CS528

Data Access Optimization Part II

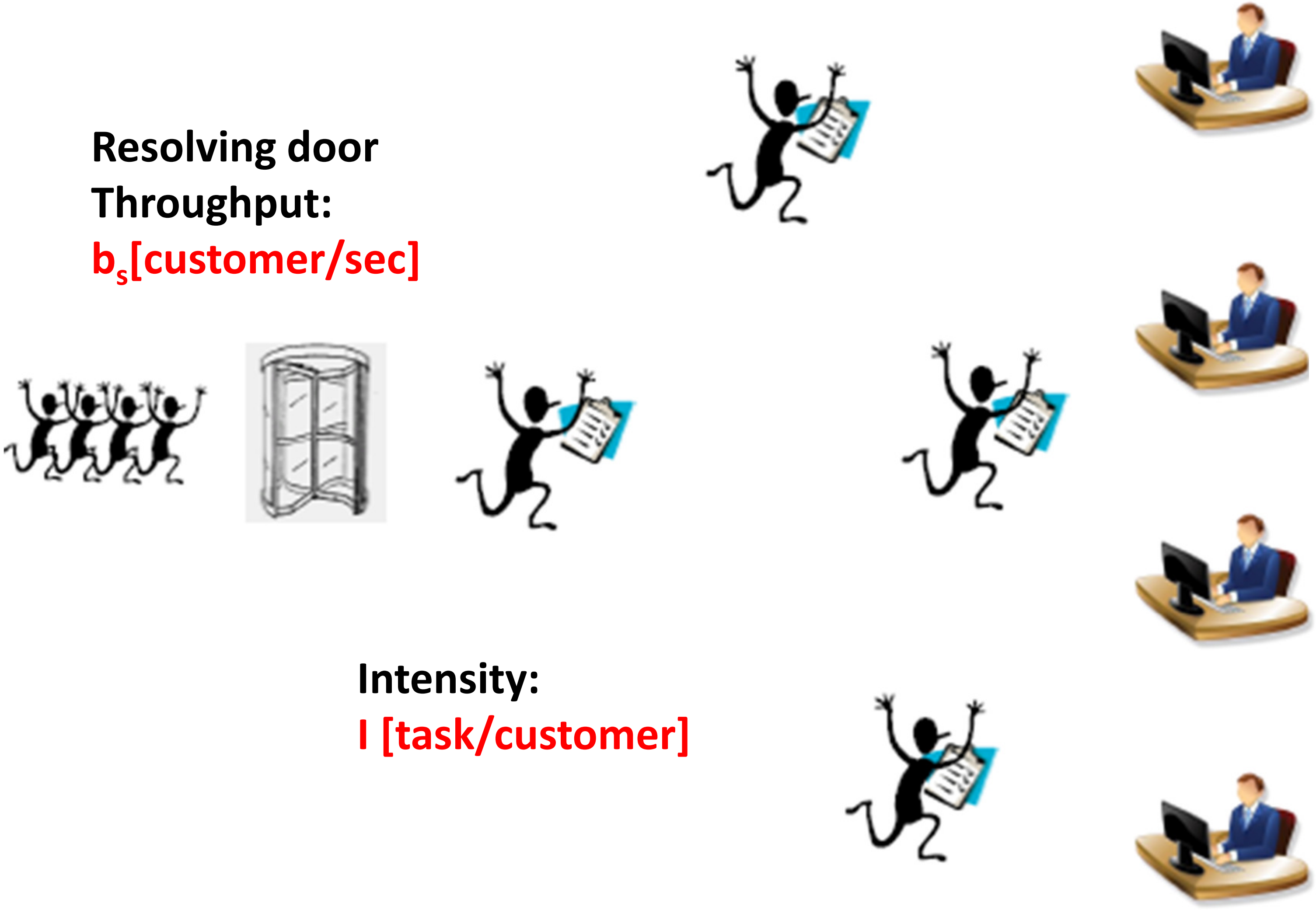
A Sahu

Dept of CSE, IIT Guwahati

Outline

- Machine Balance
- Application Balance
- Performance: **Roofline Model**
- Benchmark : Data Access
- Test Cases

Performance of System: Modeling Customer Dispatch in a Bank



Resolving door
Throughput:
b_s[customer/sec]

Processing
Capabilty:
P_{peak} [task/sec]

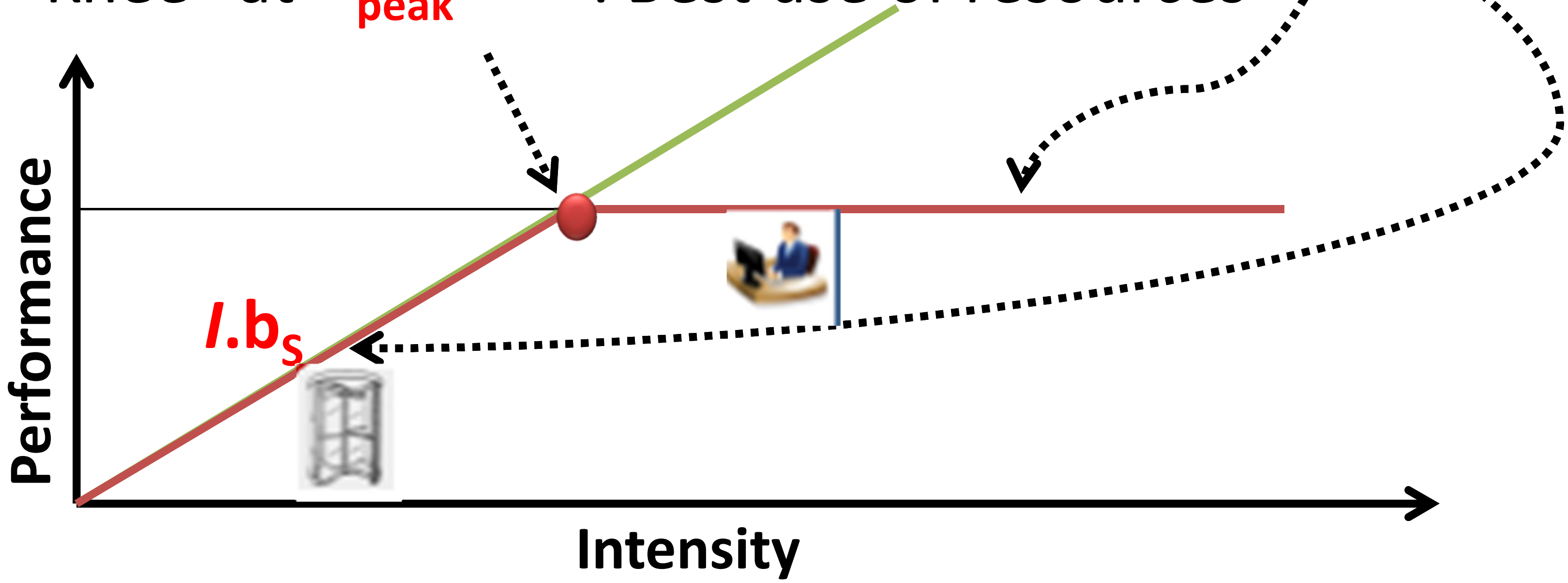
Intensity:
I [task/customer]

Modeling Customer Dispatch in a Bank

- How fast can tasks be processed? P**[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_{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 =min($_{\text{peak}}$, ·)
- **This is the “Roofline Model”**
  - High intensity: P limited by “execution”
  - Low intensity: P limited by “bottleneck”
  - “Knee” at $_{\text{peak}}$= · : Best use of resources



- Roofline is an “optimistic” model

The Roofline Model

- $\boldsymbol{P}_{\textbf{max}}$= Peak performance of the machine
- ***I***= Computational intensity ("work" per byte transferred) over the slowest data path utilized ("the bottleneck")
- $\boldsymbol{b}_{\boldsymbol{S}}$= Applicable peak bandwidth of the slowest data path utilized

Expected performance:

$$P = \min(P_{\text{peak}}, \overset{[F/B]}{I} \cdot \overset{[B/s]}{b_S})$$

Apply Roof line to Machine and Code

- Machine Parameter 1 : P_{peak} [F/s]=4 G F/S
- Machine Parameter 2 : b_S [B/s] =10 G B/s
- Application Properties: I [F/B] = 2F/8B=0.25F/B

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

- Performance = P = min(P_{peak}, I*b_s)

  =min(4 GF/s, 0.25 F/B *10 G.B/s)

  =min(4 GF/s, 2.5 GF/s)

  =2.5 G F/s

The Refine Roofline Model

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

Expected performance:

$$P = \min\left(P_{\mathrm{max}},\ \overset{\overset{[\mathrm{F/B}]}{\downarrow}}{I} \cdot \overset{\overset{[\mathrm{B/s}]}{\swarrow}}{b_S}\right)$$

$$P = \min\left(P_{\mathrm{max}},\ b_S/\mathrm{B_c}\right)$$

Factors to consider in Roofline Model

BW Bound : may be simple

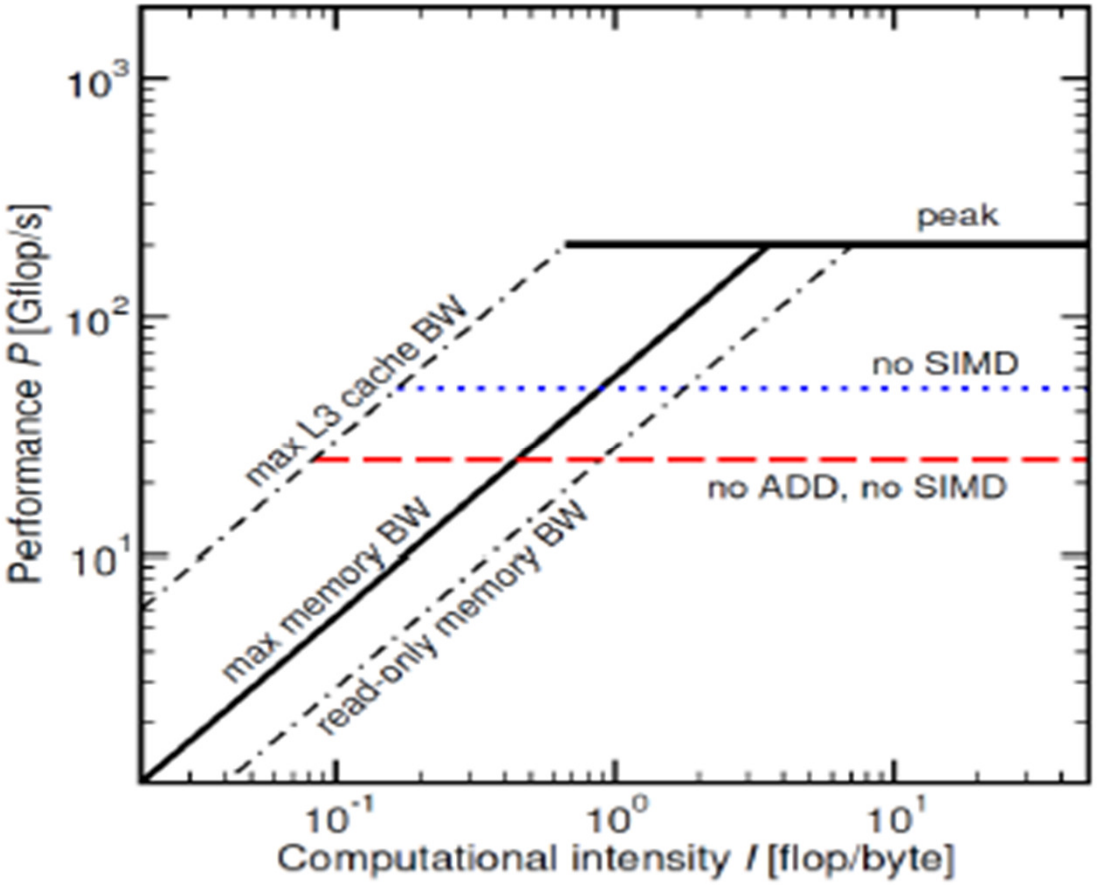
- Accurate traffic calculation (write allocate, stride,..)
- Practical not equal to theoretical BW limits
- Saturation effects -> consider full socket only

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, aby other FUs

Apply Roof line to Haswell Core to Triad Code

- Achievable Max Performance 1 : P_{max} [F/s]=12.27 G F/S
- Machine Parameter 2 : b_S [B/s] =50 G B/s
- Application Properties: I [F/B] = 2F/40B=0.05F/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 GF/s, 0.05 F/B *50 G.B/s)

  =min(12.27 GF/s, 2.5 GF/s)

  =2.5 G F/s

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 B/F
I=0.042F/B

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

B_c=24B/2F=12 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 B/F
I=0.05 F/B

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

B_c=8B/2F=4 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)

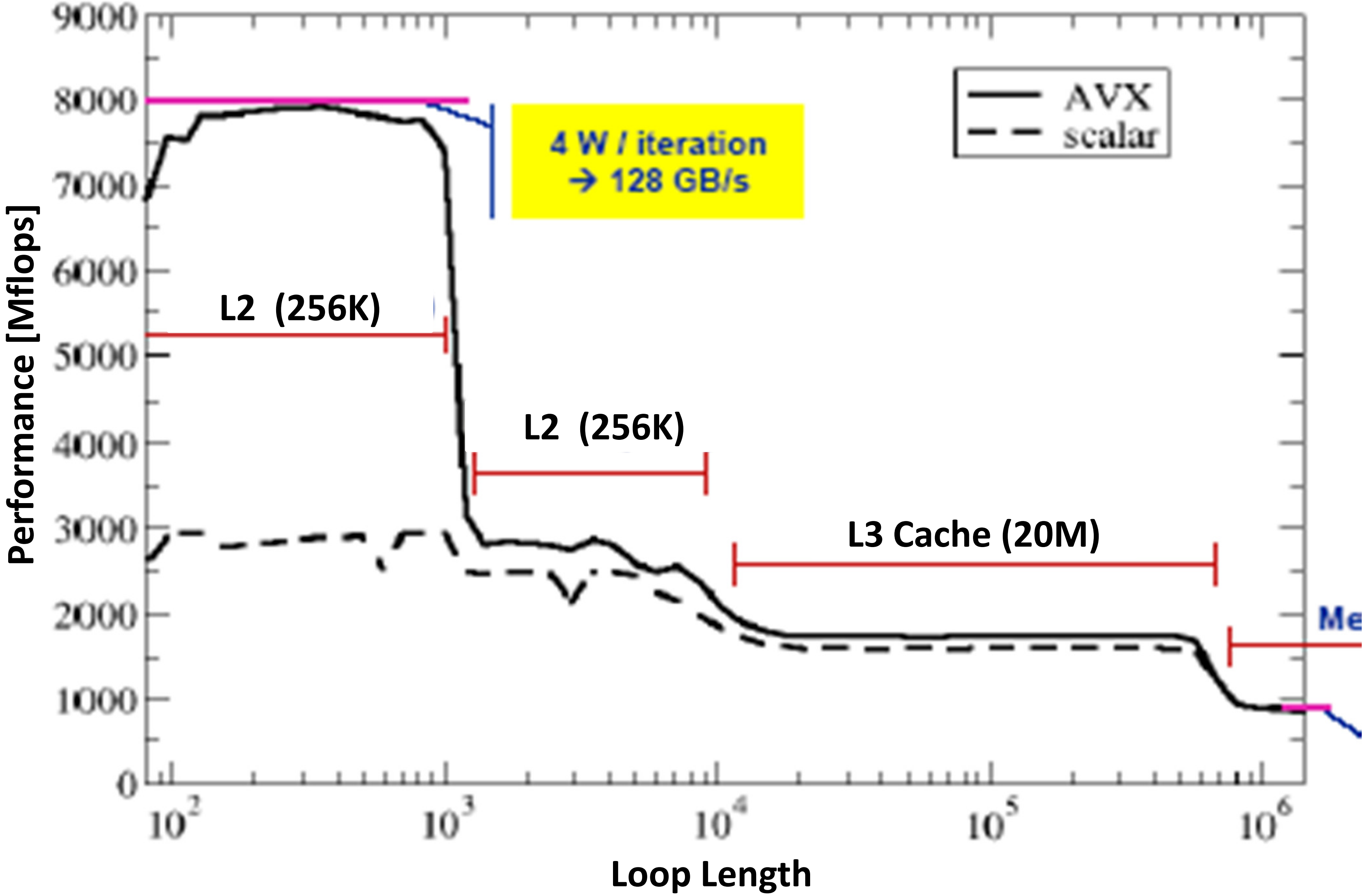
Memory Analysis of Simple Code

- 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



4 W / iteration
→ 128 GB/s

L2 (256K)

L2 (256K)

L3 Cache (20M)

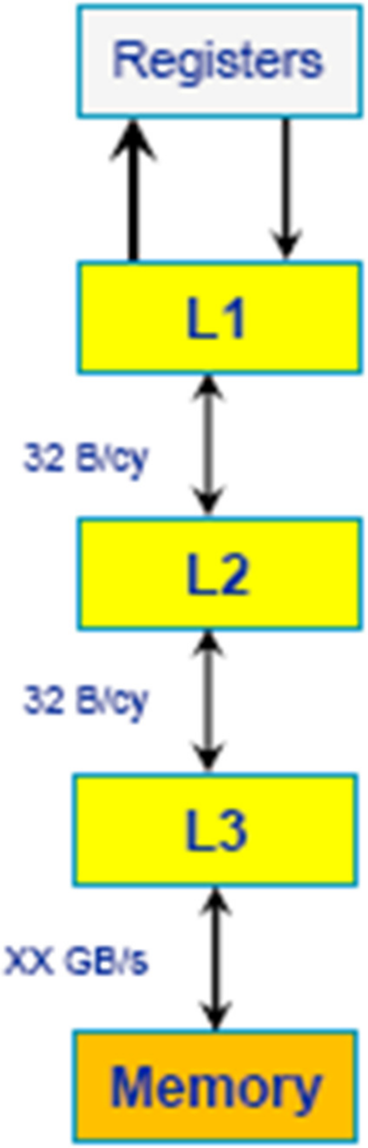
Me

Memory Hierarchy

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

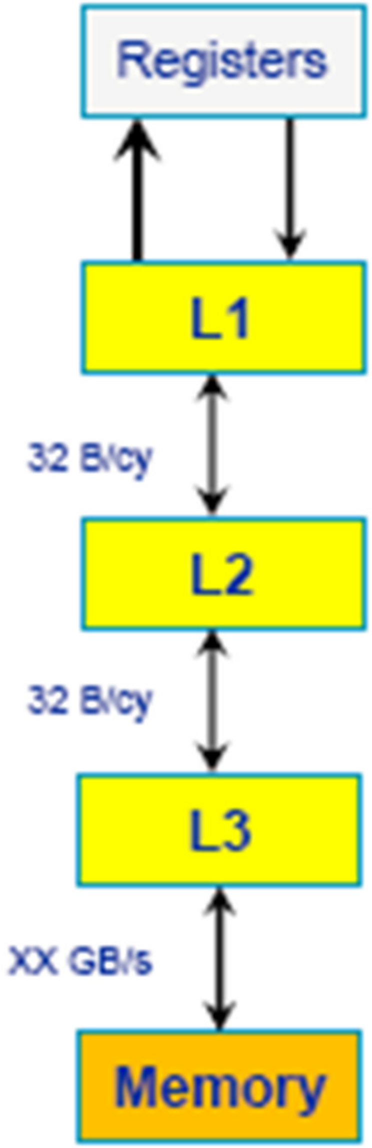
Throughput capabilities: i7 5960 Xtreme

- 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

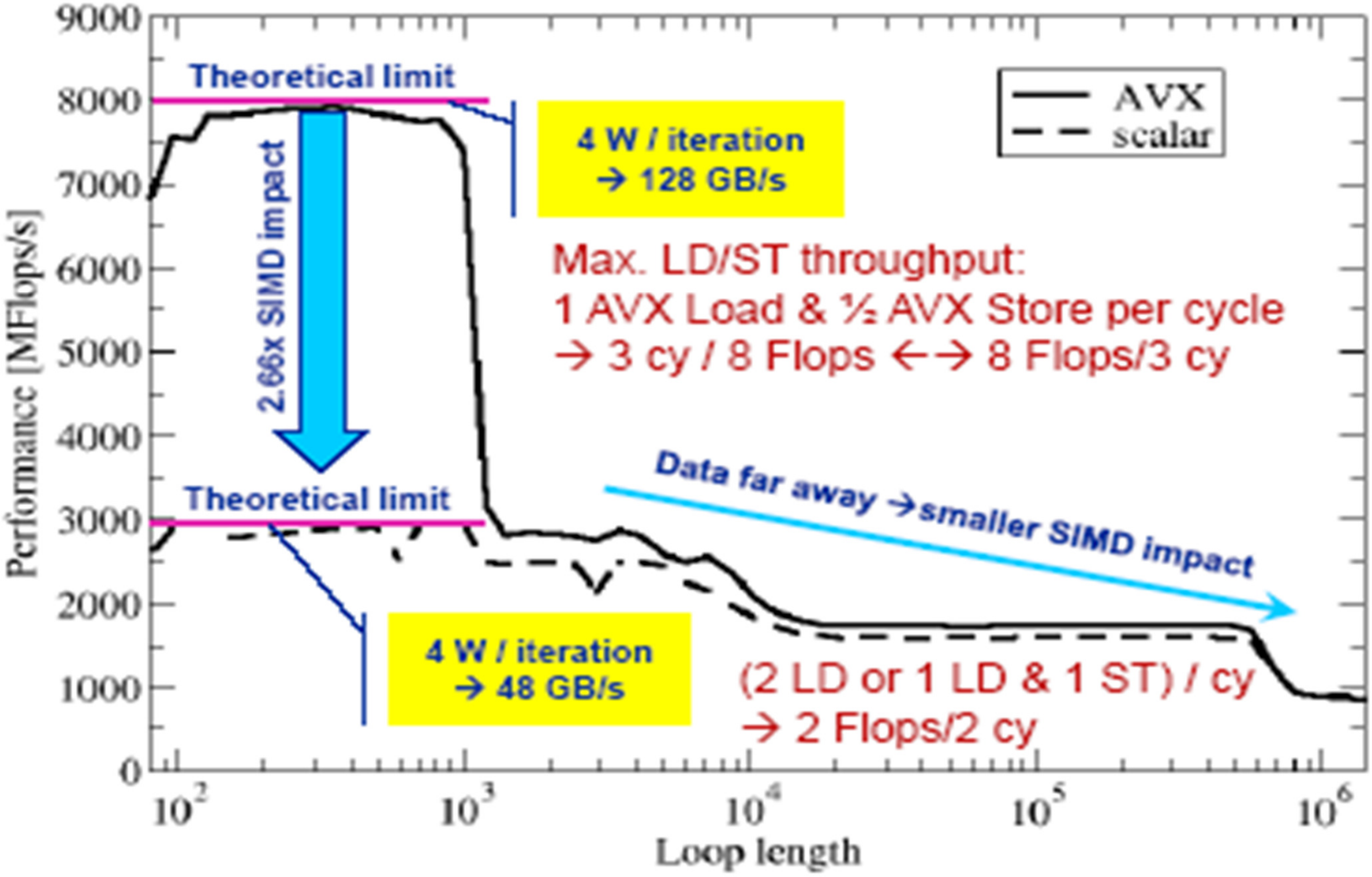


Throughput capabilities: i7 5960 Xtreme

- 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 ↔ L2, L2 ↔ L1)
  - 256 bits per cycle, half-duplex (i.e., ful CL transfer == 2 cy)



On Sandy Bridge: Core i7 5960 Xtreme



Theoretical limit

4 W / iteration
→ 128 GB/s

Max. LD/ST throughput:
1 AVX Load & ½ AVX Store per cycle
→ 3 cy / 8 Flops ←→ 8 Flops/3 cy

Data far away →smaller SIMD impact

Theoretical limit

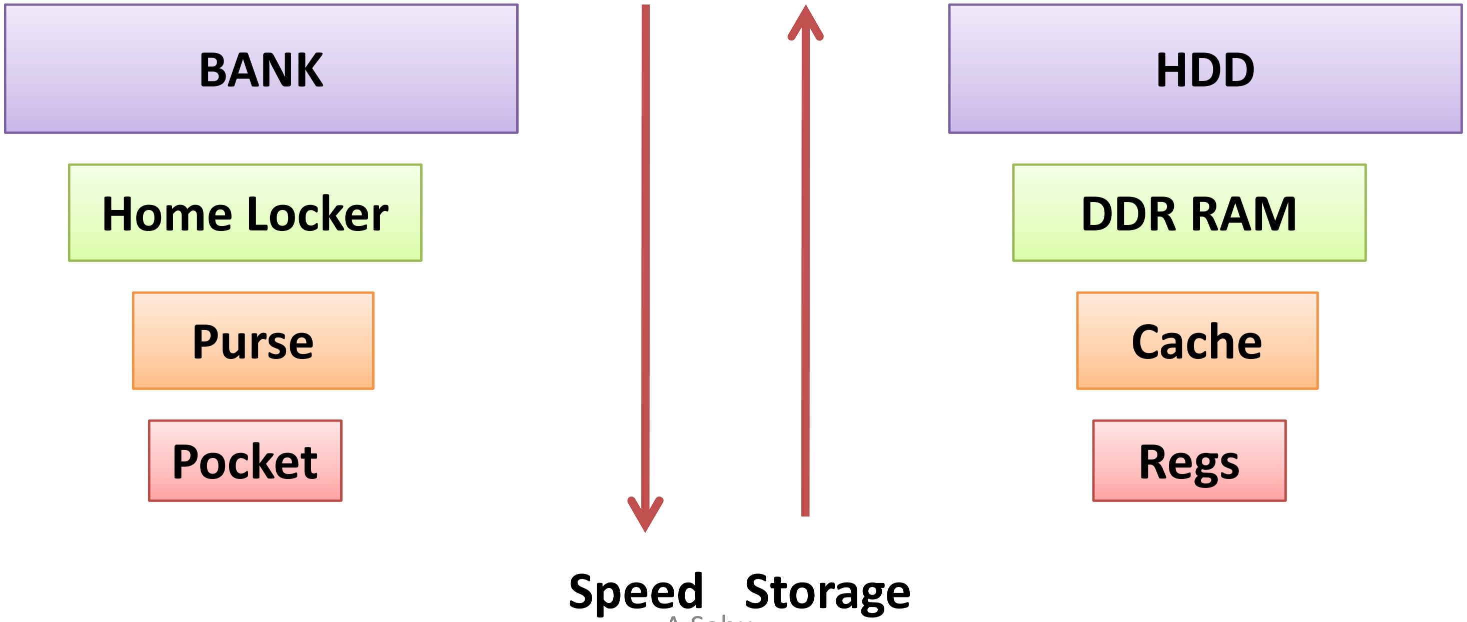
4 W / iteration
→ 48 GB/s

(2 LD or 1 LD & 1 ST) / cy
→ 2 Flops/2 cy

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

Memory Hierarchy

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



BANK

Home Locker

Purse

Pocket

HDD

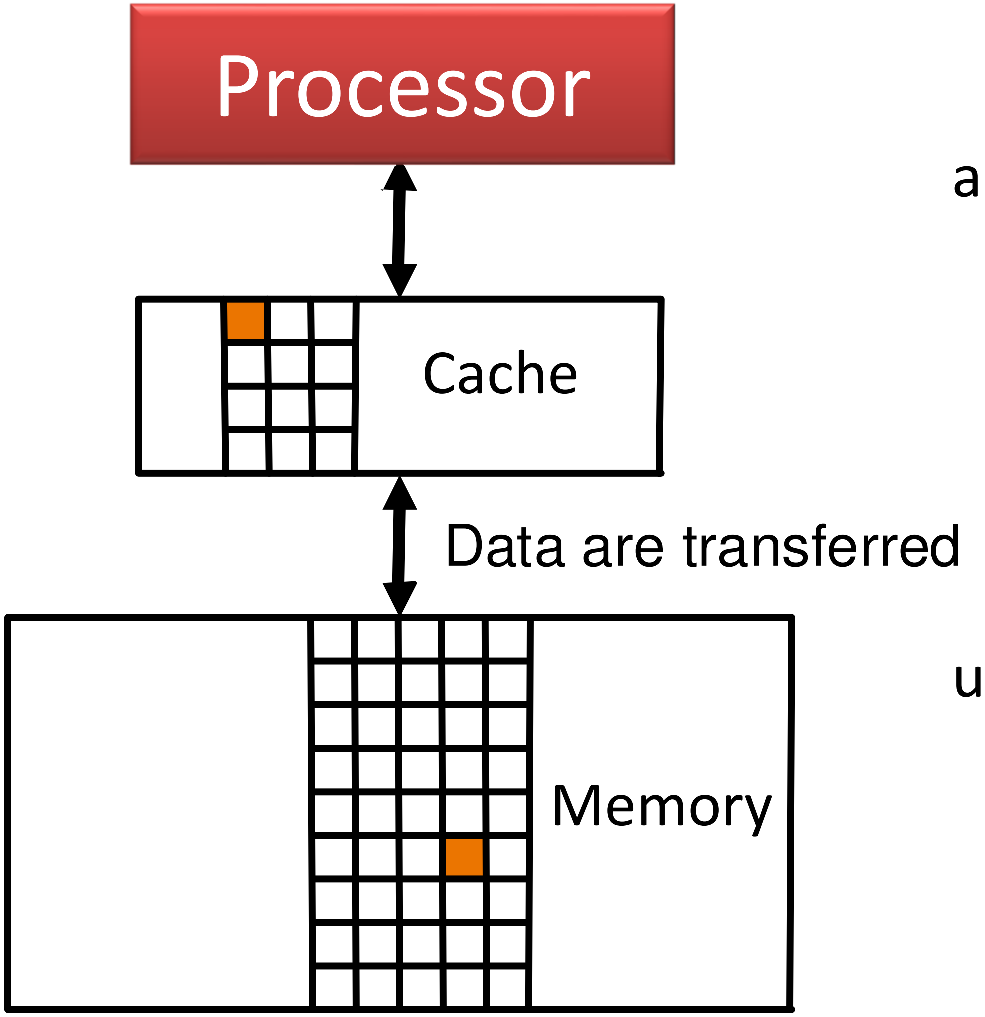
DDR RAM

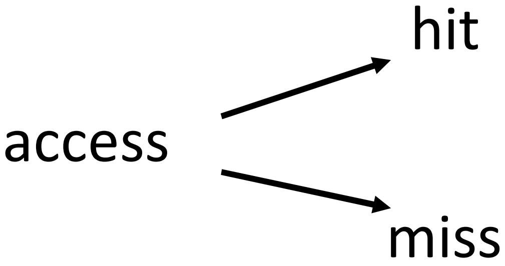
Cache

Regs

Speed Storage

Data transfer between levels





unit of transfer = block

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

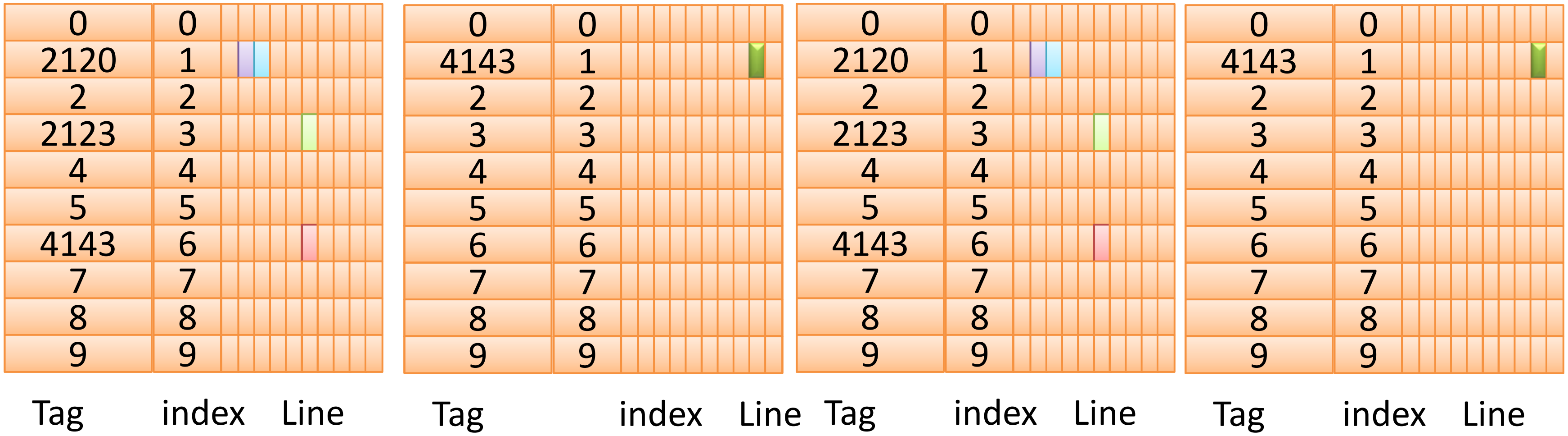
- Address is divided in to three part : TAG, Index, Offset
  - Offset = Address % Line Size,
  - Index = (Address/LineSize)%NumSet
  - TAG = Address/(LineSize*NumSet)
- If TAG matches with ExistingTAG then HIT else miss

```
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)
- Associatively (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

USABILITY

- Simple Hashing: Direct Map Cache
  - Example: Array
  - int A[10], each can store one element
  - Data stored in Addr%10 location
- Array of List
  - Int LA[10], each can store a list of element
  - Data stored in List of (Addr%10)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

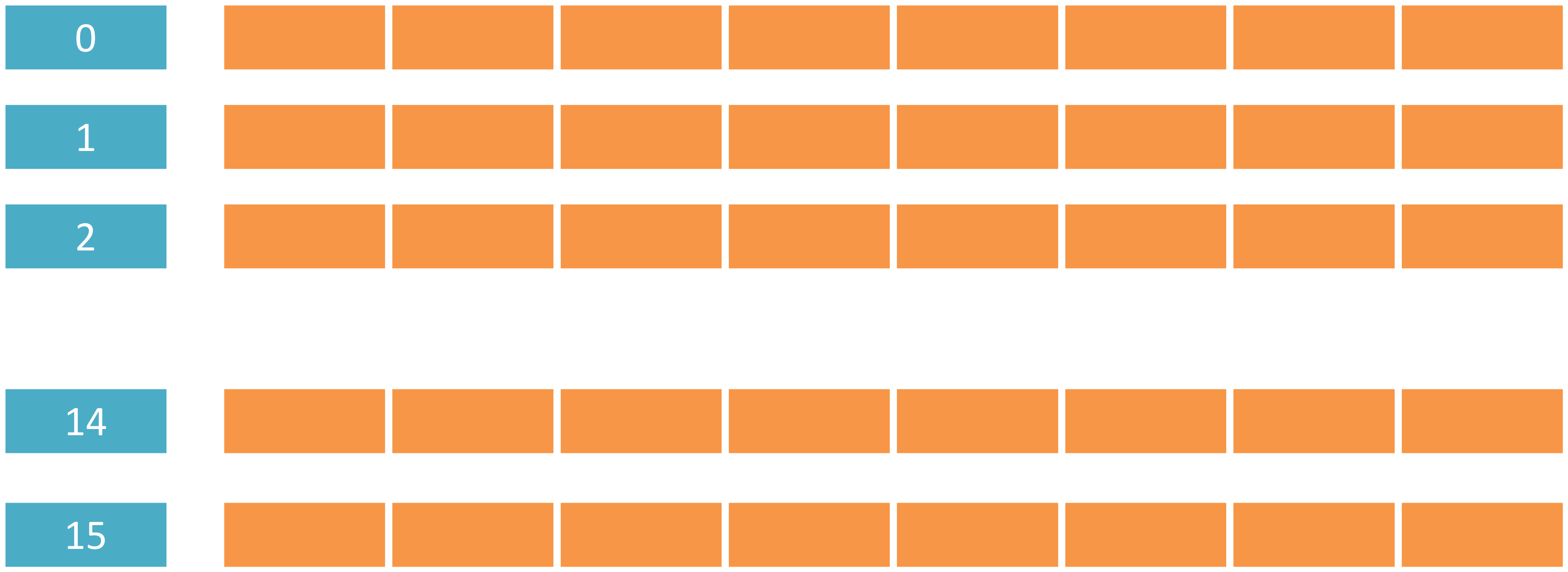
Serial/Associative Access to Element

TIME

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]

Cache perf. : Data Size > Cache Size

```
int A[512];
for(i=0;i<512;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]**

Cache perf. : Data Size <= Cache Size

```
int A[512];
for(i=0;i<512;i++){
   A[i]=i;
}
```

Scalar mapped to register
Vector mapped to memory

32:224= ==➔
64m:448h

17:120	**0**	**A[128]**	**A[129]**	**A[130]**					**A[135]**
18:127	**1**	**A[136]**	**A[137]**						**A[143]**
19:134	**2**	**A[144]**	**A[145]**						**A[151]**
	14								
32:224	**15**								**A[255]**

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;
     }
}
```