

CS528
High Performance Computing

Data Access Optimization and Roofline Model

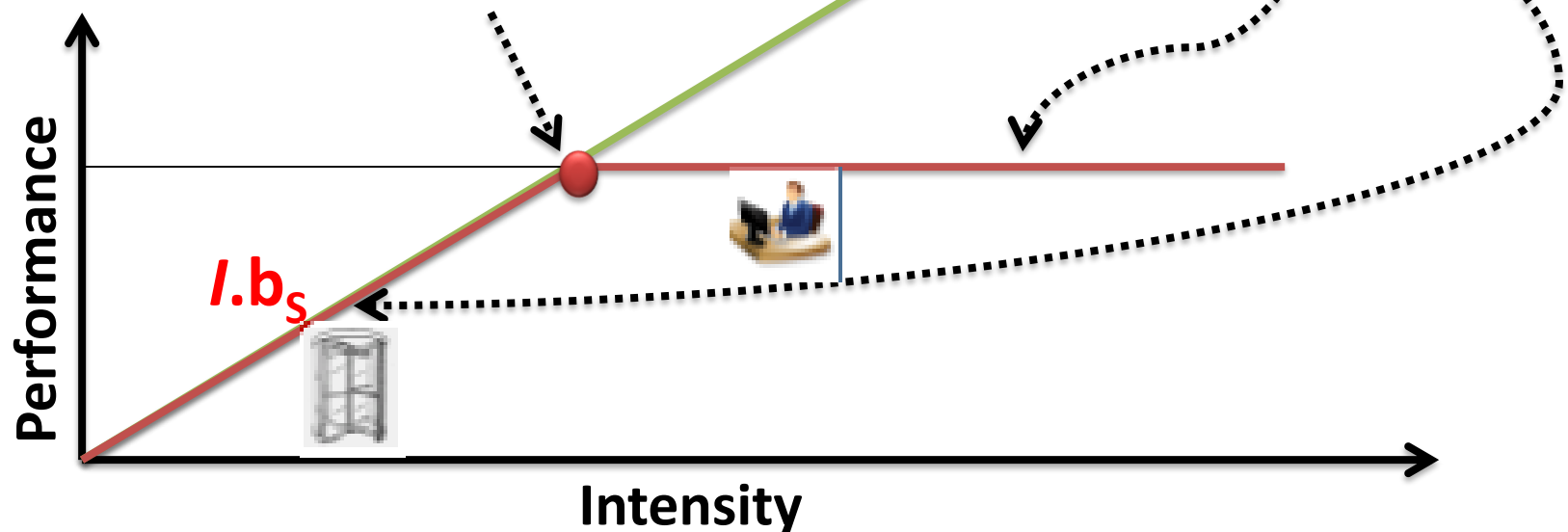
A Sahu
Dept of CSE, IIT Guwahati

Outline

- Data Access Optimization
 - Roofline Model
 - Caching optimization
 - App classification based DA: N/N , N^2/N^2 , N^3/N^2
- *[Ref: Hager Book, PDF uploaded to MS Team]*

Modeling Customer Dispatch in a Bank

- Performance $P = \min(P_{\text{peak}}, I \cdot b_s)$
- This is the “Roofline Model”
 - High intensity: P limited by “execution”
 - Low intensity: P limited by “bottleneck”
 - “Knee” at $P_{\text{peak}} = I \cdot b_s$: Best use of resources



- Roofline is an “optimistic” model

The Roofline Model

- P_{\max} = Peak performance of the machine
- I = Computational intensity (“work” per byte transferred) over the slowest data path utilized (“the bottleneck”)
- b_s = Applicable peak bandwidth of the slowest data path utilized

Expected performance:

$$P = \min(P_{\text{peak}}, I \cdot b_s)$$

[F/B]

[B/s]

Memory Analysis of Simple Triad Code on Intel i7-5960X

**i7-5960x Spec (8C-16T, 22nm, 3.5Ghz, 20MB
Smart Cache)**

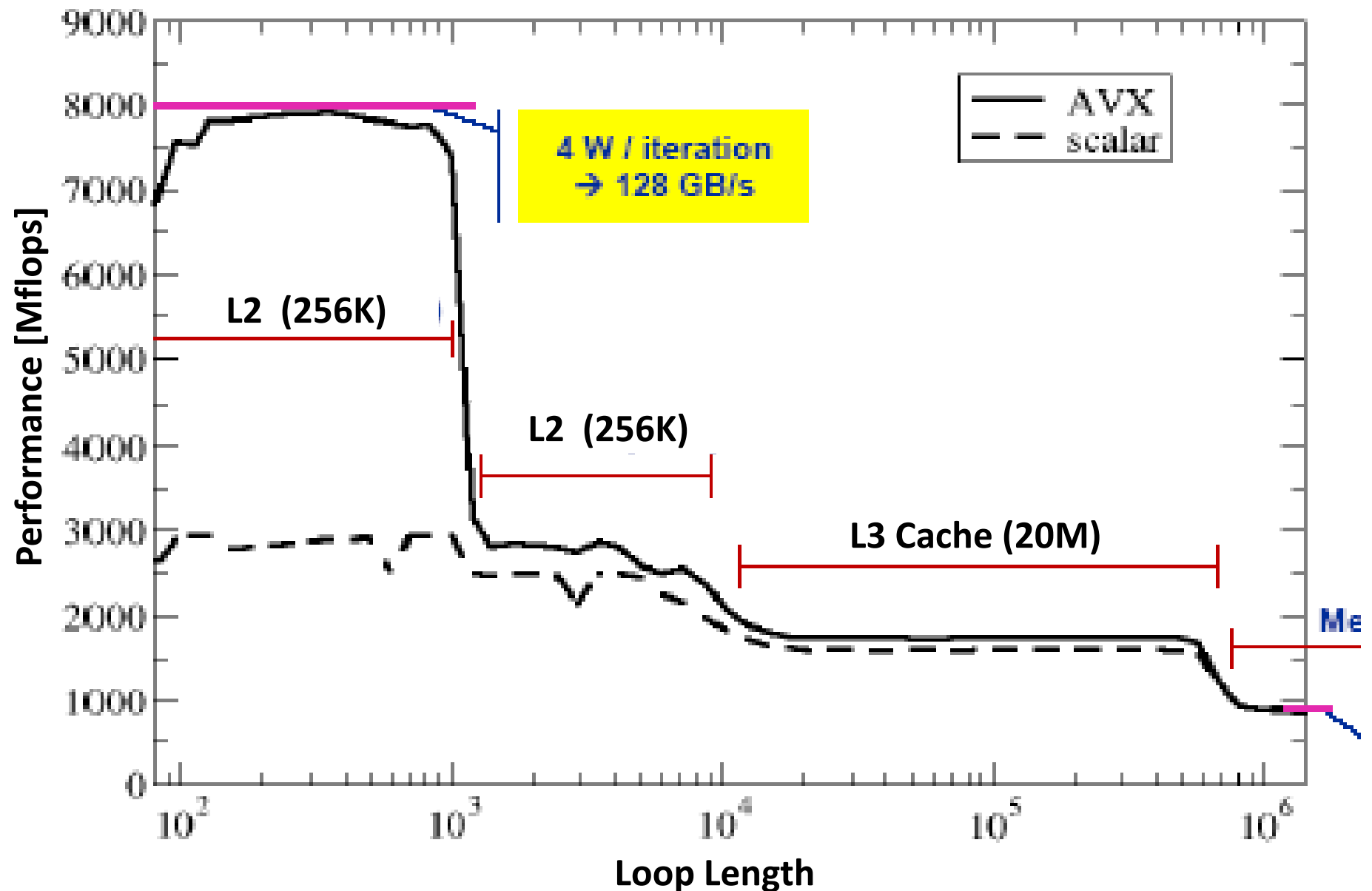
Memory Analysis of Simple Code

- Simple Streaming Benchmark
- *A “swiss army knife” for micro-benchmarking*
 - Report performance for different N
 - Choose NITER : Accurate time measurement is possible
- **This kernel is limited by data transfer performance for all memory levels on all current architectures!**

```
float    A[N], B[N], C[N], D[N];  
for( j=0; j<NITER; j++) {  
    for(i=0; i<N; i++)  
        A[i] = B[i] + C[i] * D[i];  
    if(i>N)    dummy(A, B, C, D);  
}
```

Prevents smarty-pants
compilers from doing
“clever” stuff

On Sandy Bridge: Core i7 5960 Xtreme

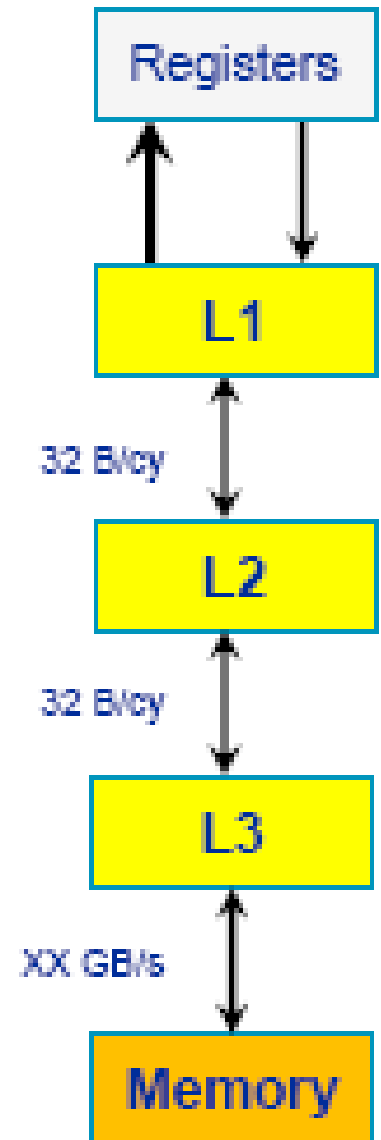


Memory Hierarchy

- Are the performance levels plausible?
- What about multiple cores?
- Do the bandwidths scale?

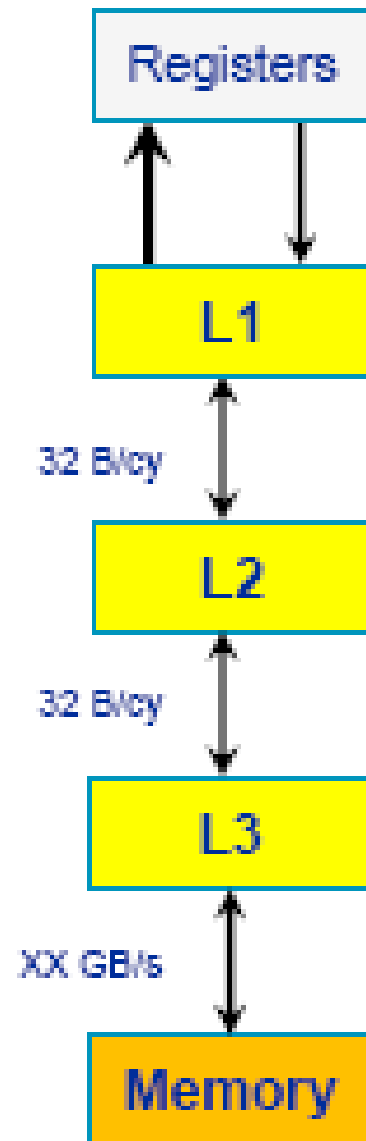
Throughput capabilities: i7 5960 Xtreme

- **Per cycle with AVX**
 - 1 load instruction (256 bits) AND ½ store instruction (128 bits)
 - 1 AVX MULT and 1 AVX ADD instruction (4 DP / 8 SP flops each)
 - Overall maximum of 4 micro-ops

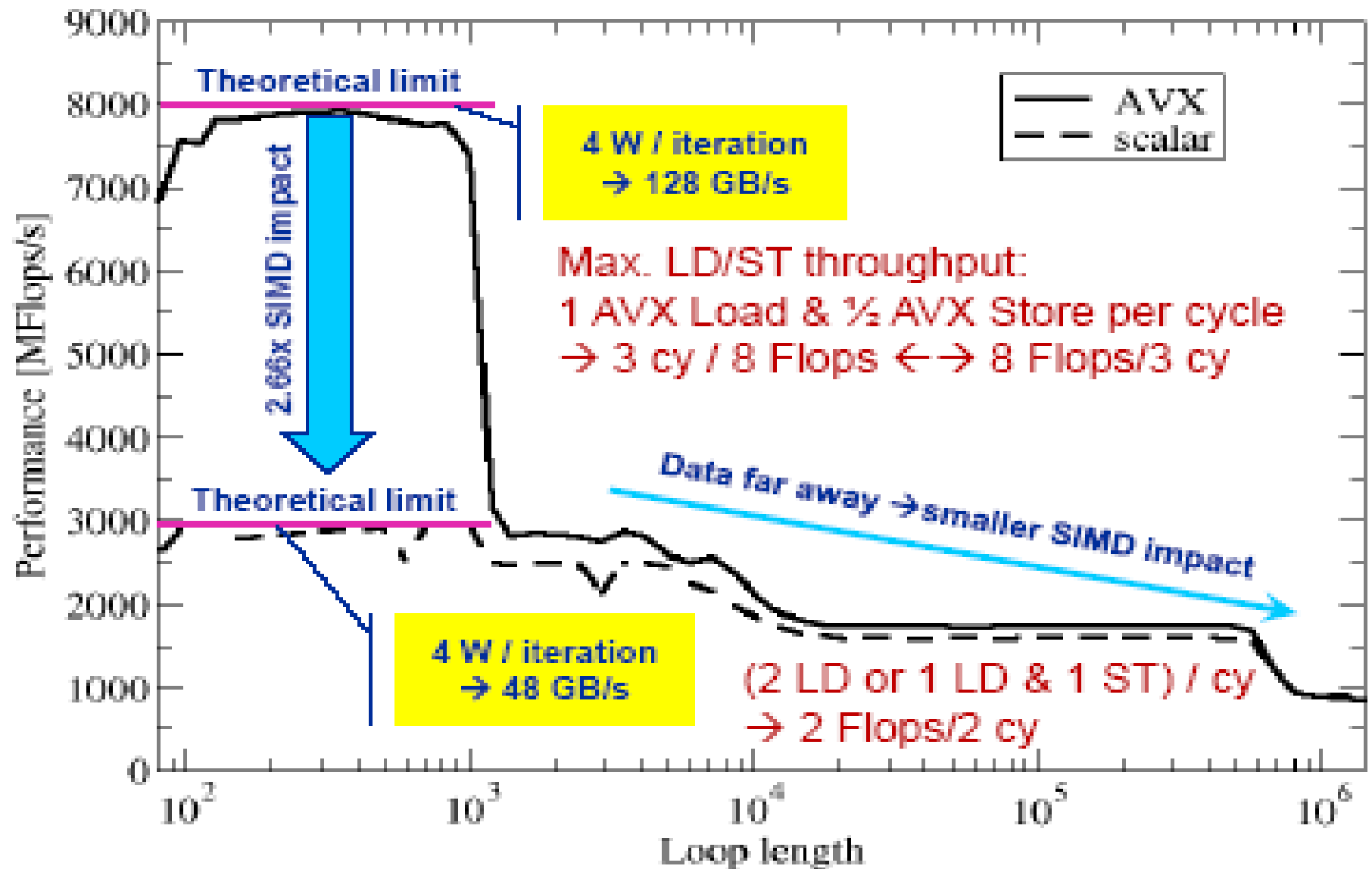


Throughput capabilities: i7 5960 Xtreme

- Per cycle with SSE or scalar
 - 2 load instruction OR 1 load and 1 store instruction
 - 1 MULT and 1 ADD instruction
 - Overall maximum of 4 micro-ops
- Data transfer between cache levels
 - (L3 \leftrightarrow L2, L2 \leftrightarrow L1)
 - 256 bits per cycle, half-duplex (i.e., full CL transfer == 2 cy)



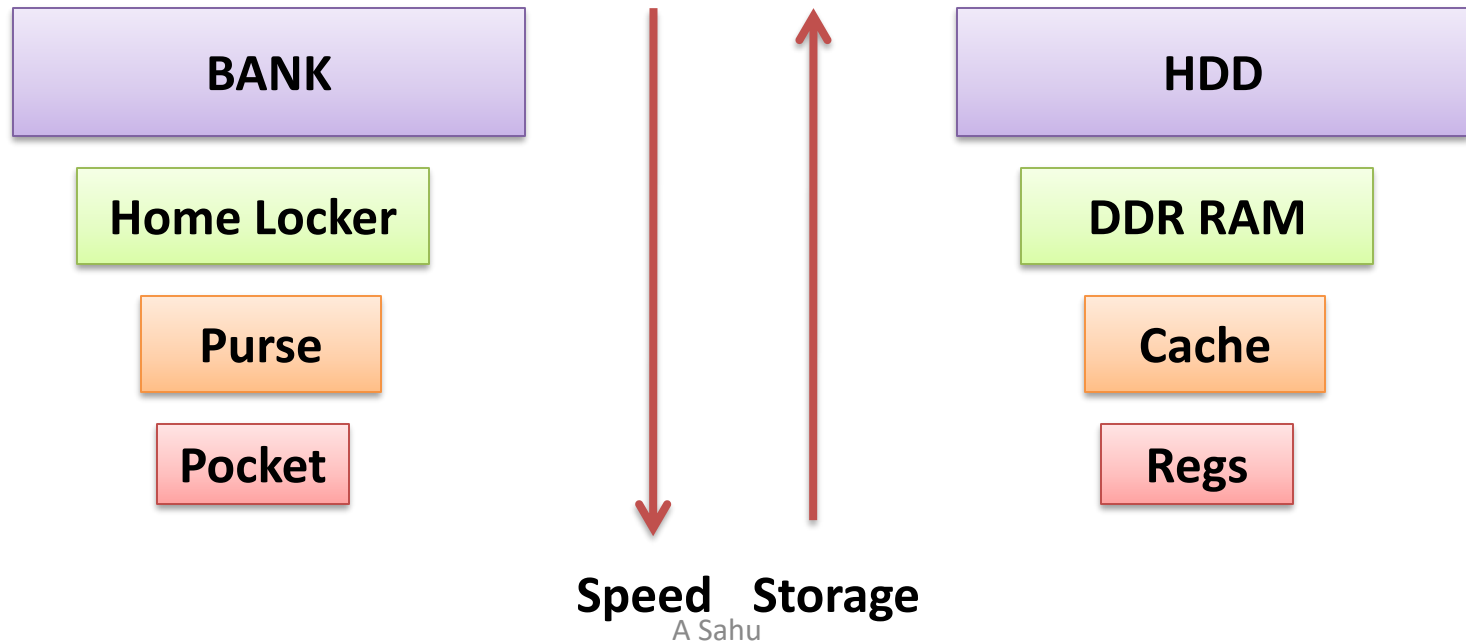
On Sandy Bridge: Core i7 5960 Xtreme



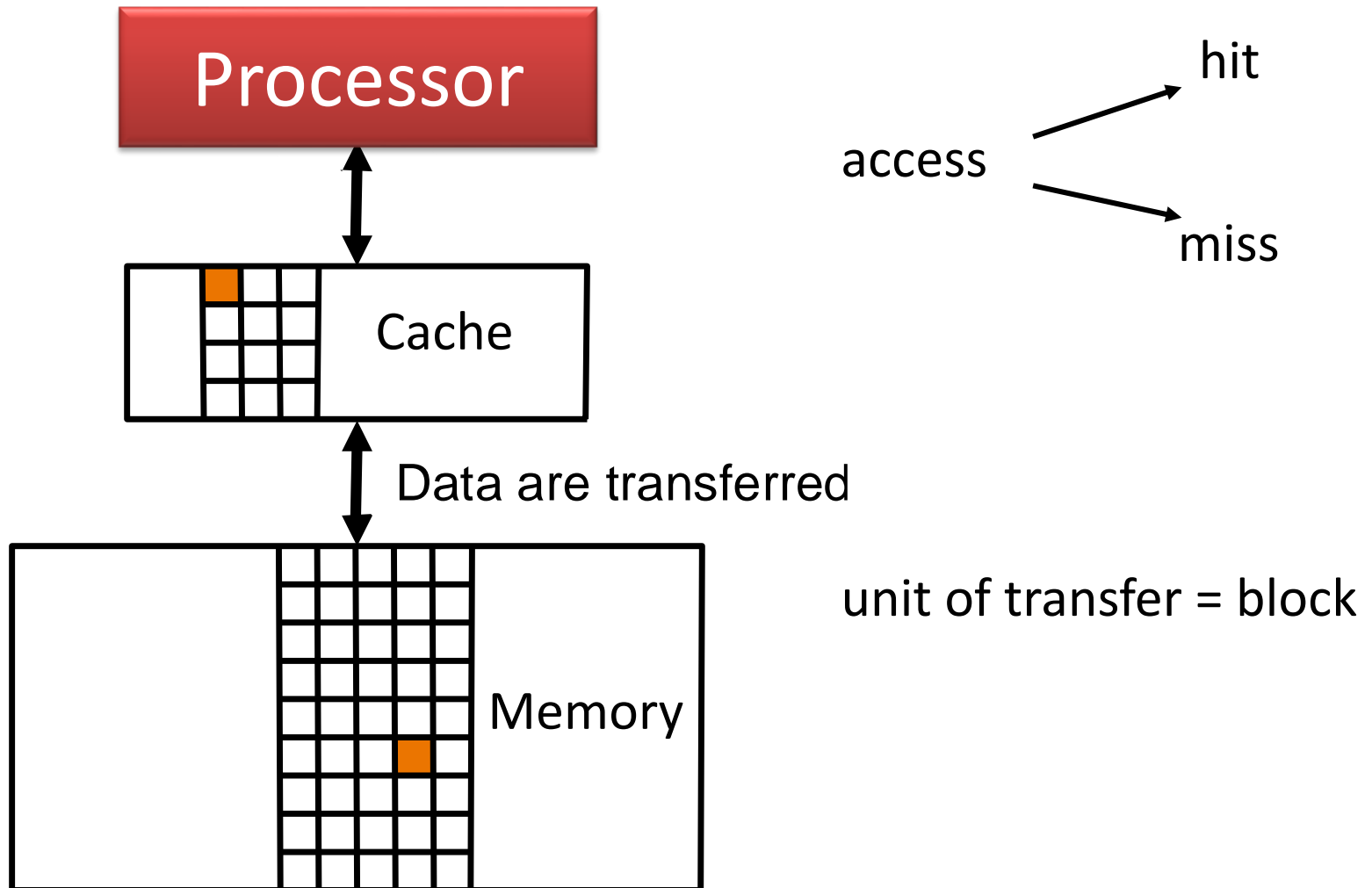
**Reducing Code Balance
(Byte/Flop) using Memory
Hierarchy : Caching**

Memory Hierarchy

- Smaller is Faster - Bigger is Slower
- Places of Cash/Money



Data transfer between levels



Principle of locality

- Temporal Locality
 - references repeated in time
- Spatial Locality
 - references repeated in space
 - Special case: Sequential Locality

```
for(i=0;i<100;i++){  
    A[i] += sqrt(i);  
} // 1D SPLocality  
Access A[i], near future  
will Access A[i+1], A[i+2]..
```

```
for(T=0;T<80;T++){  
    for(i=0;i<10;i++){  
        A[i] +=M[T]*i;  
    }  
A[i] repeated after some Time
```

Cache Access

- Address is divided in to three part : TAG, Index, Offset
 - $\text{Offset} = \text{Address} \% \text{Line Size},$
 - $\text{Index} = (\text{Address}/\text{LineSize})\% \text{NumSet}$
 - $\text{TAG} = \text{Address}/(\text{LineSize} * \text{NumSet})$
- If TAG matches with ExistingTAG then HIT else miss

```
if (TAG==CACHE[Index].TAG)
    Cache HIT
else
    Cache MISS
```
- Assume LS=10, NumSet=100, Address 2067432
 - Offset = 2, Index =43, TAG=2067

Cache Size

- No of Set (Depend on index field)
- Associativity (How many Tag)
- Line size (No of Addressable units/byte in a line)

0	0						
2120	1						
2	2						
2123	3						
4	4						
5	5						
4143	6						
7	7						
8	8						
9	9						

Tag	index	Line
-----	-------	------

0	0					
4143	1					
2	2					
3	3					
4	4					
5	5					
6	6					
7	7					
8	8					
9	9					

Tag	index	Line
-----	-------	------

0	0						
2120	1						
2	2						
2123	3						
4	4						
5	5						
4143	6						
7	7						
8	8						
9	9						


Tag	index	Line
-----	-------	------

0	0						
4143	1						
2	2						
3	3						
4	4						
5	5						
6	6						
7	7						
8	8						
9	9						

Tag	index	Line
-----	-------	------

- **Cache Size = Nset X Associativity X LineSize**
= 10 x 4 x 10 = 400 Byte

Hashing Vs Caching

- 
- Simple Hashing: Direct Map Cache
 - Example: Array
 - `int A[10]`, each can store one element
 - Data stored in $\text{Addr}\%10$ location
 - Array of List
 - `Int LA[10]`, each can store a list of element
 - Data stored in List of $(\text{Addr}\%10)^{\text{th}}$ location
 - List size is limited in Set Associative Cache
 - List of Element
 - Full Associative Cache
 - All data stored in one list



Direct/Random
Access to
Element



MIXED



Serial/Associative Access
to Element



T
I
M
E

Program Cache Behavior: Hit/Miss

Cache model

- Direct mapped 8 word per line



Program

```
int A[128];  
for (i=0; i<128; i++) {  
    A[i]=i;  
}
```

- Assume &A=000000, **Behavior of only Data**
- Scalar variable {i} mapped to register
- Data have to moved from cache/memory

Cache perf. : Data Size <= Cache Size

```
int A[128];  
for (i=0; i<128; i++) {  
    A[i]=i;  
}
```

Scalar mapped to register
Vector mapped to memory

1:7= 1miss:7hit

1:7

0

A[0]

A[1]

A[2]

A[7]

2:14

1

A[8]

A[9]

A[15]

3:21

2

A[16]

A[17]

A[23]

14

16:112

15

A[127]

Strided access: Reduce locality

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

Row major
access: Stride 1,
improve locality,
cache hit

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

Column major
access: Stride N,
No locality, cache
miss dominates

Matrix mult.c

```
int A[8][8], B[8][8], C[8][8];
for (i=0; i<8; i++) {
    for (j=0; j<8; j++) {
        S=0;
        for (k=0; k<8; k++)
            S=S+B[i][k]*C[k][j];
        A[i][j]=S;
    }
}
```


Data Size > Cache Size

- $(64+64+64) > 128$ words
- When we get into cache it can take benefit
for $(k=0; k<8; k++)$

$S = S + \mathbf{B[i][k]} * \mathbf{C[k][j]};$

- Inner loop execute for **64 times**
 - We have to get B[j] once will have 1miss/7 hit
 - C[k] have to bring every time 8miss
 - Total = **7h+9m**
- 2nd loop A have one miss in 8 access (**1miss/7hit**)
 - Total for A= **8m+56h**
- Total program : $64 * (7h+9m) + 8m+56h = 504h+584m$
- **Miss Probability = $584/(504+584)=0.5367$**

Improving Locality

Matrix Multiplication example

$$[C] = [A] \times [B]$$

$L \times M$

$L \times N$

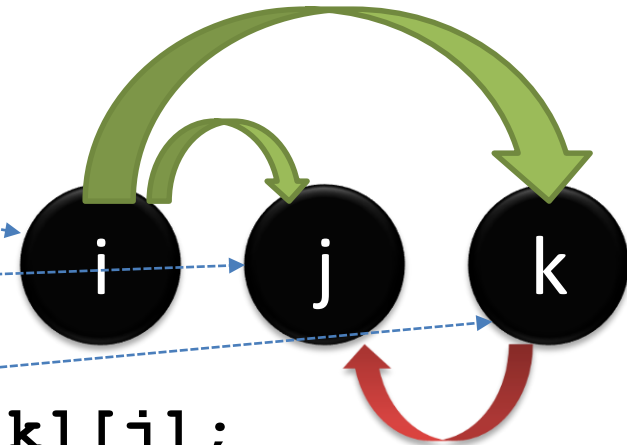
$N \times M$

Cache Organization for the example

- Cache line (or block) = 8 matrix elements.
- Matrices are stored row wise.
- **Cache can't accommodate a full row/column.**
 - **L, M and N are so large w.r.t. the cache size**
 - After an iteration along any of the three indices, when an element is accessed again, it results in a miss.
- Ignore misses due to conflict between matrices.
 - As if there was a **separate cache for each matrix.**

Matrix Multiplication : Code I

```
for (i = 0; i < L; i++)  
  for (j = 0; j < M; j++)  
    for (k = 0; k < N; k++)  
      C[i][j] += A[i][k] * B[k][j];
```



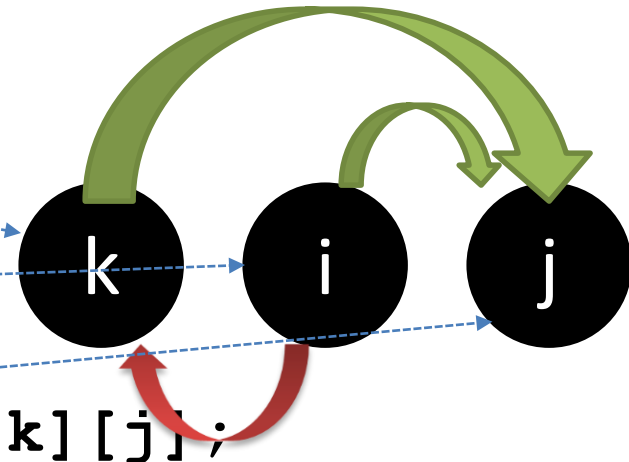
	C	A	B
accesses	LM	LMN	LMN
misses	LM/8	LMN/8	LMN

Total misses = $LM(9N+1)/8$

$L=M=N=100$; miss = $100*100*901/8=1,126,250$

Matrix Multiplication : Code II

```
for (k = 0; k < N; k++)  
  for (i = 0; i < L; i++)  
    for (j = 0; j < M; j++)  
      C[i][j] += A[i][k] * B[k][j];
```



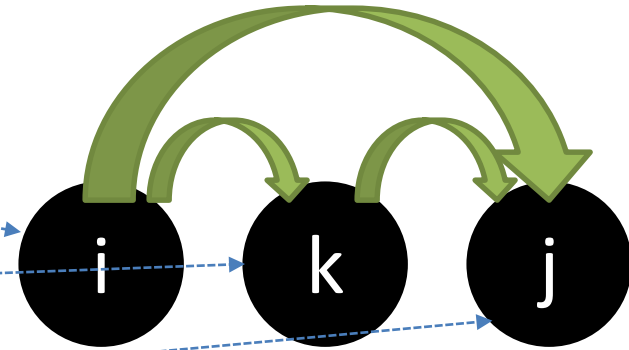
	C	A	B
accesses	LMN	LN	LMN
misses	LMN/8	LN	LMN/8

Total misses = $LN(2M+8)/8$

$L=M=N=100$; miss = $100*100*208/8=260,000$

Matrix Multiplication : Code III

```
for (i = 0; i < L; i++)  
  for (k = 0; k < N; k++)  
    for (j = 0; j < M; j++)  
      C[i][j] += A[i][k] * B[k][j];
```



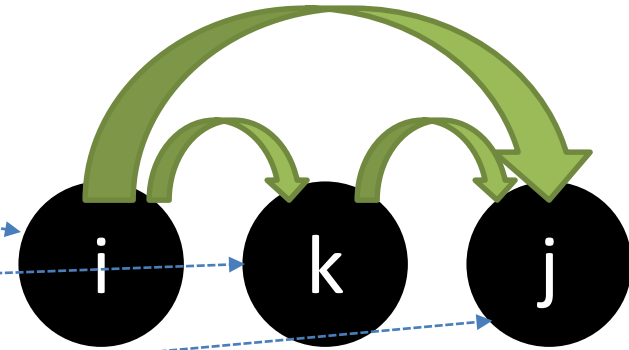
	C	A	B
accesses	LMN	LN	LMN
misses	LMN/8	LN/8	LMN/8

Total misses = $LN(2M+1)/8$

$L=M=N=100$; miss = $100*100*201/8=251,250$

Matrix Multiplication : Code III

```
for (i = 0; i < L; i++)  
  for (k = 0; k < N; k++)  
    for (j = 0; j < M; j++)  
      C[i][j] += A[i][k] * B[k][j];
```



All most all modern processor uses

- Cache block pre-fetch
- When i th block is getting used $i+1$ block prefetched
- **Perfect overlap : only three cache miss**
 - **Each for A, B, C**

