THE UNIVERSITY of York

Parallel matrix multiplication and diagonalization

Matt Probert
August-Wilhelm Scheer Visiting Prof TUM 2015
Condensed Matter Dynamics Group
Department of Physics,
University of York, U.K.
http://www-users.york.ac.uk/~mijp1

Overview

- Matrix multiplication
  - Serial optimization
  - Blocking and caching
  - Parallel matrix multiplication
- Matrix diagonalization
  - Serial algorithm
  - Parallel algorithm
- Summary

Coding for Speed

Modern PC Hardware

- Modern CPU cores have hardware features such as pipelines and superscalar architecture so can do a number of FLOP per clock cycle
  - Works well as long as uninterrupted flow of data
- But main memory DRAM is slow to access
  - Hence introduction of CPU caches – small but fast memory near CPU
  - And read-ahead with cache lines to fill caches
  - Helps bridge gap between CPU and DRAM

Implications for programming

- Need to make efficient use of cache
  - Memory access patterns should be predictable so cache re-ahead helps
  - Simplest if use unit stride
  - Avoid pointers
  - Need to re-use data in cache as much as possible before ejecting it
- Maximize vectorization potential of code
  - Avoid conditionals & dependencies in loops
  - Allows CPU to use SIMD instructions

Multi-Dimensional Arrays

- In FORTRAN, A(m,n) is stored as

- But in C, A[m][n] is stored as

- And want stride-1 access through arrays for spatial locality, hence THIS IS AWFUL:

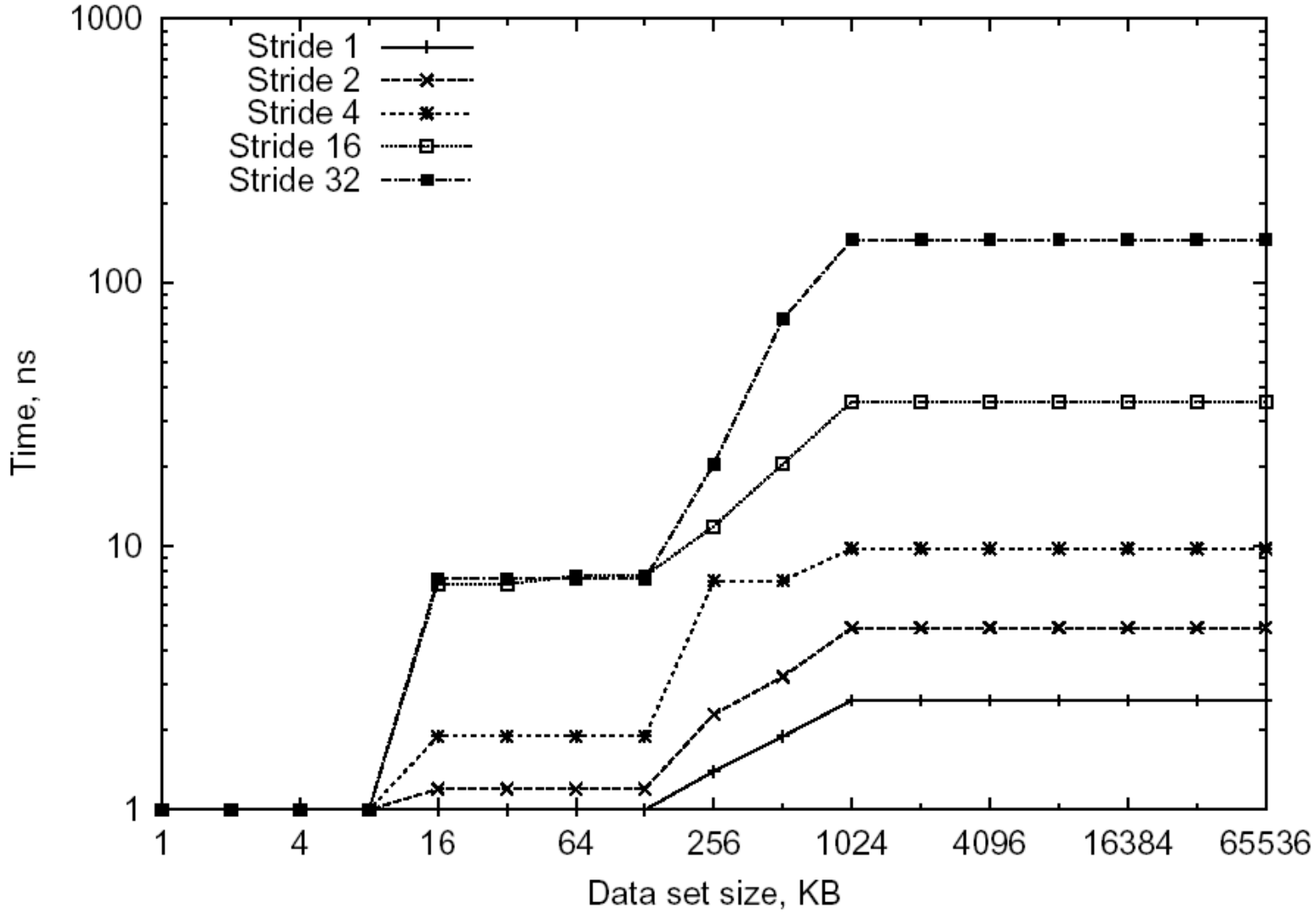
```
do i=1,n
 do j=1,m
  A(i,j)=B(i,j)+C(i,j)
 end do
end do
```

This is accessing memory with stride-m, hence unless entire A,B,C fit into cache this will run very slowly.

Worse still, can get cache thrashing if each line read into cache replaces the existing one – hence beware 2^n array sizes.

MUST reorder these loops – gives x17 speedup on 2.4 GHz P4!

Memory Stride and Performance



2.4 GHz P4 with 8kB L1 and 512 kB L2 caches ...

Matrix Multiplication

Matrix multiplication

- A simple computational kernel
- Used in many different places within CASTEP and other codes

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix}$$

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj} \, .$$

- What coding and hardware factors effect how fast this goes?

Matrix Multiplication $c_{ij} = a_{ik} b_{kj}$

- F77 version
  - Number of FLOPS is $2n^3$ yet performance is appalling:
  - Timings on my 2.26 GHz Macbook (9.04 GFLOP peak):

gfortran –O0, n=100 results in 241 MFLOPS – only 2.7% of peak!

```
!Std F77 version

do j=1,n
   do i=1,n
      t=0.0
      do k=1,n
         t=t+a(i,k)*b(k,j)
      end do
      c(i,j)=t
   end do
end do
```

Why? The inner loop contains 1 FP-add, 1 FP-multiply, 1 FP-load with unit stride (b) and 1 FP-load with stride-n (a).

Each array is 100*100*8 bytes = 78kB. Core 2 Duo has a 3 MB L2 cache so all arrays should fit in L2 cache. Why is the code so slow then?

Fast Matrix Multiplication

- Reorder operations so all memory access now unit stride
  - Timings on my 2.26 GHz Macbook (9.04 GFLOP peak) :
  - gfortran –O0, n=100, results in 200 MFLOPS?!

```
!Fast F77 version

c=0
do j=1,n
  do k=1,n
    t=b(k,j)
    do i=1,n
      c(i,j)=c(i,j)+a(i,k)*t
    end do
  end do
end do
```

Why? This new routine now has unit stride for all arrays – good – but one extra store. As all the arrays fit into cache there is no speedup due to the stride, no saving in FLOPS and one extra store => small extra cost.

BUT this approach should be better as N increases and go out of cache …

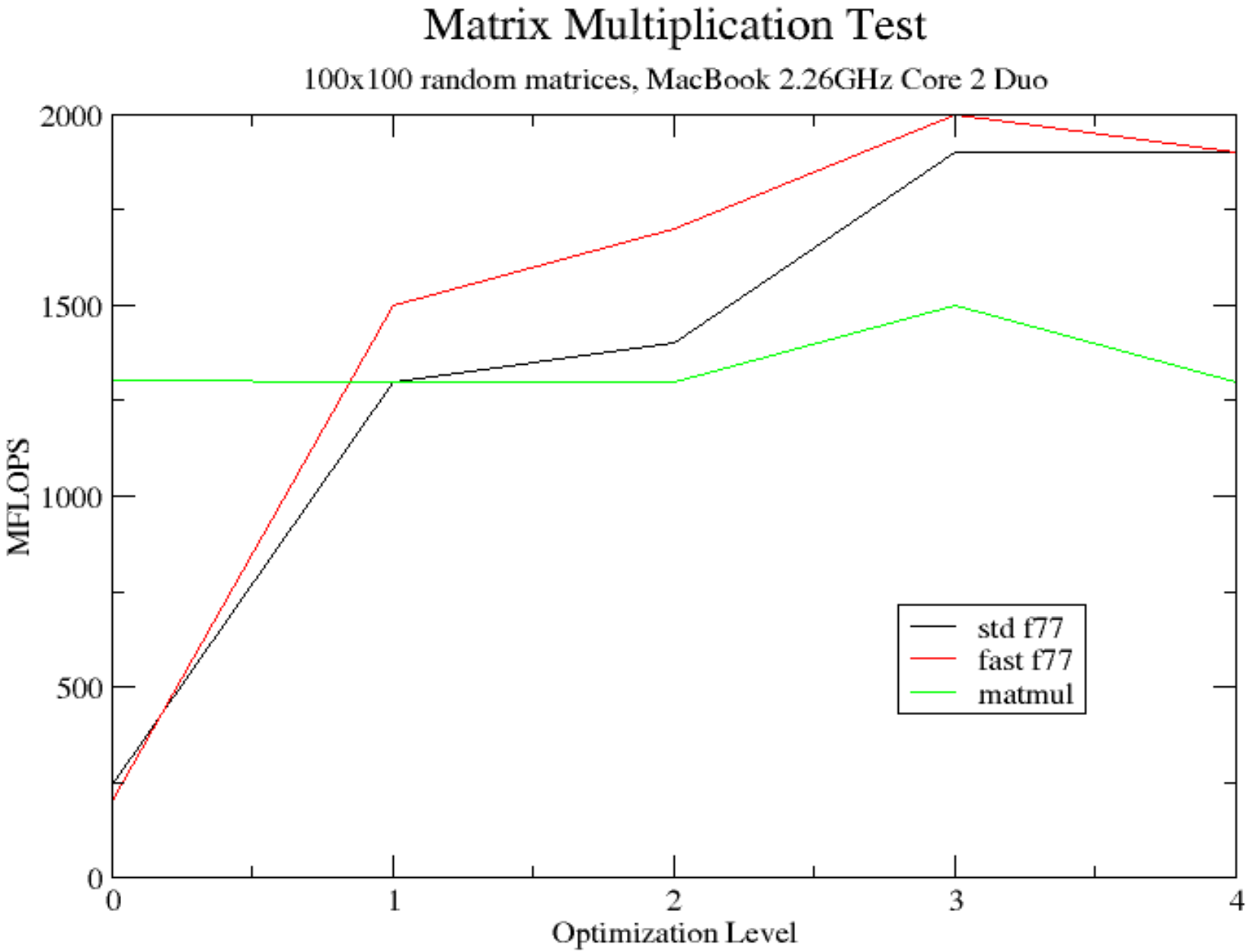
F90 matmul?

```
!F90 form
c=matmul(a,b)
```

- Would seem to be the no-brainer solution

  n=100, 1302 MFLOPS!

- Now up to 15% of peak
  - Better but still pretty poor, particularly as everything is in cache
- What are we missing?
- Compiler flags ...


Matrix Multiplication Test
100x100 random matrices, MacBook 2.26GHz Core 2 Duo
MFLOPS
2000
1500
1000
500
0
0
1
2
3
4
Optimization Level
std f77
fast f77
matmul

BLAS version

```
!BLAS form
call dgemm('N','N',m,n,k,alpha,A,m,B,k,beta,C,m)
```

- `dgemm` is part of BLAS and can evaluate

$$c_{ij} = \alpha \cdot a_{ik} b_{kj} + \beta \cdot c_{ij}$$

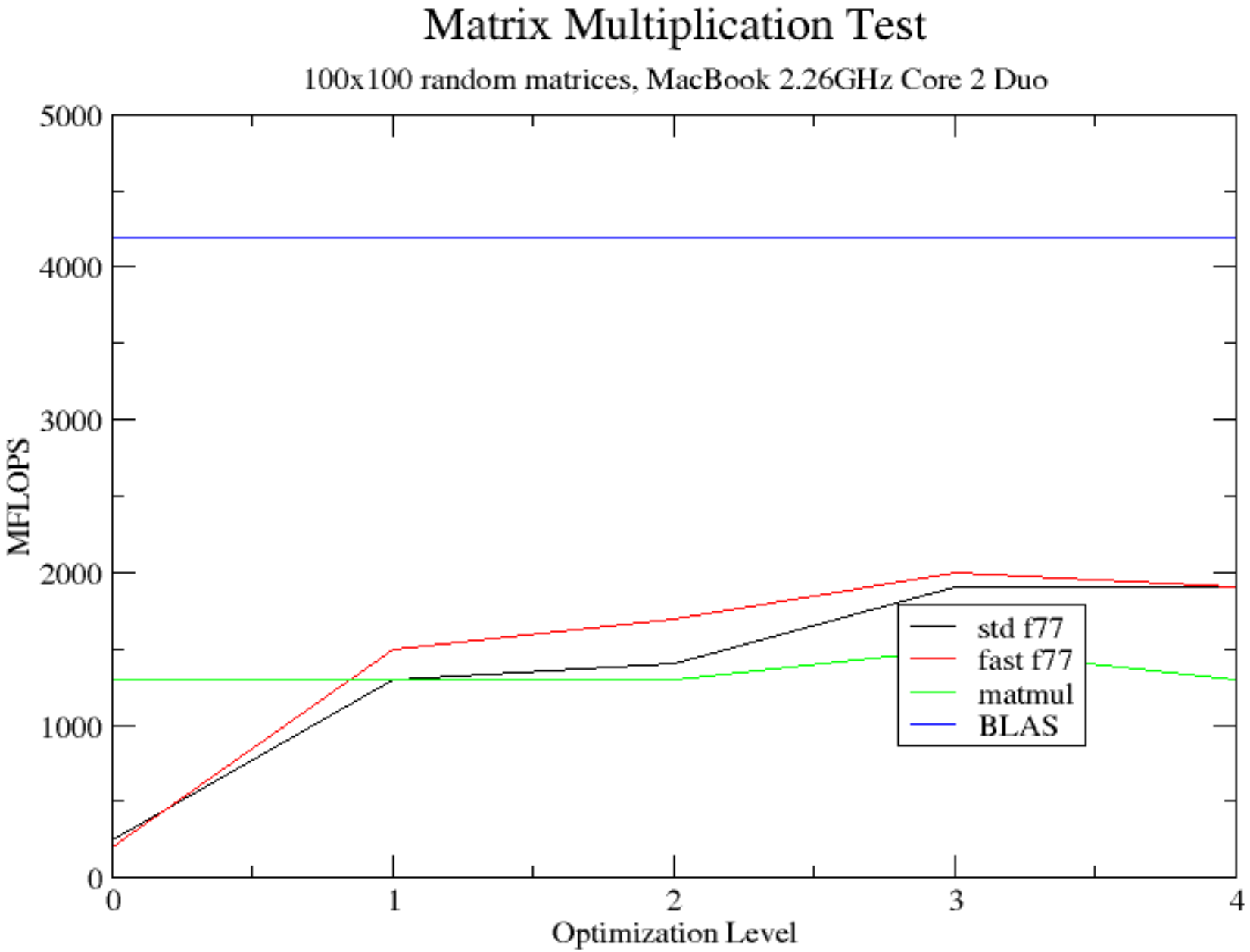
where A is of size MxK, B is KxN and C is MxN , and

A,B,C, alpha & beta are all declared as double precision

- Now have gfortran –O0, n=100 resulting in 4194 MFLOPS ~ 46% peak
- And pretty insensitive to compiler optimisation – as it should be!

NB This is with generic BLAS – using a more optimized BLAS e.g. ATLAS or OpenBLAS should be better.

NB an Mrows x Ncolumns array is declared in Fortran as A(1:M,1:N) but consecutive memory locations are *rows*


Matrix Multiplication Test
100x100 random matrices, MacBook 2.26GHz Core 2 Duo
MFLOPS
5000
4000
3000
2000
1000
0
0
1
2
3
4
Optimization Level
std f77
fast f77
matmul
BLAS

Effect of Problem Size

- The n=100 matrix multiplication is not a good test of different algorithms
  - On modern CPUs the arrays fit in cache.
  - But it does show superiority of BLAS
- What happens as increase problem size and start to go out of cache?
- What is BLAS doing better?
  - Uses *blocking* to get better access pattern

Blocking Approach

Cache Blocking

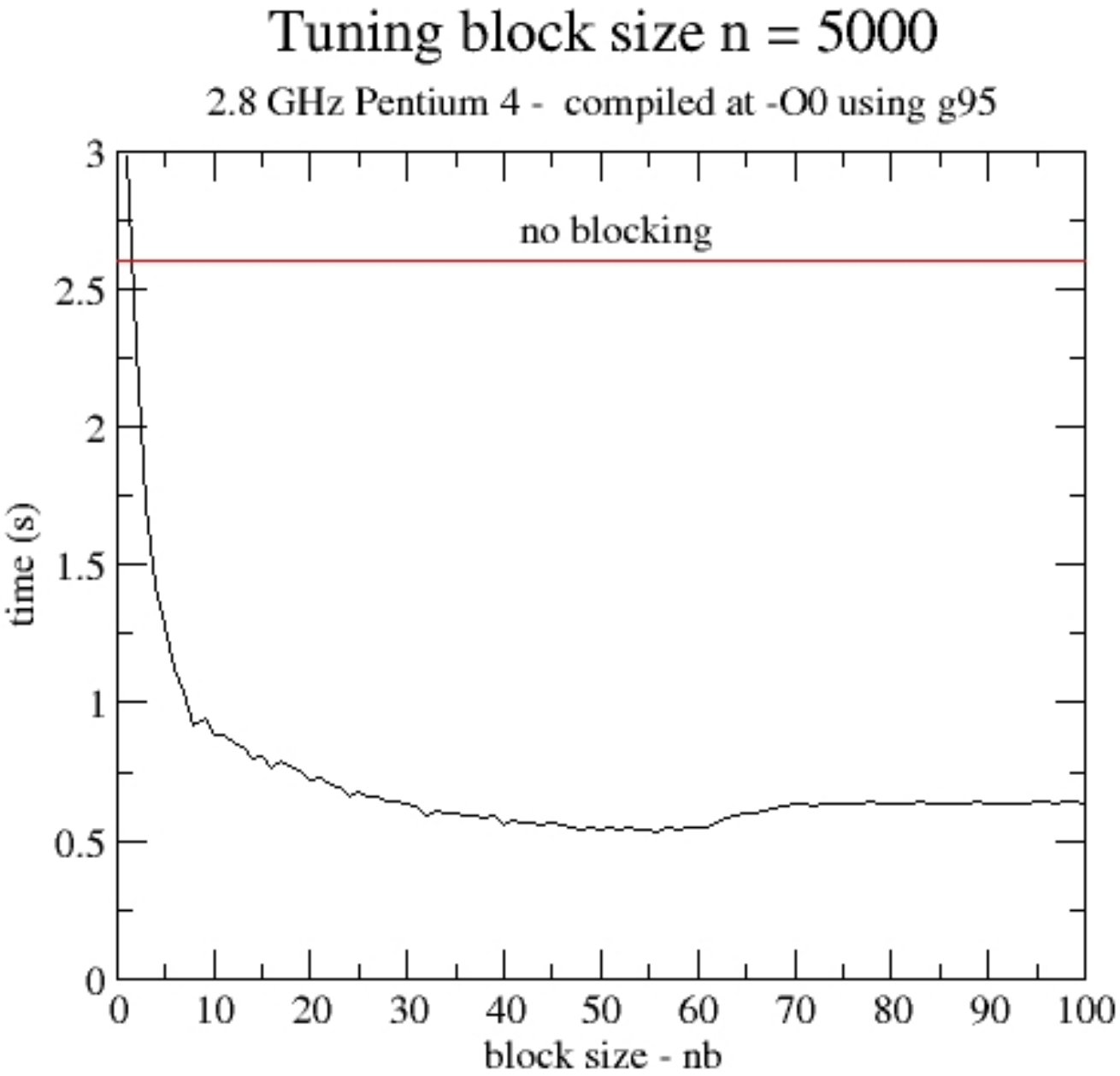
- Do as much as possible with data in cache before returning it to main memory
  - Can be useful with non-unit stride too:

```
!simple non-blocked code
do j=1,n
   do i=1,n
      s=s+a(j,i)+b(i,j)
   end do
end do
```

`a` is accessed with stride n – bad!

```
!blocked-style code
do ii=1,n,nb
   do j=1,n
      do i=ii,ii+nb-1
         s=s+a(j,i)+b(i,j)
      end do
   end do
end do
```

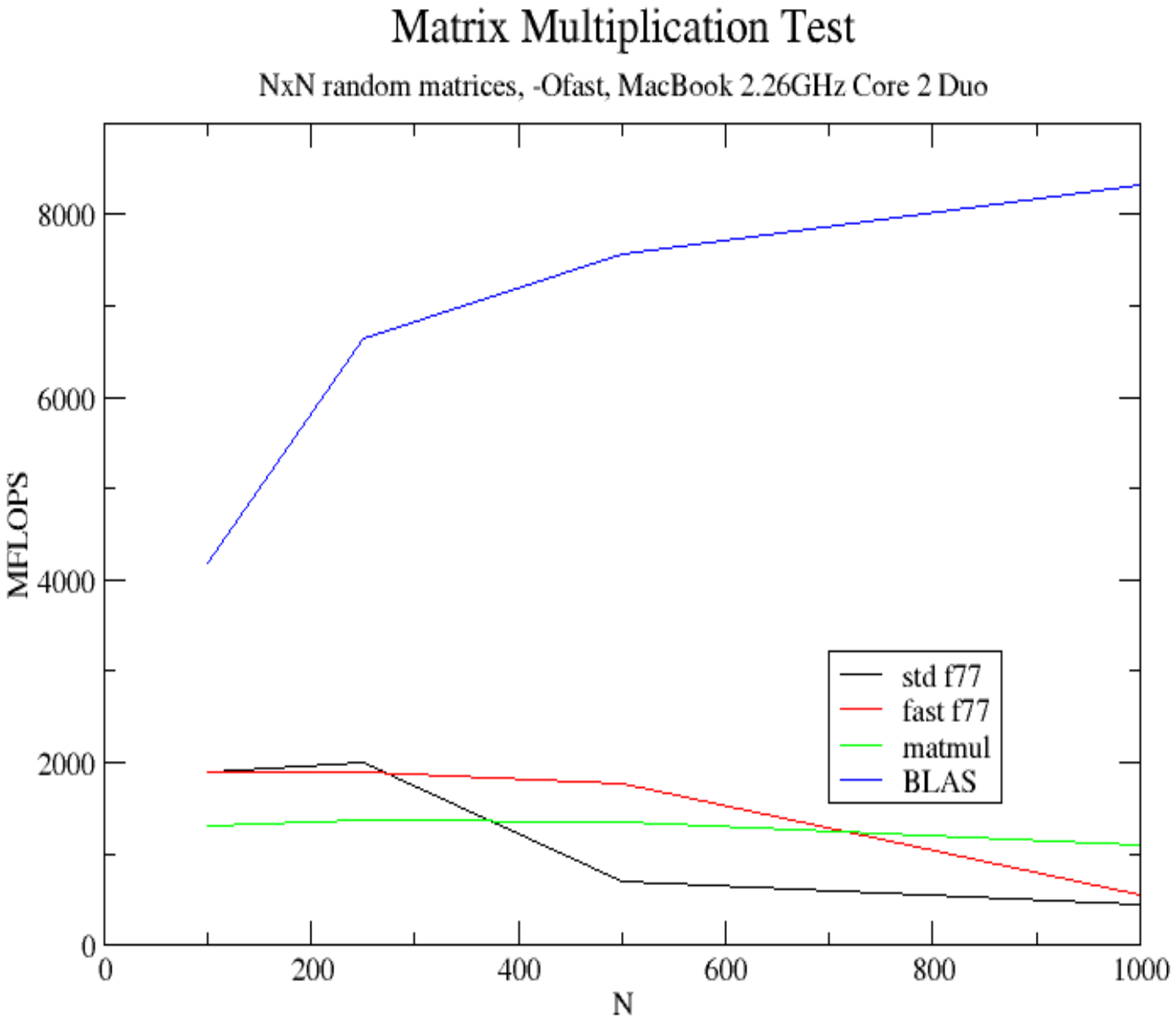
`a` still accessed with stride n but only within blocks of size nb x n. Fast if block fits in cache.



N.B. If compile at –03 then time = **0.012s** without blocking.

Compiler does a much better job than manual blocking in this simple case.

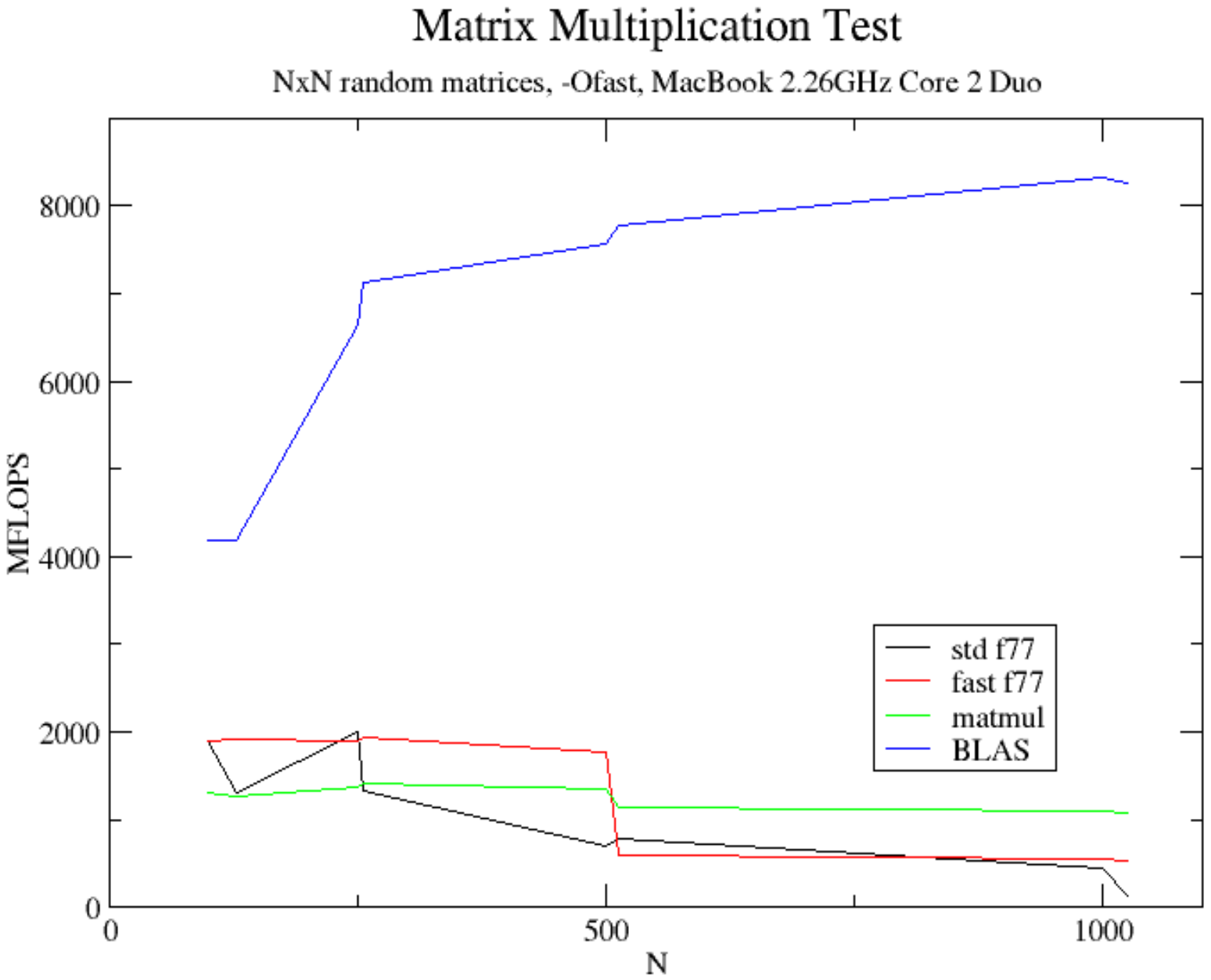
Best benefits from manual blocking occur at much higher level.



Runs with N=100, 200, 500 & 1000

NB n=500 has 1.9 MB per matrix and Core 2 Duo has 3MB of L2 cache

Can now clearly see the effects of non-unit stride on performance.

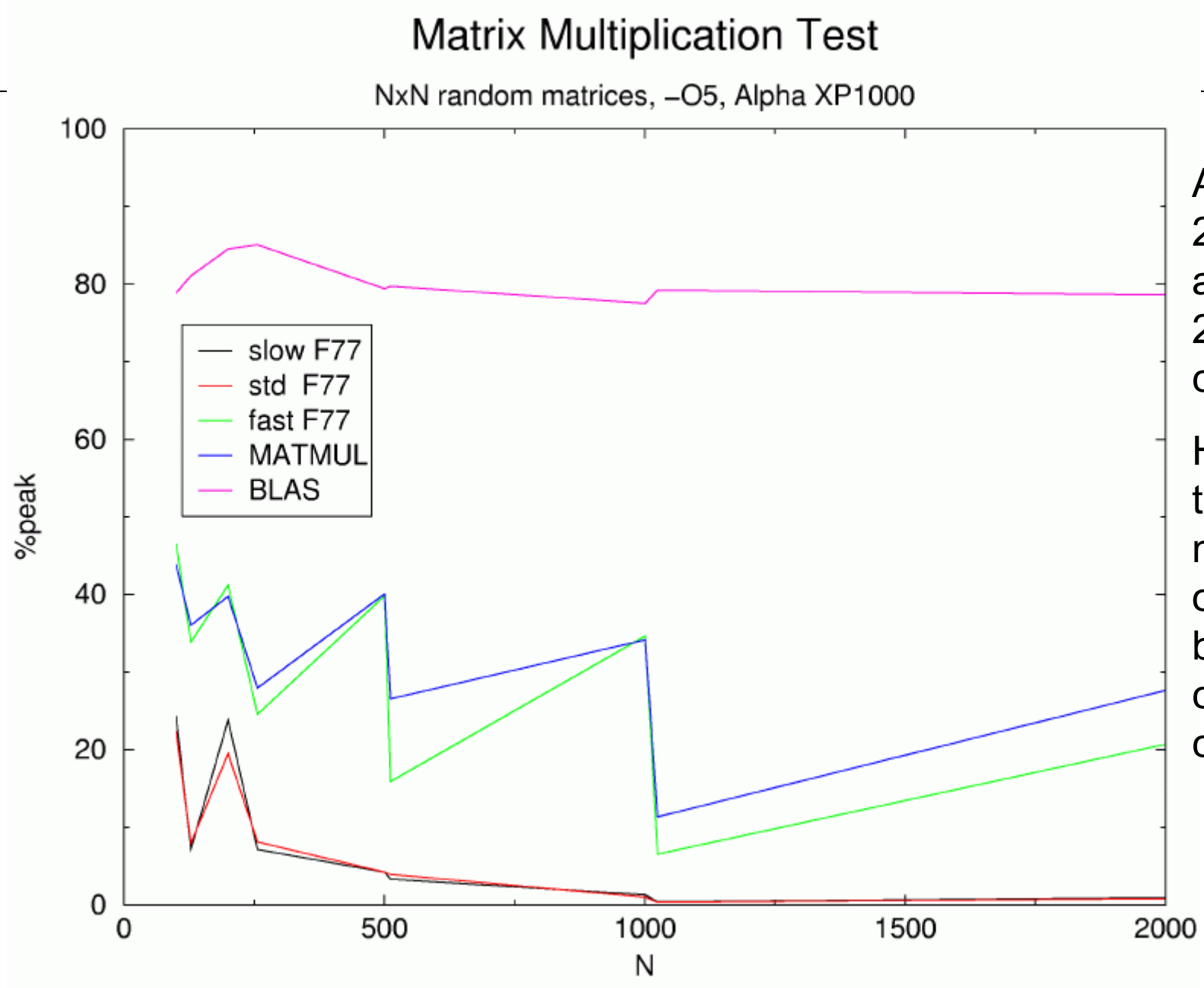


As before but now including n=128, 256, 512 and 1024

What is going on?

Cache Thrashing

- Problems with powers-of-2 array sizes
  - Cache thrashing – successive memory accesses actually go to same line in cache.
- Core 2 Duo has a 8-way set associative L1 and L2 cache made up of 64 byte lines
  - so 8 possible locations in cache for each memory address which reduces thrashing.
- Older CPUs had 2-way set associative or even direct-mapped caches – made effects of cache thrashing much worse!



Alpha has 2-way set associative 2MB L2 cache.

Hence thrashing more obvious – benefit of cache policy clear!

Parallel Matrix Multiplication

Parallel Matrix Multiplication

- Can we speed up matrix multiplication by using multiple cores?
- Yes – use *block matrix* approach and then *divide and conquer:*

$$\begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{pmatrix}$$

- Here have split into 8 smaller matrices
  - Now have 8 independent products
  - Can also repeat at finer levels …
  - Repeat until smallest block matrix is 1/3 of cache size – result is ~$O(N^2)$ in parallel

Cost of Parallel MatMul

- Serial MatMul time ~$O(N^3)$ and storage ~$O(N^2)$
- Parallel MatMul with P cores can reduce storage to ~$O(N^2/P)$ by using 2D mesh
- Comms: general data sent ~$O(N^2/\sqrt{P})$ per message with P messages per core
  - P=4: need to send 2 block matrices to 2 neighbours and receive from 2 neighbours
- Sweet spot depends on comms bandwidth, latency, number of cores & cache sizes
- Could use BLACS but not helpful in CASTEP

Other Optimizations

- Matrix multiplication ~$O(N^3)$ if square matrices
  - ~$O(MNP)$ for rectangular *MxP* with *PxN*
- Strassen's algorithm is faster ~$O(N^{log_2(7)})$
  - Uses *block matrices* recursively as before:

    $M_1=(A_{11}+A_{22})(B_{11}+B_{22})$, $M_2=(A_{21}+A_{22})B_{11}$ ….
    $M_7=(A_{12}-A_{22})(B_{21}+B_{22})$

    $C_{11}=M_1+M_4-M_5+M_7$ … $C_{22}=M_1-M_2+M_3+M_6$

    => Result in 7 multiplications not 8
  - With small caches this was faster for N>100 but on modern machines need N>1000

Matrix Orthogonalization

Matrix Orthogonalization

- In CASTEP we want to ensure that all the bands are orthogonal
  - Important to get the right states occupied!
  - Construct the band overlap matrix *S*
  - Then do a *sub-space rotation* to find new set of orthogonal bands
  - A key step, computationally costly, as saw yesterday.
  - How is it actually done?

Orthogonal Matrix Basics

- We have a set of linearly independent vectors $\{\boldsymbol{v}_i\}$ that span the solution space
- Then *orthogonalization* makes a new set $\{\boldsymbol{u}_i\}$ where each pair is orthogonal and normalized and so $\boldsymbol{u}_i.\boldsymbol{u}_j=\delta_{ij}$ (Kronecker delta)
- Write vectors as columns in a matrix $\boldsymbol{U}$:

$$U^T U=UU^T = I \text{ i.e. } U^T = U^{-1}$$

- Then U represents a rotation
  - Is unitary as does not change length of vectors

Orthogonalizing

- How to construct U?
  - Could do Gram-Schmidt projection
  - Or Householder rotation
  - And/or Givens reflection
- Or do matrix factorization
  - Any matrix A can be factorized into LU form:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

Matrix Factorization

- LU decomposition very useful in solving linear equations: $\boldsymbol{A.x=b}$
  - As can rewrite as $\boldsymbol{L.y=b}$ and $\boldsymbol{U.x=y}$ and then solve triangular equations by back substitution
- Cholesky: *iff* $\boldsymbol{A}$ is square positive definite then can also use $\boldsymbol{A = L.L^T}$ or $\boldsymbol{L.D.L^T}$
  - 2x faster than LU but less general
- QR decomposition: $\boldsymbol{A=Q.R}$ where $\boldsymbol{R}$ is upper triangular and $\boldsymbol{Q^TQ=I}$
  - 2x slower than LU but good for orthogonal'n

Matrix Eigenvalues

- But the matrix we want to orthogonalize is Hermitian => special form & properties
  - $\boldsymbol{H} = (\boldsymbol{H}^*)^\mathsf{T} = \boldsymbol{H}^+$
- Hermitian => real eigenvectors & e-values
  - $\boldsymbol{H}.\boldsymbol{x} = \lambda\boldsymbol{x}$
  - And the eigenvectors form an orthogonal basis
  - So can put e-vecs as columns in matrix $\boldsymbol{U}$
  - So finding the eigenvectors ~ orthogonalization

Matrix Diagonalization

- How to do diagonalization?
- Sequence of similarity transforms:

  $\boldsymbol{H}^1 = \boldsymbol{U}_1^T.\boldsymbol{H}.\boldsymbol{U}_1$; $\boldsymbol{H}^2 = \boldsymbol{U}_2.\boldsymbol{H}.\boldsymbol{U}_2$ …

  - where $\boldsymbol{U}$ is chosen as Jacobi or Householder transformation etc
  - And similarity transform does not change det|$\boldsymbol{H}$| or eigenvalues of $\boldsymbol{H}$
- And/or QR factorization
  - $\boldsymbol{H}=\boldsymbol{Q}.\boldsymbol{R}$ => $\boldsymbol{Q}^{-1}.\boldsymbol{H}=\boldsymbol{R}$ => $\boldsymbol{Q}.\boldsymbol{R}=\boldsymbol{Q}^{-1}.\boldsymbol{H}.\boldsymbol{Q}$
  - i.e. QR factorization ~ similarity transform

Practical Serial Diagonalization

- LAPACK routines
  - zheev – all the eigenvalues & eigenvectors of Hermitian matrix
    - Uses LU decomposition
  - Also zheevr – uses LDL^T so faster than LU
  - And zheevd – uses divide and conquer

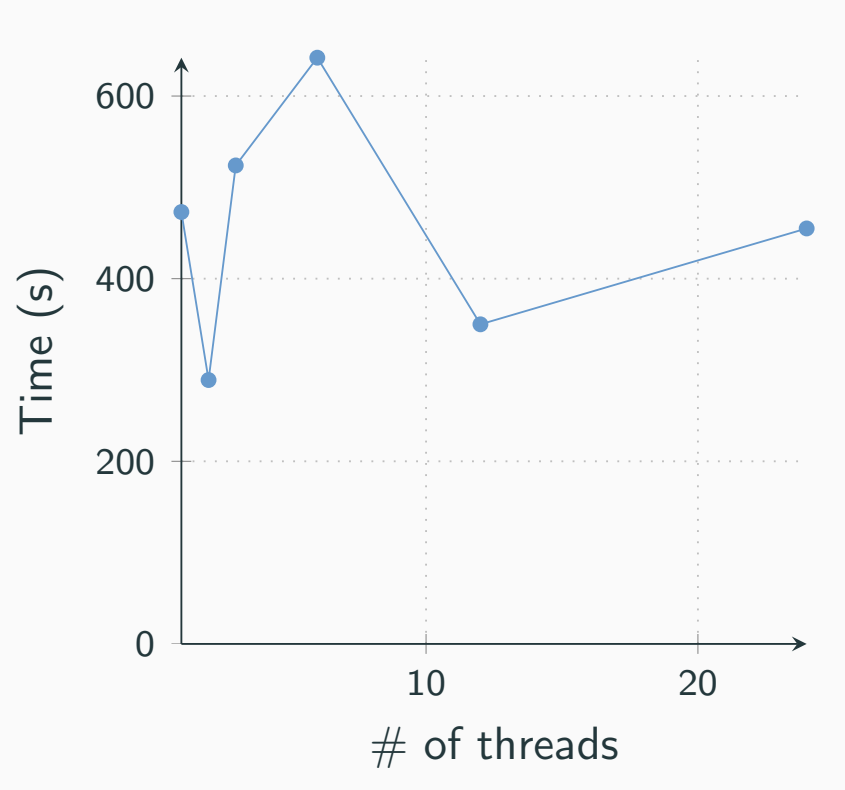
Practical Parallel Diagonalization

- ScaLAPACK does not have right parallel data layout for CASTEP usage
- Hard to parallelize most approaches
- Threaded LAPACK:

Crambin benchmark

small protein

1284 atoms!



Parallel Diagonalization

- Need a different approach in parallel
- Recent project – implement block-factored Jacobi (BFJ) for diagonalization
  - Jacobi largely forgotten in serial as slow
    - At least x10 slower than QR etc
    - But block-factored Jacobi 2x faster than Jacobi
  - BUT much easier to parallelize
    - No pivoting
    - And iterative – can reuse previous guess at eigenvalues to kick-start next matrix

Jacobi eigenvalue algorithm

- Solving $\boldsymbol{H}.\boldsymbol{V} = \lambda \boldsymbol{V}$
- Each off-diagonal element is zero'd every iteration

```
V=I
Do until converged:

   for each off-diagonal element H(i,j):
     find the matrix U that zero's H(i,j)
     H'=conjg(U)*H*U
     V'=          V*U
   end for
End do
```

- Zeroing an element un-zeros previous elements
- Magnitude of off-diagonals reduces rapidly each iteration

BFJ eigenvalue algorithm

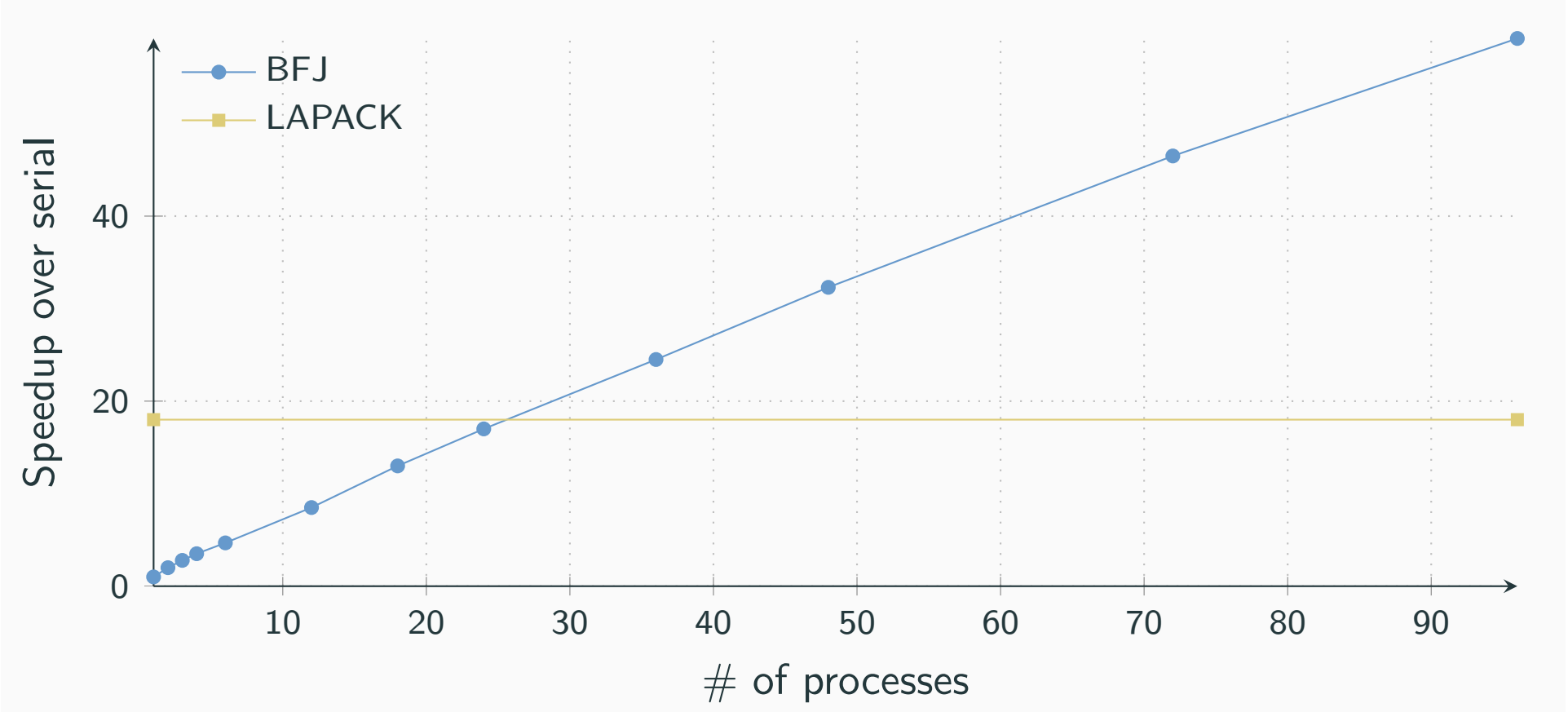
- Solving $\boldsymbol{G}.\boldsymbol{V} = \lambda$ and $\boldsymbol{G}=\boldsymbol{V}^{+}\boldsymbol{H}$

```
V=I, G=H
Do until converged:
   for each pair of cols G(:,i), G(:,j):

     H(i,j) = V(i,:) .dot. G(:,j)

     find the matrix U that zero's H(i,j)

     G'= G*U
     V'= V*U

   end for
End do
Eigenvalues(i)=V(i,:) .dot. G(:,i)
```

- By storing cols of V^T can operate on cols of V and G – caching
- Parallelize over cols

Advantages of BFJ

- Parallel, iterative, controllable accuracy
- Beats best LAPACK if N>700 and N_{cores}>24



Implementation

- Challenge to get eigenvectors to machine precision
  - Found that for a random matrix LAPACK was failing to converge accurately too
  - Had to rework numerics by smart use of trig
- Practical speed advantage: 4096 Silicon test case: LAPACK orthogonalization time ~ 30 minutes, BFJ <20 secs on >500 cores
- Only notice on big systems as ~$O(N^3)$
- Finished last week, going into CASTEP v9

Summary

Summary

- Matrix multiplication a great case study
  - Serial optimization, need to understand modern hardware, choice of algorithms
  - Parallel algorithms use recursive blocking and divide & conquer approach
- Matrix orthogonalization / diagonalization
  - Well known in serial – choice of approaches
  - Exploit matrix properties if Hermitian etc
  - Much harder to parallelize …

References

- MC Payne et al., Rev. Mod. Phys **64**, 1045 (1992)

- WH Press et al, "*Numerical Recipes: The Art of Scientific Computing*", Cambridge University Press (1989 – 2007)

- RJ Littlefield and KJ Maschoff, Theor. Chim. Acta **84**, 457 (1993)