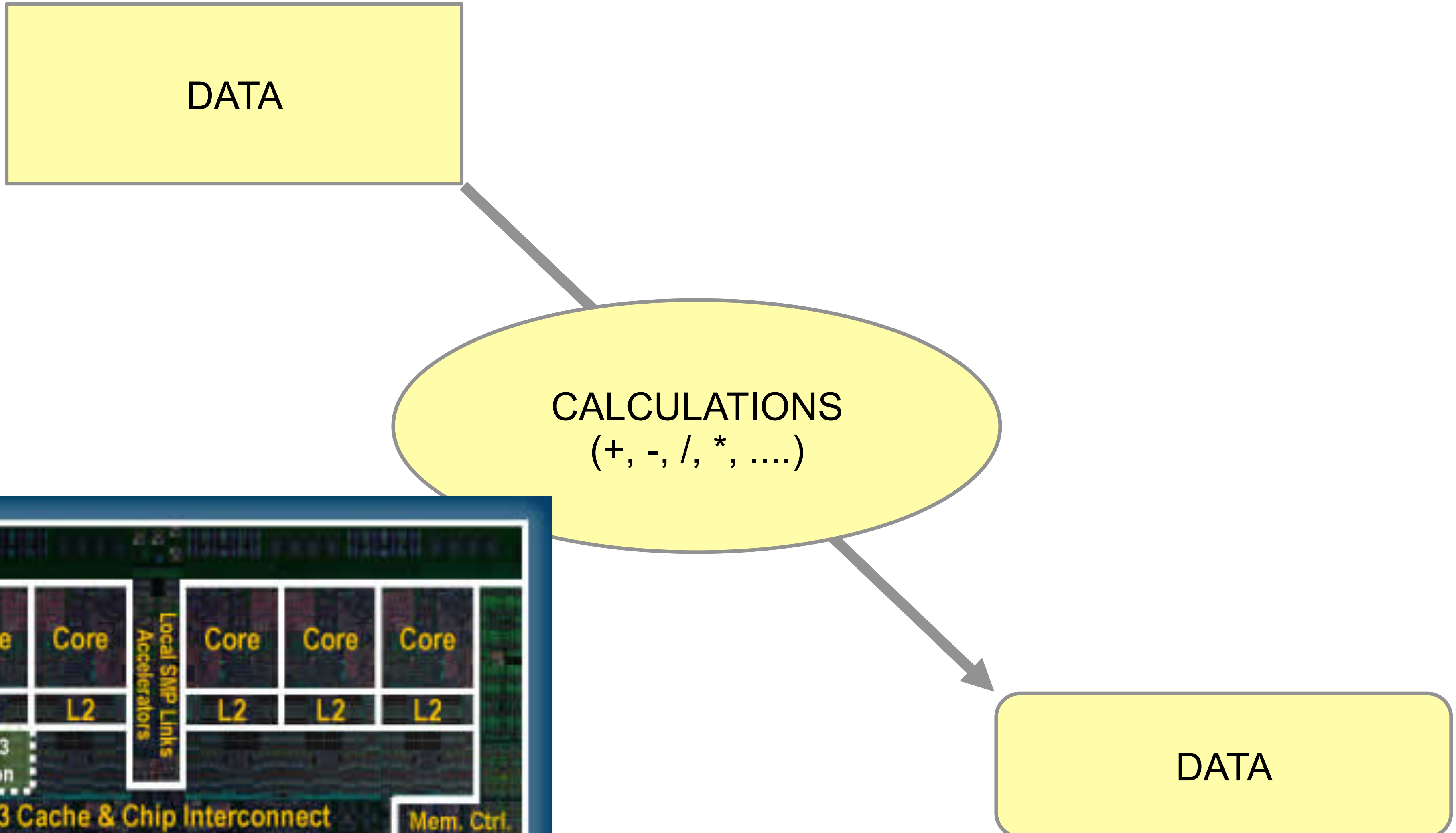
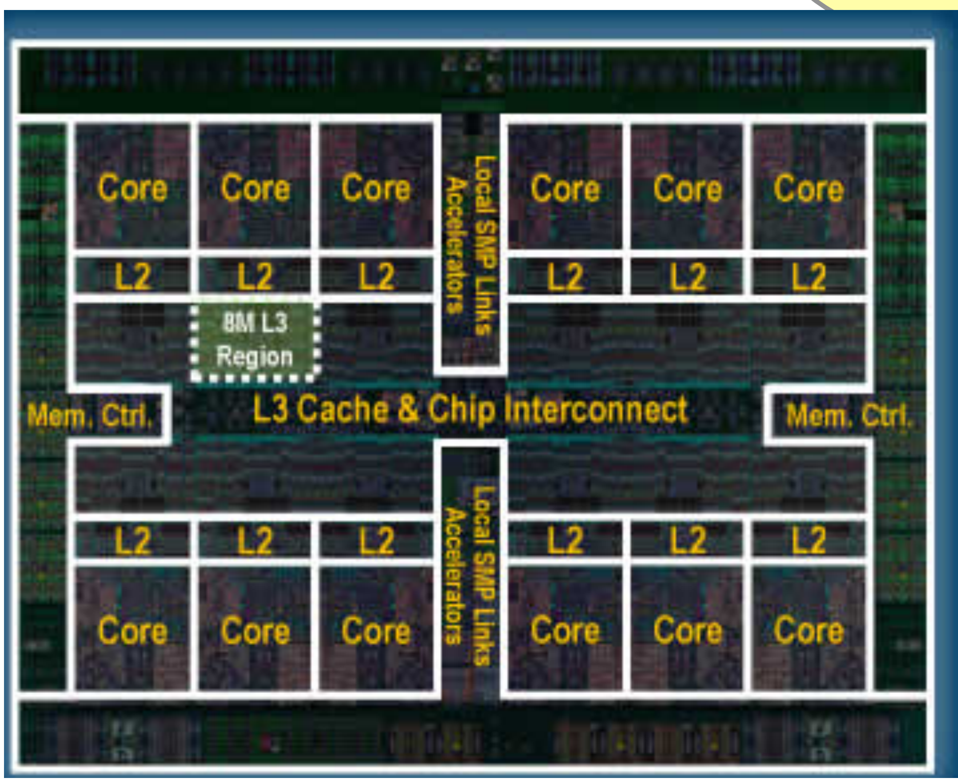
Part 1 : Roofline Model

Instructor: Leopold Grinberg
IBM, T.J. Watson Research Center, USA

e-mail: leopoldgrinberg@us.ibm.com


DATA
CALCULATIONS
(+, -, /, *, ...)
DATA


Core
Core
Core
Local SMP Links
Accelerators
Core
Core
Core
L2
L2
L2
L2
L2
L2
8M L3
Region
Mem. Ctrl.
L3 Cache & Chip Interconnect
Mem. Ctrl.
L2
L2
L2
Local SMP Links
Accelerators
L2
L2
L2
Core
Core
Core
Core
Core
Core

The Roofline Model

- The roofline model was introduced in 2009 by Williams et.al.
  - Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52, 4 (April 2009), 65-76. DOI=10.1145/1498765.1498785 http://doi.acm.org/10.1145/1498765.1498785

- It provides an easy way to get performance bounds for compute and memory bandwidth bound computations.

- It relies on the concept of Computational Intensity (CI) – sometimes also called Arithmetic or Operational Intensity.

- **The Roofline Model provides a relatively simple way for performance estimates based on the computational kernel and hardware characteristics.**

Performance [GF/s] = function (hardware and software characteristics)

FLOPS : Bytes Balance

FLOPS:Bytes ratio is the basic variable of the Roofline model

DATA

CALCULATIONS
(+, -, /, *, ...)

DATA

for (i=0; i < N; i=i+1)
a[i] = b[i]

DATA TRANSFER, NO FLOPS

for (i=0; i < N; i=i+1)
a[i] = b[i]*b[i]+b[i]

DATA TRANSFER, ADDs and MULs

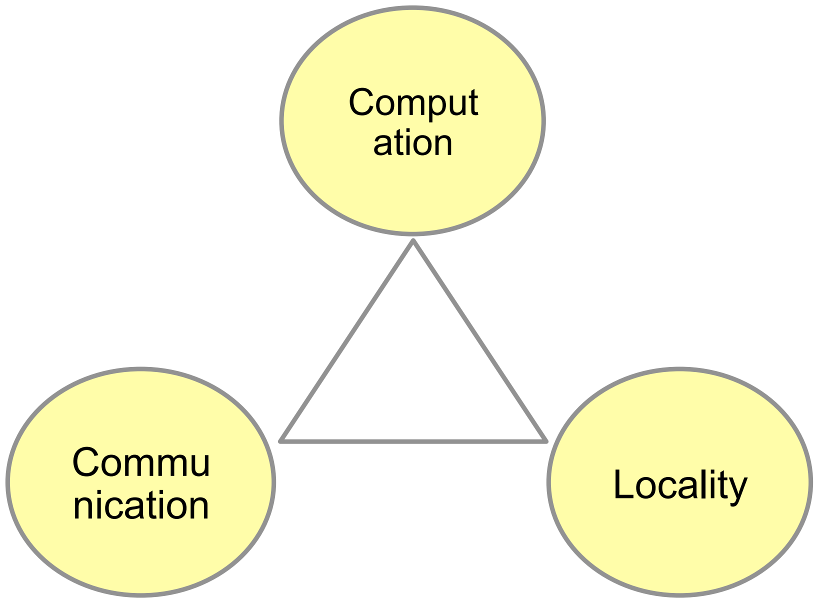
for (i=0; i < N; i=i+1)
a[i] = b[i]*b[i]+sin(b[i])+exp(b[i])

DATA TRANSFER, FLOPS

The Roofline Model: Principal Components to Performance

Performance can be estimated

from hardware and kernel characteristics



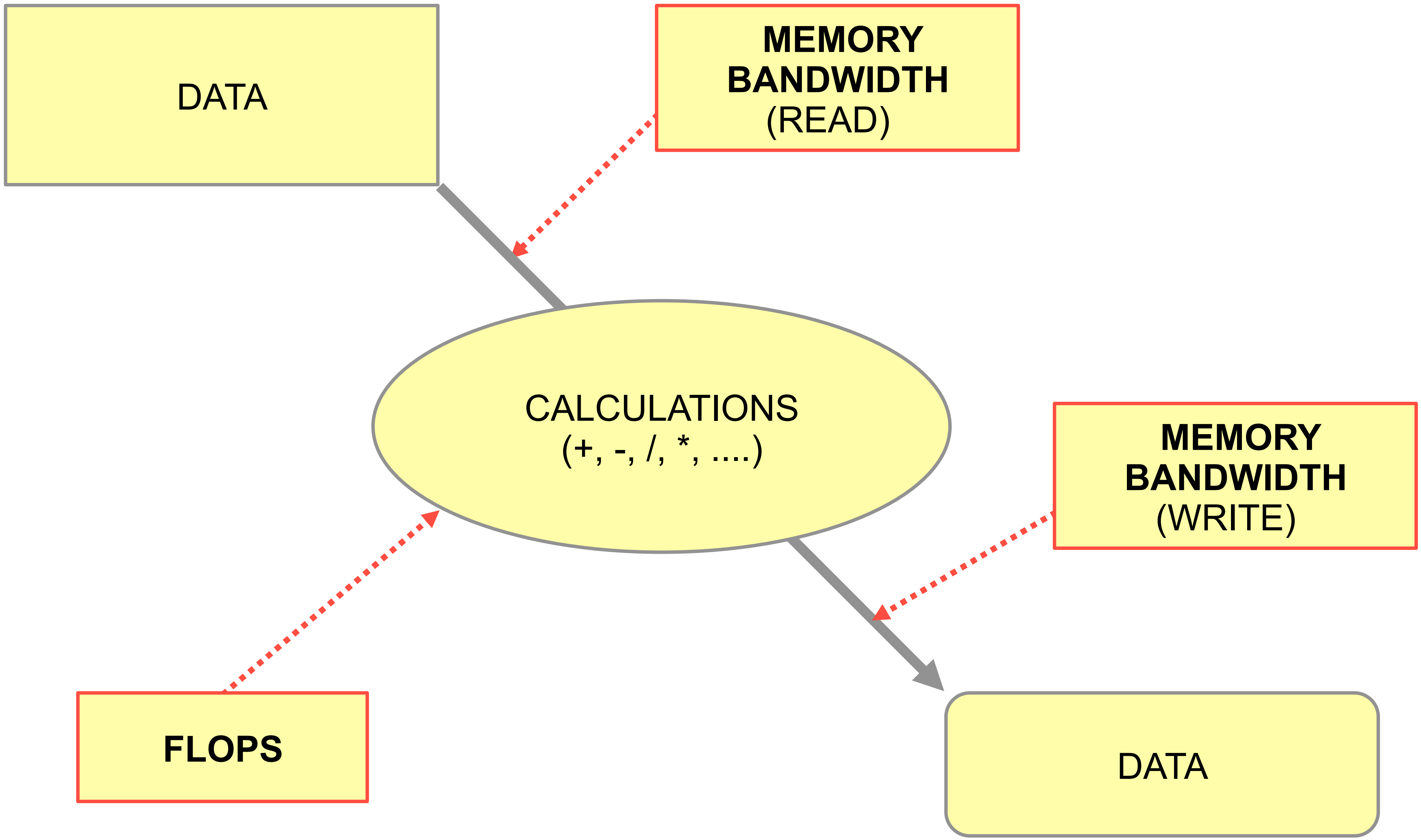
Kernels can be Compute bounded (DGEMM) or Communication bounded (DAXPY)
(kernels are rarely well balanced)

Some hardware is more communication oriented than another (high memory BW)

Some hardware is more computation oriented than another (high FLOPs)

Mapping kernel characteristics to hardware characteristics (or vice-versa) $\longrightarrow$ performance

Performance Limiting Factors

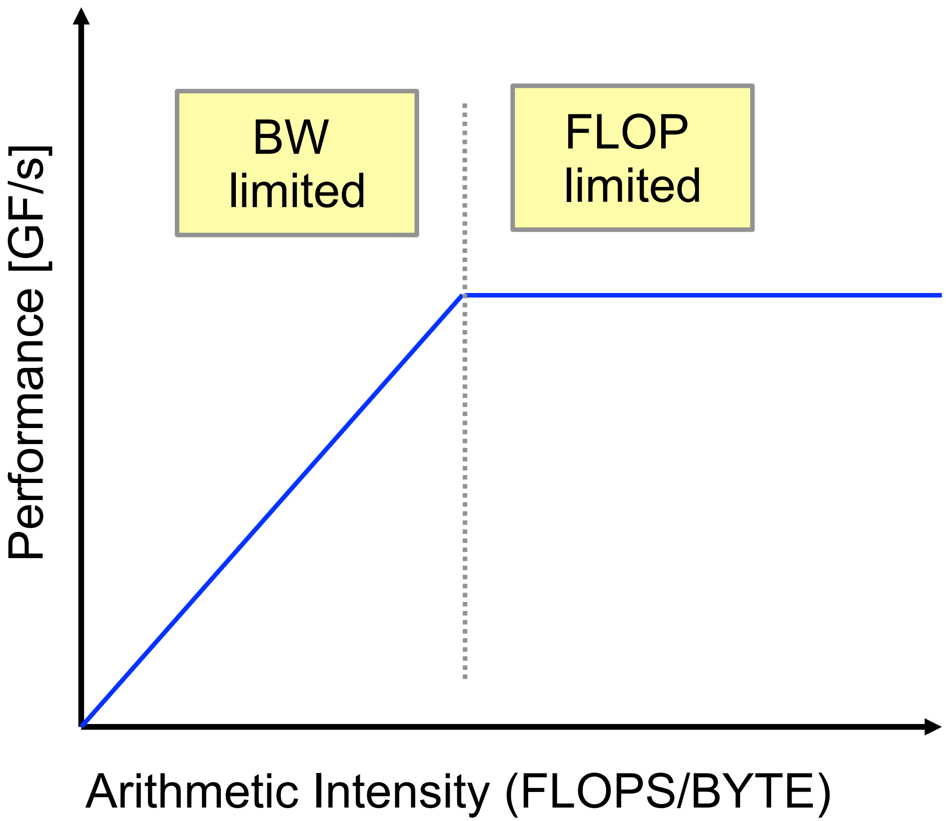


The Roofline Model - is a tool to understand the kernel/hardware limitation and it is also a tool for kernel optimization

Performance is upper **bounded** by:

1) the **peak flop** rate

2) the streaming **bandwidth**

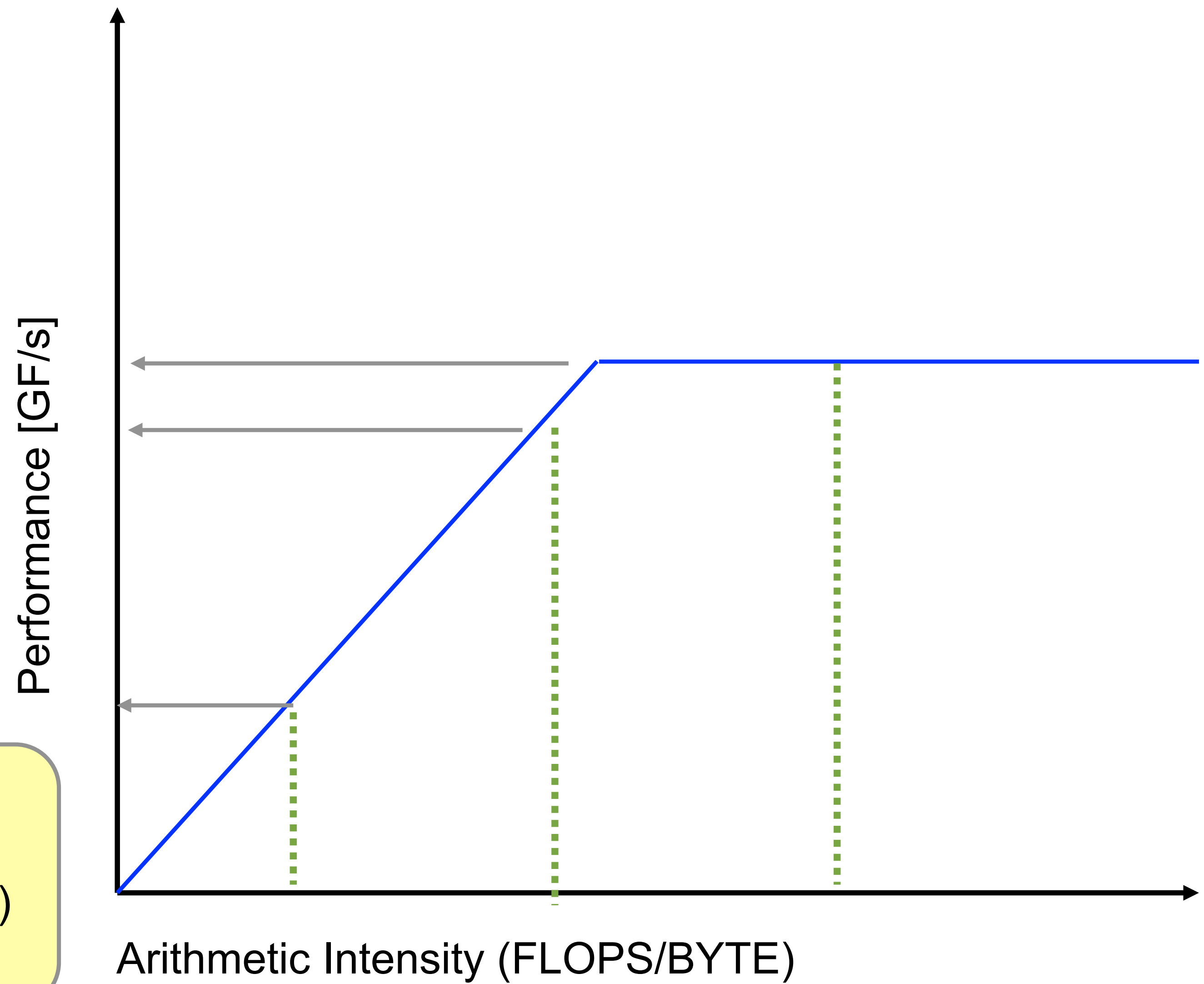


The Roofline Model

```
for (i=0; i < N; i=i+1)
    a[i] = 2.3*b[i]
```

```
for (i=0; i < N; i=i+1)
    a[i] = b[i]*b[i]+b[i]
```

```
for (i=0; i < N; i=i+1)
    a[i] = b[i]*b[i]+sin(b[i])+exp(b[i])
```



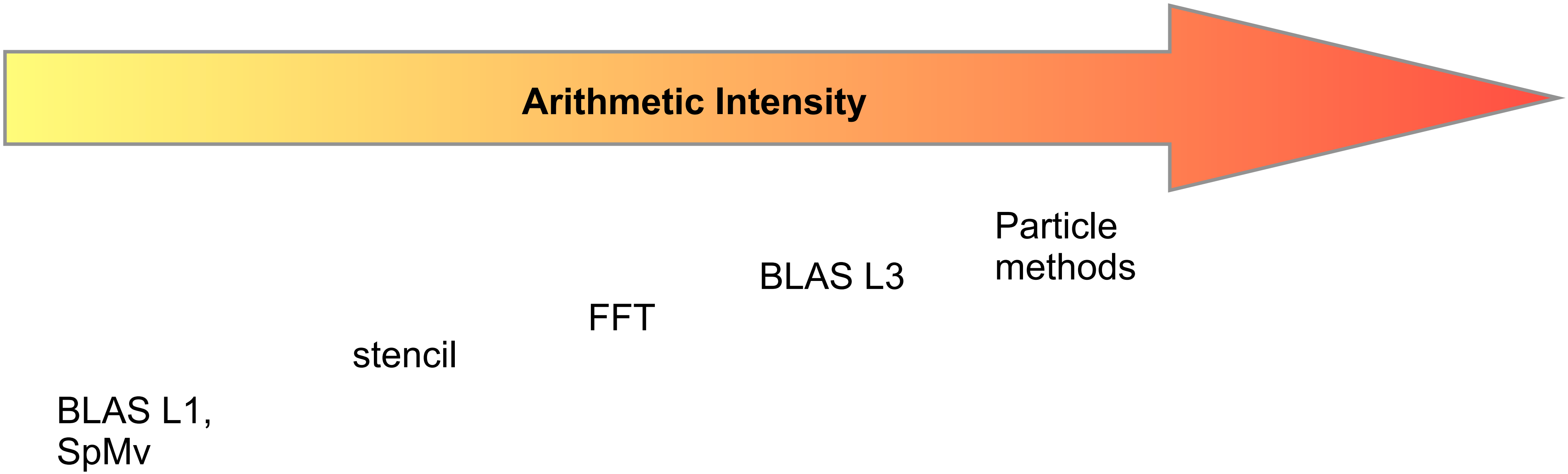
The Roofline Model: Arithmetic Intencity (AI)

FLOPS / Bytes ratio – one of the basic characteristics of a kernel

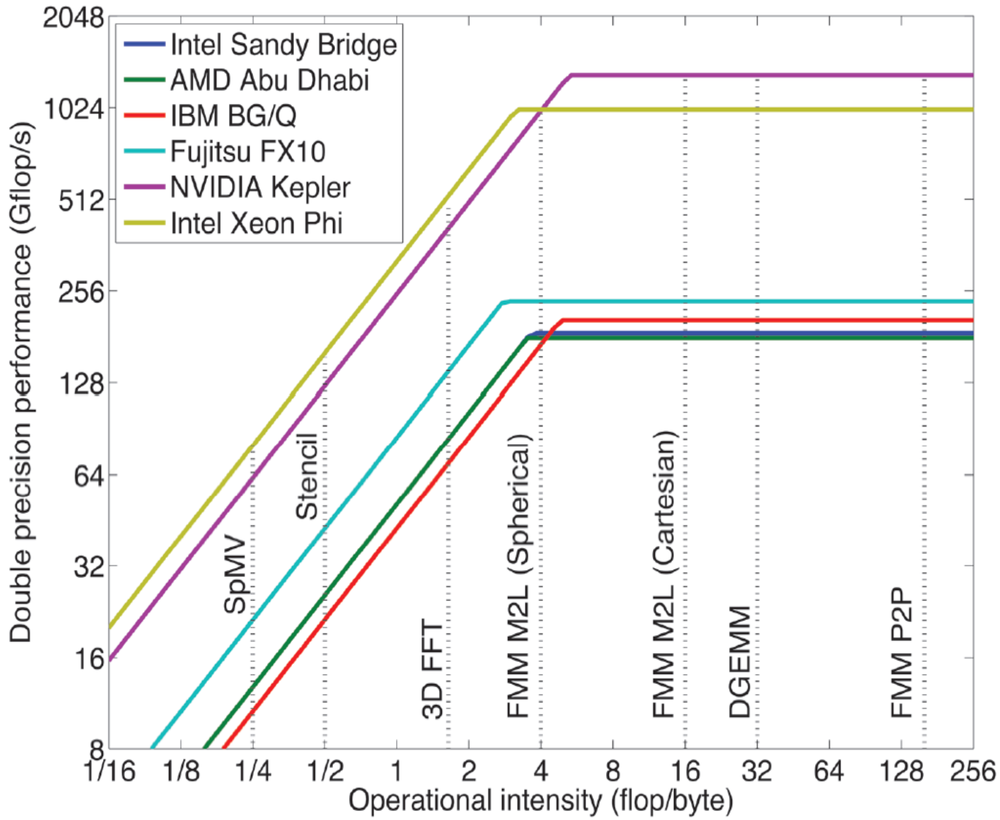
for (i = 0; i < N; ++i) z[i] = x[i]+y[i]	1 ADD 2 (8 byte) loads 1 (8 byte) write AI = 1 / (2*8 + 8) = 1/24*
for (i = 0; i < N; ++i) z[i] = x[i]+y[i]*x[i]	1 ADD 1 MUL 2 (8 byte) loads 1 (8 byte) write AI = 2 / (2*8 + 8) = 1/12*
for (i = 0; i < N; ++i){ I1 = A_offset[i]; I2 = A_offset[i+1]; sum = 0.0 for (j = 0; j < (I2-I1); ++j) sum += A[I1+j] * x[col_index [I2+j]]; y[i] = sum; }	1 ADD 1 MUL 2 (8 byte) + 1 (4 bytes) loads 1 (8 byte) write AI = 2 / (2*8 + 4 + 8) = 1/14

* because of write-allocate traffic on cache-based systems kernel would actually requires an extra read for Z and have even lower AI.

The Roofline Model: Arithmetic Intencity (AI)



The Roofline Model: Kernel-Hardware mapping

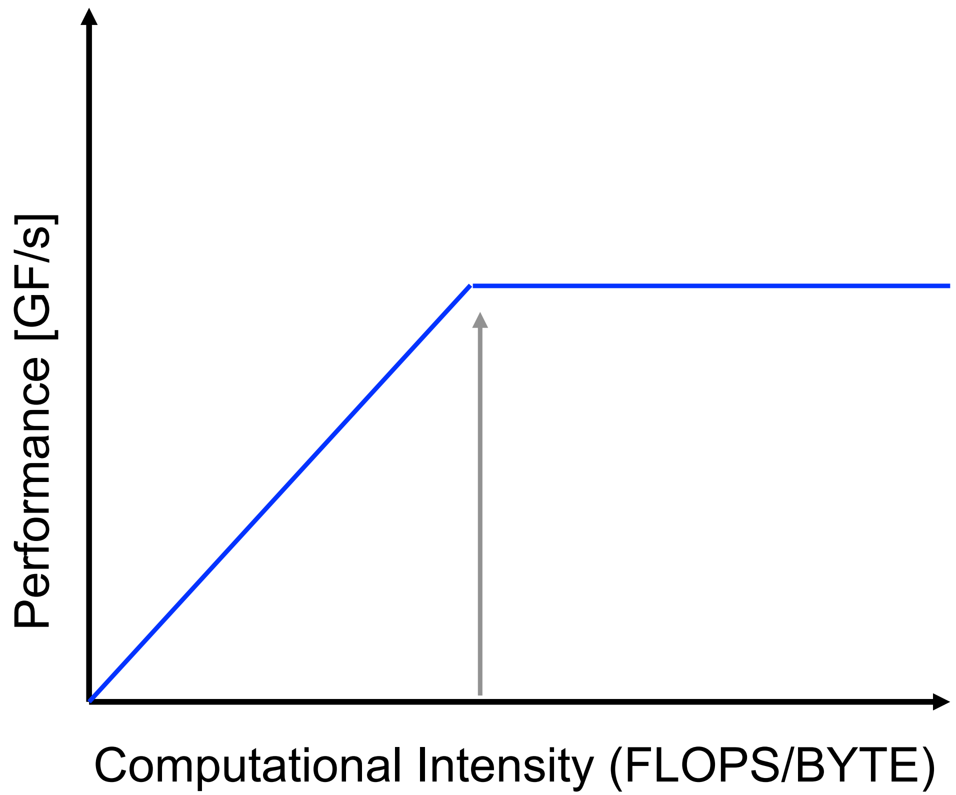


How Will the Fast Multipole Method Fare in the Exascale Era?
SIAM News, Volume 46, Number 6, July/August 2013
By Lorena A. Barba and Rio Yokota (Boston University & KAUST)

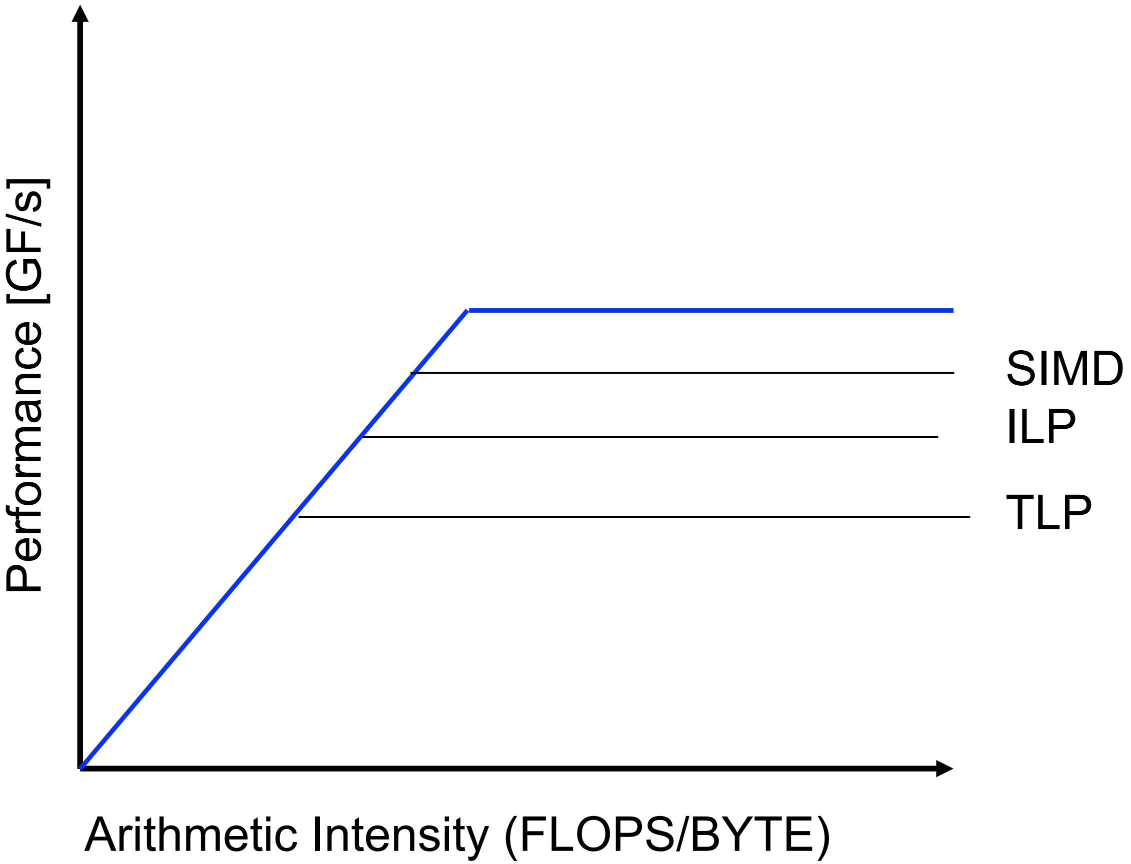
- The trend is for architectures to have ever decreasing machine balance (the point where the bandwidth roof meets the ceiling moves to the right).
- More and more algorithms are going to find themselves memory bound.
- Even DGEMM can run into trouble depending on the blocking factor chosen.
- A “balanced” architecture can also be a “crippled” one, e.g. low-end GPUs with 1/24th the DP peak performance.
  - You can achieve a higher percentage of a lower peak.

It is an art to find a perfect match between kernel and hardware characteristics

In another words it requires a lot of work to create a kernel that will exhaust both, the memory BW and FLOPs capacity *at the same time*. (many times it is even impossible ….)



The Roofline Model: Performance Limiting Factors

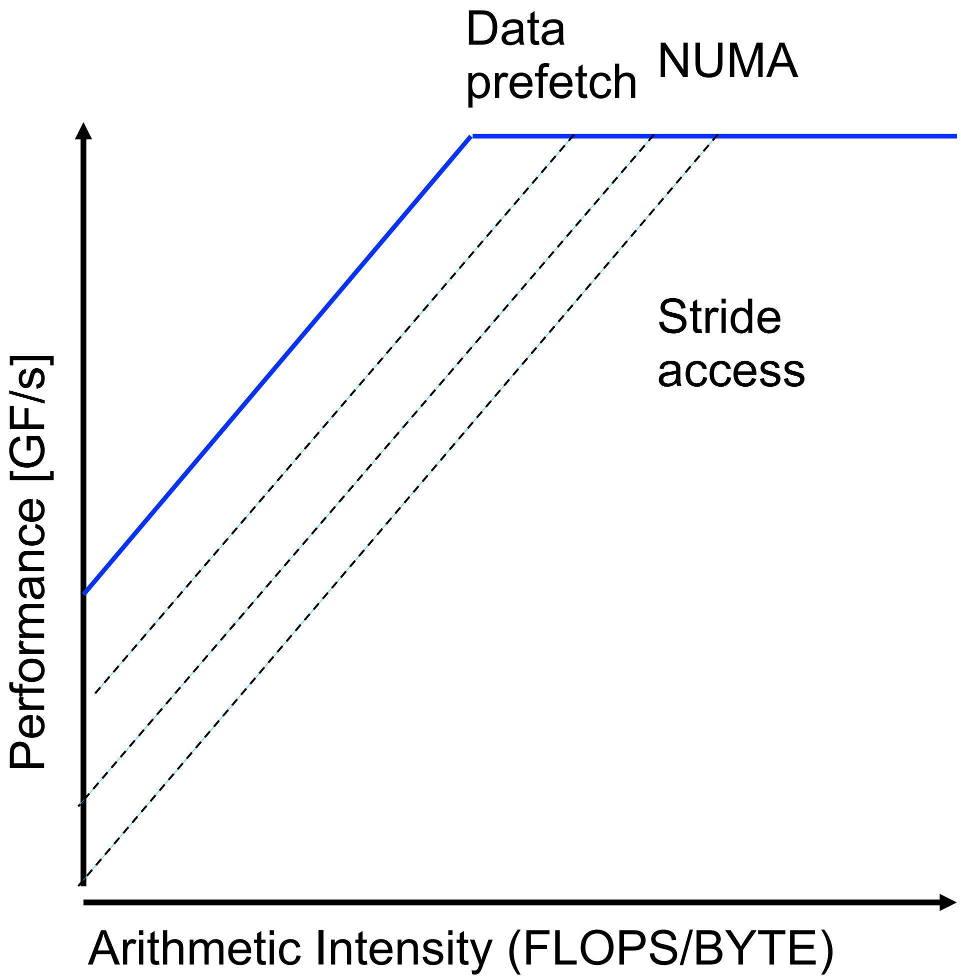


Performance depends on how well a given kernel fits node/processor architecture,

and/or how well a given kernel is translated by a compiler.

Recall: hardware-kernel characteristics mapping.

The Roofline Model: Performance Limiting Factors

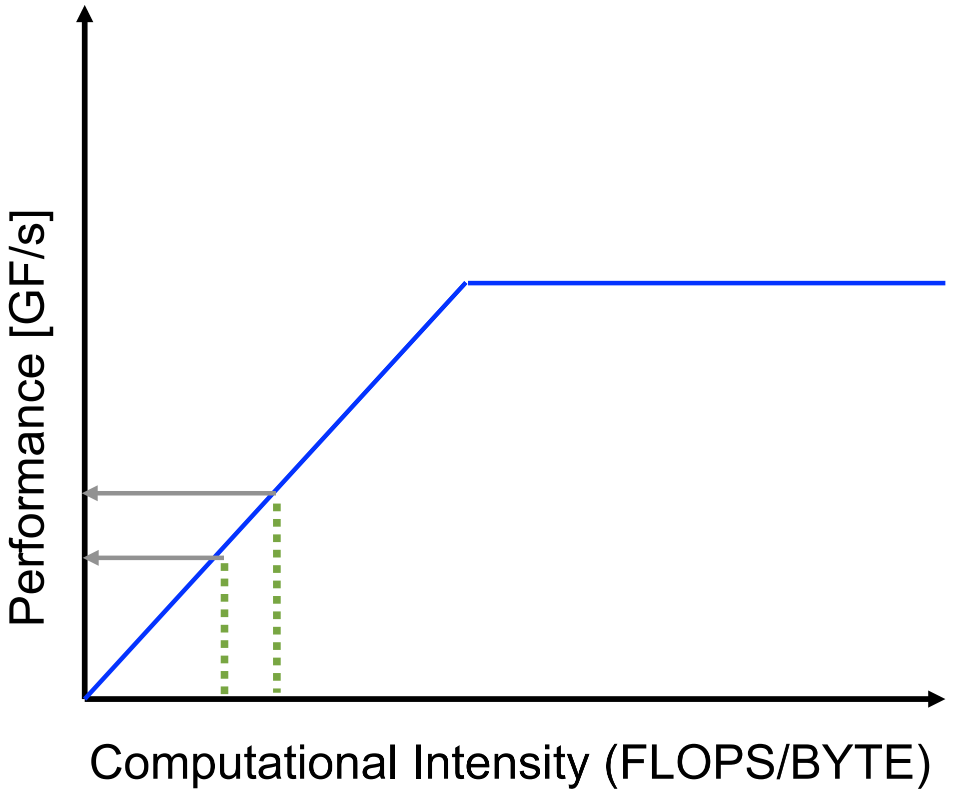


Performance depends on how well a given kernel fits node/processor architecture,

and/or how well a given kernel is translated by a compiler.

Recall: hardware-kernel characteristics mapping.

The Roofline Model: Performance limiting factors



N – is large, i.e., buffer does not fit cache

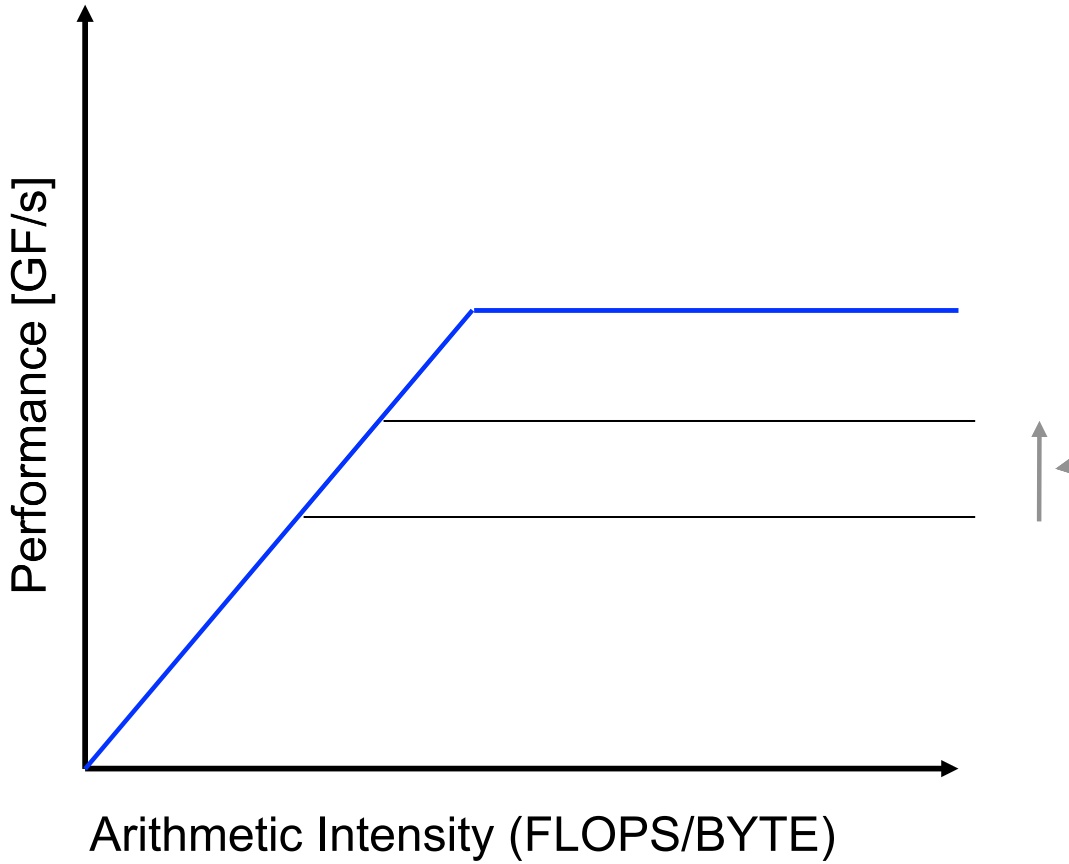
```
for (i=0; i < N; ++i)
   a[i] = buffer[i] + b[i];


for (i=0; i < N; ++i)
   c[i] = buffer[i] + d[i];
```

AI_total = 2 / (2 * 3 * 8) = 1/24;

```
for (i=0; i < N; ++i){
   a[i] = buffer[i] + b[i];
   c[i] = buffer[i] + d[i];
}
```

AI = 2/(5*8) = 1 / 20;

The Roofline Model: Performance Limiting Factors - Instruction Level Parallelism (ILP)




```
sum = 0;
for (i=0; i < N; ++i)
   sum = sum + a[i];
```

```
sum0 = sum1 = sum2 = sum3 = 0;
for (i=0; i < N; i+=4){
   sum0 = sum0 + a[i    ];
   sum1 = sum1 + a[i+1];
   sum2 = sum2 + a[i+2];
   sum3 = sum3 + a[i+3];
}
sum0 = sum0+sum1;
sum2 = sum2+sum3;
sum = sum0+sum2;
```

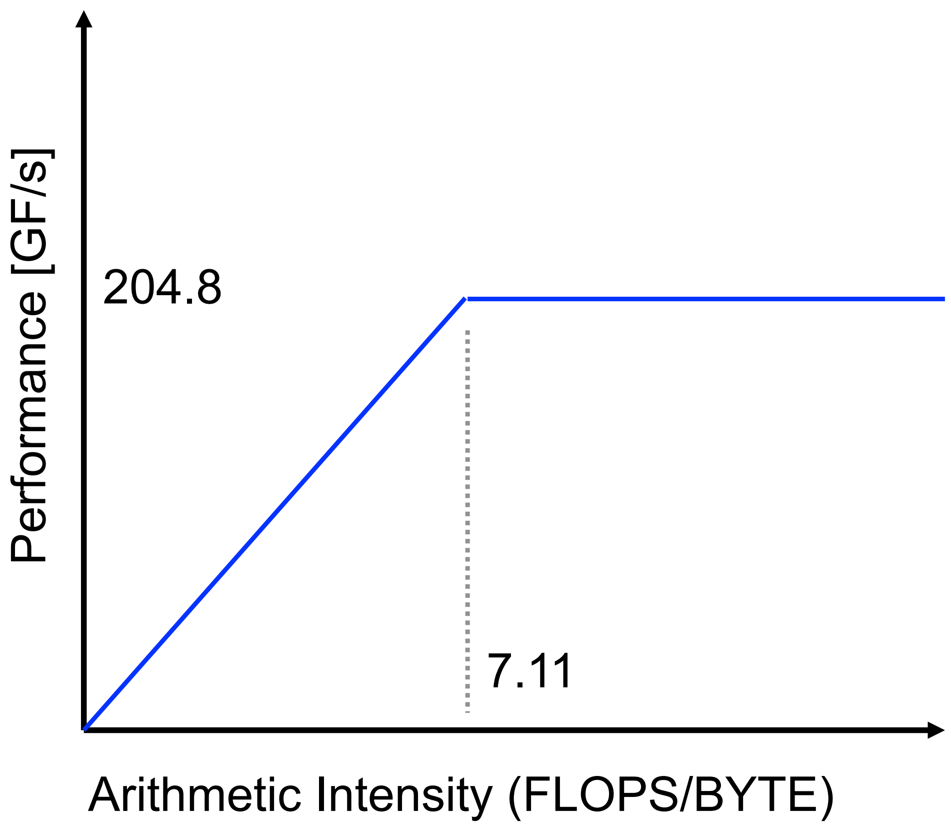
EXAMPLES and EXERCISES

Example 1: DAXPY

Consider DAXPY : for (i = 0; i < N; ++i) y[i] = a*x[i]+y[i]

For each “i” : 1 addition , 1 multiplication
2 loads of 8 bytes each
1 store

Execution on BlueGene/Q (Peak 204.8 GFLOP/node)



Performance estimates:

AI = 2/(3*8) = 1 / 12

1/12 < 7.11 →
We are in the memory BW limited area on the Roofline plot
7.11 / (1 / 12) = 85.32
204.8 / 85.32 = **2.4 GF/s**

Example 1: DAXPY

Consider DAXPY : for (i = 0; i < N; ++i) y[i] = a*x[i]+y[i]

For each "i" : 1 addition , 1 multiplication
2 loads of 8 bytes each
1 store

Execution on BlueGene/Q (Peak 204.8 GFLOP/node):

# threads	Time [s]	GFLOPS	DDR traffic per node (Bytes/cycle)
1	0.0879111	0.455	3.519
2	0.044039	0.907	7.022
4	0.022151	1.801	13.94
8	0.0174019	**2.284**	17.686
16	0.017447	**2.287**	17.719

Performance estimates:

AI = 2/(3*8) = 1 / 12

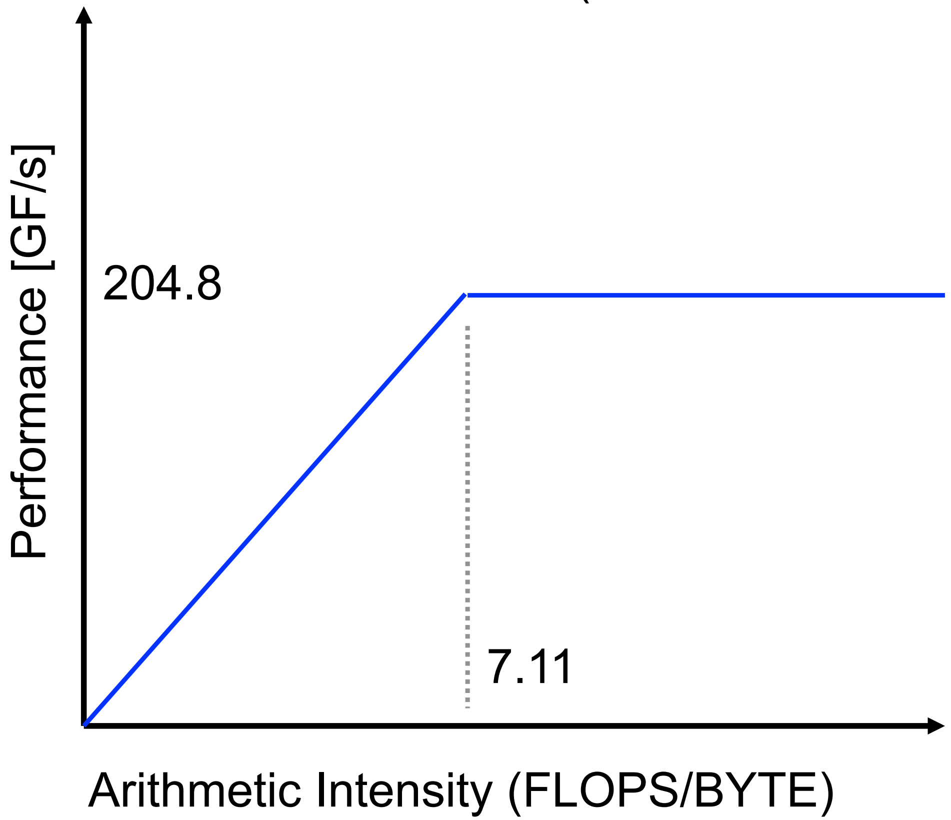
1/12 < 7 →
We are in the memory BW limited area on the roofline plot
7.11 / (1 / 12) = 85.32
204.8 / 85.32 = **2.4 GF/s**

Example 2

Consider DAXPY : for (i = 0; i < N; ++i) y[i] = a*x[i]+y[i] **+ x[i]*x[i]**

For each "i" : 2 addition , 2 multiplication
2 loads of 8 bytes each
1 store

Execution on BlueGene/Q (Peak 204.8 GFLOP/node):



Performance estimates:

AI = 4/(3*8) = 1 / 6

1/6 < 7 →
We are in the memory BW limited area on the roofline plot
7.11 / (1 / 6) = 42.66
204.8 / 42.66 = **4.8 GF/s**

Example 2

Consider : for (i = 0; i < N; ++i) y[i] = a*x[i]+y[i] +**x[i]*x[i]**

For each "i" : 2 addition , 2 multiplication
2 loads of 8 bytes each
1 store

Execution on BlueGene/Q (Peak 204.8 GFLOP/node):

# threads	Time [s]	GFLOPS	DDR traffic per node
1	0.106501	0.751	2.906
2	0.053323	1.499	5.802
4	0.0267339	2.989	11.566
8	0.0176179	**4.532**	17.545
16	0.0174541	**4.573**	17.712

Performance estimates:

AI = 4/(3*8) = 1 / 6

1/6 < 7 →
We are in the memory BW limited area on the roofline plot
7.11 / (1 / 6) = 42.66
204.8 / 42.66 = **4.8 GF/s**

Example 3

Consider for (i = 0; i < N; ++i) y[i] = a*x[i]+y[i] + **x[i]*x[i] + SIN(x[i])**

Execution on BlueGene/Q (Peak 204.8 GFLOP/node):

# threads	Time [s]	GFLOPS	DDR traffic per node
1	0.615393	1.755	0.503
2	0.307695	3.51	1.006
4	0.153861	7.018	2.244
8	0.076983	14.023	4.02
16	0.0385199	28.008	8.034
32	0.0217798	49.461	14.202
64	0.018496	**58.137**	16.73

Examples 1 and 3

y[i] = a*x[i]+y[i]

Loads that hit in L1 d-cache = 50.01 %
L1P buffer = 49.98 %
L2 cache = 0.00 %
DDR = 0.01 %

We spend too much time moving data:
2.284 GF/s

y[i] = a*x[i]+y[i] **+ x[i]*x[i] + SIN(x[i])**

Loads that hit in L1 d-cache = 97.30 %
L1P buffer = 2.70 %
L2 cache = 0.00 %
DDR = 0.00 %

We spend less time moving data than computing
58.137 GF/s

Examples 1 and 3

y[i] = a*x[i]+y[i]

Loads that hit in L1 d-cache = 50.01 %
L1P buffer = 49.98 %
L2 cache = 0.00 %
DDR = 0.01 %

We spend too much time moving data:
2.284 GF/s
solve time: 17.5 ms

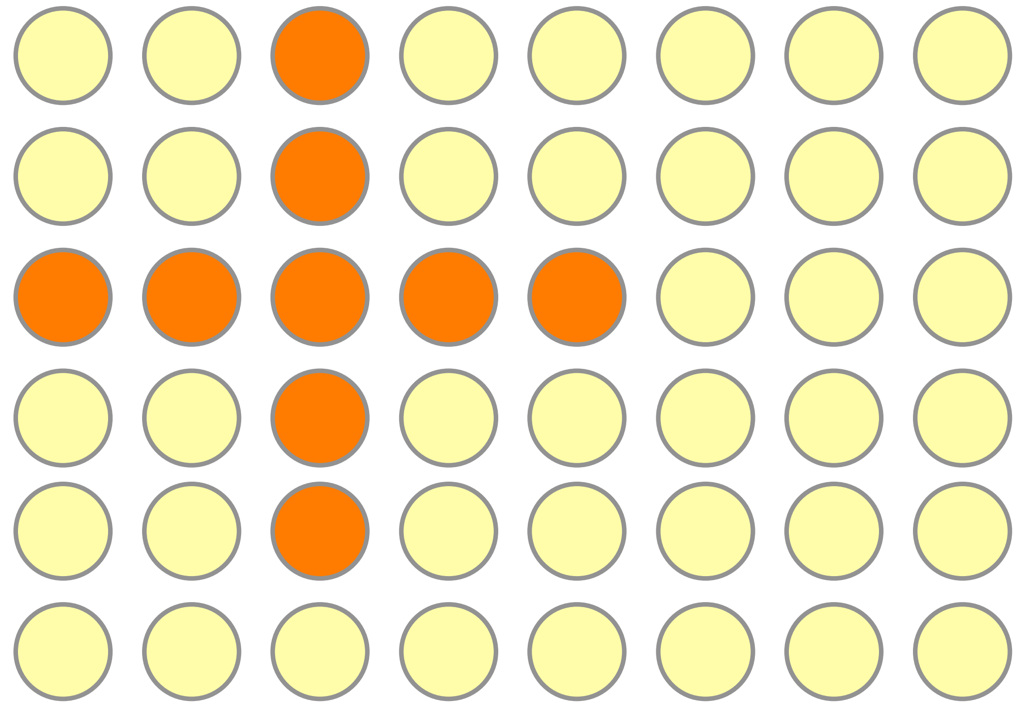
y[i] = a*x[i]+y[i] **+ x[i]*x[i] + SIN(x[i])**

Loads that hit in L1 d-cache = 97.30 %
L1P buffer = 2.70 %
L2 cache = 0.00 %
DDR = 0.00 %

We spend less time moving data than computing
58.137 GF/s
solve time: 18.5 ms

Example: 2D stencil

Consider two arrays A, and B, both have dimension of NxN



B is computed from:

$$
\begin{aligned}
B[i][j] = {} & A[i-2][j] + A[i-1][j] + C*A[i][j] + A[i+1][j] + A[i+2][j] + \\
& A[i][j-2] + A[i][j-1] + A[i][j+1] + A[i][j+2]
\end{aligned}
$$

Arithmetic intensity: 7 adds, 1 mul, 1 load and 1 store →
AI = 8 / (2*8) = 1 / 2
Estimated performance on BG/Q: 7.11 / (½) = 14.22;
204.8 / 14.22 = **14.4 GF/s**

```
#pragma omp parallel for private(row,col)

for (row  = 2; row < (N-2); ++row){
   for (col = 2; col < (N-2); ++col) {
        B[row][col] =  C*A[row][col] +
                    A[row][col-1] + A[row][col+1] +
                    A[row][col-2] + A[row][col+2] +
                    A[row-1][col] + A[row+1][col] +
                    A[row-2][col] + A[row+2][col] ;
   }
}
```

We run on a single BGQ node
1 mpi rank, 64 threads

HPM info:
Total weighted **GFlops = 4.922**
Loads that hit in L1 d-cache = 93.05 %
L1P buffer = 5.08 %
L2 cache = 0.00 %
DDR = 1.86 %
Average DDR traffic per node: **ld = 13.680, st = 2.757**, total = 16.437 (Bytes/cycle)

We estimated 14.4GF/s

What have we done wrong?

2D Stencil: Algorithm No. 2

```
#pragma omp parallel for private(rb,cb,row,col)

for (rb = 2; rb < N; rb = rb + row_block_size){   //ROW BLOCKING
 for (cb = 2; cb < N; cb = cb + col_block_size){ // COLUMN BLOCKING

   for (row  = rb; row < MIN(N-2,rb + row_block_size+1); ++row){
    for (col  = cb; col < MIN(N-2,cb + col_block_size+1); ++col){
      B_rcb[row][col] =  C*A[row][col] +
               A[row][col-1] + A[row][col+1] +
               A[row][col-2] + A[row][col+2] +
               A[row-1][col] + A[row+1][col] +
               A[row-2][col] + A[row+2][col] ;

    }
   }
 }
}
```

HPM info:
Total weighted **GFlops = 12.264**
Loads that hit in L1 d-cache = 97.69 %
L1P buffer = 1.26 %
L2 cache = 0.34 %
DDR = 0.70 %
Average DDR traffic per node: **ld = 7.599, st = 6.746**, total = 14.346 (Bytes/cycle)

We estimated 14.4GF/s
We got 12.264GF/s …

Exercise No 1.

- Compile and execute daxpy
- Use 1 to 16 threads to run the program
- Analyze performance.
- Find empirically the crossover point.
  Calculate the location (x-coordinate) of the crossover point based on hardware
  (2-socket Intel(R) Xeon(R) CPU E5-2670 @2.6GHz node) and kernel characteristics

Exercise No 2.

- Compile and execute 2D stencil code
- Use 1 to 16 threads to run the program
- Estimate performance for 2-socket Intel(R) Xeon(R) CPU E5-2670 @2.6GHz
- Compare to the achieved performance

Questions ?

How to compile

1. ssh

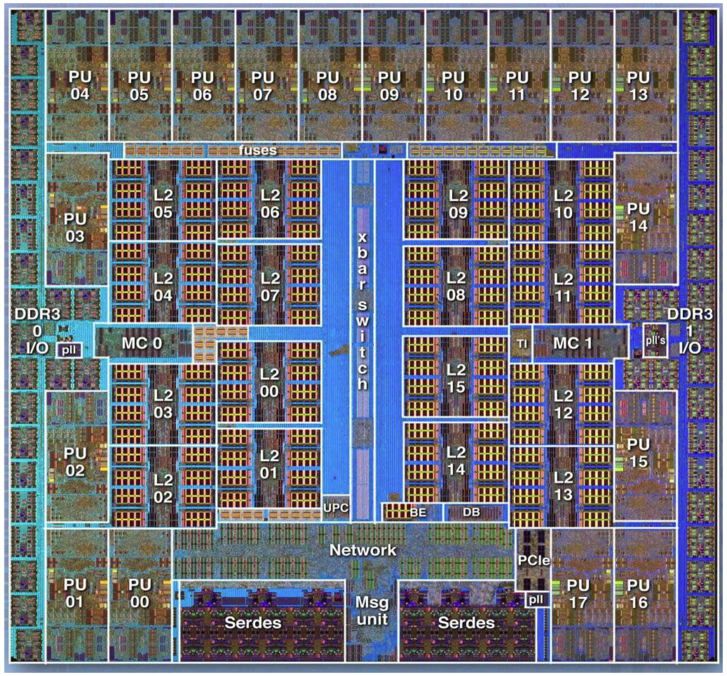
2. Type

   MODULEPATH=/lustre/utility/modulefiles:$MODULEPATH

3. Load module
   module load icc/13.1.1

Now we can use compiler icc or icpc

The Roofline Model: Principal Components to Performance



18 cores BGQ chip