

COL380

Introduction to
Parallel & Distributed Programming
2-0-2

Subodh Kumar

Programming Test:

Wednesday, April 23 5-6pm

- HPC

- ➔ Login nodes (ssh hpc.iitd.ac.in) * set up password-less login *
- ➔ Use PBS to submit jobs
- ➔ module add compiler/gcc/11.2/openmpi/4.1.6 (also see: module list)
- ➔ <https://supercomputing.iitd.ac.in>

```
$ amgr login  
$ amgr checkbalance project -n col380.subodh.course
```

- CSS

- ➔ ssh css.cse.iitd.ac.in

- perf, OpenMP, MPI, Cuda (Others)

- Process & thread
 - ➔ Scheduling, Context-switching and concurrency
 - ➔ User space, Kernel space
 - ➔ System calls
- Address space & Name space
- Virtual Memory
 - ➔ Caches
- Synchronization

- ♦ Shell (bash)
- ♦ time
- ♦ PMU (perf)
- ♦ Context switch
 - ♦ and scheduling
- ♦ System calls
- ♦ Interrupts

code.cpp:

```
int turn = 0;
void handler(int sig) { turn = !turn; alarm(1); }

void func1() {
    static int i = 0;
    while(i < many1) {
        if(turn == 0) func0();
        else dothing1(i++);
    }
    turn = 0; alarm(0);
}

void func0() {
    static int i = 0;
    while(i < many0) {
        if(turn == 1) func1();
        else dothing0(i++);
    }
    turn = 1; alarm(0);
}
```

Process, Thread

```
signal(SIGALRM, handler);
alarm(1);
```

```
g++ code.cpp -o exe
./exe &
./exe &
```

code.cpp:

```
int turn = 0;
void handler(int sig) { turn = !turn; alarm(1); }

void func1() {
    static int i = 0;
    while(i < many1) {
        if(turn == 0) func0();
        else dothing1(i++);
    }
    turn = 0; alarm(0);
}

void func0() {
    static int i = 0;
    while(i < many0) {
        if(turn == 1) func1();
        else dothing0(i++);
    }
    turn = 1; alarm(0);
}
```

Process, Thread

```
signal(SIGALRM, handler);
alarm(1);
```

```
g++ code.cpp -o exe
./exe
```

First Parallel Program

```
float dot(float *a, float *b, int n)
{
    int i;
    float s0=0, s1=0, s2=0, s3=0;
    for(i=0; i<n/4*4; i+=4)
        s0 += a[i]*b[i];
        s1 += a[i+1]*b[i+1];
        s2 += a[i+2]*b[i+2];
        s3 += a[i+3]*b[i+3];
    for(; i<n; i++)
        s0 += a[i]*b[i];
    return s0+s1+s2+s3;
}
```

vaddps (%rdi), %zmm1, %zmm0

Single Vector Instruction

Code Execution

```
float dot(float *a, float *b, int n)
{
    int i;
    float s0=0;
    for(i=0; i<n; i++)
        s0 += a[i]*b[i];
    return s0;
}
```

	test %rdx,%rdx	// n == 0? (rdx)
	jle 29 <dot+0x29>	// exit if 0
	mov \$0x0,%eax	// i = 0 (eax/rax)
	pxor %xmm1,%xmm1	// s0 = 0 (xmm1)
e.	movss (%rdi,%rax,4),%xmm0	// a[i] -> tmp (xmm0)
	mulss (%rsi,%rax,4),%xmm0	// mul b[i] to tmp (xmm0)
	addss %xmm0,%xmm1	// add tmp to s0
	add \$0x1,%rax	// i += 1 (rax)
	cmp %rax,%rdx	// i == n? (rdx)
25.	jne e <dot+0xe>	// loop if ne
	movaps %xmm1,%xmm0	// s0-> xmm0
	retq	// return (answer in xmm0)
29.	pxor %xmm1,%xmm1	// s0 = 0 (xmm1)
	jmp 25 <dot+0x25>	// goto exit point

Code Execution

```
float dot(float *a, float *b, int n)
{
    int i;
    float s0=0;
    for(i=0; i<n; i++)
        s0 += a[i]*b[i];
    return s0;
}
```

What's in %rdi?

Virtual address of "a"

1. translate to physical
2. request memory controller

Goes through cache hierarchy

Indeterminate latency:

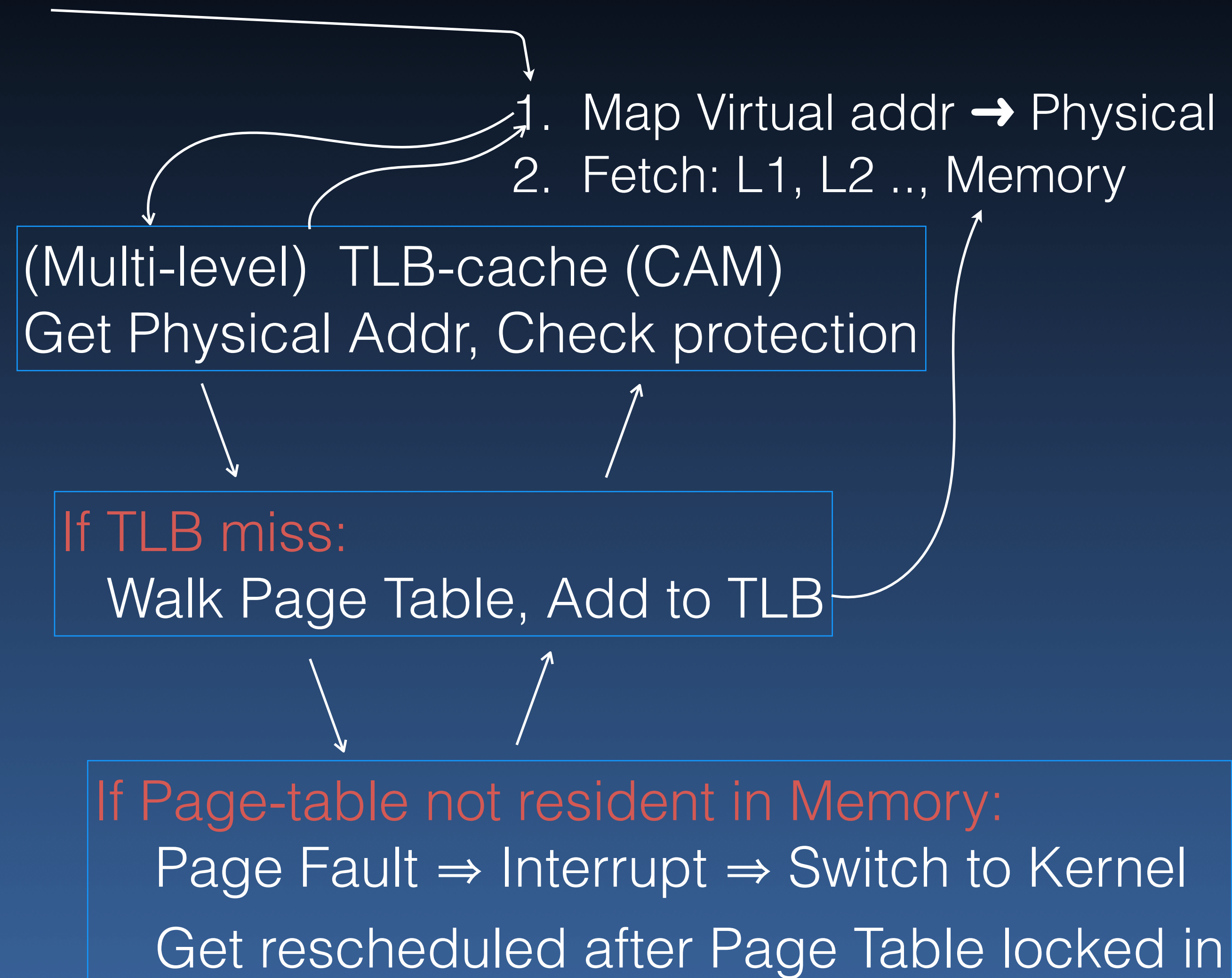
blocks subsequent instructions

i not read/written (from memory)

	test %rdx,%rdx	// n == 0? (rdx)
	jle 29 <dot+0x29>	// exit if 0
	mov \$0x0,%eax	// i = 0 (eax/rax)
	pxor %xmm1,%xmm1	
e.	movss (%rdi,%rax,4),%xmm0	Compiler may swap: // movss (%rsi,%rax,4),%xmm0 // mulss (%rdi,%rax,4),%xmm0
	mulss (%rsi,%rax,4),%xmm0	
	addss %xmm0,%xmm1	Architecture may co-issue/swap : // add tmp to s0 // i += 1 (rax) // i == n (rdx) // loop if ne
	add \$0x1,%rax	
	cmp %rax,%rdx	
25.	jne e <dot+0xe>	
	movaps %xmm1,%xmm0	// s0-> xmm0
	retq	// return (answer in xmm0)
29.	pxor %xmm1,%xmm1	// s0 = 0 (xmm1)
	jmp 25 <dot+0x25>	// goto exit point

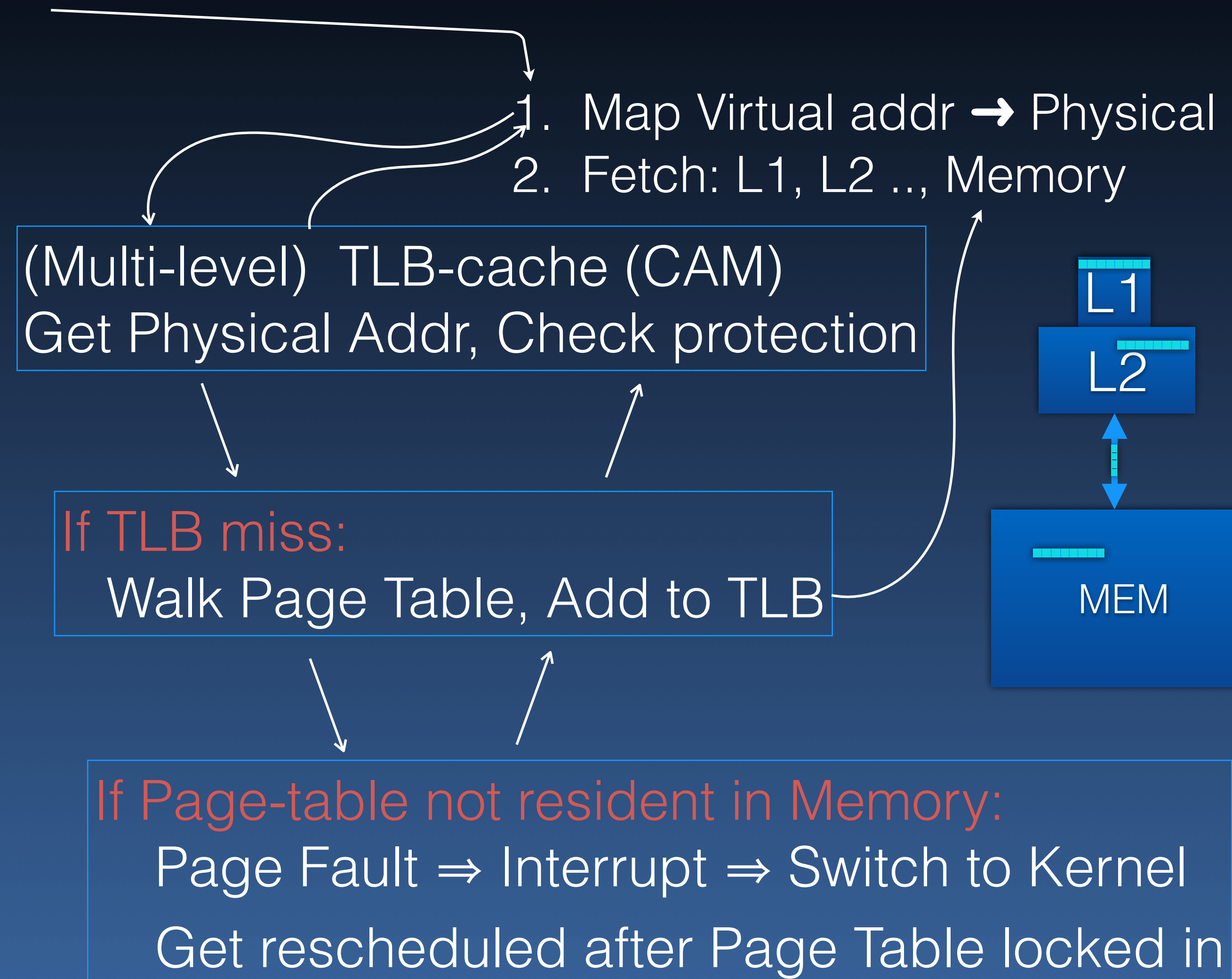
Execution Steps

```
movss (%rdi,%rax,4),%xmm0
```



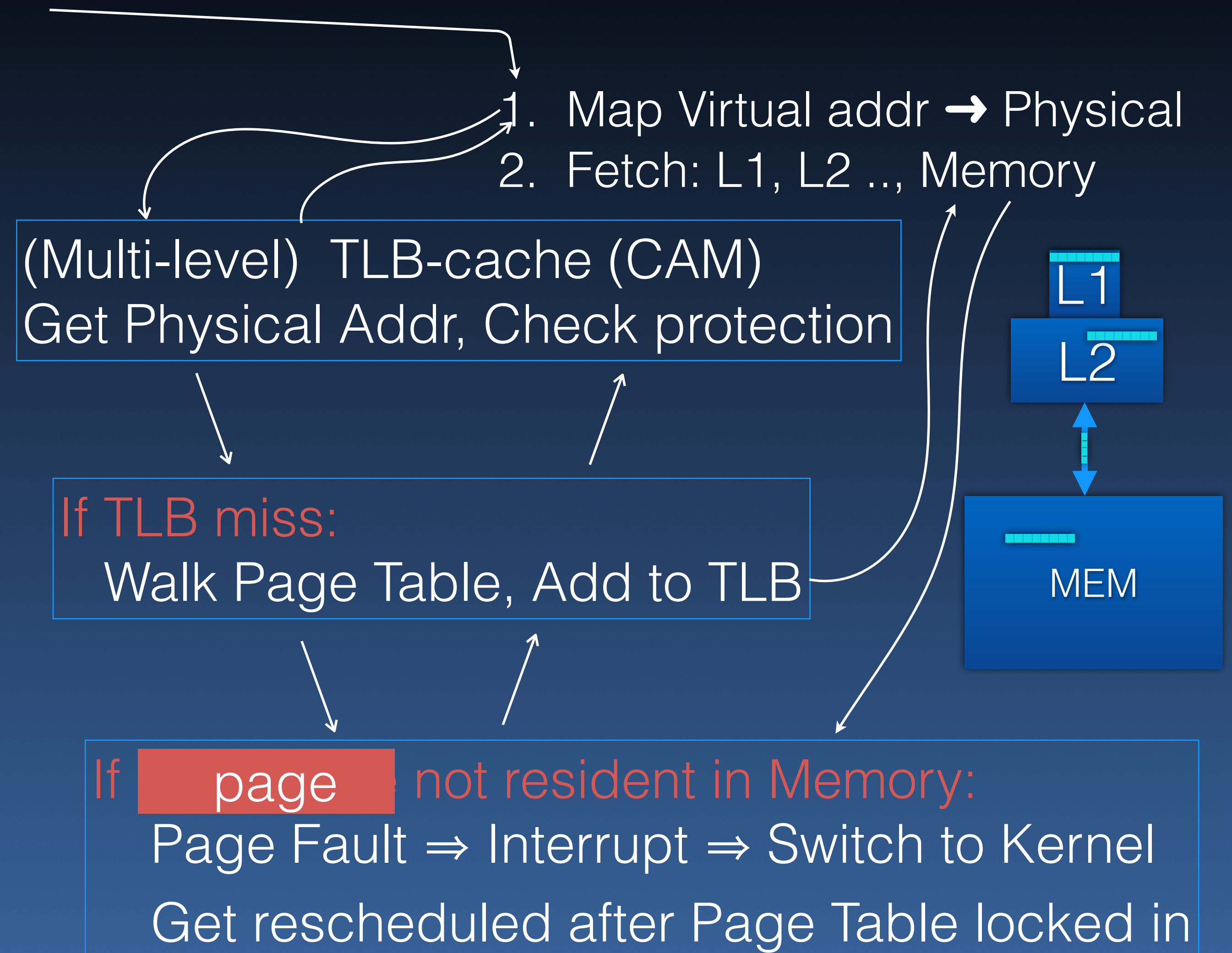
Execution Steps

```
movss (%rdi,%rax,4),%xmm0
```



Execution Steps

```
movss (%rdi,%rax,4),%xmm0
```



It's the Memory Stupid!

- Accessing variables is ^{much} 'slower' than computing
- Memory sizes are often sufficient
 - ➔ Particularly, when distributed among many nodes
 - ▶ Out-of-core computation needed sometimes (we won't focus)
- Caches are small
 - ➔ Not very helpful if traversing large swaths of memory
 - ➔ But, mind the line
 - ▶ And, prefetching is prevalent
 - ➔ Beware of sharing across caches, particularly false-sharing

Cache Example

```
for(int i=0; i<n; i++) {  
    a[i] = b[i] + c[i];  
}  
for(int i=0; i<n; i++) {  
    d[i] = e[i] + f[i];  
}
```

vs

```
for(int i=0; i<n; i++) {  
    a[i] = b[i] + c[i];  
    d[i] = e[i] + f[i];  
}
```

```
for(int i=0; i<N; i++) {  
    d[i] = x;  
}
```

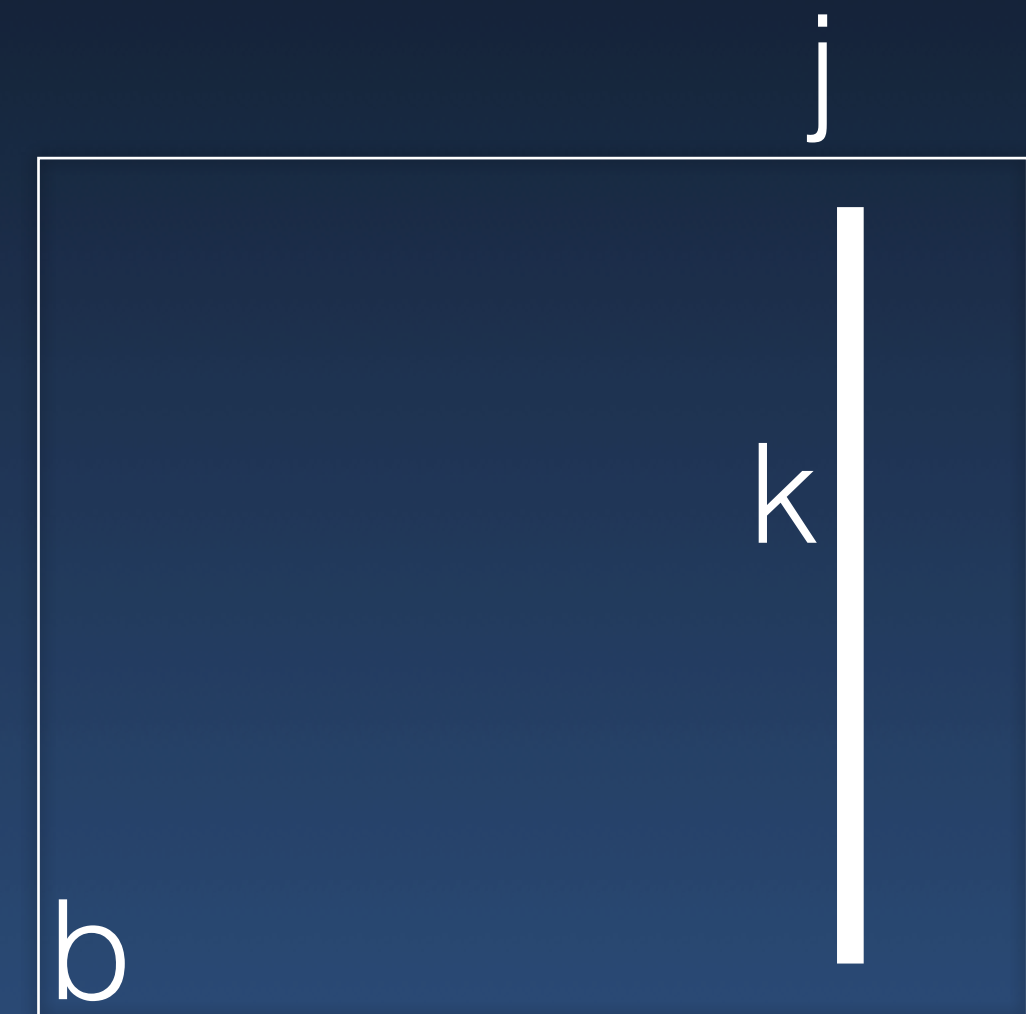
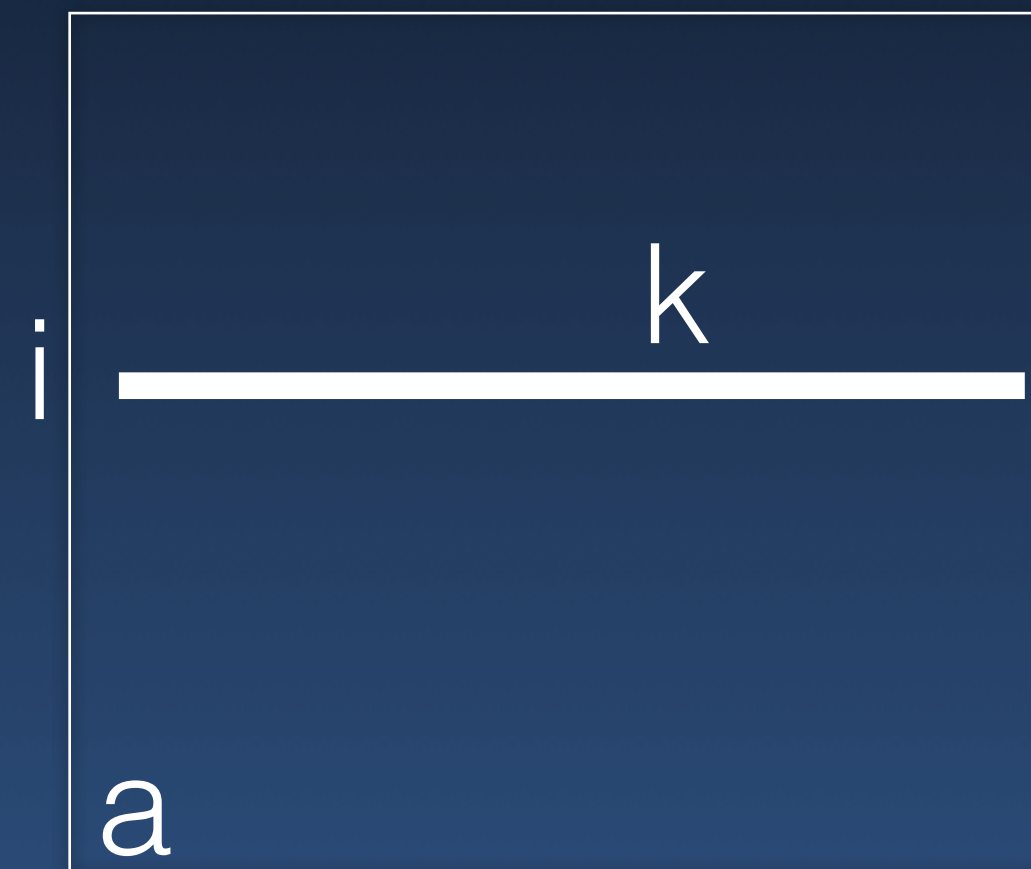
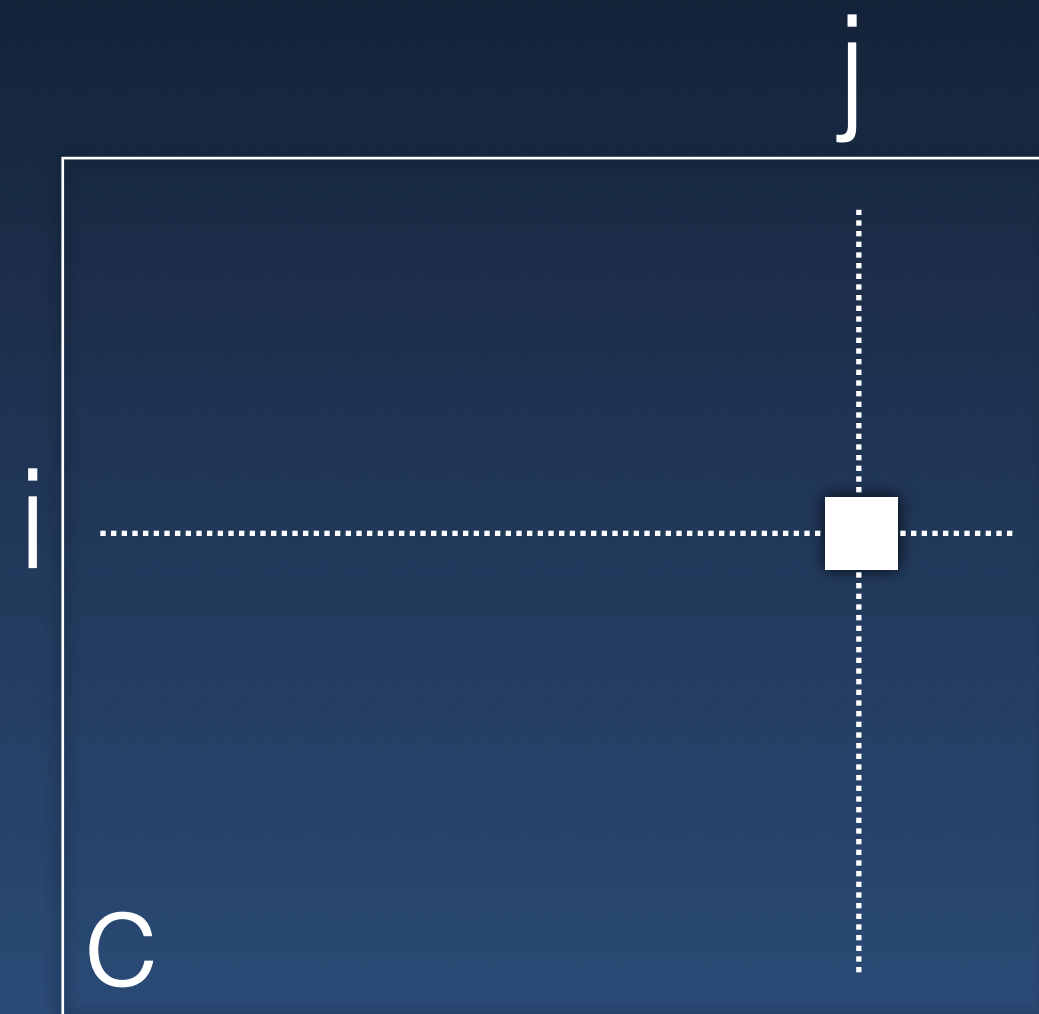
```
for(int i=0; i<N*STEP; i++) {  
    d[(i*STEP)] = x;  
}
```

- Processes, Threads, Context
 - ➔ Address spaces
- Virtual memory
- Caches

Agenda

- Cache behavior, false sharing
- Flavors of parallelism
- Memory bottleneck
- Intro to architecture

Matrix Multiplication

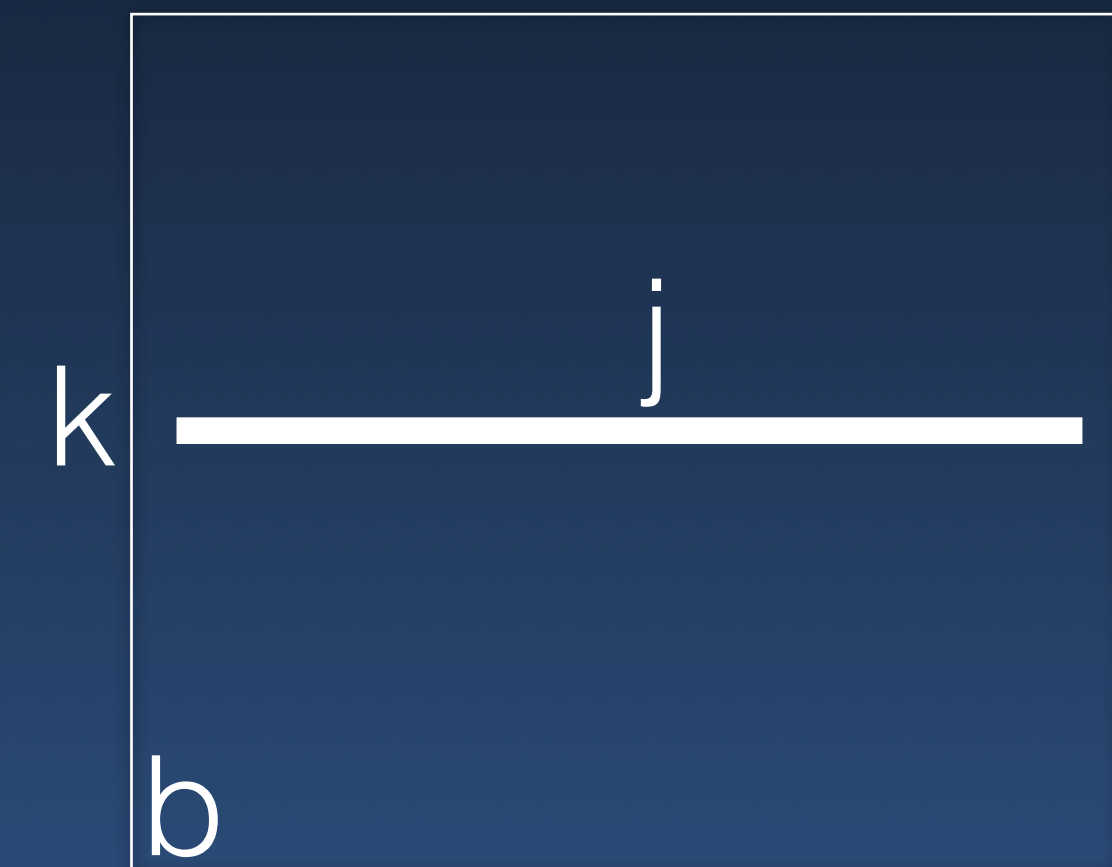
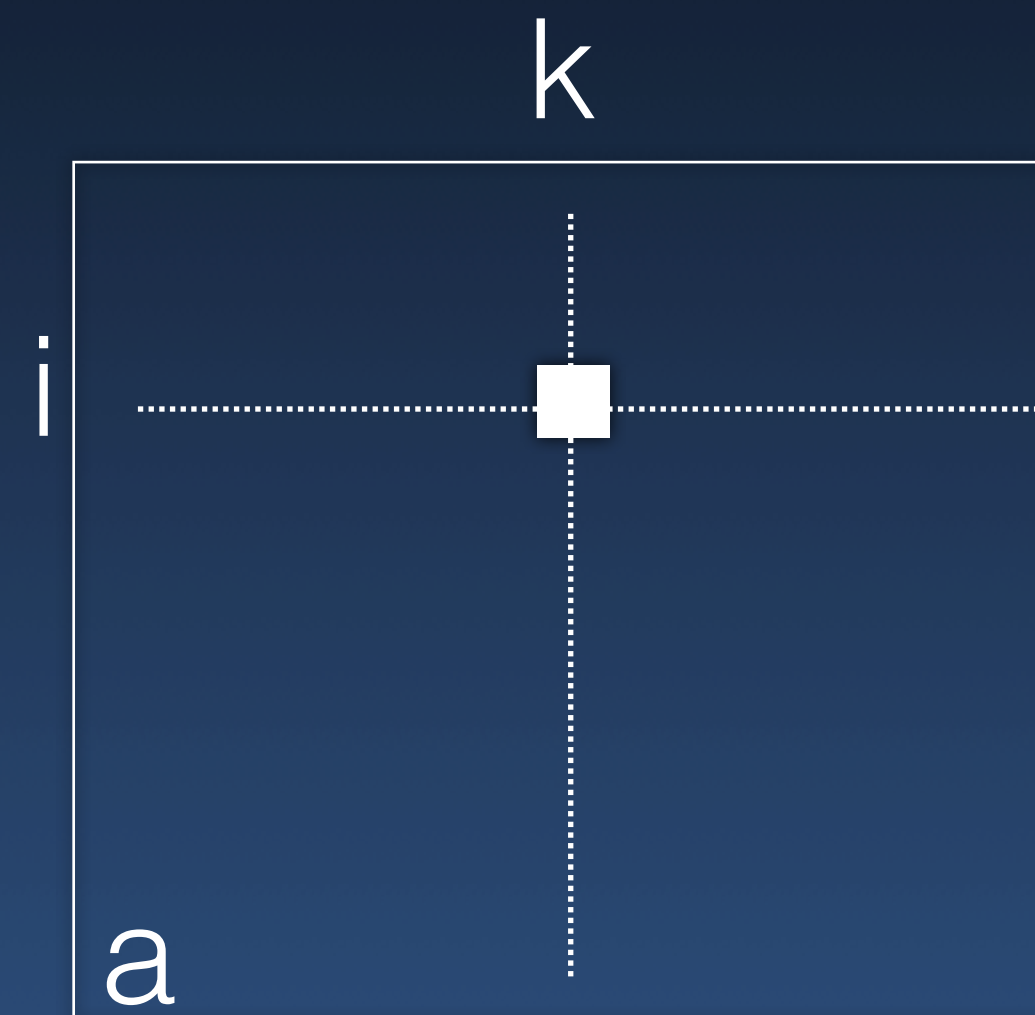


```
for (int i = 0; i < N; i++)  
  for (int j = 0; j < N; j++)  
    for (int k = 0; k < N; k++)  
      c[i][j] += a[i][k]*b[k][j];
```

Matrix Multiplication



```
for (int i = 0; i < N; i++)  
  for (int j = 0; j < N; j++)  
    for (int k = 0; k < N; k++)  
      c[i][j] += a[i][k]*b[k][j];
```

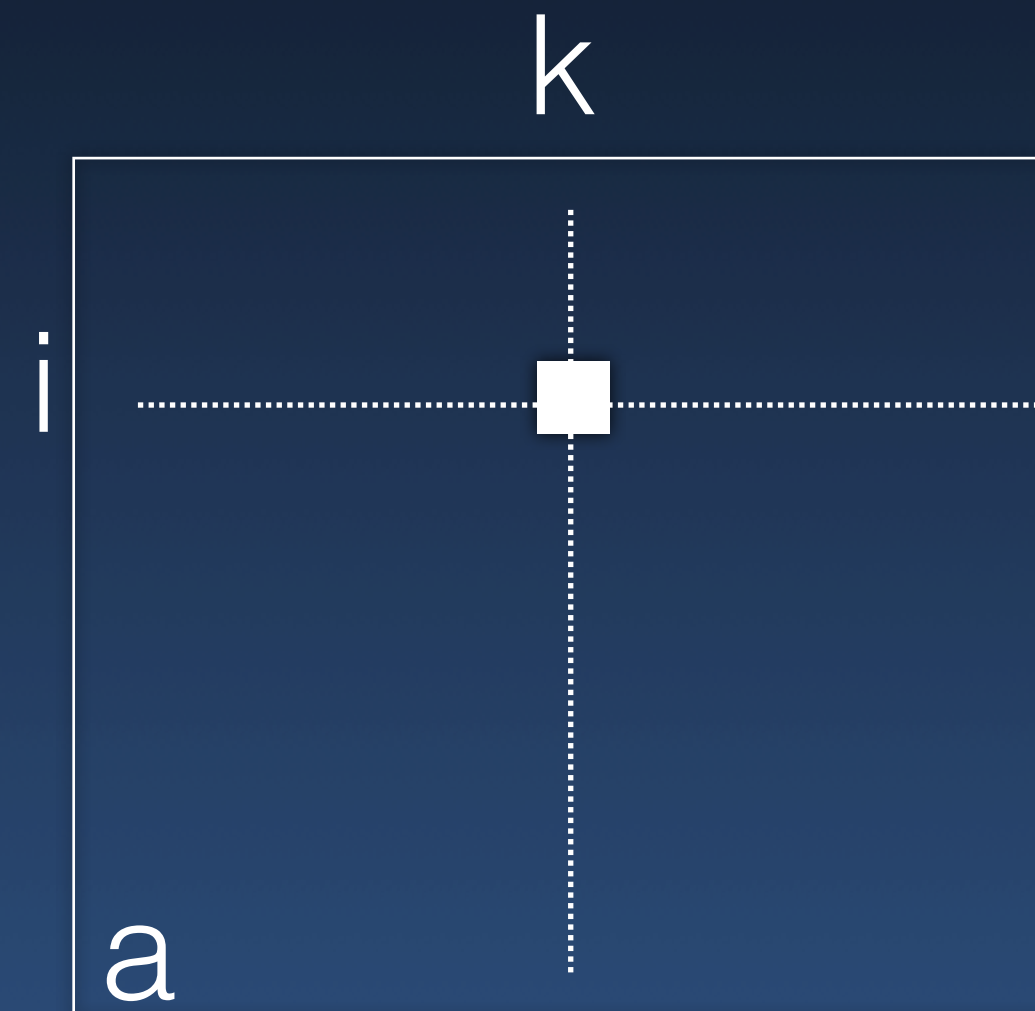


```
for (int i = 0; i < N; i++)  
  for (int k = 0; k < N; k++)  
    for (int j = 0; j < N; j++)  
      c[i][j] += a[i][k]*b[k][j];
```

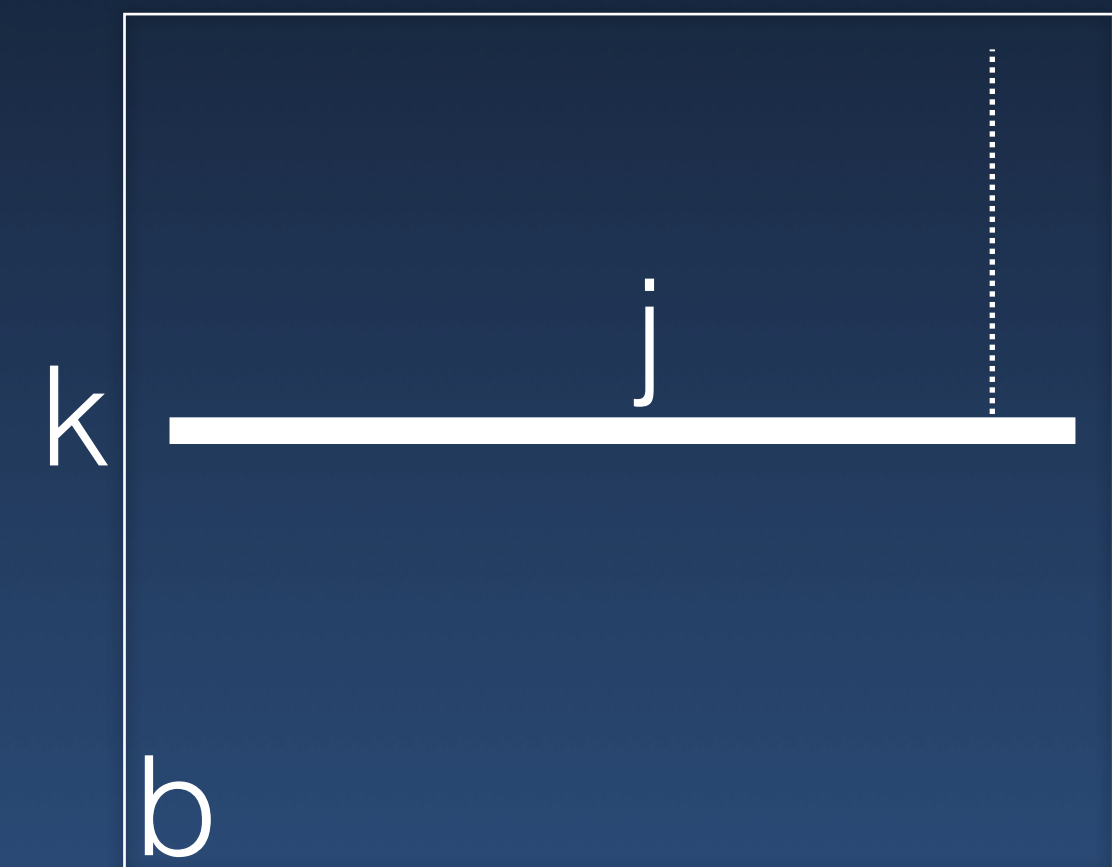
Matrix Multiplication



```
for (int i = 0; i < N; i++)  
  for (int j = 0; j < N; j++)  
    for (int k = 0; k < N; k++)  
      c[i][j] += a[i][k]*b[k][j];
```



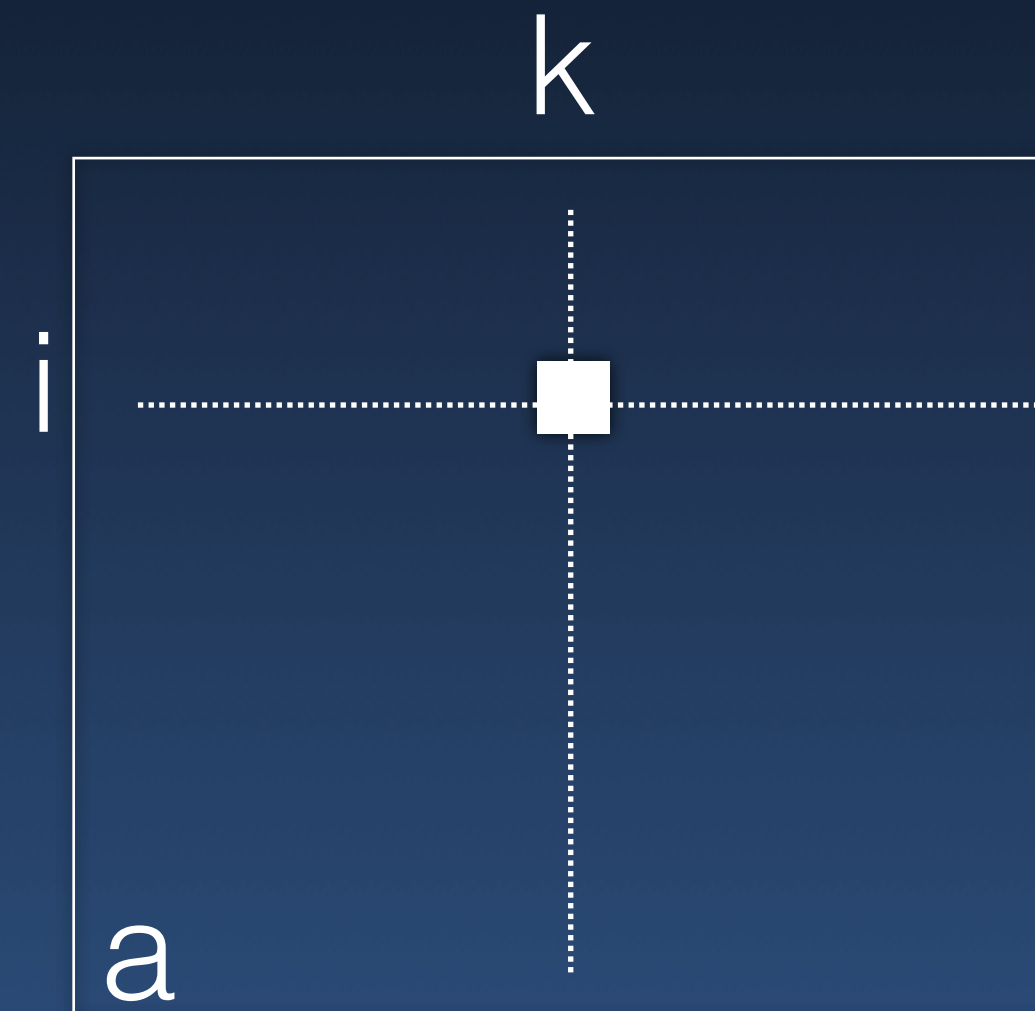
```
for (int i = 0; i < N; i++)  
  for (int k = 0; k < N; k++)  
    for (int j = 0; j < N; j++)  
      c[i][j] += a[i][k]*b[k][j];
```



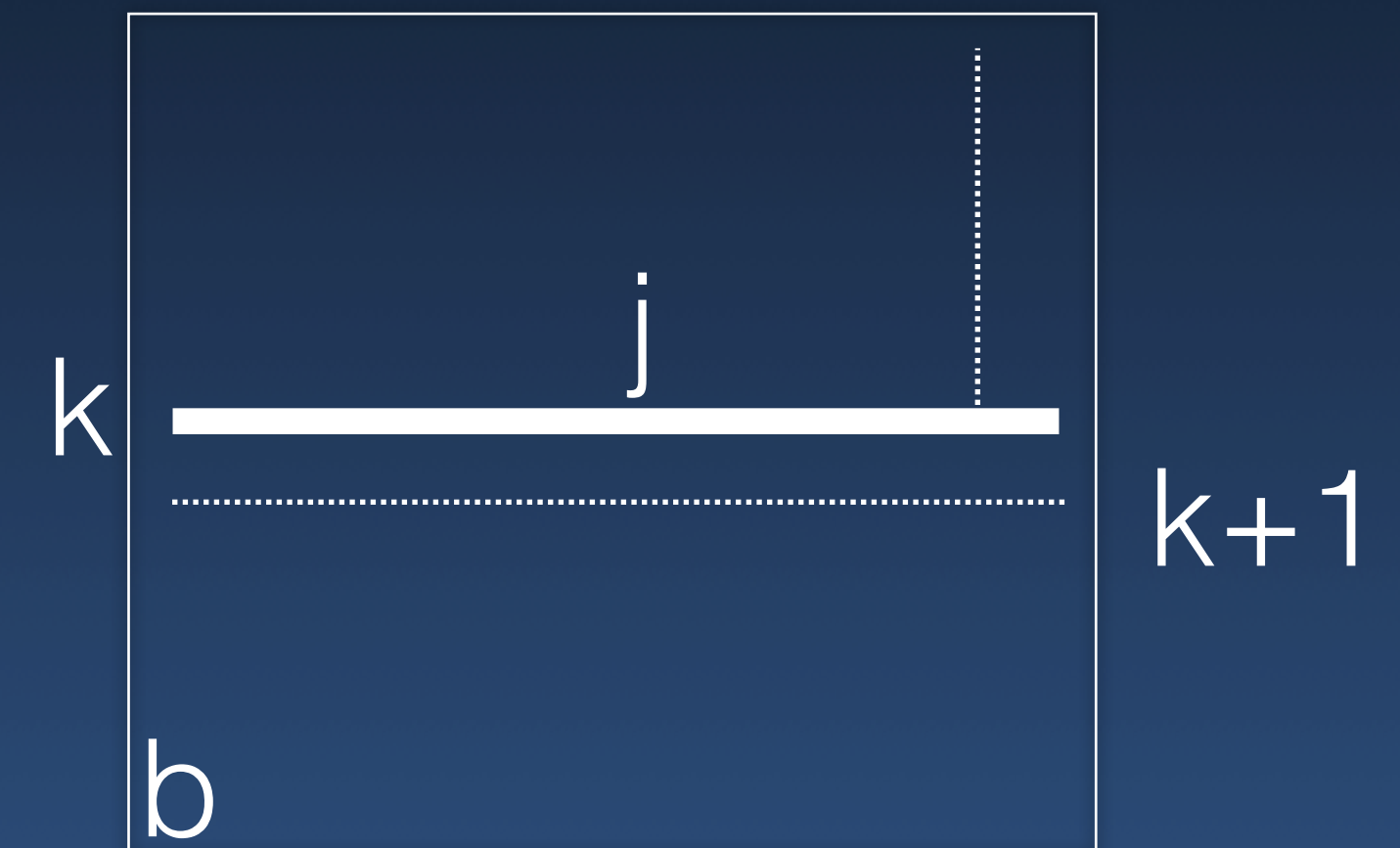
Matrix Multiplication



```
for (int i = 0; i < N; i++)  
  for (int j = 0; j < N; j++)  
    for (int k = 0; k < N; k++)  
      c[i][j] += a[i][k]*b[k][j];
```



```
for (int i = 0; i < N; i++)  
  for (int k = 0; k < N; k++)  
    for (int j = 0; j < N; j++)  
      c[i][j] += a[i][k]*b[k][j];
```



Multiple Caches

- Coherent caches
 - ➔ Writes in on cache are updated in all cache copies
- Caches operate on line granularity
 - ➔ To write one item, may need to update the other items first

