

CS528

Data Access Optimization and Caching

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

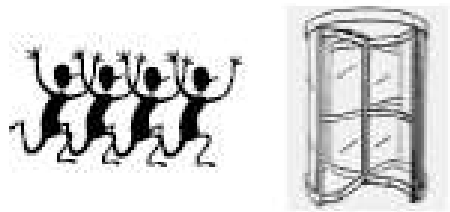
[Ref: Hager Book, PDF uploaded to Website]

Performance of System: Modeling Customer Dispatch in a Bank

Resolving door

Throughput:

b_s [customer/sec]



Processing
Capability:

P_{peak} [task/sec]

Intensity:

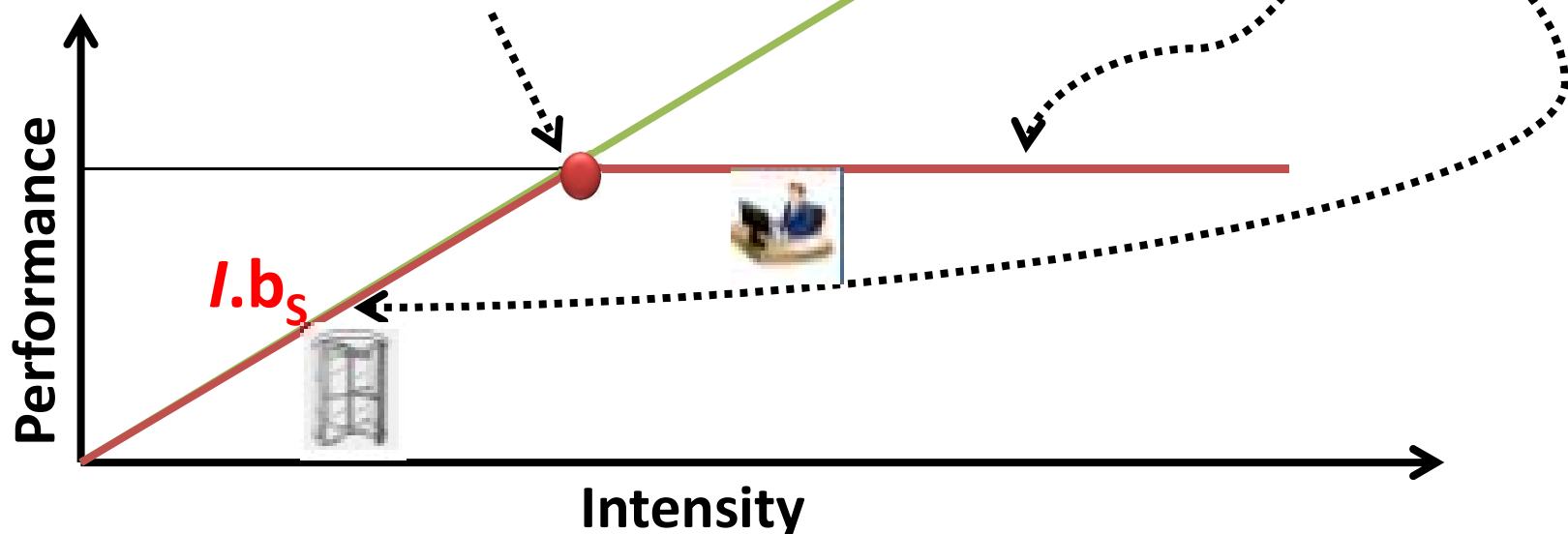
I [task/customer]

Modeling Customer Dispatch in a Bank

- How fast can tasks be processed? $P[\text{tasks/sec}]$
- The bottleneck is either
 - The service desks (peak. tasks/sec): P_{peak}
 - The revolving door (max. customers/sec): $I \cdot b_s$
- 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”

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]

Apply Roof line to Machine and Code

- Machine Parameter 1 : $P_{\text{peak}} [\text{F/s}] = 4 \text{ G F/s}$
- Machine Parameter 2 : $b_s [\text{B/s}] = 10 \text{ G B/s}$
- Application Properties: $I [\text{F/B}] = 2\text{F}/8\text{B} = 0.25\text{F/B}$
for(i=0;i<N;i++) s=s+a[i]*a[i]; // double s, a[]
- Performance = $P = \min(P_{\text{peak}}, I * b_s)$
 $= \min(4 \text{ GF/s}, 0.25 \text{ F/B} * 10 \text{ G.B/s})$
 $= \min(4 \text{ GF/s}, 2.5 \text{ GF/s})$
 $= 2.5 \text{ G F/s}$

The Refine Roofline Model

- P_{\max} = Peak performance of a loop assuming that data comes from L1 cache (**is not necessarily P_{peak}**)
- I = Computational intensity (“work” per byte transferred) over the slowest data path utilized (“the bottleneck”),
- **Code Balance $B_c = I^{-1}$ = in Byte per Flop**
- b_s = Applicable peak bandwidth of the slowest data path utilized

Expected performance:


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

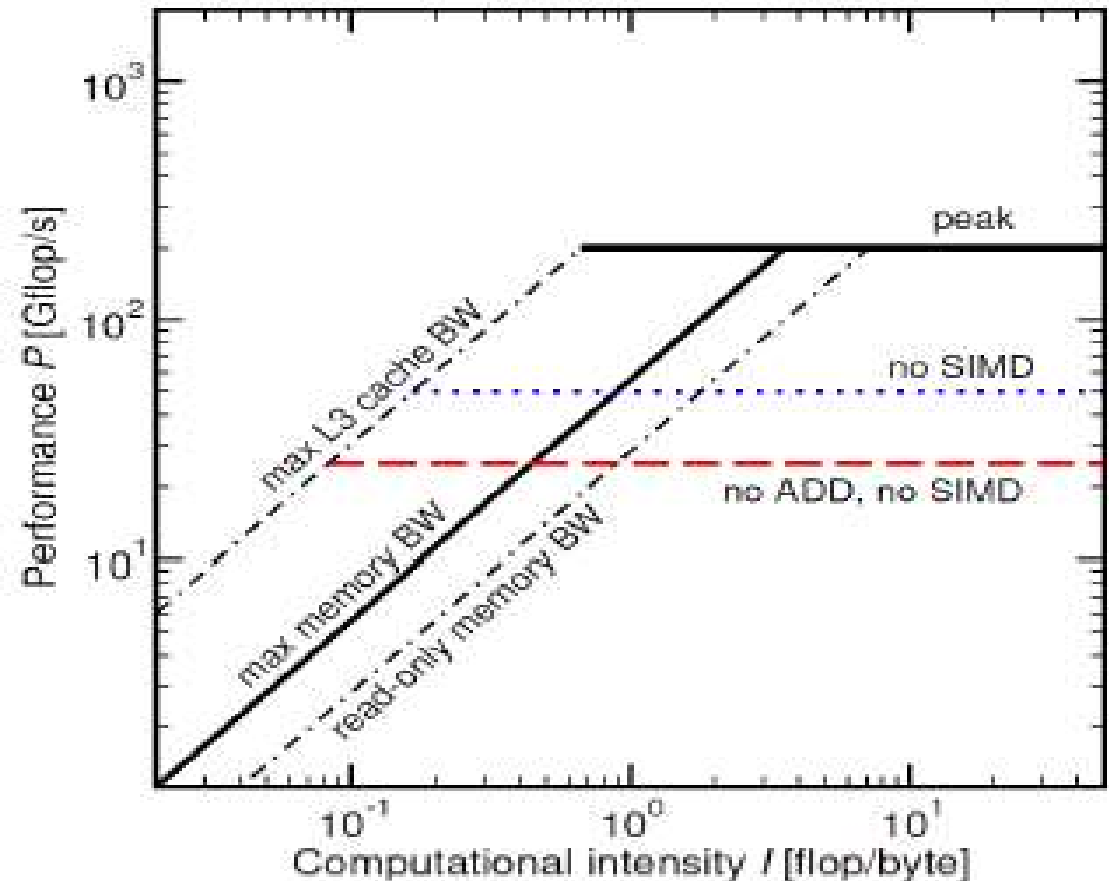
$$P = \min(P_{\max}, b_s / B_c)$$

Factors to consider in Roofline Model

- **BW Bound** : may be simple
 - Accurate traffic calculation (write allocate, stride,..)
 - Practical not equal to theoretical BW limits
 - Saturation effects -> consider full socket only
- **Core Bound** : may be complex
 - Multiple bottlenecks: LD/ST, arithmetic, pipeline, SIMD, execution port
 - Limit is linear in # of cores

Refine RFL model: Graphical Representation

- Multiple ceiling may apply
 - Diff BW/Data paths
-> Diff inclined ceilings
 - Different P_{\max} ->
Diff flat ceilings



P_{\max} comes from code analysis :
with/without SIMD, add other FUs

Apply Roof line to Haswell Core to Triad Code

- Achievable Max Performance 1: $P_{\max} [\text{F/s}] = 12.27 \text{ G F/s}$
- Machine Parameter 2 : $b_s [\text{B/s}] = 50 \text{ G B/s}$
- Application Properties: $I [\text{F/B}] = 2F/40B = 0.05 \text{ F/B}$
for(i=0;i<N;i++) a[i]=b[i]+c[i]*d[i]; // double a,b,c,d
- Performance = $P = \min(P_{\max}, I * b_s)$
 $= \min(12.27 \text{ GF/s}, 0.05 \text{ F/B} * 50 \text{ G.B/s})$
 $= \min(12.27 \text{ GF/s}, 2.5 \text{ GF/s})$
 $= 2.5 \text{ G F/s}$

Code Balance/Intensity Examples

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

$$B_c = 24B/0F = NA$$

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

$$B_c = 24B/1F = 24 \text{ B/F}$$

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

$$B_c = 32B/1F = 32 \text{ B/F}$$

`A[i]=B[i];` //Require reading of A[i], B[i] and writing to A[i]
incase of L1 cache access; so 2 read and one write

Code Balance/Intensity Examples

```
double a[N], b[N], c[N], d[N];  
for (i=0; i<N; i++)  
    a[i] = a[i] + b[i];
```

$$B_c = 24B/1F = 24 \text{ B/F}$$
$$I = 0.042 F/B$$

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

$$B_c = 24B/2F = 12 \text{ B/F}$$
$$I = 0.083 F/B$$

```
for (i=0; i<N; i++) //float a[]  
    s = s + a[i]*a[i];
```

$$B_c = 4B/2F = 2 \text{ B/F}$$
$$I = 0.5 F/B$$

```
for (i=0; i<N; i++) //float a[], b[]  
    s = s + a[i]*b[i];
```

$$B_c = 8B/2F = 4 \text{ B/F}$$
$$I = 0.25 F/B$$

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

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

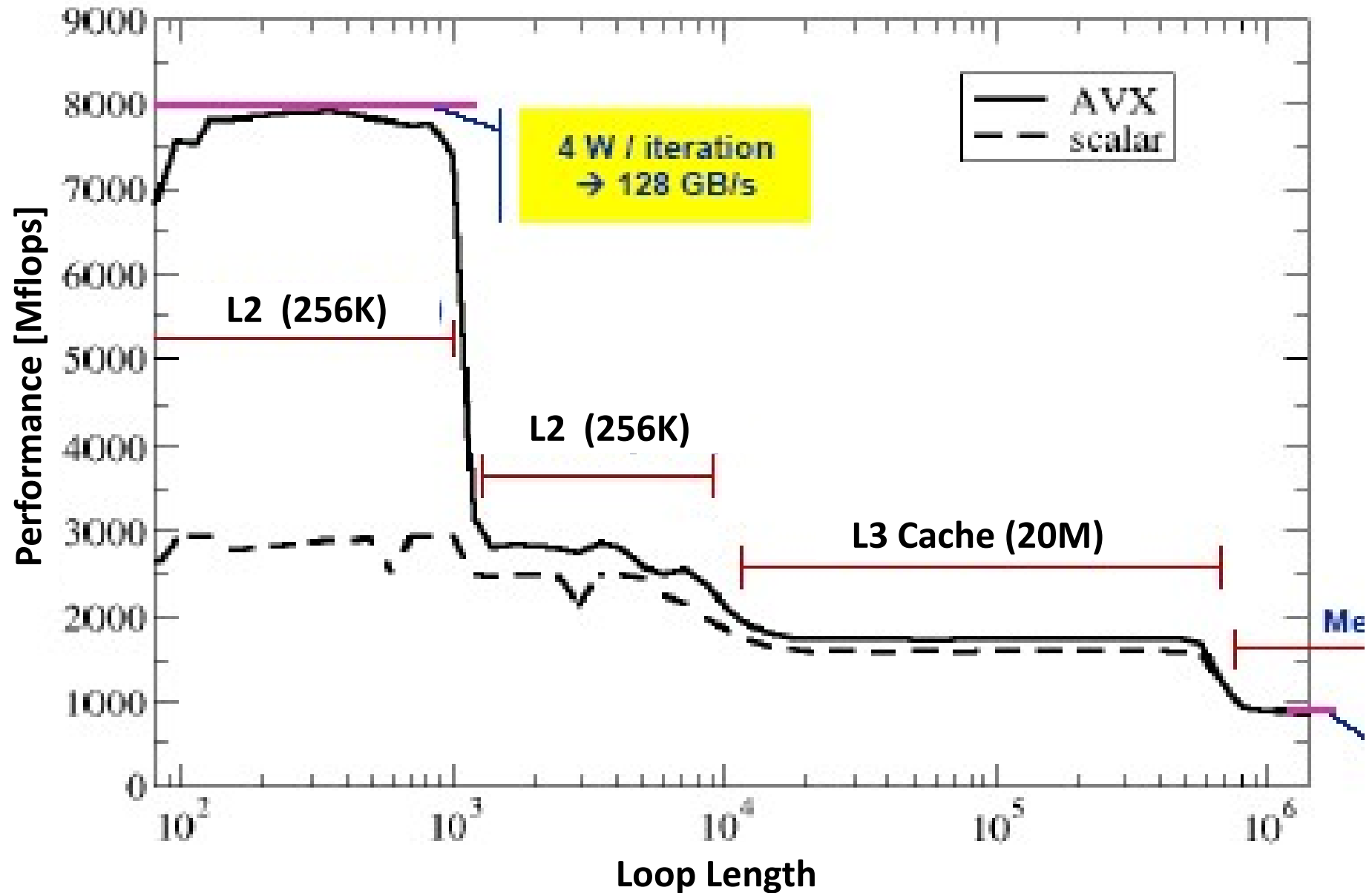
Memory Analysis of Simple Code

- 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

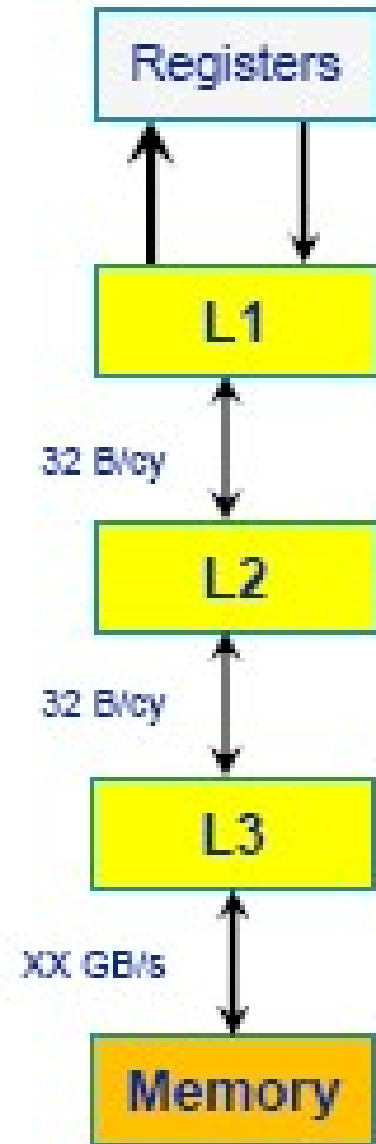


Memory Hierarchy

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

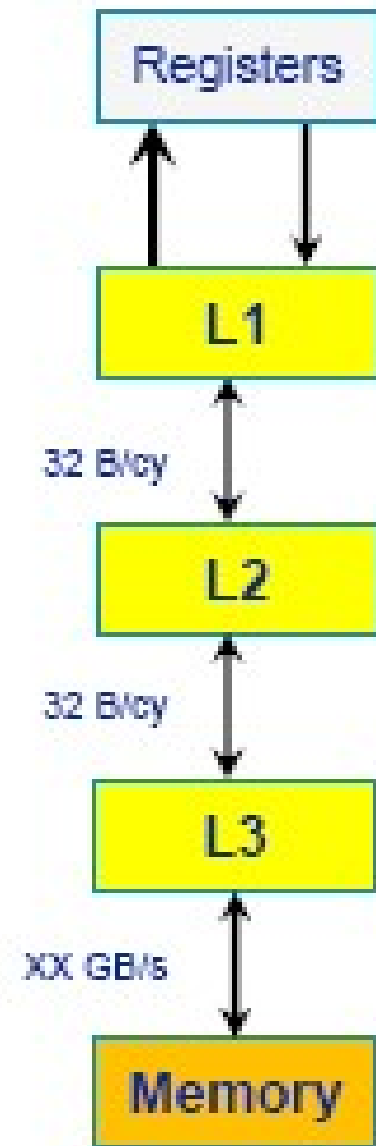
Throughput capabilities: i7 5960 Xtreme

- **Per cycle with AVX**
 - 1 load instruction (256 bits) AND $\frac{1}{2}$ store instruction (128 bits)
 - 1 AVX MULT and 1 AVX ADD instruction (4 DP / 8 SP flops each)
 - Overall maximum of 4 micro-ops

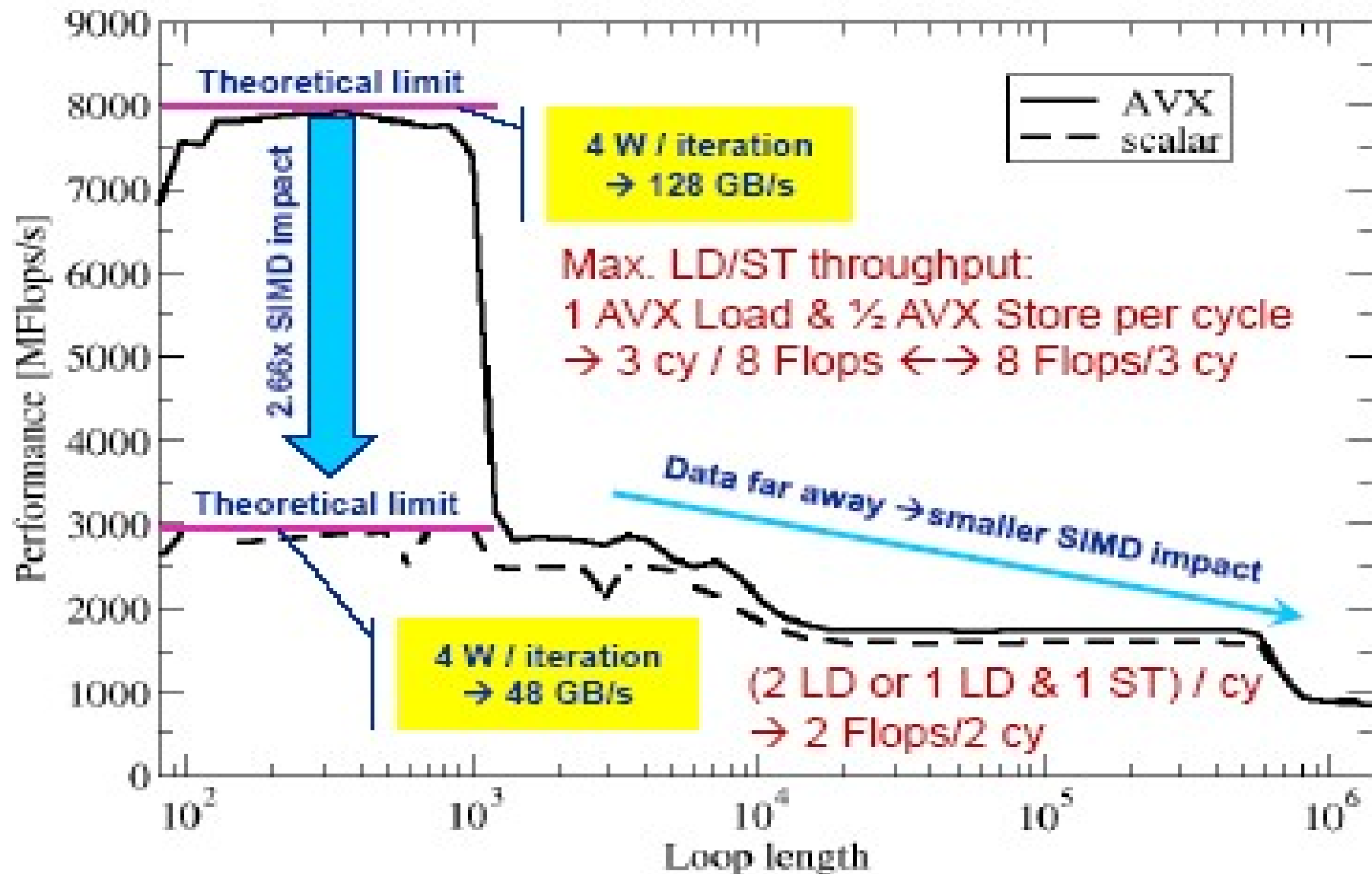


Throughput capabilities: i7 5960 Xtreme

- 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)



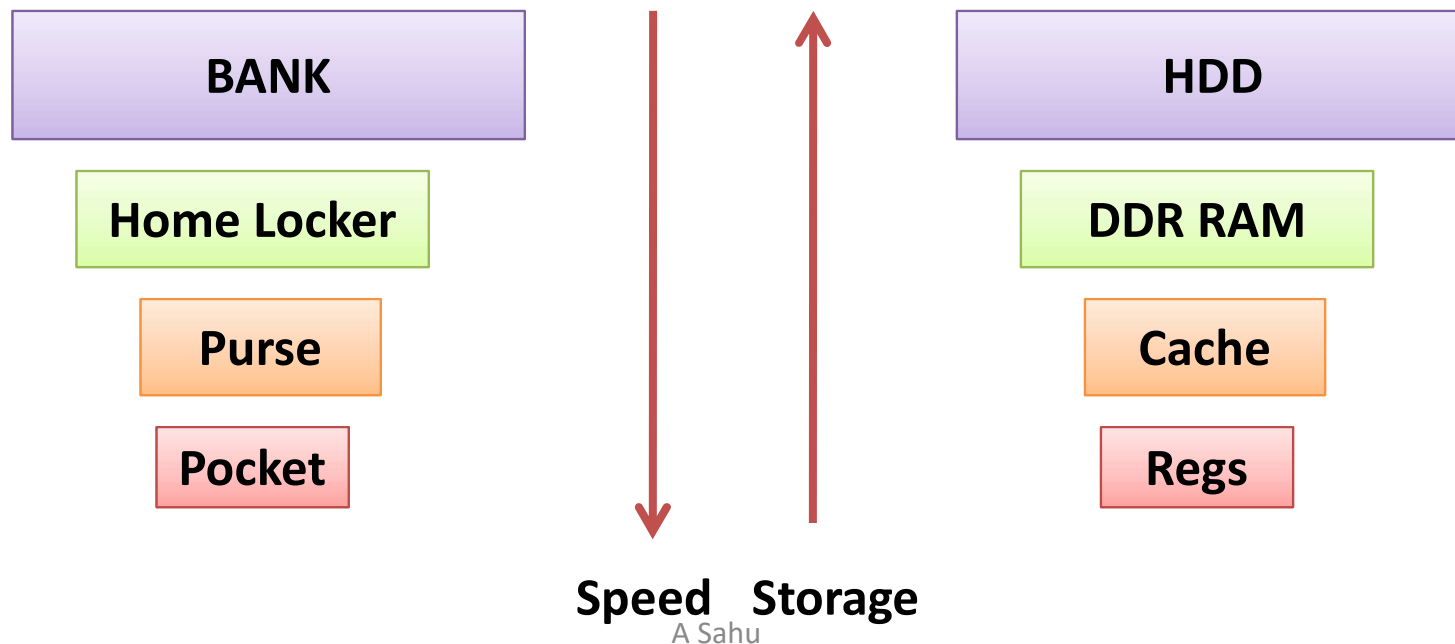
On Sandy Bridge: Core i7 5960 Xtreme



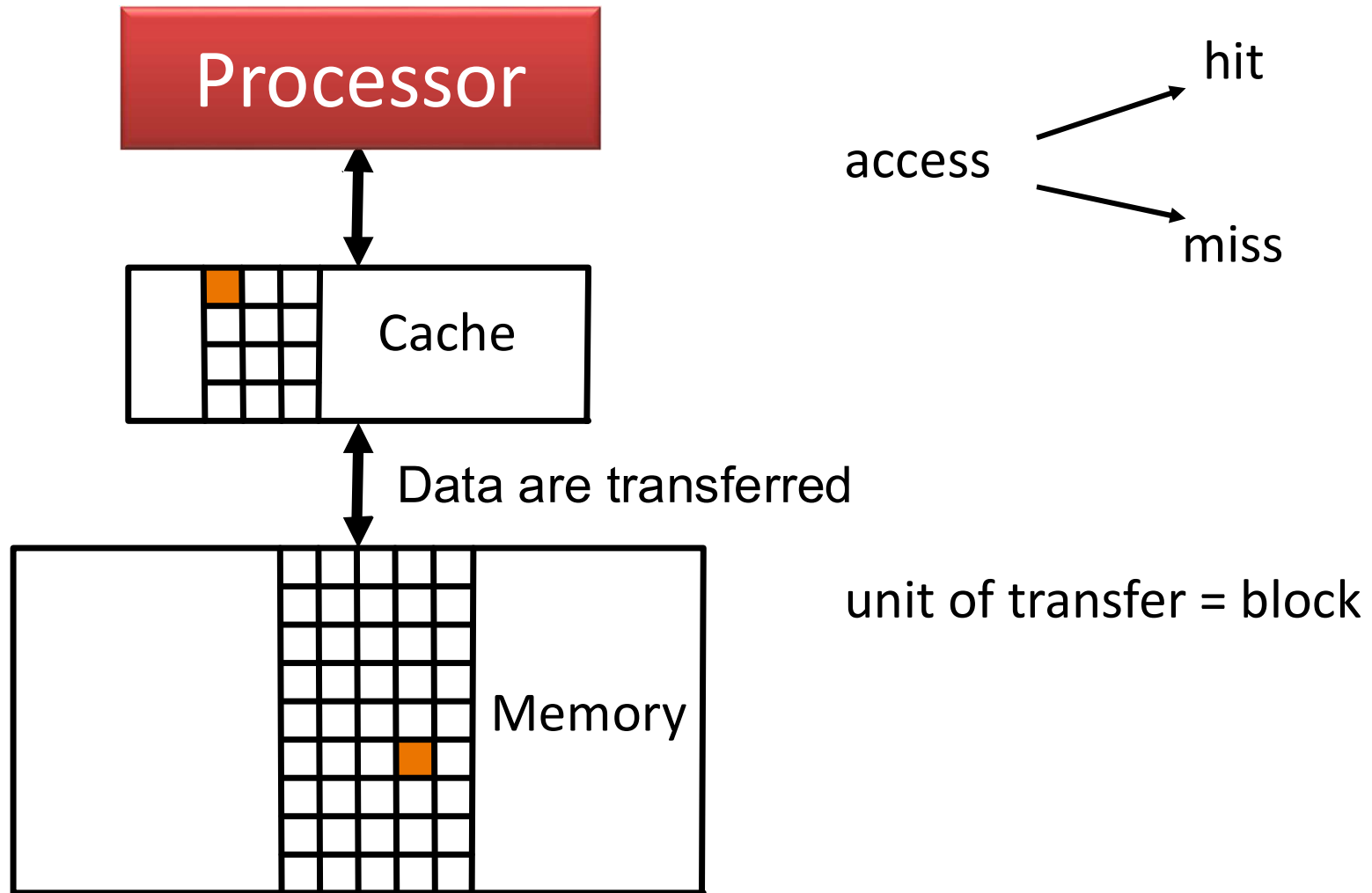
**Reducing Code Balance
(Byte/Flop) using Memory
Hierarchy : Caching**

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)

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

- **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

Direct/Random
Access to
Element

- 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

MIXED

- List of Element

- Full Associative Cache
- All data stored in one list

Serial/Associative Access
to Element

U
S
A
B
I
L
I
T
Y

T
I
M
E