CS257 Advanced Computer Architecture

Topic 6: Algorithmic Optimisations and Code Refactoring

Matthew Leeke and Graham Martin

{matt, grm}@dcs.warwick.ac.uk

Department of Computer Science

University of Warwick

WARWICK

Introduction - Topics

- **Workshop 1:**
  - Introduction to Code Optimisation
  - Algorithmic Optimisations
  - Code Refactoring:
    - Cache Behaviour
    - Loop Optimisation

Introduction - Goals

- At the end of the two workshops you should be able to:
  - Understand performance metrics
  - Understand caching behaviour, and its impact on application performance.
  - Apply code-refactoring optimisations such as loop interchange, loop blocking, loop unrolling, loop fusion and loop fission
  - Apply programming techniques such as SIMD intrinsics (such as SSE) and threading (multi-core) to execute operations in parallel

Introduction - Performance Optimisation

- **Aim of Programming:**
  - *Automate* the completion of some work/task.
  - Complete the work *faster* (than humanly possible).

- **Aim of Optimisation:**
  1. Reduce the *amount of work* required; or
  2. Reduce execution *time per work-unit*.

Introduction - Influencing Factors

- Computers possess multiple sub-systems that can influence performance:
  - Processor speed;
  - Memory bandwidth/latency;
  - Network bandwidth/latency.
- **Best Case Scenario:**
  Every sub-system operates at maximum capacity (100% utilisation).
- **Worst-Case Scenario:**
  One sub-system (termed a *bottleneck*) operates at maximum capacity and others remain idle.

  1. Memory Bound: Performance dominated by loads/stores of data.
  2. Compute Bound: Performance dominated by integer/floating-point operations.

Introduction - FLOPs

- Most common metric for performance of scientific codes is FLOPs (floating-point operations per second).
- **Peak FLOPs:**
  - Maximum achievable FLOPs for a given machine.
  - Theoretical maximum, presuming no cost to memory operations.
- **Program Efficiency:**
  (Achieved FLOPs / Peak FLOPs) * 100

Introduction - Peak FLOPs

- Peak FLOPs is a factor of clock speed (the number of cycles per second) and the max number of floating point operations per cycle.

Peak GFLOPs	=	Clock Speed(GHz)	*	Cores	*	FLOP/Cycle

- Example: Peak Flops calculation for a modern 2.4GHz Quad Core machine capable of SSE4.2 (assuming single precision)

Peak FLOPs =	2.4GHz	*	4	*	4	*	2	=	76.8 GFLOPs

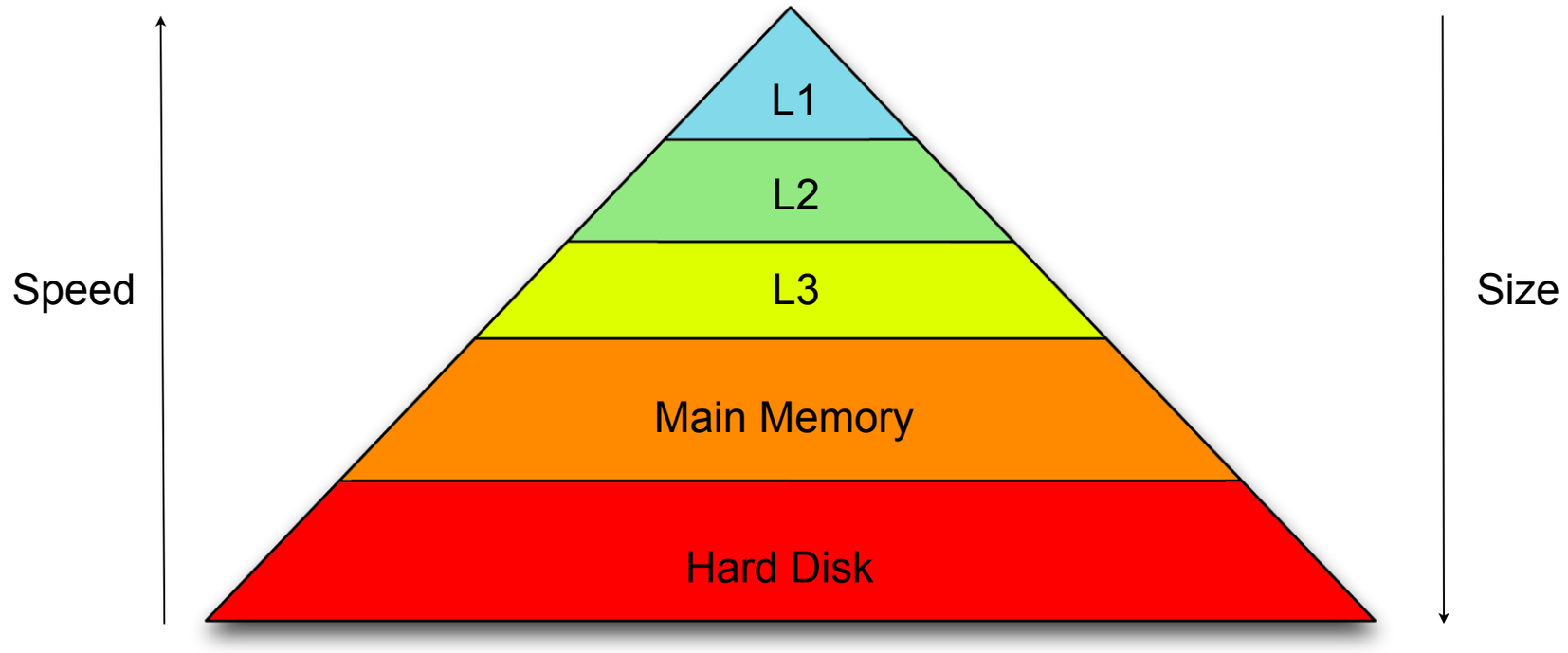
Clock Speed

Cores

SSE SIMD

1 Multiply + 1 Add Operation / Cycle

Introduction - Memory Hierarchy



- Data storage is a balance of size and speed:
  - Memory closer to the CPU (Cache - L1, L2, L3) is typically faster, but has a far smaller capacity.
  - Main memory is further, and slower, but has a far greater capacity
  - Hard disks have the greatest capacity, and do not need power for long term storage, but exhibit the slowest access times.
- **Problem:**
  - We wish to use fastest memory possible for high performance, but typical problems can have large datasets.
- **Solution:**
  - Reuse data already in cache as much as possible.
  - Prefetch data from main memory into cache before it is needed (masking the load time behind other work.)

Introduction - Cache and Memory Layout

- **Cache Line:**
  A sequential “chunk” of data loaded into cache memory. Typically 32, 64 or 128 bytes.
- **Cache Hit:**
  Data requested by execution unit already stored in the cache.
- **Cache Miss:**
  Data is not stored in cache; system must traverse the *memory hierarchy* until the data is found.

  1. *Compulsory* Miss: Data has never been loaded into cache before.
  2. *Capacity* Miss: Data was *evicted* from cache due to limited capacity.
  3. *Conflict* Miss: Expected data location in cache already occupied (more on this later...)

- **Performance Concerns:**
  - Cache misses are *much* more expensive than cache hits.
  - L1/L2/L3/DRAM memories have increasing latencies and decreasing bandwidths.
  - Memory performance largely depends on cache hit:miss *ratio.*

Introduction - Efficient Use of Cache

- **Temporal Locality:**
  Frequently accessed data should remain in cache.
- **Spatial Locality**:
  Data elements stored close together in memory should enter/leave cache at the same time.
  - When a data element is loaded, an entire cache-line is pulled in from memory.
  - *Prefetchers* attempt to predict data that may be needed by a program and pull it into cache in advance to prevent a cache-miss.
    (*e.g.* if accessing a[0]...a[10], it's a good bet a[11] will be accessed too)

Introduction - Speedup

- **Speedup:**
  The ratio of performance increase between two implementations
- **Optimisation speedup:**
  - Unoptimised time / Optimised time
  - E.g. 2x speedup = Halved time

Introduction - Optimisation Approaches

- Optimisation techniques can fall under one of three categories:
  - **Algorithmic**:
    Implementations with lower *algorithmic complexity*.
  - **Code Refactoring:**
    Addressing inefficiencies in machine resource utilisation.
    (*e.g.* instructions per cycle, cache hit:miss ratio)
  - **Parallelisation**:
    Executing multiple tasks (or sub-tasks) in *parallel.*
    (*e.g.* vectorisation, multi-threading, clusters/supercomputers)

Optimisation - Algorithmic

- Algorithmic optimisations reduce the number of operations:
  - Merge Sort: O(n log n)
  - Bubble Sort O(n^2)
- Floating-point numbers are imprecise - e.g. 1.0/3.0 = 0.33333333...,
- Operations are *not* necessarily commutative, even if they "should" be:
  - (1.01 + 2.03) / 3.0 = 1.0133333333333341919
  - (1.01/3.0) + (2.03/3.0) = 1.0133333333333319715
- If consistency or accuracy within a strict tolerance is required, maintaining the order of operations is important.

Optimisation - Code Refactoring

- **Aims:**
  - Convince the compiler to apply optimisations, and for on-core parallelism (Instruction-Level Parallelism, Pipelining etc)
  - Promote efficient cache use via good spatial and temporal locality.
  - Minimise overheads and simplify/un-obfuscate code (for the compiler!)

Optimisation - Compiler Optimisations

- Compilers designed to apply many optimisations, controlled by flags:
  - Automatic parallelisation, vectorisation, fast (approximate) math, function inlining...
  - Optimisation levels (e.g. -O0, -O1, -O2); enable *lots* of optimisations.
  - Too many to go through here -- read the man page!

- **Pragma:**
  Developer hint to the compiler.
  #pragma <optimisation> <options>
  #pragma unroll(4)
  #pragma ivdep

Optimisation - Loop Optimisations

- The majority of an application's runtime tends to be (though not always!) in executing loops, so many loop-focused optimisations:
  - Loop Interchange
  - Loop Blocking
  - Loop Fusion/Fission
  - Loop Unrolling
  - Loop Pipelining

Optimisation - Inter-Loop Dependencies

- **Loop Dependency:**
  - One or more loop iterations are dependent upon the completion of another loop iteration.
  - Not an optimisation, but may inhibit the application of optimisations.
- **Removing Inter-loop Dependencies:**
  Do not necessarily inhibit performance, but they can interfere with a variety of optimisations due to the need to preserve order (especially parallelisation)

```
for(i=0;i<n;i++)
{
    a[i] = i + a[0];
}
```

All loops depend on loop 0

```
for(i=1;i<n;i++)
{
    a[i] = a[i] + a[i-1];
}
```

An iteration, i, depends upon the completion of i-1, creating a chain dependency

```
for(i=1;i<n;i++)
{
    a[i] = a[i] + a[i+2];
}
```

Iteration i depends upon iteration i+2 having not yet run (potential parallelism issue).

Optimisation - Pointer Aliasing

- **Pointer Aliasing**:
  Multiple pointers point to the same or overlapping memory location.
  - Compiler may assume pointers are aliased, even if they are not, preventing optimisation.
  - Aliased pointers introduce hidden loop dependencies.
- **Aiding the Compiler:**
  - If not aliased, use the *restrict* keyword in declarations: int * restrict a, double * restrict b ....
  - Must not use restrict keyword if pointer is actually aliased!

```
for(i=1;i<n;i++)
{
    a[i] = a[i] + b[i-1];
}
```

This loop can appear independent...

```
b=a;
for(i=1;i<n;i++)
{
    a[i] = a[i] + b[i-1];
}
```

However with more information on b, we can perform some substitution.

```
b=a;
for(i=1;i<n;i++)
{
    a[i] = a[i] + a[i-1];
}
```

And we see the hidden inter-loop dependency on iteration i-1.

Optimisation - Loop Peeling

- **Loop Peeling**:
  One or more iterations from a compute loop are extracted and moved outside of the body of the loop, with the loop range being reduced to reflect this.
- Not strictly an optimisation, but may be a pre-requisite for others.

```
unpeeled(){
    ...
    for(i=0;i<n;i++)
    {
        a[i] = a[i]+ a[0];
    }
    ...
}
```

Dependancy on a[0] means any iteration where i>0 is dependant

```
peeled(){
    ...
    a[0] = a[0] + a[0];
    for(i=1;i<n;i++)
    {
        a[i] = a[i] + a[0];
    }
    ...
}
```

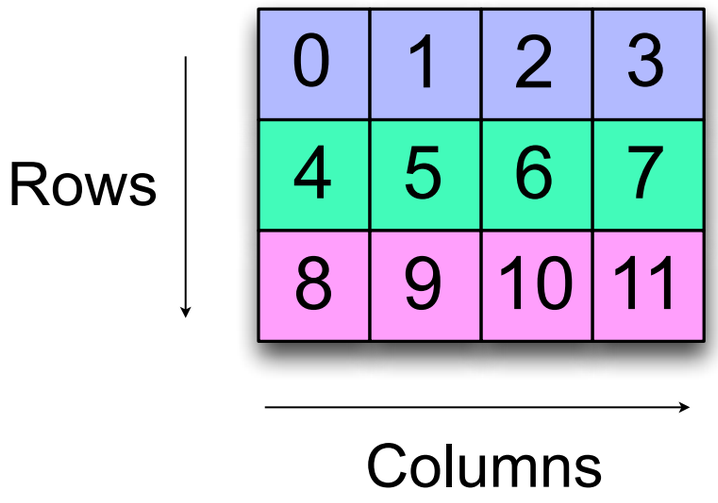
Iteration 0 is peeled from the loop, removing the loop dependancy.

Optimisation - Loop Interchange

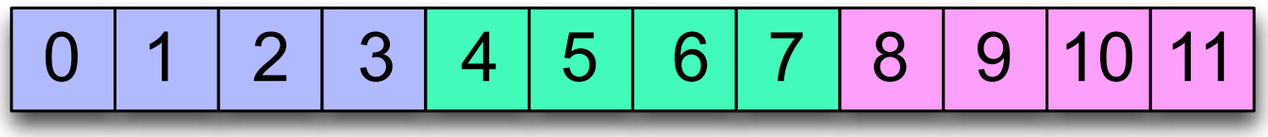
- **Loop Interchange:**
  Modify the order of memory accesses by switching the order of loops in the code.
- **Requirements:**
  - A nested loop (*e.g.* for i, for j)
  - Inter-loop dependencies must not be violated.
- **When to Use:**
  - Loop order is such that memory accesses are suboptimal.
  - Data is stored in a contiguous block of memory
    (*i.e.* a cache-line encompasses multiple elements of the array)

Optimisation - Loop Interchange - Memory Layout

C 2D-Matrix Layout - A[row][column]
(Row Major)



Sequential Memory



Data Index = column + (row * no. of columns)

Traversing columns before rows - Sequential memory access	A[0][0] = 0 + (0*4) = 0 A[0][1] = 1 + (0*4) = 1 A[0][2] = 2 + (0*4) = 2	A[0][3] = 3 + (0*4) = 3 A[1][0] = 0 + (1*4) = 4 A[1][1] = 1 + (1*4) = 5
Traversing rows before columns - Memory access in strides of 4 (stride equivalent to size of a row.)	A[0][0] = 0 + (0*4) = 0 A[1][0] = 0 + (1*4) = 4 A[2][0] = 0 + (2*4) = 8	A[0][1] = 1 + (0*4) = 1 A[1][1] = 1 + (1*4) = 5 A[2][1] = 1 + (2*4) = 9

Optimisation - Loop Interchange - Memory Layout

- **Performance Impact:**
  - Dataset fits into cache- little impact
  - Dataset does not fit into cache - significant impact

- With non-sequential access we access memory with a *stride* of the row length.
- Since data is loaded in cache-lines, for a sufficiently large number of rows there is a likelihood that the first cache-line is evicted before it is reused.

Optimisation - Loop Interchange - Row Traversal

- Let us assume a machine that possesses:
  - A single-level cache that can hold 8 “data elements”
  - A cache-line that consists of 4 “data elements”
  - A main memory that can hold 20 “data elements”
- We wish to calculate the cumulative sum of a 2D array, consisting of 4 rows and 5 columns in a C program

```
Row-First Traversal() {
    ...
    for(j=0;j<5;j++) {
        for(i=0;i<4;i++) {
            sum += a[i][j]
        }
    }
}
```

Optimisation - Loop Interchange - Row Traversal

Operation - Step 1:

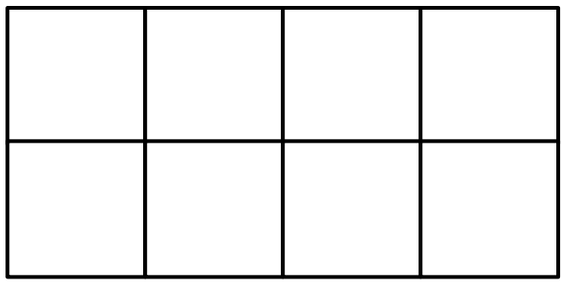
sum += a[0][0]

1. Data Index 0 is requested.
2. Data Index 0 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2



Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 1:

sum += a[0][0]

1. Data Index 0 is requested.
2. Data Index 0 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

0	1	2	3

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 2:

sum += a[1][0]

1. Data Index 4 is requested.
2. Data Index 4 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1	0	1	2	3
Cache Line 2				

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 2:

sum += a[1][0]

1. Data Index 4 is requested.
2. Data Index 4 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

0	1	2	3
4	5	6	7

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 3:

sum += a[2][0]

1. Data Index 8 is requested.
2. Data Index 8 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

0	1	2	3
4	5	6	7

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 3:

sum += a[2][0]

1. Data Index 8 is requested.
2. Data Index 8 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

4	5	6	7

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 3:

sum += a[2][0]

1. Data Index 8 is requested.
2. Data Index 8 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

8	9	10	11
4	5	6	7

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 4:

sum += a[3][0]

1. Data Index 12 is requested.
2. Data Index 12 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

8	9	10	11
4	5	6	7

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 4:

sum += a[3][0]

1. Data Index 12 is requested.
2. Data Index 12 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

8	9	10	11

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 4:

sum += a[3][0]

1. Data Index 12 is requested.
2. Data Index 12 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1	8	9	10	11
Cache Line 2	12	13	14	15

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 5:

sum += a[4][0]

1. Data Index 16 is requested.
2. Data Index 16 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

8	9	10	11
12	13	14	15

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 5:

sum += a[4][0]

1. Data Index 16 is requested.
2. Data Index 16 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1				
Cache Line 2	12	13	14	15

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 5:

sum += a[4][0]

1. Data Index 16 is requested.
2. Data Index 16 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

16	17	18	19
12	13	14	15

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 6:

sum += a[4][1]

1. Data Index 1 is requested.
2. Data Index 1 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

16	17	18	19
12	13	14	15

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 6:

sum += a[4][1]

1. Data Index 1 is requested.
2. Data Index 1 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

16	17	18	19

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 6:

sum += a[4][1]

1. Data Index 1 is requested.
2. Data Index 1 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

16	17	18	19
0	1	2	3

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 7:

sum += a[4][2]

1. Data Index 5 is requested.
2. Data Index 5 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

16	17	18	19
0	1	2	3

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 7:

sum += a[4][2]

1. Data Index 5 is requested.
2. Data Index 5 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

0	1	2	3

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation - Loop Interchange - Row Traversal

Operation - Step 7:

sum += a[4][2]

1. Data Index 5 is requested.
2. Data Index 5 is moved from main memory into cache, along with the cache-line
3. Data in cache used for compute

Cache

Cache Line 1

Cache Line 2

4	5	6	7
0	1	2	3

Main Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Optimisation- Loop Interchange - Column Traversal

- **Problem:**
  - Constant eviction of cache line without reuse.
  - Total of 20 cache-line reads and writes for only 20 data elements
- **Solution:**
  - Loop Interchange to traverse columns instead of rows
  - Improved spatial locality to fully utilise cache-line
  - Total of 5 cache-line reads and writes for 20 data elements

```
Column-First Traversal() {
    ...
    for(i=0;j<4;i++) {
        for(j=0;j<5;j++) {
            sum += a[i][j]
        }
    }
}
```

Optimisation - Loop Blocking

- **Loop Blocking/Tiling:**
  Rearrange loops to improve cache-reuse of a *block* or *tile* of memory.
- **Requirements:**
  - Nested loops.
- **When to Use:**
  - Poor temporal/spatial cache locality (between loop iterations).
  - Largely memory-bound.

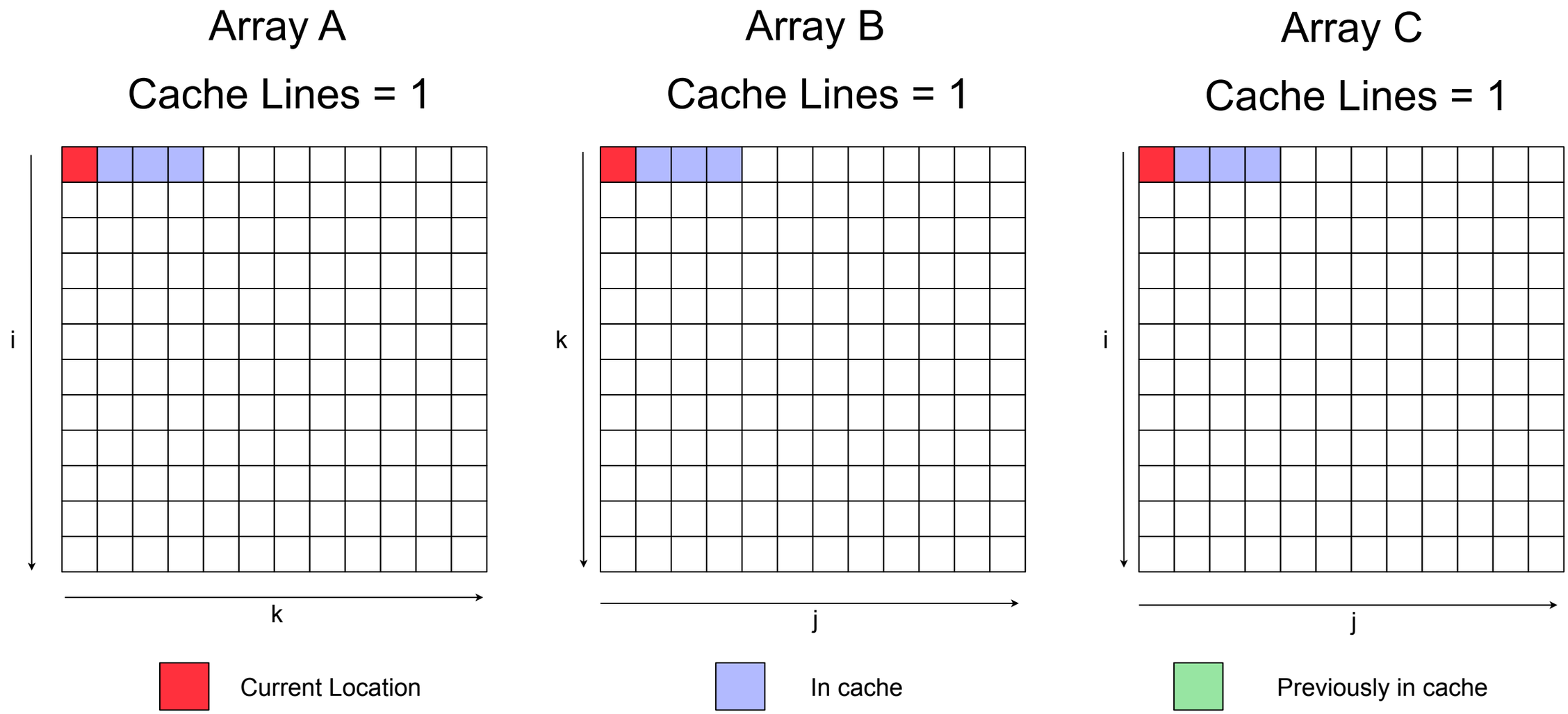
Optimisation - Loop Blocking - Matrix Multiply

- Consider the problem of Matrix-Multiply
  - The memory access pattern for C and A is sequential, however for B it is non-optimal.
  - If we apply loop interchange to loops j and k we can improve the spatial locality of B, but harm the temporal locality of C. In addition the temporal locality of B remains poor, looping across the entire array before revisiting the first location,

```
matrix_multiply(a,b,c,n){
    for(i=0;i<n;i++){
        for(j=0;j<n;j++){
            for(k=0;k<n;k++){
                c[i,j] = c[i,j] + (a[i,k] * b[k,j])
            }
        }
    }
}
```

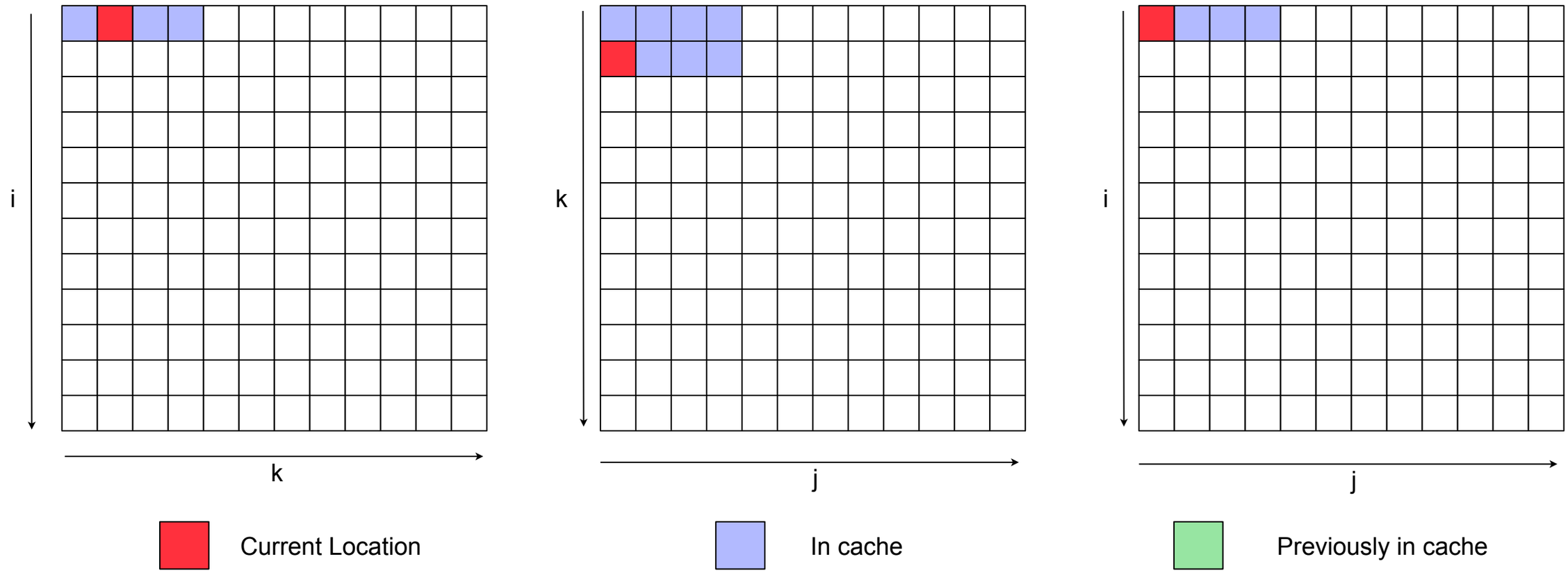
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 9 cache lines at a time, length 4 data elements



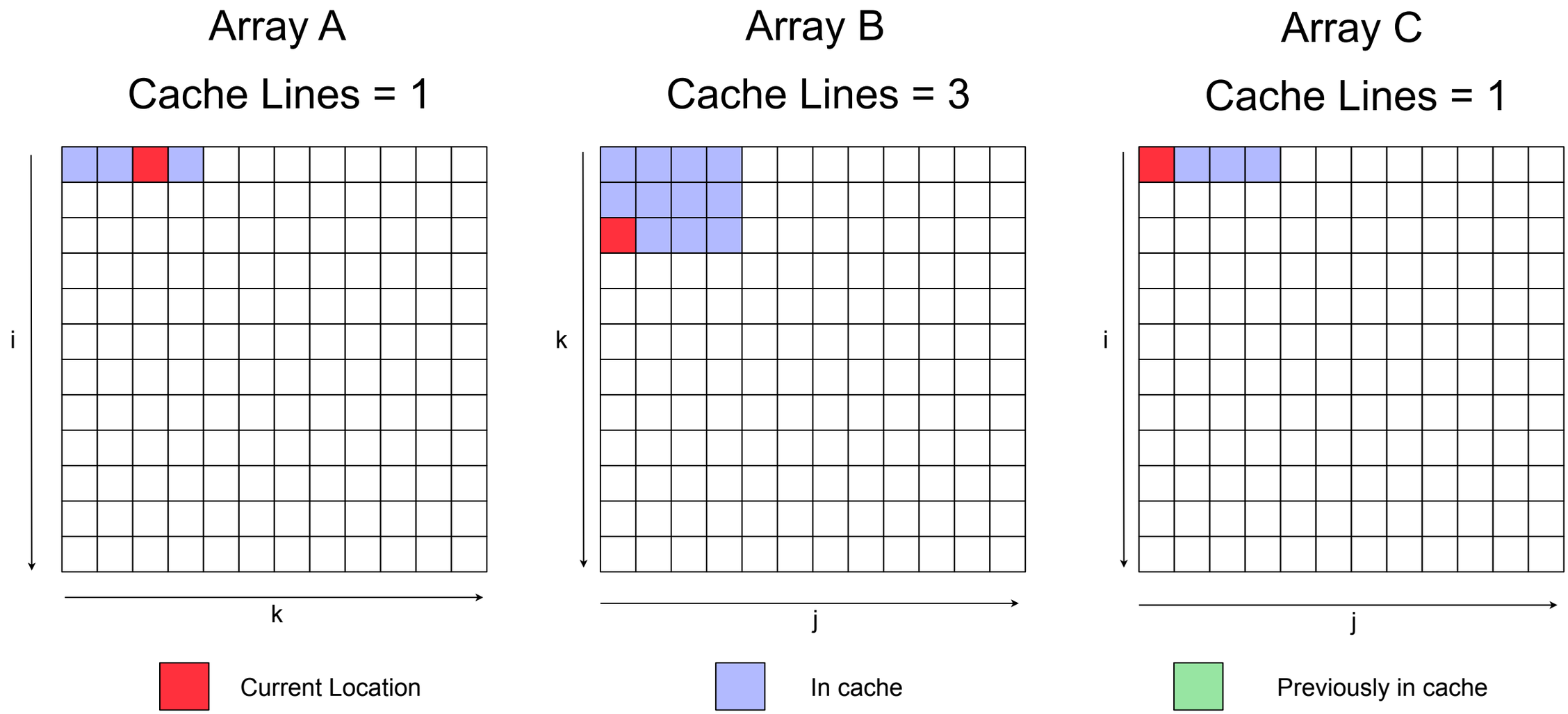
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 9 cache lines at a time, length 4 data elements



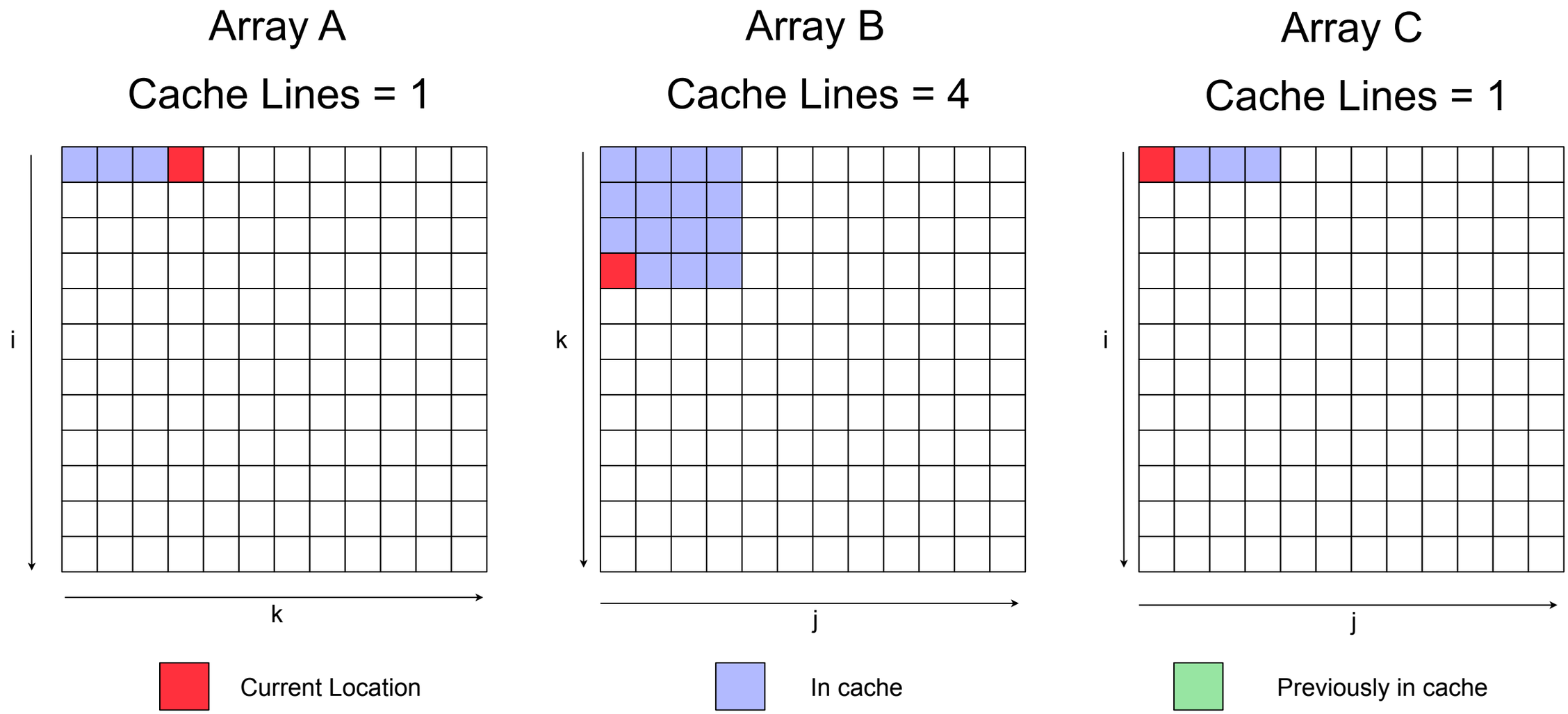
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 9 cache lines at a time, length 4 data elements



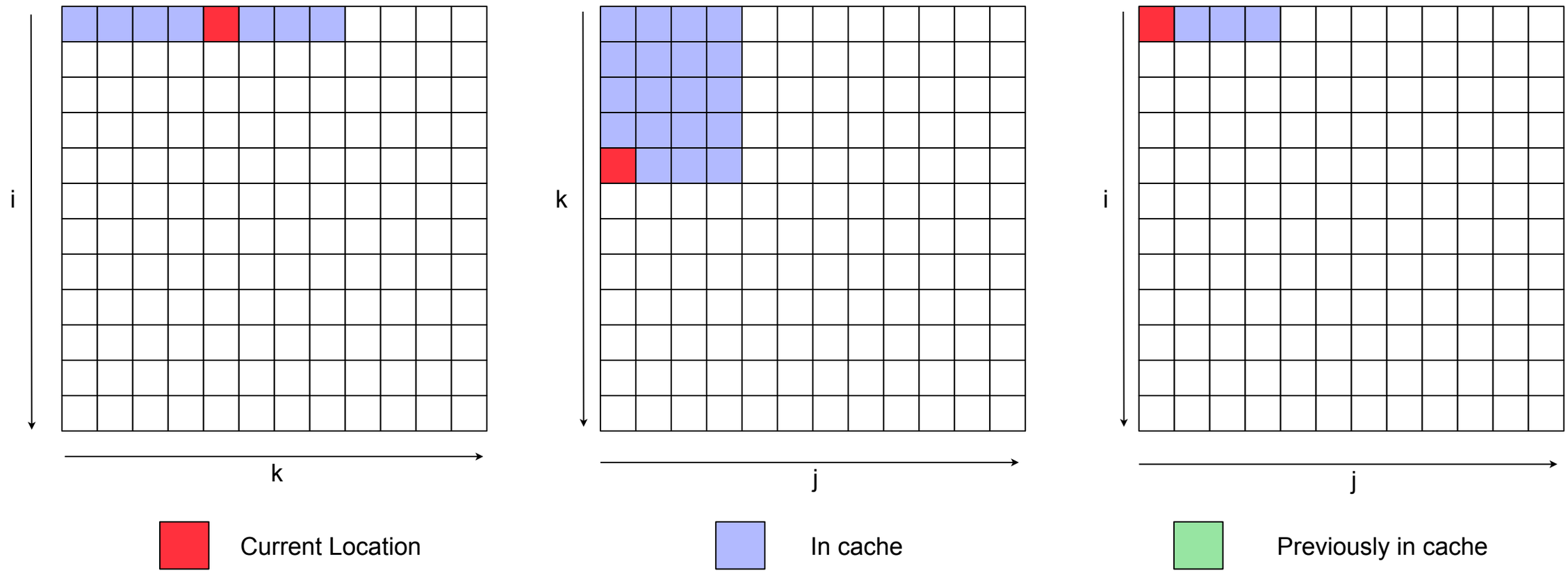
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 9 cache lines at a time, length 4 data elements



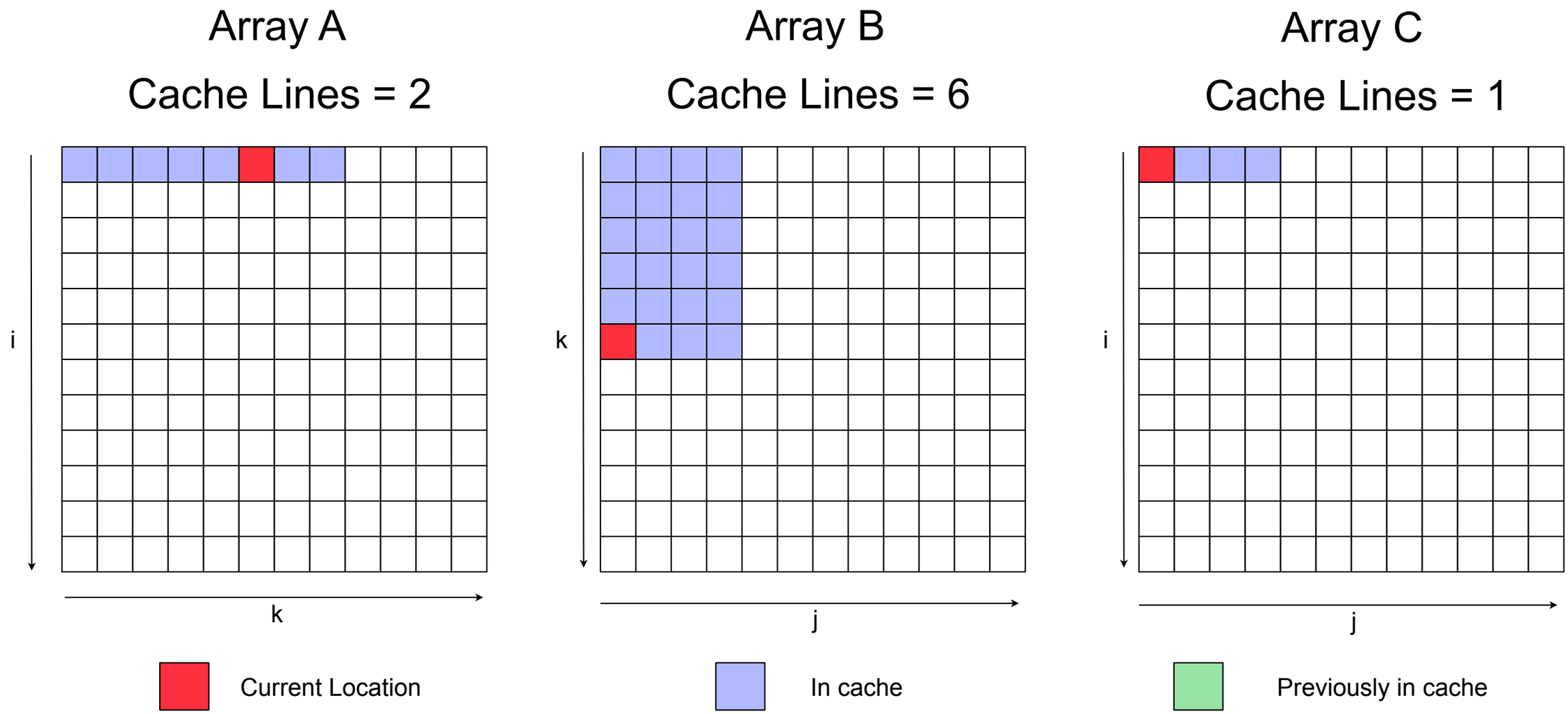
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 9 cache lines at a time, length 4 data elements



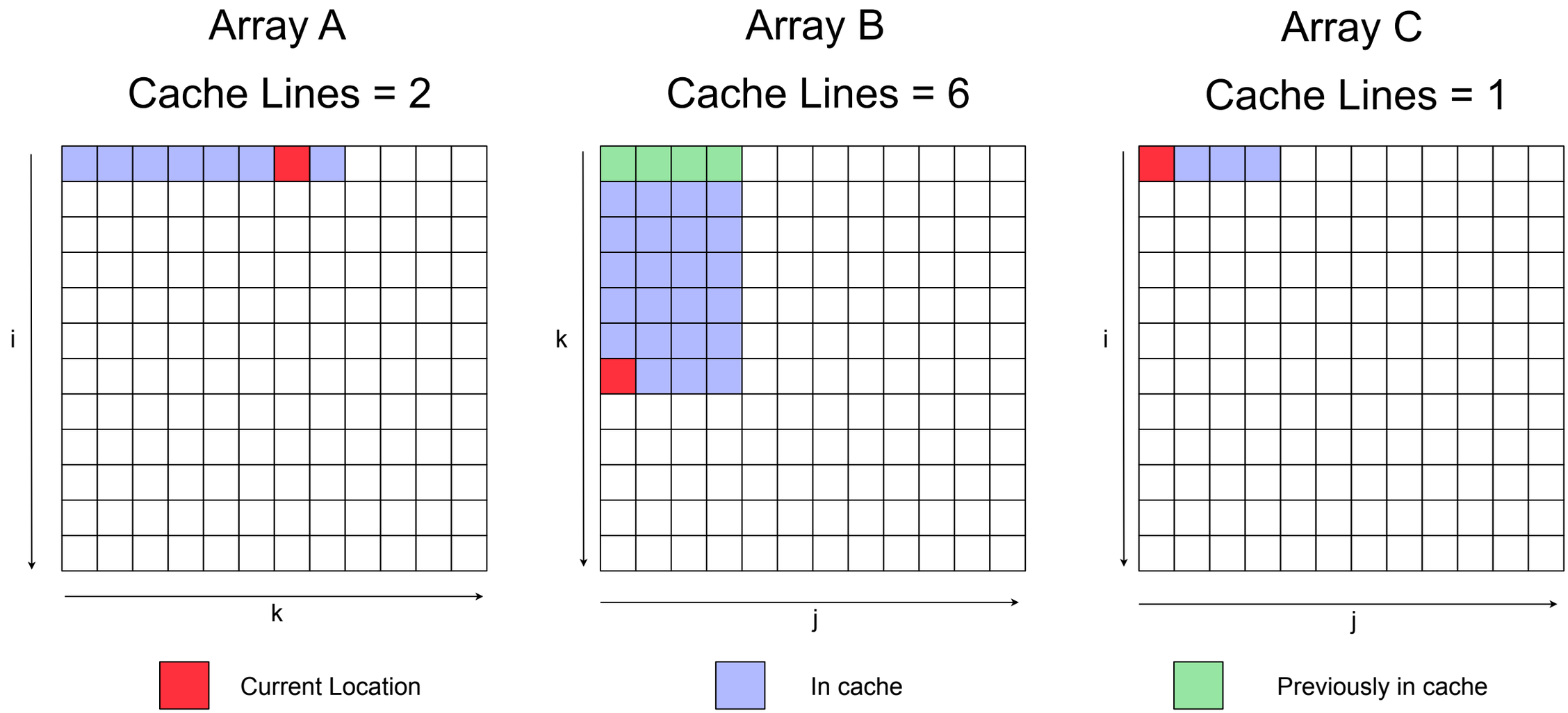
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 9 cache lines at a time, length 4 data elements



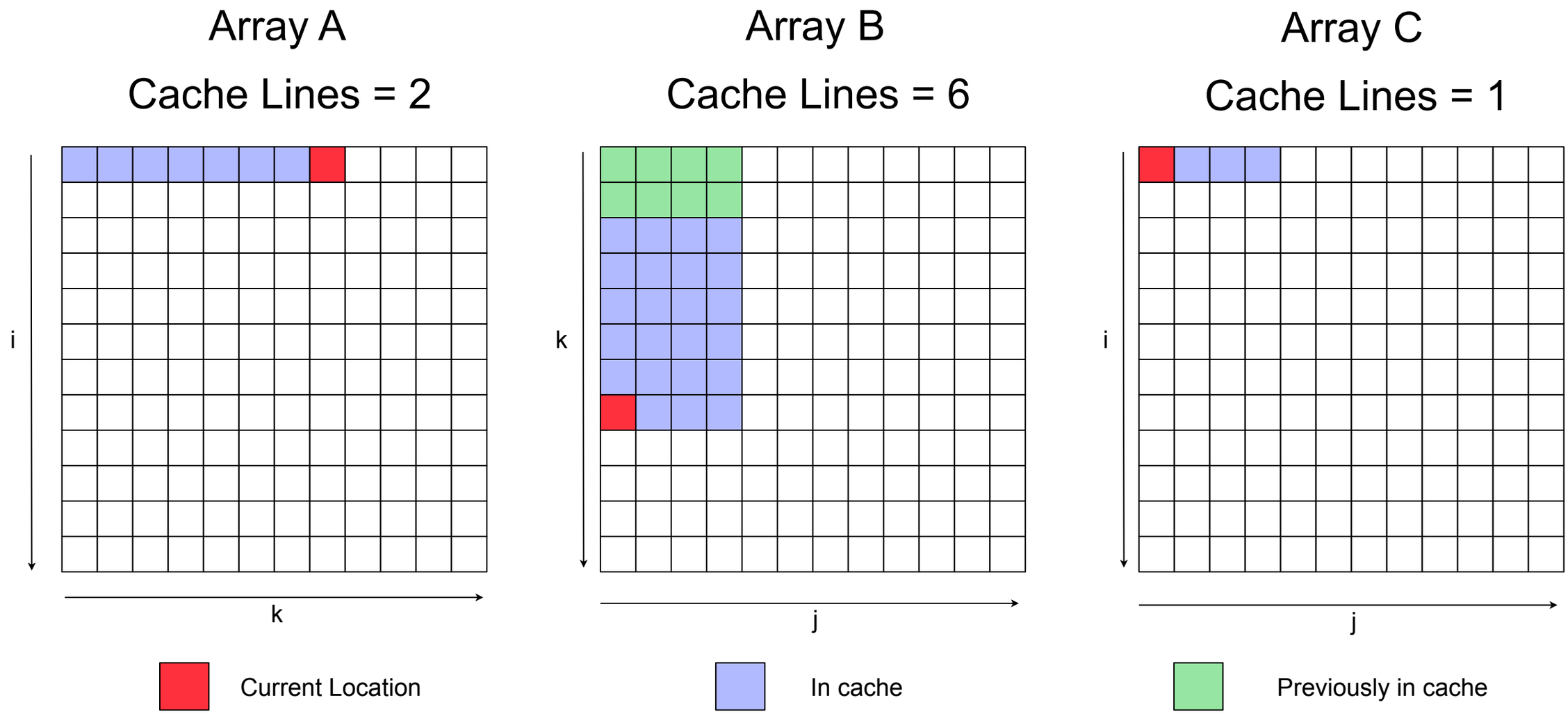
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 9 cache lines at a time, length 4 data elements



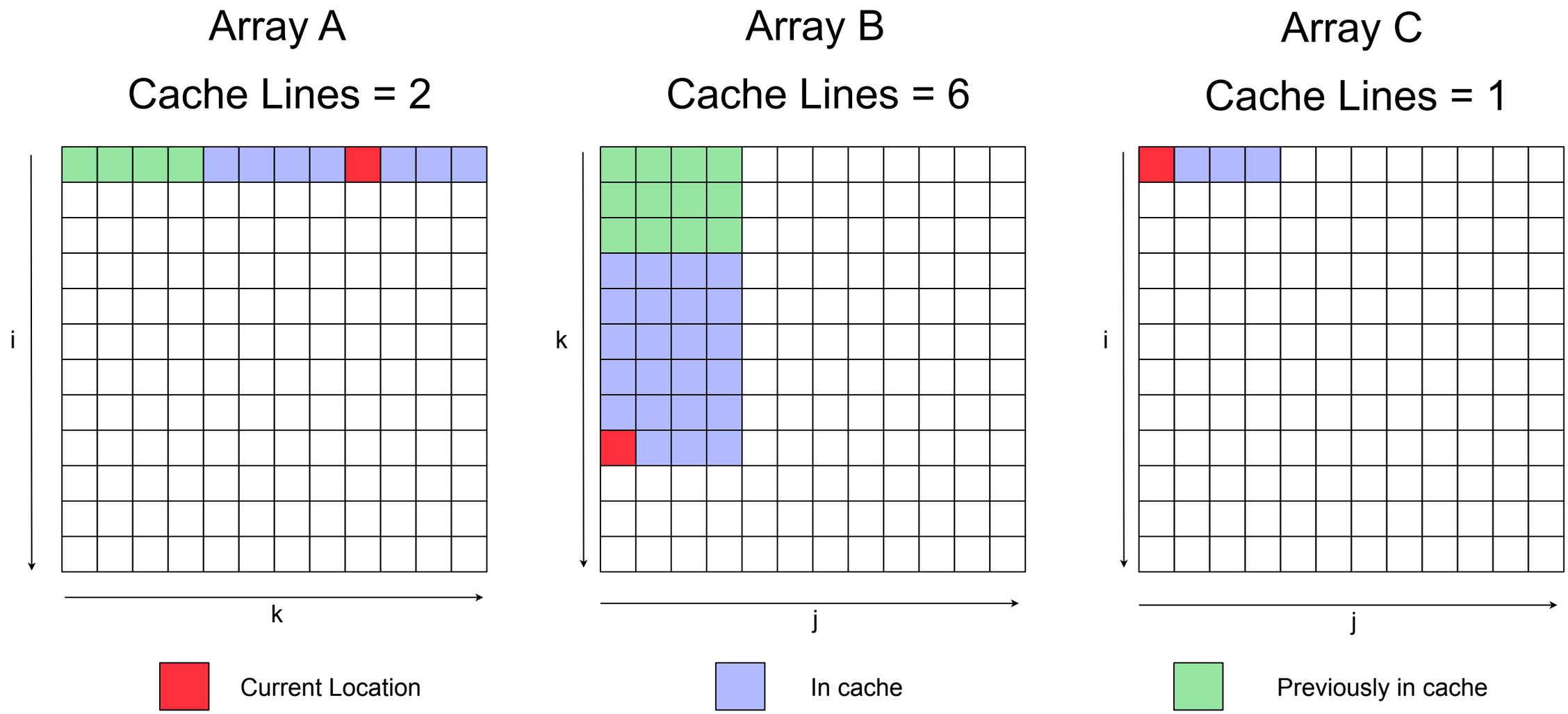
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 9 cache lines at a time, length 4 data elements



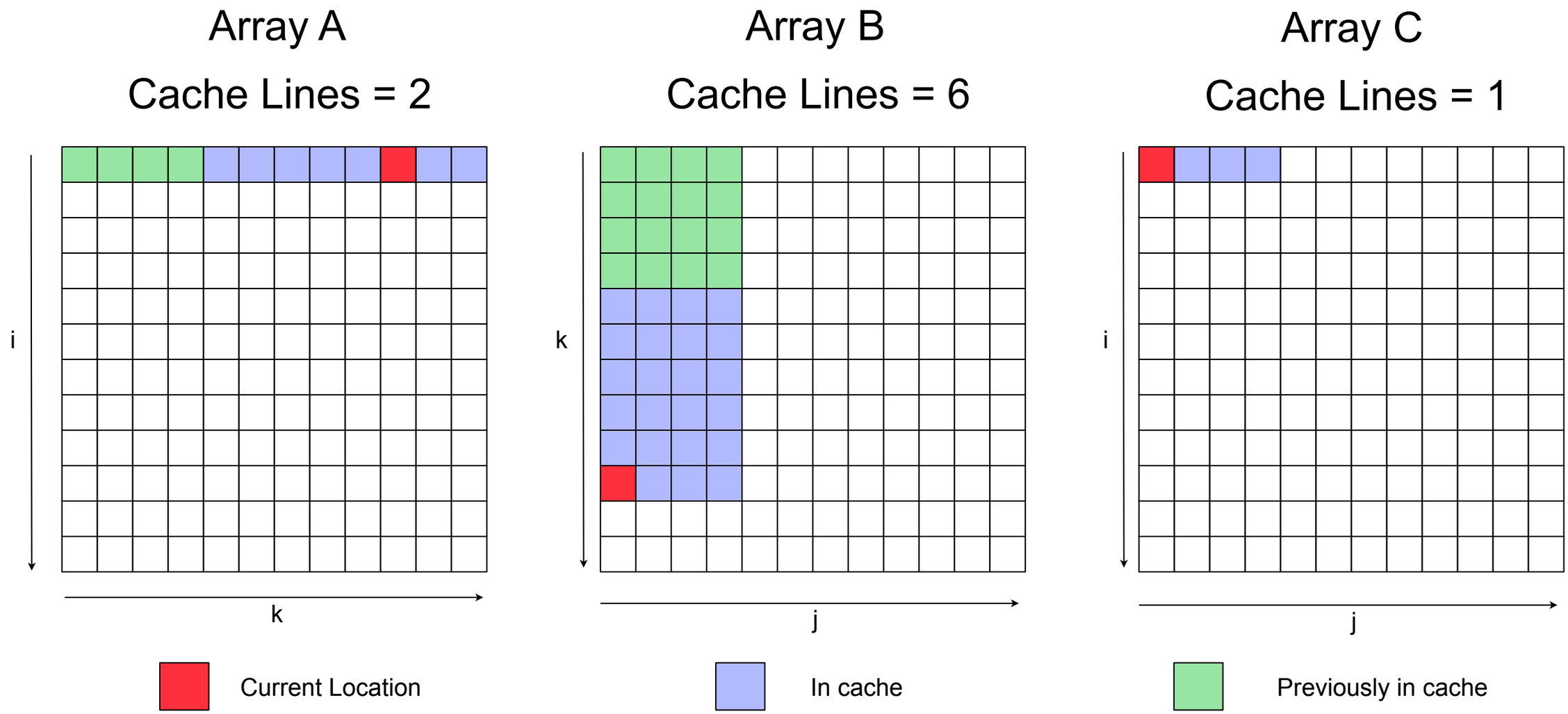
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 9 cache lines at a time, length 4 data elements



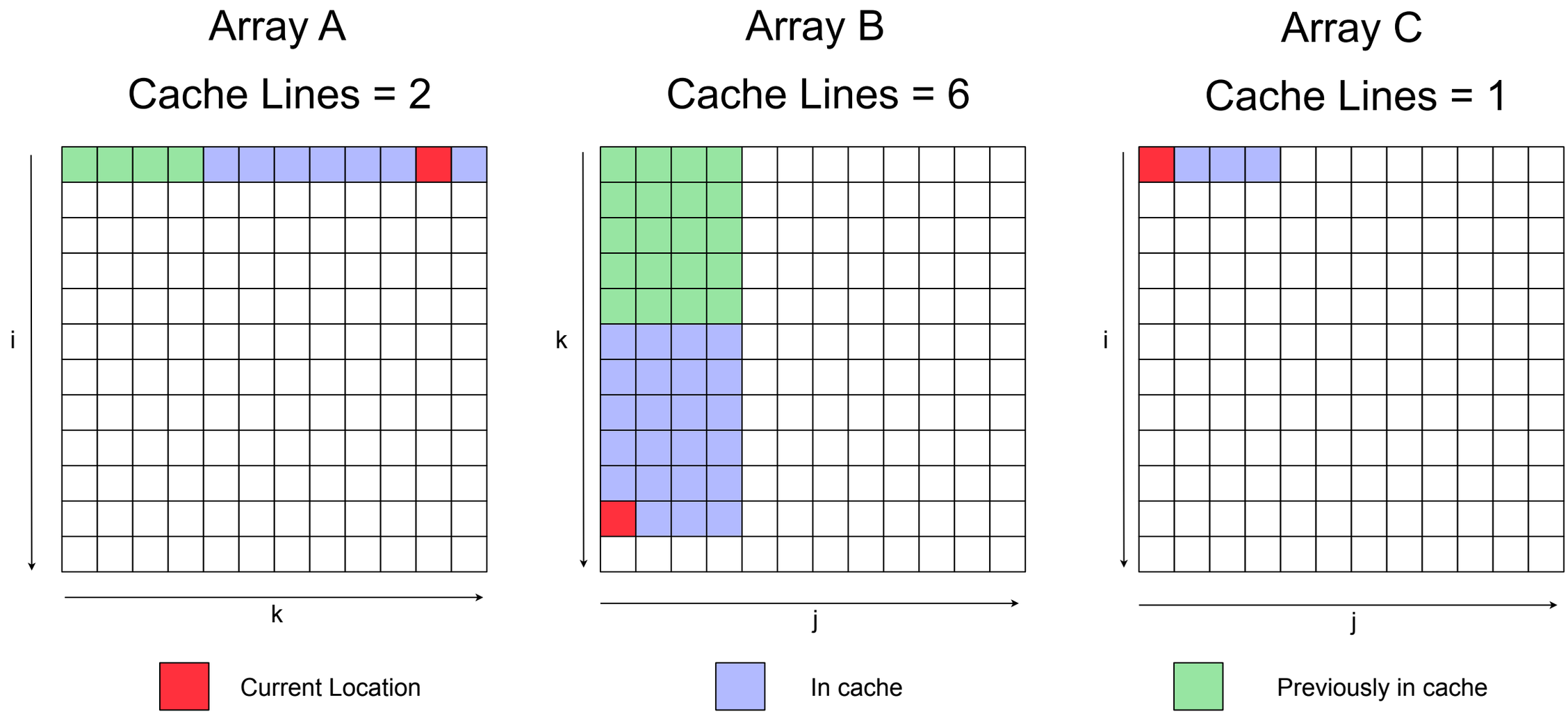
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 9 cache lines at a time, length 4 data elements



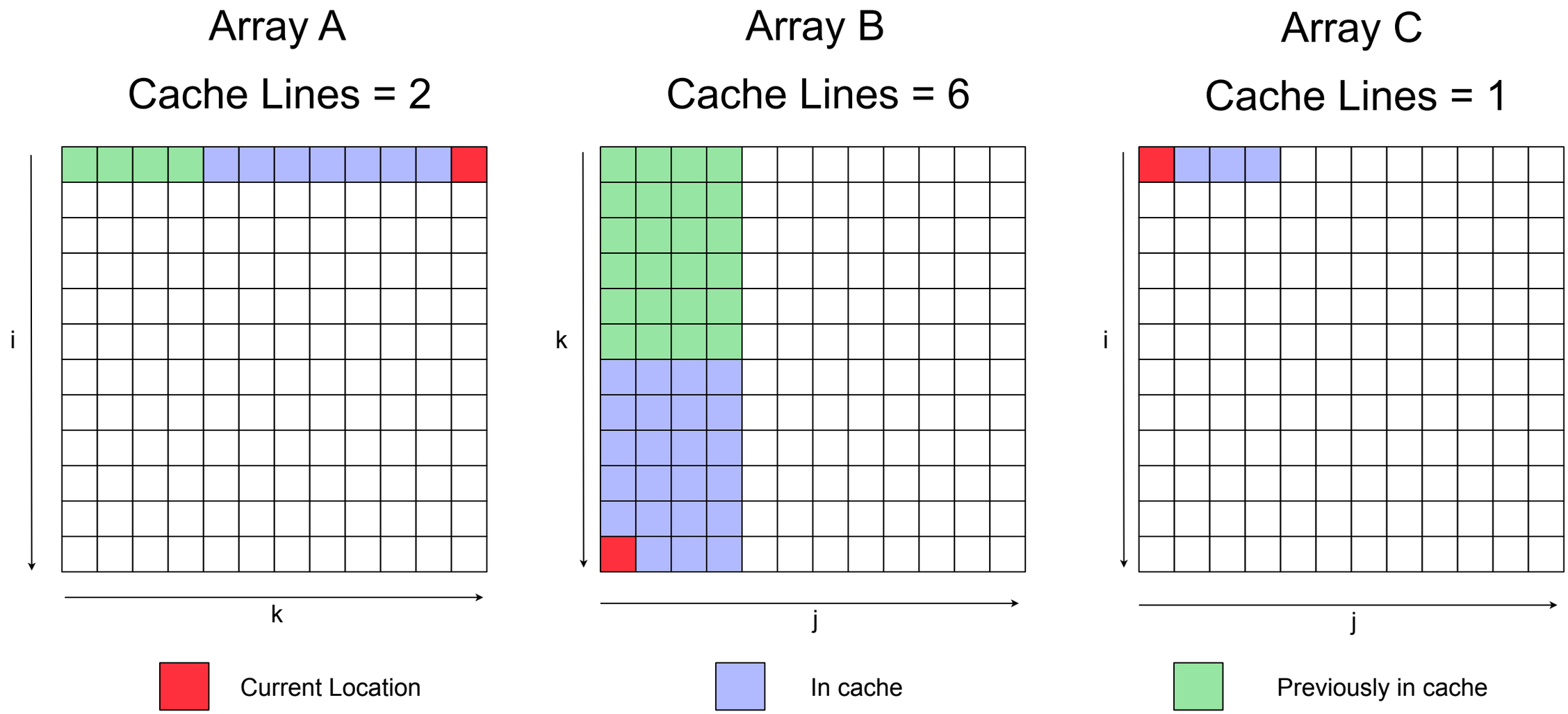
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 9 cache lines at a time, length 4 data elements



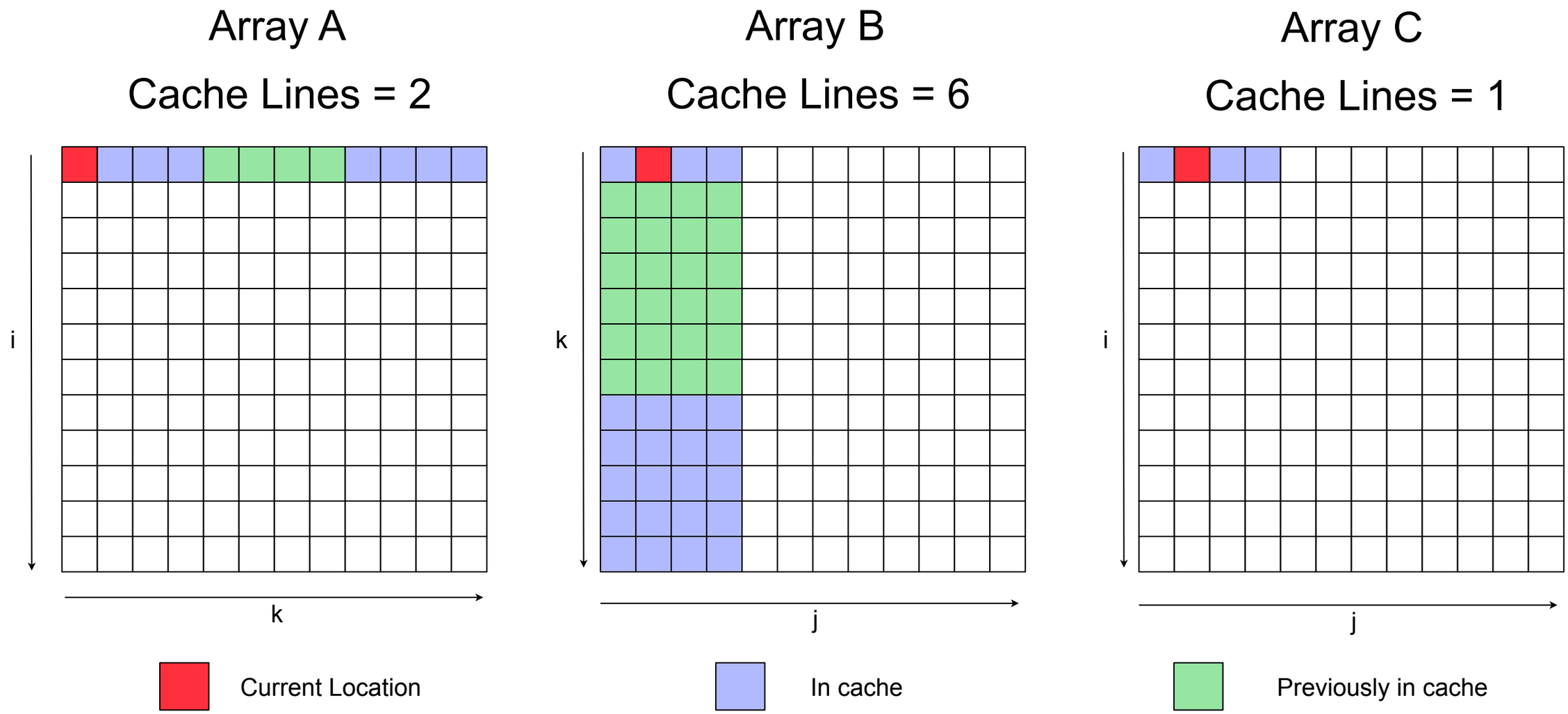
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 9 cache lines at a time, length 4 data elements



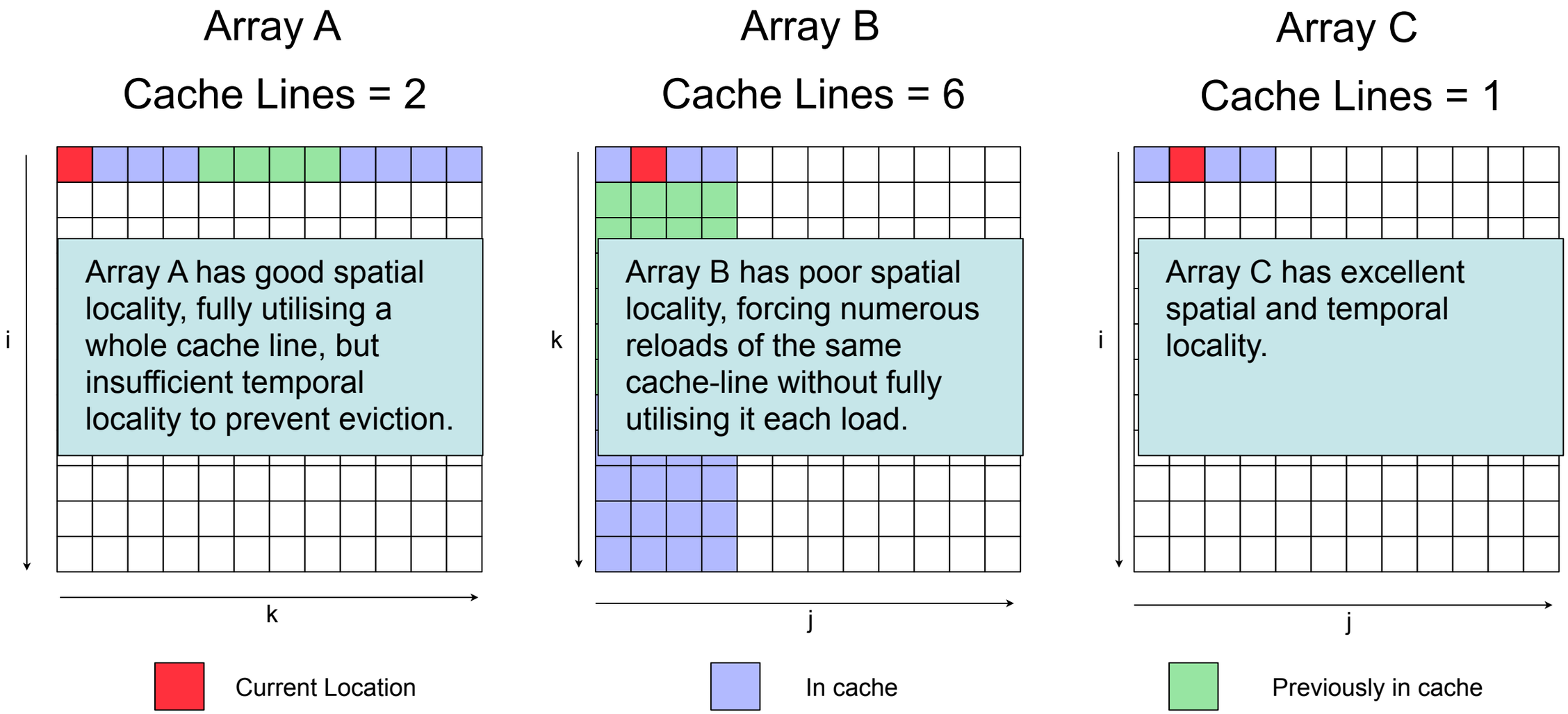
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 9 cache lines at a time, length 4 data elements



Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 9 cache lines at a time, length 4 data elements



Optimisation - Loop Blocking - Matrix Multiply

```
matrix_multiply(a,b,c,n){
    int b = BLOCK_SIZE;
    for(i0=0; i<n; i+=b){
        for(j0=0; j<n; j+=b){
            for(k0=0; k<n; k+=b){
                for(i=i0; i<min(i0+b,n); i++){
                    for(j=j0; j<min(j0+b,n); j++){
                        for(k=k0; k<min(k0+b,n); k++){
                            c[i,j] = c[i,j] + (a[i,k] * b[k,j]);
                        }
                    }
                }
            }
        }
    }
}
```

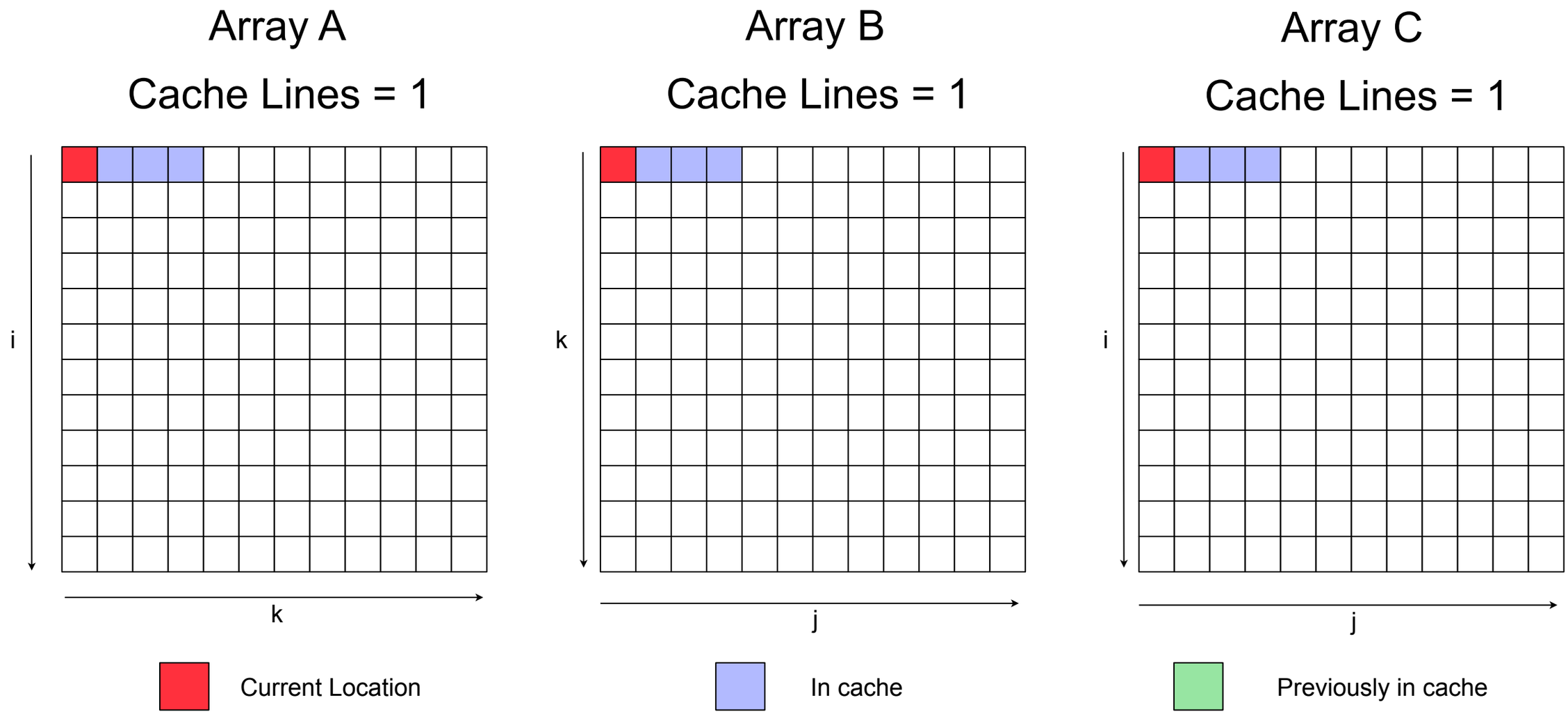
Rather than looping over an nxn block, the 3 internal loops now loop over ((n*n)/b) sub-blocks of size bxb.

The spatial locality of a[i,k] and b[k,j] in memory is closer, resulting in a better chance of values being retained in cache.

The optimal block size,b, is dependant upon the size of cache-lines and total cache size, requiring fine-tuning between machines. Aim is to select a block size such that a sub-block of data fits in cache.

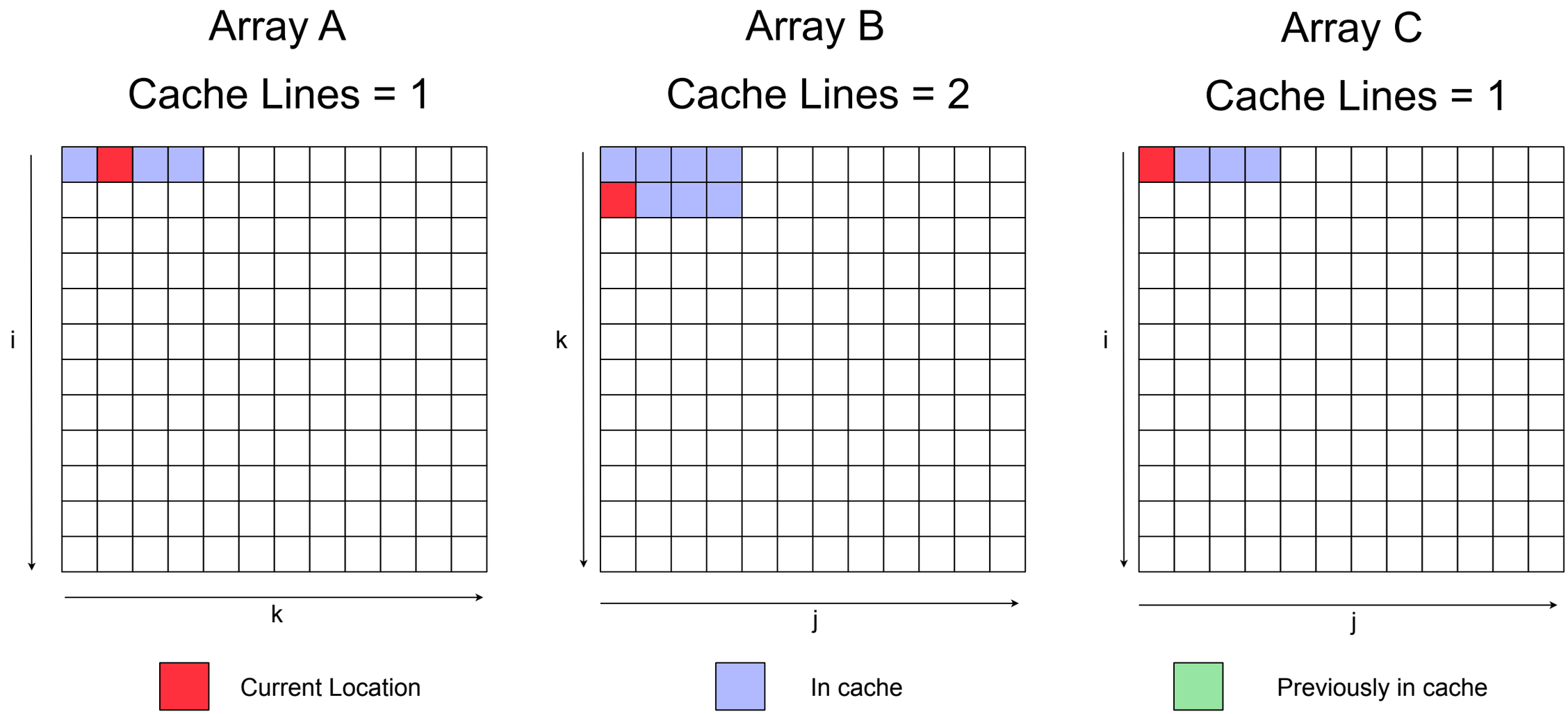
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



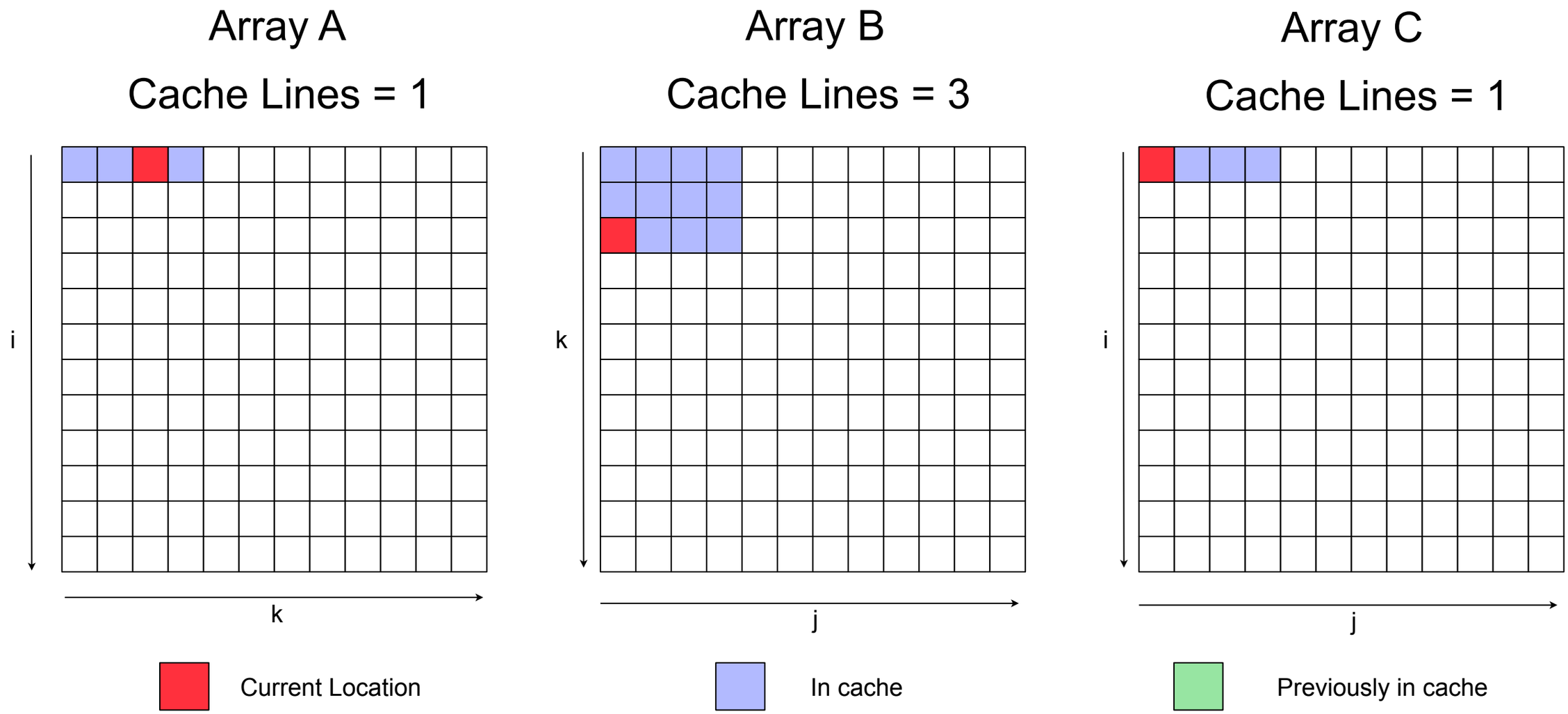
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



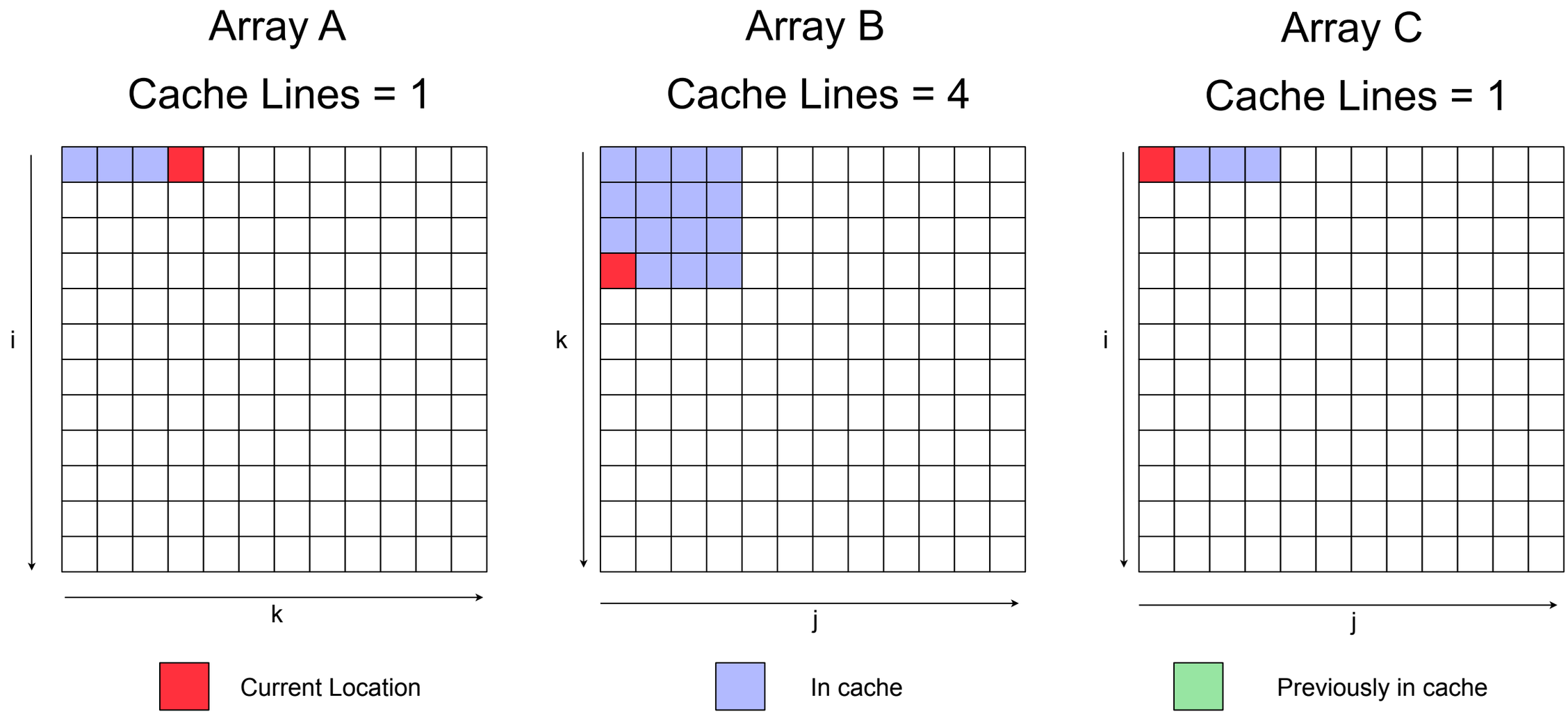
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



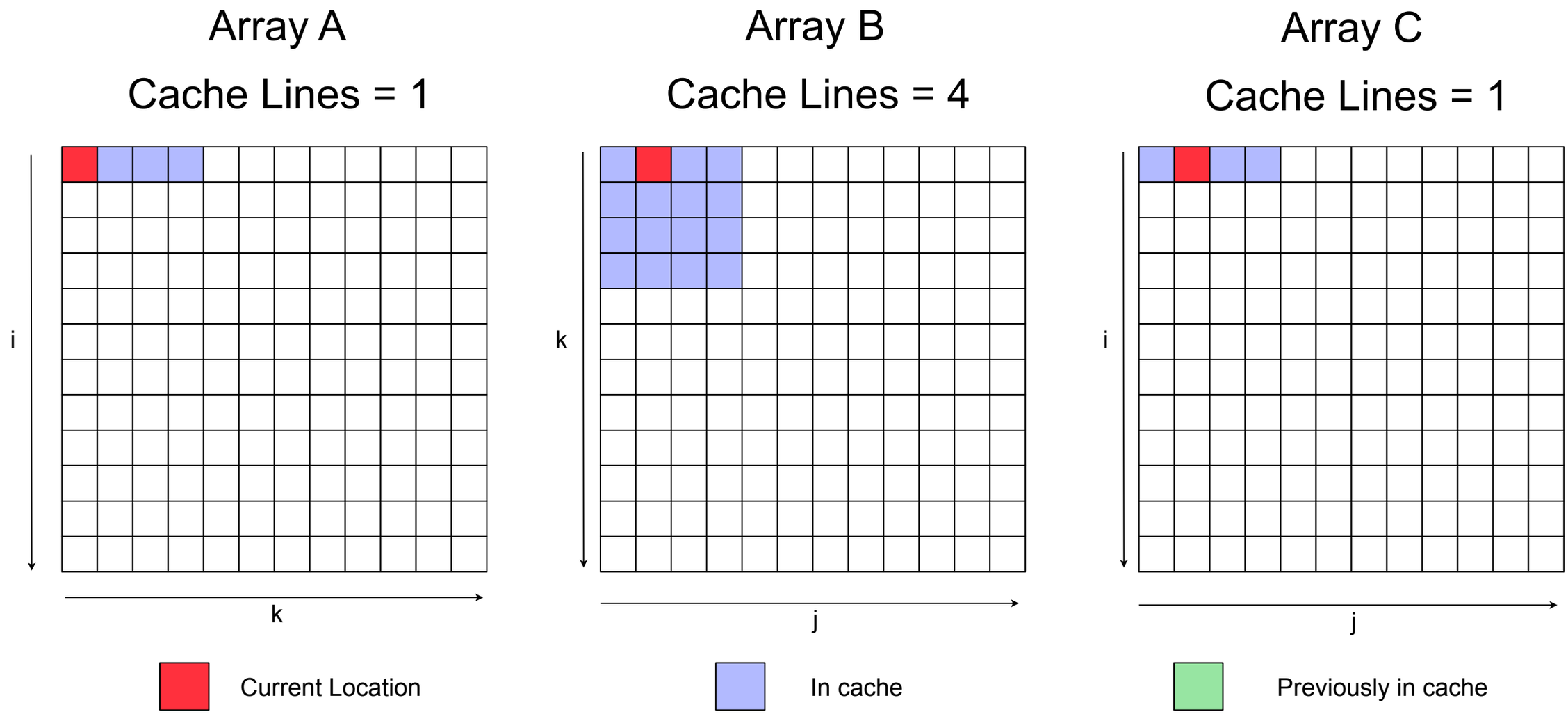
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



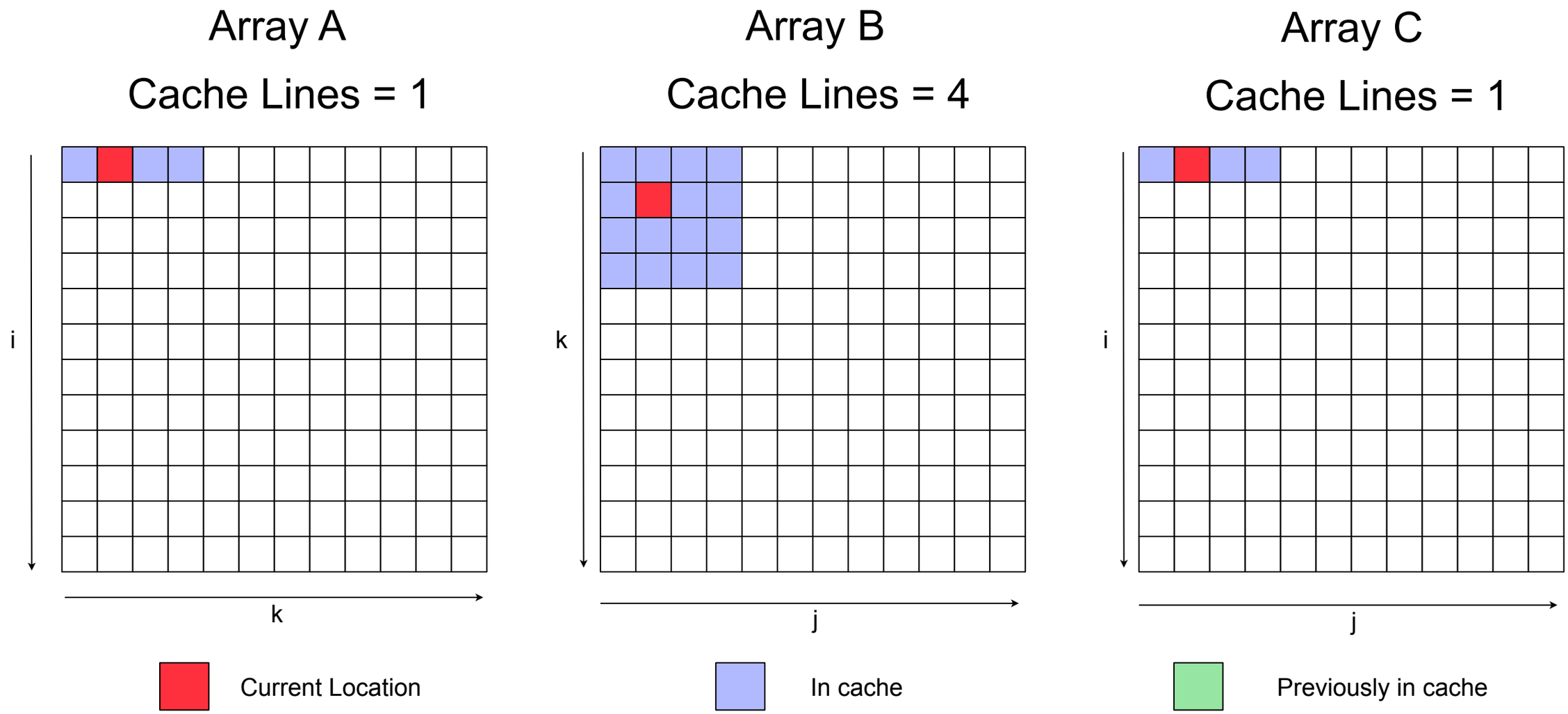
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



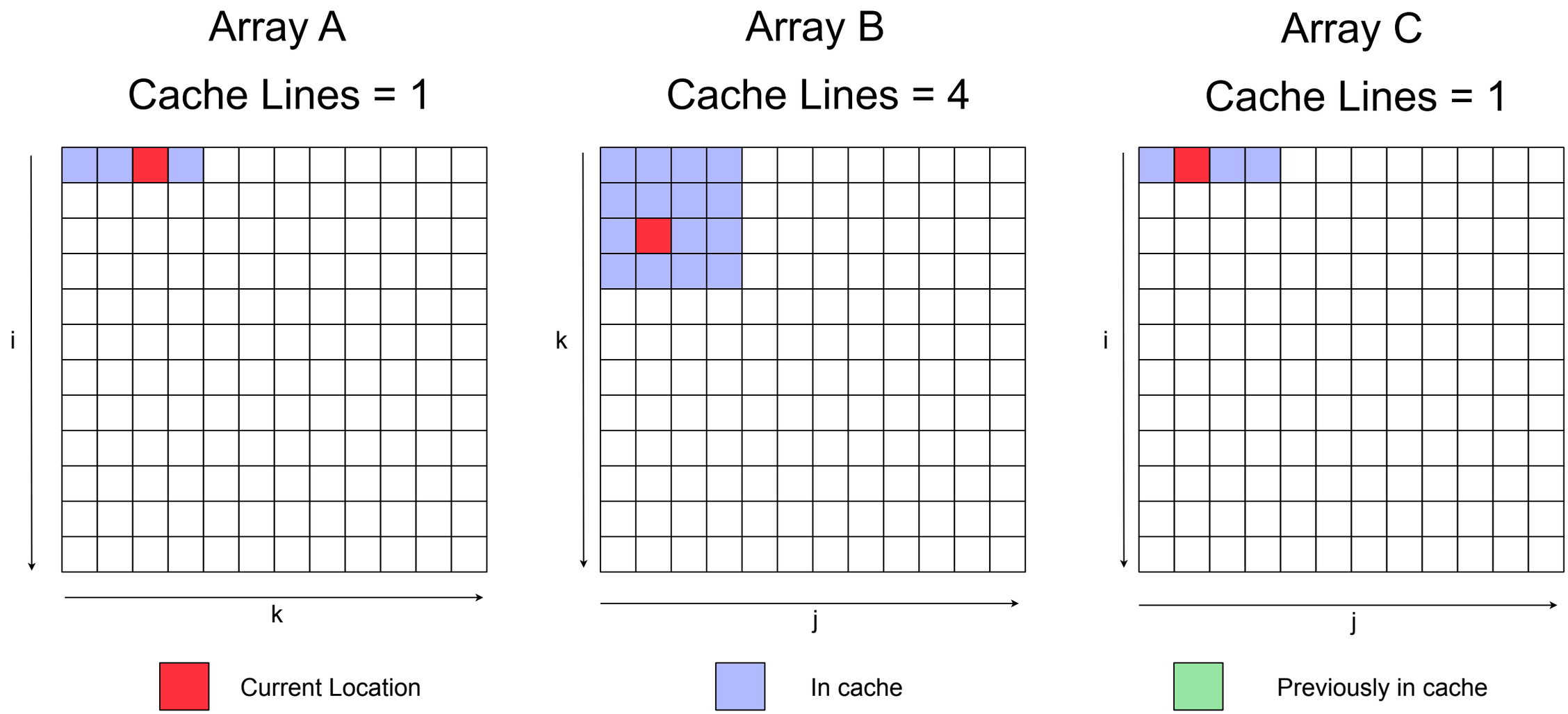
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



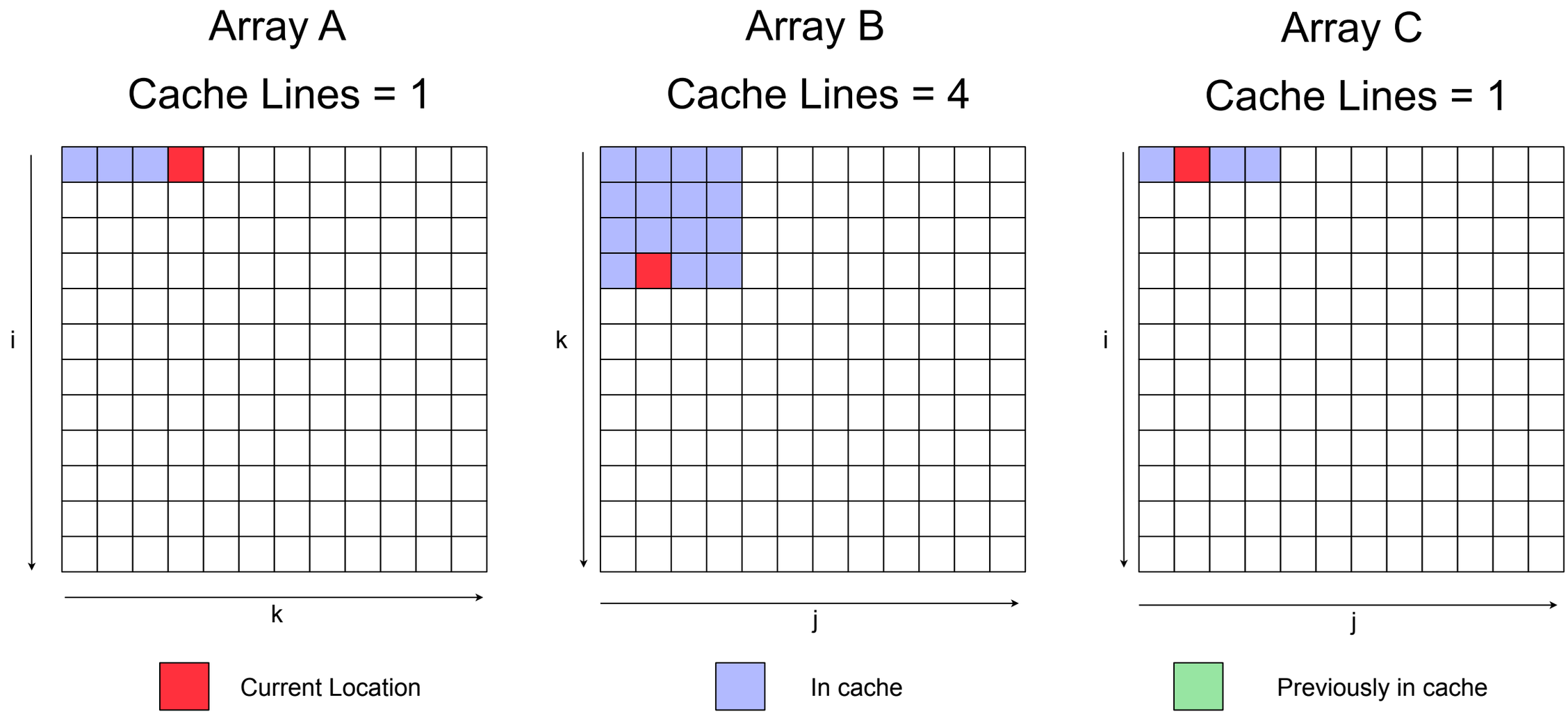
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



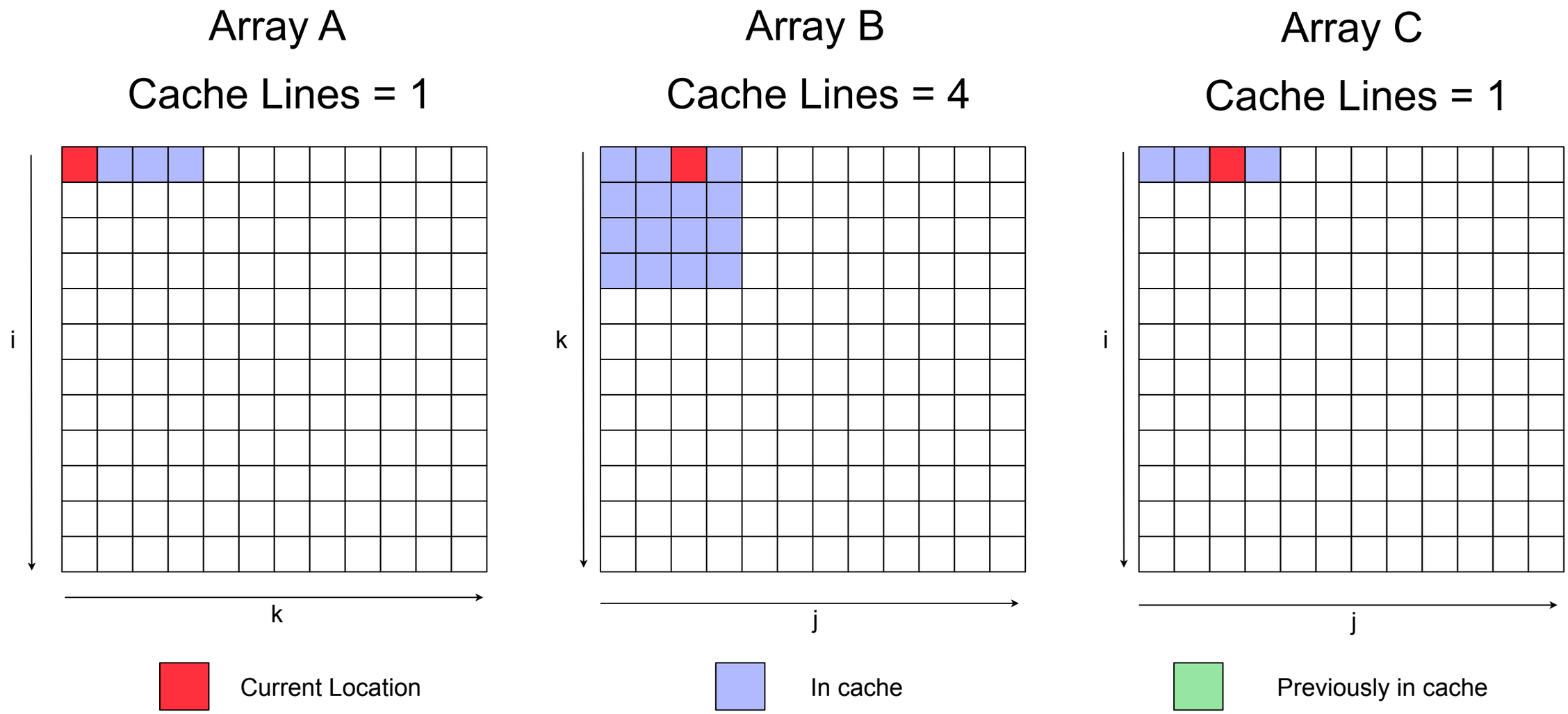
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



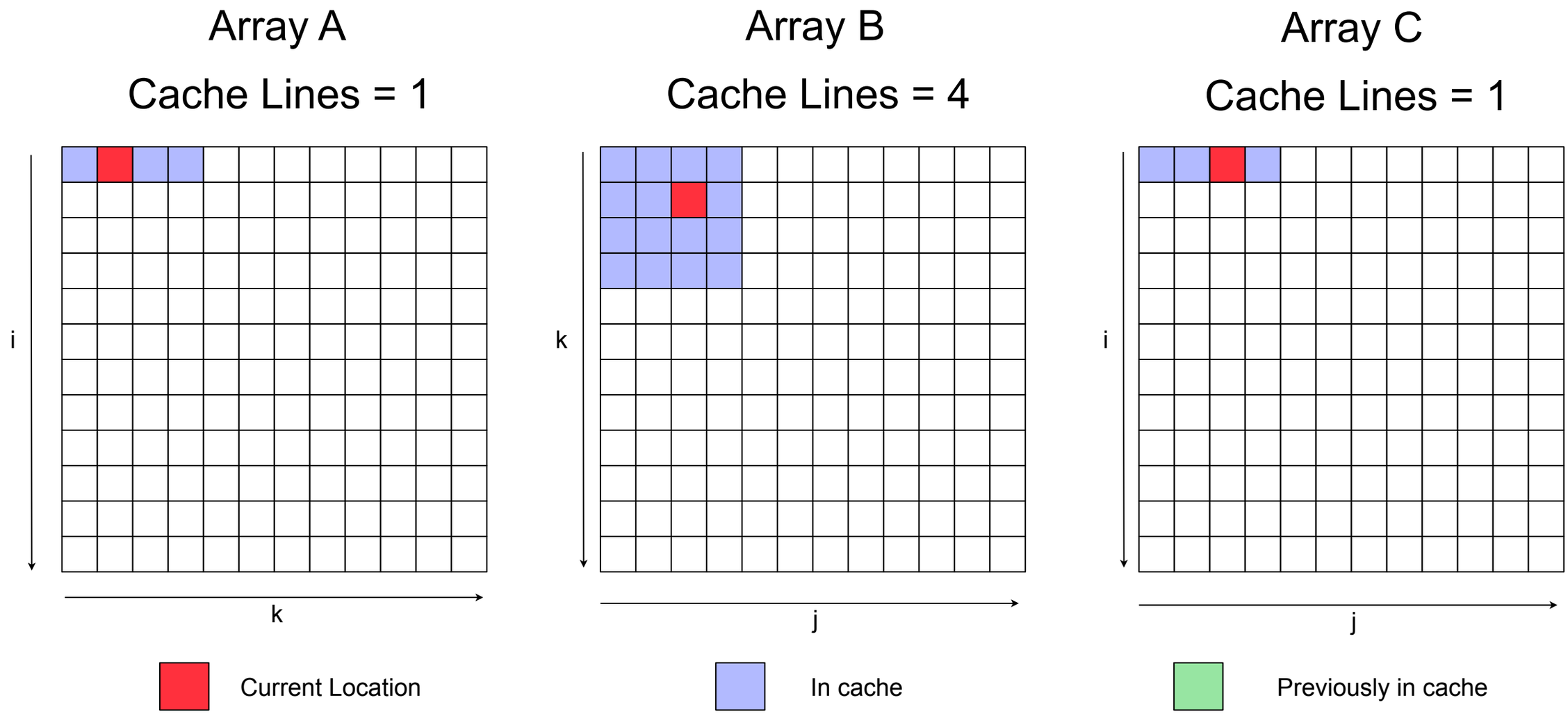
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



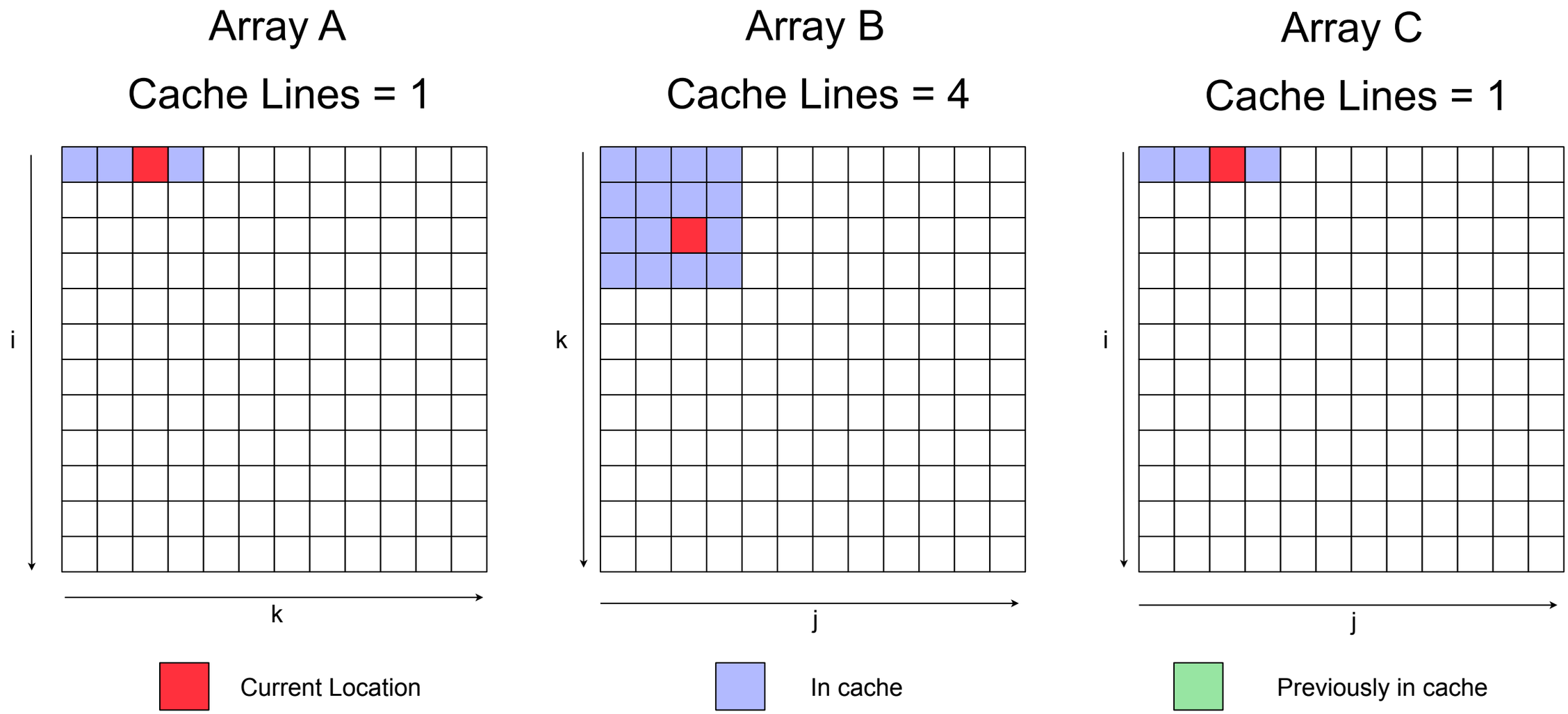
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



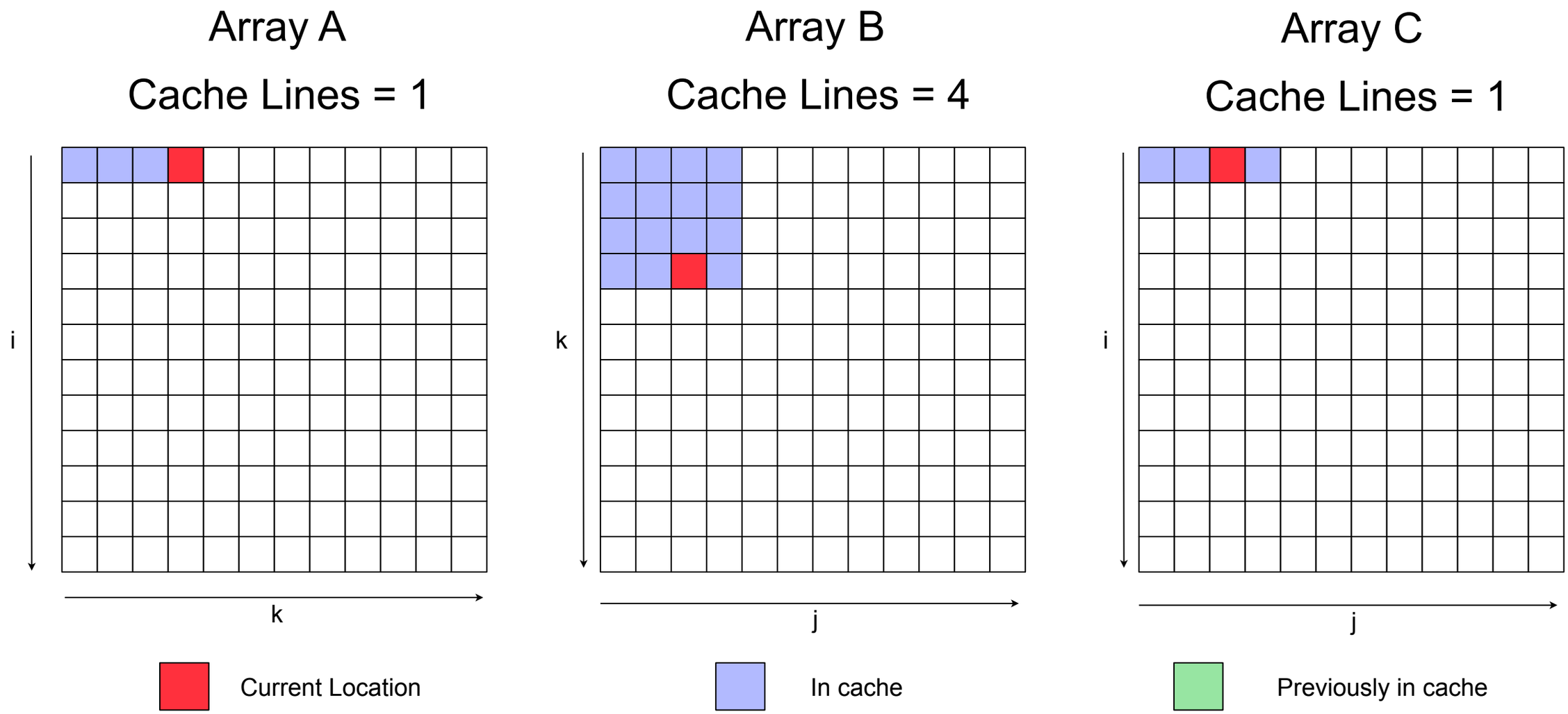
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



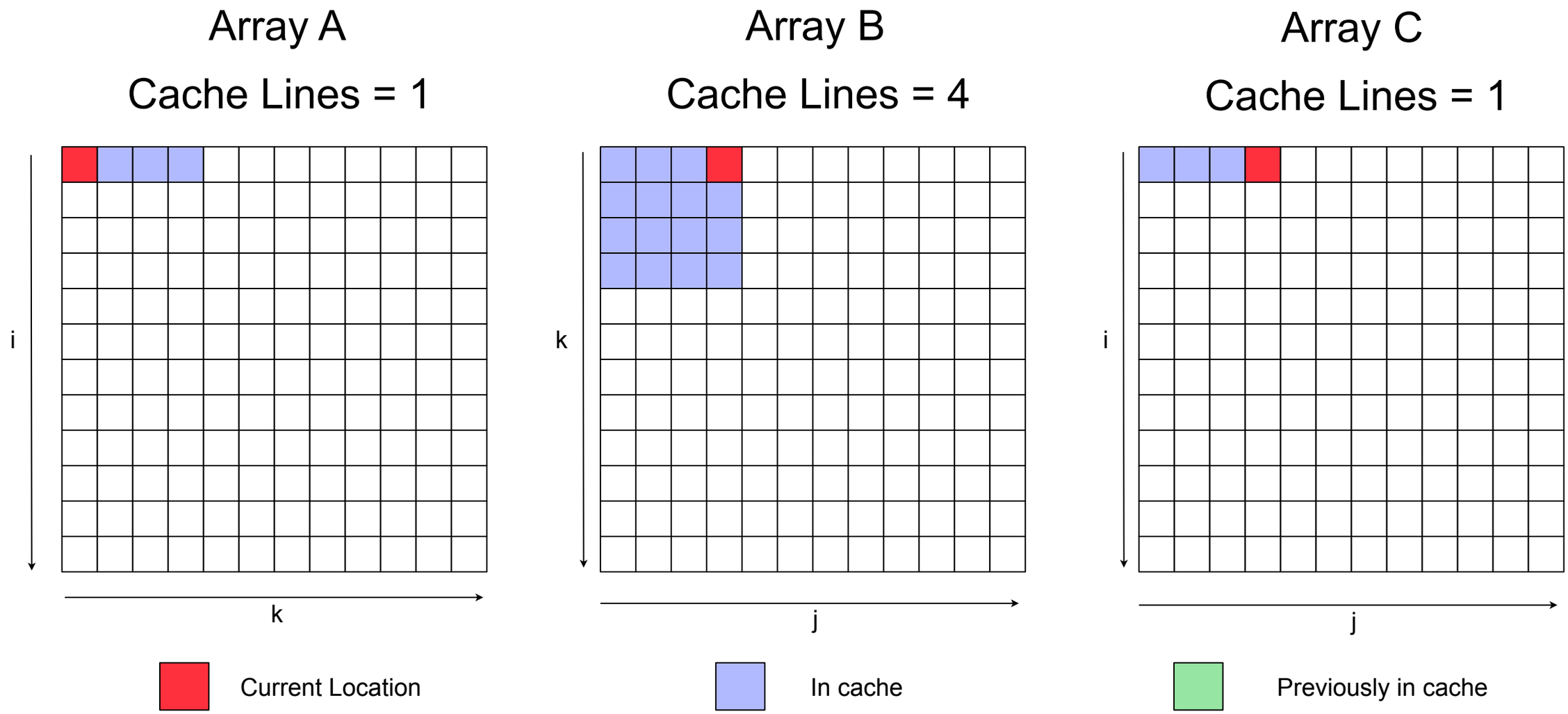
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



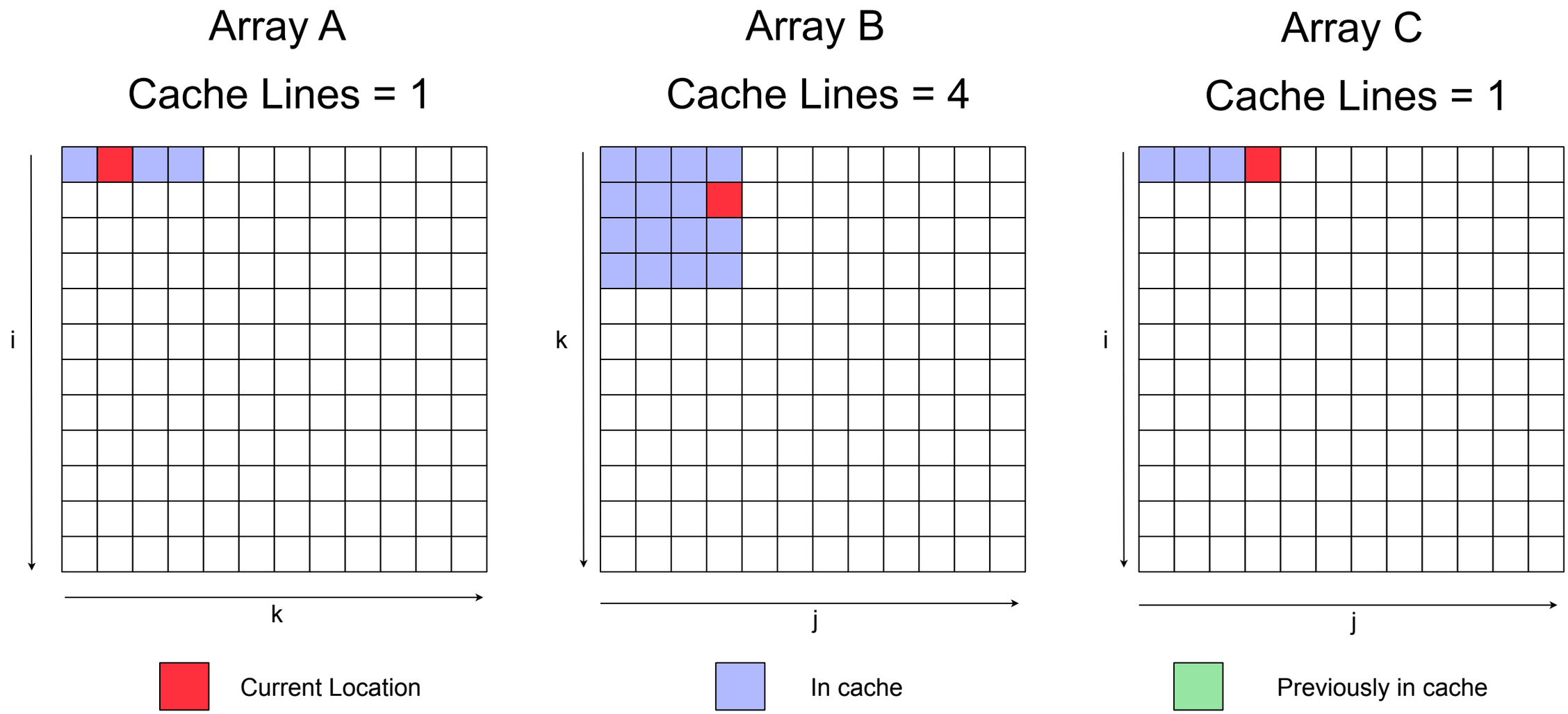
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



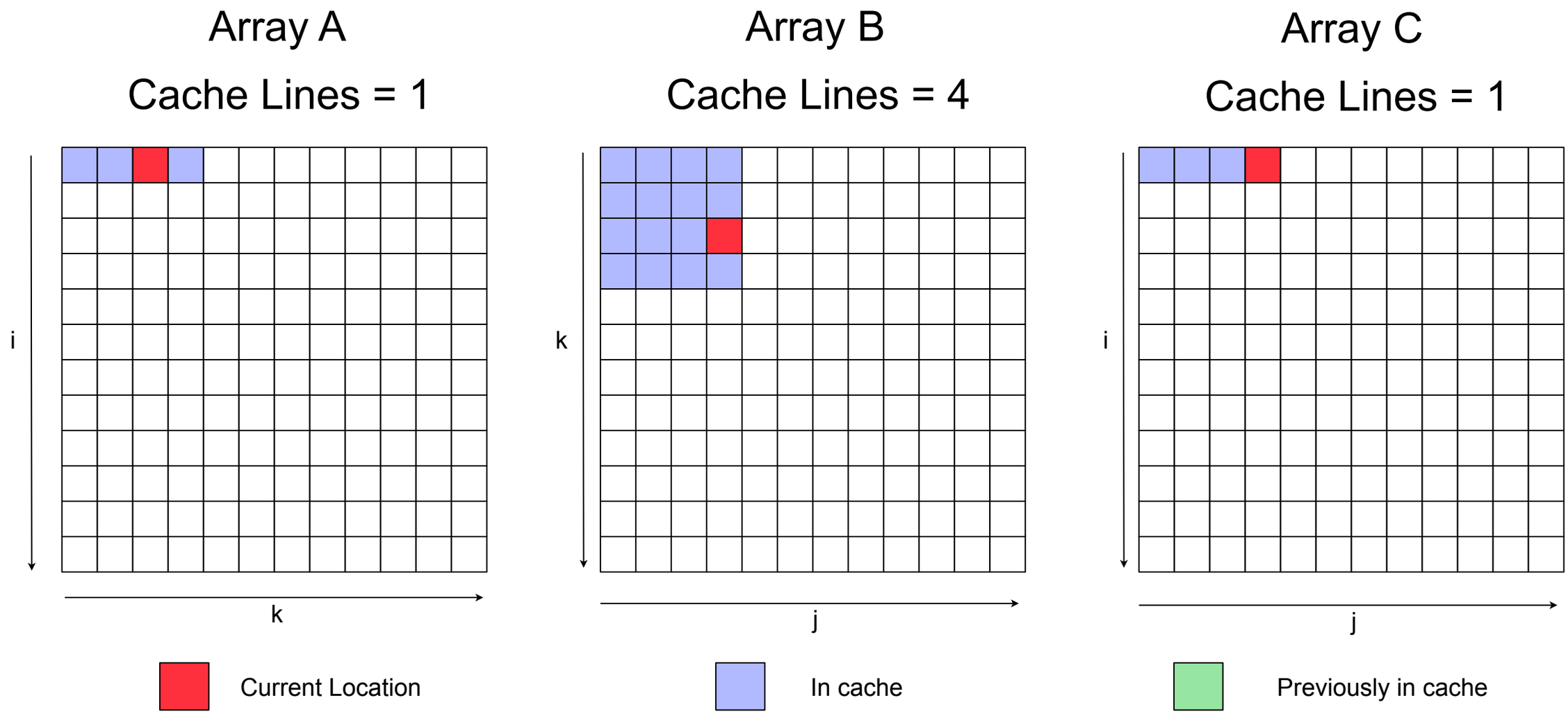
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



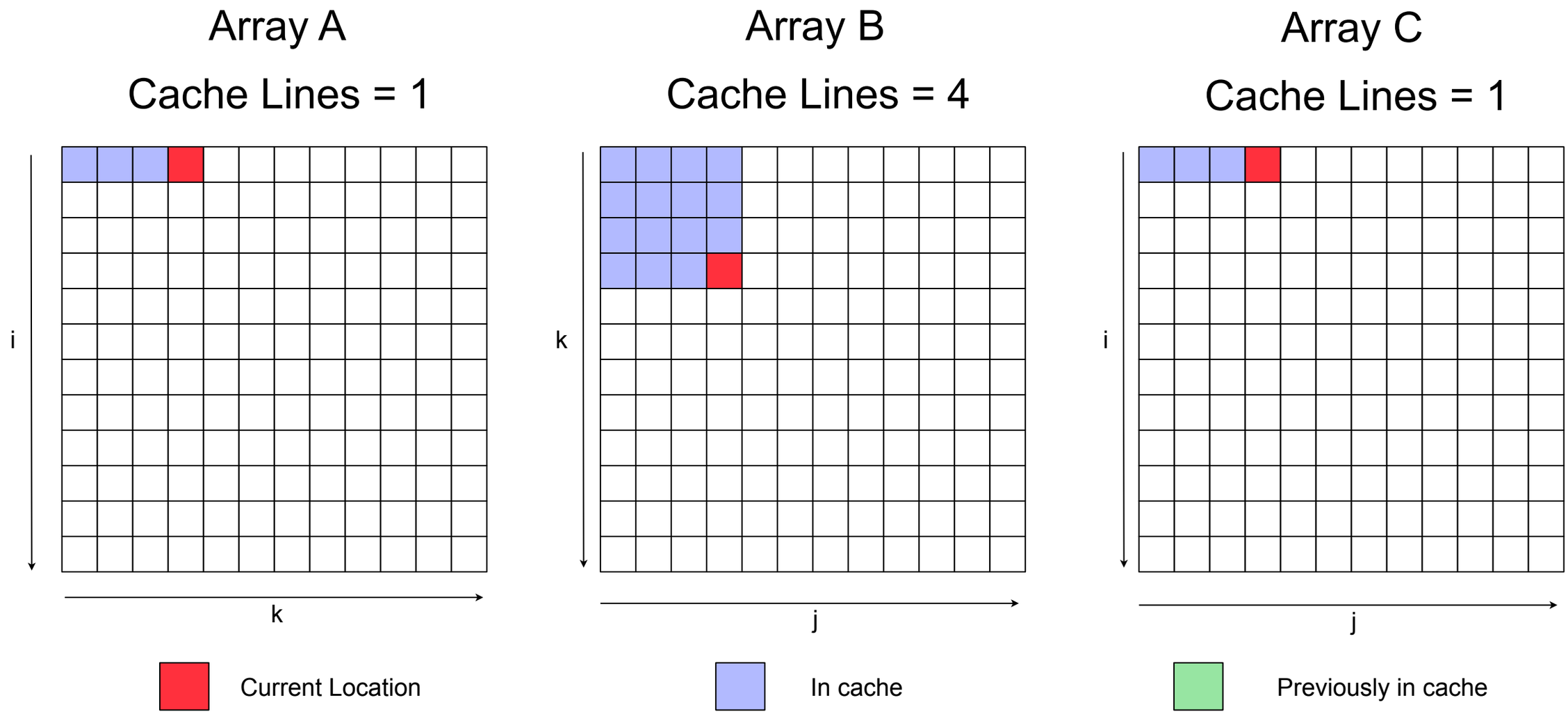
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



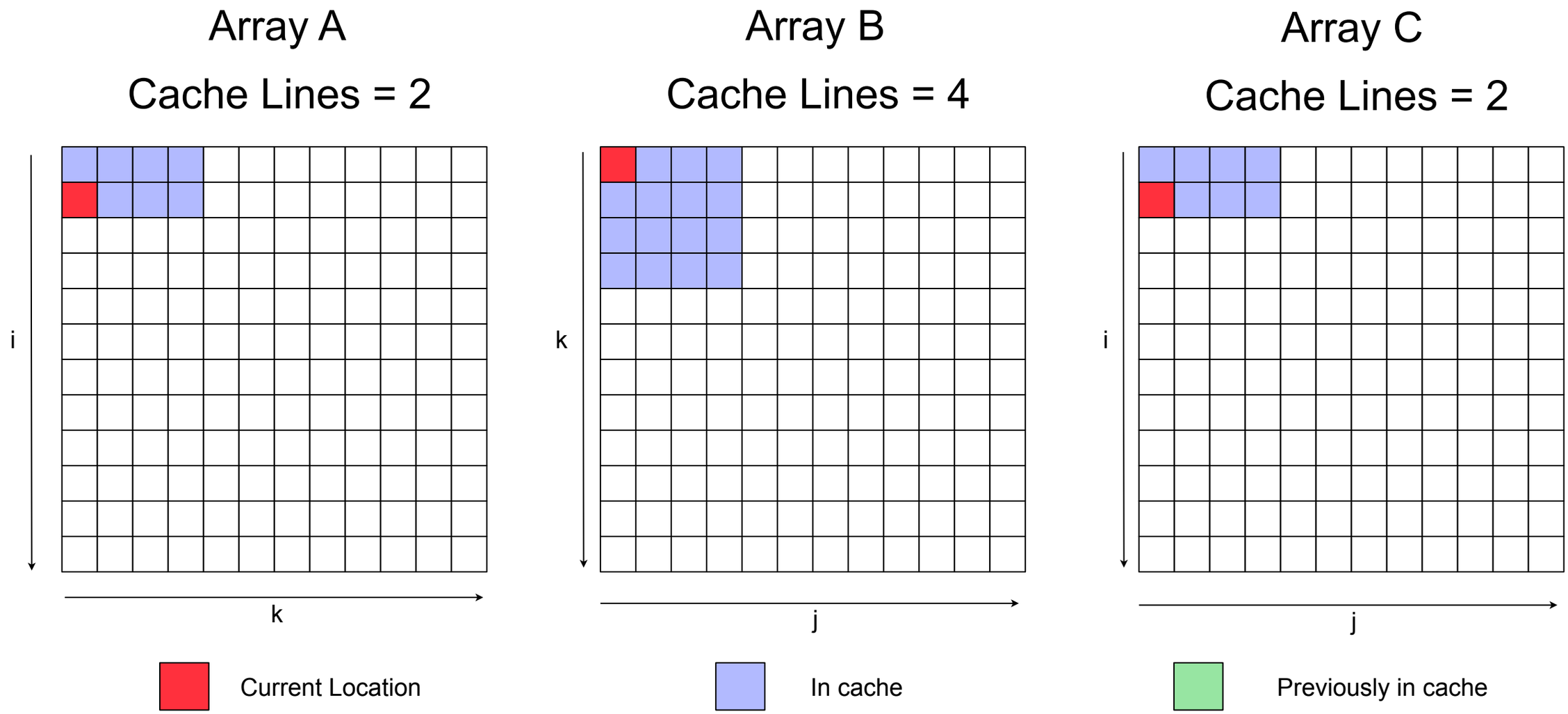
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



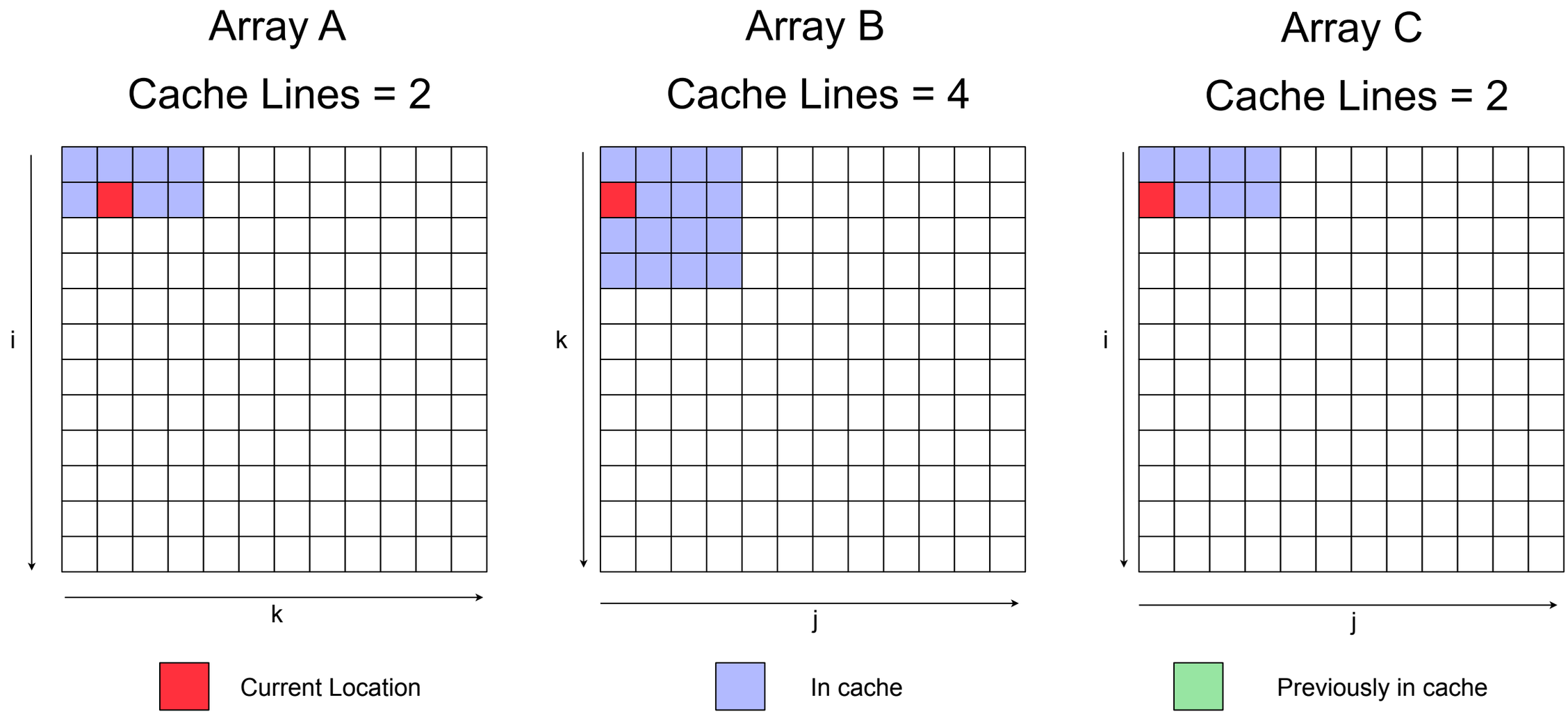
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



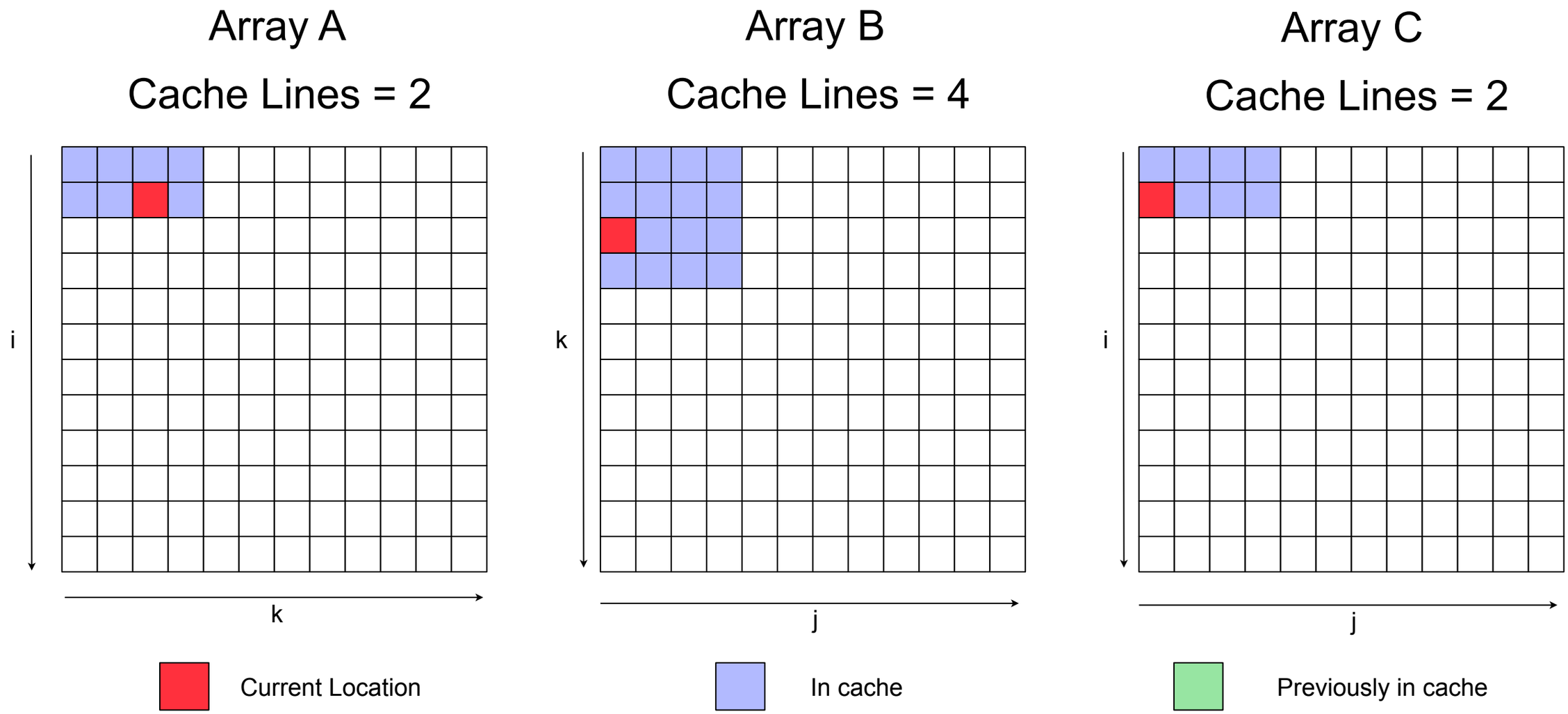
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



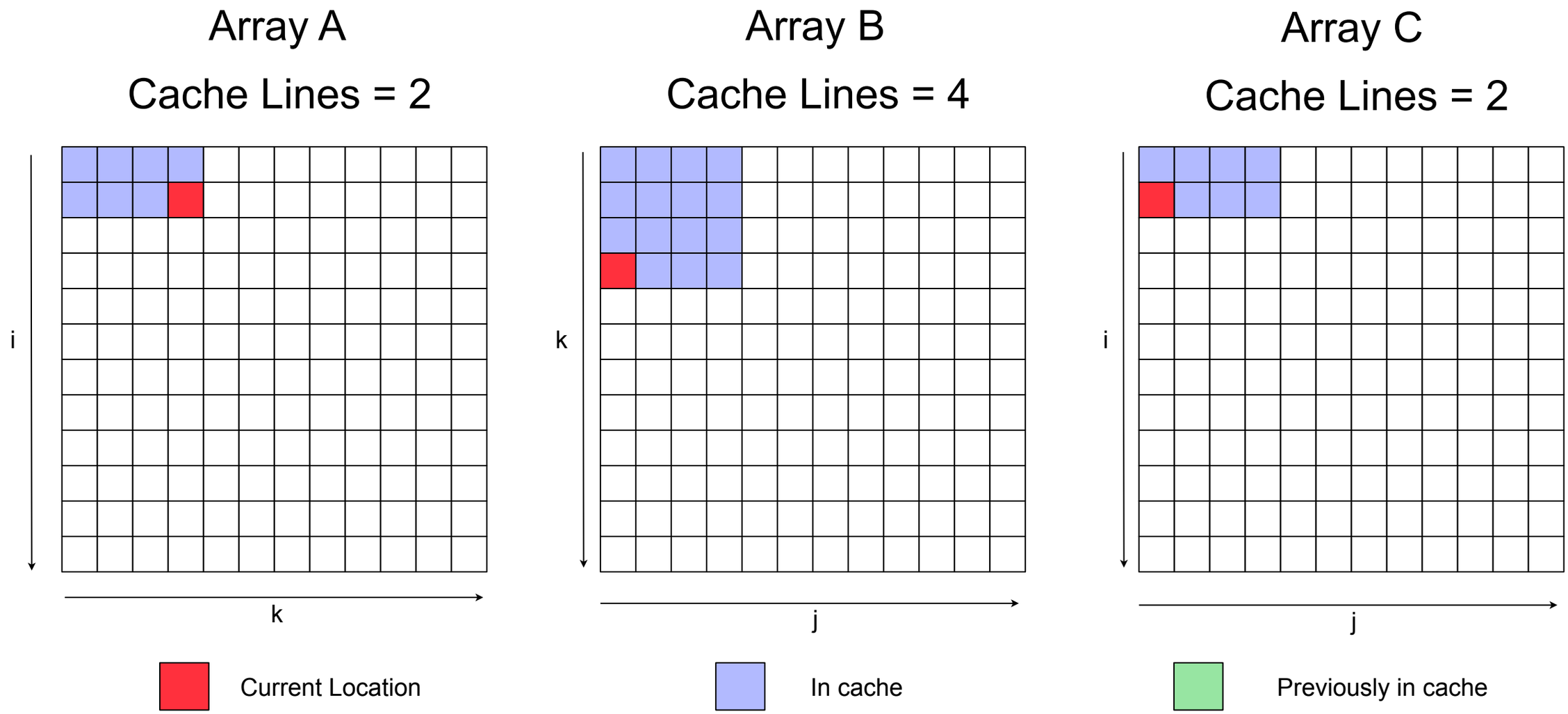
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



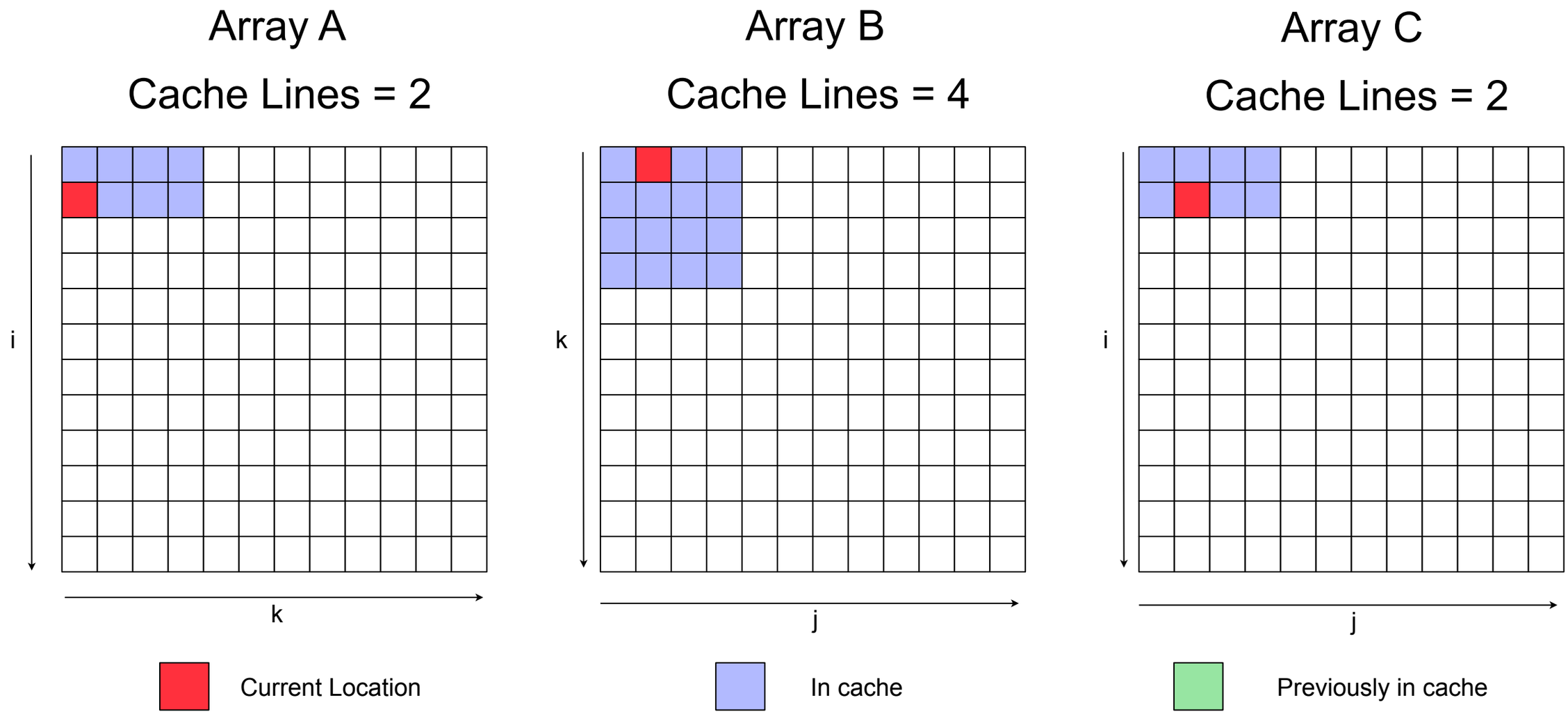
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



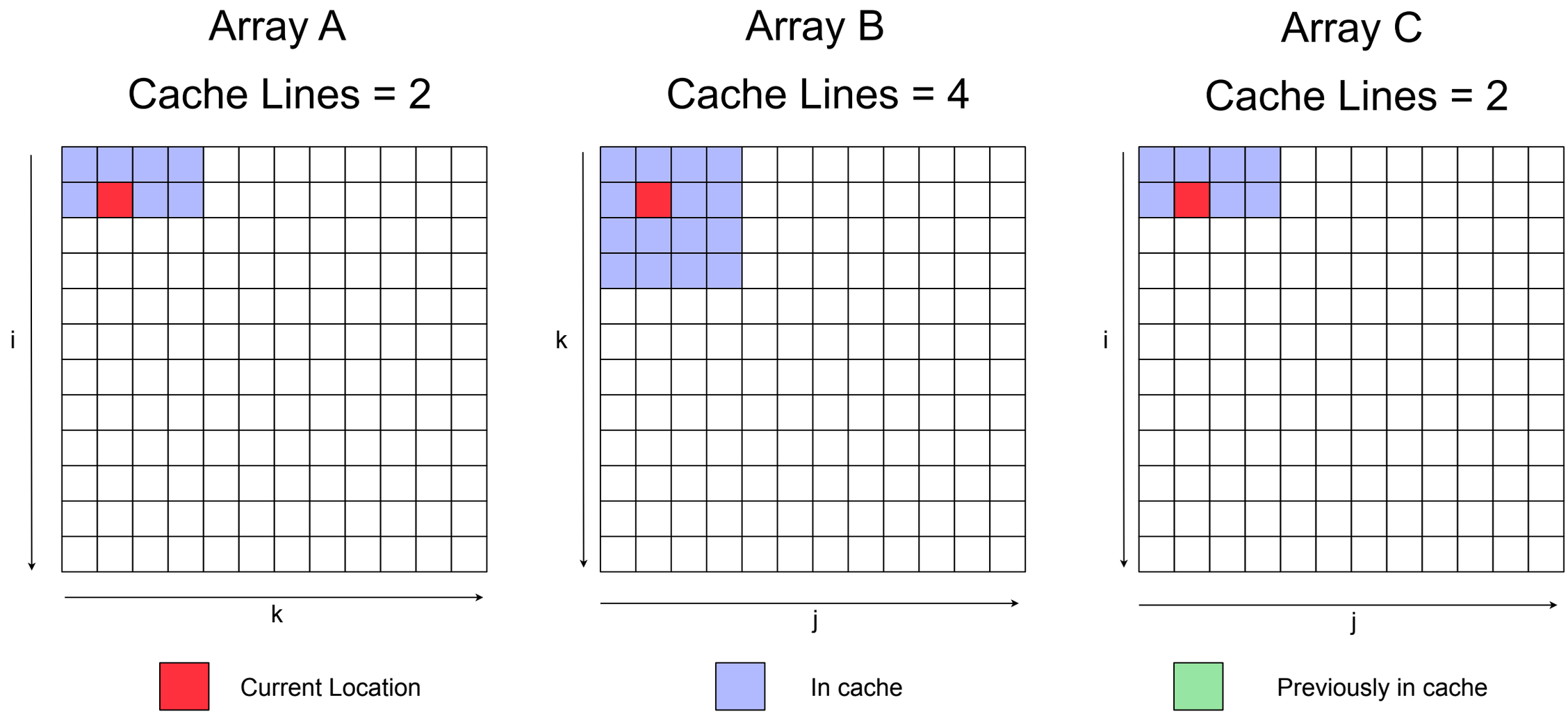
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



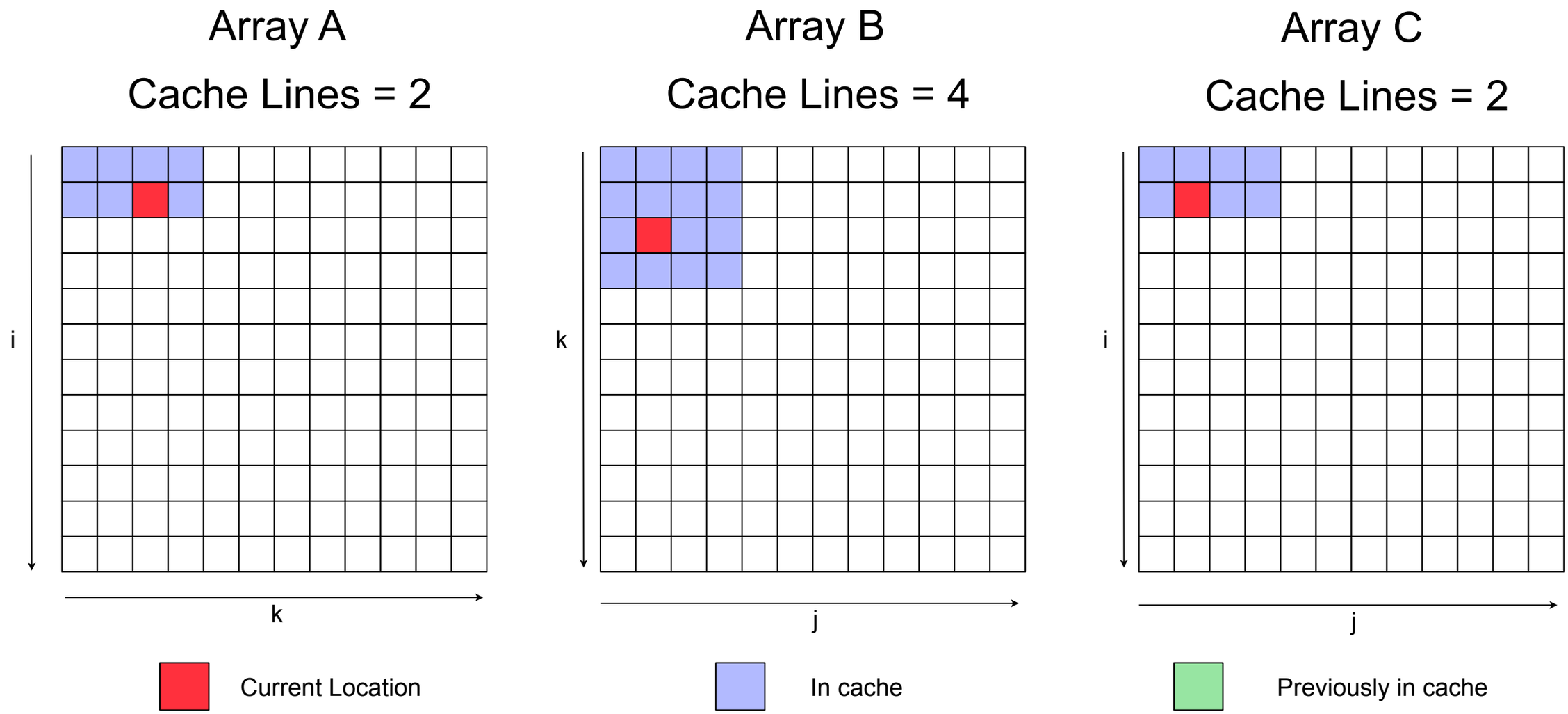
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



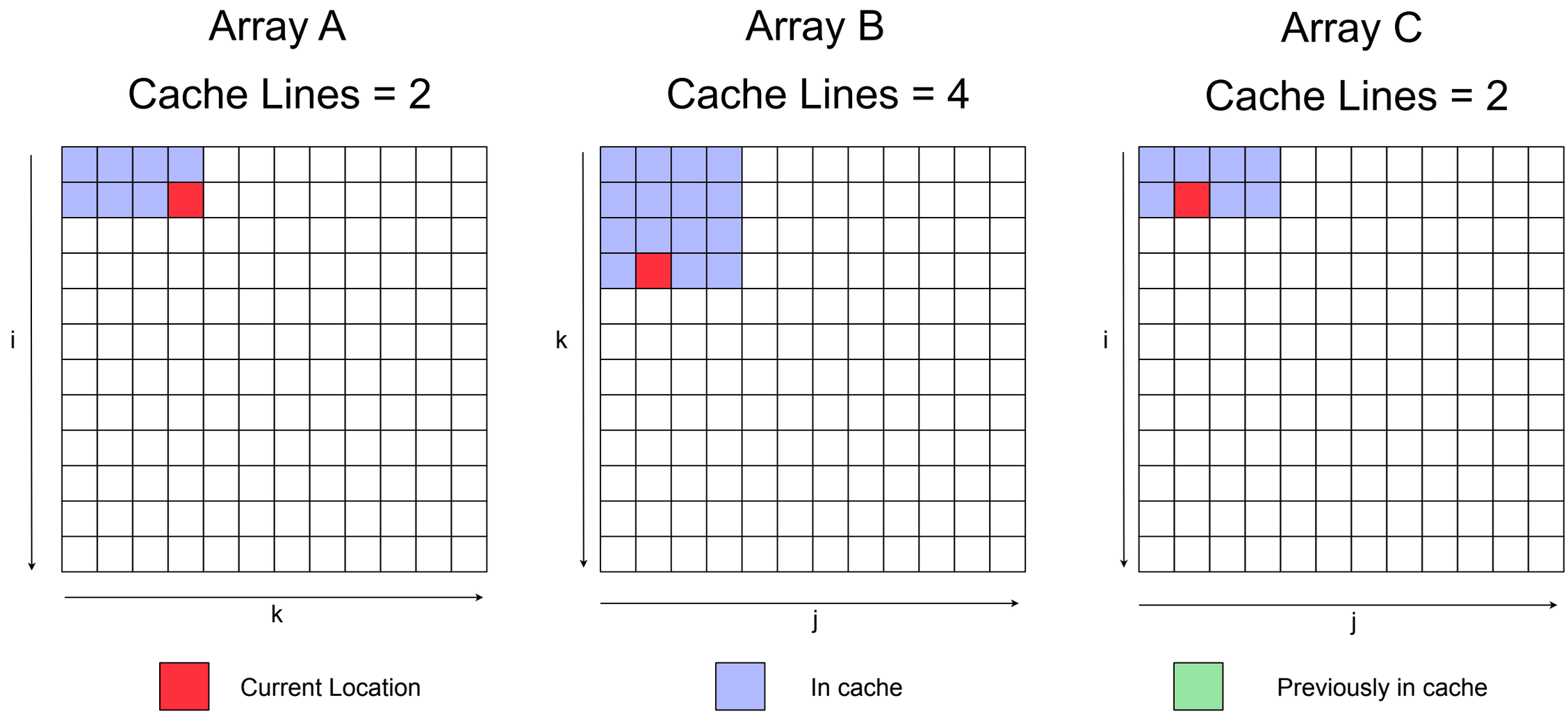
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



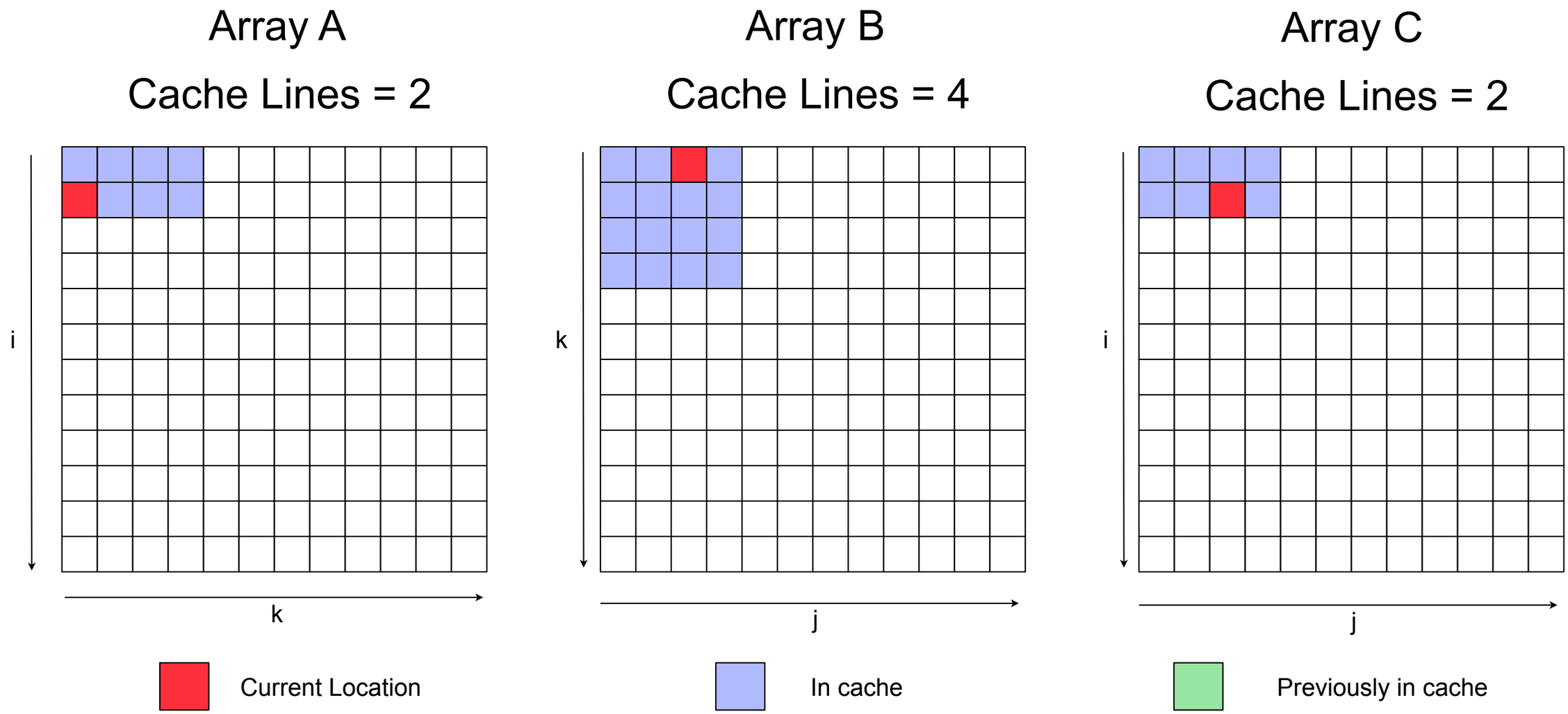
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



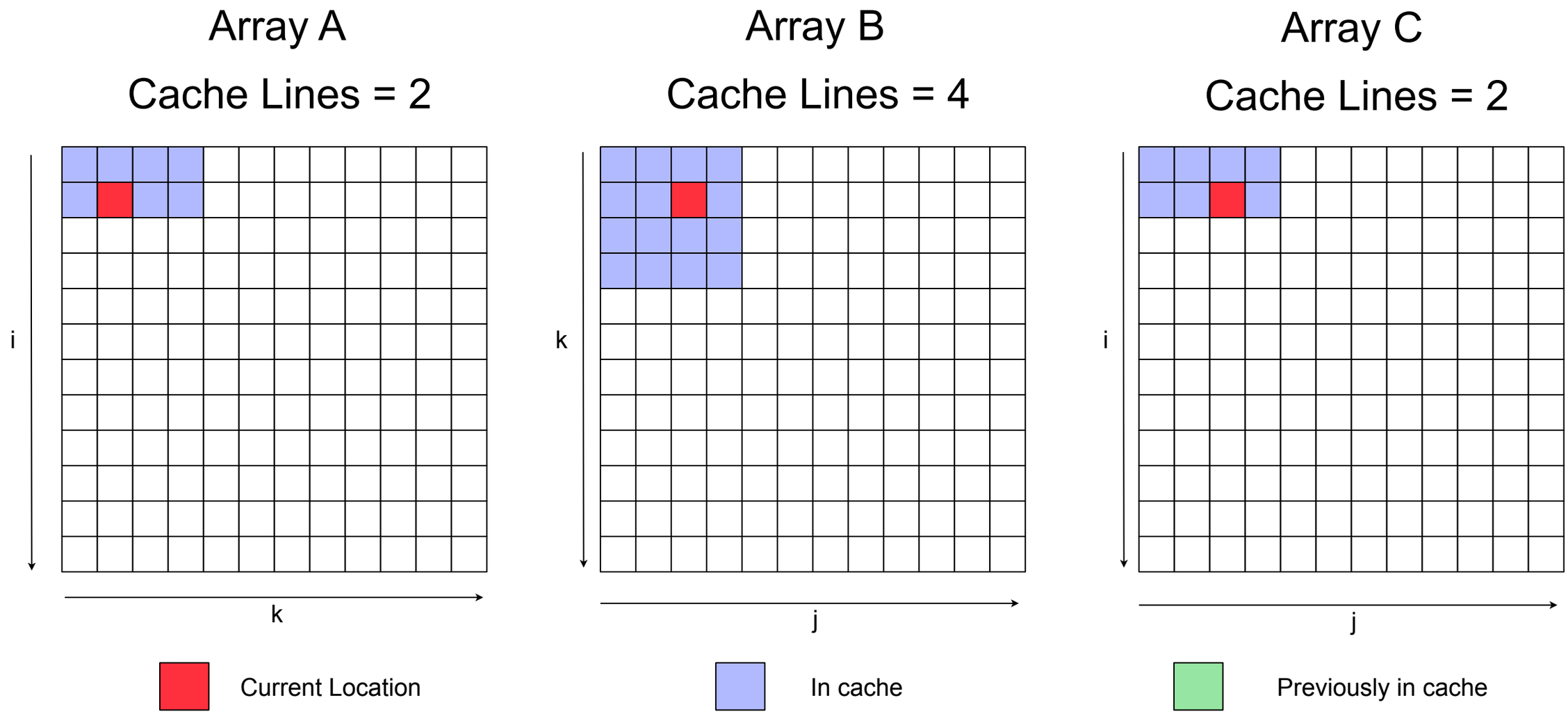
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



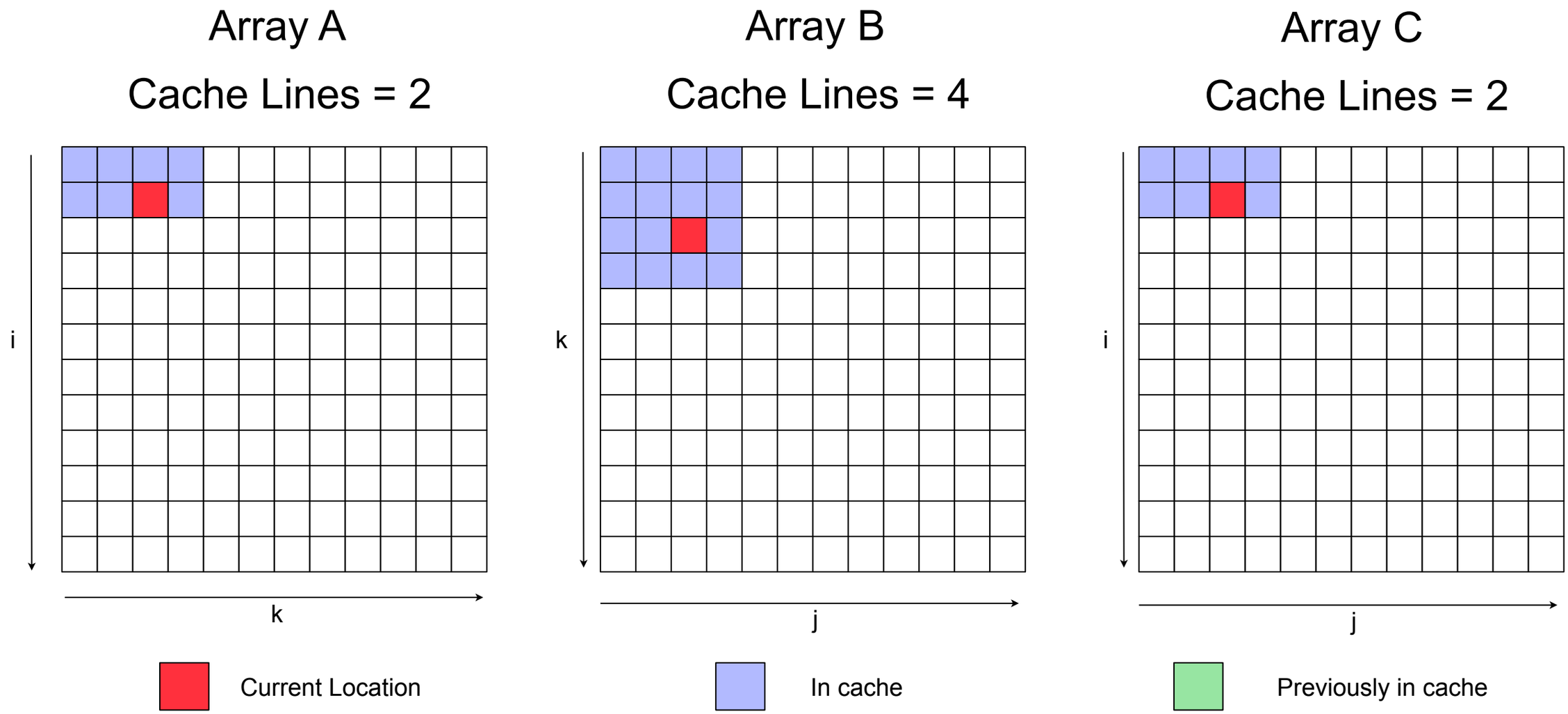
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



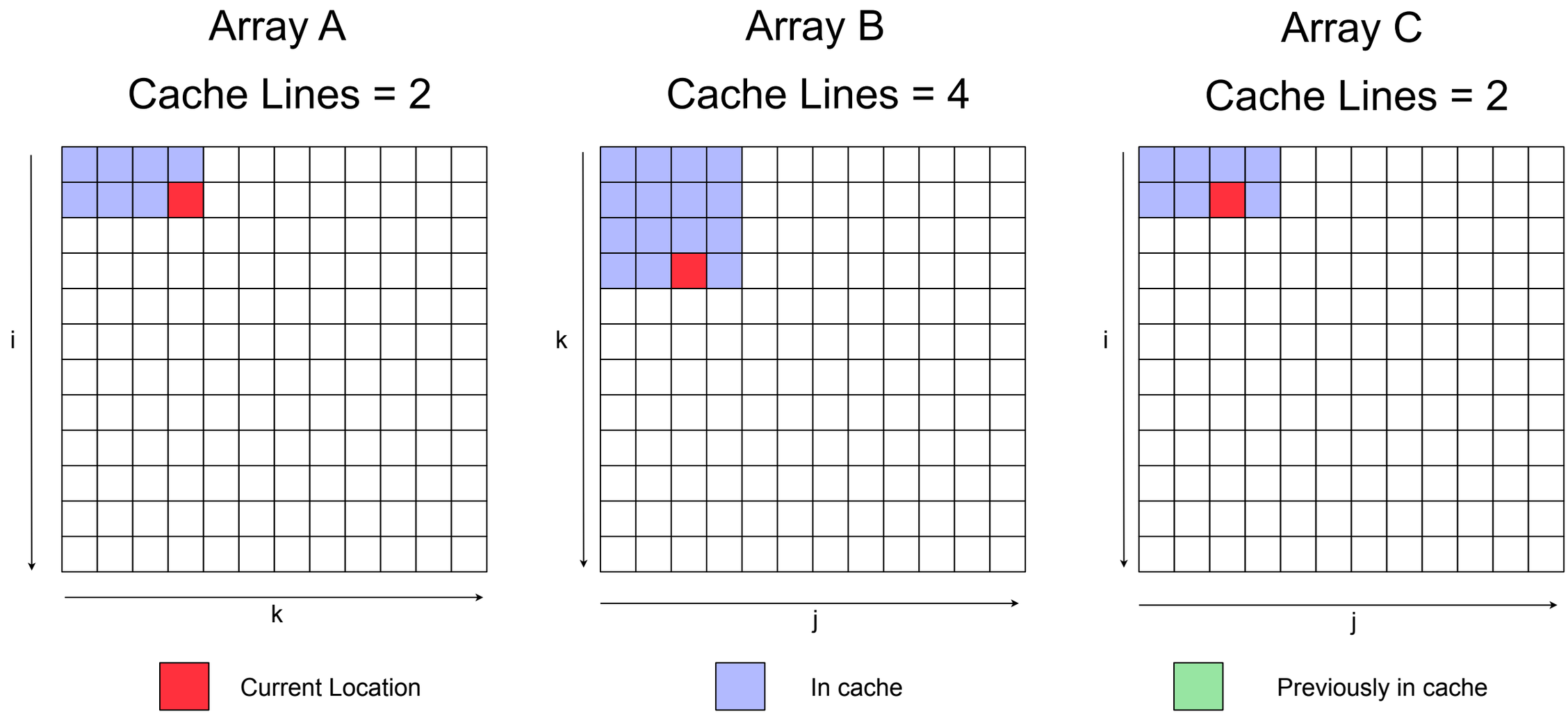
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



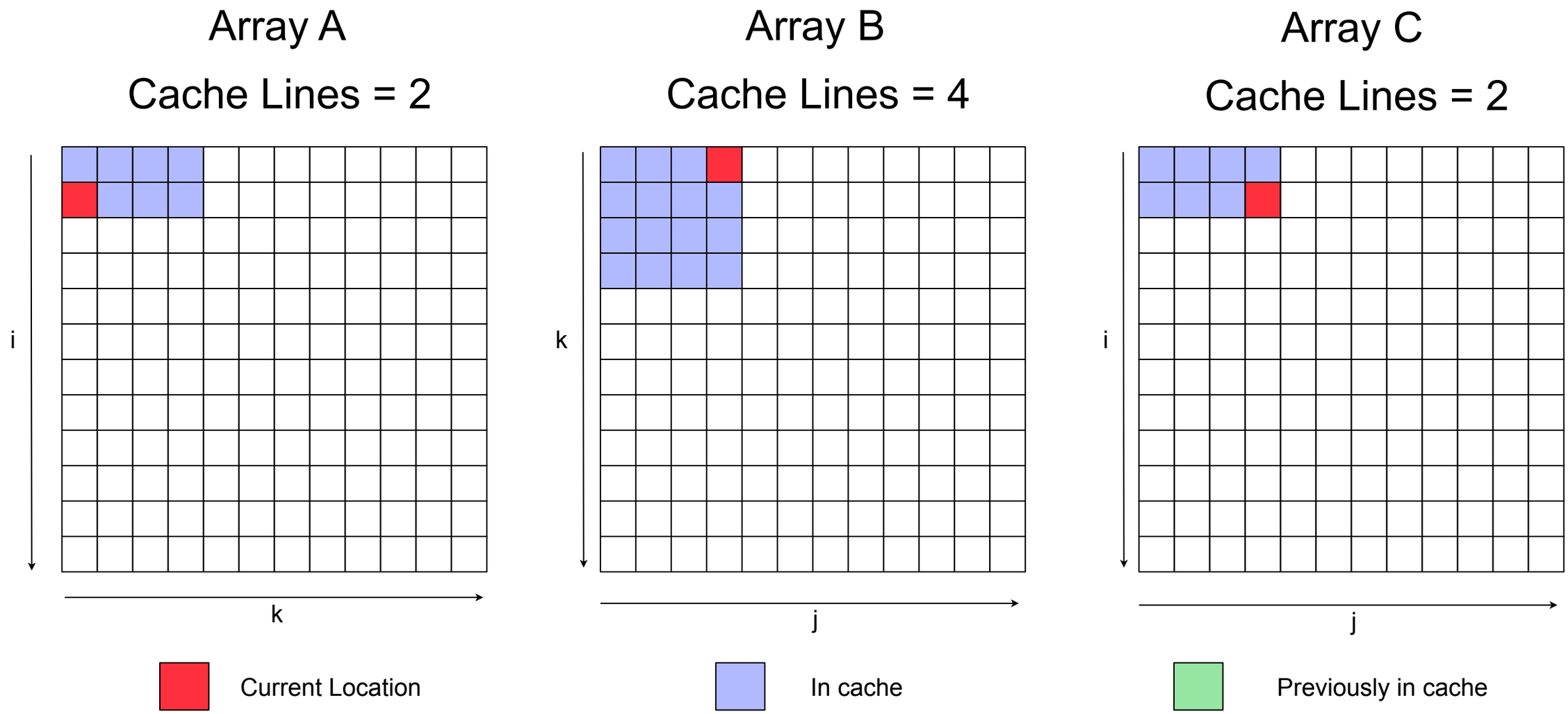
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



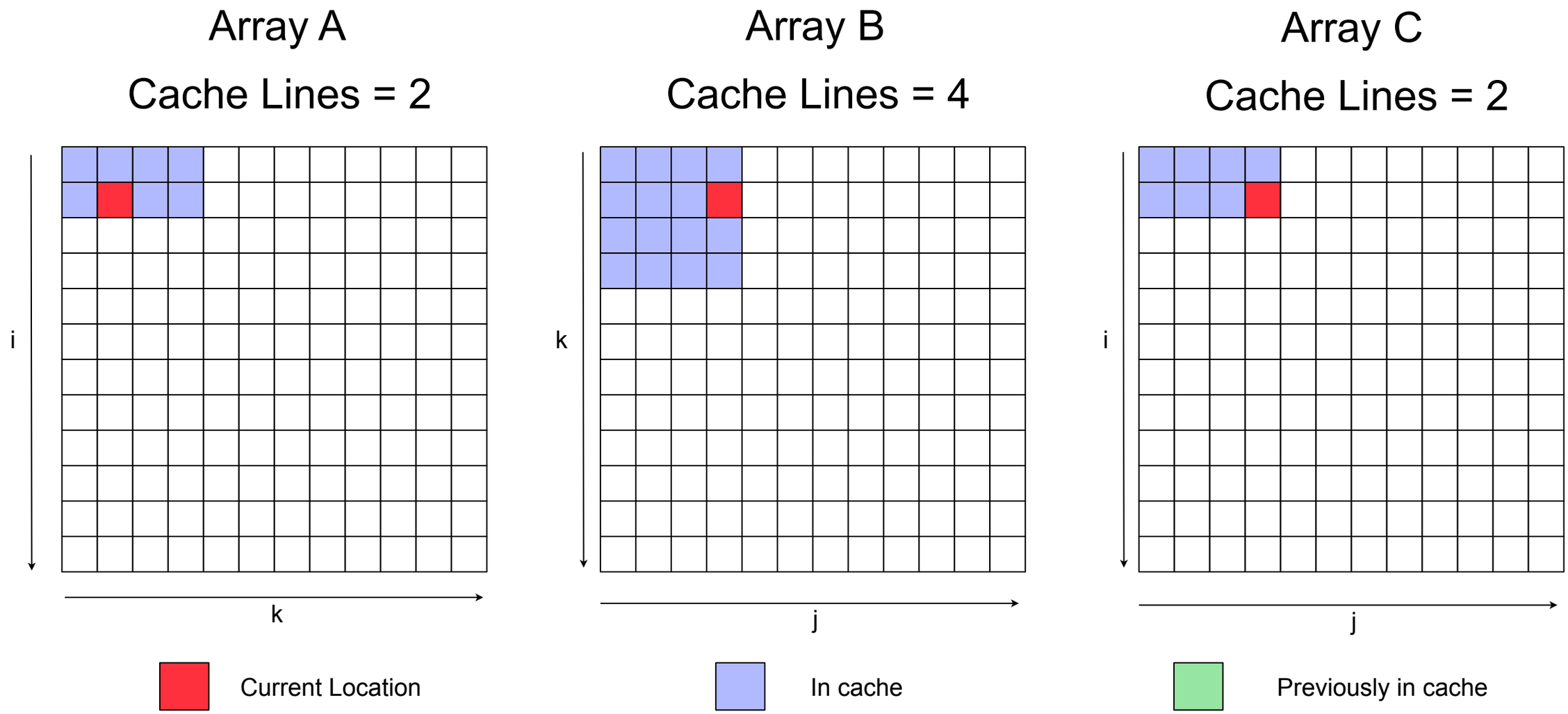
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



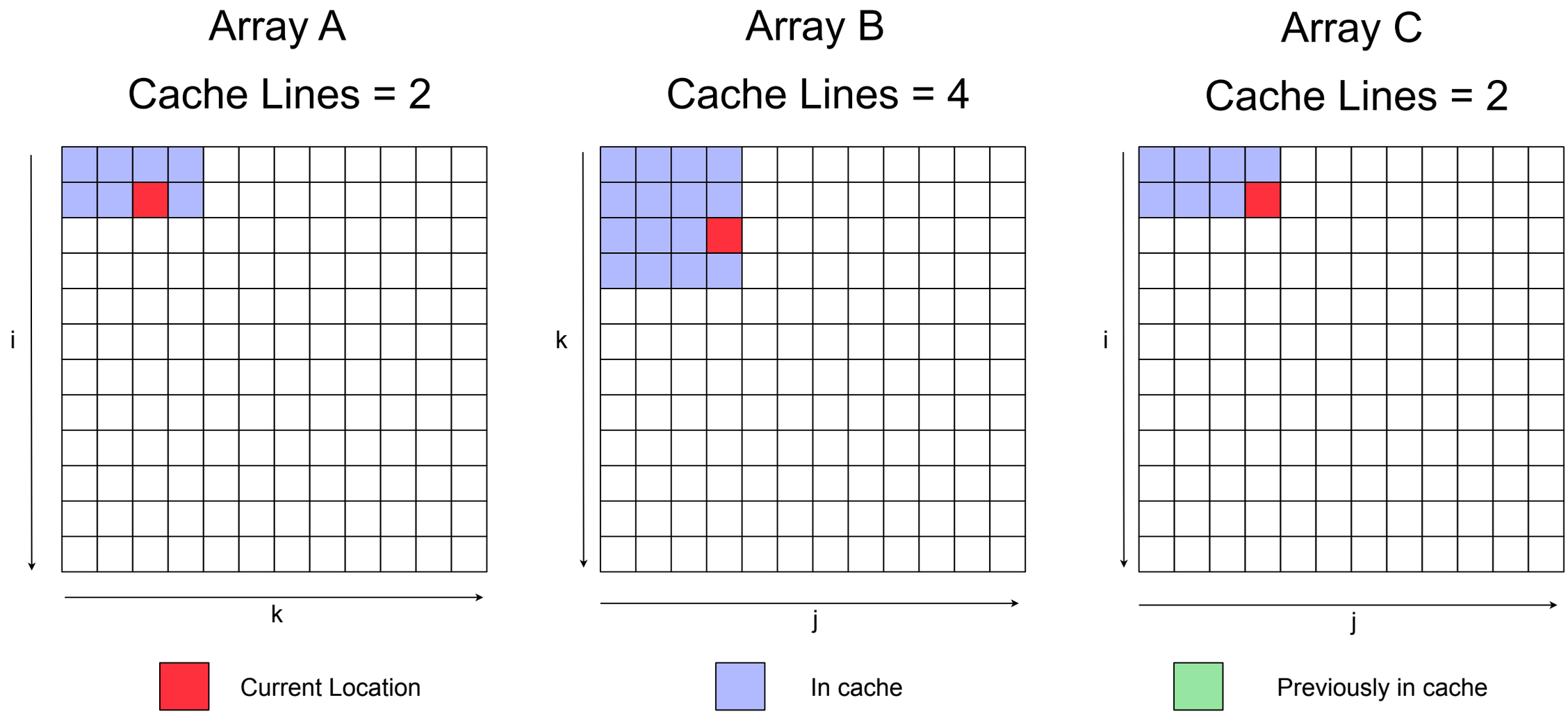
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



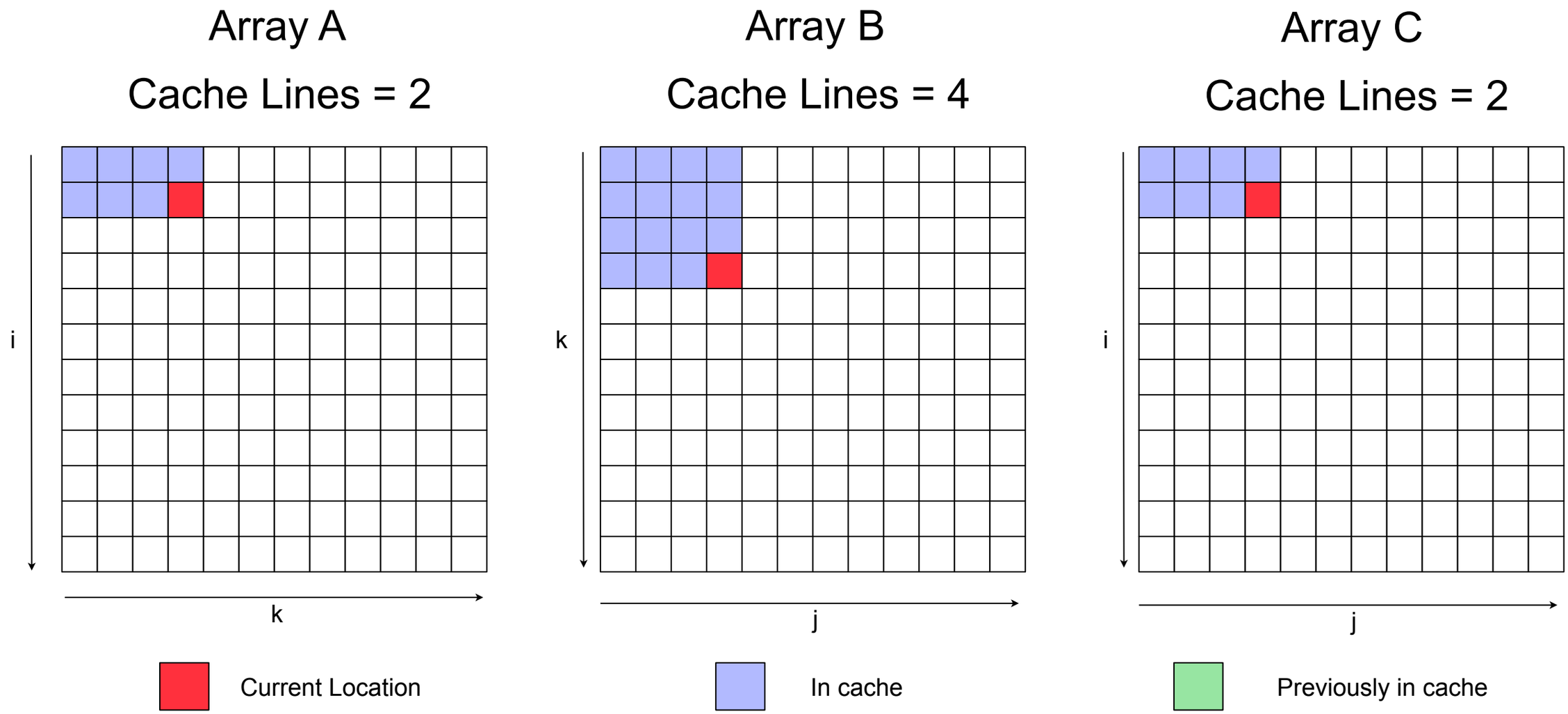
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



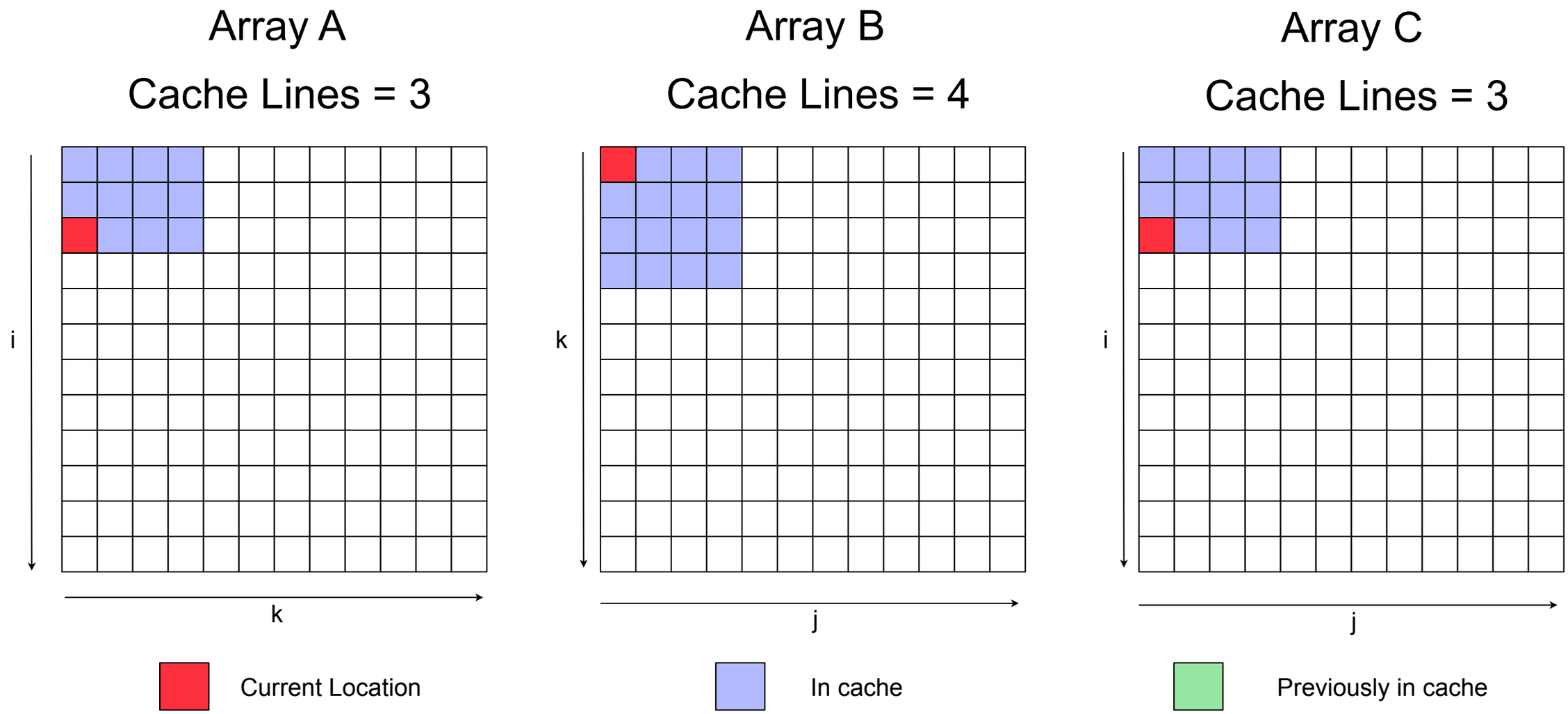
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



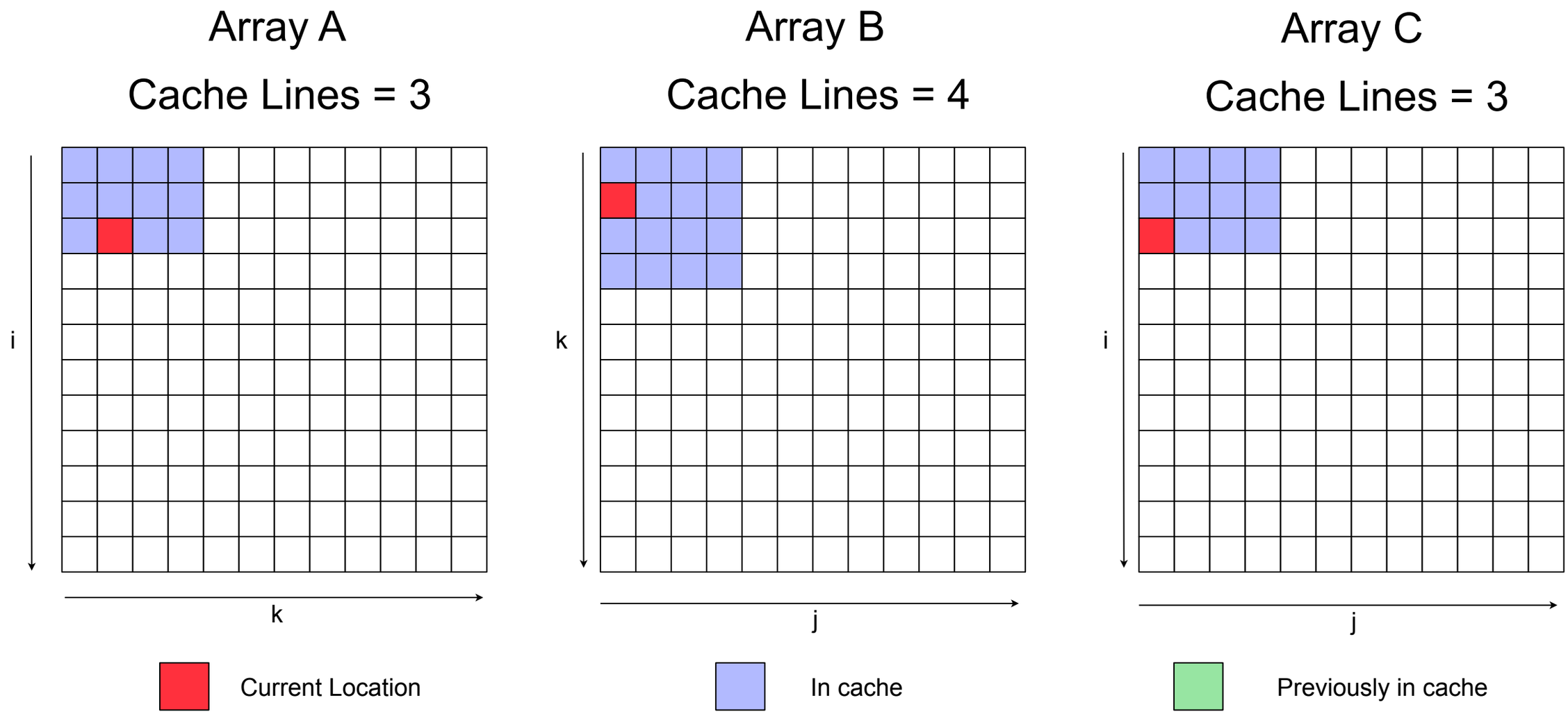
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



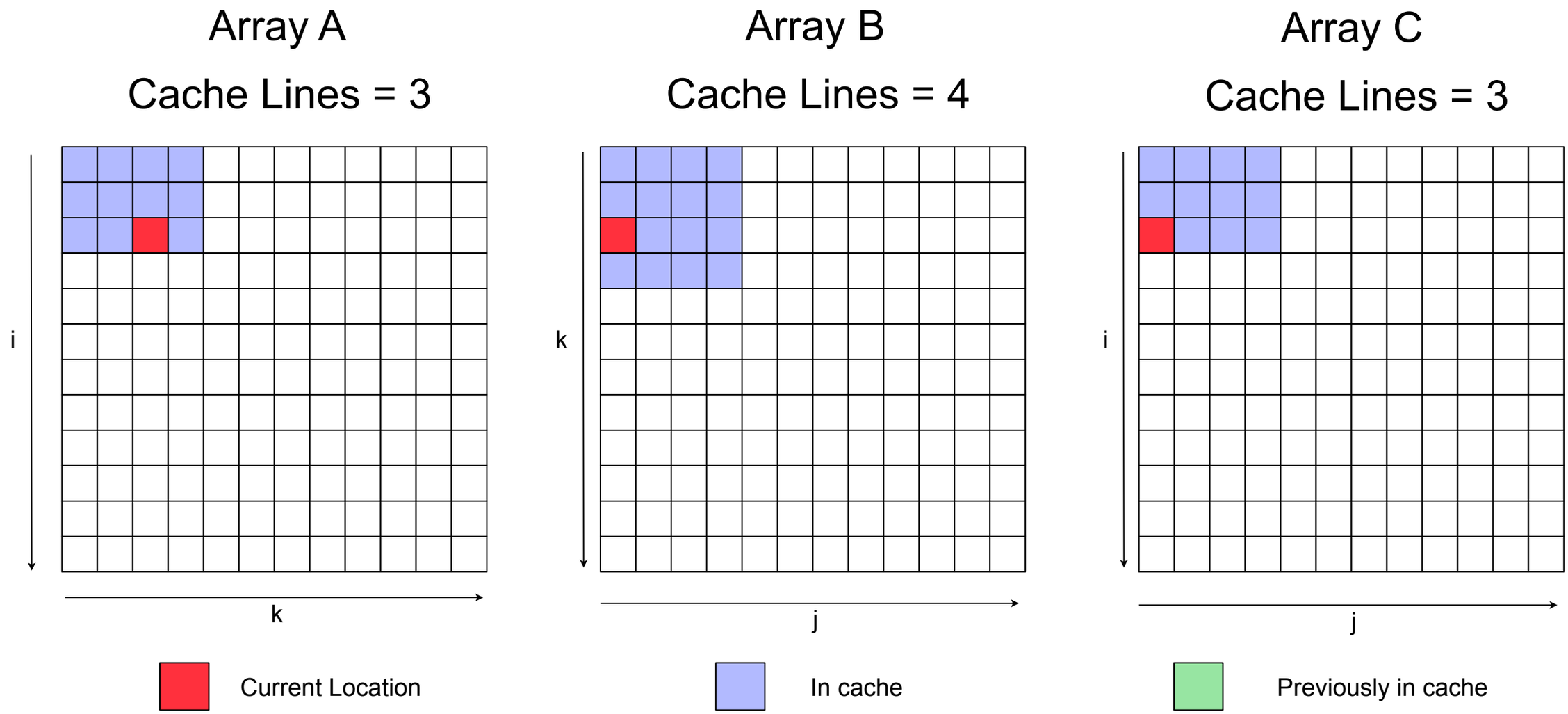
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



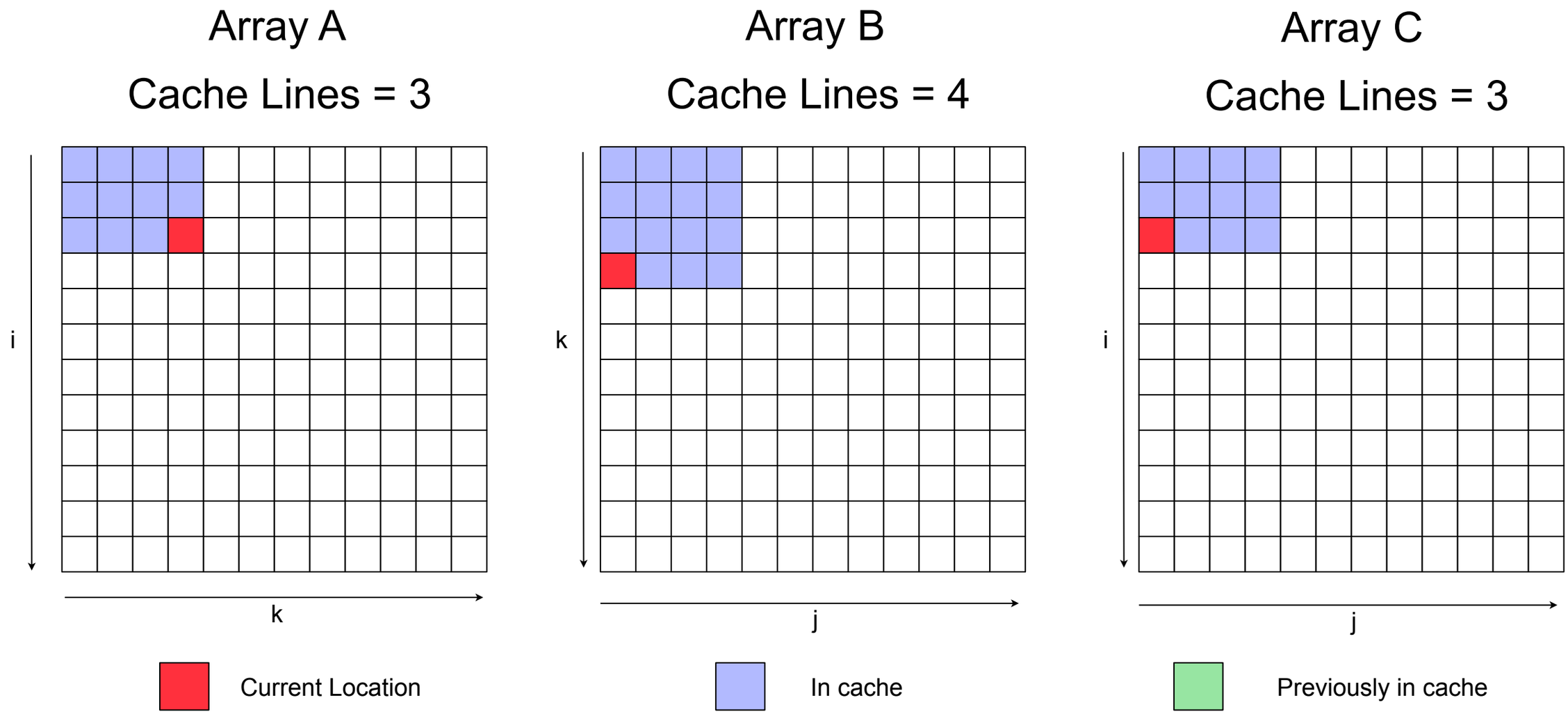
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



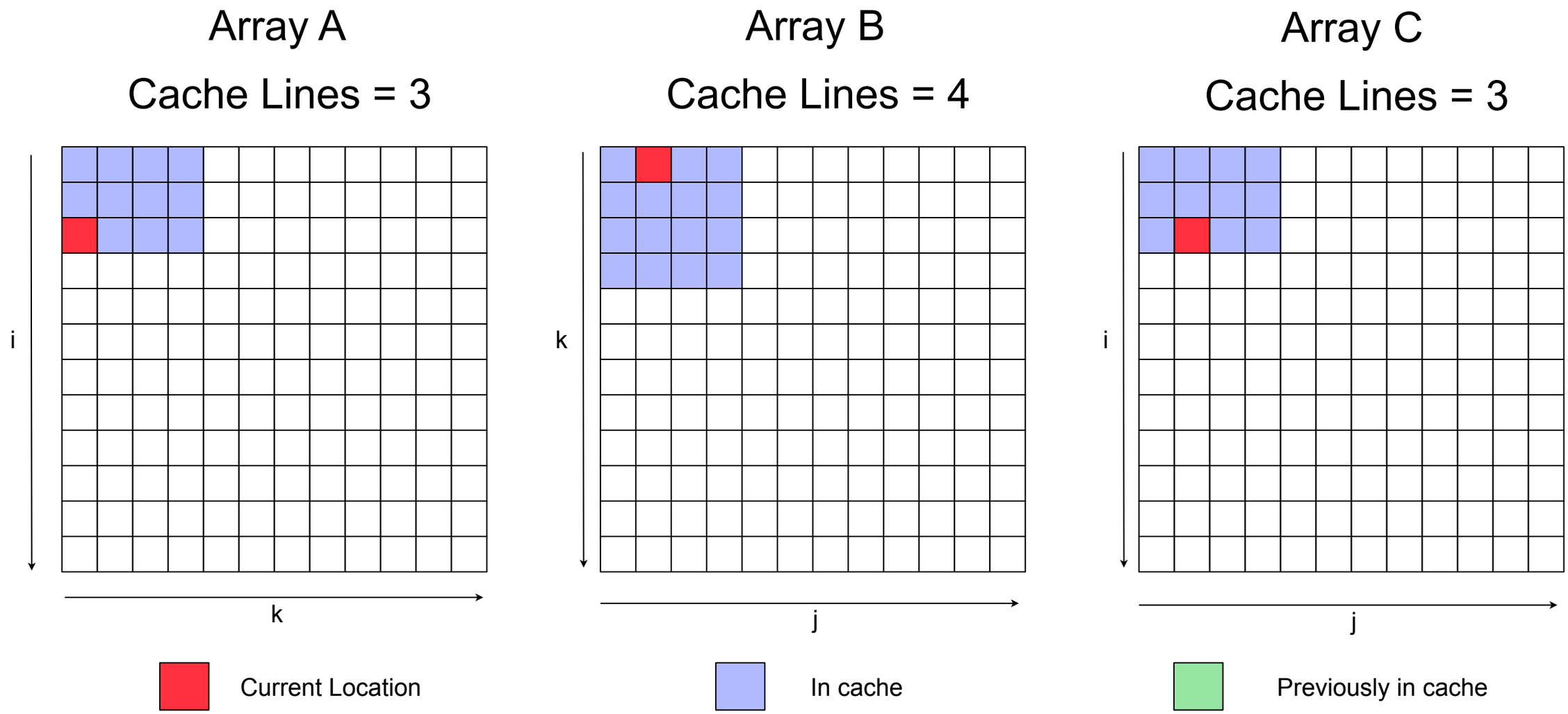
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



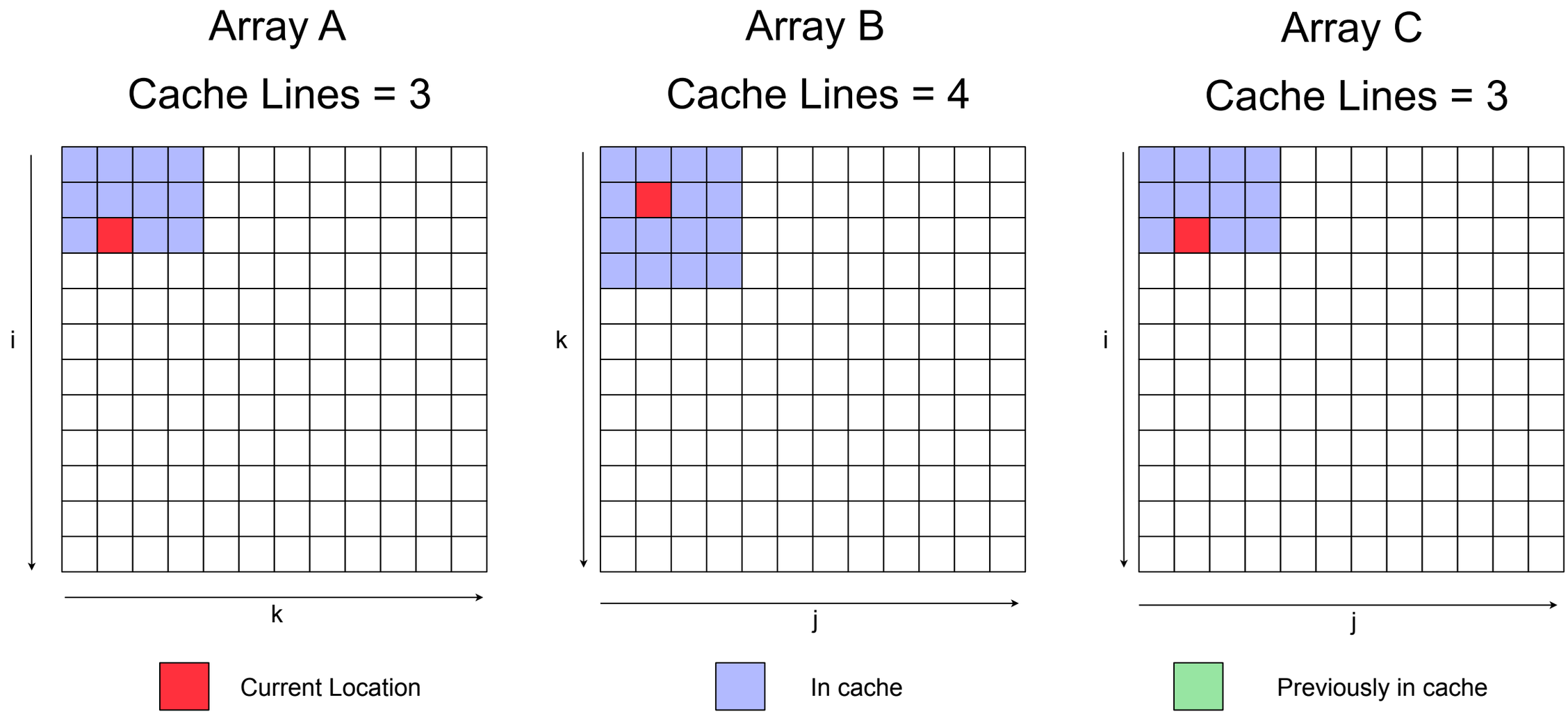
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



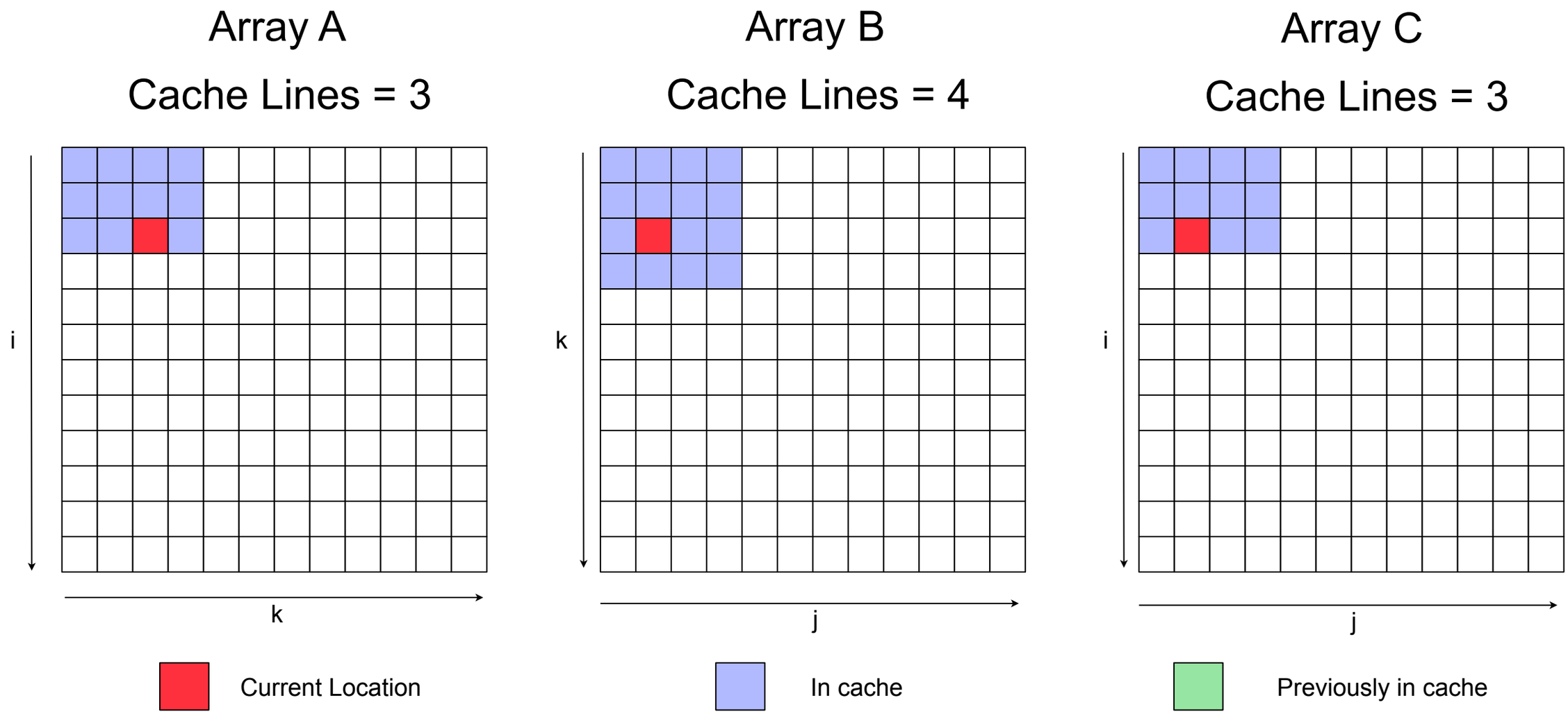
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



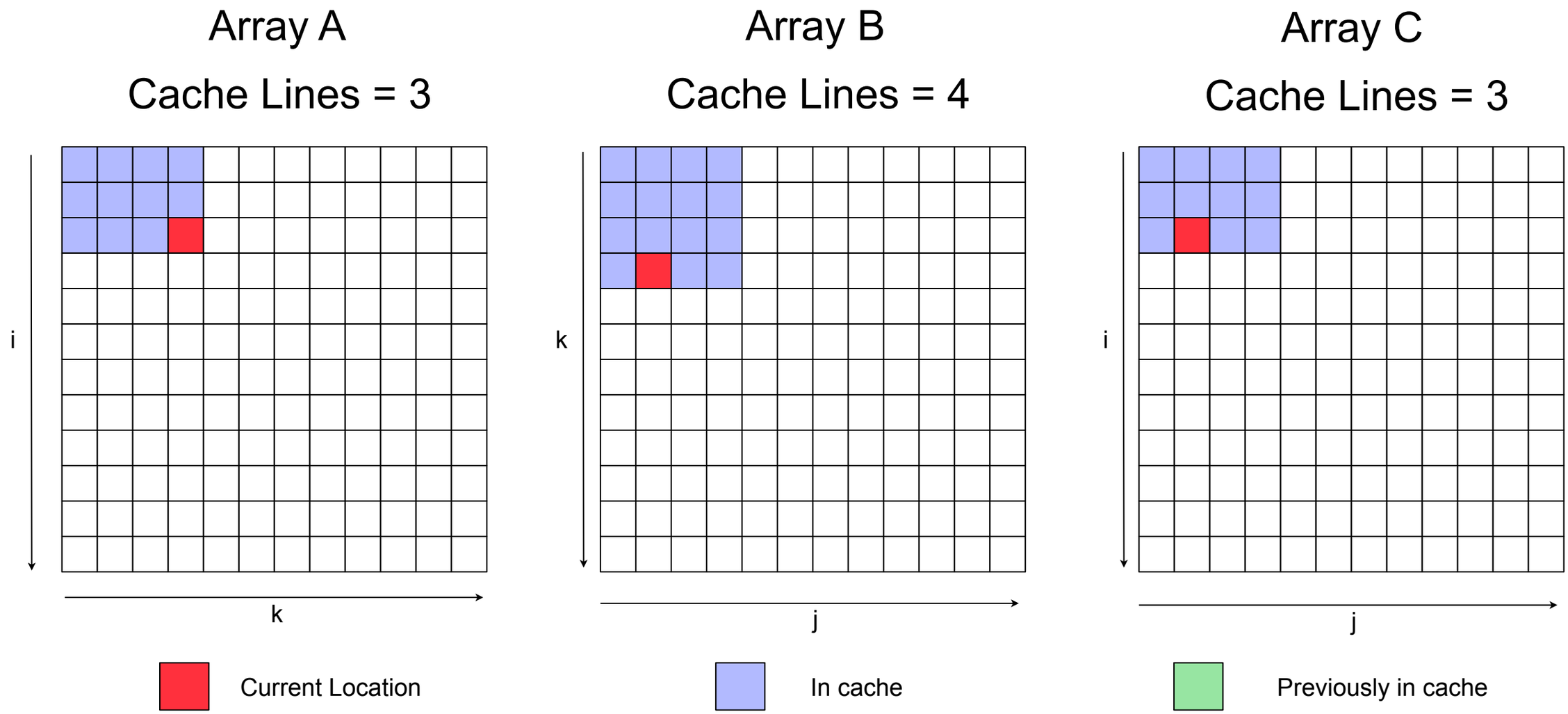
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4

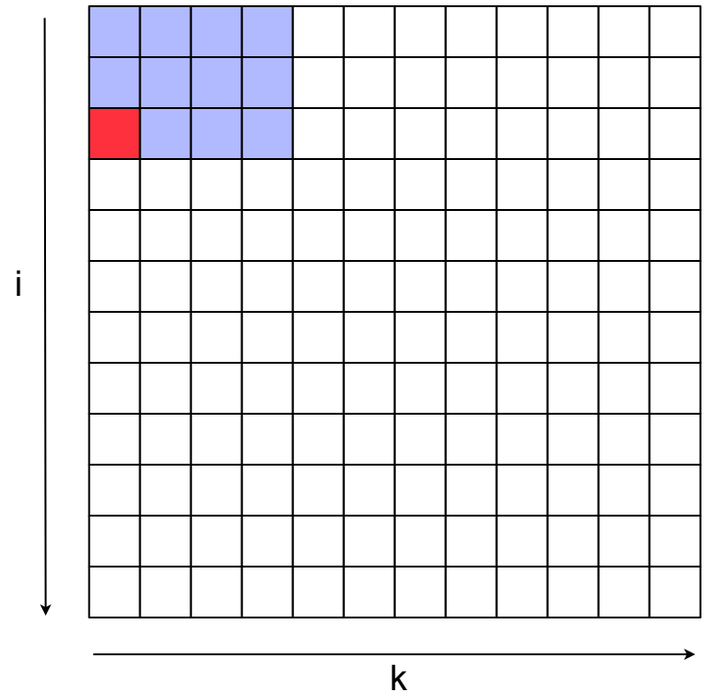


Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4

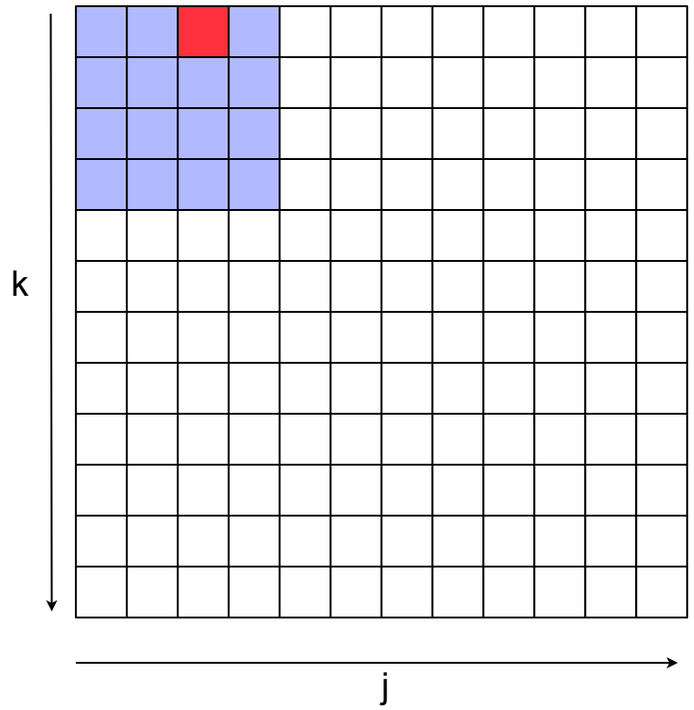
Array A

Cache Lines = 3



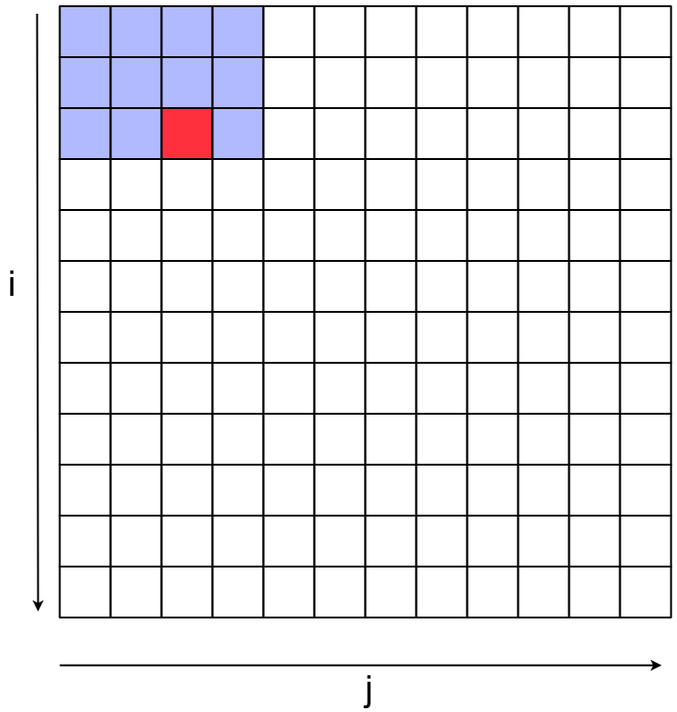
Array B

Cache Lines = 4



Array C

Cache Lines = 3



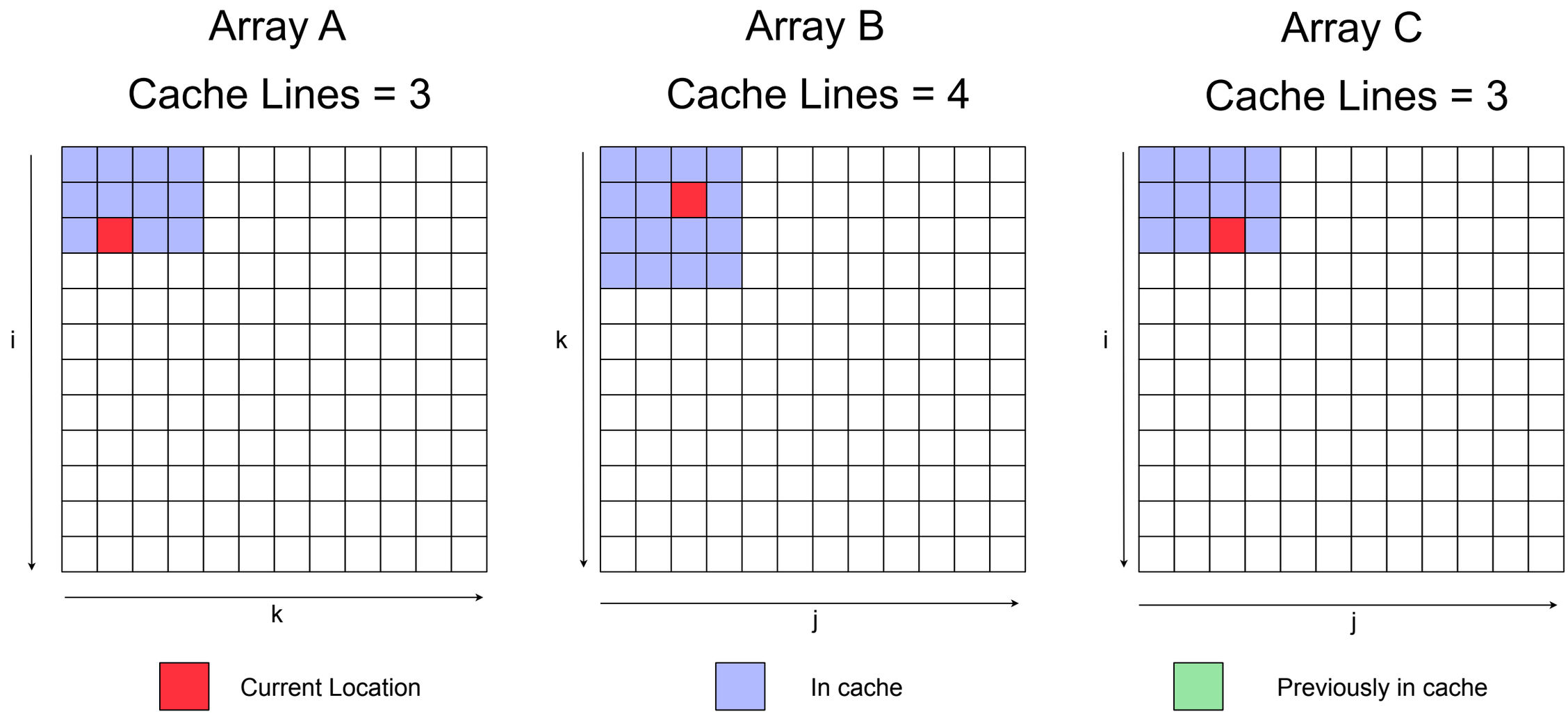
Current Location

In cache

Previously in cache

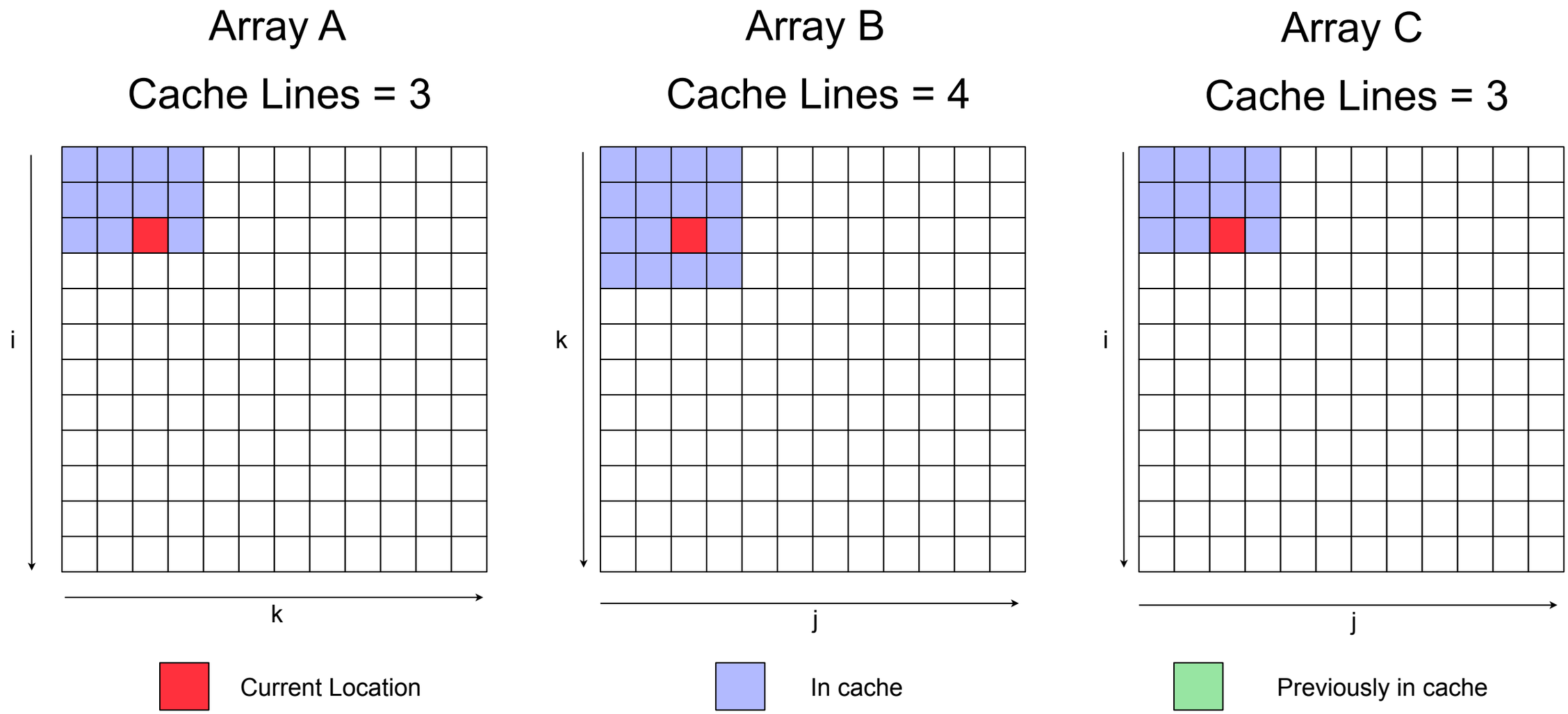
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



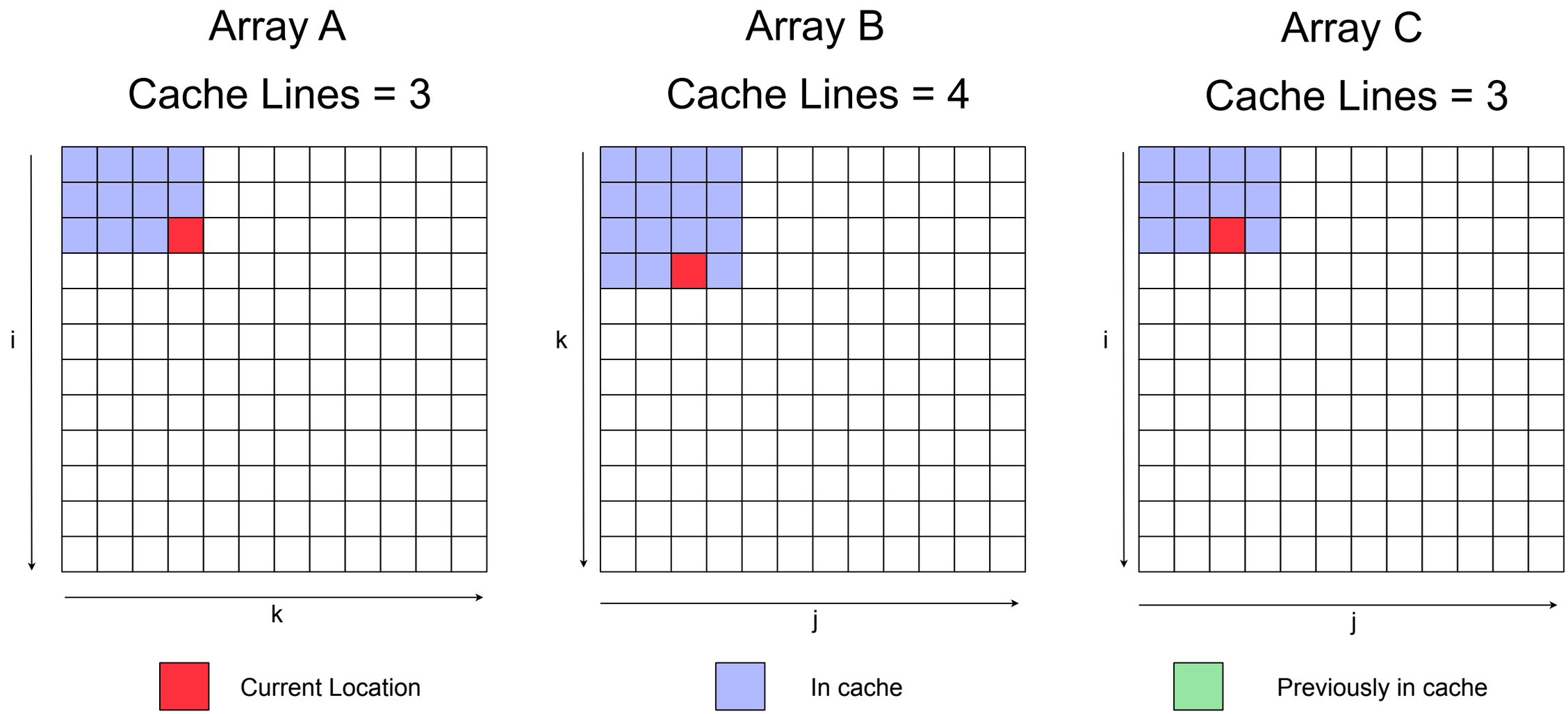
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4

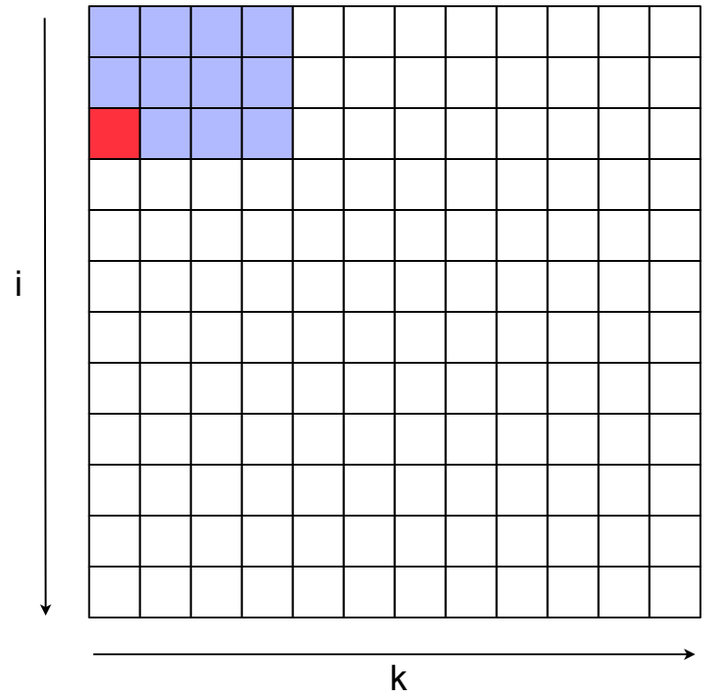


Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4

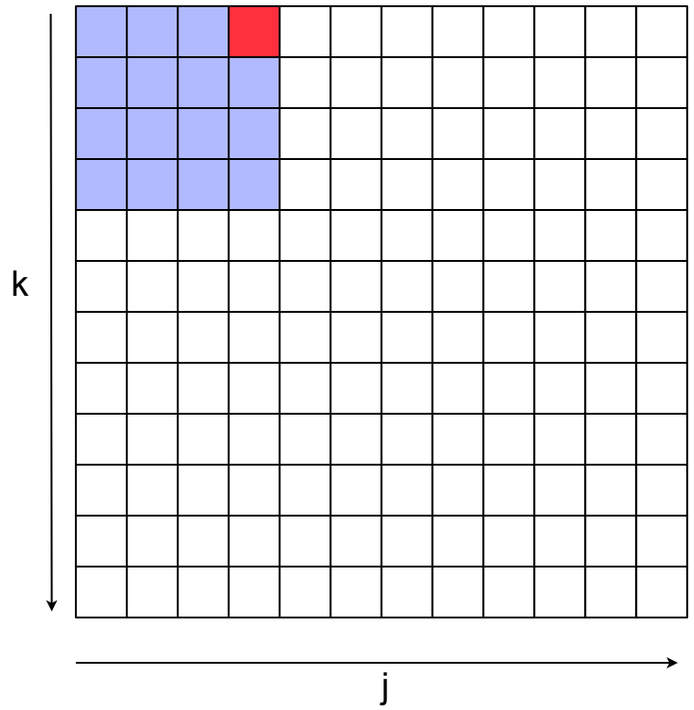
Array A

Cache Lines = 3



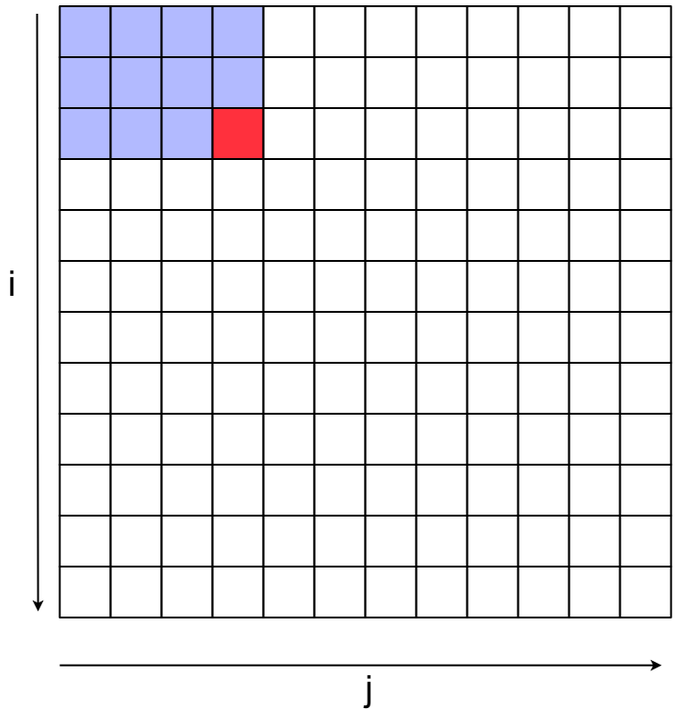
Array B

Cache Lines = 4



Array C

Cache Lines = 3



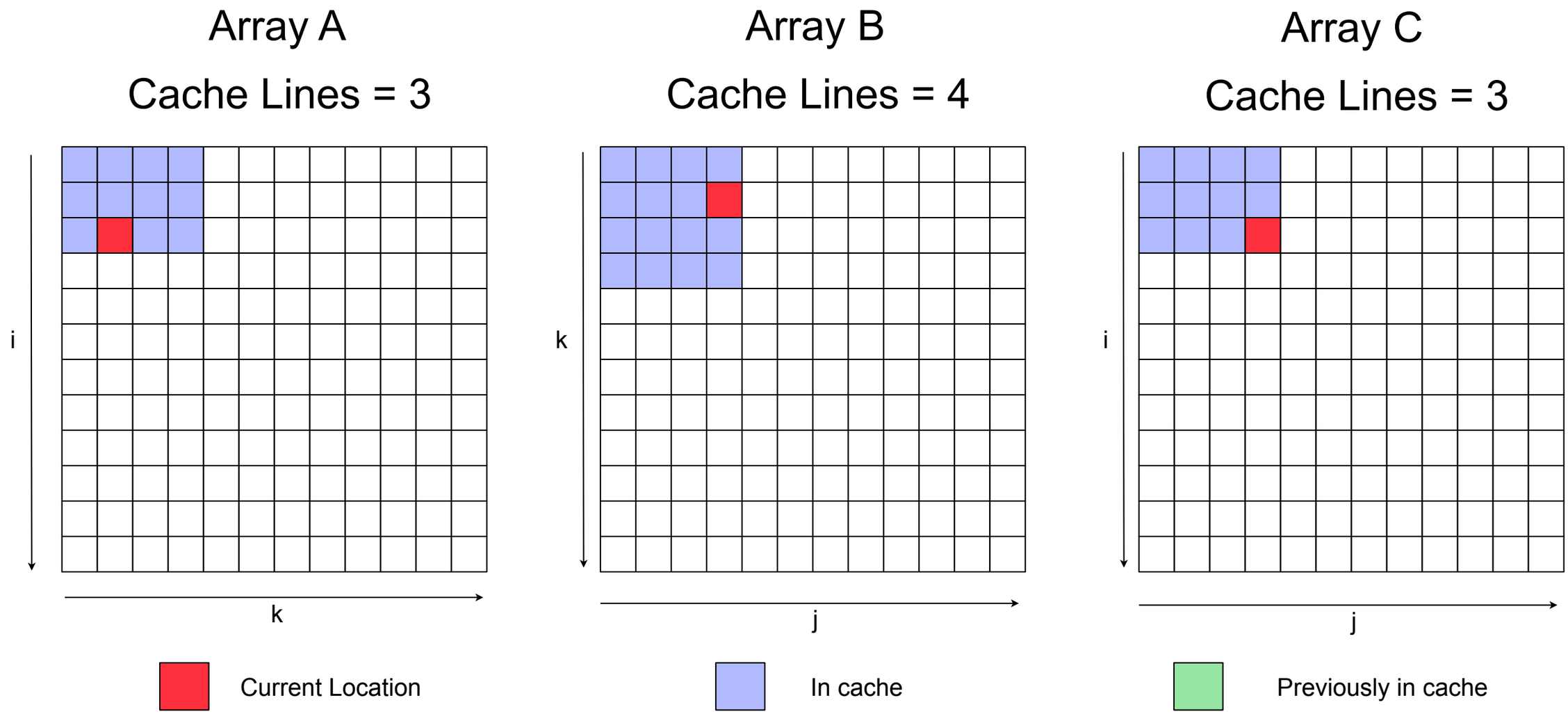
Current Location

In cache

Previously in cache

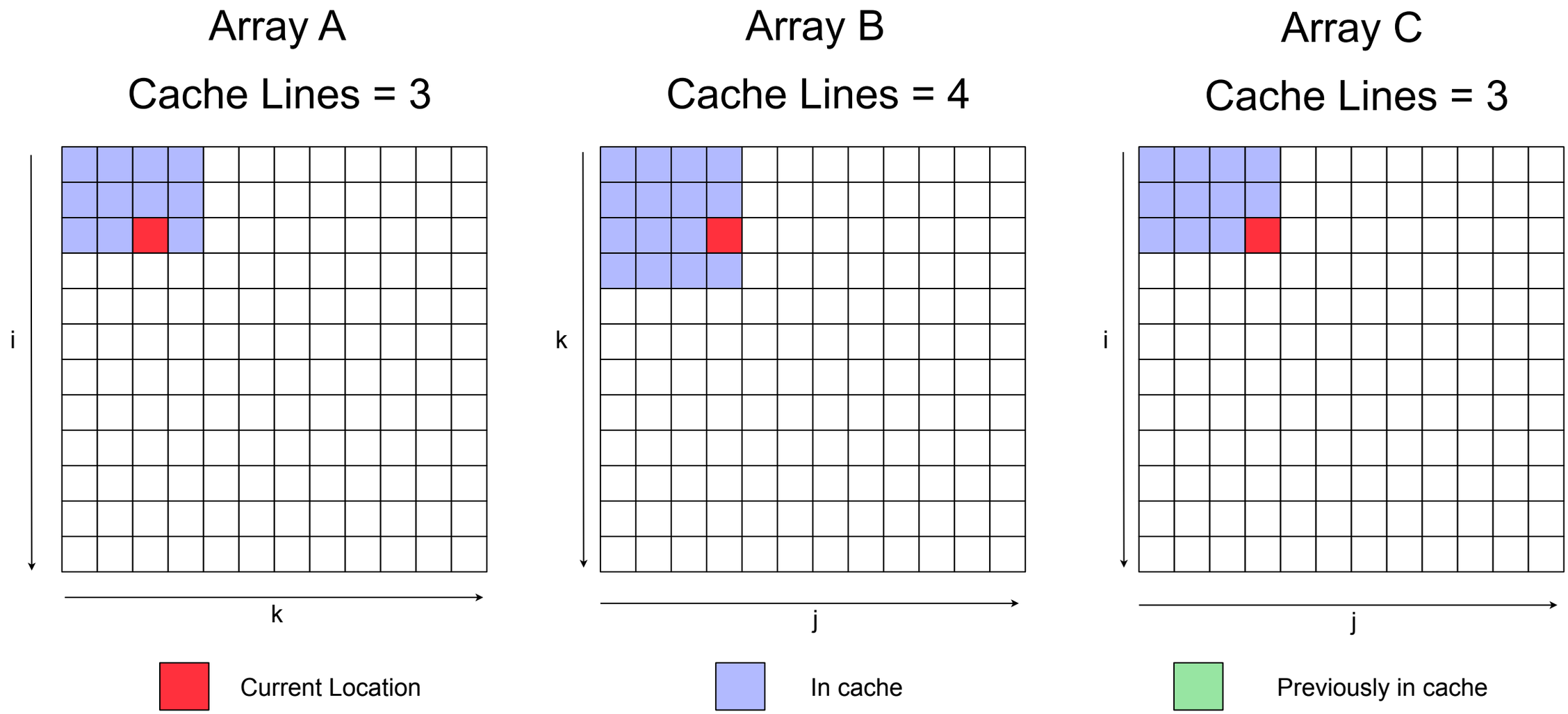
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



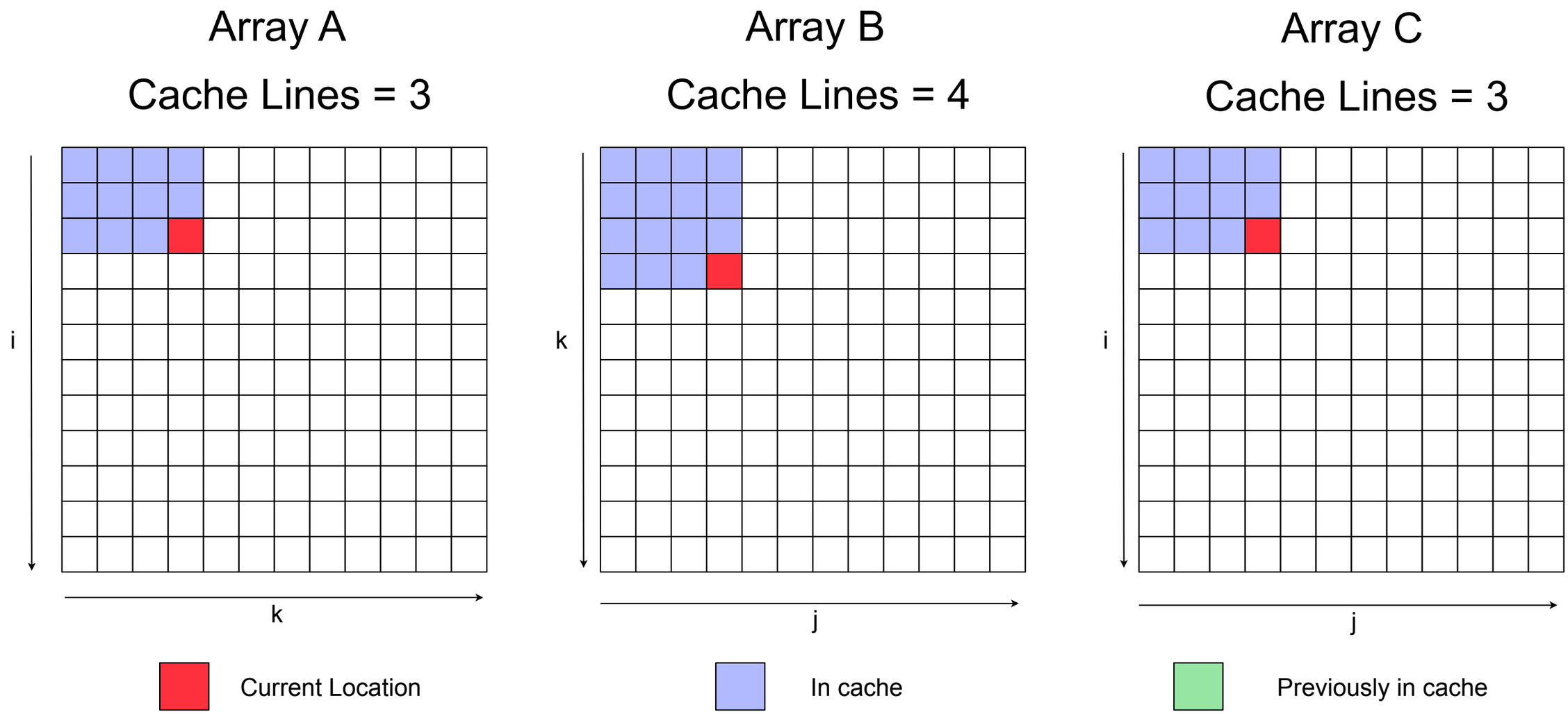
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



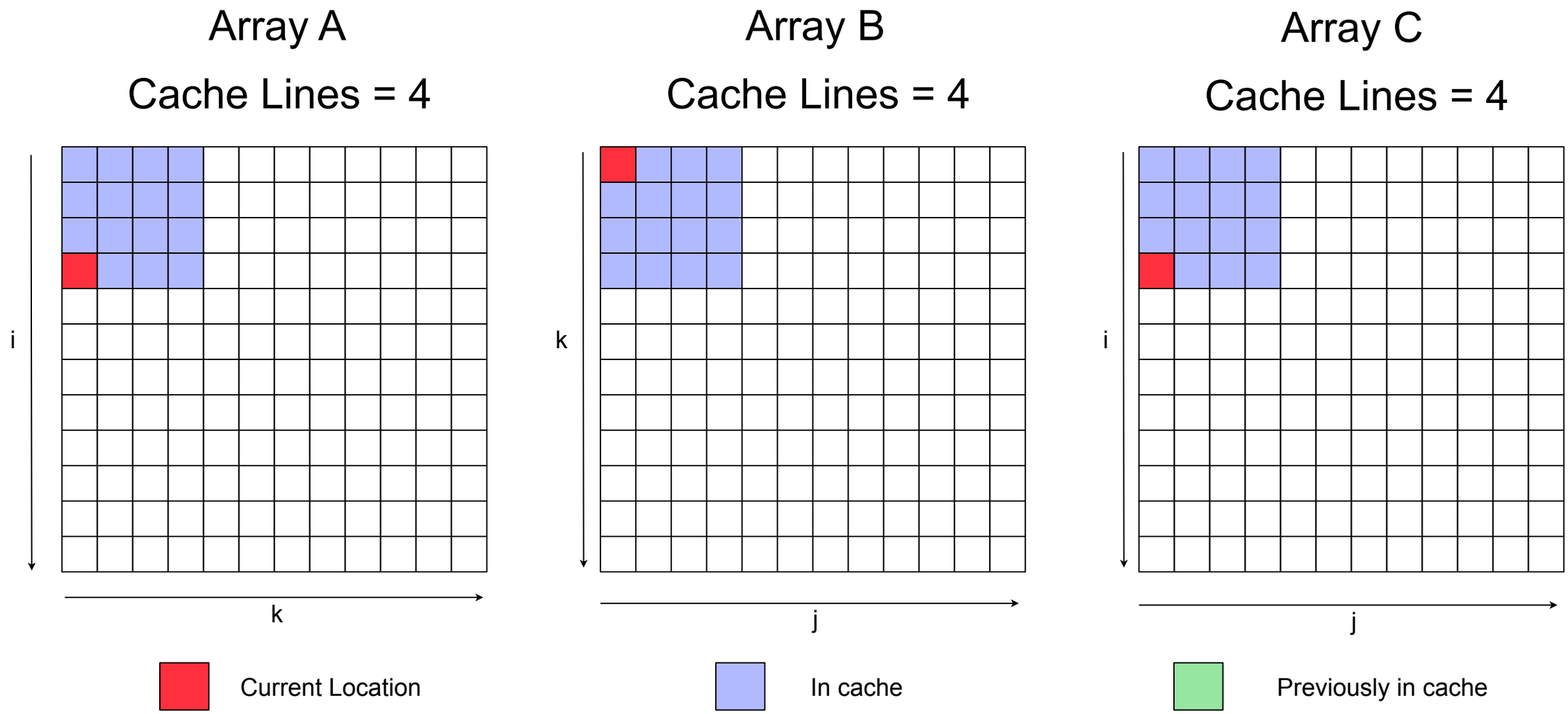
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



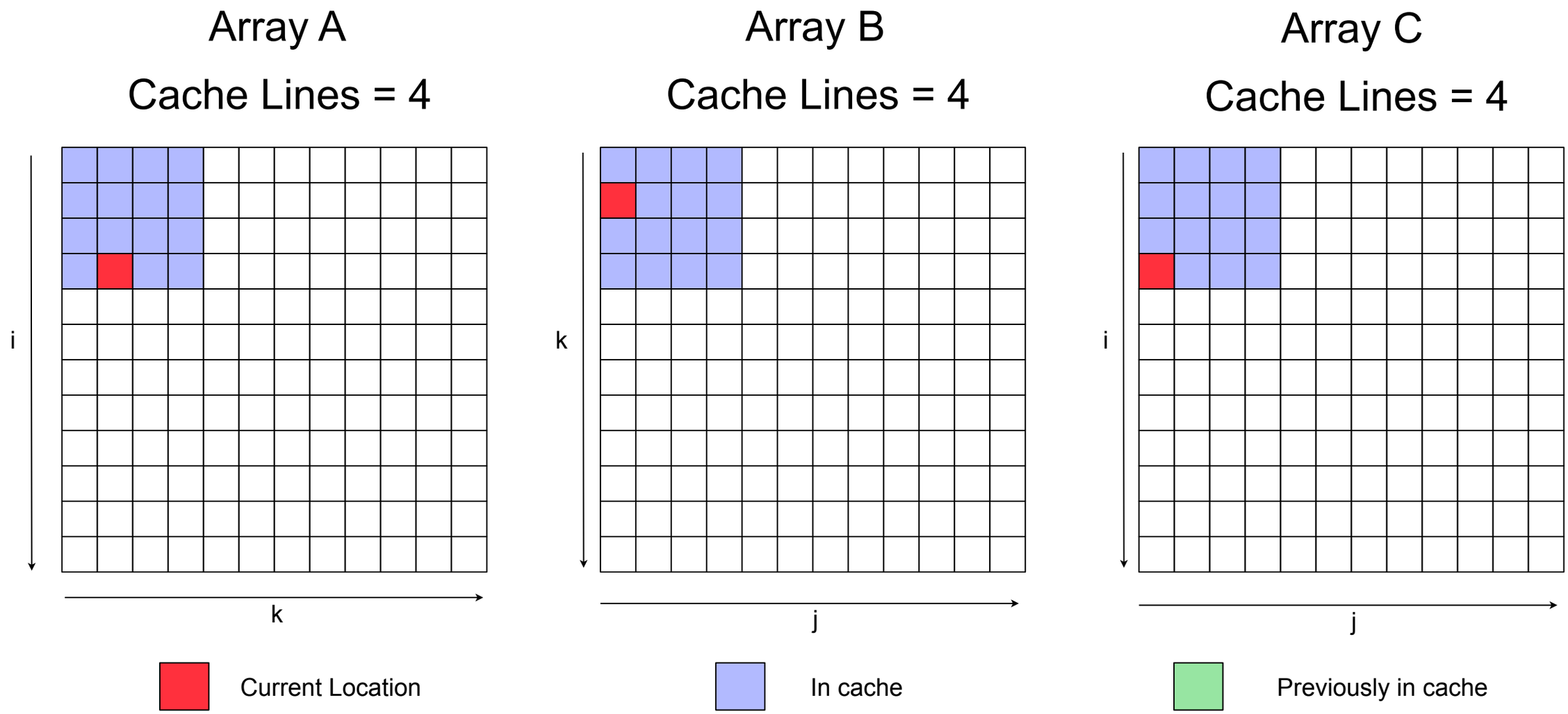
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



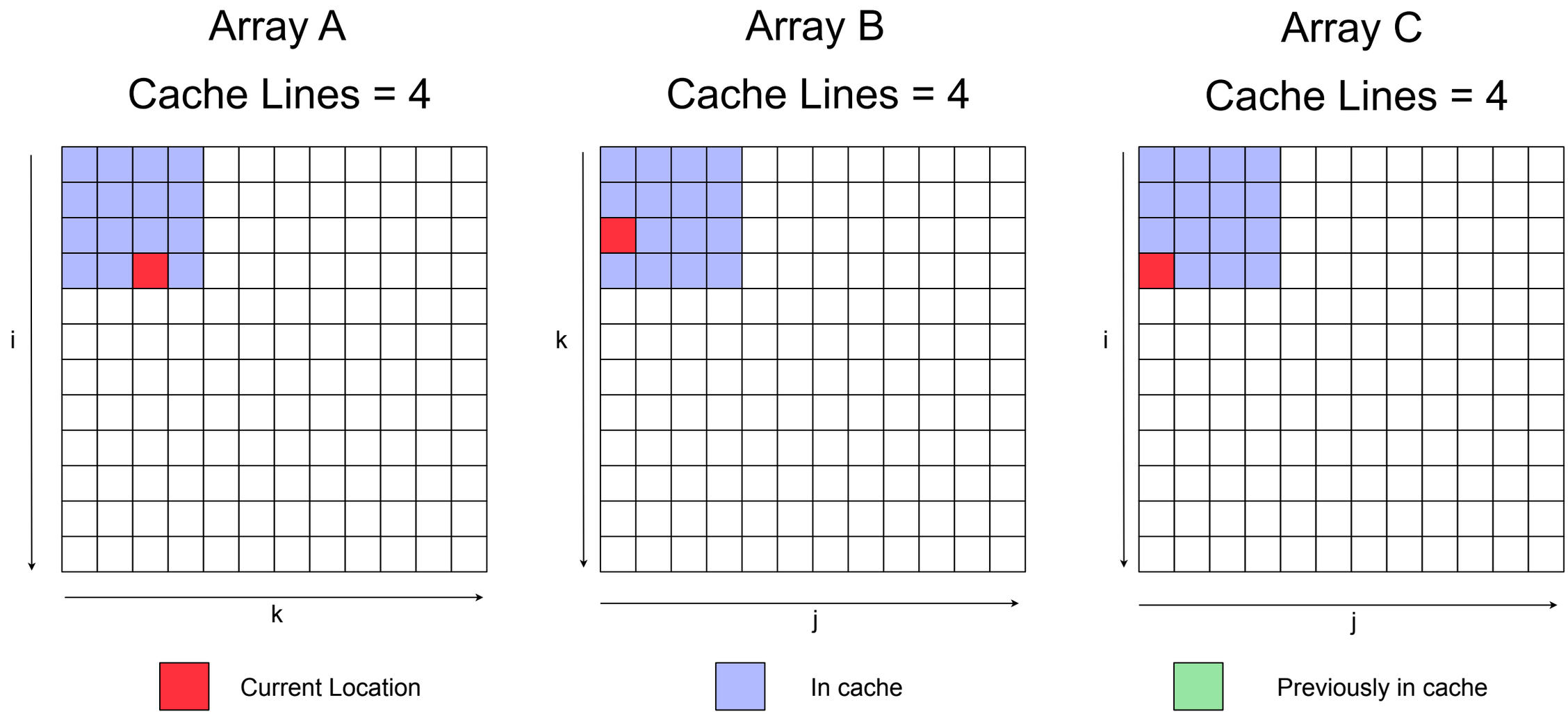
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



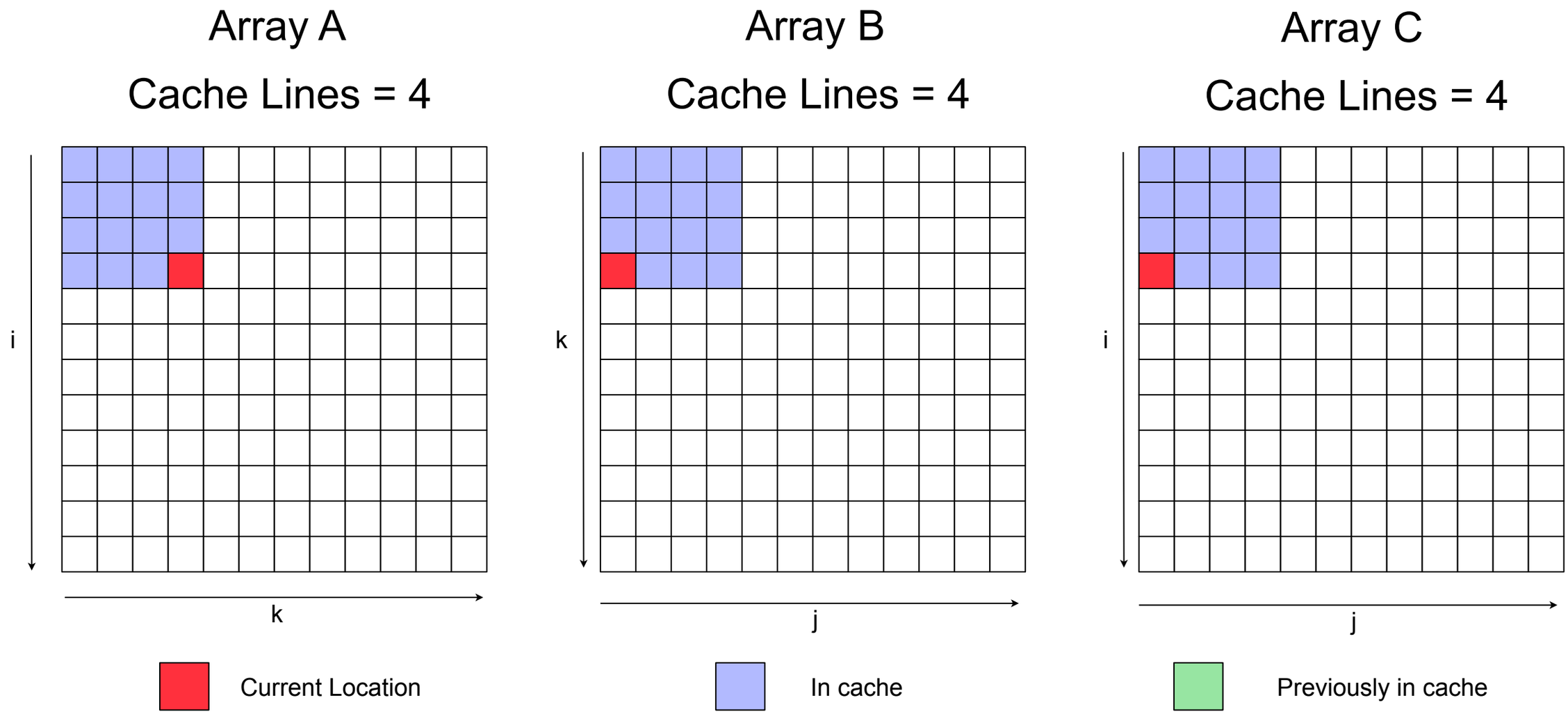
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



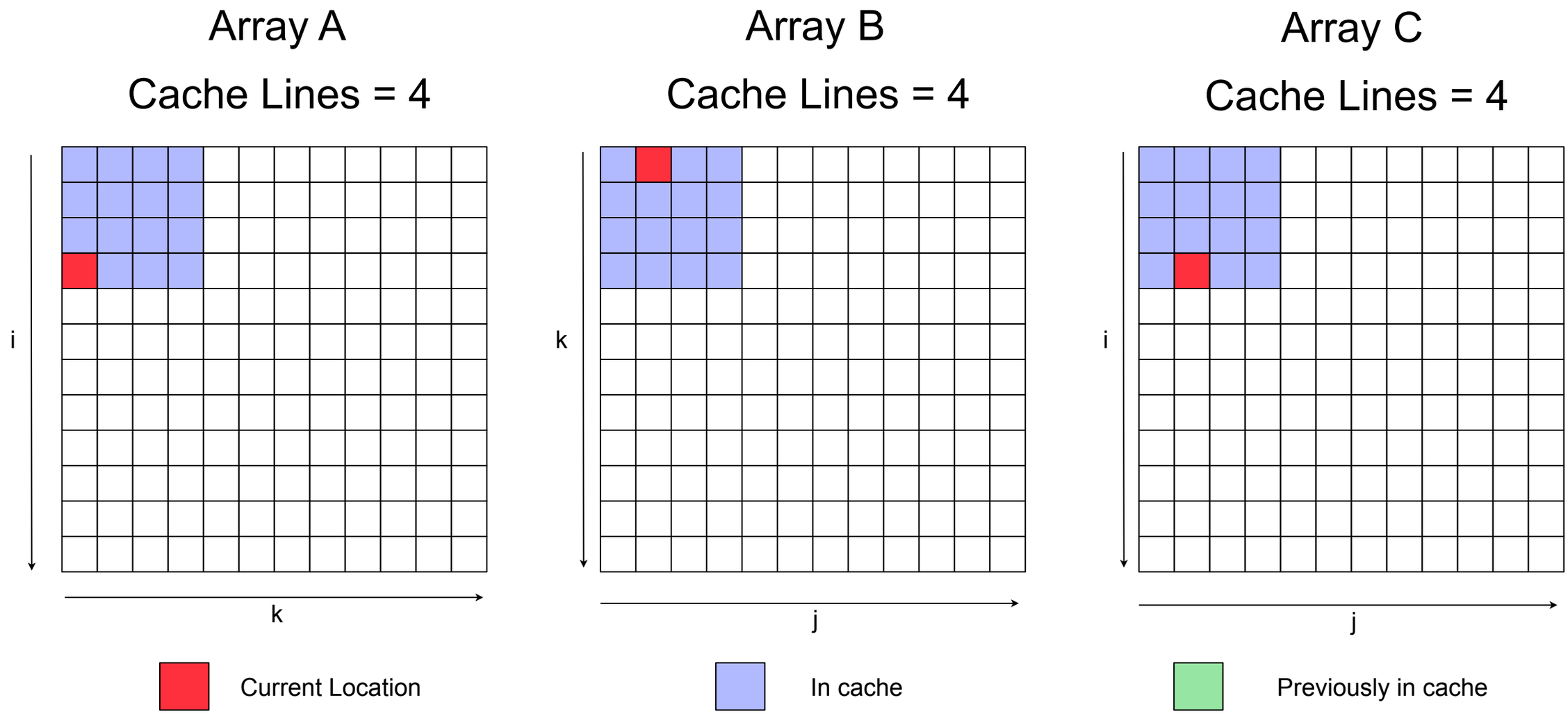
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



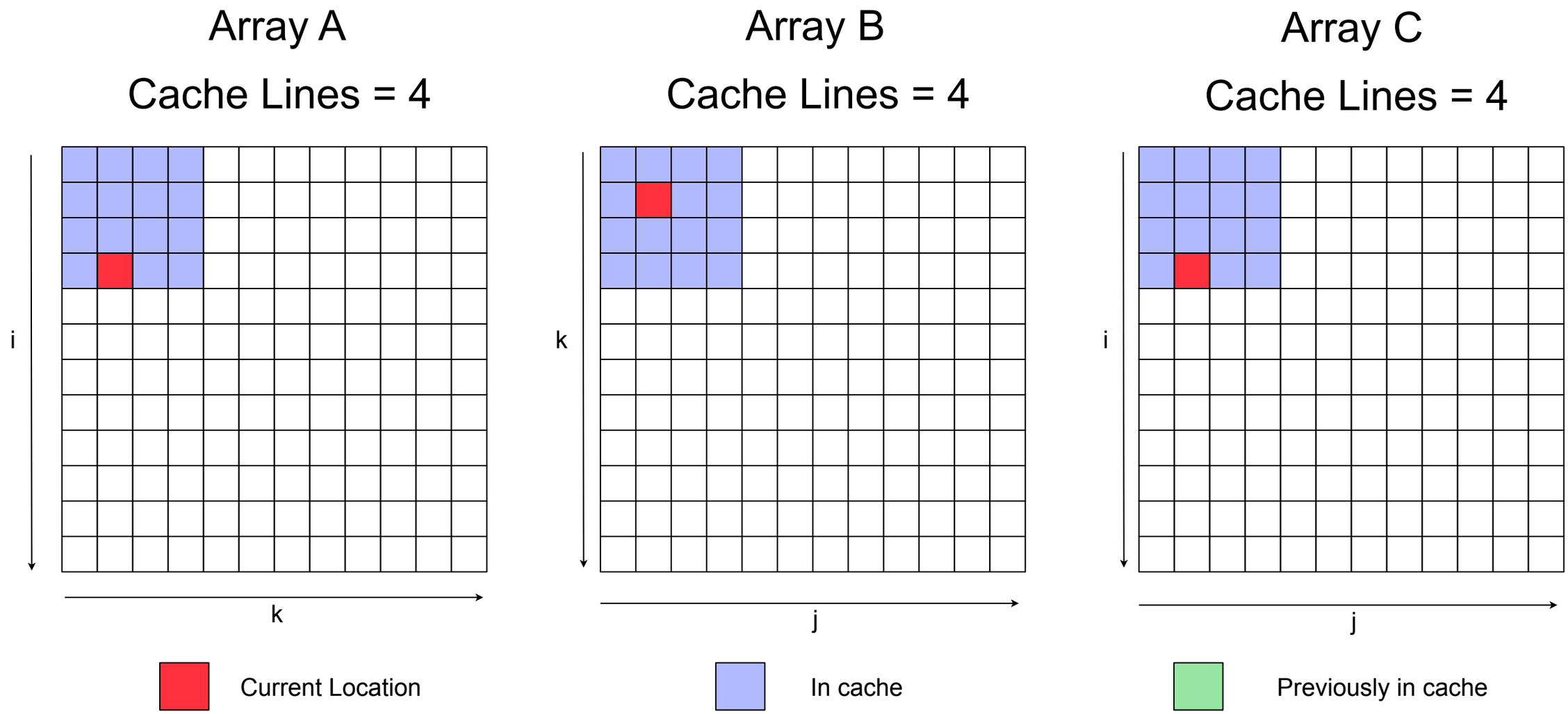
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



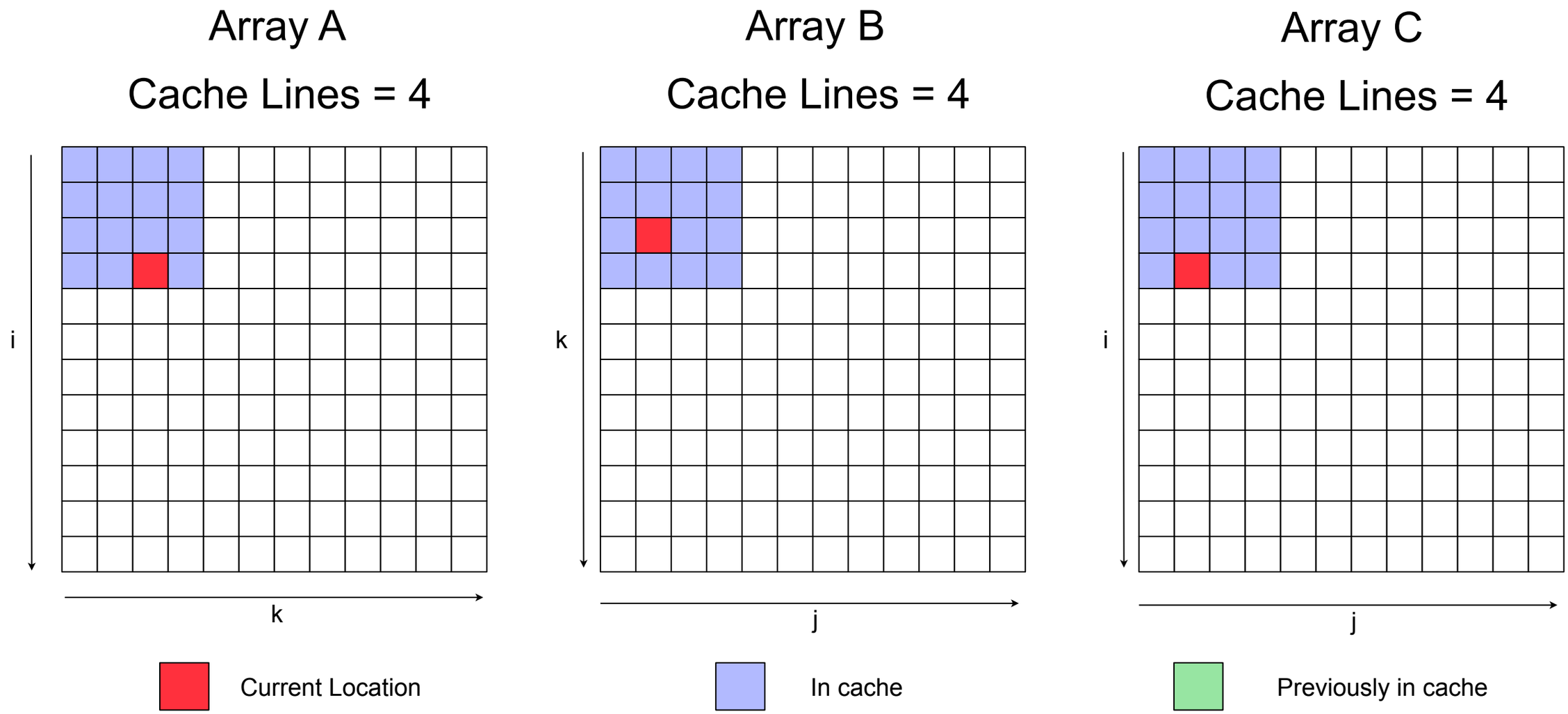
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



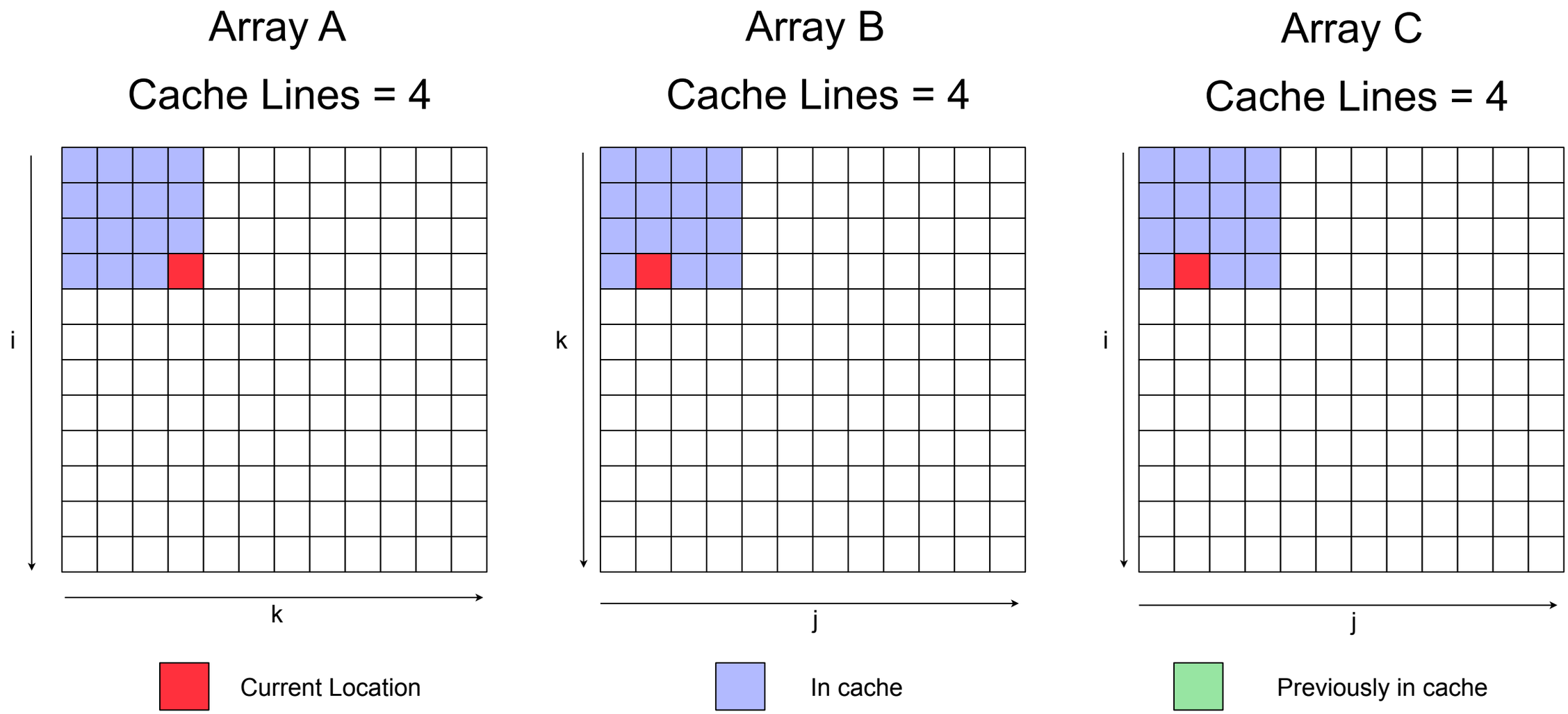
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



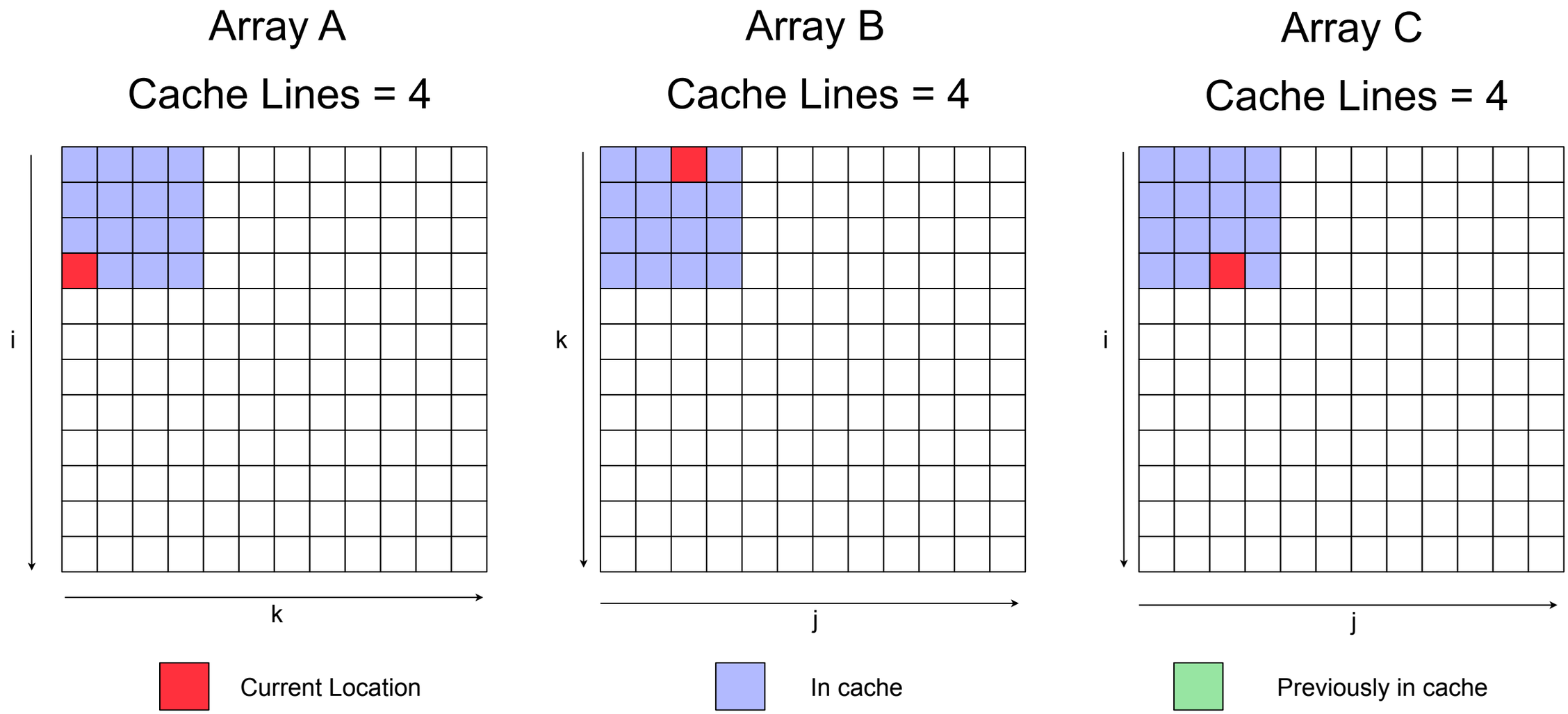
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



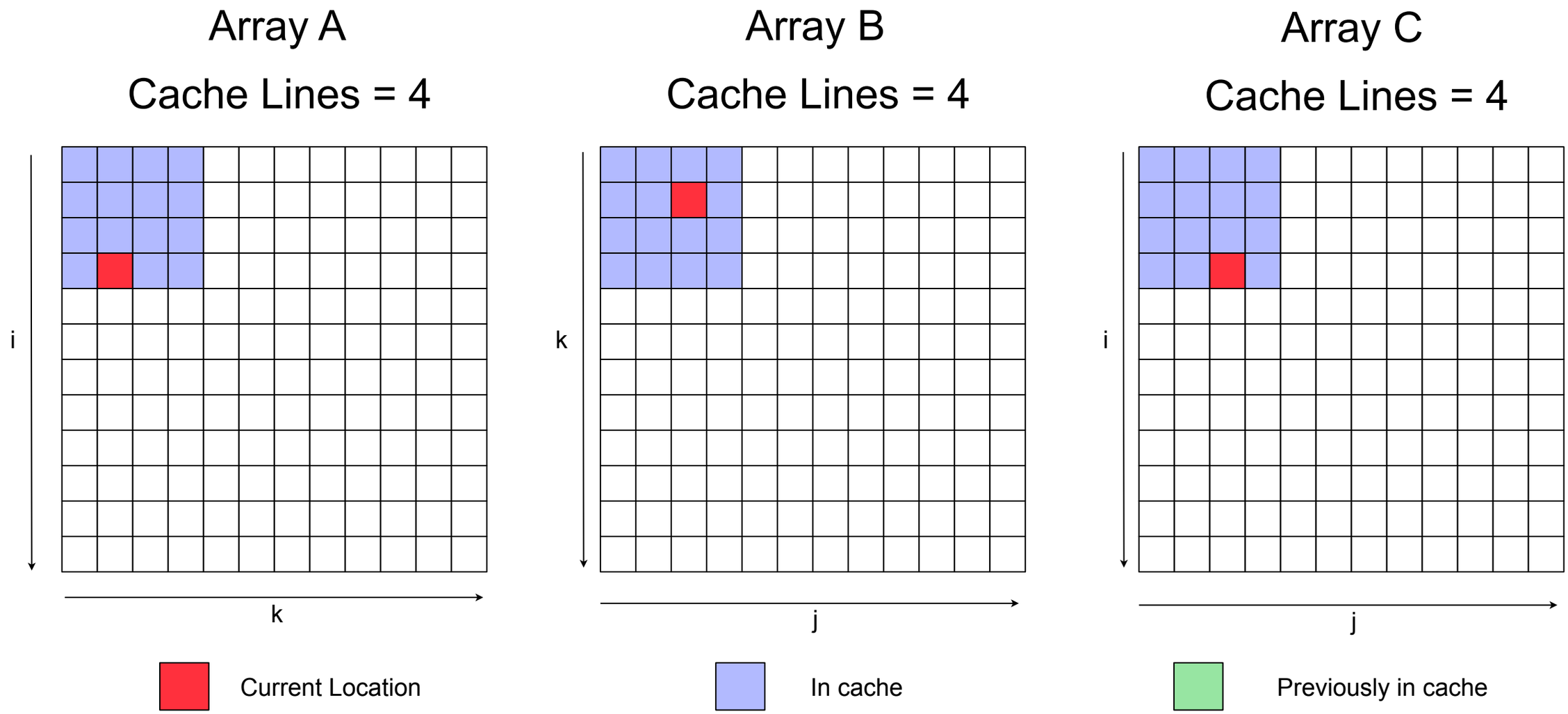
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



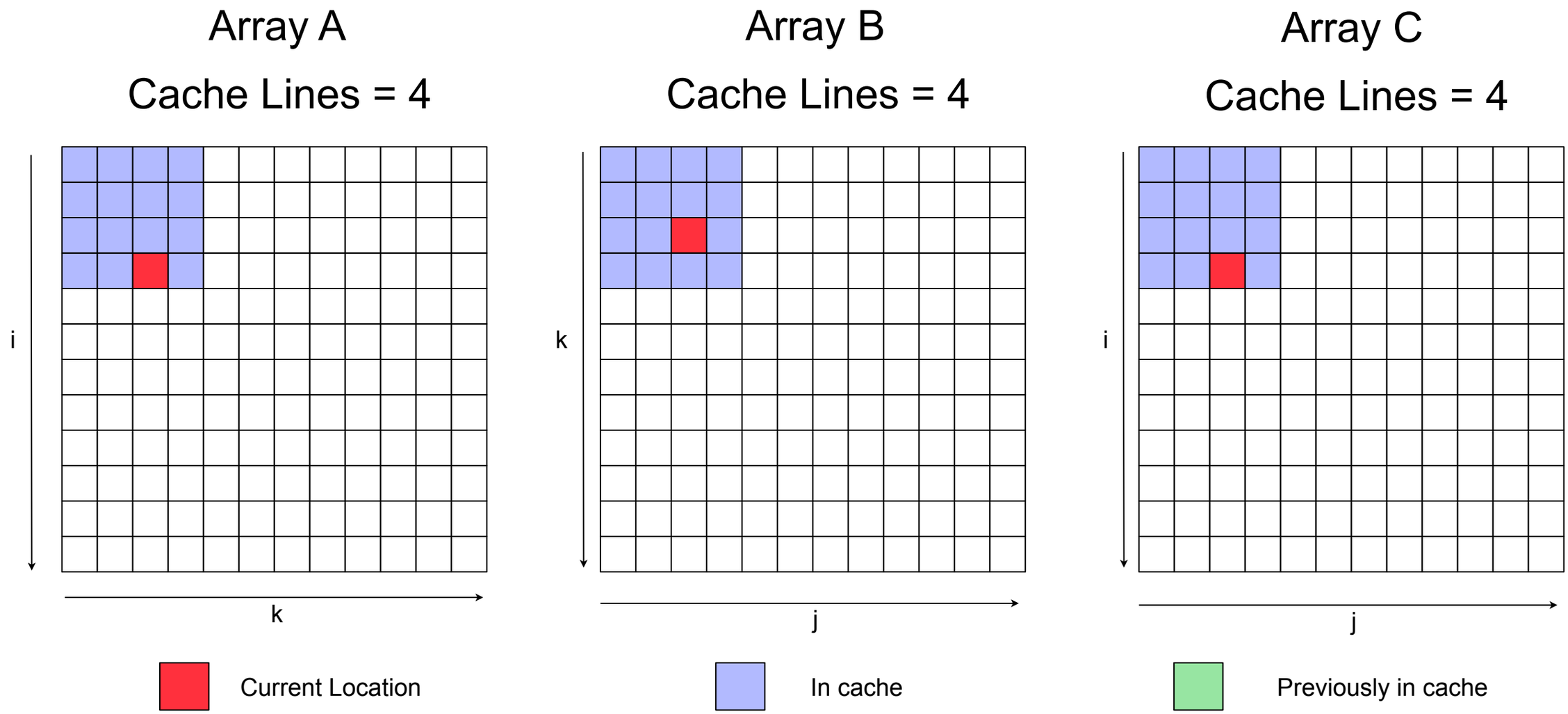
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



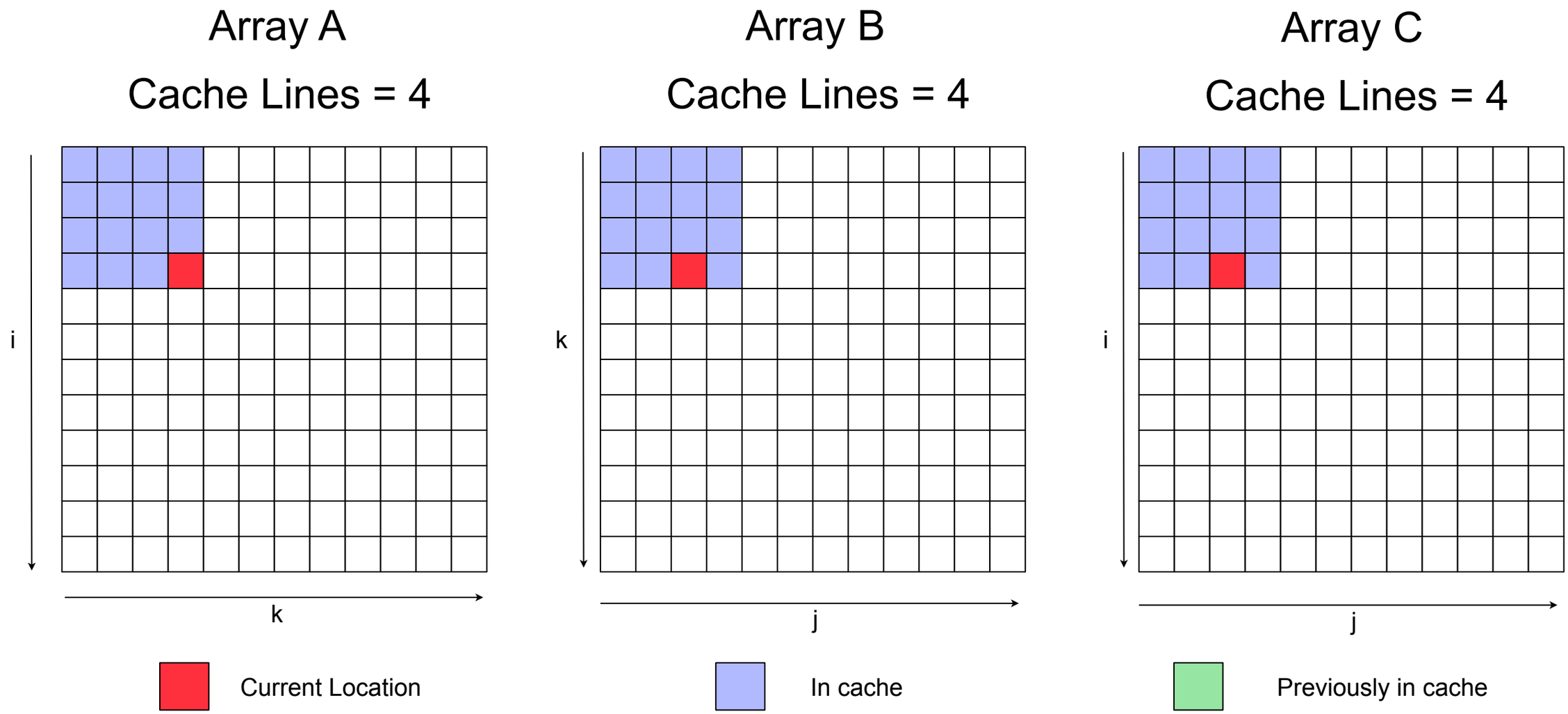
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



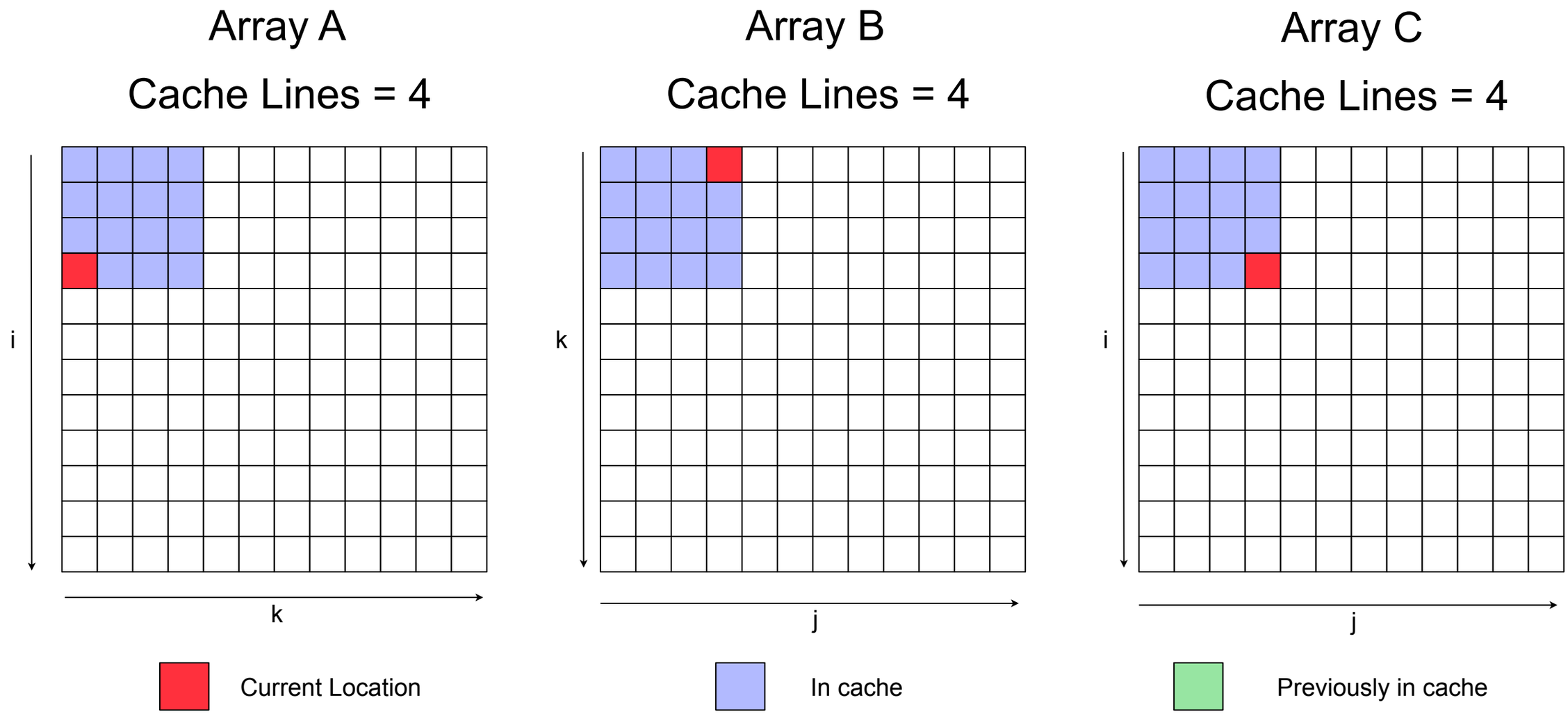
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



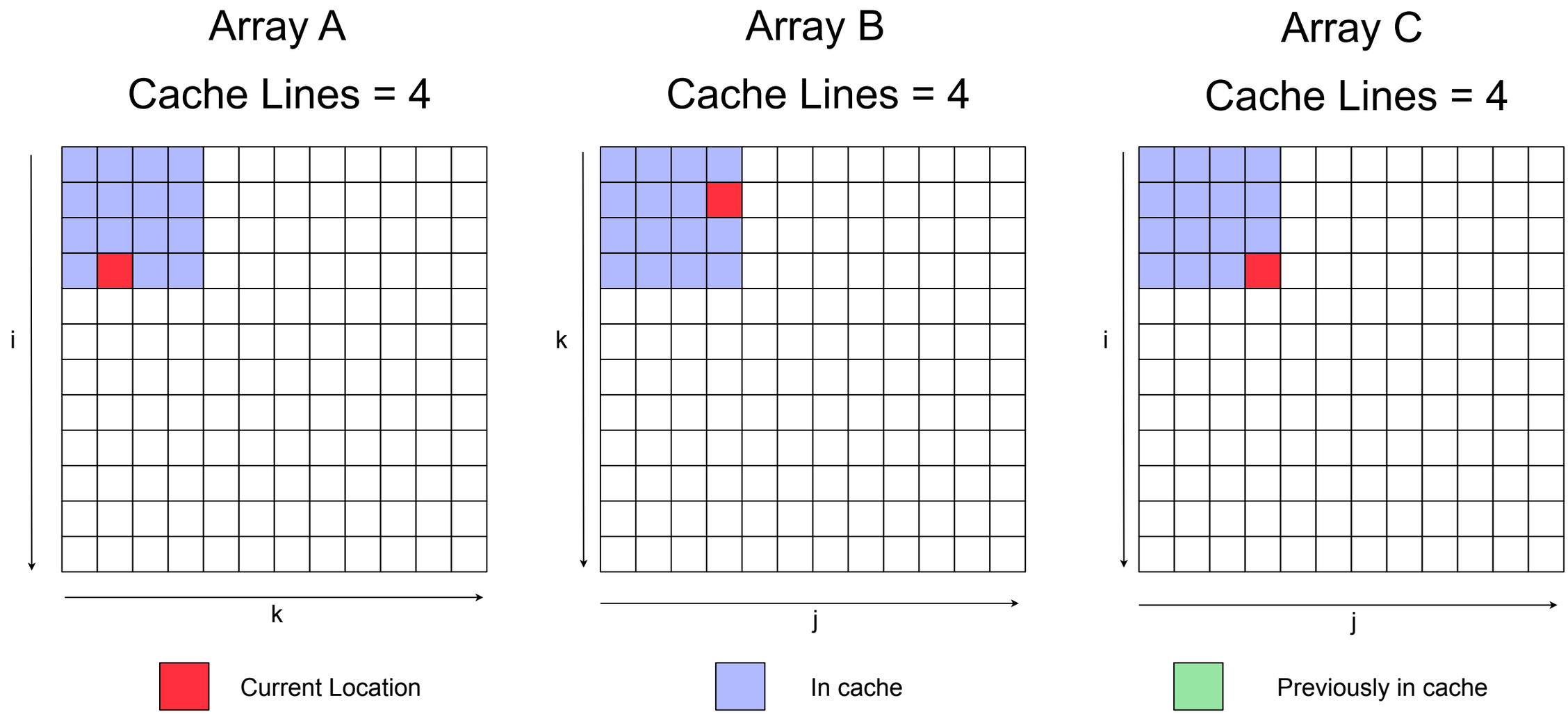
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



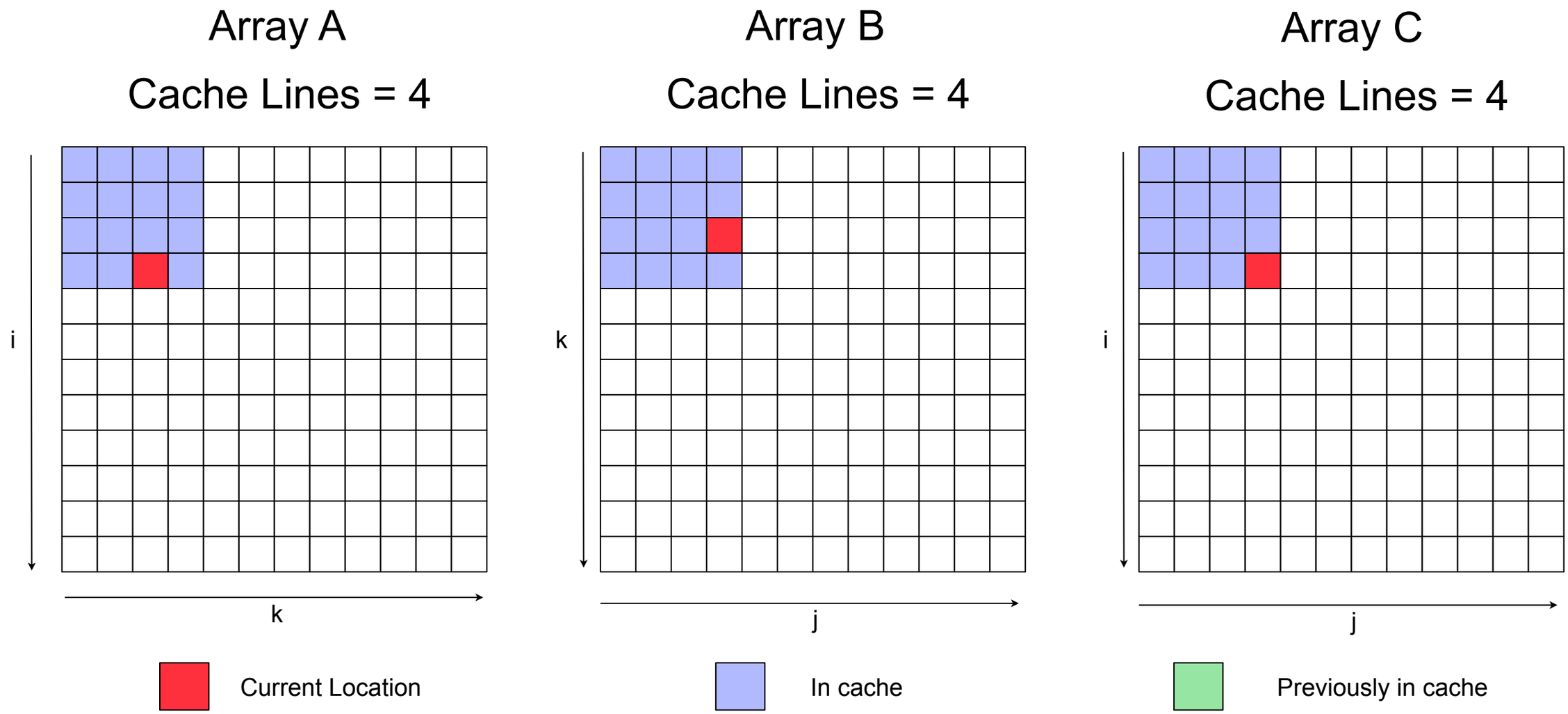
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



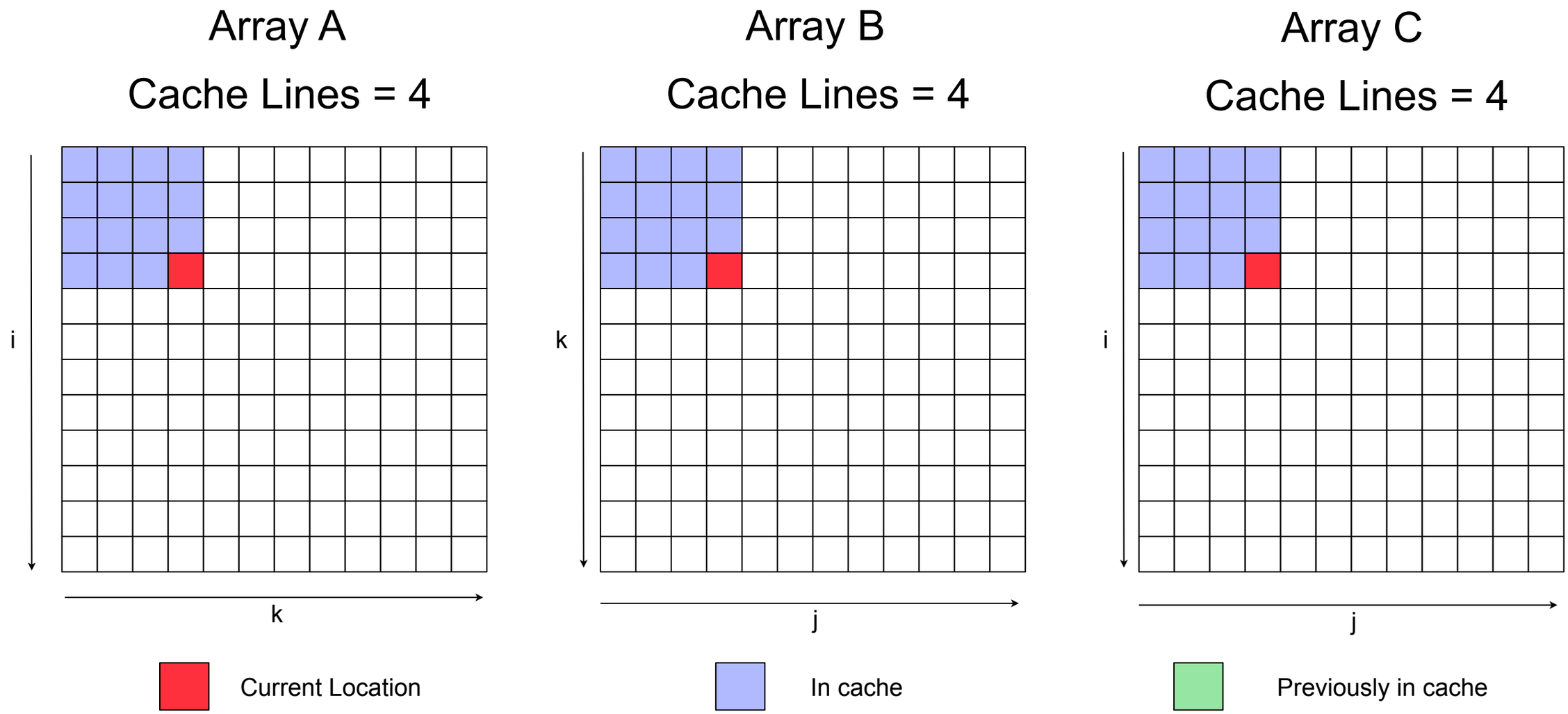
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



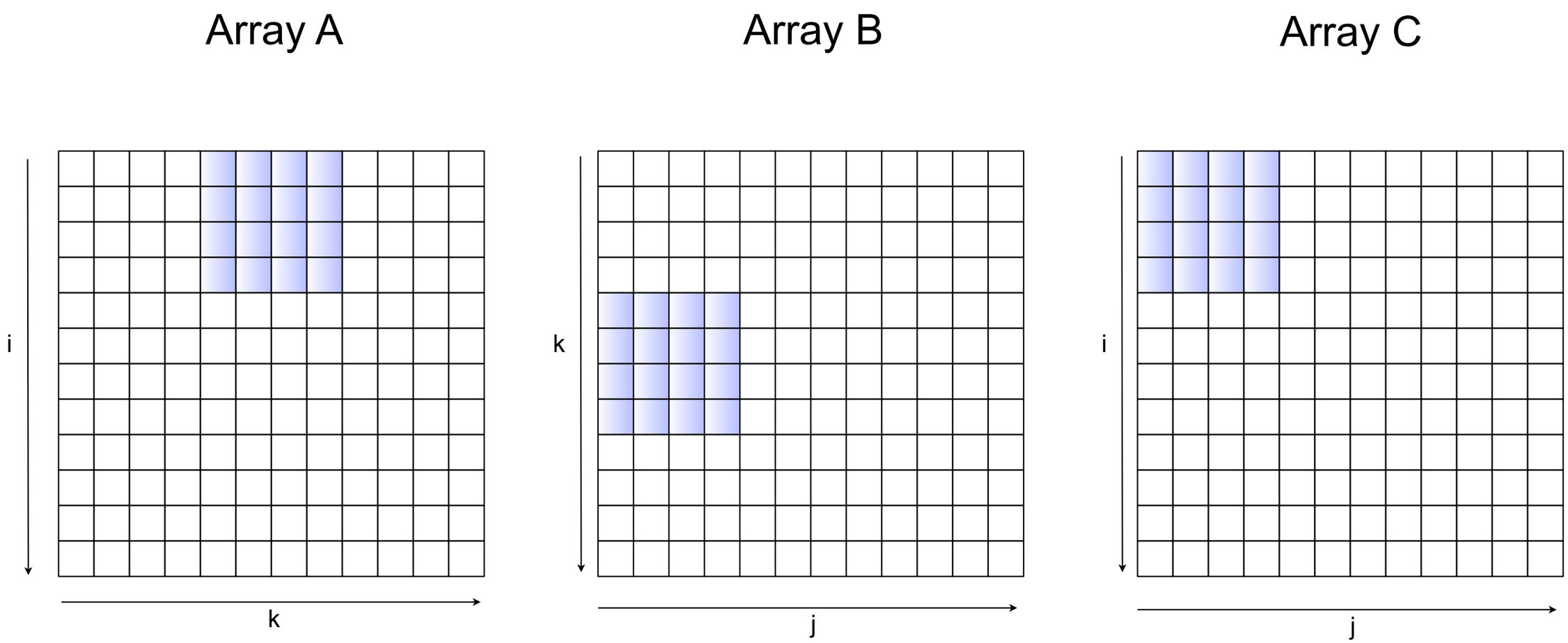
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4



Optimisation - Loop Blocking - Matrix Multiply

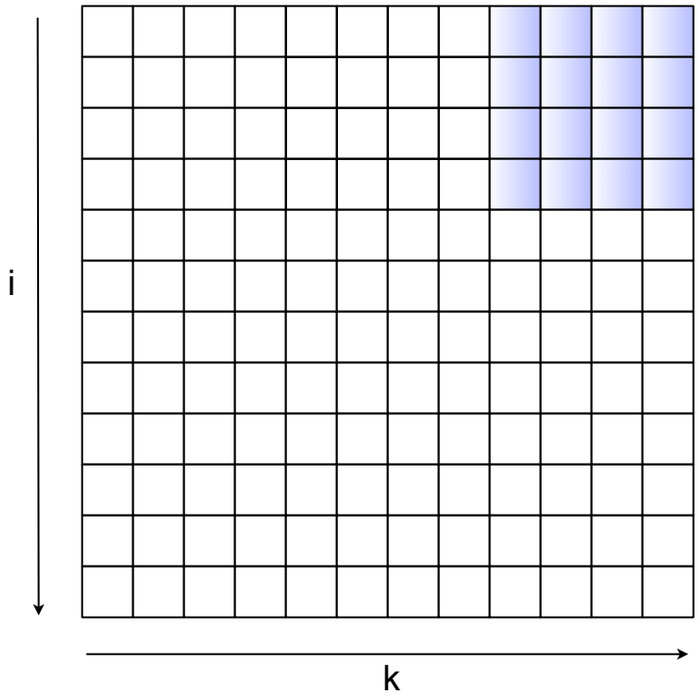
- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks



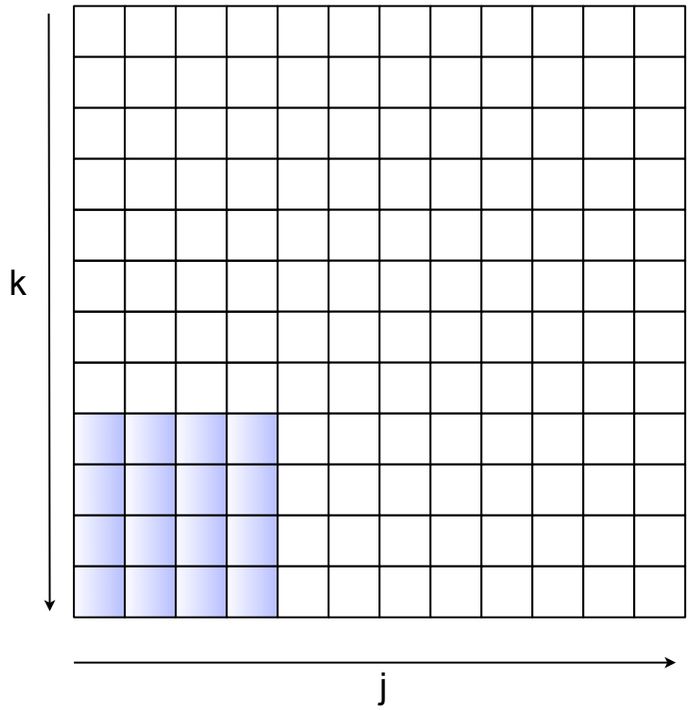
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

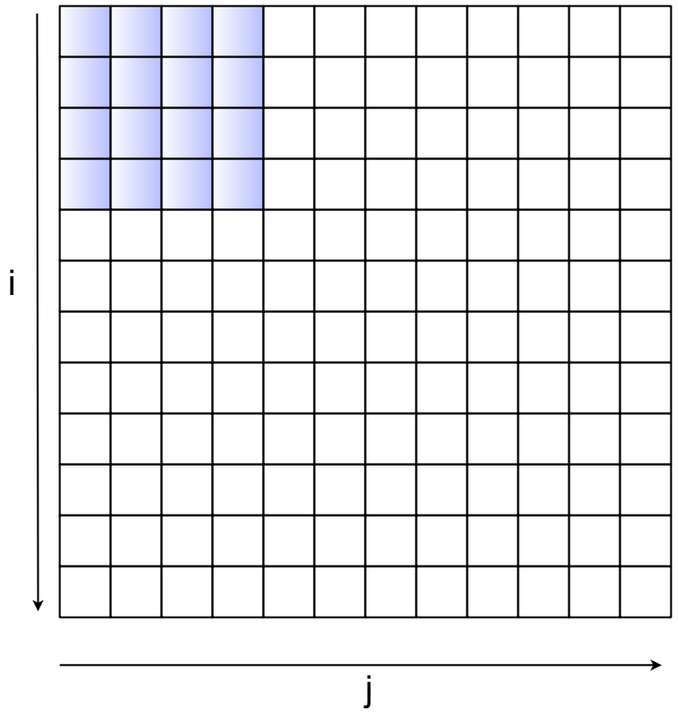
Array A



Array B



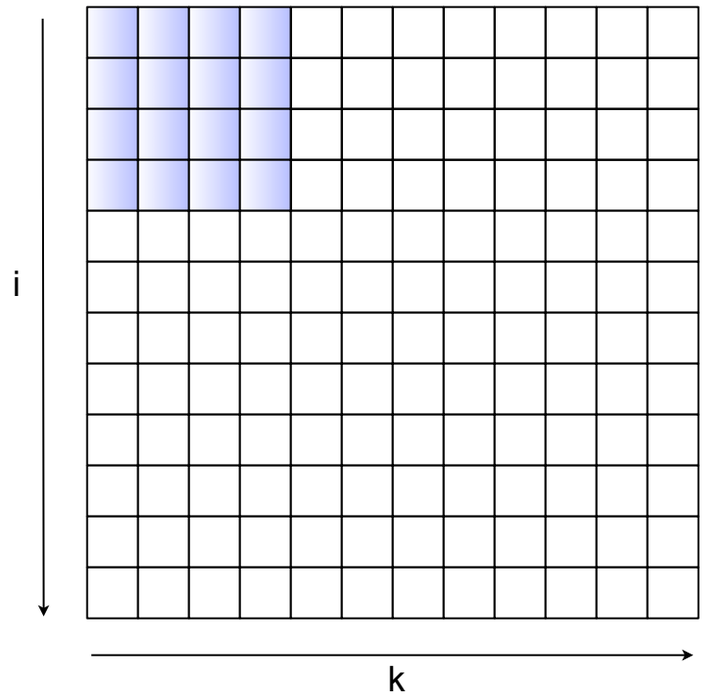
Array C



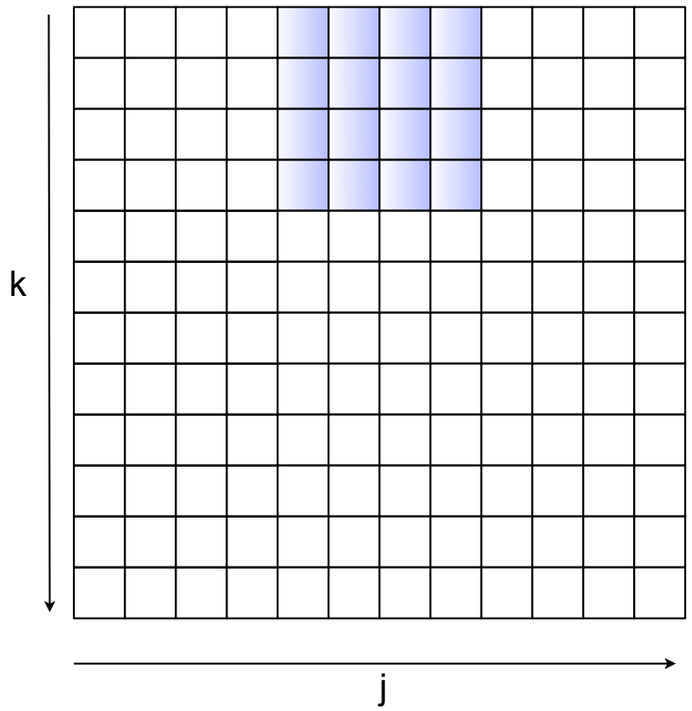
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

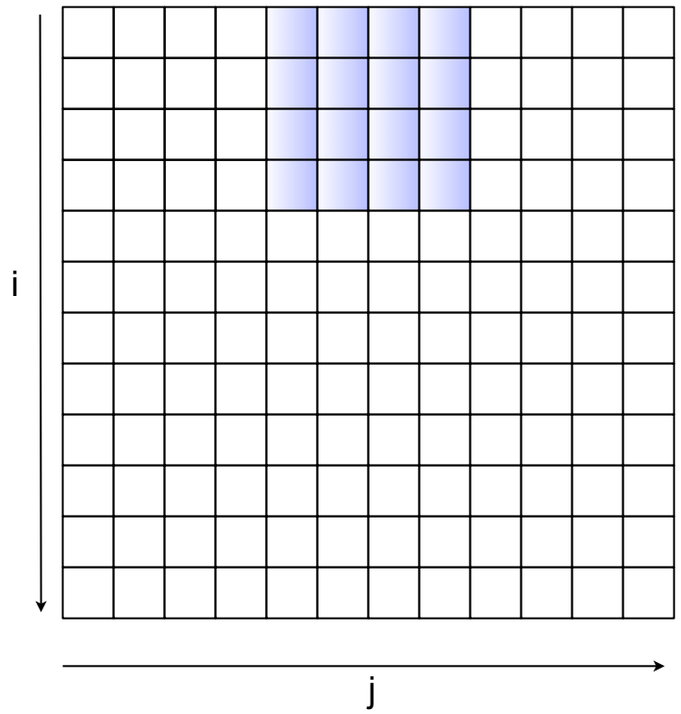
Array A



Array B



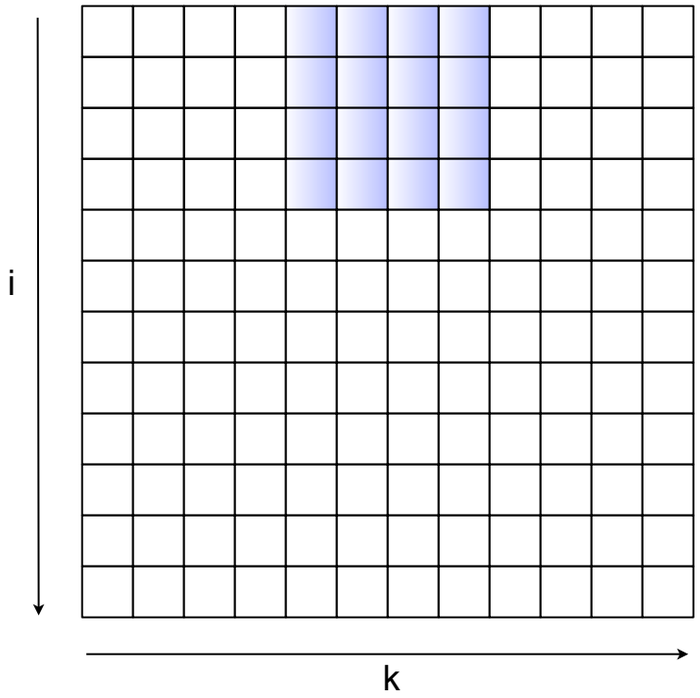
Array C



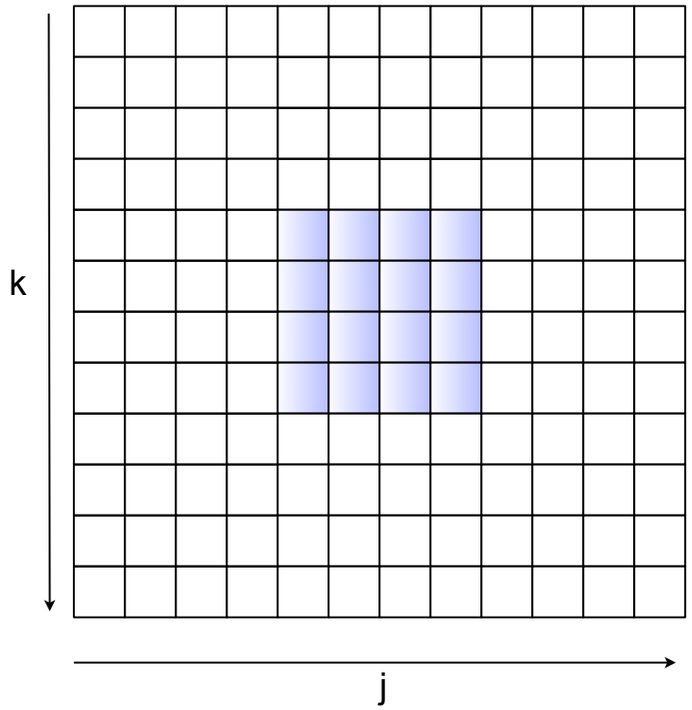
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

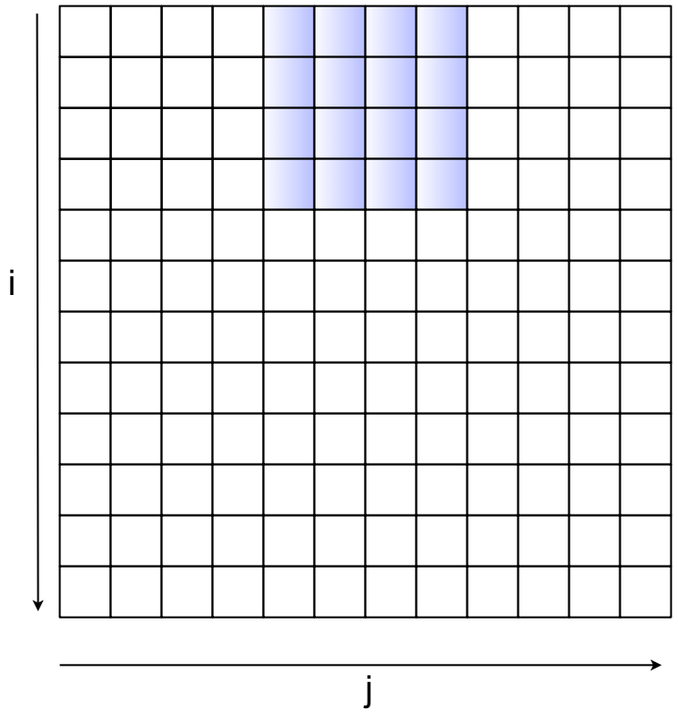
Array A



Array B



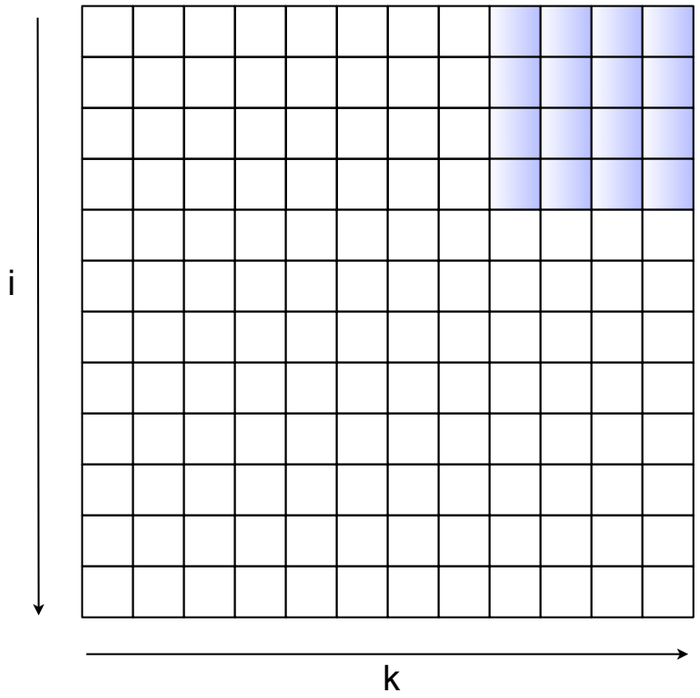
Array C



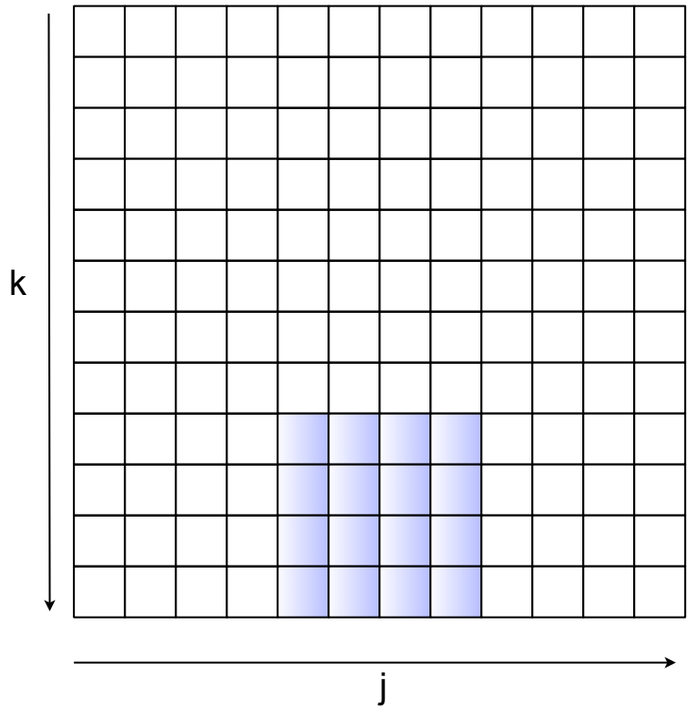
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

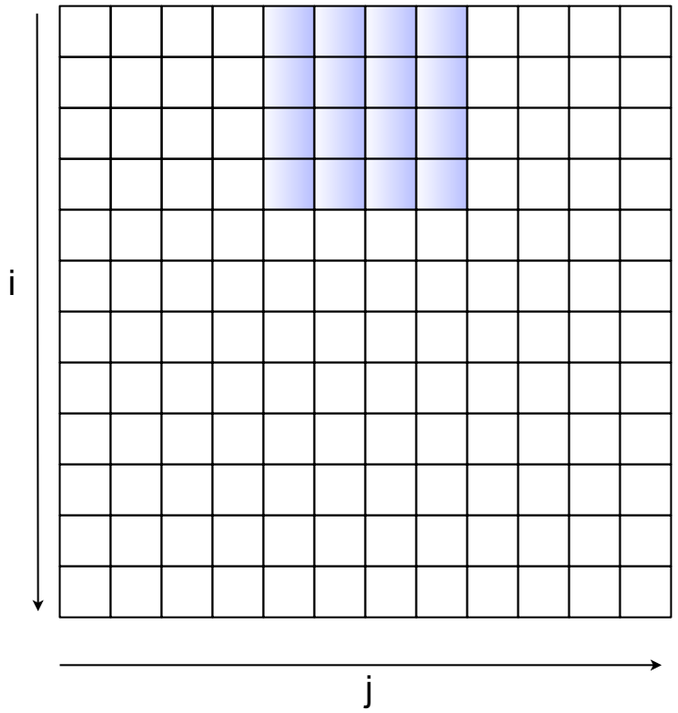
Array A



Array B



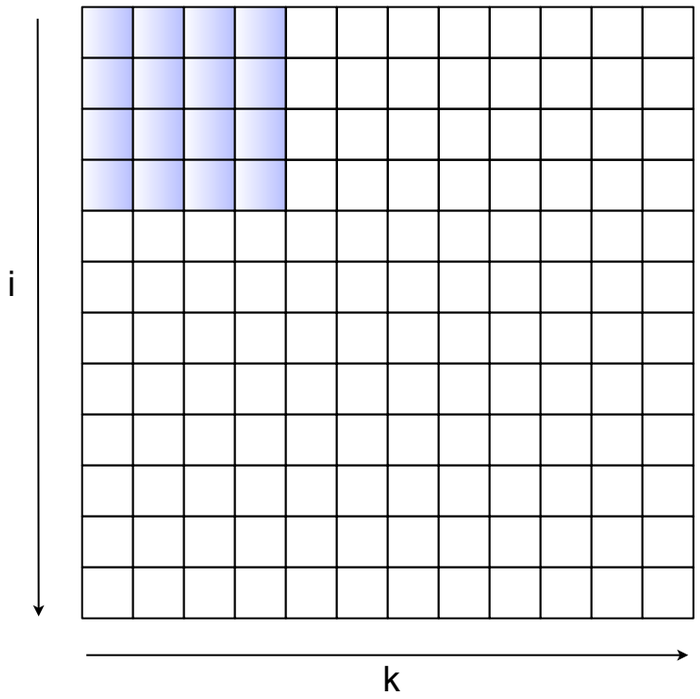
Array C



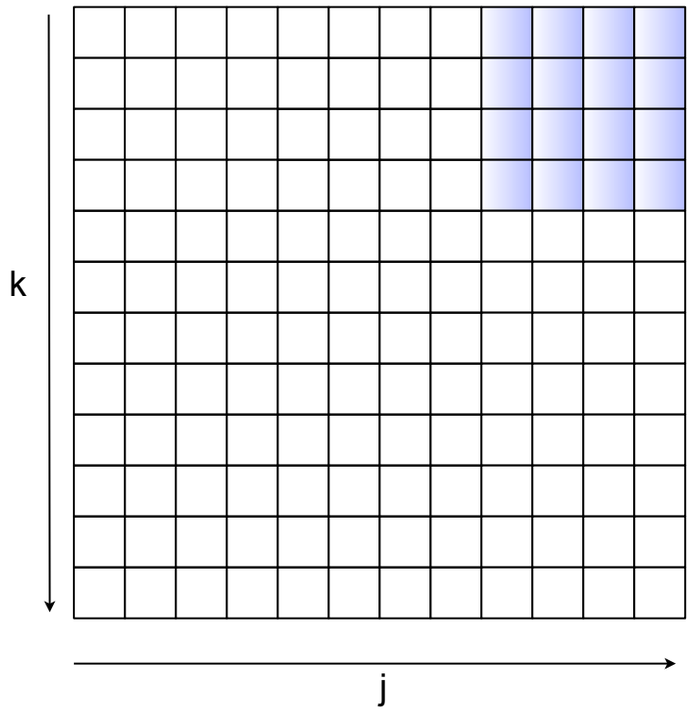
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

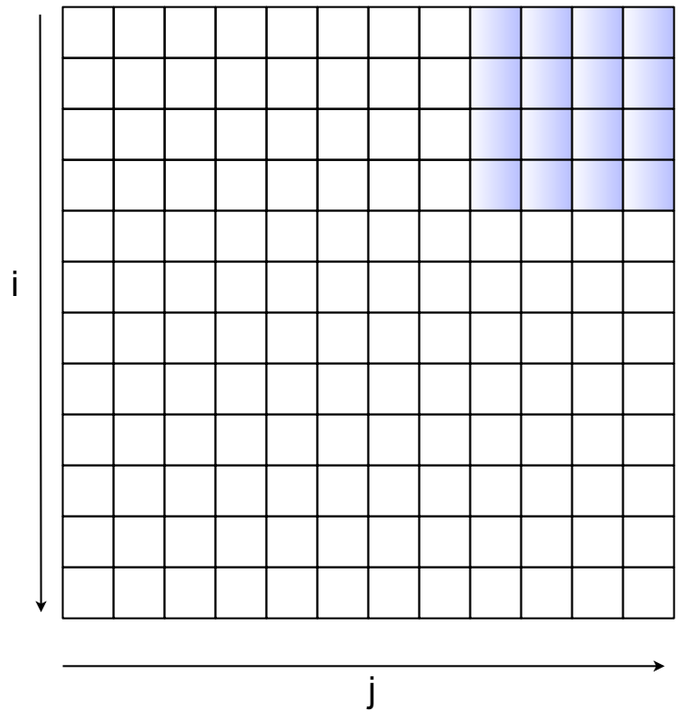
Array A



Array B



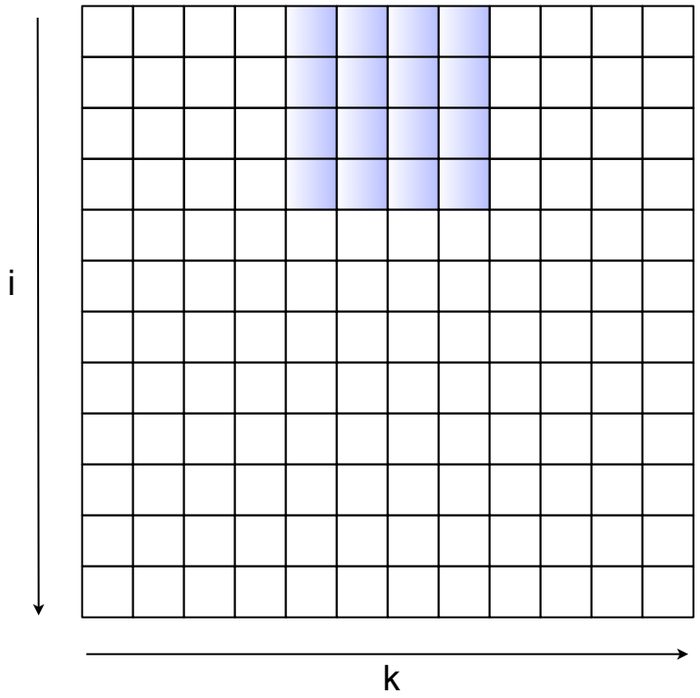
Array C



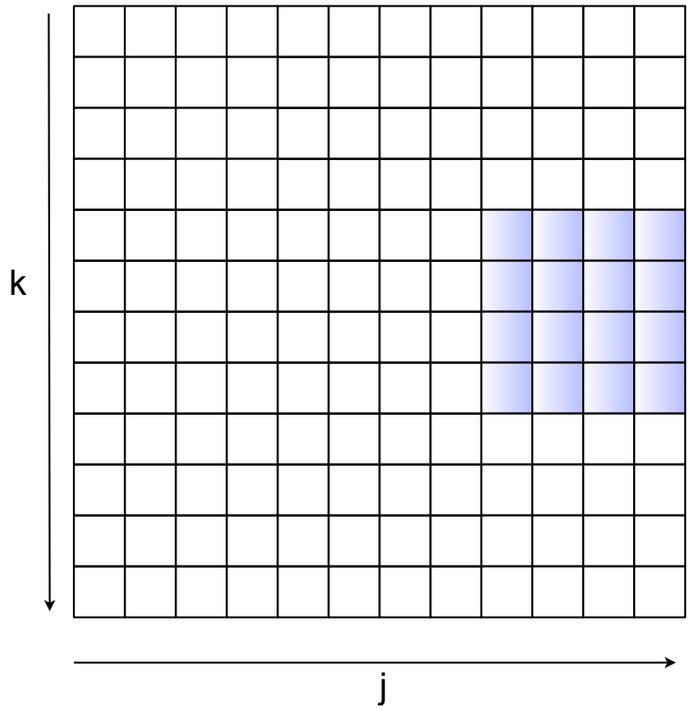
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

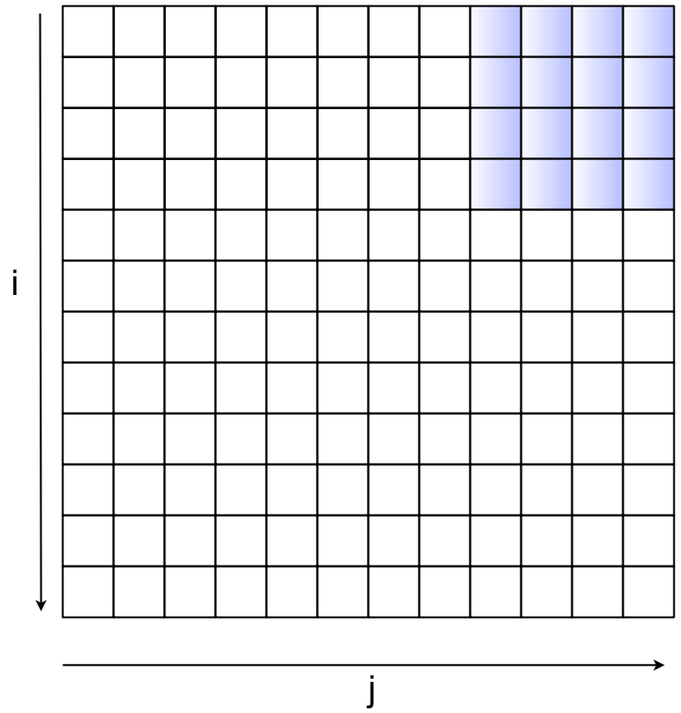
Array A



Array B



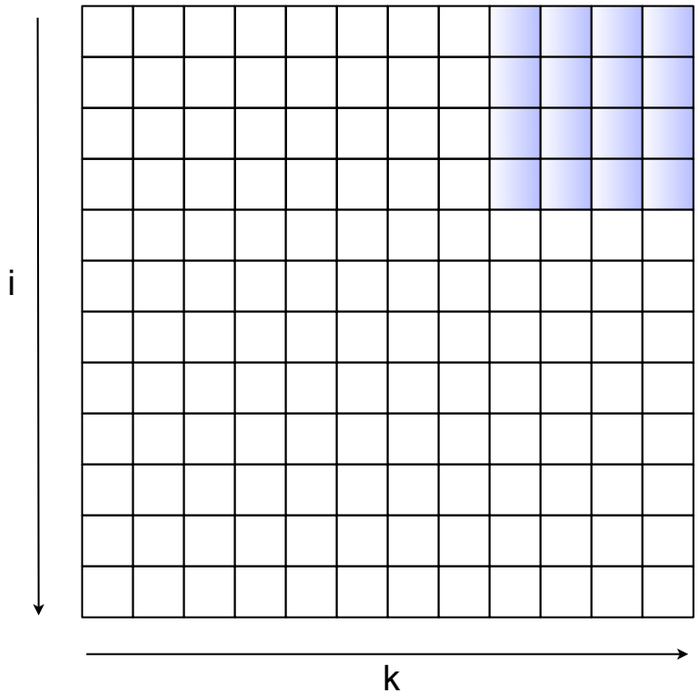
Array C



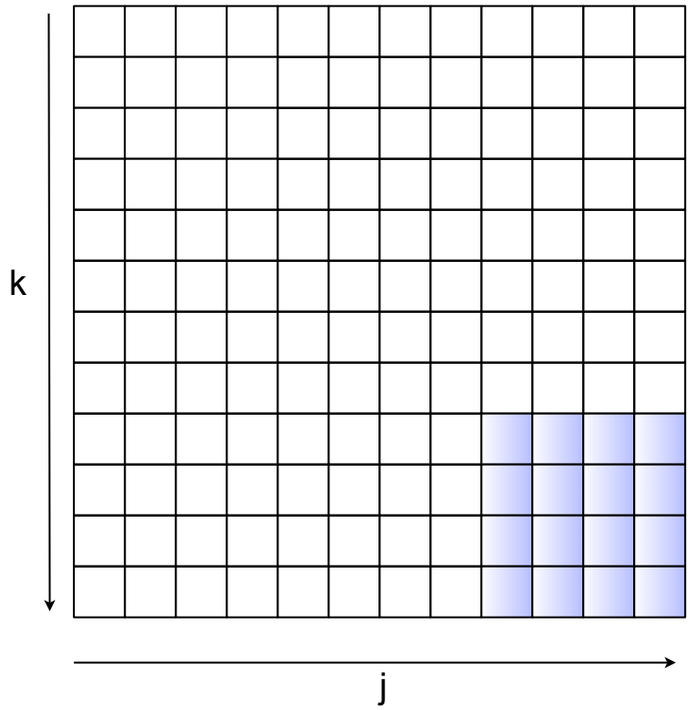
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

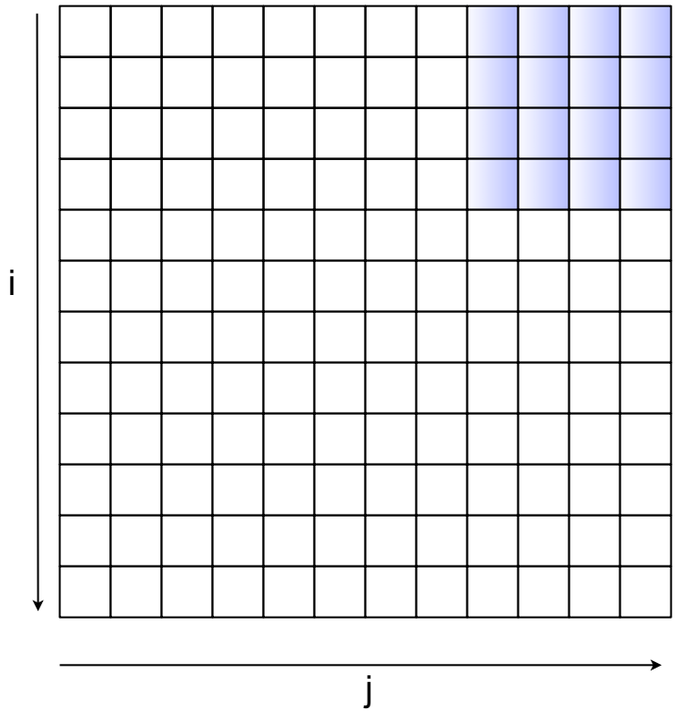
Array A



Array B



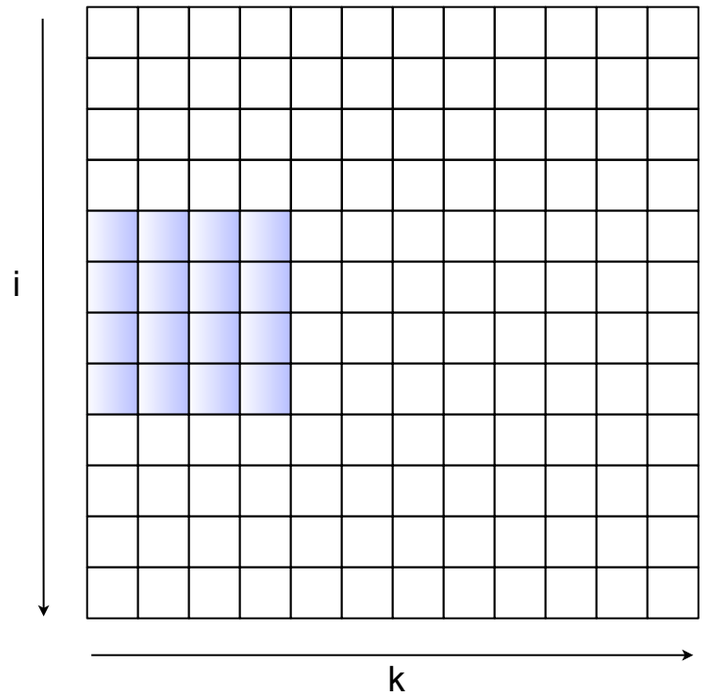
Array C



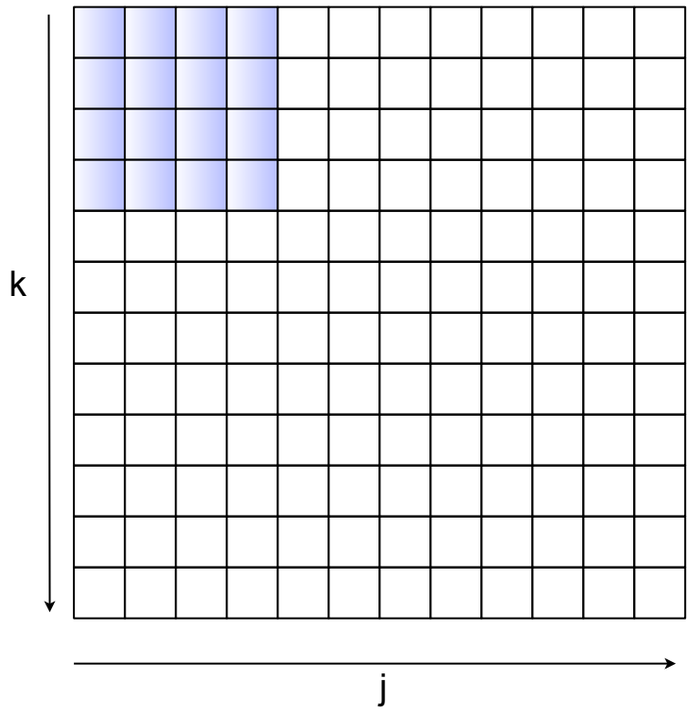
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

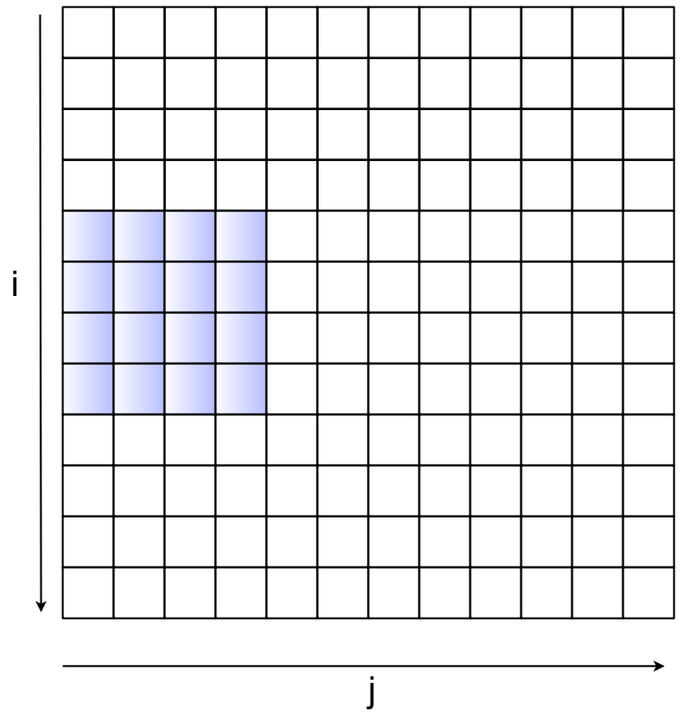
Array A



Array B



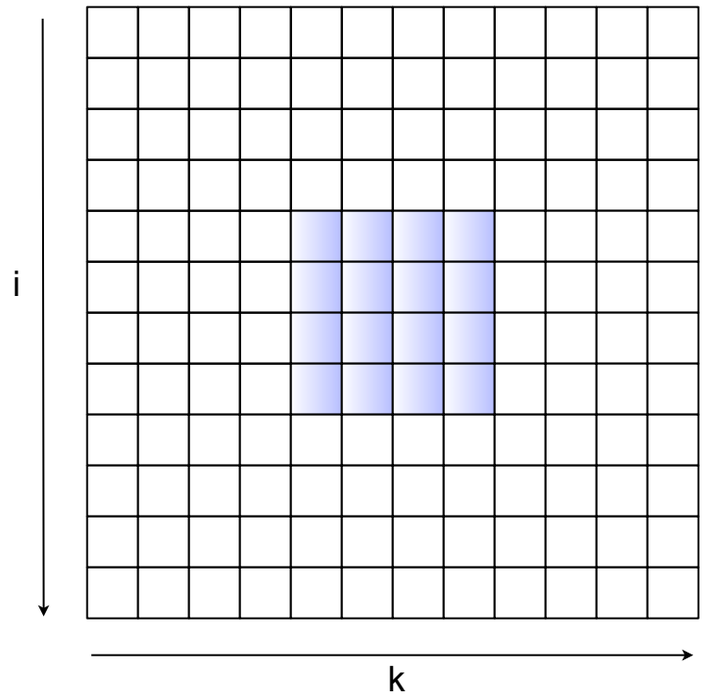
Array C



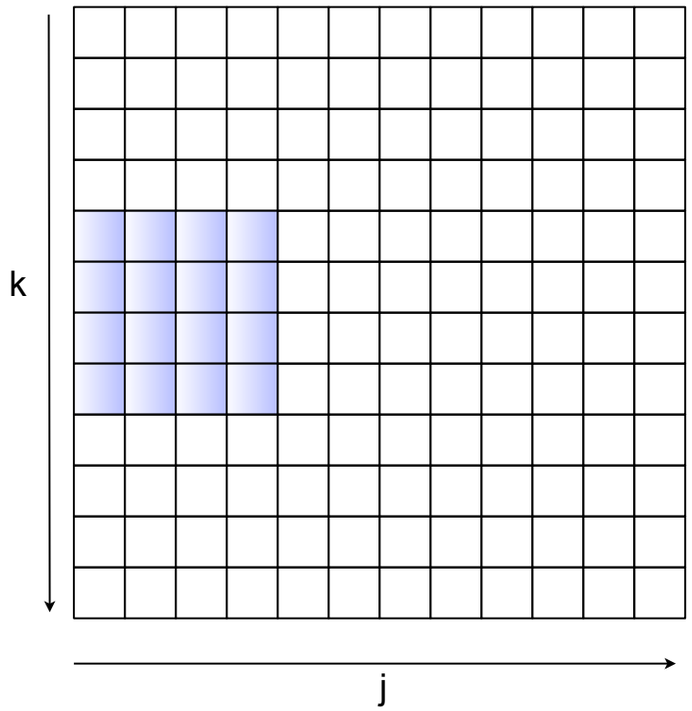
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

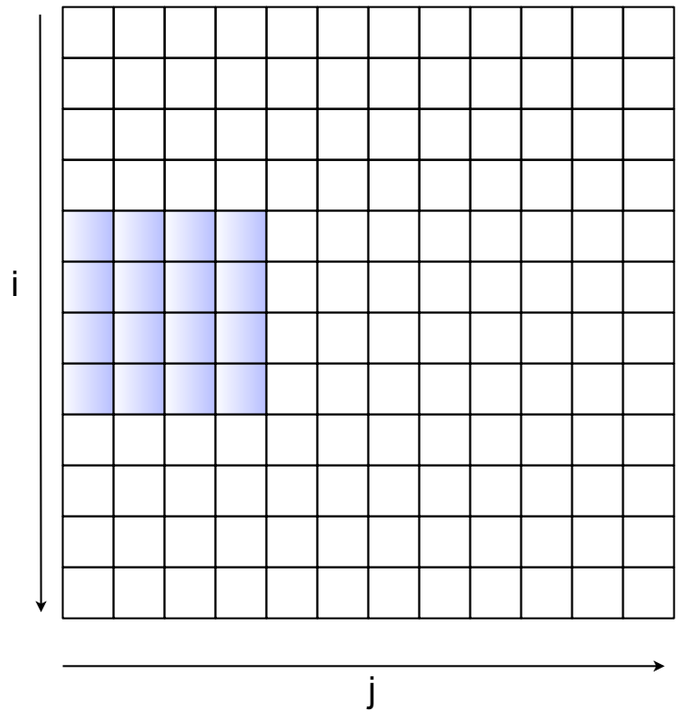
Array A



Array B



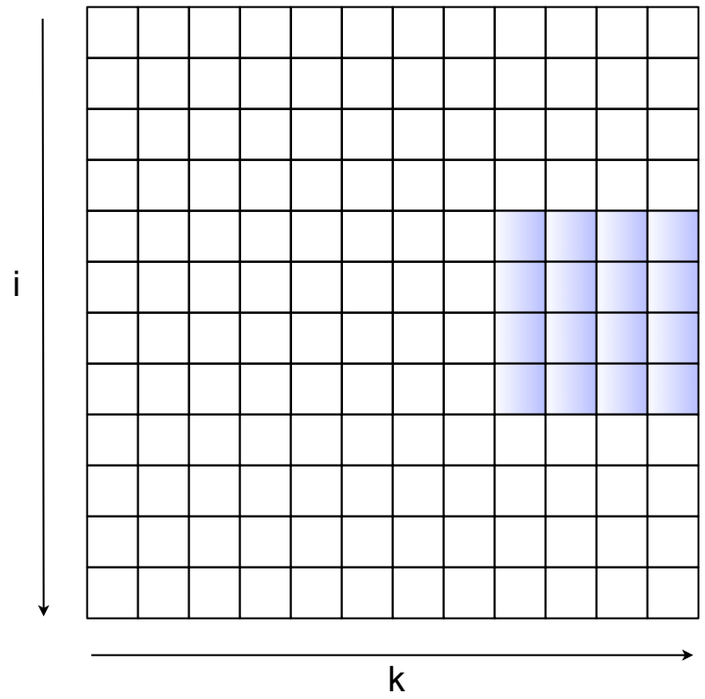
Array C



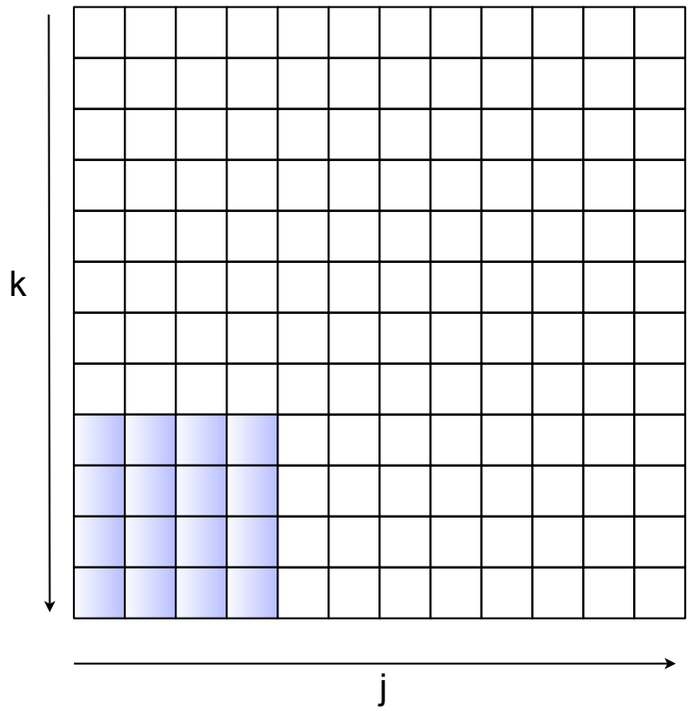
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

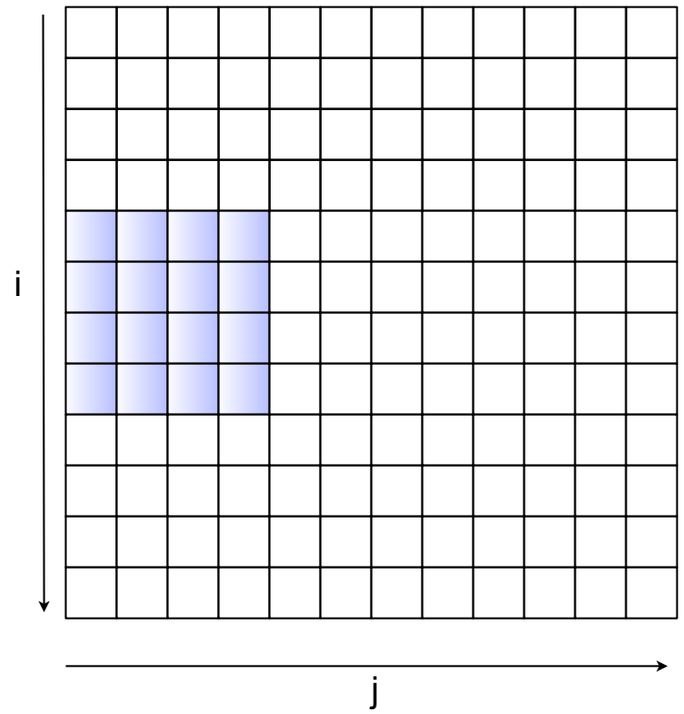
Array A



Array B

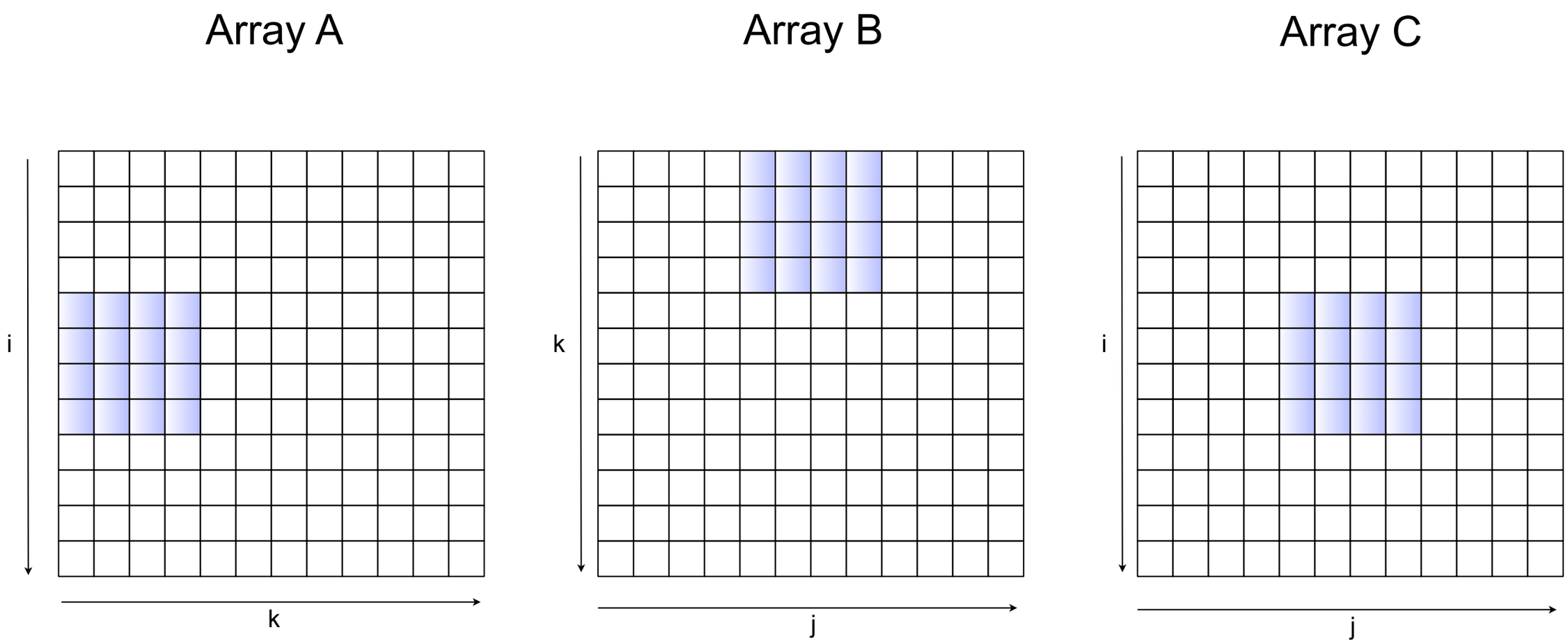


Array C



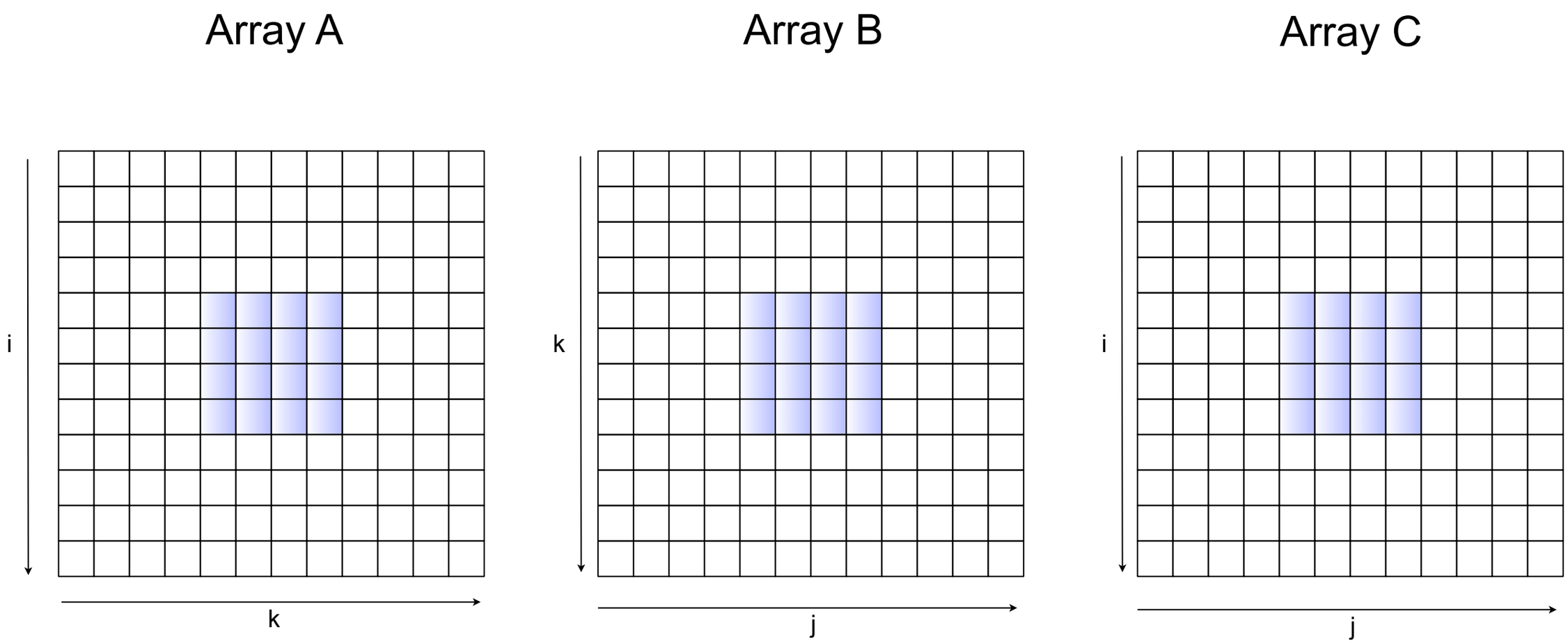
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks



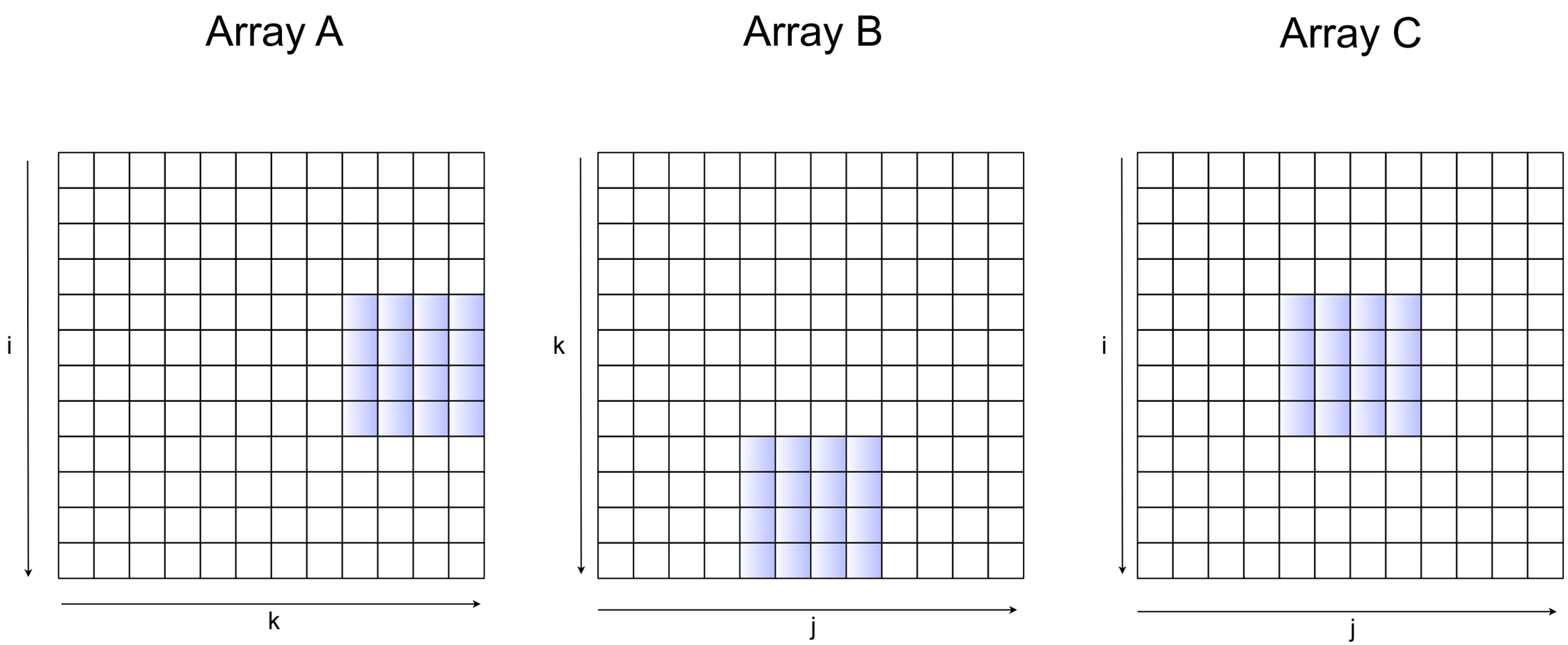
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks



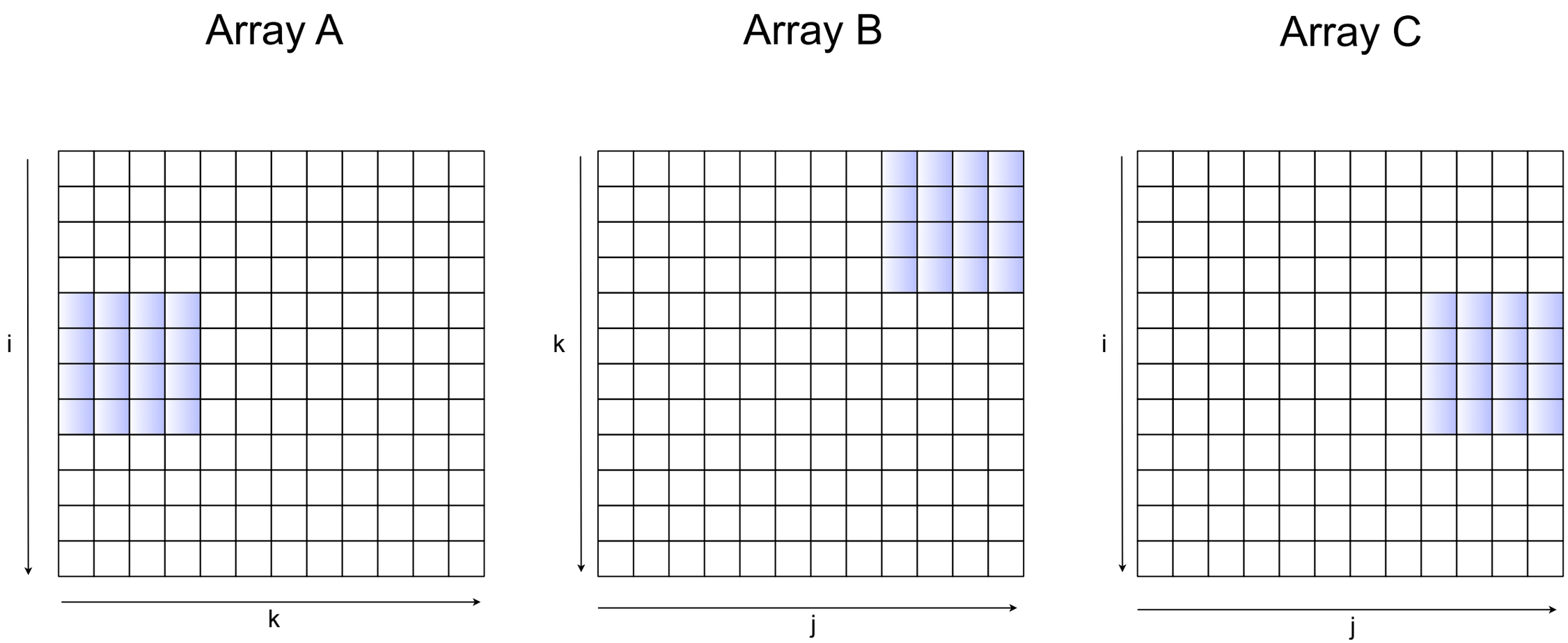
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks



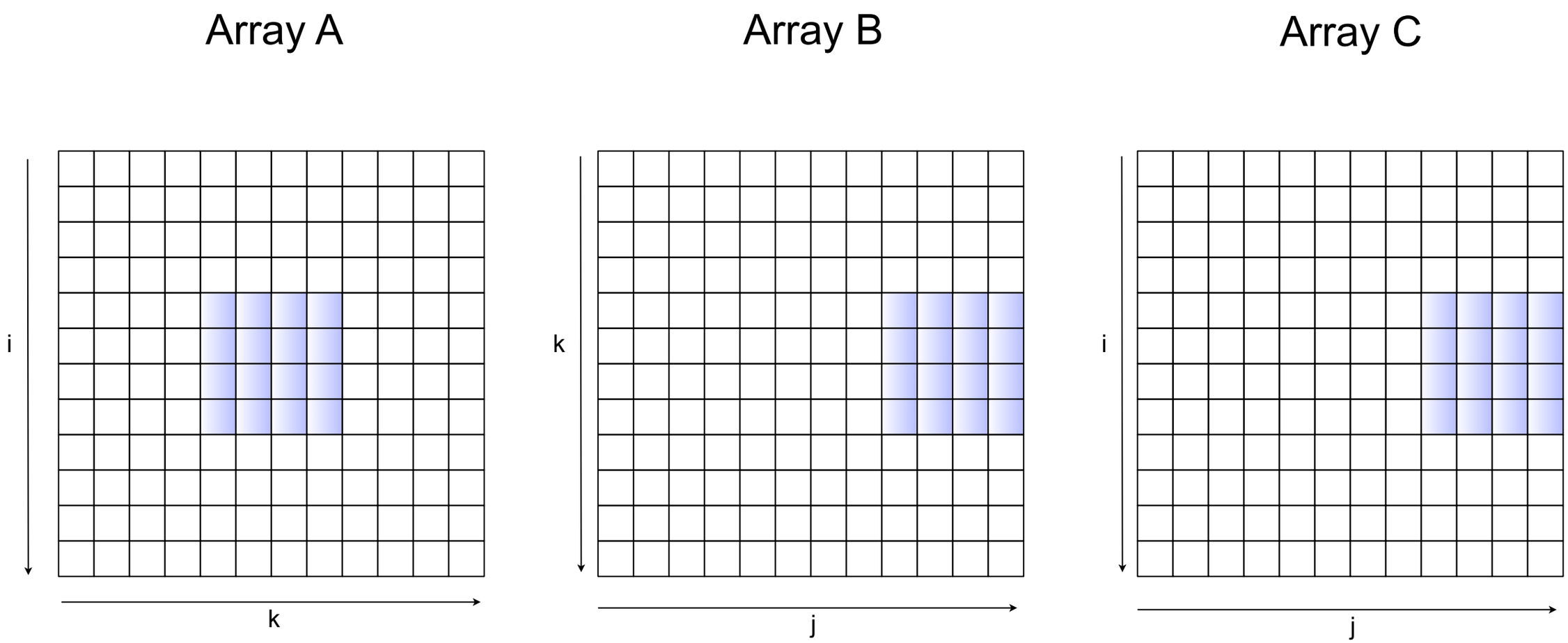
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks



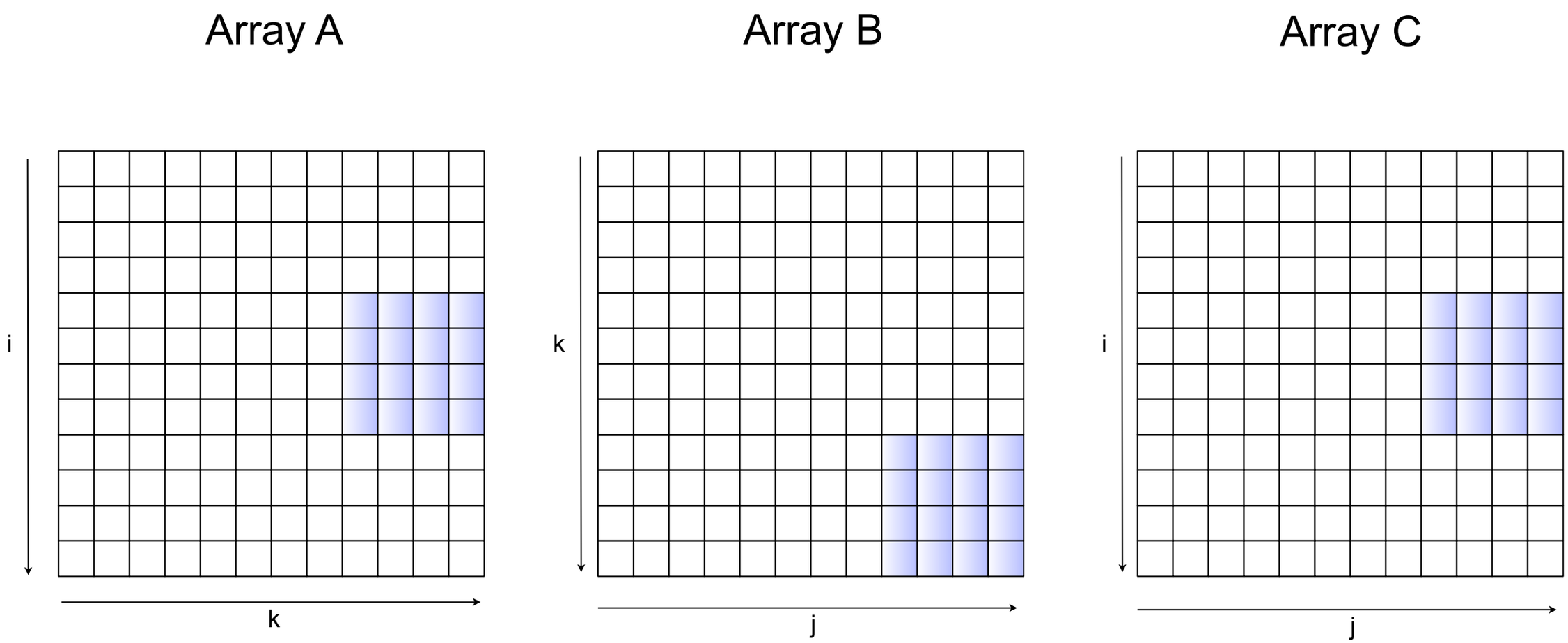
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks



Optimisation - Loop Blocking - Matrix Multiply

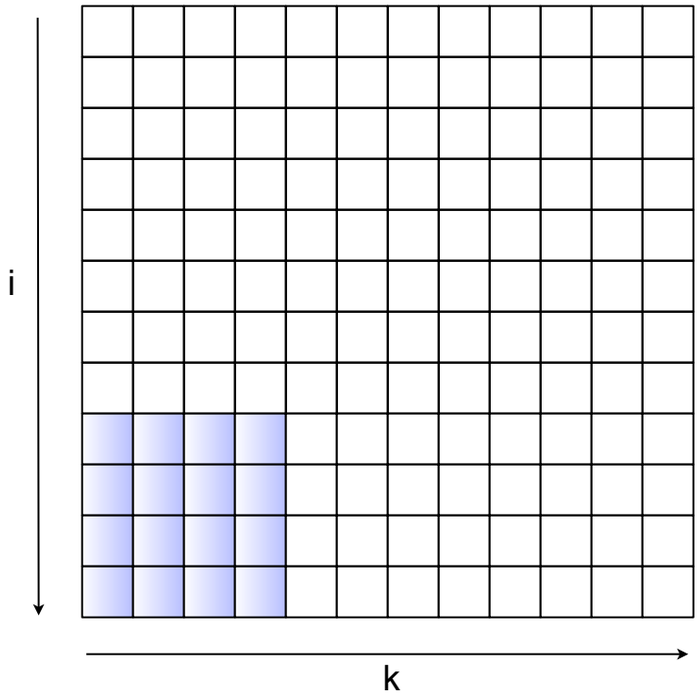
- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks



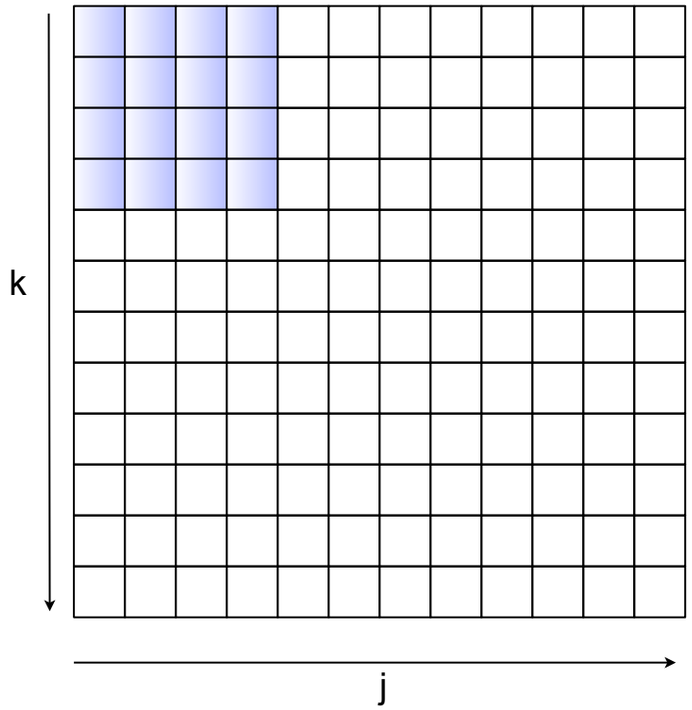
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

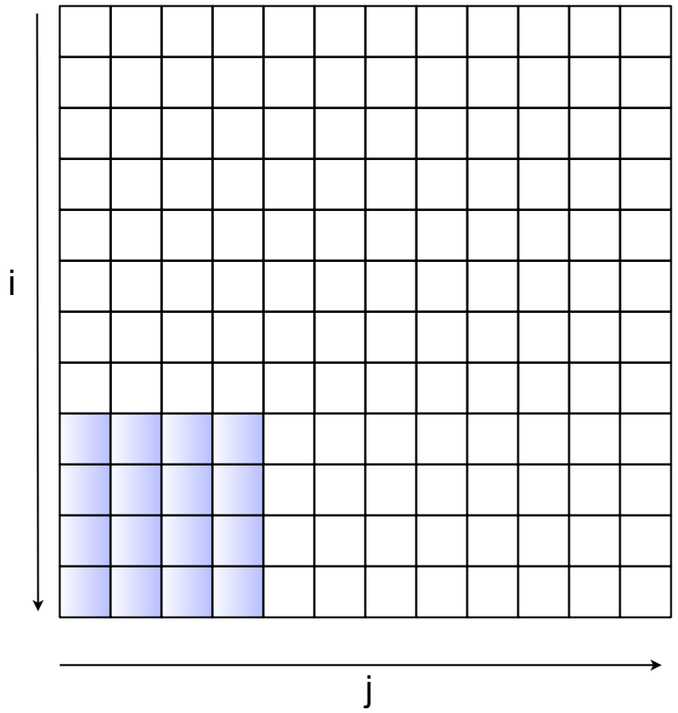
Array A



Array B

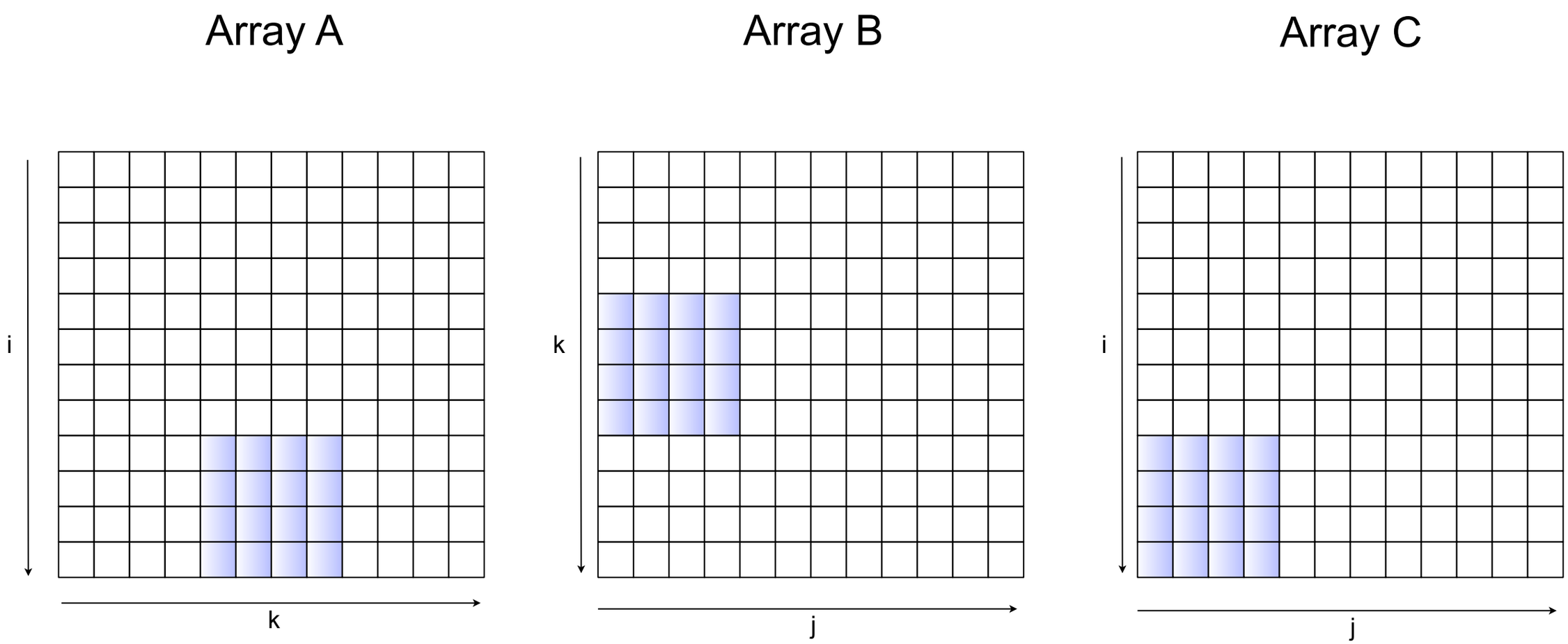


Array C



Optimisation - Loop Blocking - Matrix Multiply

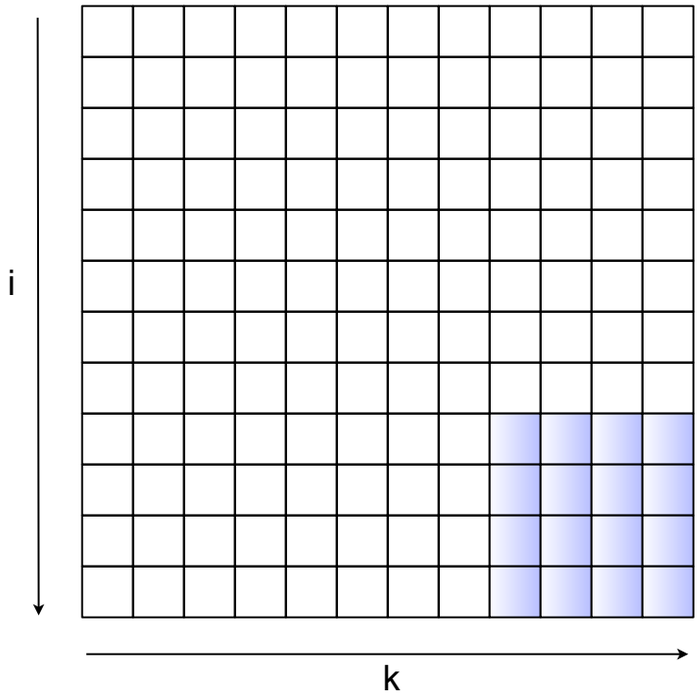
- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks



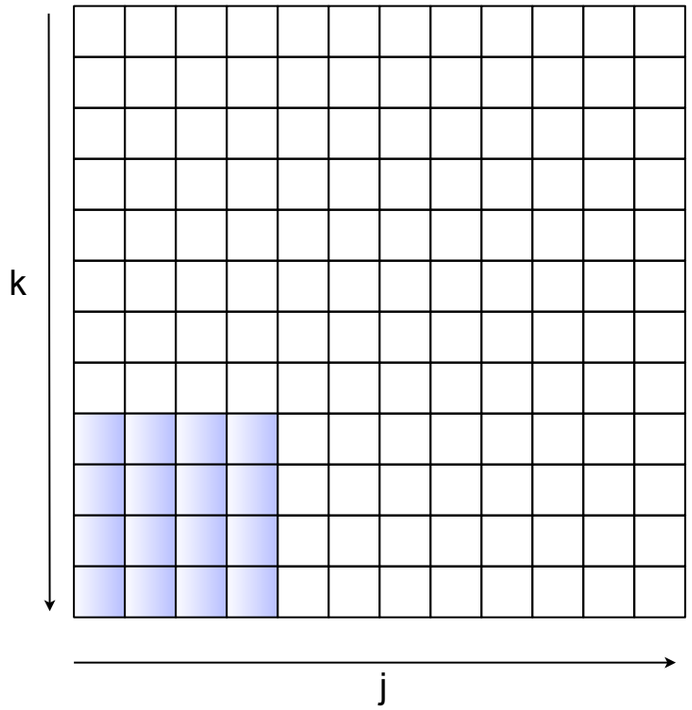
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

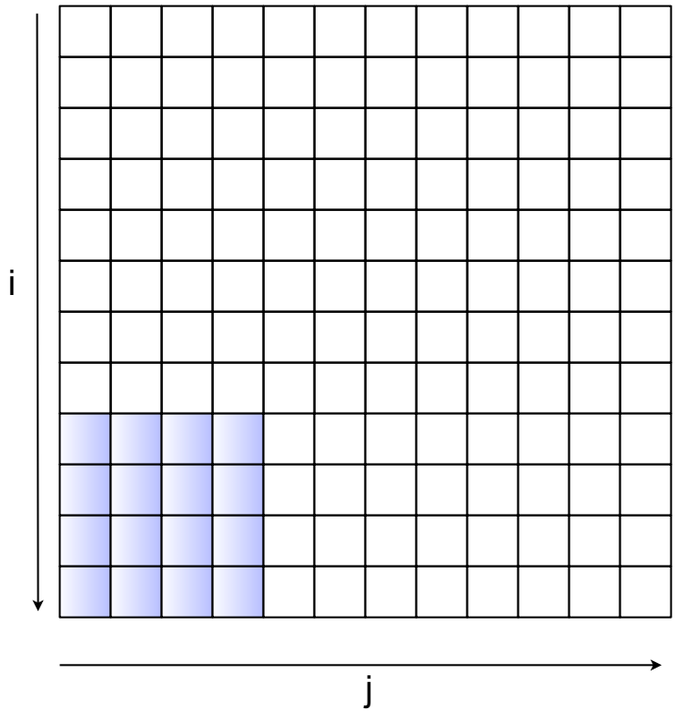
Array A



Array B



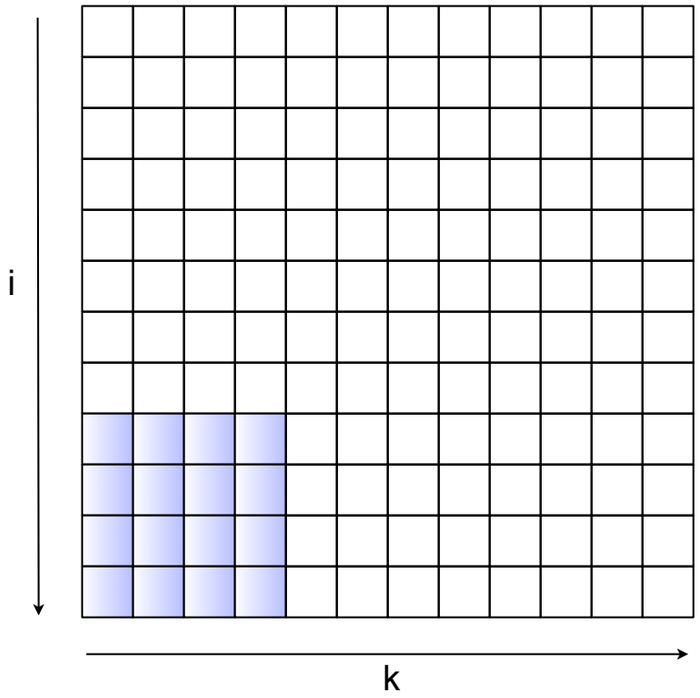
Array C



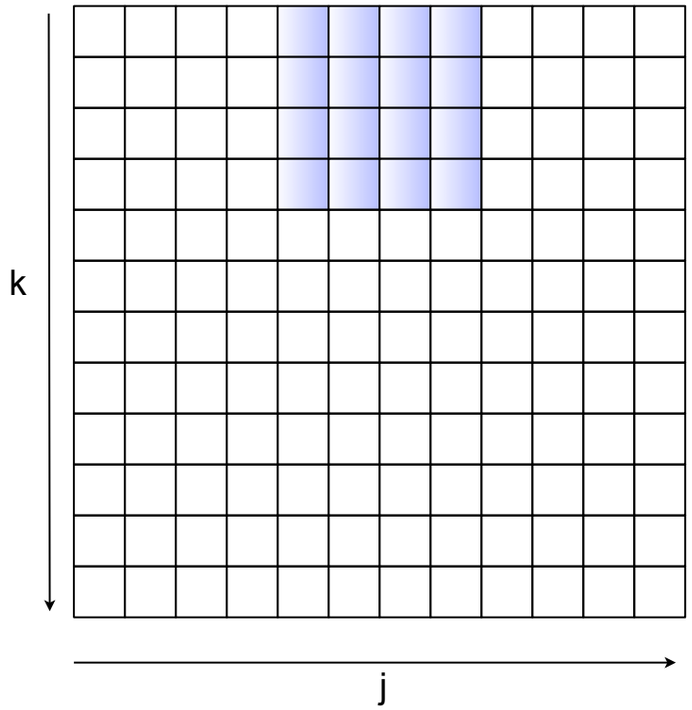
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

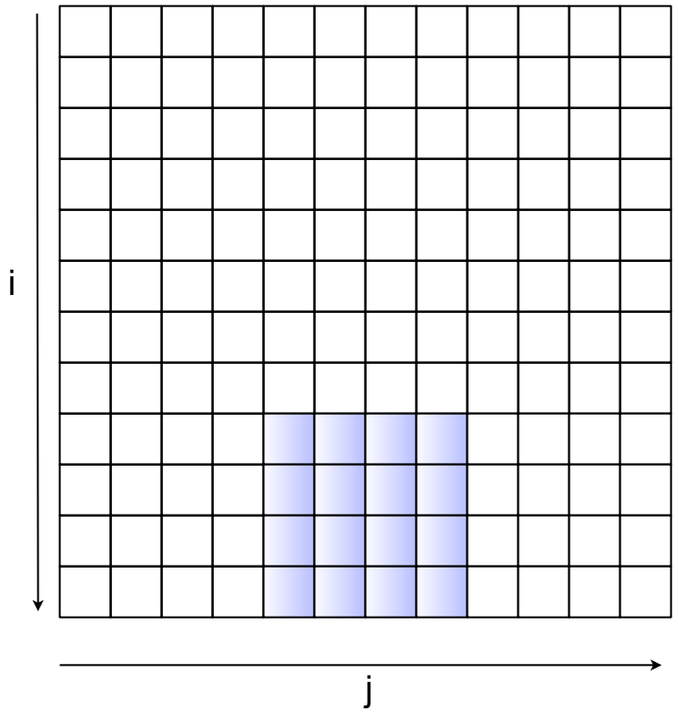
Array A



Array B

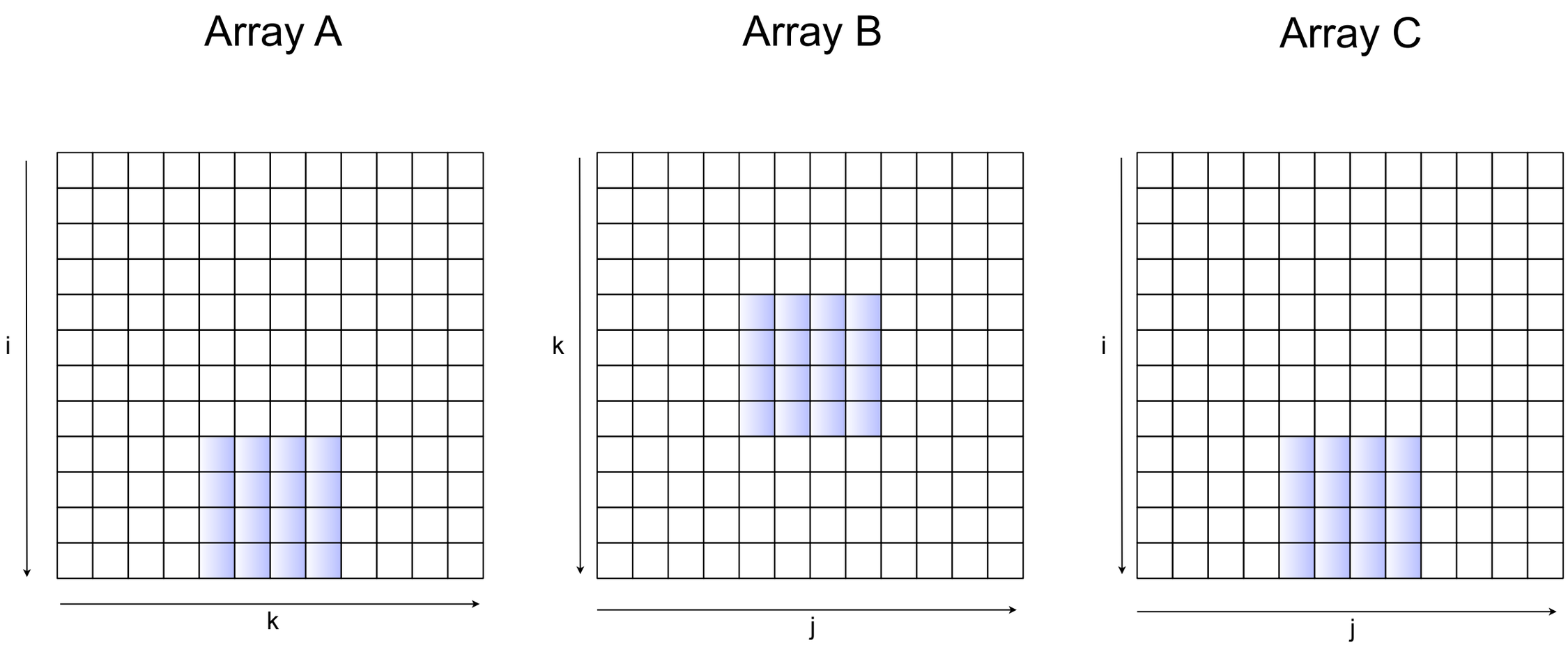


Array C



Optimisation - Loop Blocking - Matrix Multiply

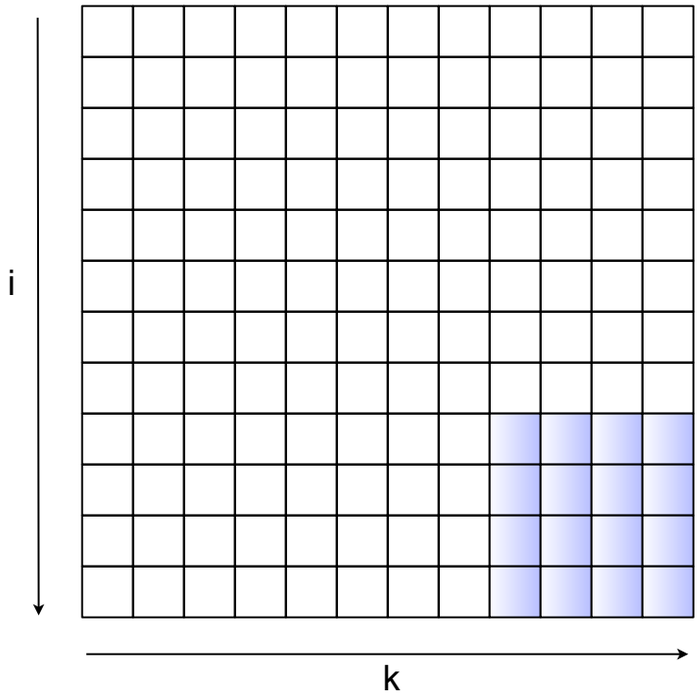
- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks



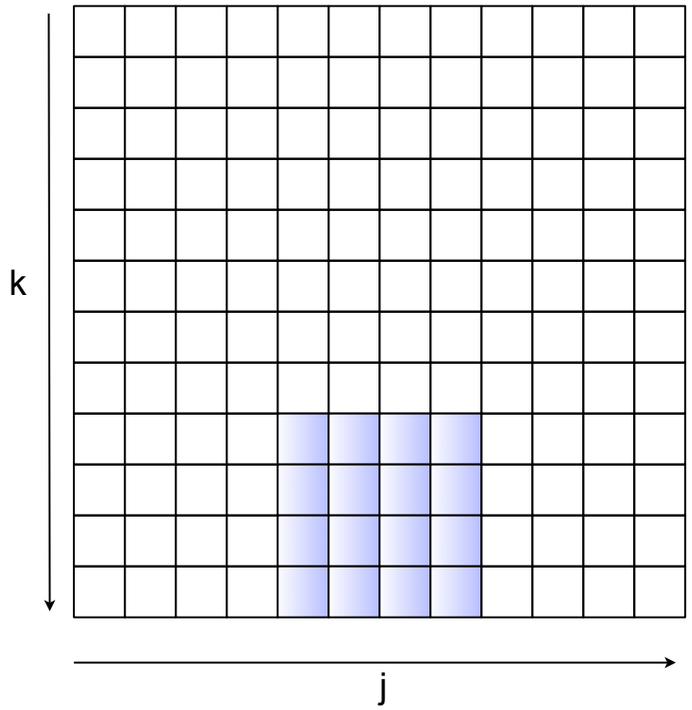
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

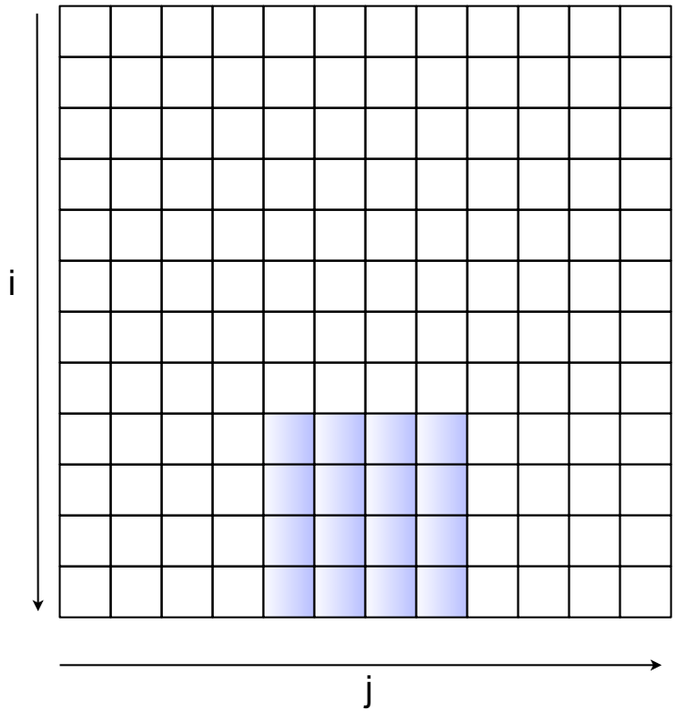
Array A



Array B

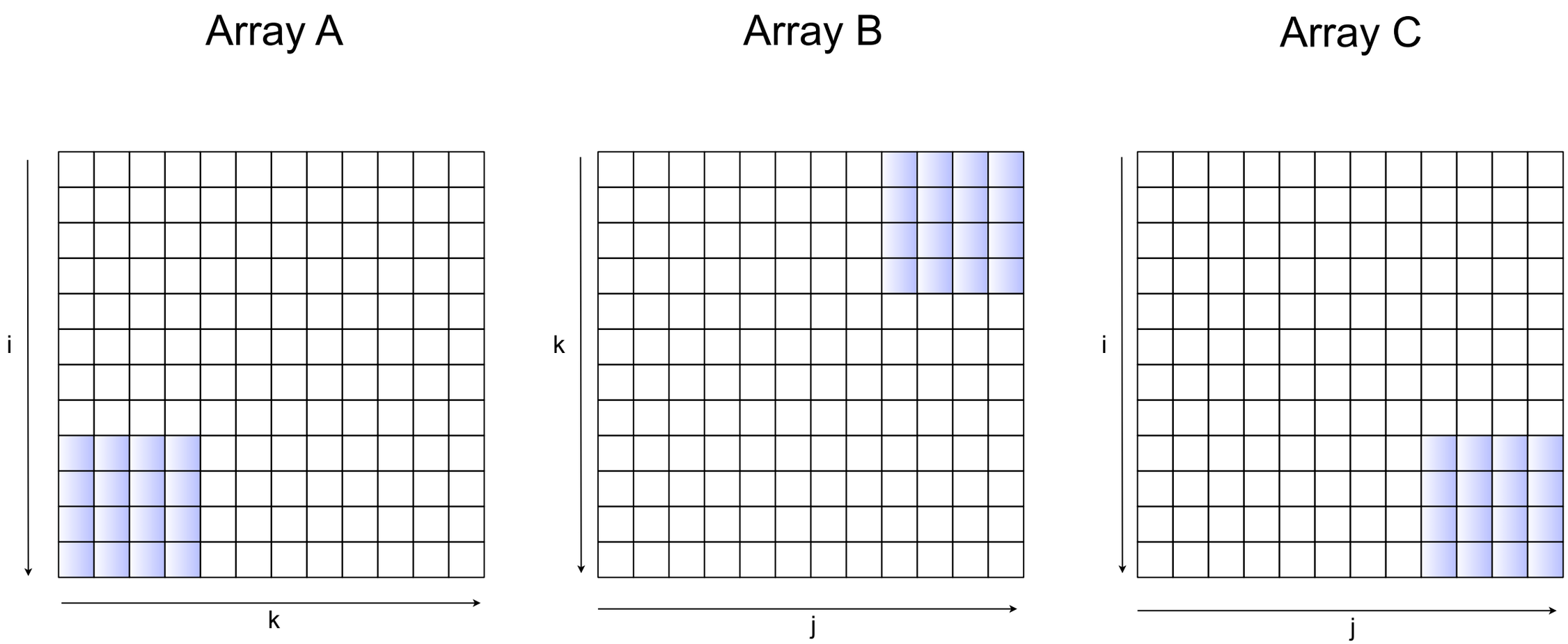


Array C



Optimisation - Loop Blocking - Matrix Multiply

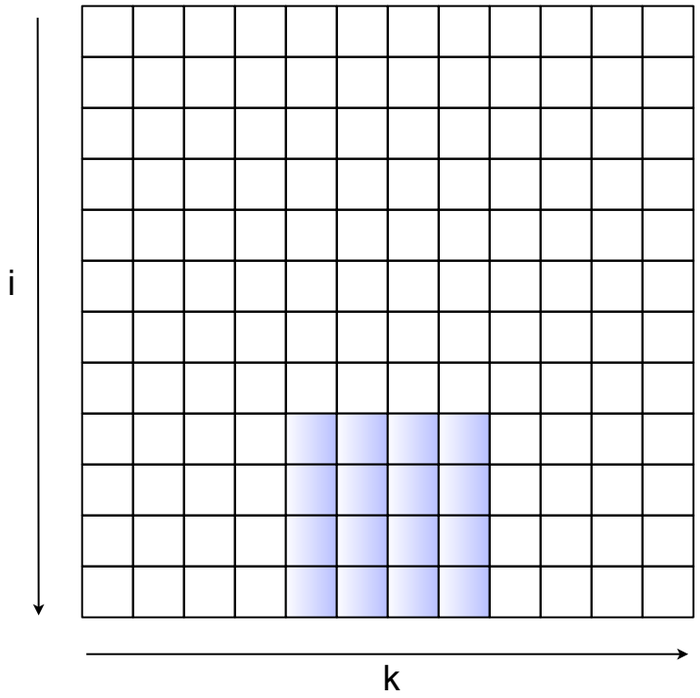
- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks



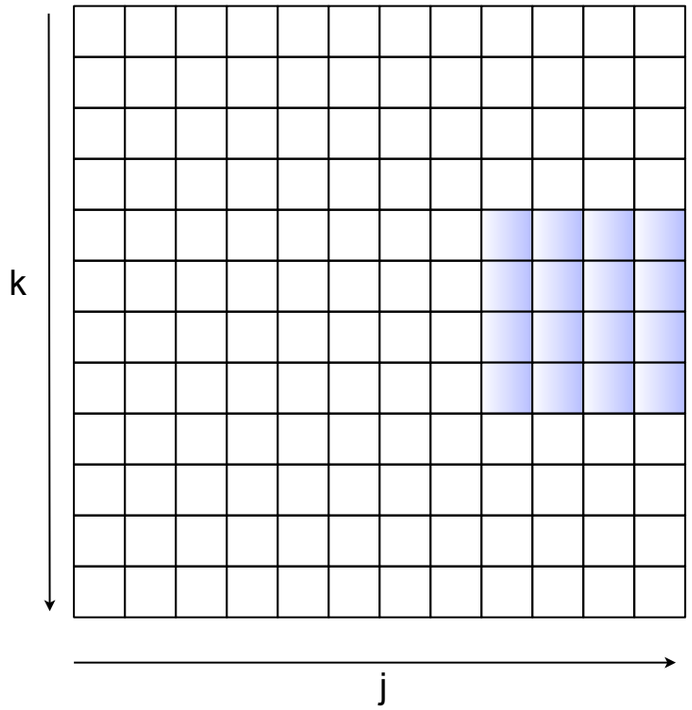
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

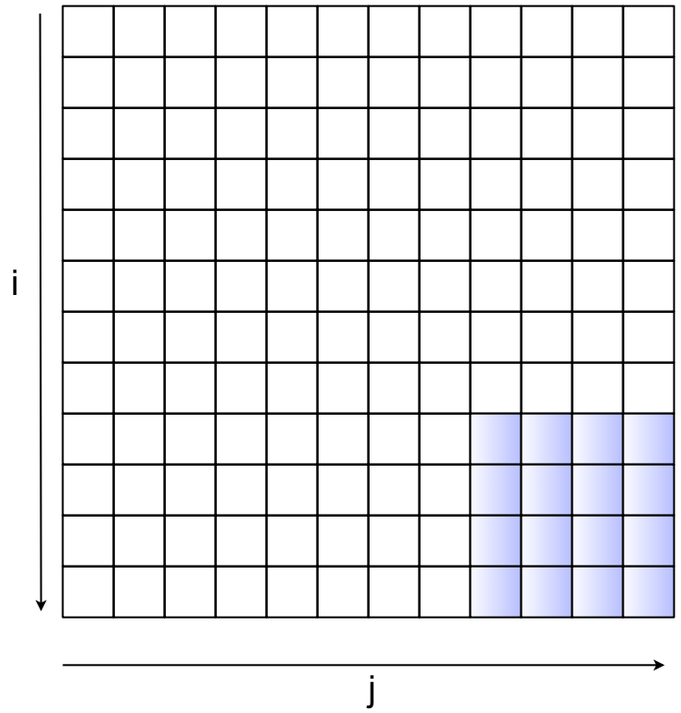
Array A



Array B



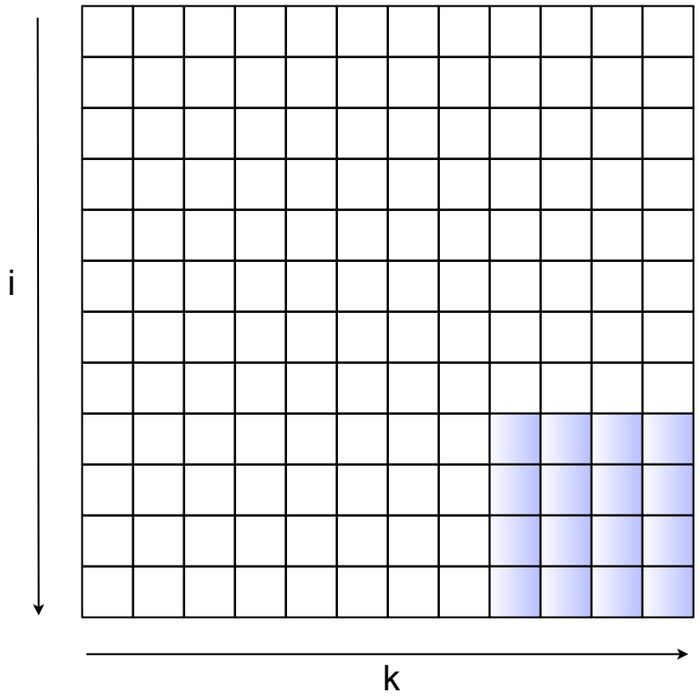
Array C



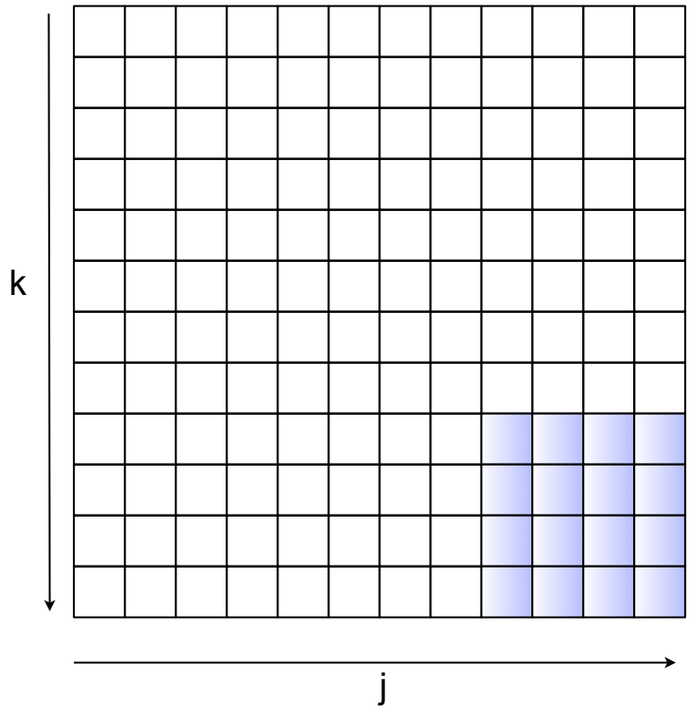
Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks

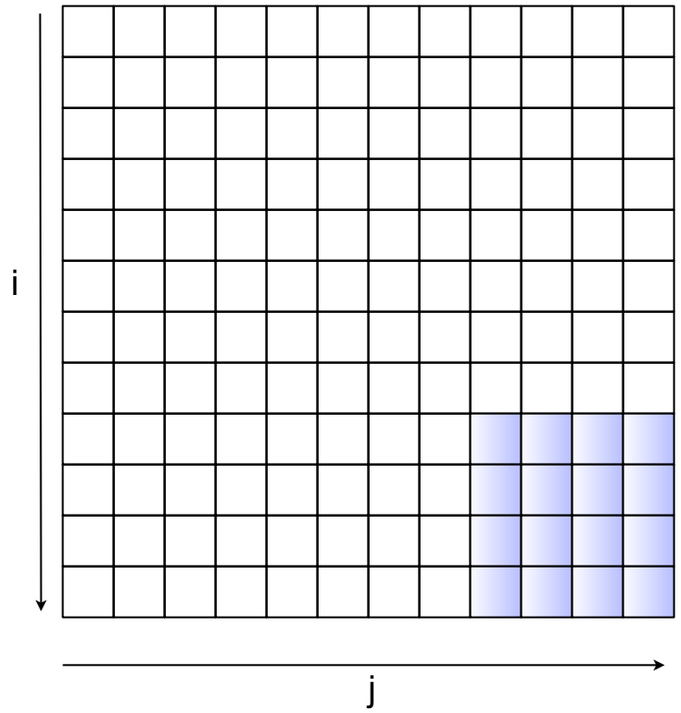
Array A



Array B

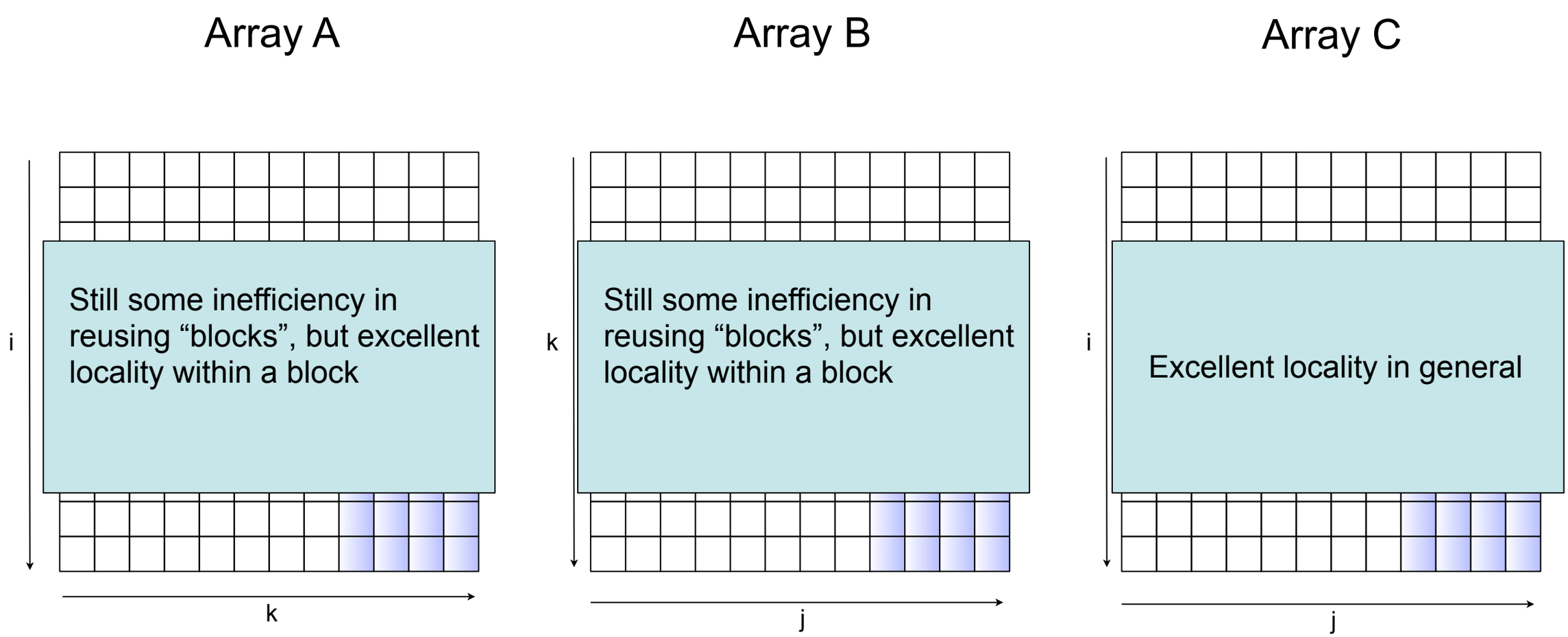


Array C



Optimisation - Loop Blocking - Matrix Multiply

- Assume we can hold 12 cache lines at a time, length 4 data elements
- Use a fixed block size of 4
- Process is repeated, shifting sub-blocks



Optimisation - Loop Fusion

- **Loop Fusion**:
  Merge multiple loops into a single loop.
- **Requirements:**
  - Two or more loops traversing the same range.
  - No loop-dependancies that would be broken by fusion.
- **When to Use:**
  - High overhead of loop conditional checks.
  - Poor temporal locality of memory accesses (between *loops*).
  - Multiple arrays storing intermediate values (between *loops*).

Optimisation - Loop Fusion

```
Kernel_Unfused()
{
    ....
    for(i=0;i<n;i++){
        b[i] = a[i]*2;
    }
    for(i=0;i<n;i++){
        c[i] = b[i] + 4;
    }
    for(i=1;i<n;i++){
        d[i] = c[i-1] - 5;
    }
    ...
}
```

Unfused, we have the cost of writing to the temporary array, as well as multiple loop overheads

```
Kernel_Fused()
{
    ....
    for(i=0;i<n;i++){
        b[i] = a[i]*2;
        c[i] = b[i] + 4;
    }

    for(i=1;i<n;i++){
        d[i] = c[i-1] - 5;
    }
    ...
}
```

Fusing eliminates the overhead of a loop, and improves the temporal locality of b[i].

The third loop cannot be fused due to the loop range.

```
Kernel_Fused_Peeled()
{
    ...
    b[0] = a[0]*2;
    c[0] = b[0]+4;

    for(i=1;i<n;i++){
        b[i] = a[i]*2;
        c[i] = b[i] + 4;
        d[i] = c[i-1] - 5;
    }
}
```

Loop peeling allows us to modify the loop range and merge the two loops.

The temporal locality of c[i-1], c[i] is improved.

```
Kernel_Fused_Peeled_NI()
{
    ....
    for(i=1;i<n;i++){
        d[i] = ((a[i-1]*2)+4)-5;
    }
    ...
}
```

The merged loops allow us to eliminate b[] and c[] entirely (assuming they are not reused outside the kernel.)

The use of a[i-1] also allows us to eliminate the loop dependency entirely.

Optimisation - Loop Fission

- **Loop Fission:**
  Improve locality by splitting loops (opposite of fusion).
- **Requirements:**
  - One loop with many (unrelated) operations.
- **When to Use:**
  - Poor temporal locality of memory accesses (between *loop iterations*).
  - Registers spilling into cache.

Optimisation - Loop Fission

```
unfissioned() {
    ...
    for(i=0;i<n;i++) {
        a[i] = a[i] * 0.3;
        b[i] = b[i] * 0.6;
    }
    ...
}
```

```
fissioned(){
    ...
    for(i=0;i<n;i++) {
        a[i] = a[i] * 0.3;
    }
    for(i=0;i<n;i++) {
        b[i] = b[i] * 0.6;
    }
}
```

Operations on *a* and *b* are independant of one another, but during the course of the loop the cache is shared between the two

The cache is now dedicated to operations on *a* in the first loop, and operations on *b* in the second loop, improving temporal locality in each.

Optimisation - Loop Unrolling

- **Loop Unrolling**:
  Expand a loop body to contain multiple iterations
- **Requirements:**
  - No broken inter-loop dependencies
- **When to Use:**
  - High overhead of loop conditional checks.
  - Multiple arrays/data values storing intermediate values (between *loop iterations*).
- Can be hand-unrolled (i.e. hard-coded unrolling) or implemented by the compiler with the correct flags/pragmas.
- #pragma unroll(n) written before a loop declaration will unroll that loop by a factor of n (within reason).

Optimisation - Loop Unrolling

```
sum()
{
      ...
      for(i=0;i<n;i++)
      {
            a[i] = b[i] + 4;
      }
      ...
}
```

```
sum_unrolled_factor_4()
{
      ...
      int unroll_n = (n/4)*4;
      for(i=0; i<unroll_n; i+=4){
            a[i]     = b[i] + 4;
            a[i+1] = b[i+1] + 4;
            a[i+2] = b[i+2] + 4;
            a[i+3] = b[i+3] + 4;
      }

      for(;i<n;i++){
            a[i] = b[i] + 4;
      }
      ...
}
```

Our loop now increases in increments of 4, the unroll factor

The unrolled loop copies the body q times, where q is the unroll factor. Each copy has an index matching that of the loop iteration it replaces.

If n is not a factor of 4 we must catch the remainder using a second “tidyup” loop!

Optimisation - Loop Pipelining

- **Loop Pipelining**:
  Reorder independent operations across iterations within a loop to enable instruction pipelining, overlapping their execution.
- **Requirements:**
  - Loop iterations are independent (to at least the pipeline length).
- **When to Use:**
  - CPU is capable of instruction pipelining.

Optimisation - Loop Pipelining

- **Problem:**
  - Three operations, A, B and C in a loop, each of which is dependent upon the completion of the previous operation.
  - A CPU capable of instruction pipelining cannot execute dependent instructions in parallel, yet the iterations of the loop are independent of one another

- **Solution**:
  - Refactor code to initiate the next iteration before the current iteration is complete.
  - Completion of current iteration and initiation of next iteration are independent - can exploit instruction level parallelism to overlap.

Optimisation - Loop Pipelining

	Iteration			
Step	0	1	2	3
0	A			
1	B			
2	C			
3		A		
4		B		
5		C		
6			A	
7			B	
8			C	
9				A
10				B
11				C

Serial - Iteration 1 cannot begin until the completion of iteration 2, nor can instruction B (dependent on A) or instruction C (dependent on B) be completed until their respective dependency has finished.

	Iteration			
Step	0	1	2	3
0	A			
1	B	A		
2	C	B	A	
3		C	B	A
4			C	B
5				C
6				
7				
8				
9				
10				
11				

Pipelined - The prolog and epilog each take two steps each and can only be partially pipelined. The remainder of the iterations however consist of three independent instructions which can be executed in parallel with suitable hardware capability.

Optimisation - Loop Pipelining - Example

```
...
for(i=0;i<n;i++)
{
    b[i] = a[i]*2;
}
...
```

We start with a relatively simple kernel, with no inter-loop dependencies

```
...
int unroll = (n/3)*3;
for(i=0;i<unroll;i+=3)
{
    b[i] = a[i]*2;
    b[i+1] = a[i+1]*2;
    b[i+2] = a[i+2]*2;
}
for(;i<n;i++)
{
    b[i] = a[i]*2;
}
...
```

With the application of loop unrolling, we are able to reveal more independent operations within the same iteration

Optimisation - Loop Pipelining - Example

```
...
int unroll = (n/3)*3

for(i=0;i<unroll;i+=3)
{
     c1 = a[i];
     c1 = c1*2; c2 = a[i+1];
     b[i] = c1; c2 = c2 * 2; c3 = a[i+2];
     b[i+1] = c2; c3=c3*2;
     b[i+2] = c3;
}
for(;i<n;i++)
{
     b[i] = a[i]*2;
}
...
```

```
...
int unroll = (n/3)*3;
c1 = a[0];
c1 = c1*2; c2 = a[1];

for(i=0;i<(unroll-3);i++)
{
     b[i] = c1; c2 = c2 * 2; c3 = a[i+2];
     b[i+1] = c2; c3 = c3 * 2; c1 = a[i+3];
     b[i+2] = c3; c2 = c1 * 2; c2 = a[i+4];
}

for(;i<n;i++)
{
     b[i] = a[i]*2;
}

...
```

Prolog

Pipeline

Epilog

Reordering the code, we can see that each individual line represents a set of independent instructions (load, mult, store.)

The prolog and epilog represent the portions of our pipeline leading up to the full overlap capability. Within the loop body we see three independent operations, belonging to three separate iterations, which can now be looped upon with instruction pipelining capability.