THE UNIVERSITY of York

High Performance Computing - Optimizing a Serial Code

Prof Matt Probert

http://www-users.york.ac.uk/~mijp1

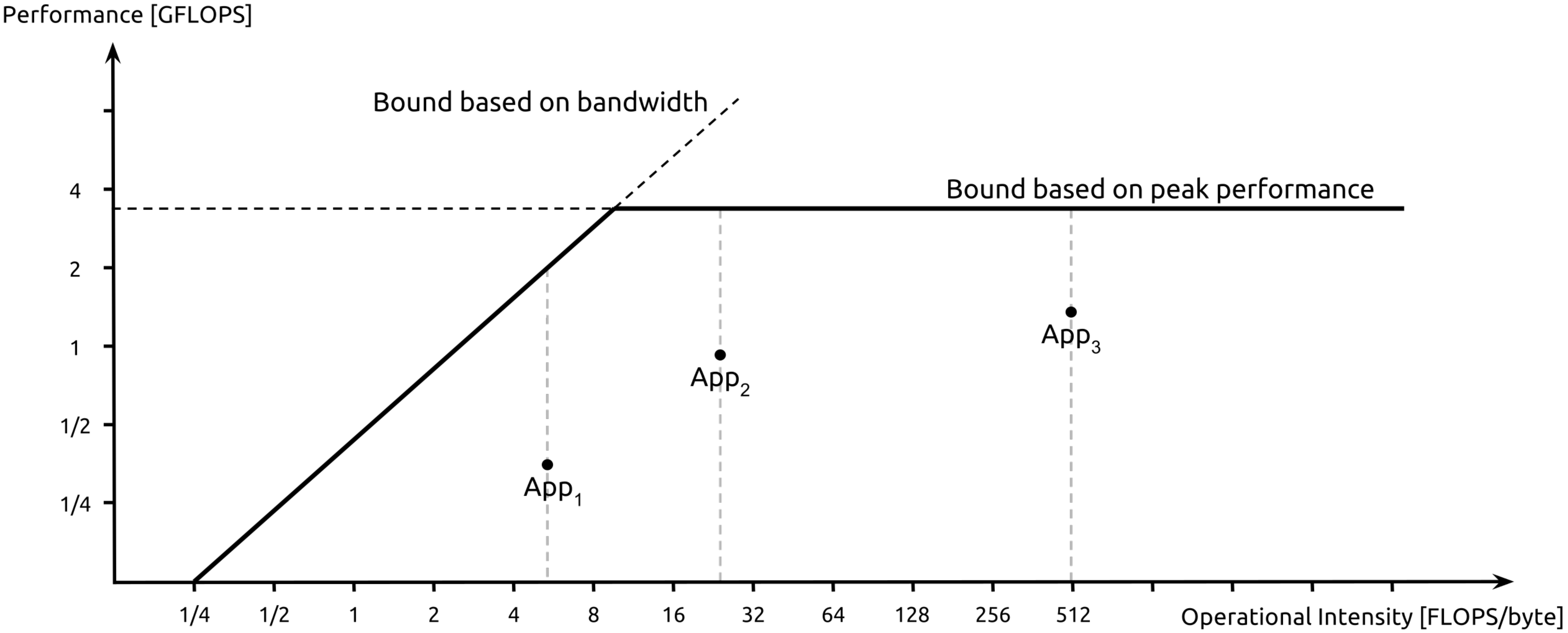
Overview

- Performance potential
- Compiler optimizations
- Manual optimizations
- Case study – matrix multiplication

Roofline Model

- A simple model to provide performance estimates
  - What determines the maximum FP available?
  - Is a code compute-bound or memory-bound?
- Constraints are:
  - Peak performance π (GFLOP/s)
  - Memory bandwidth β (GB/s)
- Inputs are:
  - Arithmetic intensity I (FLOPS/B)
- Performance $P=\min(\pi,\beta*I)$

Visualization



Picture from https://en.wikipedia.org/wiki/Roofline_model

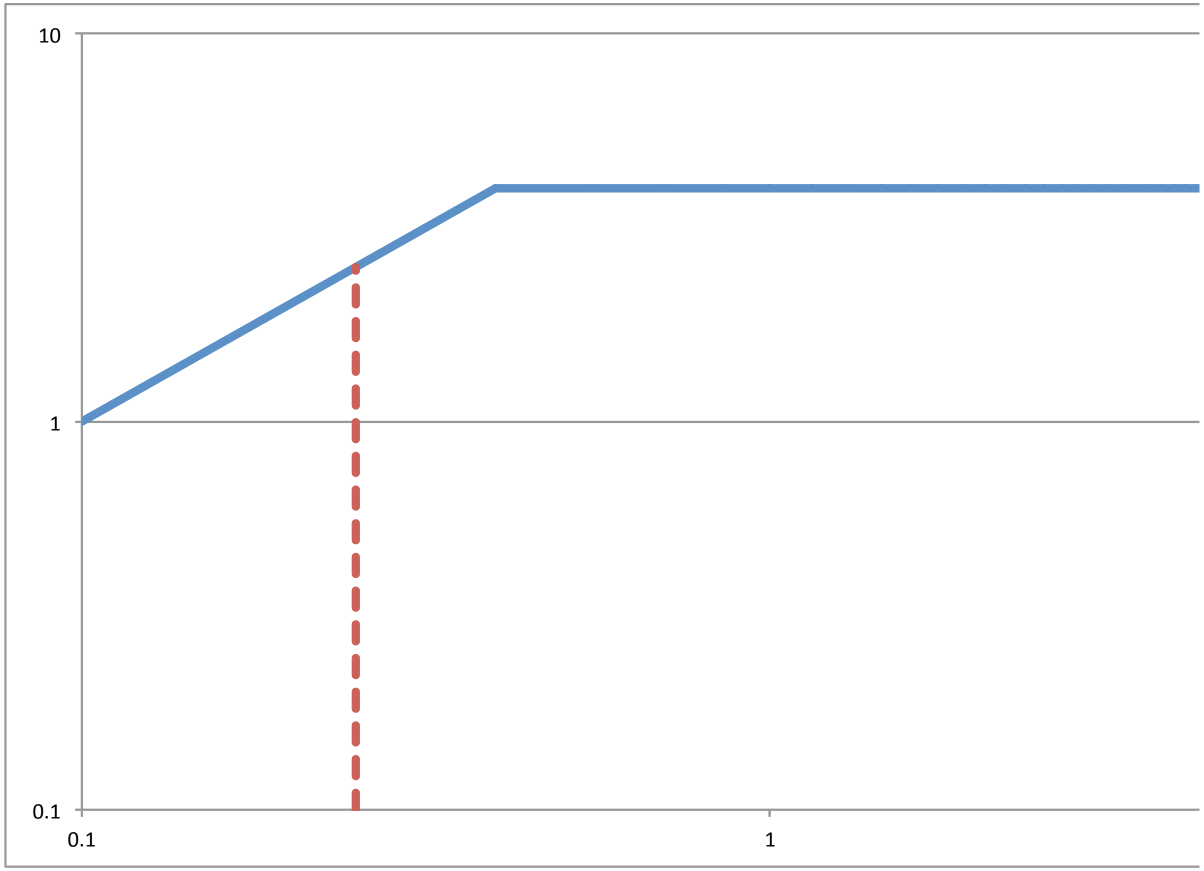
Comments on Roofline

- Ridge is where $\pi=\beta*I$
- An important hardware metric is the *machine balance*: $B=\pi/\beta$
- The algorithmic intensity I is also known as the "*memory-code balance*" : B_c
- If $B_c<B$ then have *memory-bound algorithm*

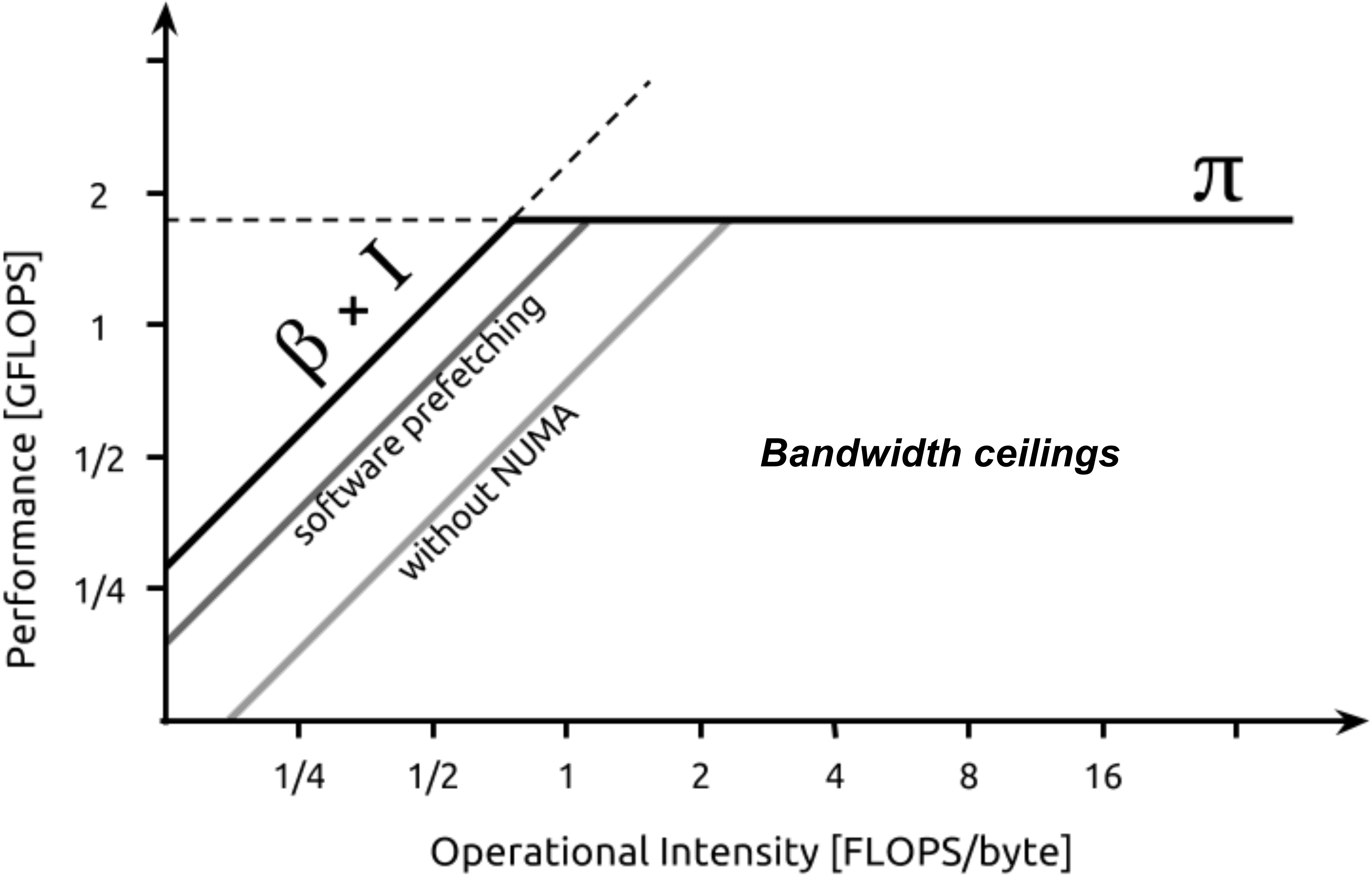
Simple Roofline Example

```
double s, a[];
for (i=0, i<N, i++){
      s = s+a[i]*a[i]}
```

- Peak CPU performance π = 4 GF/s
- Memory bandwidth β = 10 GB/s
- I = 2F/8B = 0.25
- Memory bound!

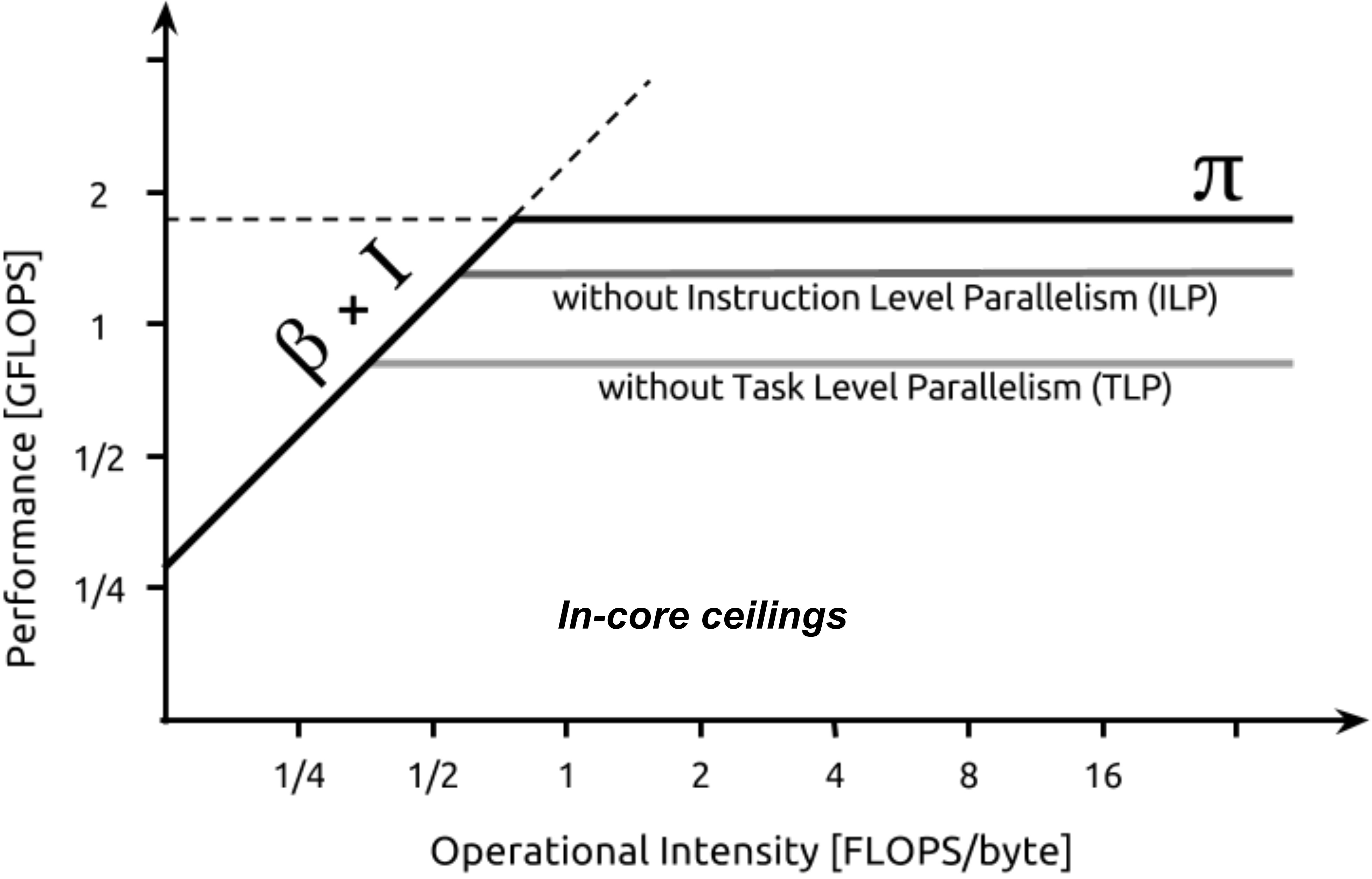


Extended Roofline Models (I)



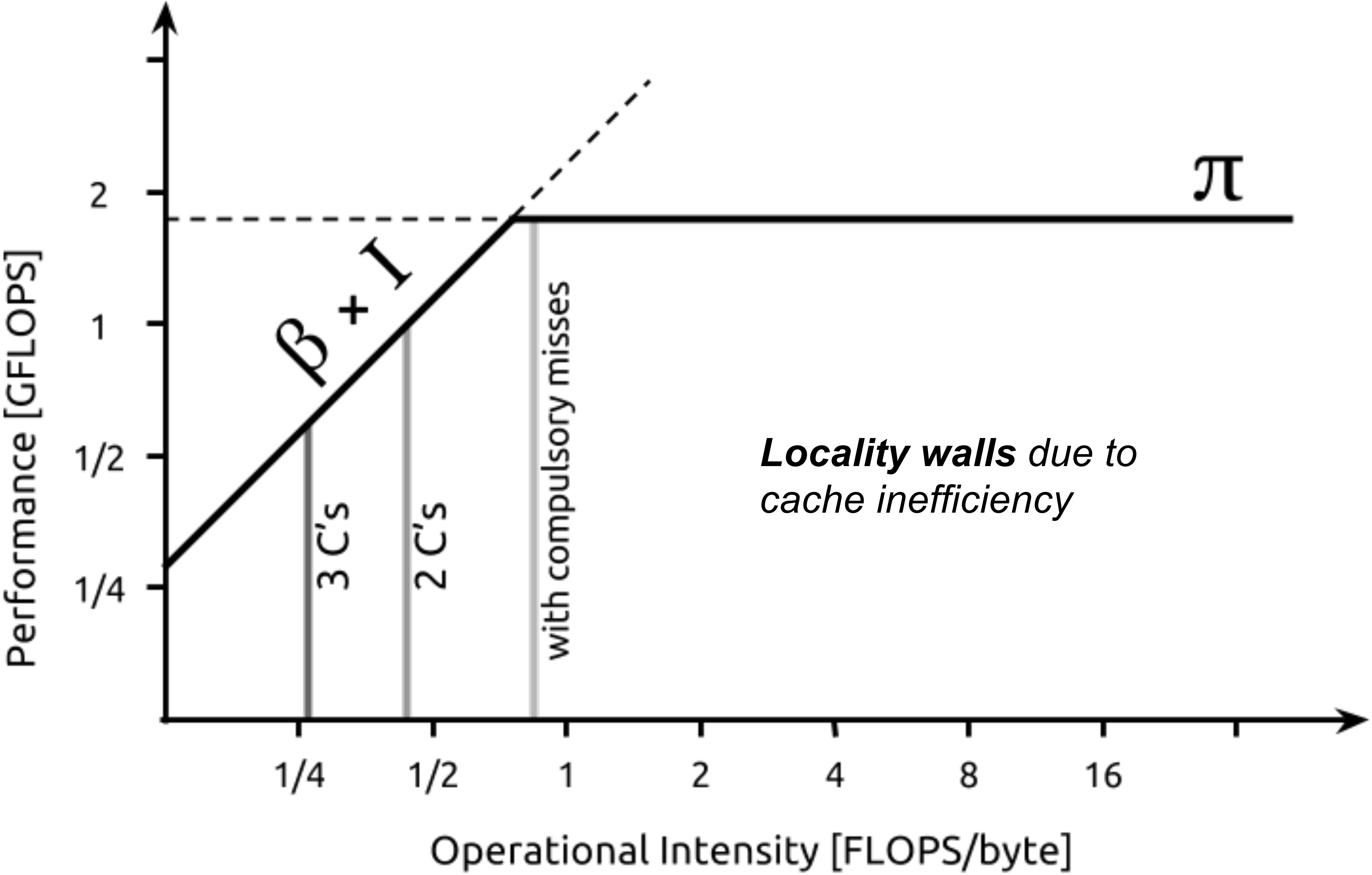
Bandwidth ceilings

Extended Roofline Models (II)



In-core ceilings

Extended Roofline Models (III)



Locality walls *due to cache inefficiency*

Optimizing Compilers

- Most modern compilers are good at basic code optimisations (but not memory opts)
- Usually best to let the compiler do the optimisation
  - Avoid machine-specific coding – a tweak that improves performance on one machine may degrade performance on a different architecture
  - Compilers break code less often than people!
- Not all optimisations are beneficial!
  - Might break code – more so if not strictly standards compliant
  - Might reduce accuracy of answers – floating-point reordering ...

Aliasing

- The compiler has to make assumptions about *aliasing*
  - Can one or more variables occupy same space in memory?
  - Common blocks/equivalence in F77
  - Pointers in C, C++, F90
- Aliasing prevents many optimisations
  - A fundamental reason why Fortran often optimises better than C as Fortran codes typically have much less reliance on pointers
  - Can sometimes set a compiler flag (e.g. `-no_alias` in ifort/icc) to tell compiler that there is no aliasing present

`-fstrict-aliasing` in gcc/gfortran is not the same.

Helping the Compiler

- Write clear and simple code
  - Easier for compiler to spot optimisation potential
  - Use flags and directives to give compiler hints
- But what if that is not enough?
  - Need to resort to code modification
  - Need some idea as to what code modification the compiler would like to do but cannot
  - Get compiler to produce an optimisation report
    - Intel has `-opt-report` and GNU has `-fopt-info-note`

Local vs Global variables

- Compiler analysis is more effective with local variables
  - Has to make worst-case assumptions about global variables – they might be modified by any called procedure
  - Hence always use local variables where possible
    - NB Automatic variables are allocated on the *stack* – low cost but limited amount of space
    - Dynamic allocation (F90 allocate, C malloc, C++ new) goes on *heap* – usually much larger
  - In C, use file scope globals in preference to externals

Key Optimizations

- MEMORY is most often the bottleneck in modern computers
  - Very obvious in roofline models
  - Both bandwidth (MB/sec) and latency (time to read a single value)
  - Need to optimise memory access patterns
    - memory structures
  - Need to optimise cache usage
    - loop structures
  - Need for good code design by programmer – not something the compiler can do!

Spatial Locality

- An important factor in cache success: the assumption that nearby addresses tend to be accessed close together in time
  - If `y(i)` is read now then it is very likely that `y(i+1)` will be read soon
  - Hence cache controllers read whole lines of memory not single bytes
  - Hence best if move through memory in sequential order – implications for array indices – Fortran stores arrays in *column order*, C/C++ in *row order*

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 in Fortran!

Enhancing Spatial Locality

- Place items which are accessed in the same block of code close to each other
  - E.g. careful use of structures
  - Don’t include infrequently used variables in structures – avoid clutter – and aim for alignment with cache block boundaries
  - Avoid gaps in structures
    - Compiler will add gaps to ensure address of variable is aligned with its size for maximum efficiency in address translation
    - Hence place items of same size next to each other – best if put all doubles first, then integers, etc.

Temporal Locality

- Another important factor in cache success: if a data item has been recently accessed, then it is likely to be read again, or written, soon.
  - Once the cost has been paid of getting an item into cache, use it as much as possible before returning it to main memory
  - E.g. *Loop fusing* – classic example:

```
do i = 1,N
    av = av + A(i)
end do
av = av/real(N,kind=dp)
do i = 1,N
    var = var + (av-A(i))**2
end do
var = var/real(N,kind=dp)
```

```
do i = 1,N
    av = av + A(i)
    sum_sq = sum_sq + A(i)**2
end do
av = av/real(N,kind=dp)
var = sum_sq/real(N,kind=dp) &
- av*av
```

2 x faster with A ~ 450 Mb

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.

Reducing Memory Accesses

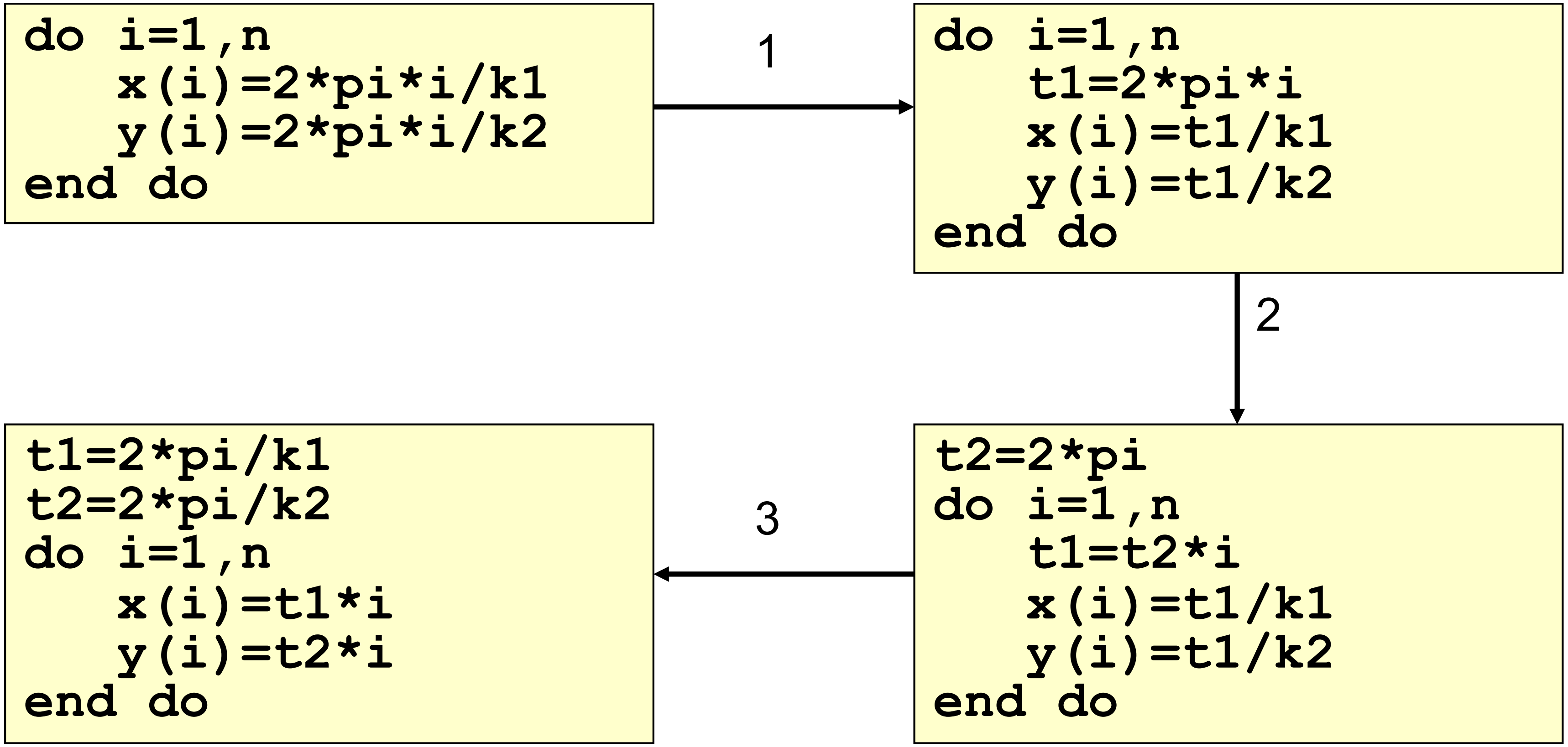
- Many old codes were written in pre-cache days when memory access was cheap
  - Not true today!
- Watch out for and reduce as much as possible:
  - Array temporaries
  - Long trip-count loops with little work in the body
  - Look-up tables of values that are now cheap to recalculate – the balance of calculation cost to lookup cost has now changed

Pointer Problems

- Pointers are useful but can seriously inhibit code performance on modern machines
- Compilers try very hard to reduce memory accesses
  - Only load data from memory once
  - Keep variables in registers for as long as possible and only update memory copy when necessary
- But pointers can point anywhere, so to be safe must:
  - Reload all values after write through pointer
  - Synchronize all variables with memory before read through pointer

- F77 has no pointers, F90+ has restricted pointers
  - Can only point to a pre-declared “target” – more info for compiler
- C/C++ has unrestricted pointers and very hard to do without them – can use explicit scalar temporaries to help

Simple Optimisations



```
do i=1,n
   x(i)=2*pi*i/k1
   y(i)=2*pi*i/k2
end do
```

```
do i=1,n
   t1=2*pi*i
   x(i)=t1/k1
   y(i)=t1/k2
end do
```

```
t2=2*pi
do i=1,n
   t1=t2*i
   x(i)=t1/k1
   y(i)=t1/k2
end do
```

```
t1=2*pi/k1
t2=2*pi/k2
do i=1,n
   x(i)=t1*i
   y(i)=t2*i
end do
```

1. Common sub-expression elimination – compiler OK
2. Invariant removal – compiler OK
3. Division to multiplication – may need to force it! Multiplication much faster than division and many times faster than exp or log. Conversion of x**2 to x*x should be automatic.

NB NEVER do x**2.0!

Common Sub-Expression Elimination

- Whilst compilers are good at spotting the simple example above, might not be so good if:
  - Changed order of operands
    ```
    d=a+c
    e=a+b+c
    ```
  - Or function calls
    ```
    d=a+func(c)
    e=b+func(c)
    ```
  - Hence might need to help compiler by introducing explicit scalar temporaries

Loop Optimizations

- Loop unrolling
  - Useful for reducing dependencies
- Loop elimination
  - Useful for short loops – if know trip count
- Loop fusing
  - As before - increases work done in loop, better cache usage, less overhead
- Loop blocking
  - Can help optimize memory patterns
  - Particularly useful with 2D arrays
  - Similar idea to domain decomposition in parallel codes

Stopping Loop Optimizations

- Conditionals
  - Especially transfer out of loop
  - Eliminate wherever possible, e.g.

```
!original form
do i=1,k
   if (n==0) then
      a(i)=b(i)+c
   else
      a(i)=0
   end if
end do
```

```
!manual rewrite
if (n==0) then
   do i=1,k
      a(i)=b(i)+c
   end do
else
 do i=1,k
      a(i)=0
 end do
end if
```

- Function calls
  - Except if can inline
- Pointer/array aliasing as discussed above

More Impediments …

- Non-obvious data dependencies, e.g.

```
!original form
do i=1,m
   a(i)=a(i)+b(i)*a(n)
end do
```

- Compiler may not know if `a(i)` and `a(n)` overlap or not – hence reduced choice of optimisations
- May be unrolled but only limited benefit as cannot interleave instructions from different iterations
- Hence, if you know it is safe, better to re-write as:

```
!manual rewrite
t1=a(n)
do i=1,m
   a(i)=a(i)+b(i)*t1
end do
```

Case Study

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, stream = 16400 MB/s):

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

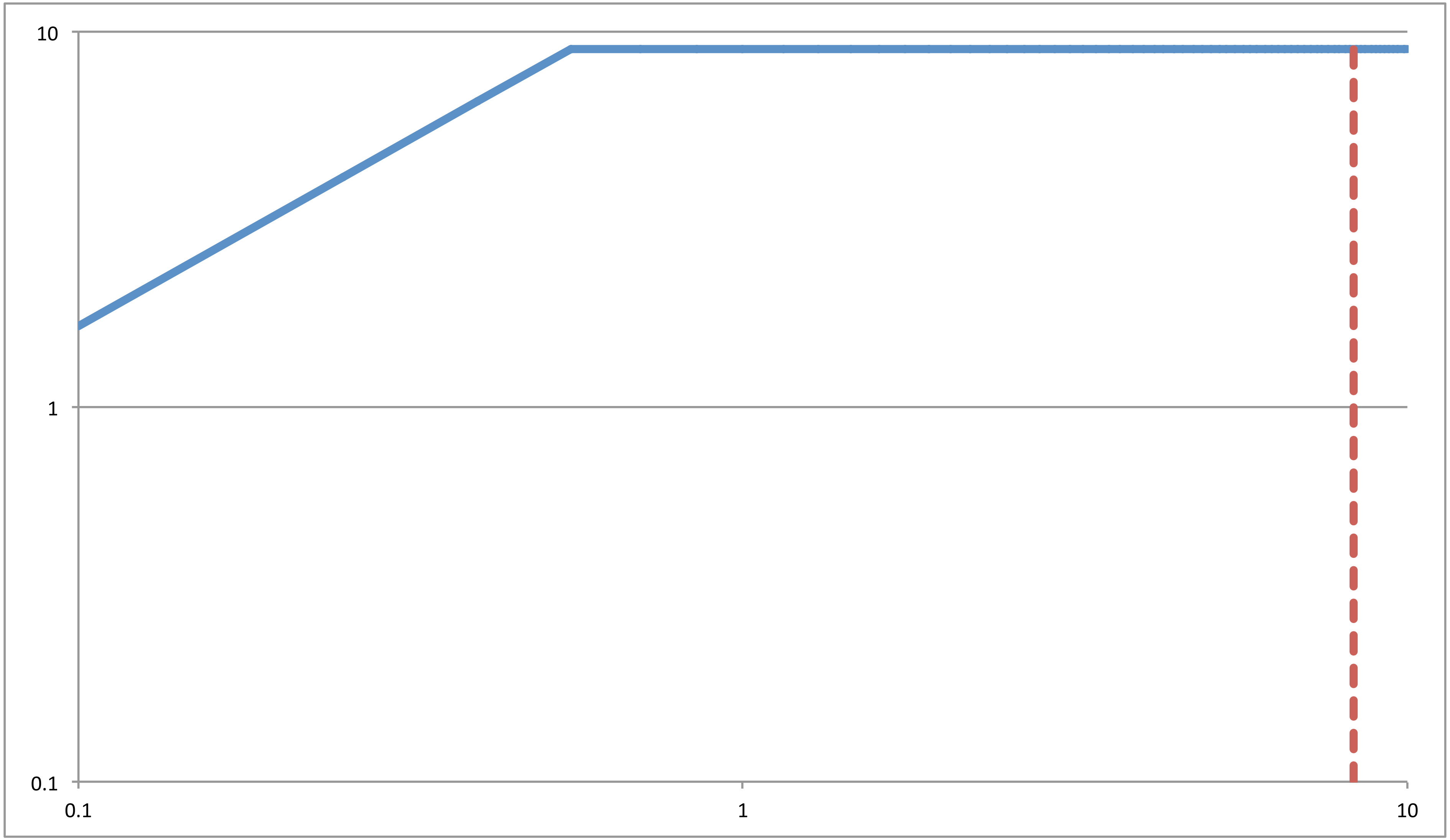
```
!Std F77 version

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?

Matrix Multiplication Roofline



Roofline: π=9.04, β=16.4, $I=(2n^3)/(3*8*n^2)=n/12=8.3$ with $n=100$

=> code ought to be compute-bound, should be able to get near to 9 GFLOPs

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.

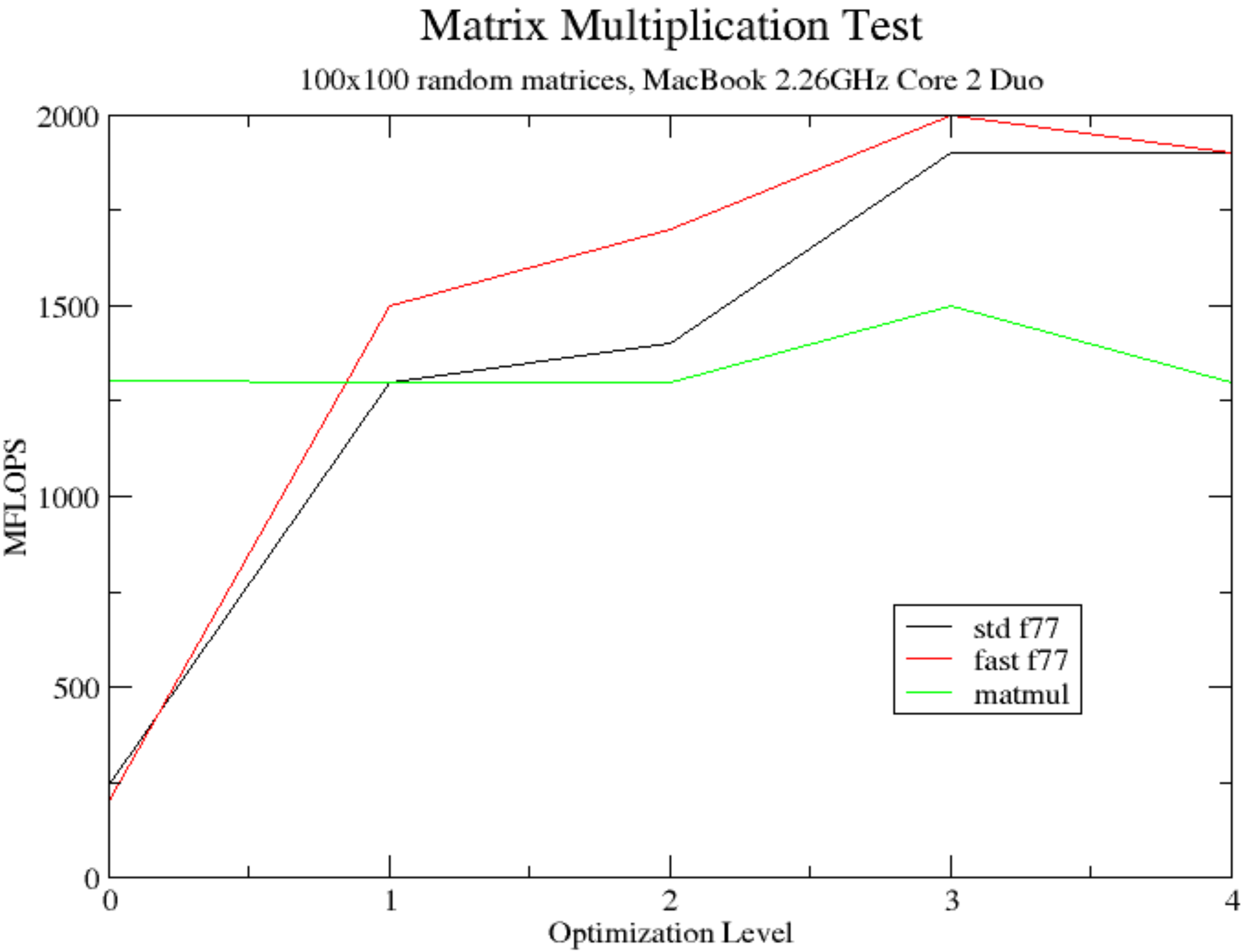
BUT this approach should be better as N increases and go out of cache …

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

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

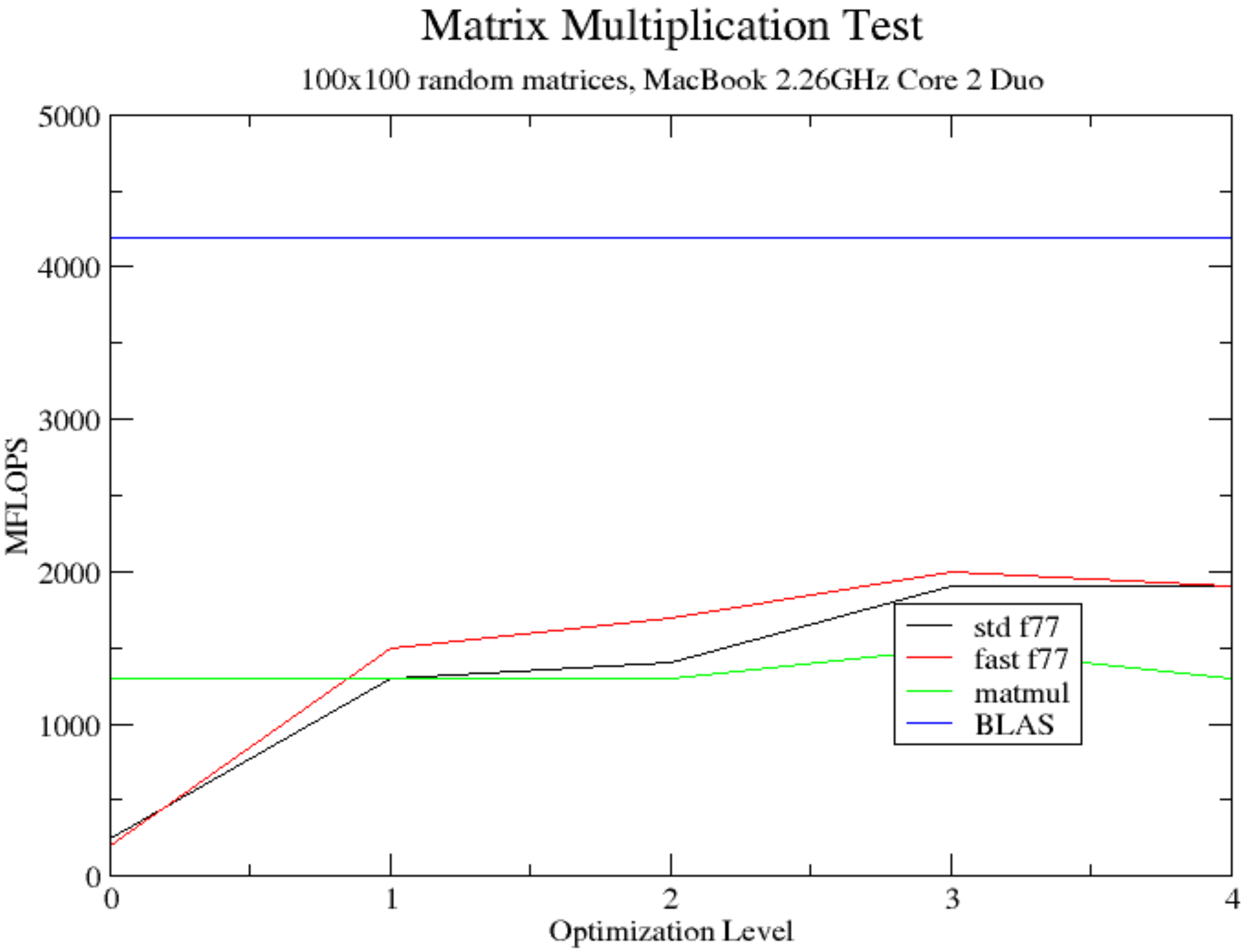
A,B,C, alpha & beta are all declared as double precision

- Now have gfortran90 –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.

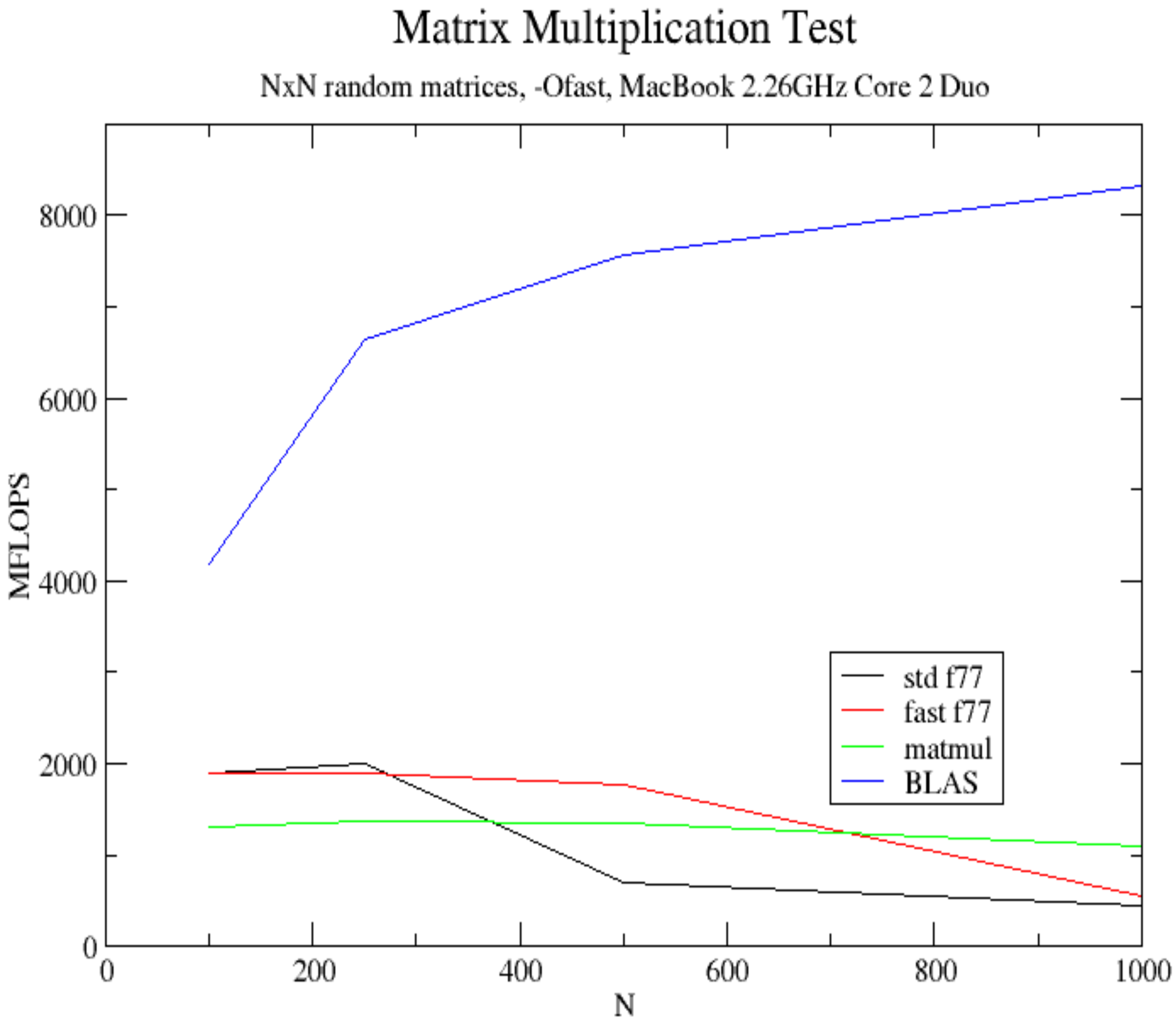
NB an Mrows x Ncolumns array is declared in Fortran as A(1:M,1:N) but consecutive memory locations are *rows*

NB Can use BLAS from C/C++ as well …


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

Effect of Problem Size

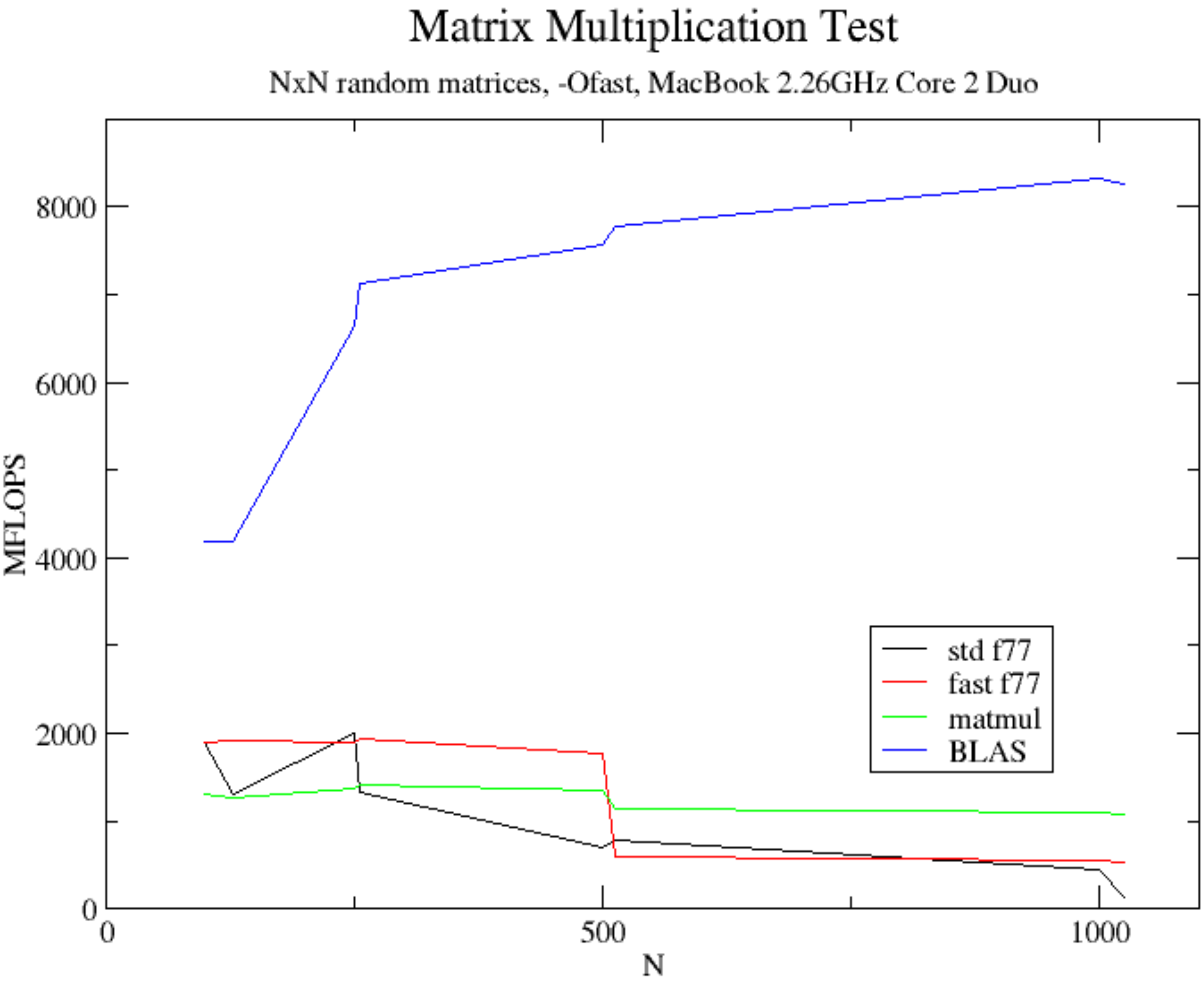
- The n=100 matrix multiplication is not a good test of different algorithms (although it shows the superiority of BLAS) as on most modern computers the arrays fit in cache.
  - Was used as the basis for the original LINPACK 100x100 test but is now obsolete
  - LINPACK based upon LAPACK and used as basis of Top500 supercomputer league!
- What then happens as increase problem size and start to go out of cache?



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?

Further Reading

- https://en.wikipedia.org/wiki/Roofline_model
- Chapter 3 of “Introduction to High Performance Computing for Scientists and Engineers”, Georg Hager and Gerhard Wellein, CRC Press (2011).
- Intel optimization manual at https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html
- http://www.openblas.net
- http://www.netlib.org/lapack/

Cache Thrashing

- Problems with powers-of-2 array sizes are particularly prone to cache thrashing, where 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 there are 8 possible locations in cache for each memory address which reduces thrashing.
- But can still see the (weaker) effects of cache thrashing in previous figure! BLAS probably uses blocking to boost performance for large N.

Libraries

- Not all implementations of a library are equal – better if know about caches etc
- LINPACK 5000x5000 test on a quad-core 3.4 GHz Intel i7 (Haswell)

Implementation	**Speed (GFLOPs)**
MKL (serial)	43.0
OpenBLAS (serial)	39.0
NAG Mk 24	4.9
Netlib	3.3
Fortran, original source	2.4