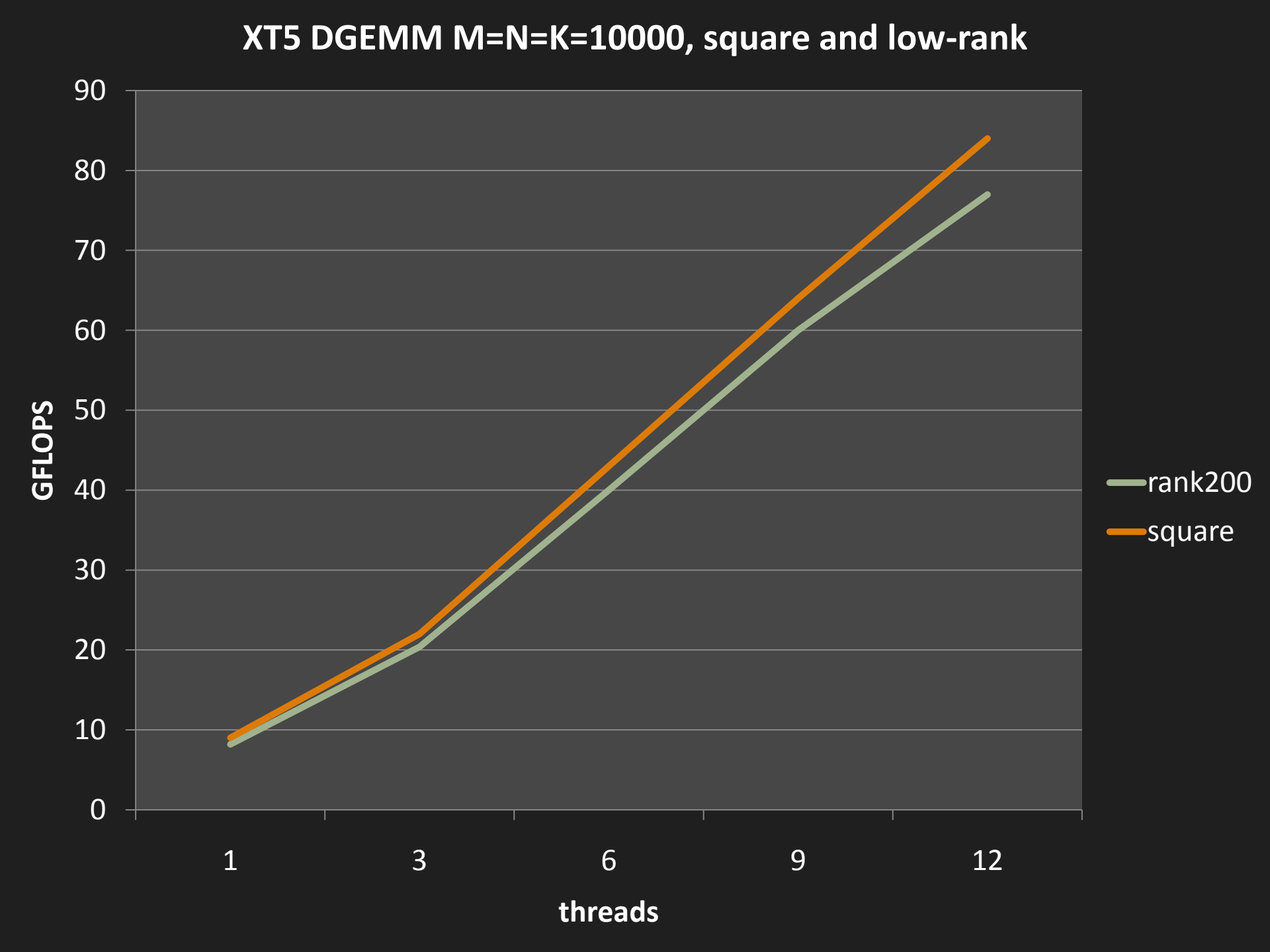
  call dgemm(…) single thread dgemm is used

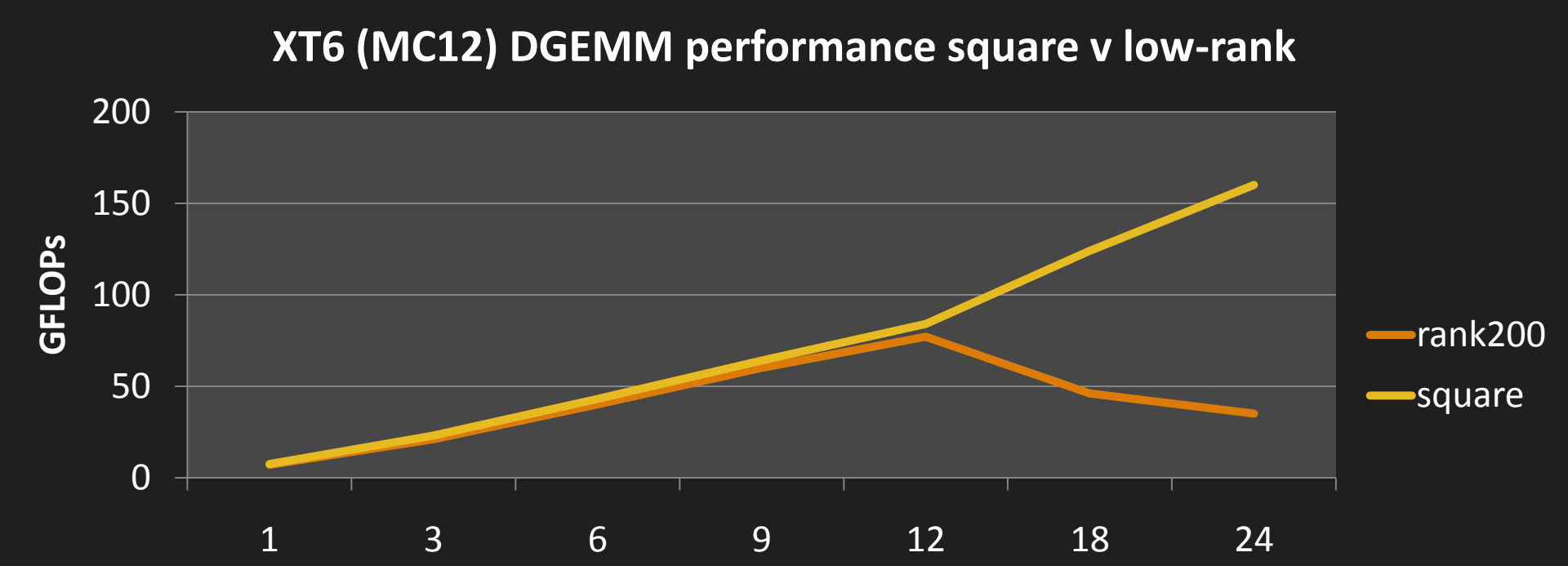
end do

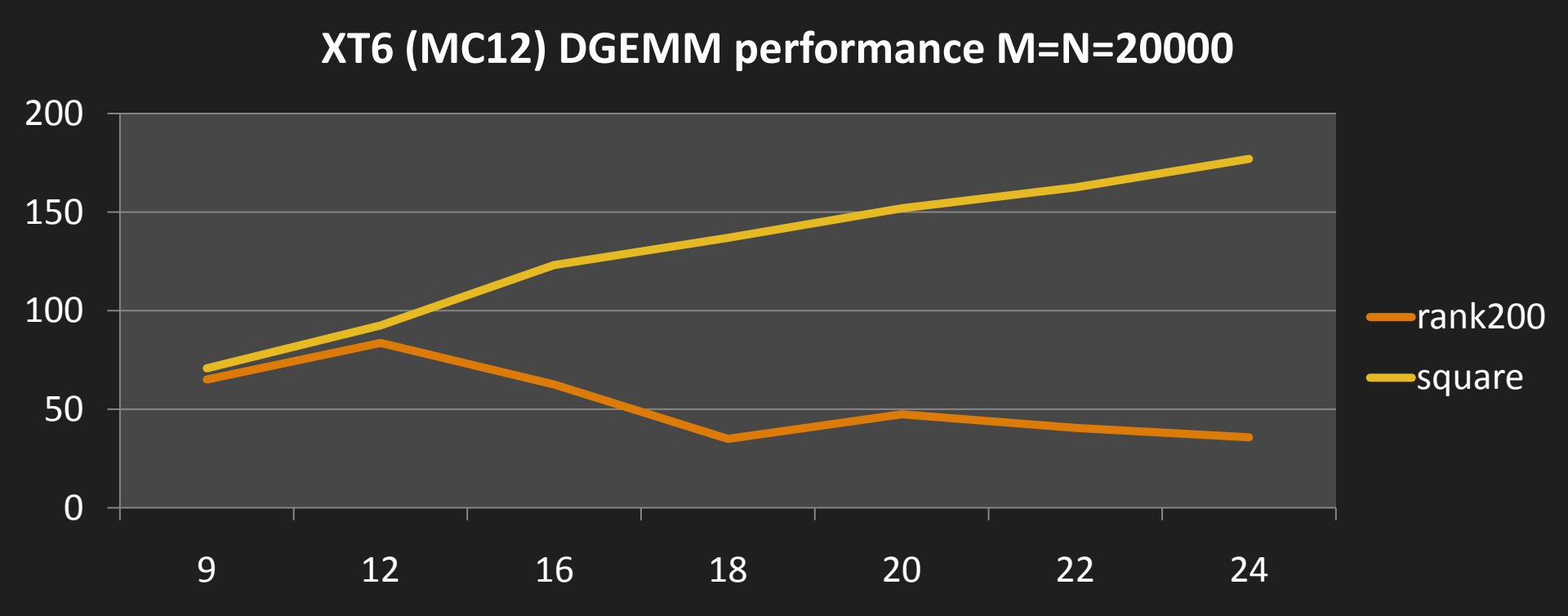
Other situations

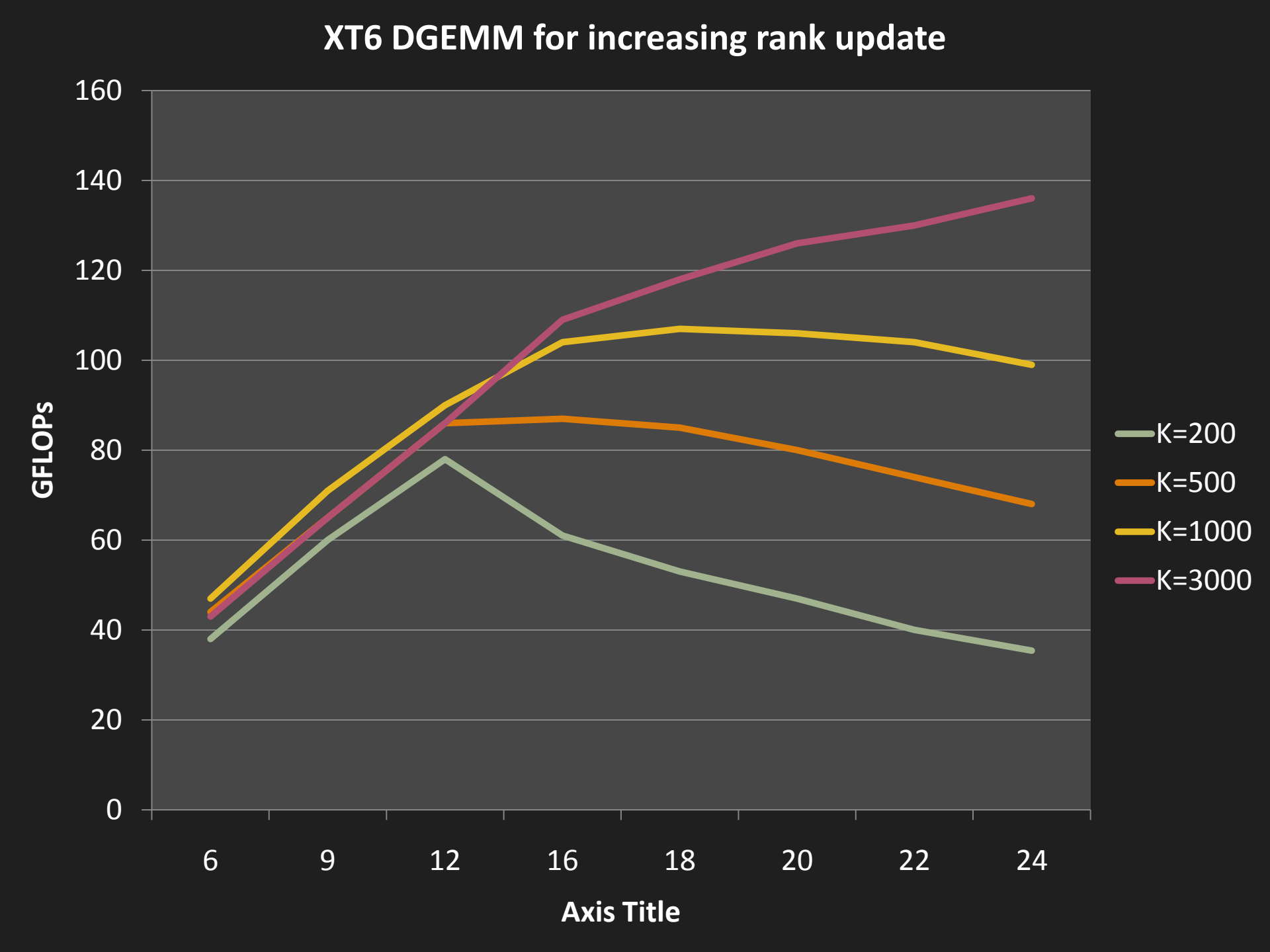
- OMP_NUM_THREADS controls both types of parallelism
- Library sets buffers based on OMP_NUM_THREADS on first call
- The side effect to this model it is not possible to have ‘split-parallelism’
- Changing dynamically OMP_SET_NUM_THREADS is not possible!
- We are working on a more flexible scheme for release early 2011


Libsci DGEMM efficiency
GFLOPs
120
100
80
60
40
20
0
200
600
1000
1400
1800
2200
2600
3000
3400
3800
4200
4600
5000
5400
5800
Dimension (square)
1thread
3threads
6threads
9threads
12threads


XT5 DGEMM M=N=K=10000, square and low-rank
GFLOPS
90
80
70
60
50
40
30
20
10
0
1
3
6
9
12
threads
rank200
square


XT6 (MC12) DGEMM performance square v low-rank
GFLOPs
200
150
100
50
0
1
3
6
9
12
18
24
rank200
square


XT6 (MC12) DGEMM performance M=N=20000
200
150
100
50
0
9
12
16
18
20
22
24
rank200
square


XT6 DGEMM for increasing rank update
160
140
120
100
80
60
40
20
0
GFLOPs
6
9
12
16
18
20
22
24
Axis Title
K=200
K=500
K=1000
K=3000

BLAS2 and BLAS1 performance

- Memory-bound code doesn't thread well.
- But, you can still obtain a little speed-up because you use more memory channels when you use threads.
- Some of the BLAS2 can exhibit some speed-up with threading
- In the lab : benchmark the times for DDOT – can we observe any speed-up using openMP?

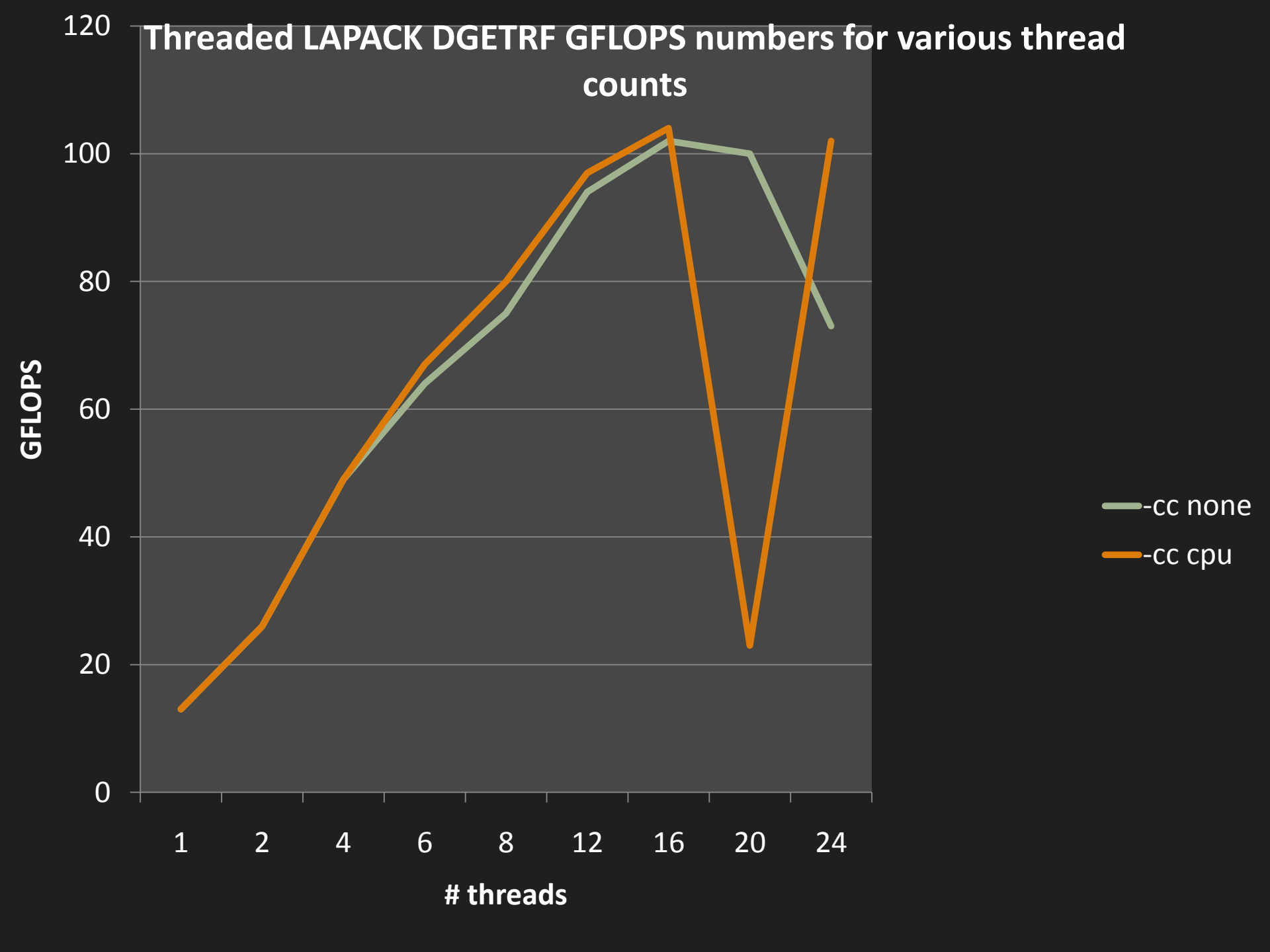
Usage

- module load xtpe-mc12
- No need to explicit link
- Add –Wl,-ydgemm_ to link line
- Set OMP_NUM_THREADS in job script
- Run with aprun –n 1 –d12 ./exec (for 12 threads)

Threaded LAPACK

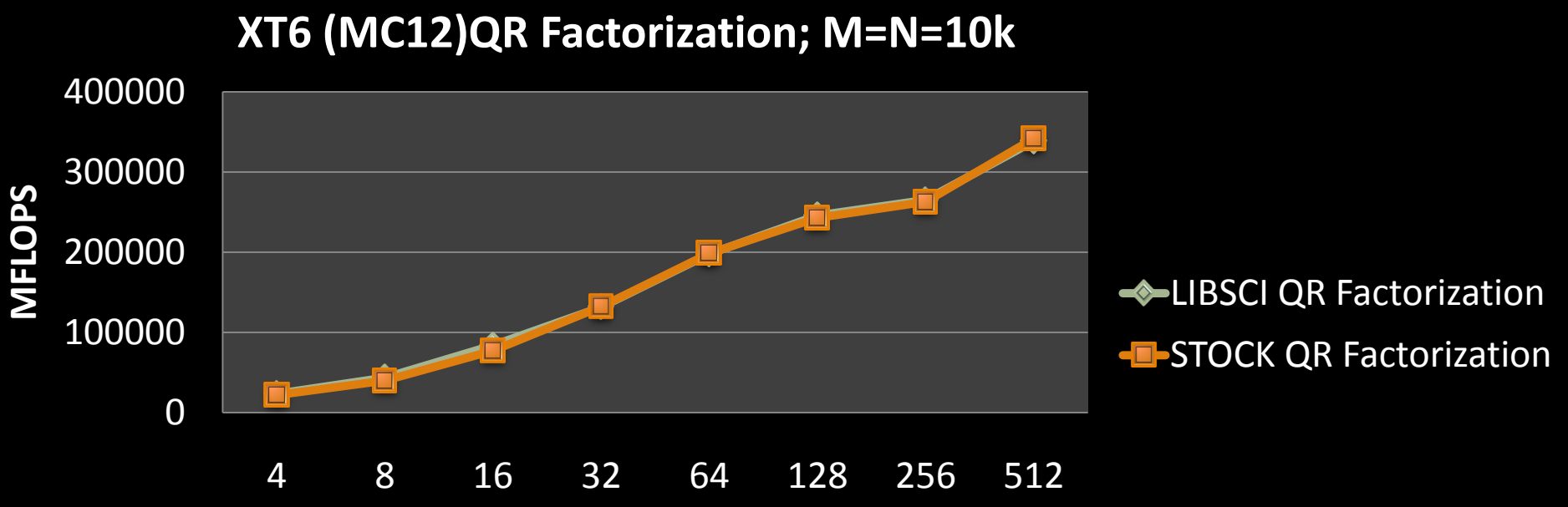
- LAPACK is the very popular linear algebra library for on-node
- Cray's implementation of LAPACK is tuned.
- LAPACK is threaded in the same way as BLAS
- In some routines, the threading is at a higher level than the BLAS updates (LU, Cholesky, QR, some eigensolvers)
- Usage is exactly the same as with the BLAS

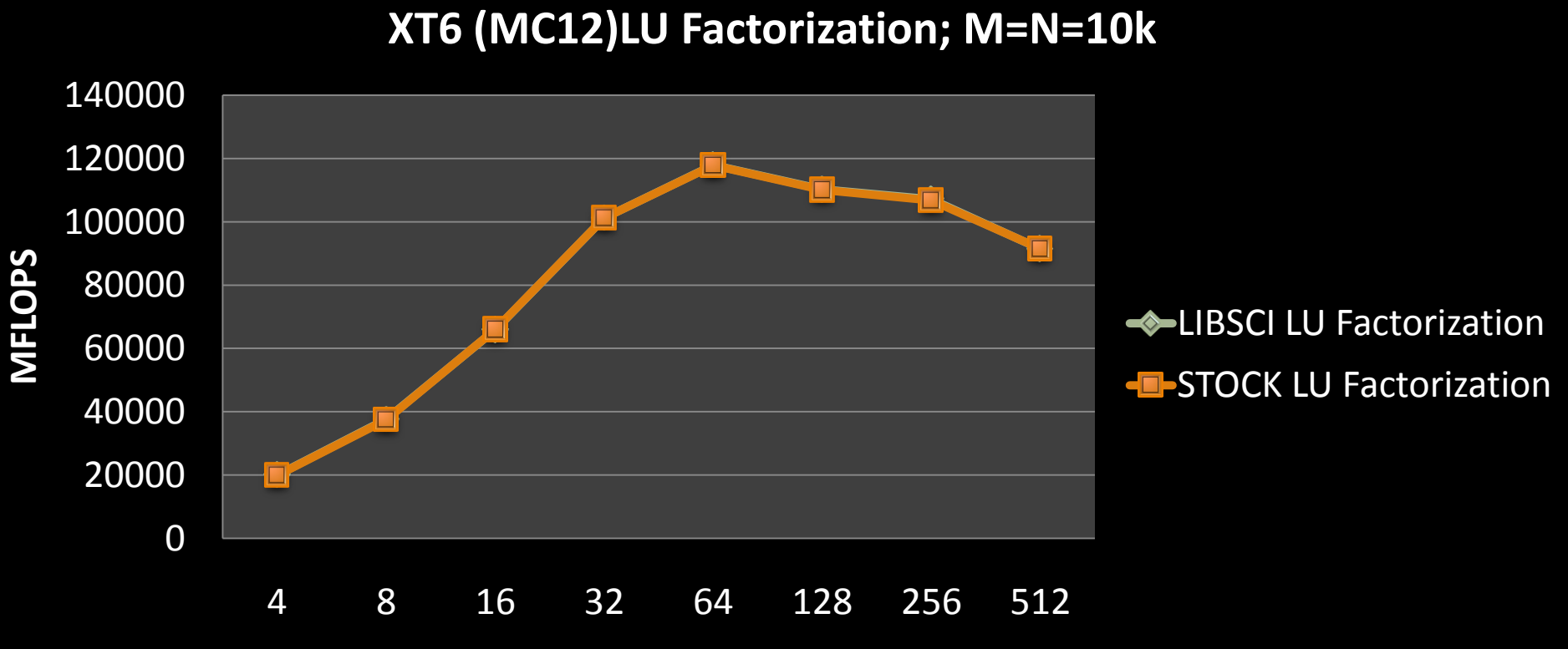
- In the lab, benchmark a call to dgetrf and show the threaded performance

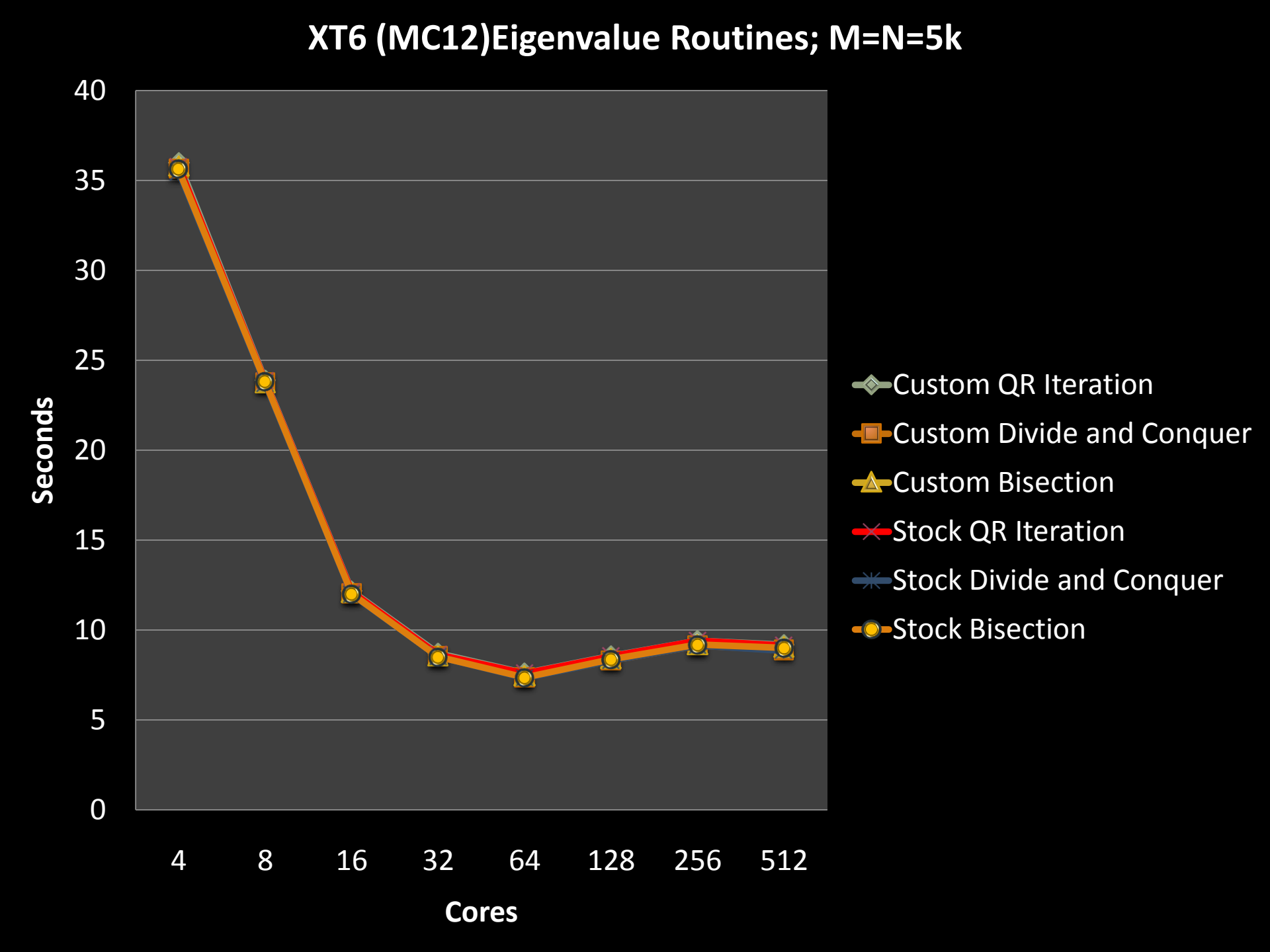

Threaded LAPACK DGETRF GFLOPS numbers for various thread counts
GFLOPS
120
100
80
60
40
20
0
1
2
4
6
8
12
16
20
24
threads
-cc none
-cc cpu

ScaLAPACK

- ScaLAPACK is the near-standard parallel linear algebra library
- Uses distributed memory BLAS, PBLAS
- Uses BLACS for communication
- Using scalapack across nodes and threaded BLAS within nodes is the simplest way to obtain hybrid MPP + thread functionality
- Cray have tuned ScaLAPACK on previous machines, and we are doing so now on XE6.

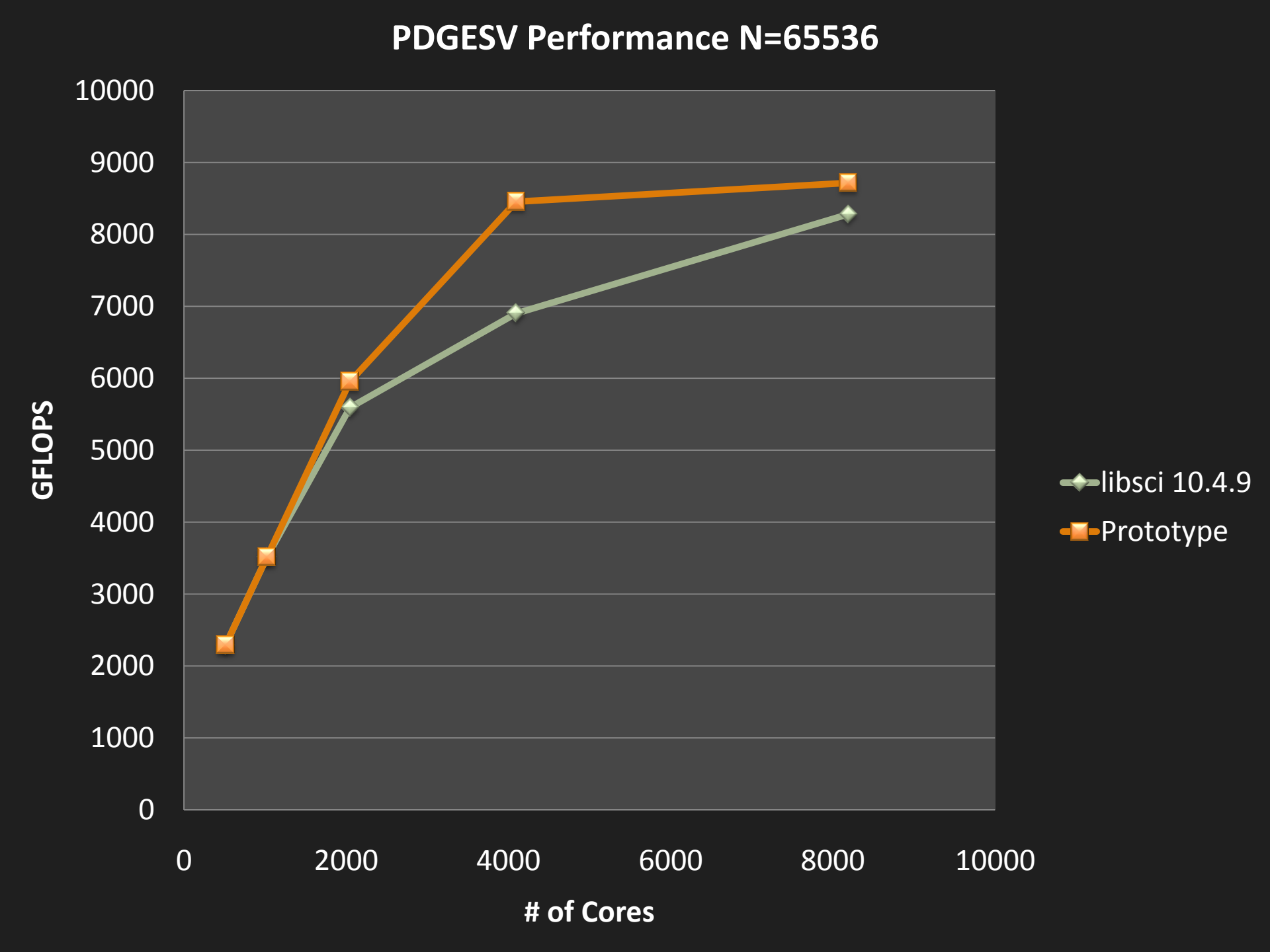

XT6 (MC12)QR Factorization; M=N=10k
MFLOPS
400000
300000
200000
100000
0
4
8
16
32
64
128
256
512
LIBSCI QR Factorization
STOCK QR Factorization

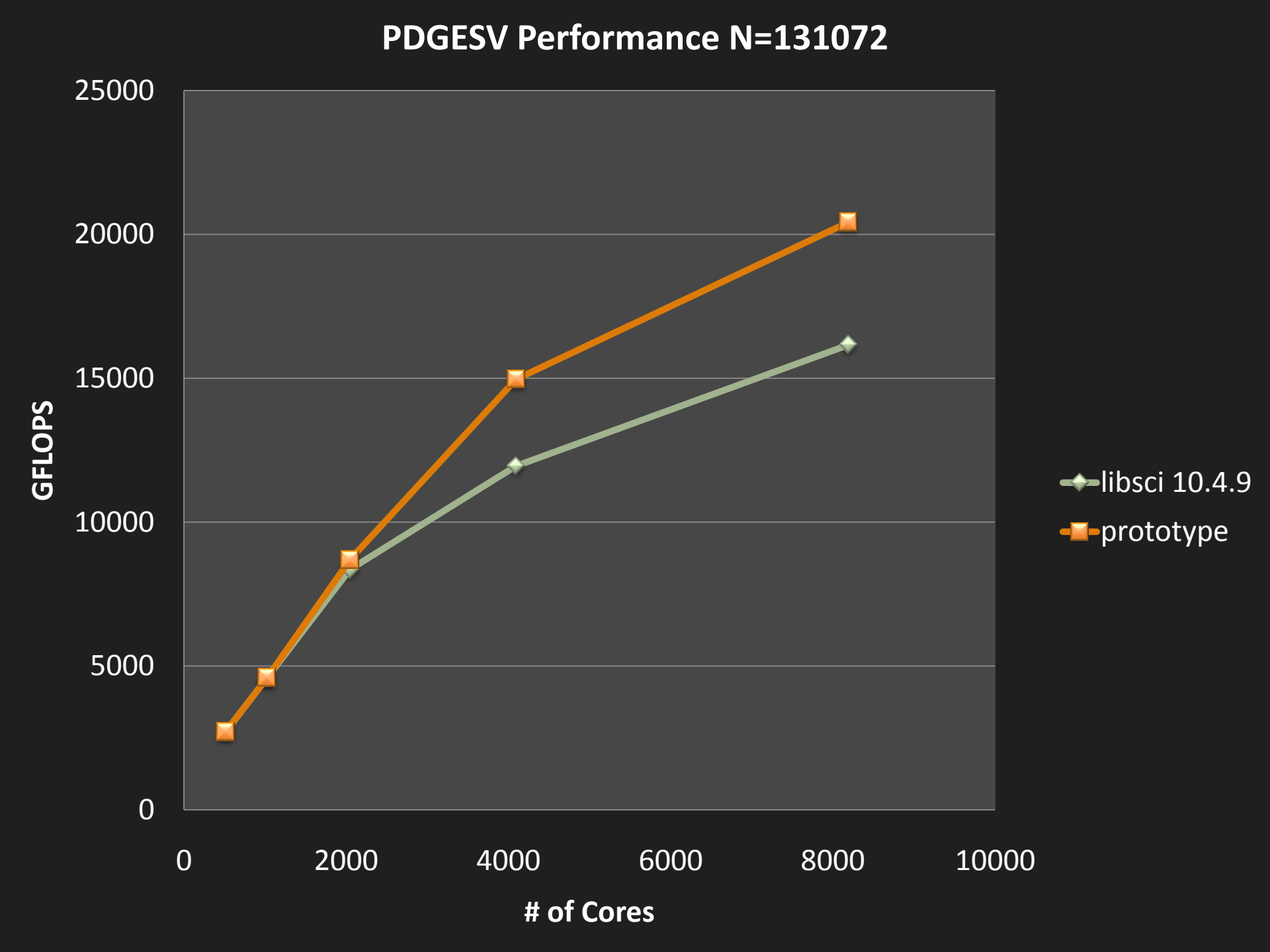

XT6 (MC12)LU Factorization; M=N=10k
MFLOPS
140000
120000
100000
80000
60000
40000
20000
0
4
8
16
32
64
128
256
512
LIBSCI LU Factorization
STOCK LU Factorization


XT6 (MC12)Eigenvalue Routines; M=N=5k
Seconds
0
5
10
15
20
25
30
35
40
4
8
16
32
64
128
256
512
Cores
Custom QR Iteration
Custom Divide and Conquer
Custom Bisection
Stock QR Iteration
Stock Divide and Conquer
Stock Bisection

Tuning ScaLAPACK for XE6

- Cray has a strong track record of tuning parallel linear algebra for older systems – T3E, X1
- Used shmem to replace key communication schemes
- On XE, use many of the same techniques and some new ones
  - Focusing on the LU, Cholesky, divide and conquer eigensolver, tridiagonal reduction
  - Using more asynchronous commuinications in factorizations
  - Replacing MPI with co-array fortran and shmem


PDGESV Performance N=65536
GFLOPS
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
2000
4000
6000
8000
10000
# of Cores
libsci 10.4.9
Prototype


PDGESV Performance N=131072
25000
20000
15000
10000
5000
0
GFLOPS
0
2000
4000
6000
8000
10000
# of Cores
libsci 10.4.9
prototype

Using ScaLAPACK in hybrid mode on XE6

- Use the number of scalpack grid points you want to correspond to the number of MPI ranks you want
- Rely on the BLAS to operate with the number of threads you desire
- Use OMP_NUM_THREADS and the aprun options to set the number of threads you need for on-node parallelism
- Set the threads per node from libsci BLAS with OMP_NUM_THREADS
- Use aprun options –n and –d for nodes and threads

In the examples

- The example calls the scalapack cholesky factorization
- It is easy to run in hybrid mode
- The number of mpi ranks is the number of scalapack / BLACS grid points, (in this example that is hard-coded)
- Set the threads per node using OMP_NUM_THREADS and using aprun option –d

Iterative Refinement Toolkit

- Mixed precision can yield a big win on x86 machines.
- SSE (and AVX) units issue double the number of single precision operations per cycle.
- On CPU, single precision is always 2x as fast as double
- Accelerators sometimes have a bigger ratio
  - Cell – 10x
  - Older NVIDIA cards – 7x
  - New NVIDIA cards (2x)
  - Newer AMD cards (> 2x)
- IRT is a suite of tools to help exploit single precision
  - A library for direct solvers
  - An automatic framework to use mixed precision under the
  - A domain-specific language and preprocessor to convert codes to use mixed precision without active code change

Iterative Refinement Toolkit - Library

- Various tools for solves linear systems in mixed precision
- Obtaining solutions accurate to double precision
  - For well conditioned problems
- Serial and Parallel versions of LU, Cholesky, and QR
- 2 usage methods
  - **IRT Benchmark routines**
    - Uses IRT 'under-the-covers' without changing your code
      - Simply set an environment variable
      - Useful when you cannot alter source code
  - **Advanced IRT API**
    - If greater control of the iterative refinement process is required
      - Allows
        - condition number estimation
        - error bounds return
        - minimization of either forward or backward error
        - 'fall back' to full precision if the condition number is too high
        - max number of iterations can be altered by users

IRT library usage

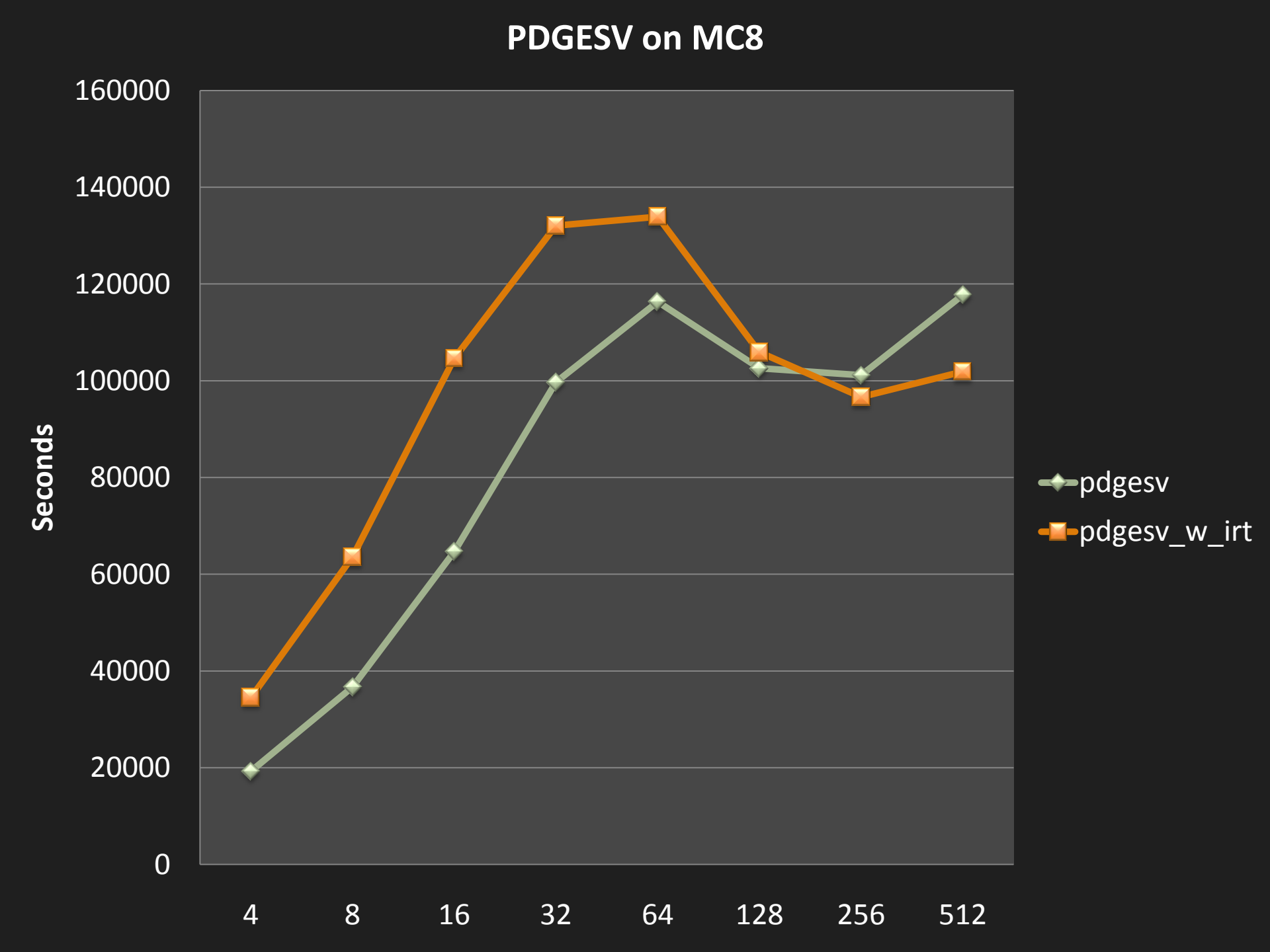
Decide if you want to use advanced API or benchmark API

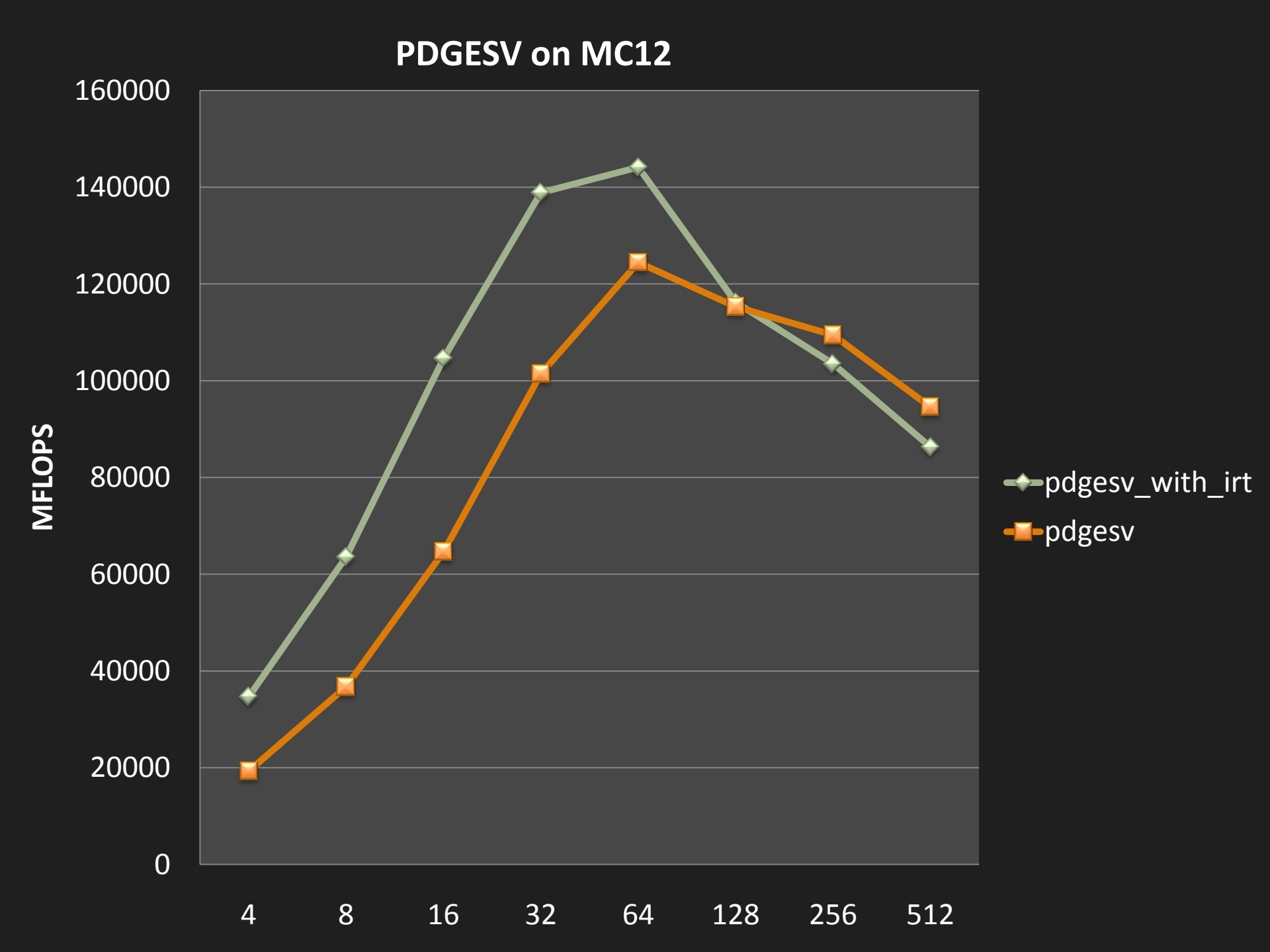
benchmark API :

setenv IRT_USE_SOLVERS 1

advanced API :

1. locate the factor and solve in your code (LAPACK or ScaLAPACK)
2. Replace factor and solve with a call to IRT routine
   - e.g. dgesv -> irt_lu_real_serial
   - e.g. pzgesv -> irt_lu_complex_parallel
   - e.g pzposv -> irt_po_complex_parallel
3. Set advanced arguments
   - Forward error convergence for most accurate solution
   - Condition number estimate
   - “fall-back” to full precision if condition number too high


PDGESV on MC8
160000
140000
120000
100000
80000
60000
40000
20000
0
Seconds
4
8
16
32
64
128
256
512
pdgesv
pdgesv_w_irt


PDGESV on MC12
160000
140000
120000
100000
80000
60000
40000
20000
0
MFLOPS
4
8
16
32
64
128
256
512
pdgesv_with_irt
pdgesv

IRT – Mixed Precision Preprocessor (MXP)

- A domain-specific language for the expression of mixed precision
- Allows the user to denote a factor/solve region and a set of inputs outputs

- MXP will convert the code to a mixed-precision equivalent.

- 2 modes
  - 1. Iterative mode for iterative solvers (define inputs and outputs and main loop in MXP
  - 2. Direct mode for direct solvers. Factor and solve loops are replaced with an iterative refinement scheme

Pseudo-example Mode=iterative

```
!$MXP mode = iterative, matrix = A

Double precision, allocatable :: A( : , : )

allocate (A(M,N))
!$MXP loop, output = x
Do
    operations on matrix A
    calculate residual
    exit criteria
End
```

```
Double precision, allocatable :: A( : , : )
Real, allocatable A_lp( : , : )
Allocate( A( M, N), A_lp( M, N ) )

Do
    operations on matrix A_lp
    calculate residual in X_lp
    exit criteria (same)
End
X = X_lp
```

Pseudo-example mode=direct

```
!$MXP mode = direct, matrix = A, rhs = x

Double precision, allocatable :: A( : , : ), x (: )

allocate (A(M,N), x (N) )

!$MXP factorization

Factorization of matrix A

!$MXP end_factorization

!$MXP solve

Solve of matrix A

!$MXP end_solve
```

```
Double precision, allocatable :: A( : , : )
real, allocatable :: A_lp(:,:)
allocate ( A( M, N ) , X(N),  A_lp(M, N), X_lp(N ) )

b32 = b64
L32U32 = A32
x32 = b32 ( A32 ) ^-1
x64 = x32
do
      i = i + 1
      r64_i = b32 - A32x32
      r32 = r64
      z32 = r32 ( L32U32)^-1
      x_i+1 = x_i + z_i
end do
deallocate(A32, X32)
```

In the examples

- The ScaLAPACK example can be made to work with IRT's automatic interface
- After you have done the scalapack example, then set IRT_USE_SOLVERS and repeat the experiment
- You only need to re-run, not recompile or relink
- If you write a simple LAPACK code to show threading in lapack, you can do the same thing.
- Those very interested can call directly e.g. irt_real_parallel

Cray Adaptive FFT (CRAFFT)

- Serial CRAFFT is largely a productivity enhancer
- Some FFT developers have problems such as
  - Which library choice to use?
  - How to use complicated interfaces (e.g., FFTW)
- Standard FFT practice
  - Do a plan stage
  - Do an execute
- CRAFFT is designed with simple-to-use interfaces
  - Planning and execution stage can be combined into one function call
  - Underneath the interfaces, CRAFFT calls the appropriate FFT kernel

CRAFFT usage

1. Load module fftw/3.2.0 or higher.
2. Add a Fortran statement “use crafft”
3. call crafft_init()
4. Call crafft transform using none, some or all optional arguments (as shown in red)

In-place, implicit memory management :

```
call crafft_z2z3d(n1,n2,n3,input,ld_in,ld_in2,isign)
```

in-place, explicit memory management

```
call crafft_z2z3d(n1,n2,n3,input,ld_in,ld_in2,isign,work)
```

out-of-place, explicit memory management :

```
crafft_z2z3d(n1,n2,n3,input,ld_in,ld_in2,output,ld_out,ld_out2,isign,work)
```

Note : the user can also control the planning strategy of CRAFFT using the CRAFFT_PLANNING environment variable and the do_exe optional argument, please see the intro_crafft man page.

Parallel CRAFFT

- Parallel CRAFFT is meant as a performance improvement to FFTW2 distributed transforms
  - Uses FFTW3 for the serial transform
  - Uses ALLTOALLV where possible
  - Uses a more adaptive communication scheme based on input
- Can provide impressive performance improvements over FFTW2
- Currently implemented
  - complex-complex
  - Real-complex and complex-real
  - 3-d and 2-d
  - In-place and out-of-place
  - 1 data distribution scheme but looking to support more (please tell us)
- Just released :
  - C language support for serial and parallel

parallel CRAFFT usage

1. Add “use crafft” to Fortran code
2. Initialize CRAFFT using crafft_init
3. Assume MPI initialized and data distributed (see manpage)
4. Call crafft, e.g. (optional arguments in red)

2-d complex-complex, in-place, internal mem management :

```
call crafft_pz2z2d(n1,n2,input,isign,flag,comm)
```

2-d complex-complex, in-place with no internal memory :

```
call crafft_pz2z2d(n1,n2,input,isign,flag,comm,work)
```

2-d complex-complex, out-of-place, internal mem manager :

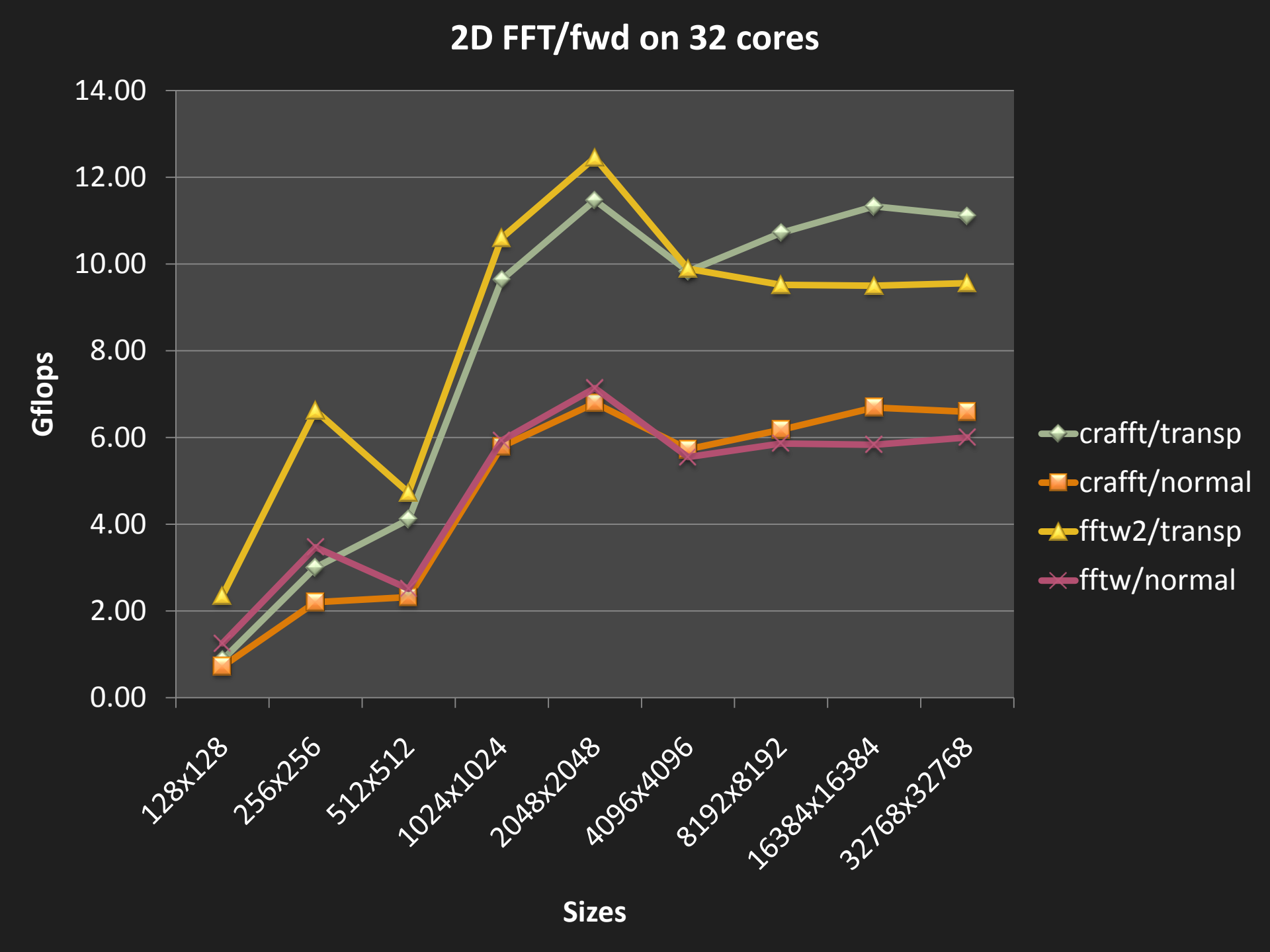
```
call crafft_pz2z2d(n1,n2,input,output,isign,flag,comm)
```

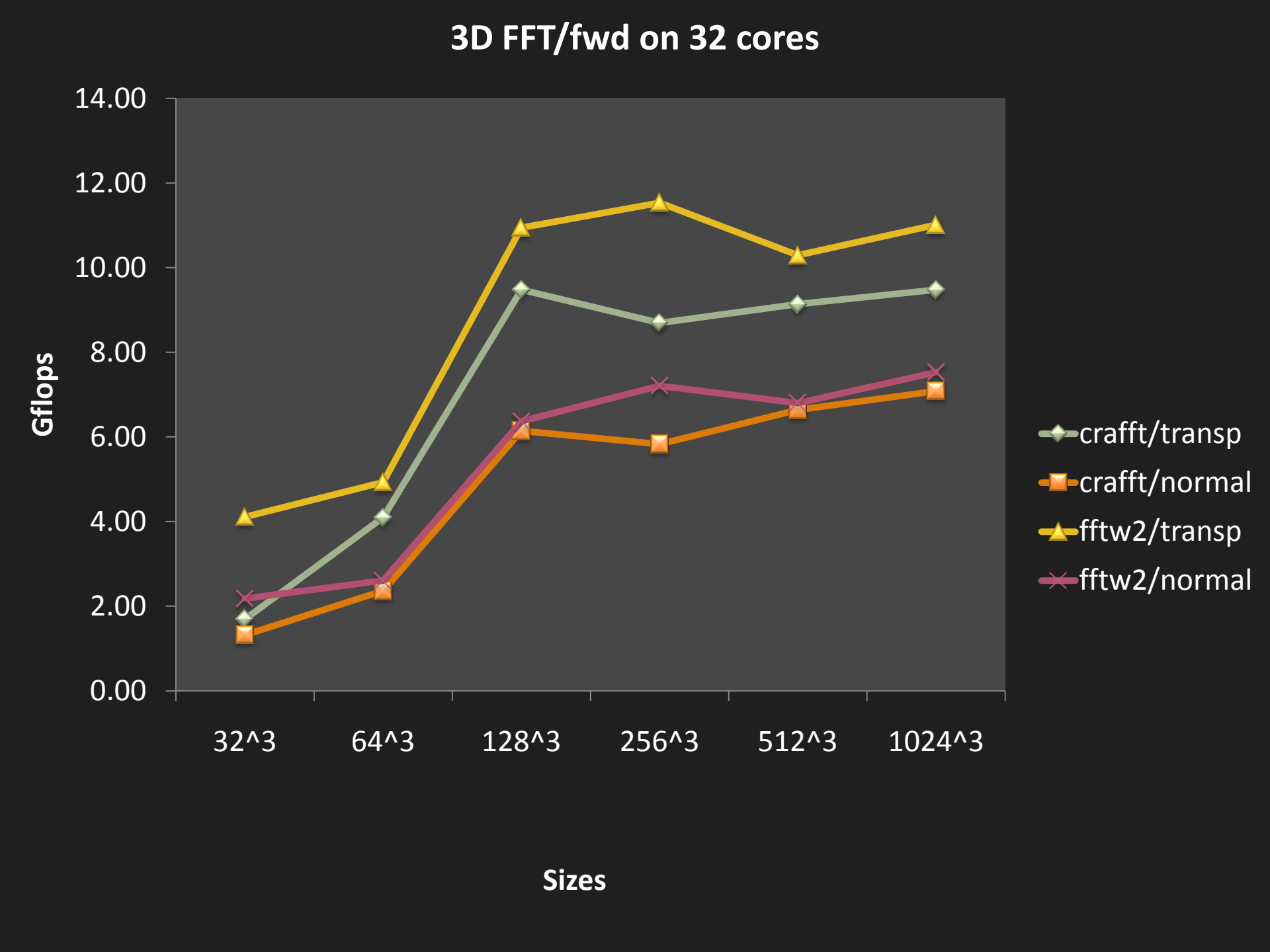
2-d complex-complex, out-of-place, no internal memory :

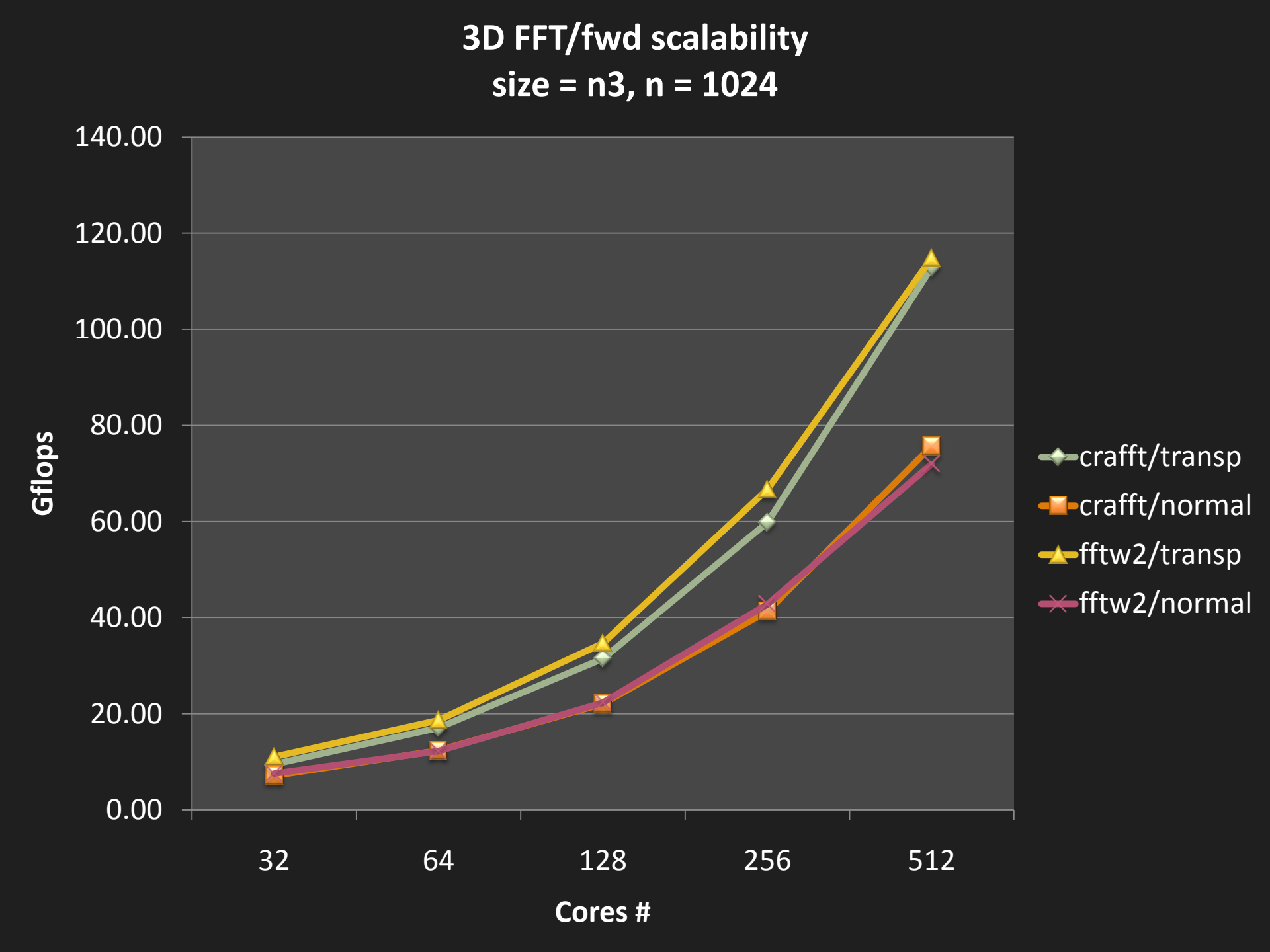
```
crafft_pz2z2d(n1,n2,input,output,isign,flag,comm,work)
```

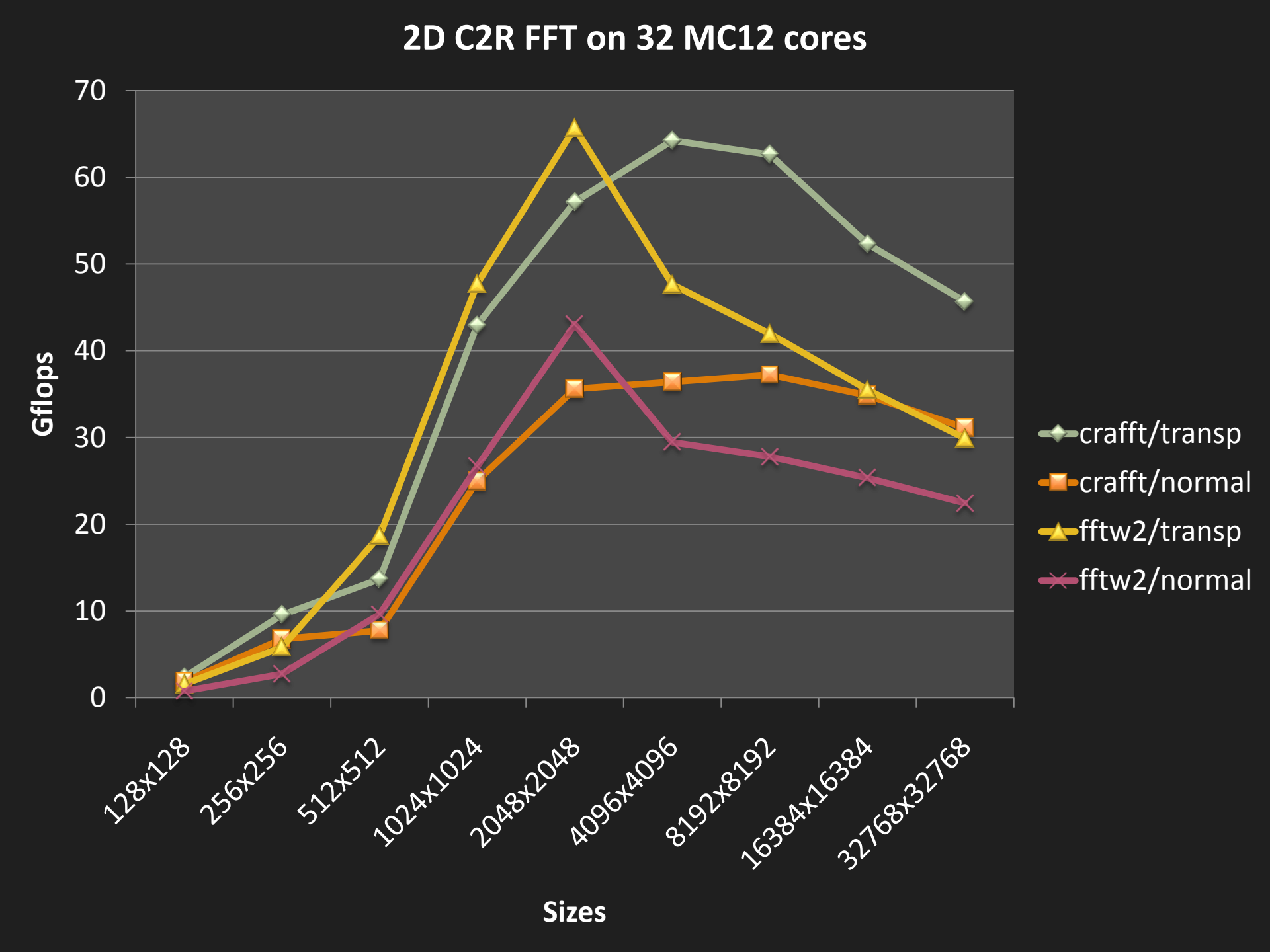
Each routine above has manpage. Also see 3d equivalent :

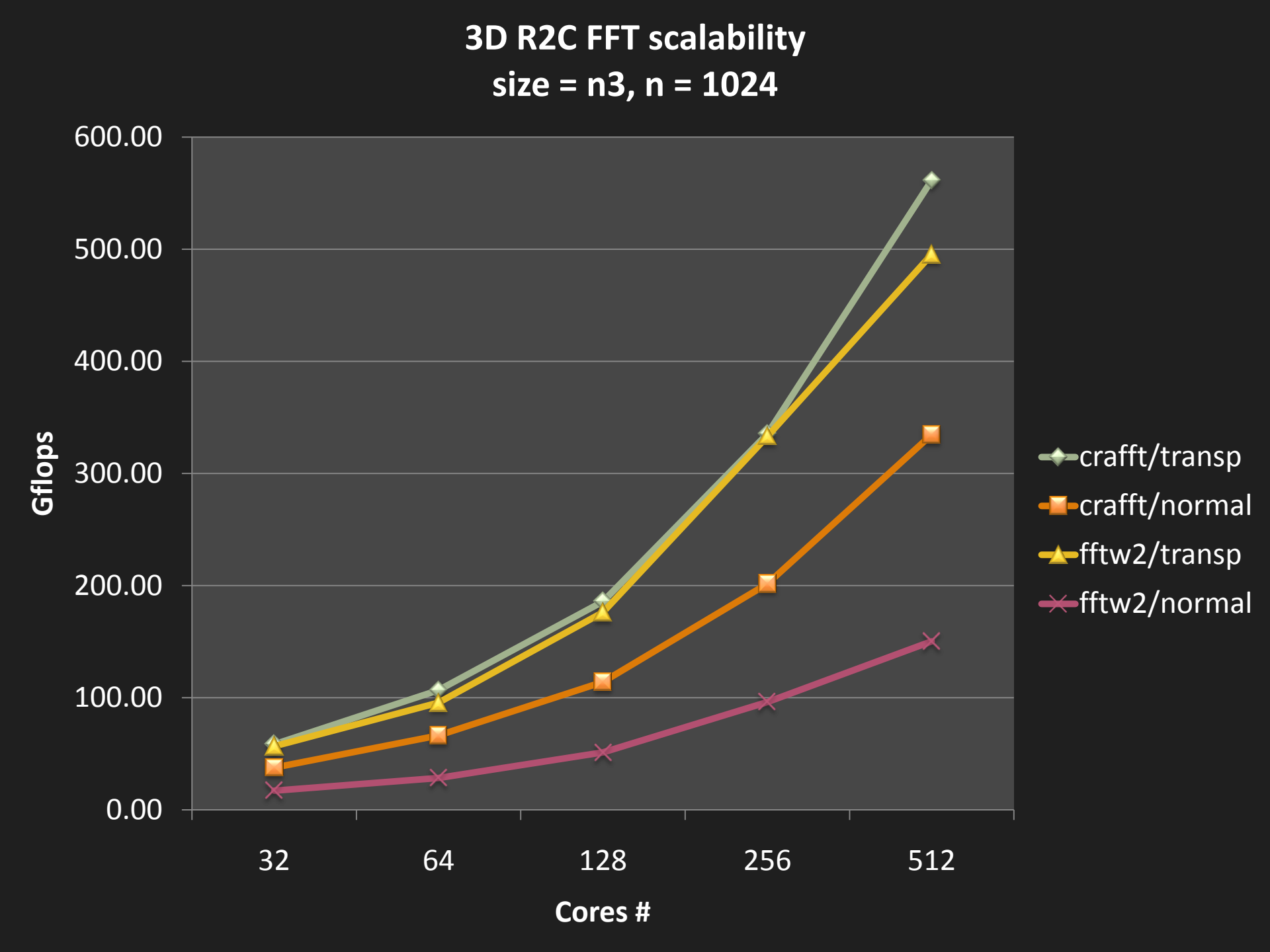
```
man crafft_pz2z3d
```

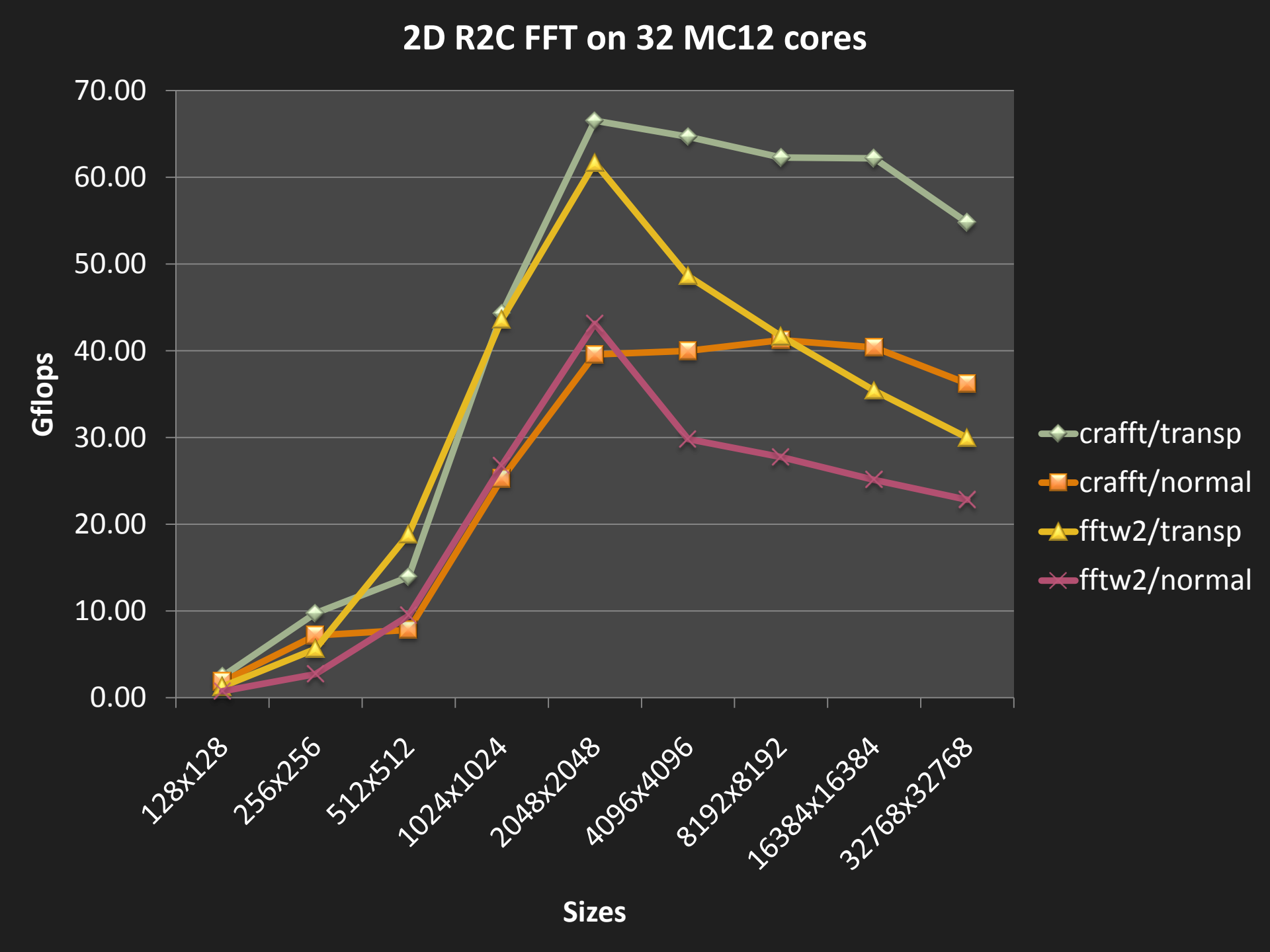

2D FFT/fwd on 32 cores
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Gflops
128x128
256x256
512x512
1024x1024
2048x2048
4096x4096
8192x8192
16384x16384
32768x32768
Sizes
crafft/transp
crafft/normal
fftw2/transp
fftw/normal

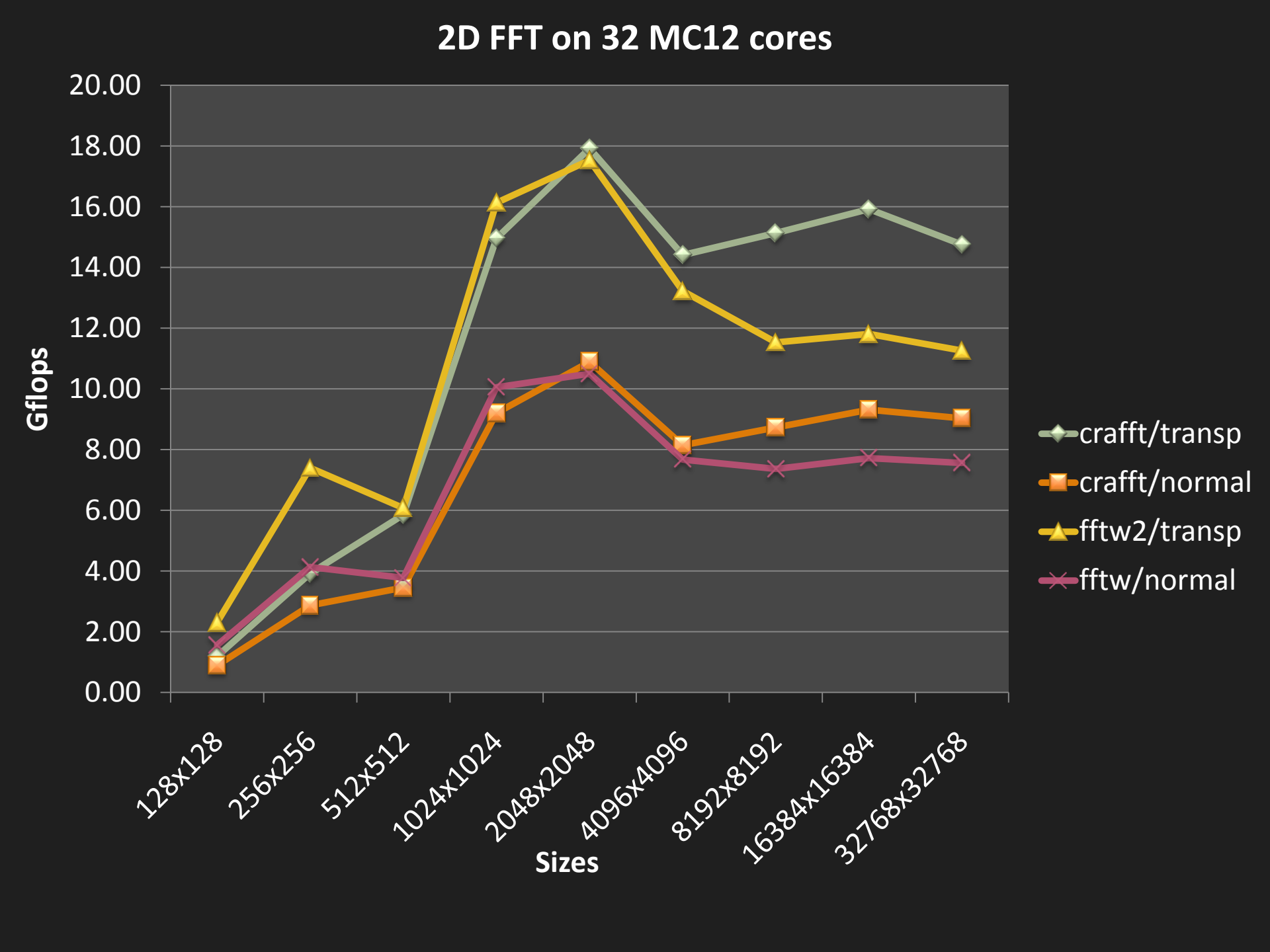

3D FFT/fwd on 32 cores
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Gflops
32^3
64^3
128^3
256^3
512^3
1024^3
Sizes
crafft/transp
crafft/normal
fftw2/transp
fftw2/normal

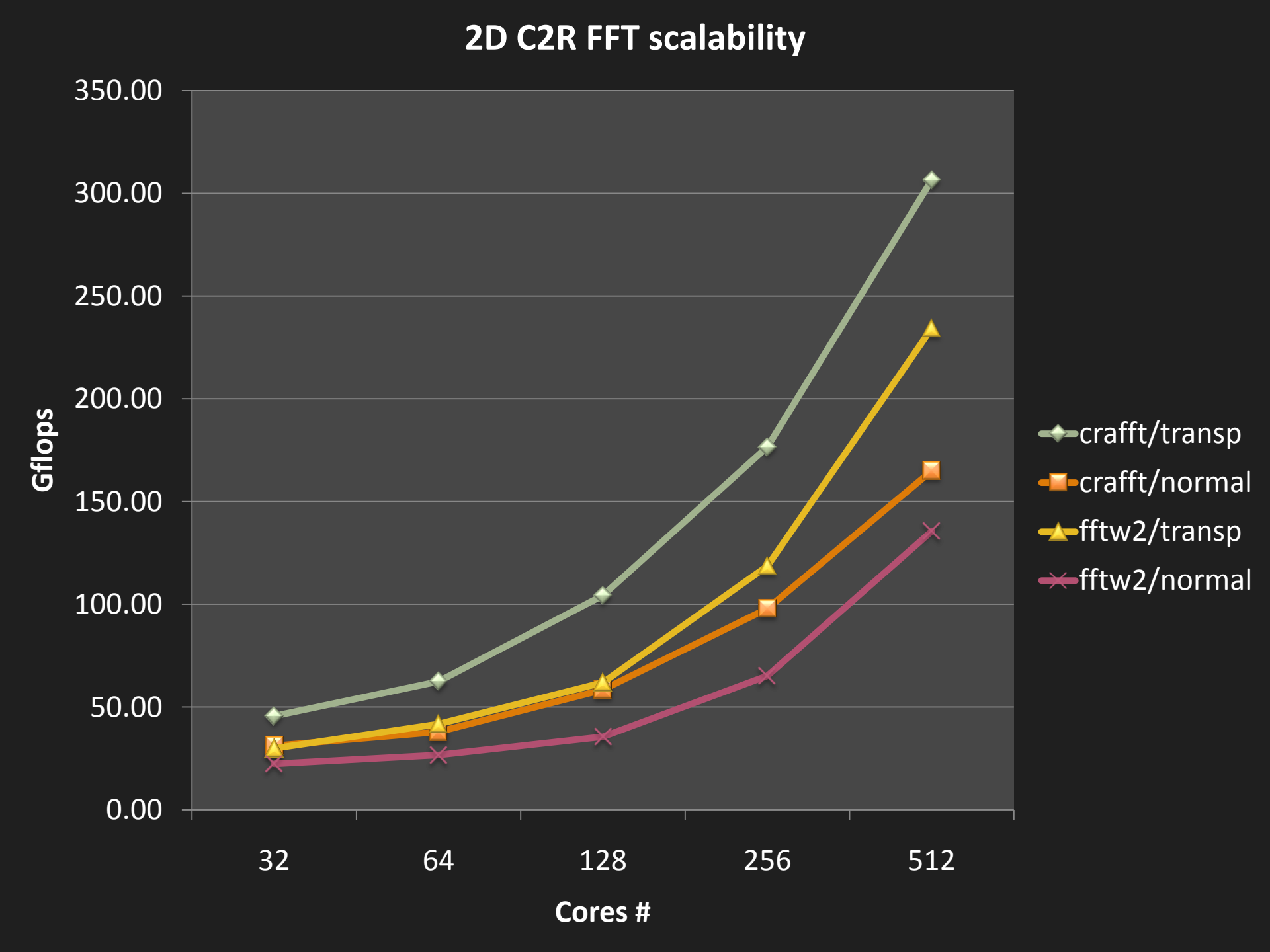

3D FFT/fwd scalability
size = n3, n = 1024
140.00
120.00
100.00
80.00
60.00
40.00
20.00
0.00
Gflops
32
64
128
256
512
Cores #
crafft/transp
crafft/normal
fftw2/transp
fftw2/normal

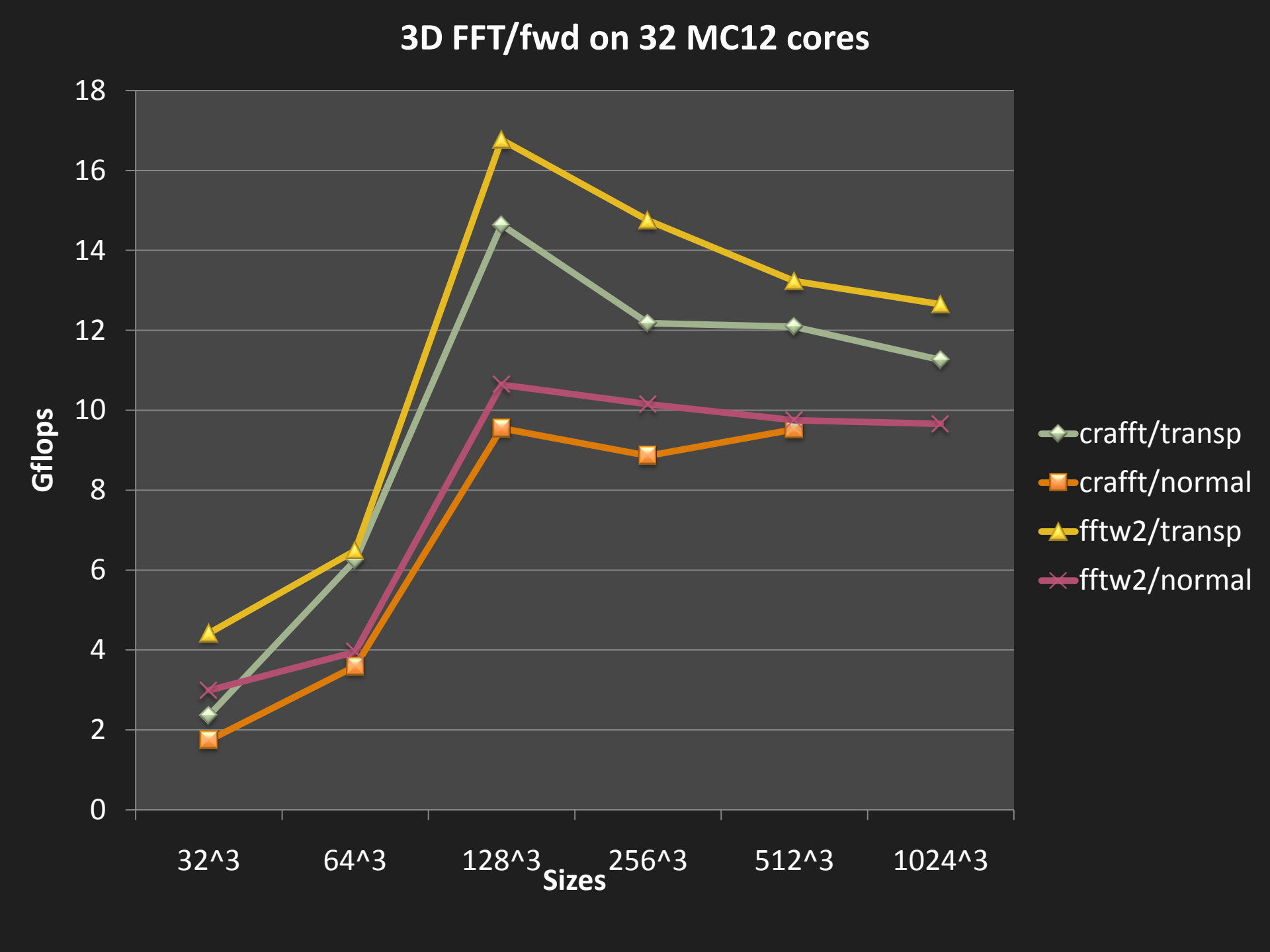

2D C2R FFT on 32 MC12 cores
Gflops
70
60
50
40
30
20
10
0
128x128
256x256
512x512
1024x1024
2048x2048
4096x4096
8192x8192
16384x16384
32768x32768
Sizes
crafft/transp
crafft/normal
fftw2/transp
fftw2/normal

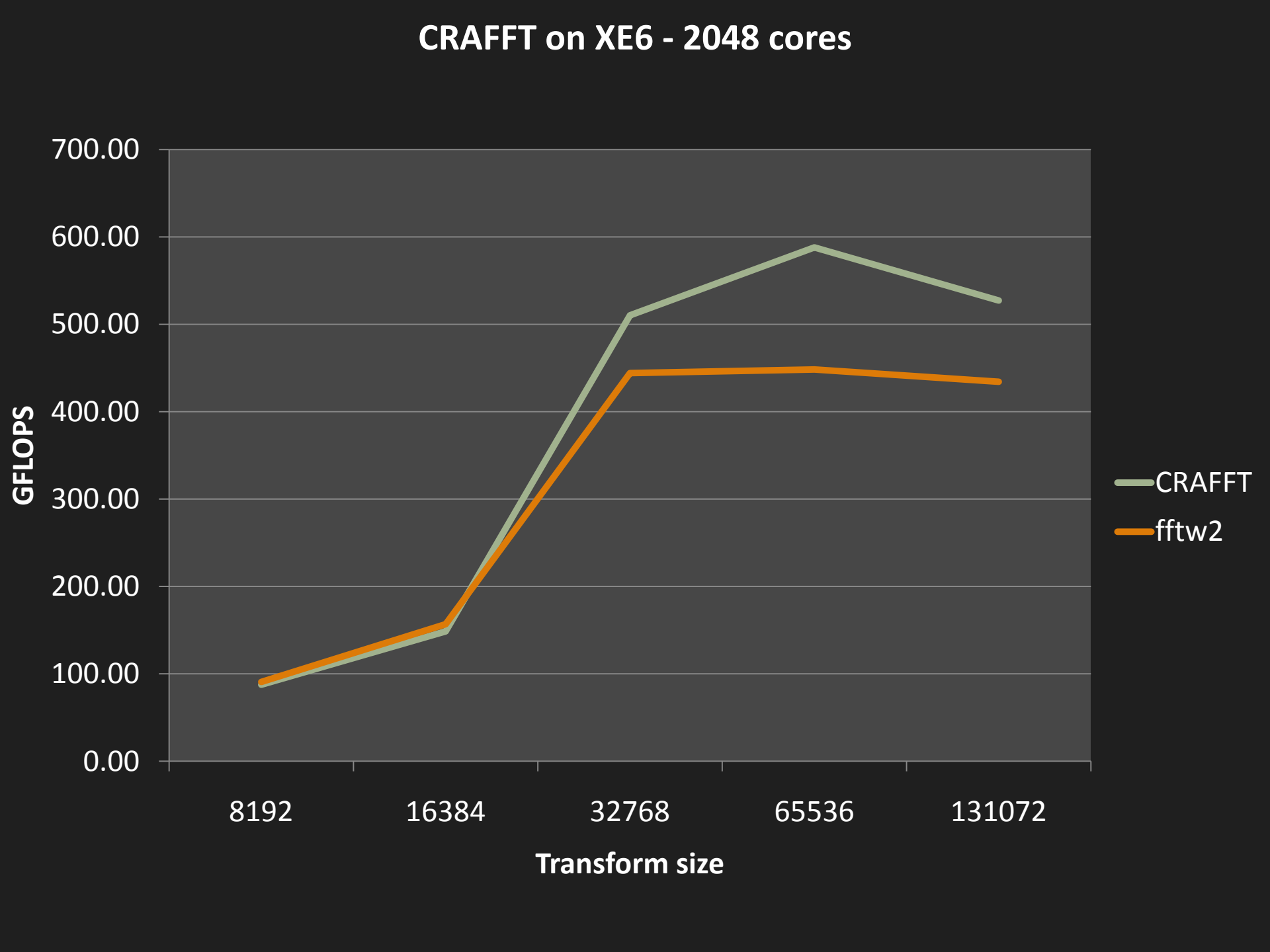

3D R2C FFT scalability
size = n3, n = 1024
600.00
500.00
400.00
300.00
200.00
100.00
0.00
Gflops
32
64
128
256
512
Cores #
crafft/transp
crafft/normal
fftw2/transp
fftw2/normal


2D R2C FFT on 32 MC12 cores
70.00
60.00
50.00
40.00
30.00
20.00
10.00
0.00
Gflops
128x128
256x256
512x512
1024x1024
2048x2048
4096x4096
8192x8192
16384x16384
32768x32768
Sizes
crafft/transp
crafft/normal
fftw2/transp
fftw2/normal


2D FFT on 32 MC12 cores
Gflops
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
128x128
256x256
512x512
1024x1024
2048x2048
4096x4096
8192x8192
16384x16384
32768x32768
Sizes
crafft/transp
crafft/normal
fftw2/transp
fftw/normal


2D C2R FFT scalability
350.00
300.00
250.00
200.00
150.00
100.00
50.00
0.00
Gflops
32
64
128
256
512
Cores #
crafft/transp
crafft/normal
fftw2/transp
fftw2/normal


3D FFT/fwd on 32 MC12 cores
Gflops
0
2
4
6
8
10
12
14
16
18
32^3
64^3
128^3
256^3
512^3
1024^3
Sizes
crafft/transp
crafft/normal
fftw2/transp
fftw2/normal


CRAFFT on XE6 - 2048 cores
700.00
600.00
500.00
400.00
300.00
200.00
100.00
0.00
GFLOPS
8192
16384
32768
65536
131072
Transform size
CRAFFT
fftw2

CRAFFT examples

- We have an example program that calls parallel CRAFFT
- It is simple to run on XE6
- You can write similar from scratch after looking at it
- Or, modify to perform a different transform type

Sparse

- At one time Cray provided both
  - Custom sparse direct solvers
  - Custom sparse iterative solvers
- There has been an evolution towards using standardized frameworks such as Trilinos & PETSc
- Today, we attempt to provide that same performance boost while maintaining productivity
- CASK library – optimizes sparse matrix operations on Cray computers whilst being invisible to the user
  - Cray Trilinos distribution
  - Cray PETSc distribution

PETSc (Portable, Extensible Toolkit for Scientific Computation)

- Serial and Parallel versions of sparse iterative linear solvers
  - Suites of iterative solvers
    - CG, GMRES, BiCG, QMR, etc.
  - Suites of preconditioning methods
    - IC, ILU, diagonal block (ILU/IC), Additive Schwartz, Jacobi, SOR
  - Support block sparse matrix data format for better performance
  - Interface to external packages (ScaLAPACK, SuperLU_DIST)
  - Fortran and C support
  - Newton-type nonlinear solvers
- Extremely large user community in US and Europe
- http://www-unix.mcs.anl.gov/petsc/petsc-as

Usage and External Packages

- Cray provides
  - Hypre: scalable parallel preconditioners
  - ParMetis: parallel graph partitioning package
  - MUMPS: parallel multifrontal sparse direct solver
  - SuperLU: sequential version of SuperLU_DIST
- To use Cray-PETSc, load the appropriate module :

  module load petsc

  (or) module load petsc-complex

  (no need to load a compiler specific module)
- Treat the Cray distribution as your local PETSc installation

PETSc is not threaded!

Trilinos

- The Trilinos Project http://trilinos.sandia.gov/

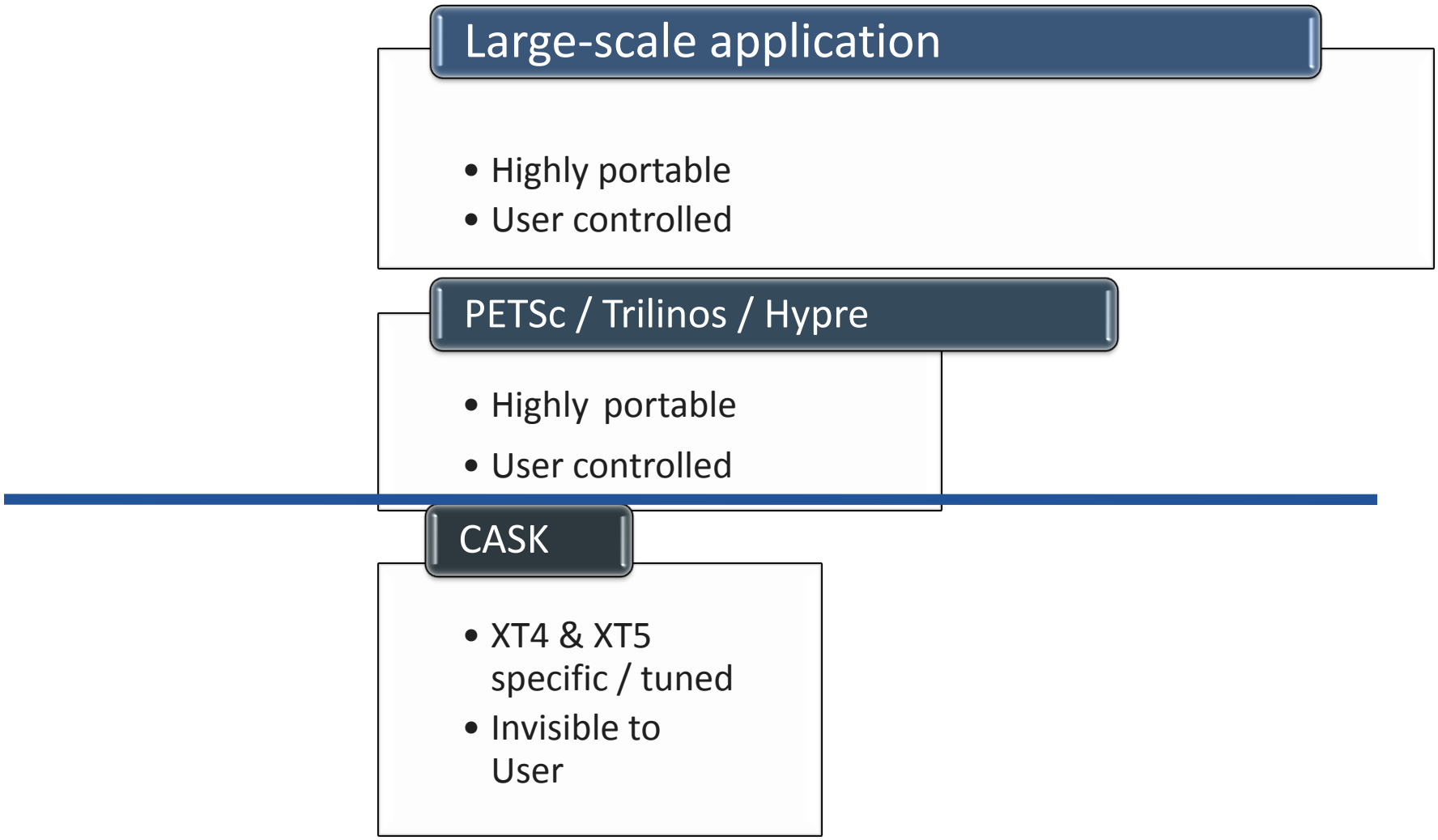
  “an effort to develop algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems”
- A unique design feature of Trilinos is its focus on packages.
- Very large user-base and growing rapidly. Important to DOE.
- Cray’s optimized Trilinos released on January 21 2010
  - Includes 50+ trilinos packages
  - Optimized via CASK
  - Any code that uses Epetra objects can access the optimizations
- Usage :

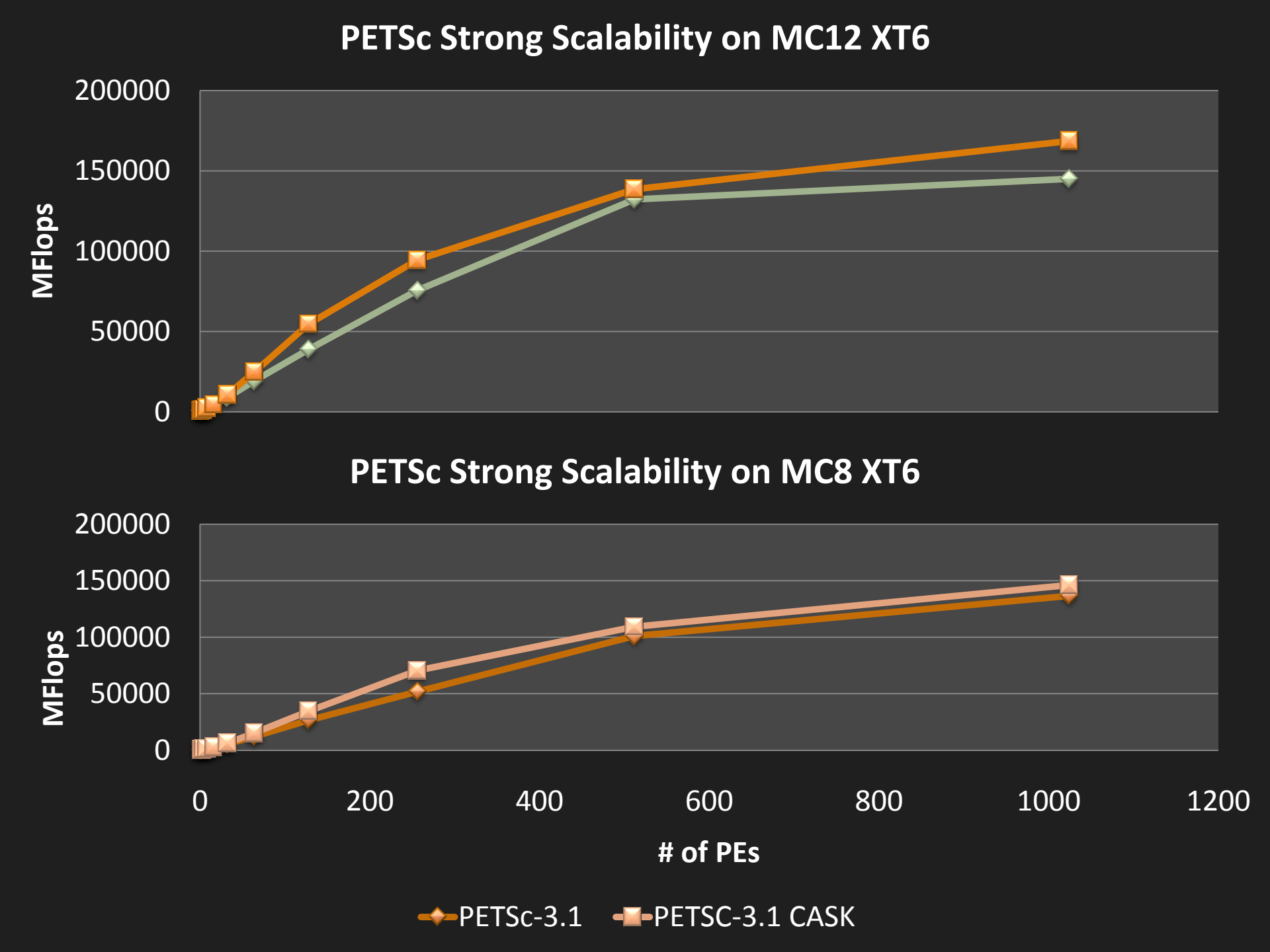
  module load trilinos

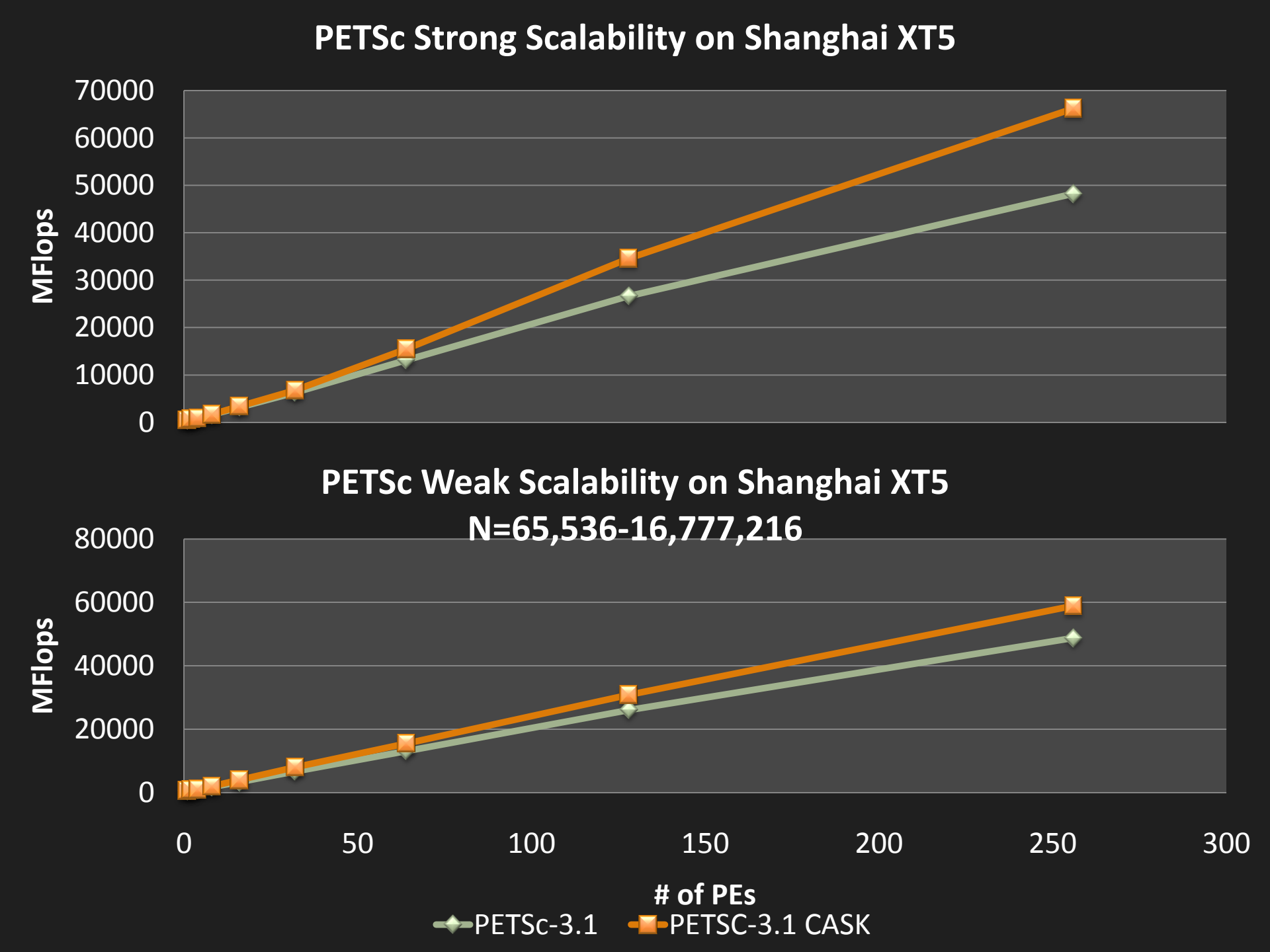
Cray Adaptive Sparse Kernel (CASK)

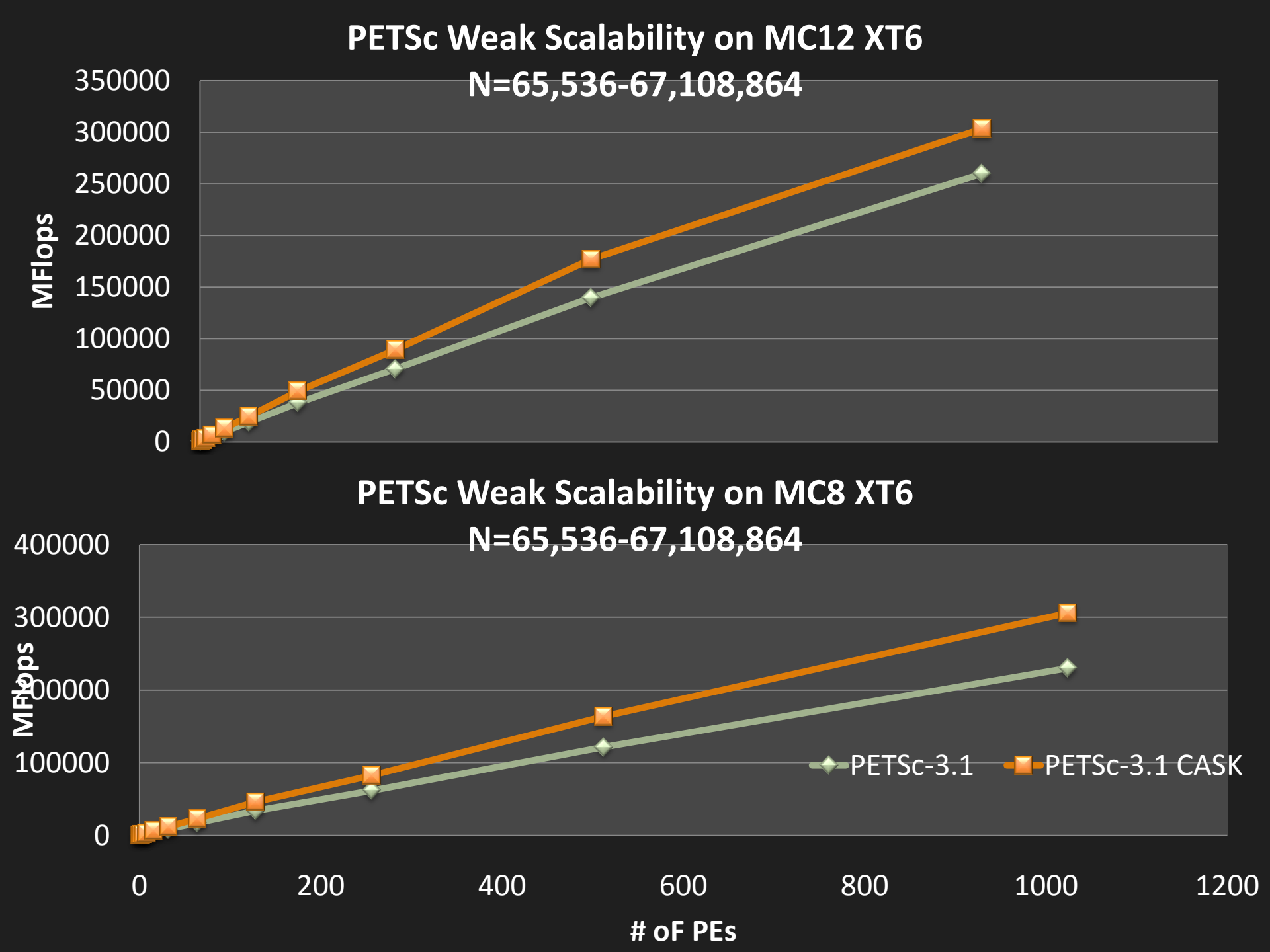
- CASK is a product developed at Cray using the Cray Auto-tuning Framework (Cray ATF)
- Uses ATF auto-tuning, specialization and Adaptation concepts
- Offline :
  - ATF program builds many thousands of sparse kernel
  - Testing program defines matrix categories based on density, dimension etc
  - Each kernel variant is tested against each matrix class
  - Performance table is built and adaptive library constructed
- Runtime
  - Scan matrix at very low cost
  - Map user's calling sequence to nearest table match
  - Assign best kernel to the calling sequence
  - Optimized kernel used in iterative solver execution

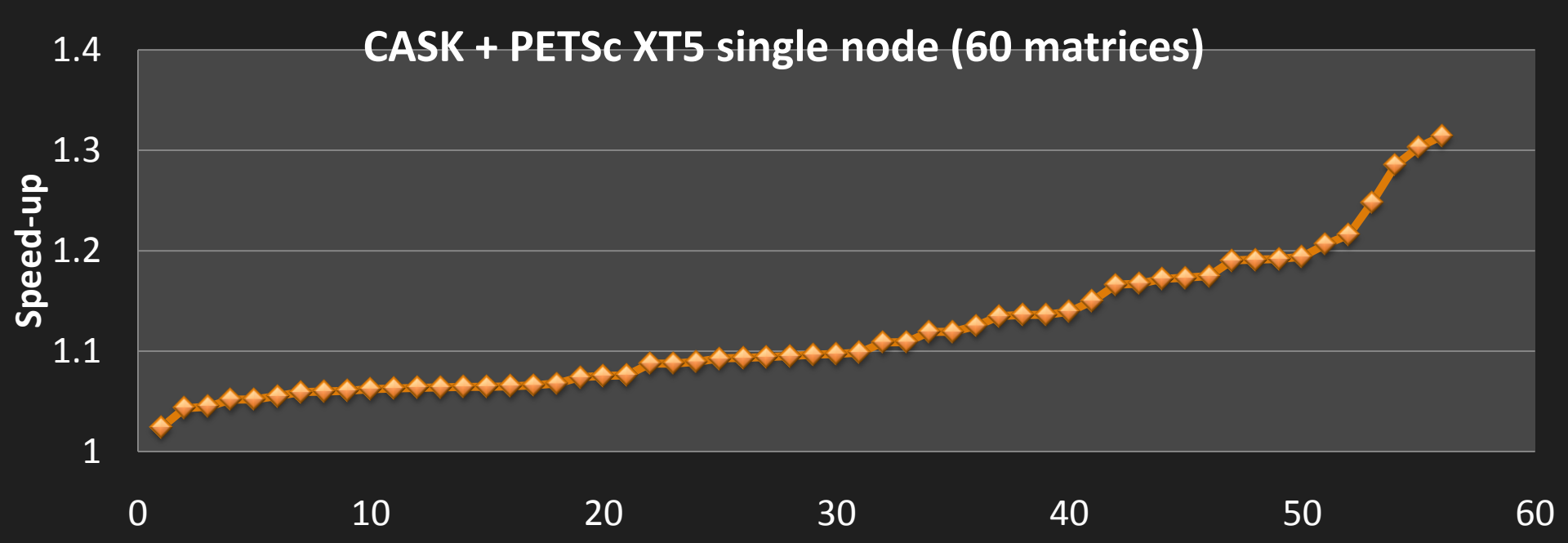
Support Model

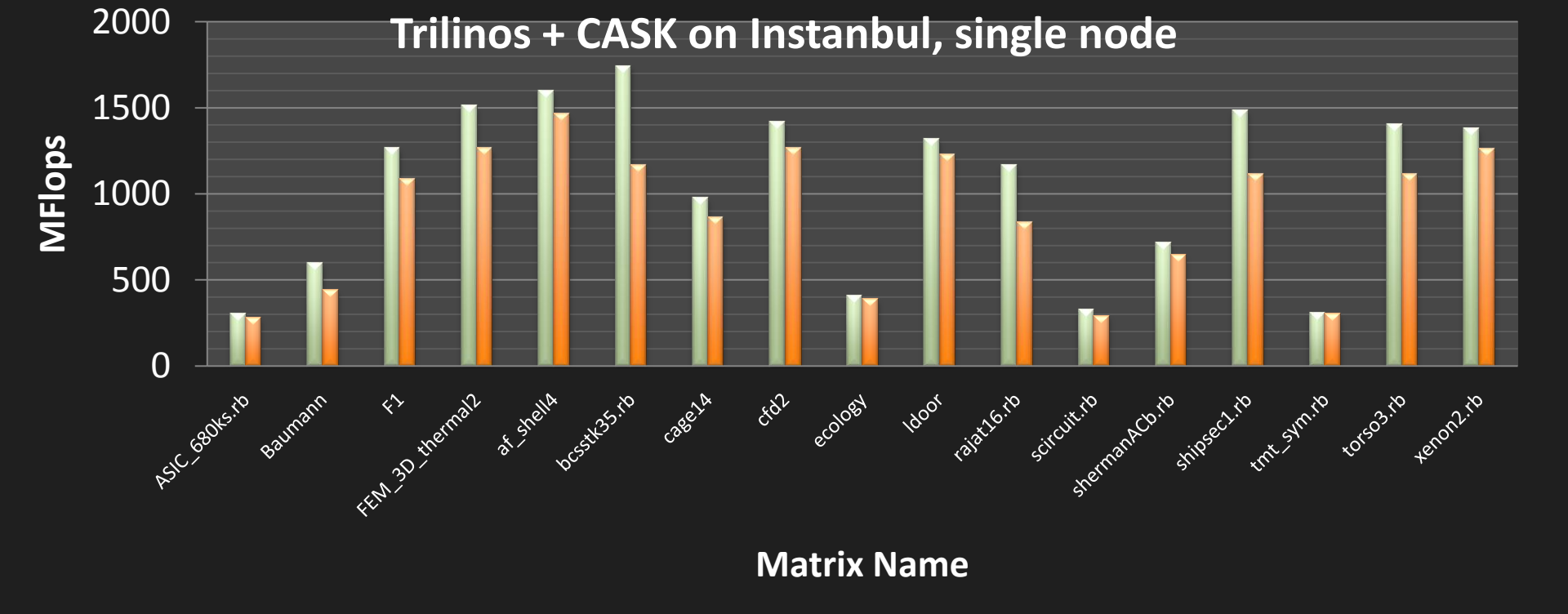



PETSc Strong Scalability on MC12 XT6
MFlops
200000
150000
100000
50000
0
PETSc Strong Scalability on MC8 XT6
MFlops
200000
150000
100000
50000
0
0
200
400
600
800
1000
1200
# of PEs
PETSc-3.1
PETSC-3.1 CASK


PETSc Strong Scalability on Shanghai XT5
MFlops
70000
60000
50000
40000
30000
20000
10000
0
PETSc Weak Scalability on Shanghai XT5
N=65,536-16,777,216
MFlops
80000
60000
40000
20000
0
0
50
100
150
200
250
300
# of PEs
PETSc-3.1
PETSC-3.1 CASK


PETSc Weak Scalability on MC12 XT6
N=65,536-67,108,864
MFlops
350000
300000
250000
200000
150000
100000
50000
0
PETSc Weak Scalability on MC8 XT6
N=65,536-67,108,864
MFlops
400000
300000
200000
100000
0
0
200
400
600
800
1000
1200
oF PEs
PETSc-3.1
PETSc-3.1 CASK

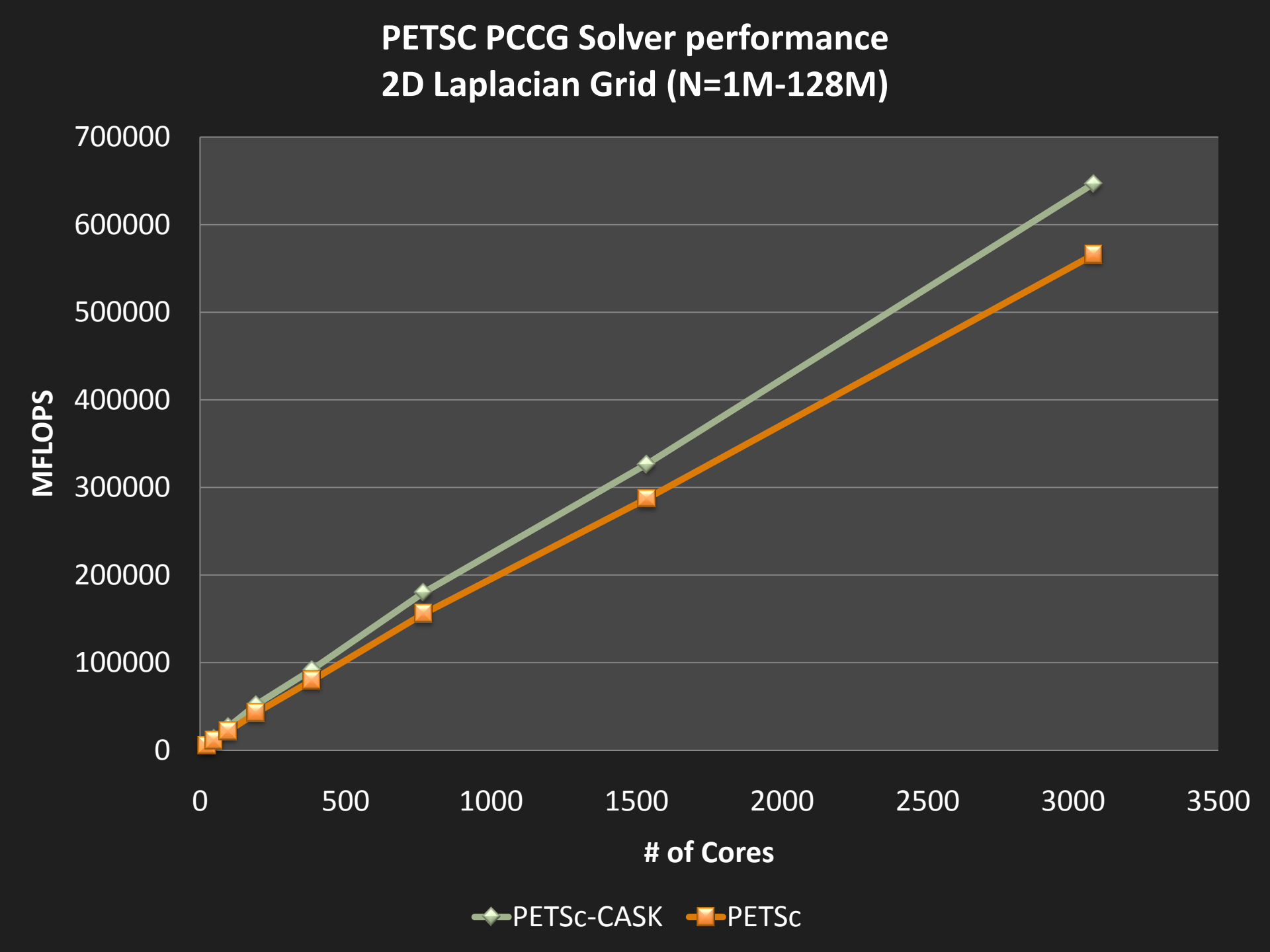

CASK + PETSc XT5 single node (60 matrices)
Speed-up
1.4
1.3
1.2
1.1
1
0
10
20
30
40
50
60


Trilinos + CASK on Instanbul, single node
MFlops
2000
1500
1000
500
0
ASIC_680ks.rb
Baumann
F1
FEM_3D_thermal2
af_shell4
bcsstk35.rb
cage14
cfd2
ecology
ldoor
rajat16.rb
scircuit.rb
shermanACb.rb
shipsec1.rb
tmt_sym.rb
torso3.rb
xenon2.rb
Matrix Name

CASK on MC12 and XE6

- MC12 is the first entirely automated CASK
- ATF used for all stages
  - Codegen
  - Testing, search
  - Execution
  - Automation of adaptive lirbrary

- Released September 2010


PETSC PCCG Solver performance
2D Laplacian Grid (N=1M-128M)
MFLOPS
700000
600000
500000
400000
300000
200000
100000
0
0
500
1000
1500
2000
2500
3000
3500
# of Cores
PETSc-CASK
PETSc

CASK in examples

- We have an example program to help show how CASK can be used to speed up petsc and trilinos.
- Compile the program using the Cray Petsc distribution
- Compile the program using the local CASK-free PETSc.
- Compare the performance

CASE – Cray Adaptive Simplified Eigensolver

- Eigensolvers are extremely complicated to use
- Often require quite complicated callign sequences
- Also often require complicated work array set-up
- CASE is a simplified interface into the existing eigensolers
- CASE is also an adaptive framework to use a faster eigensolver

CASE details

- real and complex, serial and parallel wrappers for eigensolvers

- Very simple overloaded/generic interfaces
  - Use a fortran module (‘use case’ in fortran file)
  - Use a C++ header (c users)
- Creates all work arrays for you

- Deduces form the arugments that you pass what type of functionality you require, and calls the best eigensolver for the problem you want
- Can also get adaptive eigensolver by setting CASE_USE_FASTEST

LibSci_acc - Cray scientific Library for Accelerators

- GPU and hybrid library execution
- BLAS, LAPACK, FFT, Sparse MV
- BLAS is tuned via the auto-tuning framework
- LAPACK is tuned to avoid as much of the communications cost as possible
- FFT is tuned assuming that the
- If you want to obtain accelerated library codes, add –lsci_acc to the link (likely a xtpe-accel module will be available), then relink.
- Later I will show some examples of BLAS tuning via the ATF

The future of LibSci

- Work with the code developers (you) to make applications scale to the next level
- Prepared to go outside of the bounds of what library vendors normally provide
  - Specialization model, and auto-specialization with training runs
  - Kernel auto-tuning using a framework for advanced users
- Highly optimized hyrbid libraries for CPU and Accelerator

1. specialization

- Scientific Libraries are tuned in the general sense
- Tuned in general – reasonably good general purpose performance, plus tunings of popular sizes
- Problem : nobody uses ‘popular sizes’
- Knowing the specifics of a calling problem is extremely useful in applying the best tunings

2. Application Training runs

- A near-future approach towards highly specialized libraries is the use of a training model
- A framework derived from ATF is used and offers the general entry points to the user
- The first time an application runs on t

3. Application Auto-tuning

- A tool such as ATF could be available to the user so that he can auto-tune his own application

- We have some anecdotal evidence that the auto-tuning approach can be used outside of numerical libraries

- This then does not only apply to the libraries

Exercises (please use PGI compiler for all)

1. Compile a program that calls dgemm (or use blas_test.c and example0.c)
2. Show that you are picking up dgemm from the correct libsci
3. Run with multi-threading libsci on different numbers of threads and report times
4. Write a simple dgemm and compare against libsci (or #include <example1.c>)
5. Write a blocked matmul and compare against the simplest (or use exercise2.c
6. Write a blocked matmul that calls dgemm, comare again (or use exercise3.c)
7. Add rudimentary threading to it – report the times. (example4.c)
8. Note that in the last example, dgemm is called within a loop, what is happening
9. What is the ideal block-size for our simple tuned dgemm?
10. Write a simple test of scalapack LU or Cholesky or QR (or use example5.f)
11. Run on multiple nodes, one scalapack grid point per core
12. Run on multiple nodes, one gridpoint per node, using the threaded BLAS
13. What is the ideal configuration?
14. Compile the CASK example. CASK-PETSC/PETSC/src/ksp/ksp/examples/tutorials/ex2.c
15. Run against the non-tuned PETSc in these folders
16. Run the CASK example against Cray libsci and note performance difference .
17. Change the run-time options to get better performance
18. Try the CRAFFT example programs. (example7.c)