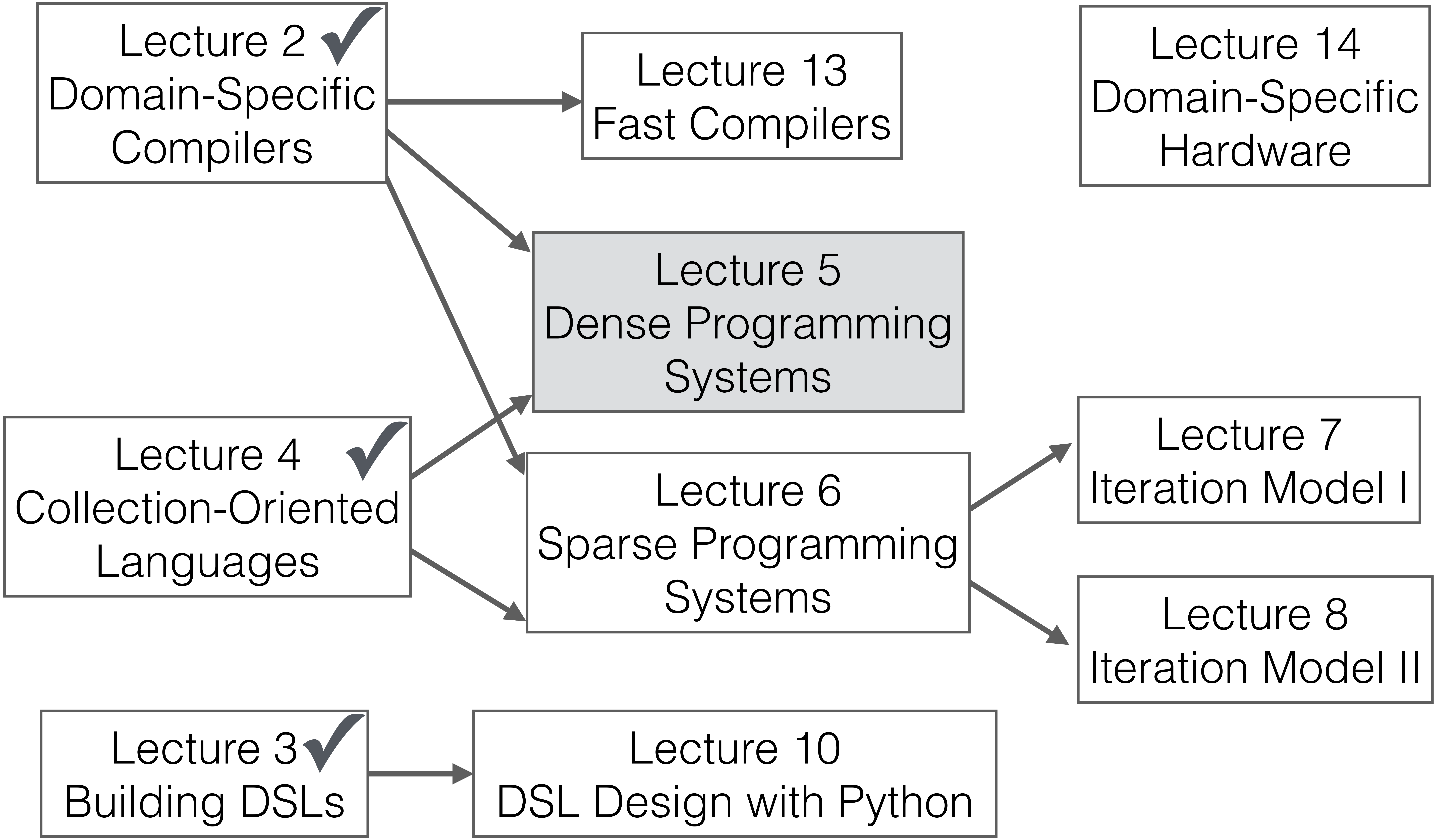
Lecture 5 - Dense Programming Systems

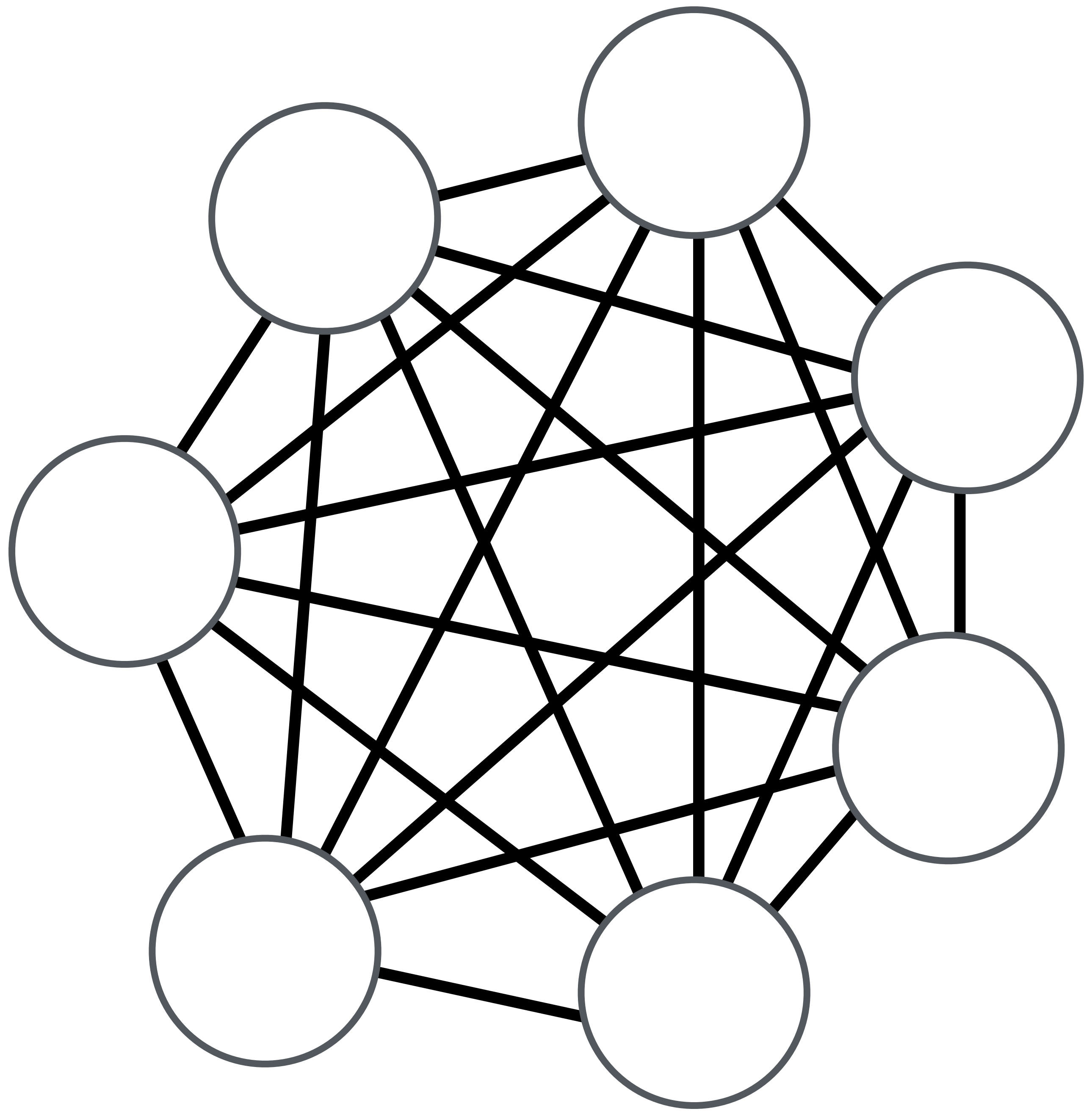
Stanford CS343D (Fall 2021)
Fred Kjolstad

Lecture Overview

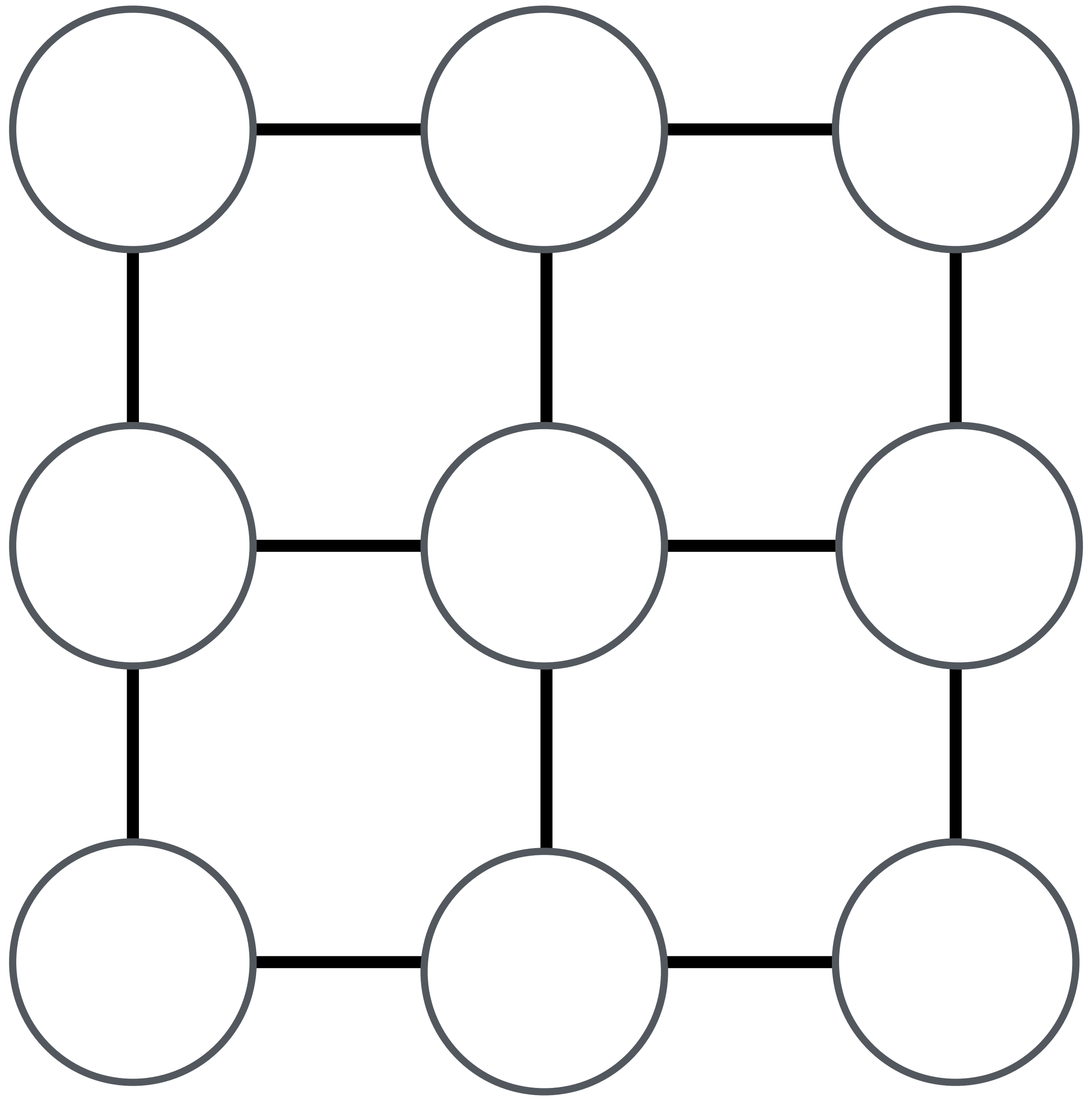


Terminology: Regular and Irregular

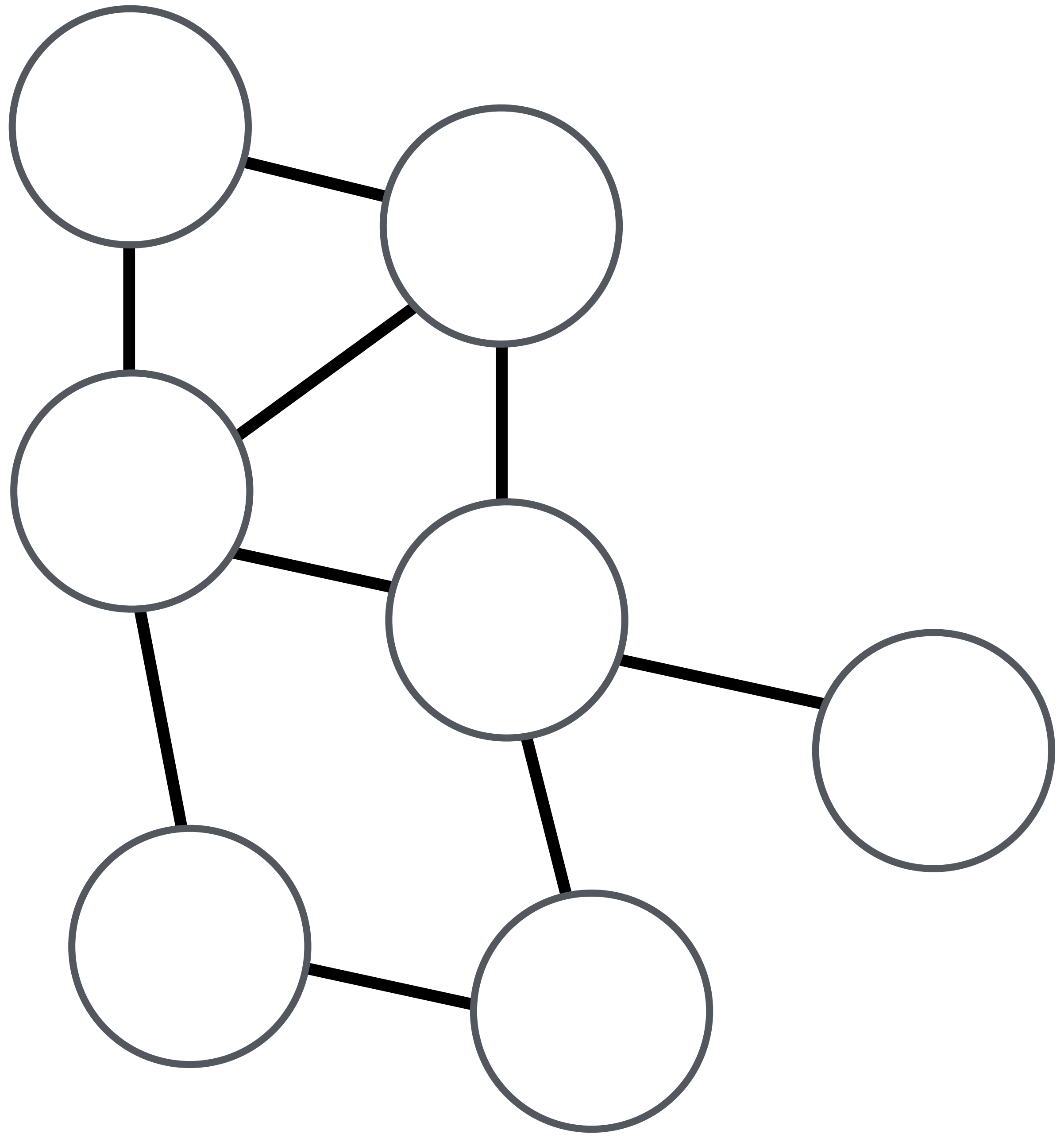
Fully Connected Regular System



Regular System

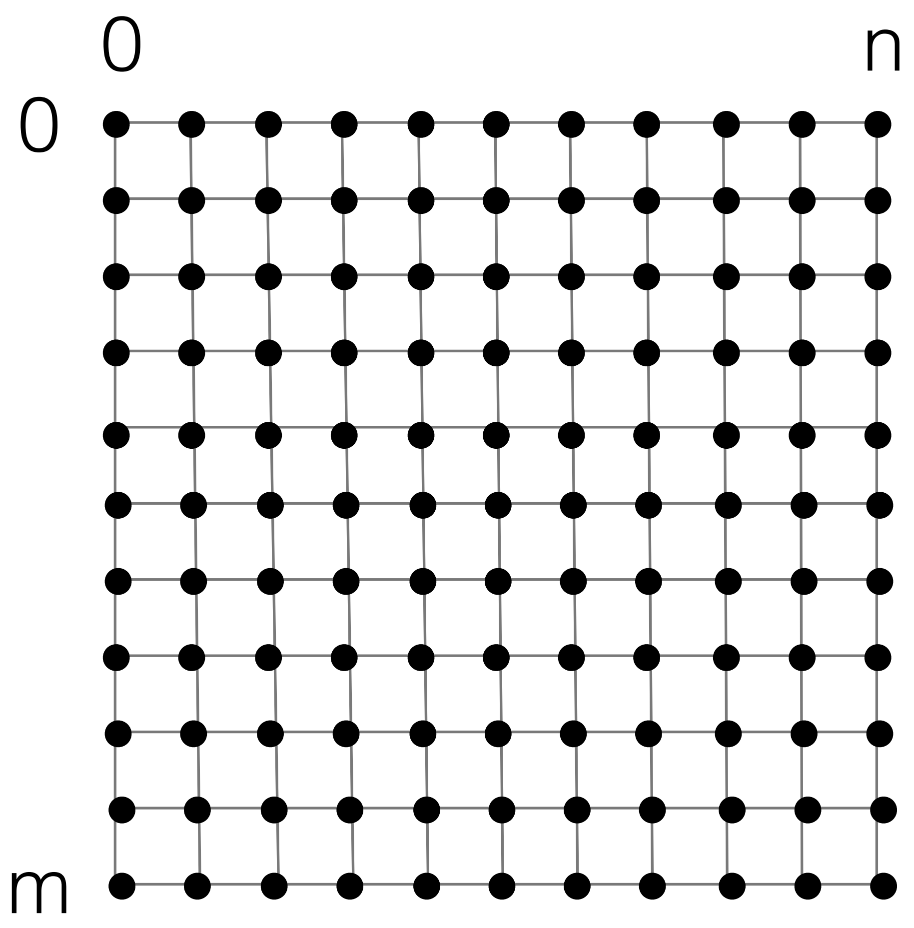


Irregular System



Terminology: Dense and Sparse

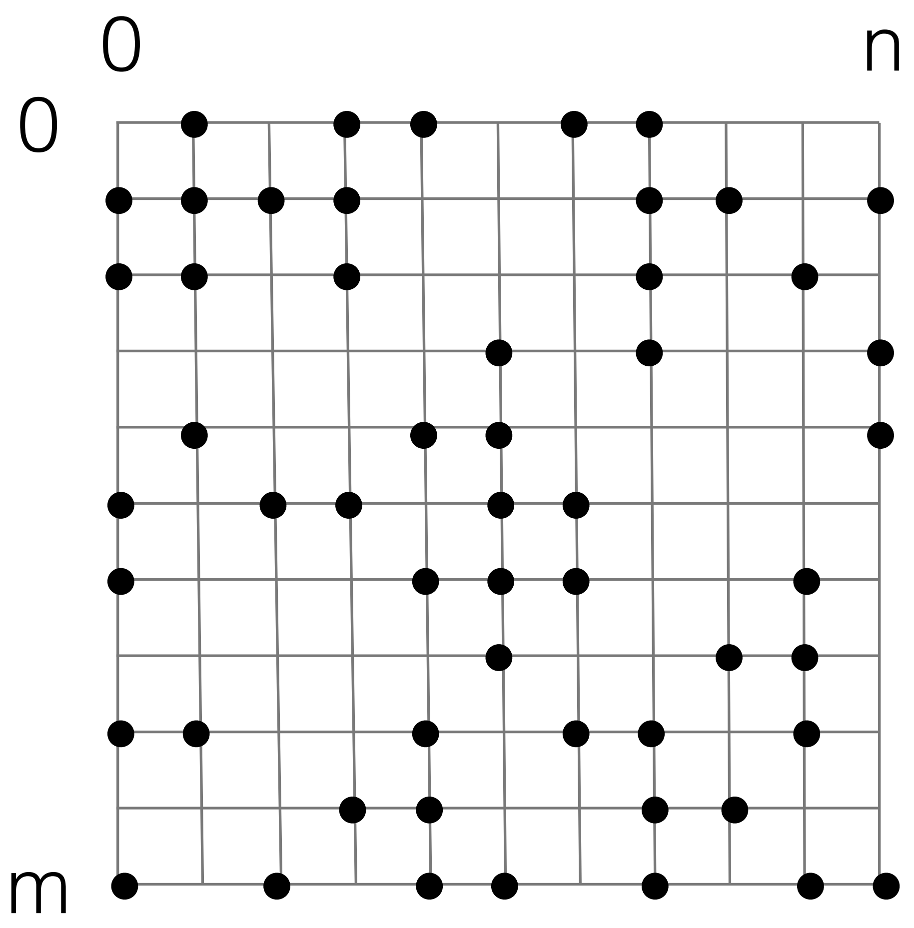
Dense loop iteration space




```
for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++) {
    y[i] += A[i*n+j] * x[j];
  }
}
```

$$y = Ax$$

Sparse loop iteration space

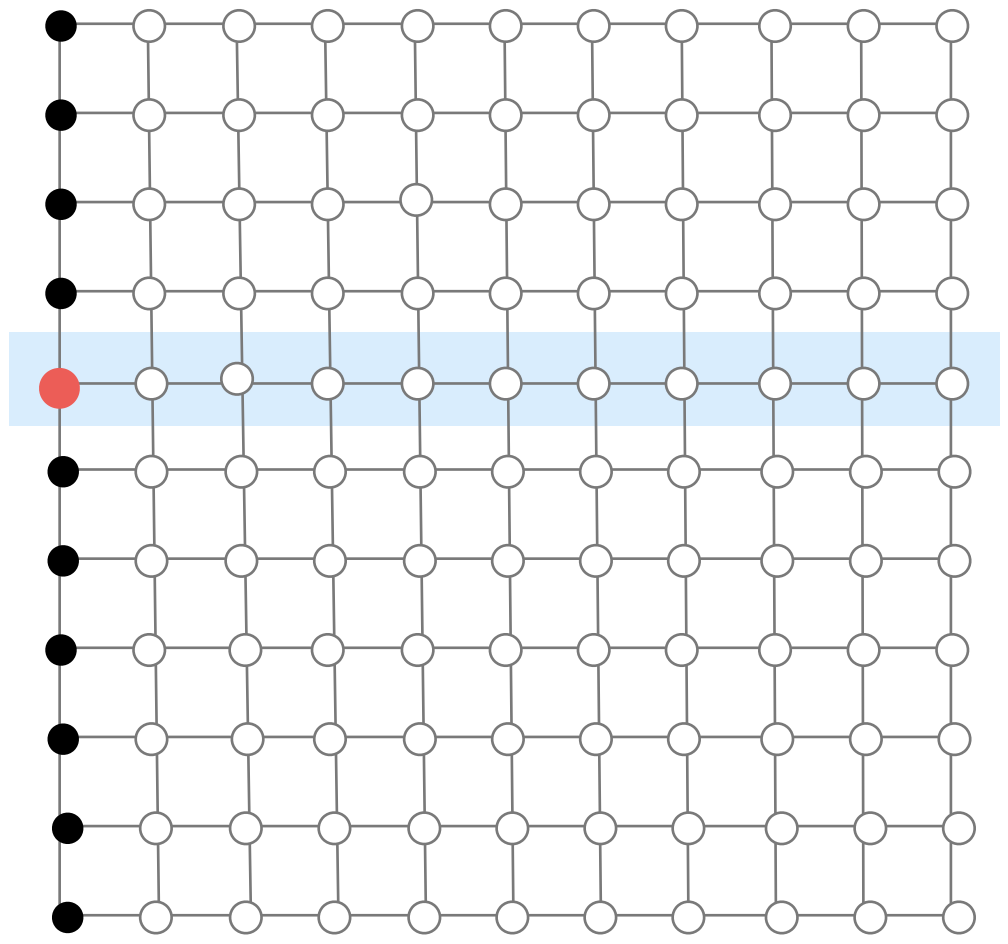



```
for (int i = 0; i < m; i++) {
 for (int pA = A2_pos[i]; pA < A_pos[i+1]; pA++) {
    int j = A_crd[pA];
    y[i] += A[pA] * x[j];
  }
}
```

$$y = Ax$$

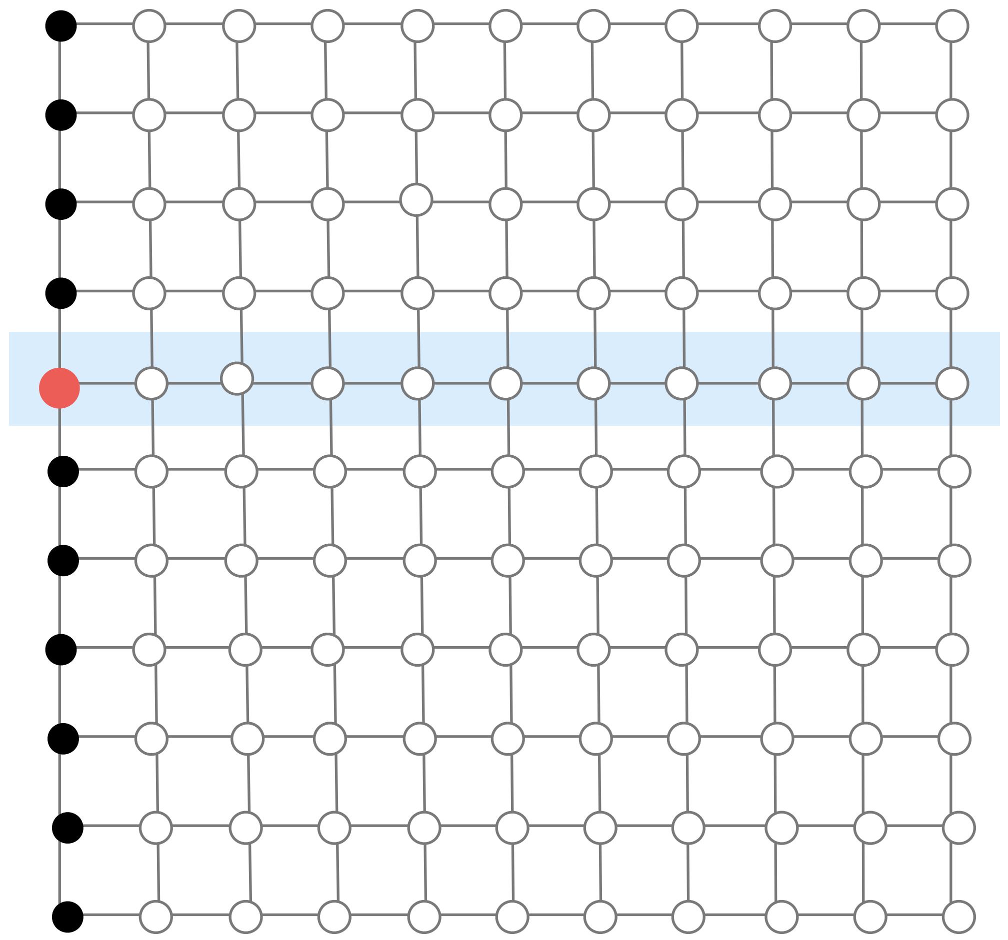
Dense applications

Dense Matrix-Vector Multiplication

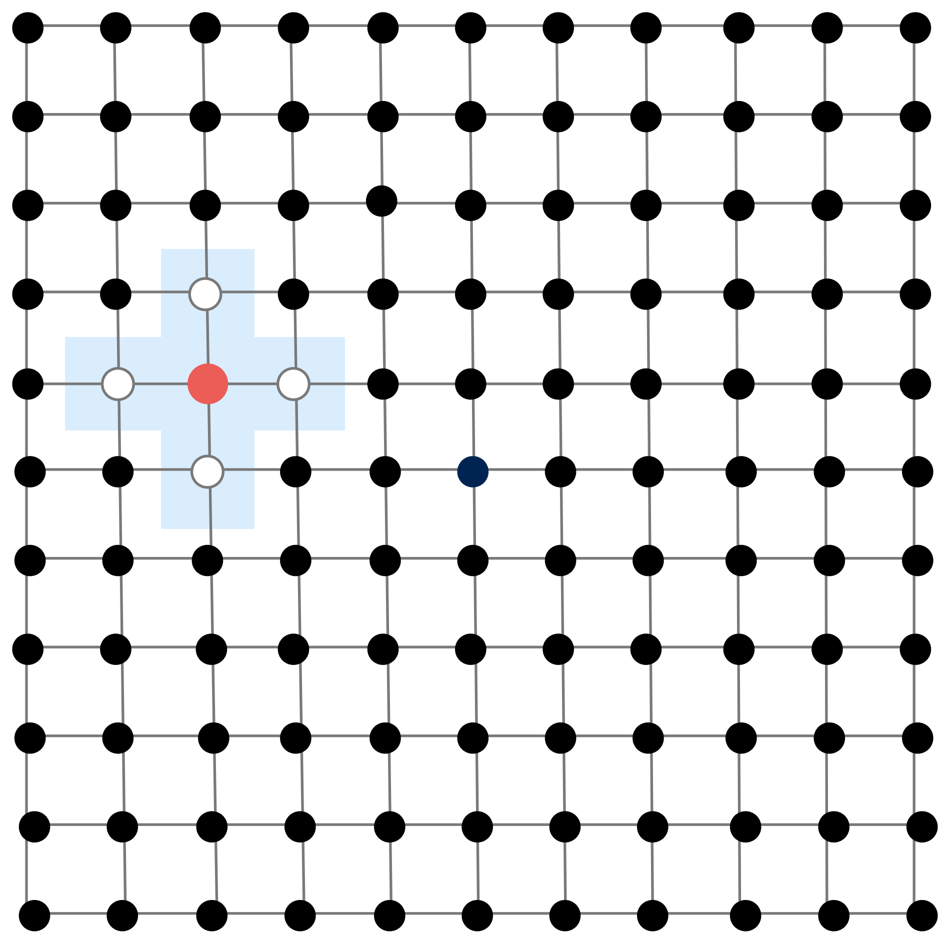


Dense applications

Dense Matrix-Vector Multiplication

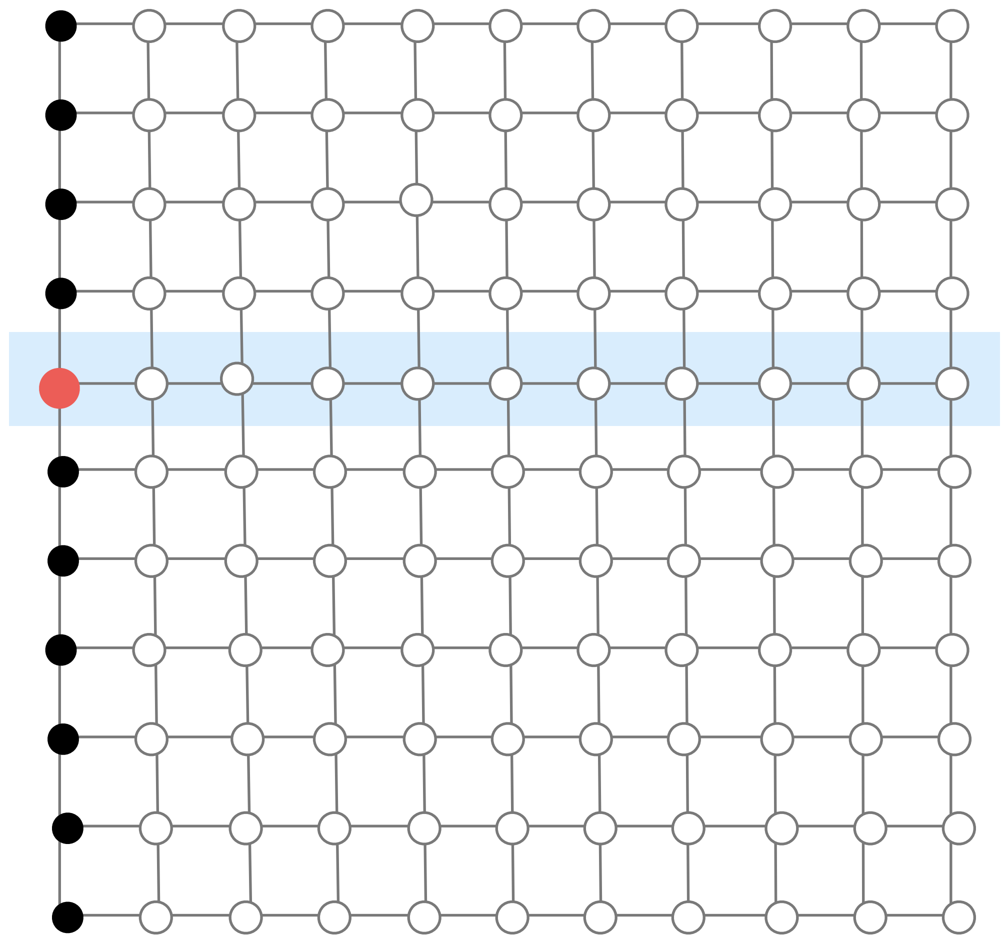


Stencils

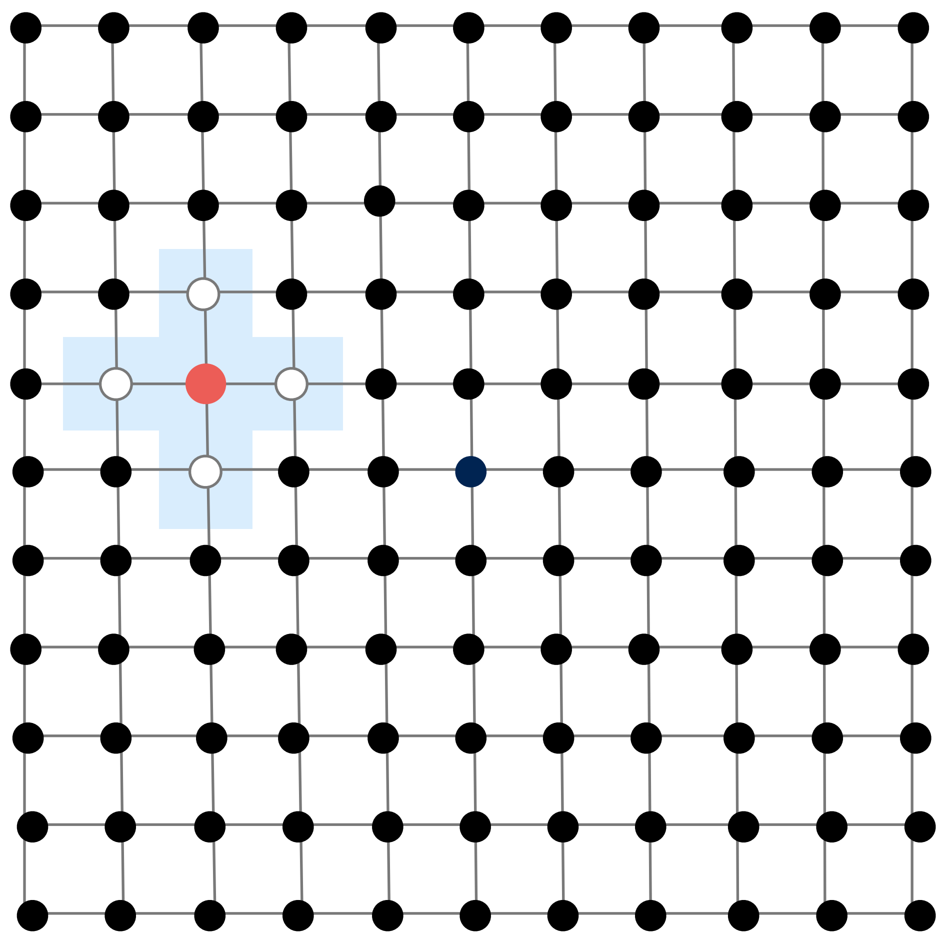


Dense applications

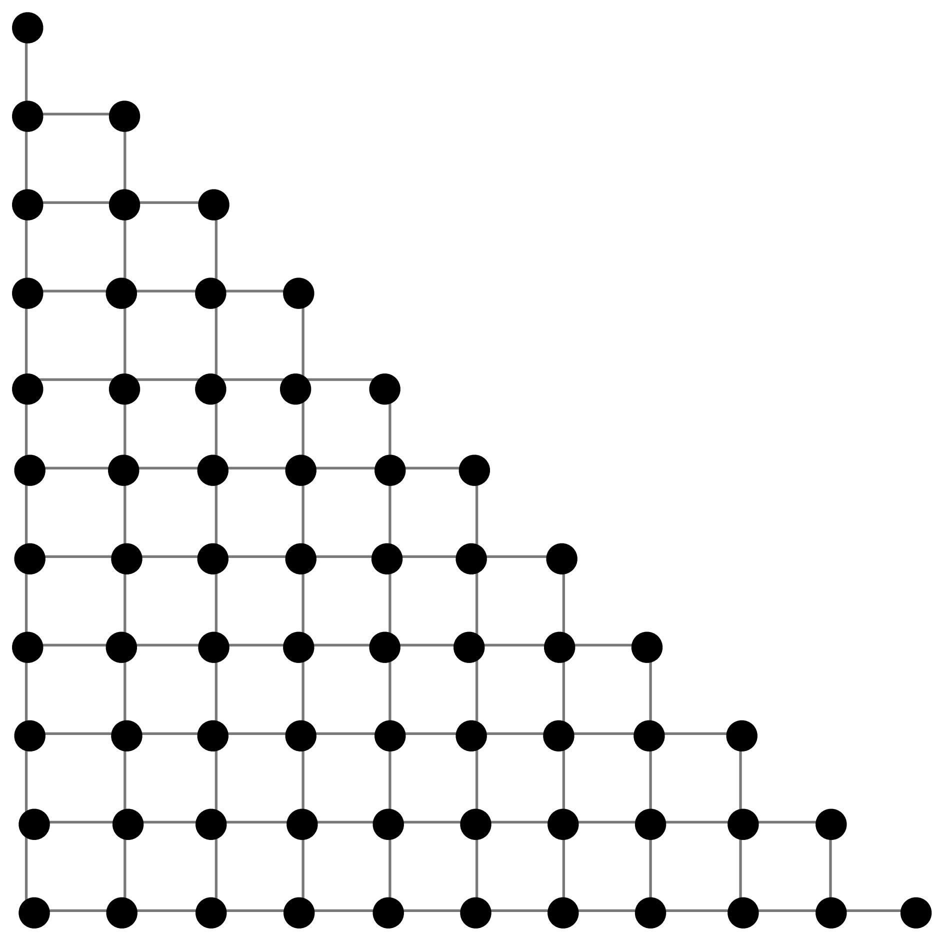
Dense Matrix-Vector Multiplication



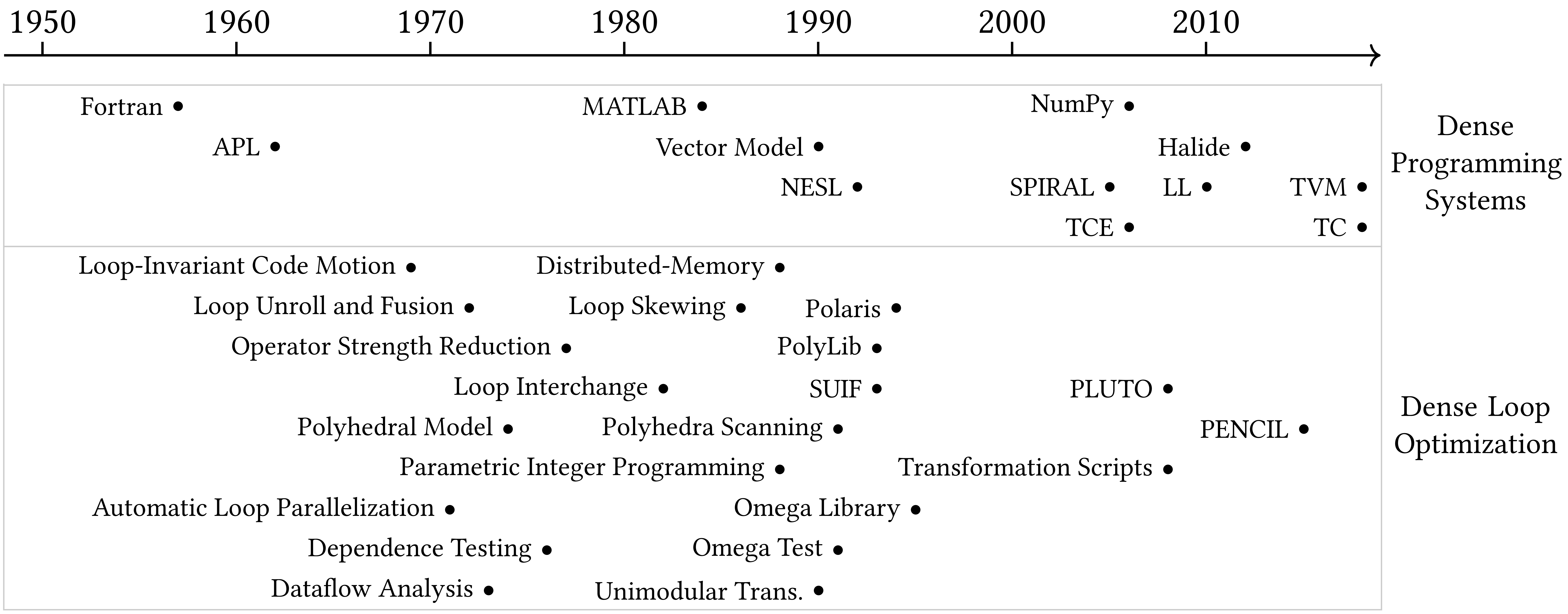
Stencils



Triagonal Solve



Timeline of some important developments in compilers and programming languages for dense compilers



Traditional compiler loop transformations

Reorder (interchange)

→

```
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    A[i][j] = B[i][j] + C[i][j];
```

```
for (int j=0; j<n; j++)
  for (int i=0; i<m; i++)
    A[i][j] = B[i][j] + C[i][j];
```

Traditional compiler loop transformations

Split (Stripmine)

→

```
for (int i=0; i<m; i++)
  a[i] = b[i] + c[i];
```

```
for (int k=0; k<m; k+=4)
  for (int i=k; i<k+4; i++)
    a[i] = b[i] + c[i];
```

Traditional compiler loop transformations

Vectorize →

```
for (int k=0; k<m; k+=4)
  for (int i=k; i<k+4; i++)
    a[i] = b[i] + c[i];
```

```
for (int k=0; k<m; k+=4)
  a[k:k+4] = b[k:k+4] + c[k:k+4];
```

Traditional compiler loop transformations

```
for (int i=0; i<m; i++)
  a[i] = b[i] + c[i];

for (int i=0; i<m; i++)
  d[i] = -b[i];
```

Fusion →

```
for (int i=0; i<m; i++)
  a[i] = b[i] + c[i];
  d[i] = -b[i];
```

Traditional compiler loop transformations

Collapse (flatten)

→

```
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    A[i*m+j] = -B[i*m+j];
```

```
for (int ij=0; ij<m*n; ij++)
  A[ij] = -B[ij];
```

Two models of loop optimization: source code rewrite and mathematical frameworks

Source Code Rewrite

```
for (int i=0; i<m; i++) {
  a[i] = b[i] + c[i];
}
```

split(4) →

```
for (int k=0; k<m; k+=4) {
  for (int i=k; i<k+4; i++) {
    a[i] = b[i] + c[i];
}
```

Mathematical loop optimization frameworks include the polyhedral model

Two models of loop optimization: source code rewrite and mathematical frameworks

Source Code Rewrite

```
for (int i=0; i<m; i++) {
  a[i] = b[i] + c[i];
}
```

split(4) →

```
for (int k=0; k<m; k+=4) {
  for (int i=k; i<k+4; i++) {
    a[i] = b[i] + c[i];
}
```

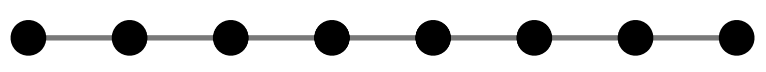
Mathematical Frameworks

```
for (int i=0; i<m; i++) {
  a[i] = b[i] + c[i];
}
```

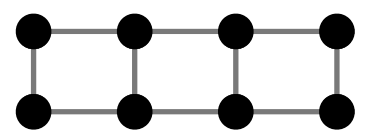
```
for (int k=0; k<m; k+=4) {
  for (int i=k; i<k+4; i++) {
    a[i] = b[i] + c[i];
}
```

convert to integer domain ↓

code generation ↑



split(4) →



Mathematical loop optimization frameworks include the polyhedral model

Optimizing dense codes require complex tradeoffs between parallelism, locality, and work efficiency

Clean C++: 9.94 ms per megapixel

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

Fast x86 C++: 0.9 ms per megapixel

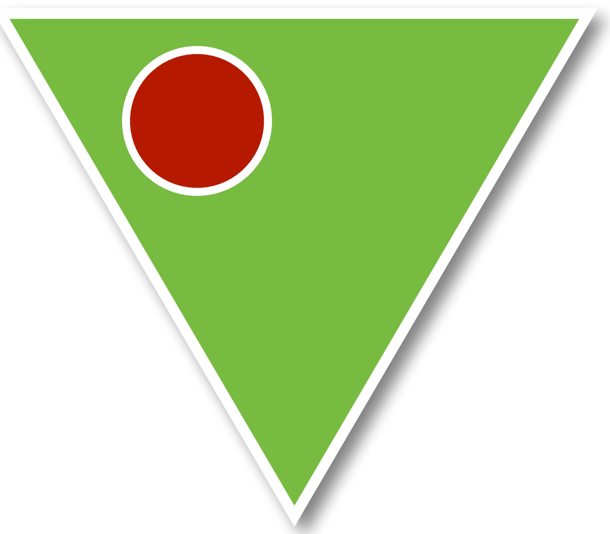
```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *Ragan-Kelley et al.* (2012)

Optimizing dense codes require complex tradeoffs between parallelism, locality, and work efficiency

Clean C++: 9.94 ms per megapixel

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

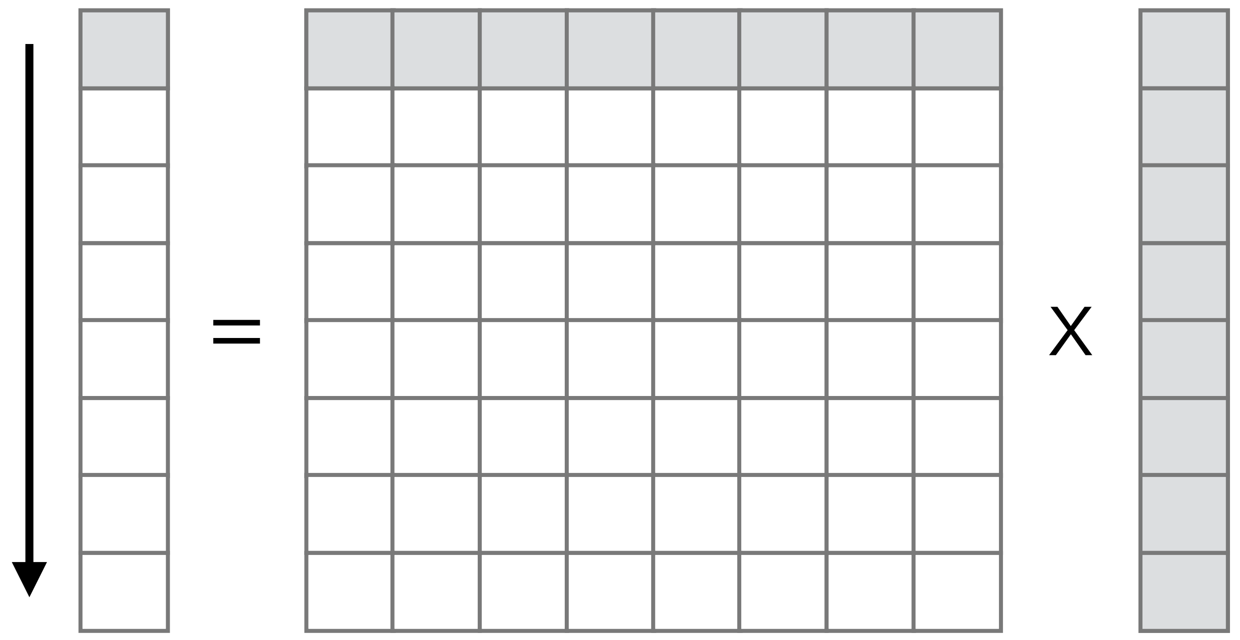


Fast x86 C++: 0.9 ms per megapixel

```
void fast_blur(const Image &in, Image &blurred) {
 _m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  _m128i a, b, c, sum, avg;
  _m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   _m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((_m128i*)(inPtr-1));
     b = _mm_loadu_si128((_m128i*)(inPtr+1));
     c = _mm_load_si128((_m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    _m128i *outPtr = (_m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

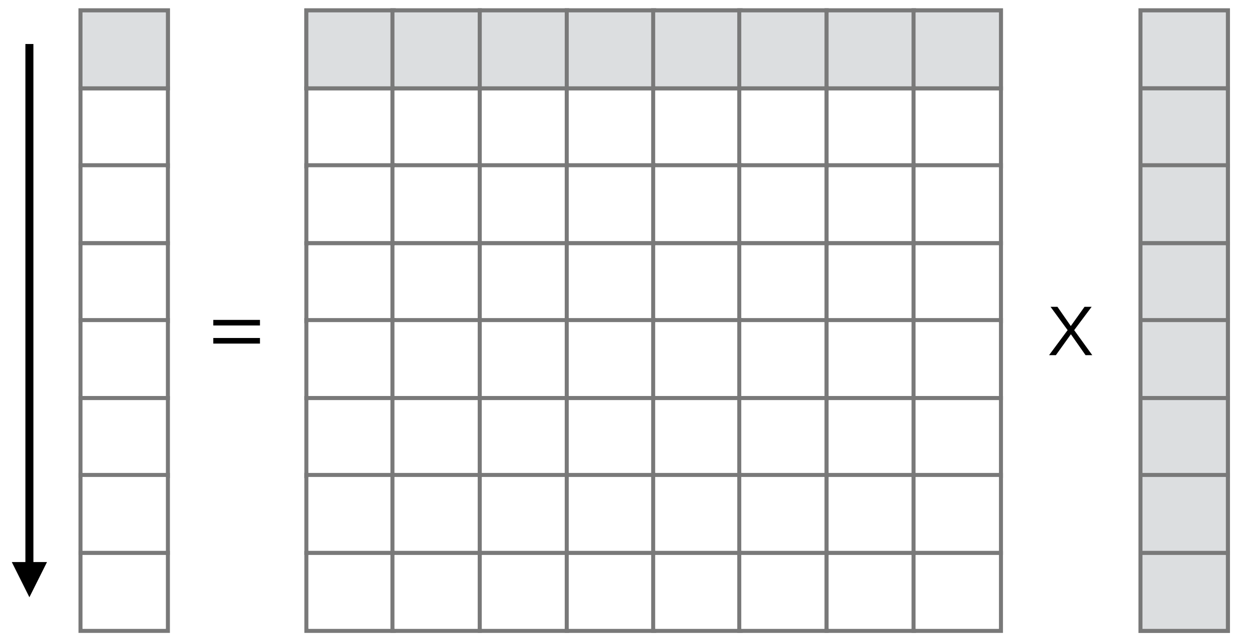
Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *Ragan-Kelley et al.* (2012)

Parallelism in matrix-vector multiplication

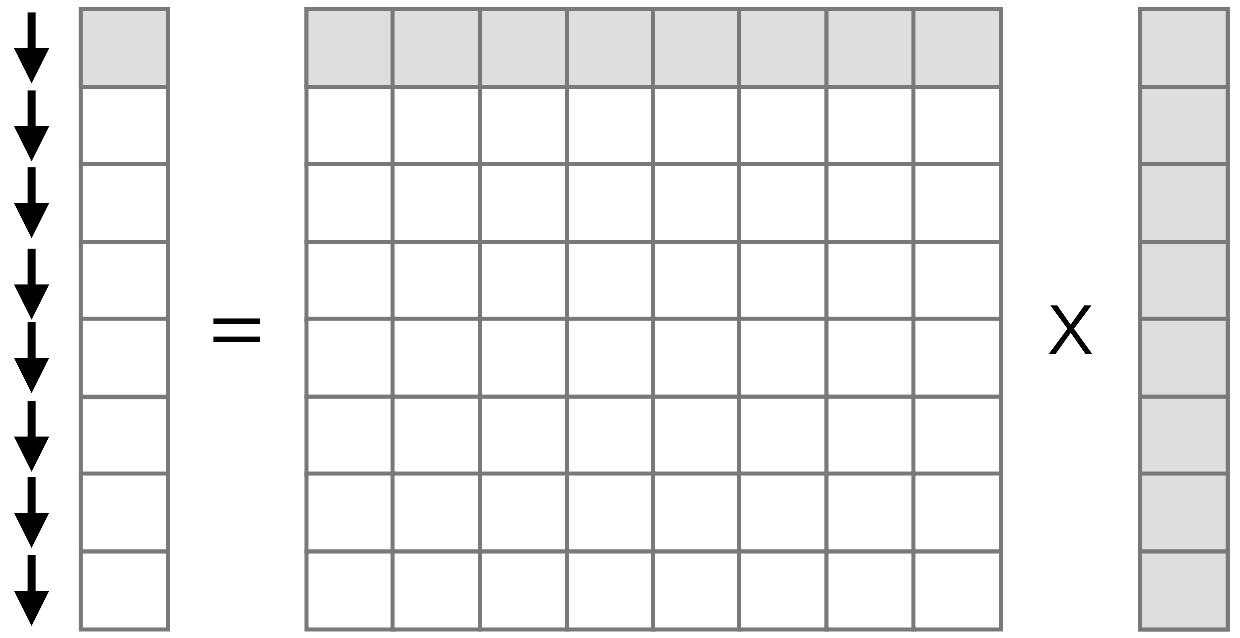


```
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    y[i] += A[i*n+j] * x[j];
```

Parallelism in matrix-vector multiplication

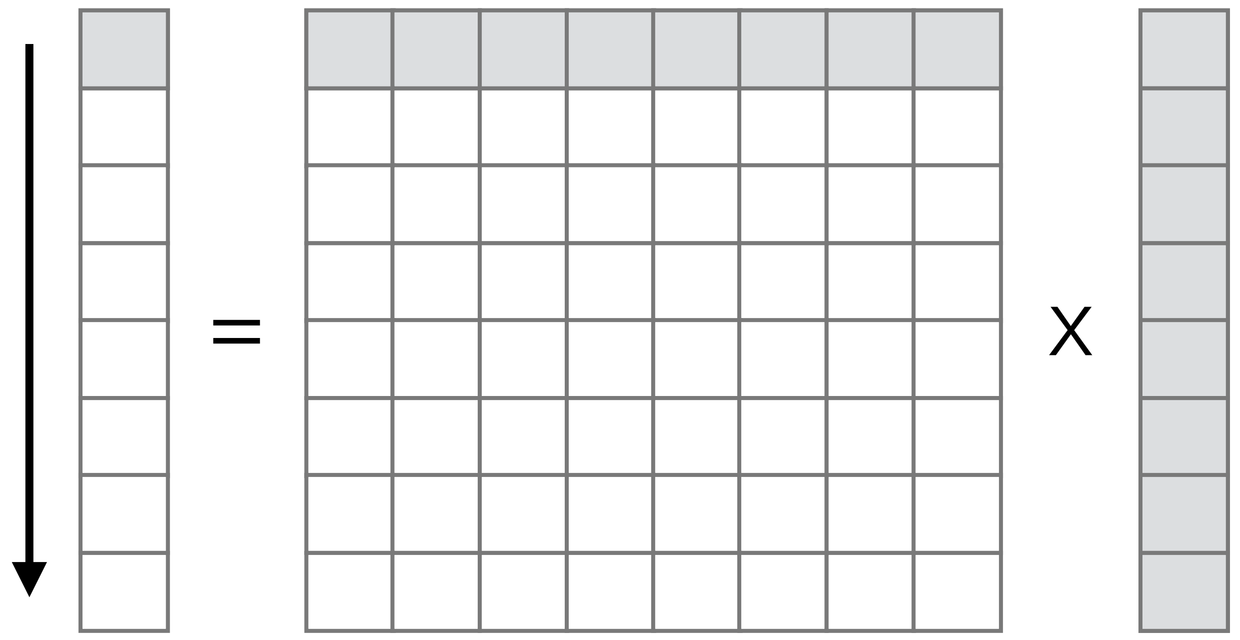



```
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    y[i] += A[i*n+j] * x[j];
```

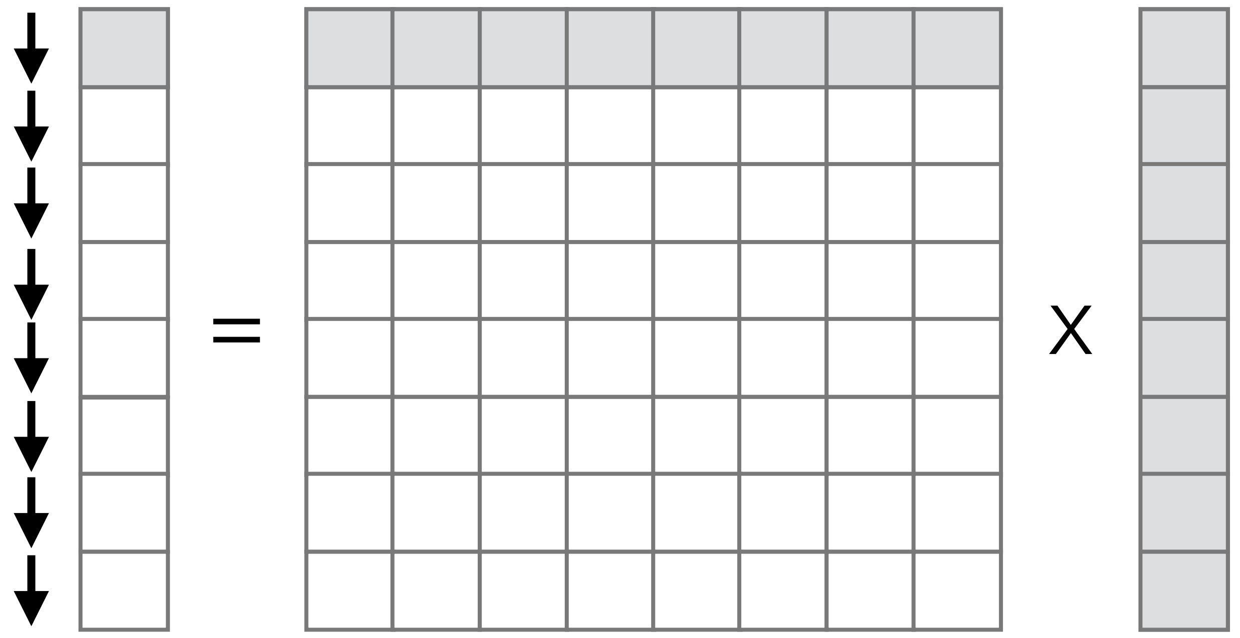



```
#pragma omp parallel for
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    y[i] += A[i*n+j] * x[j];
```

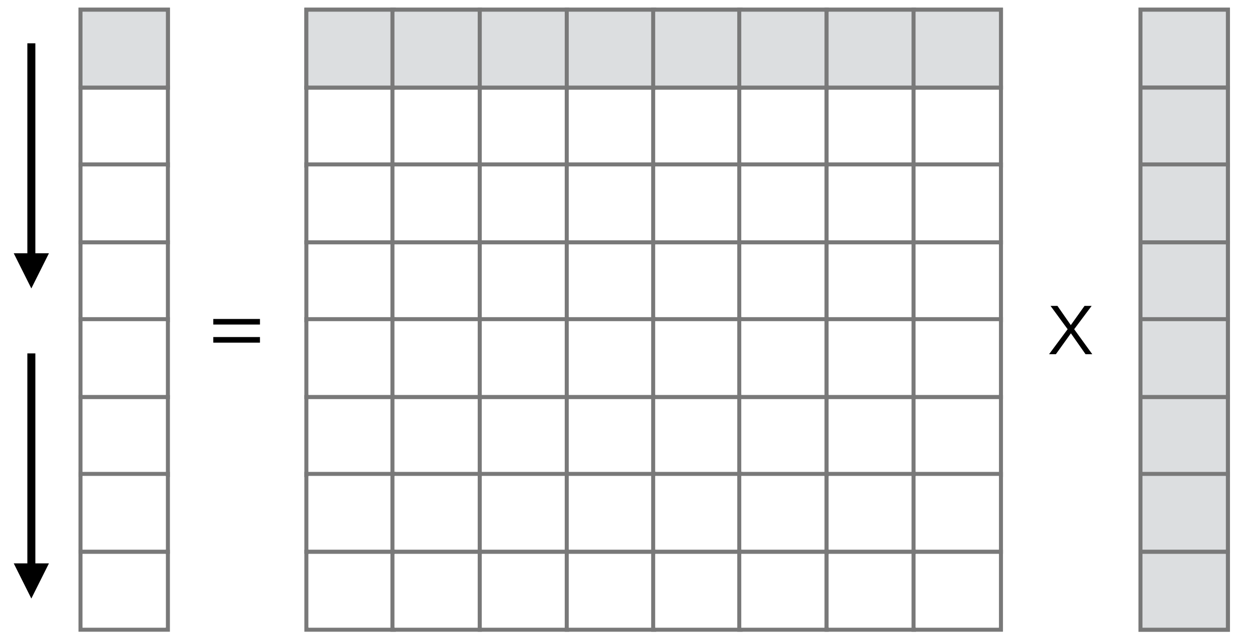
Parallelism in matrix-vector multiplication



```
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    y[i] += A[i*n+j] * x[j];
```

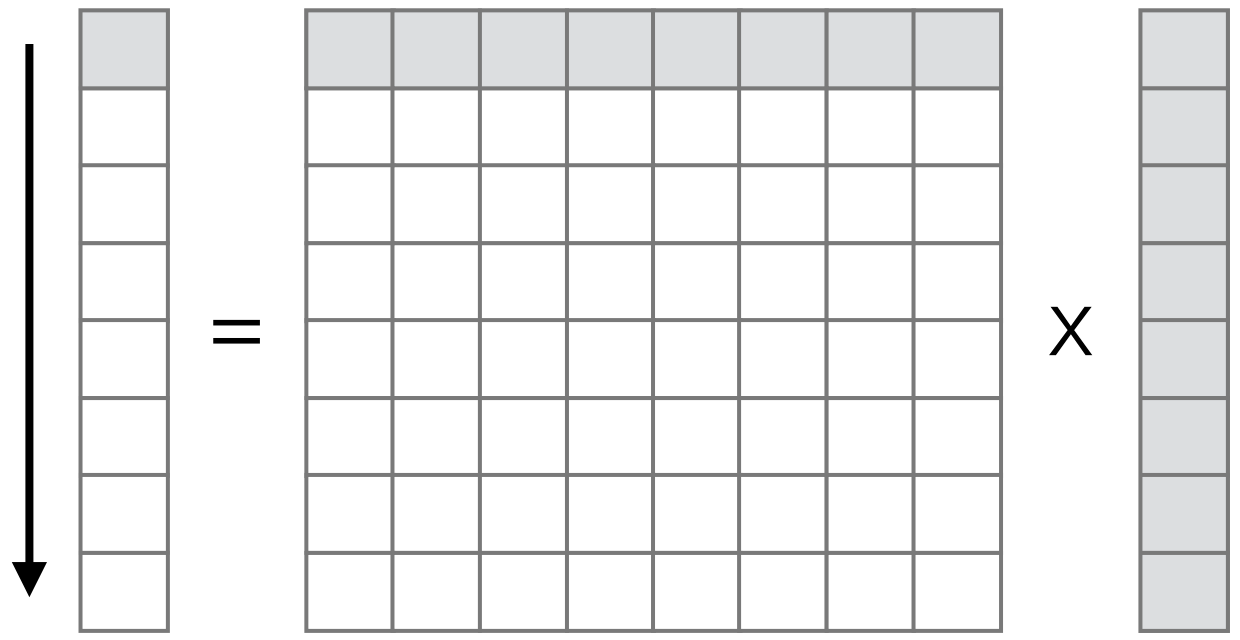


```
#pragma omp parallel for
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    y[i] += A[i*n+j] * x[j];
```

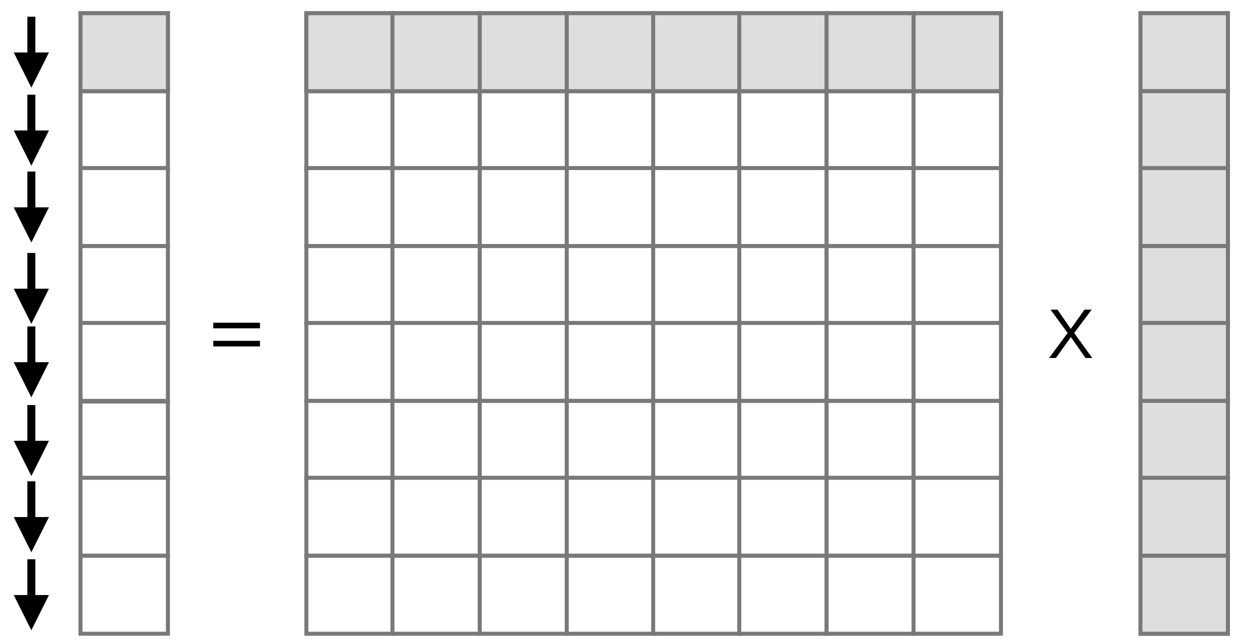


```
#pragma omp parallel for
for (int k=0; k<m; k+=4)
  for (int i=k; i<k+4; i++)
    for (int j=0; j<n; j++)
      y[i] += A[i*n+j] * x[j];
```

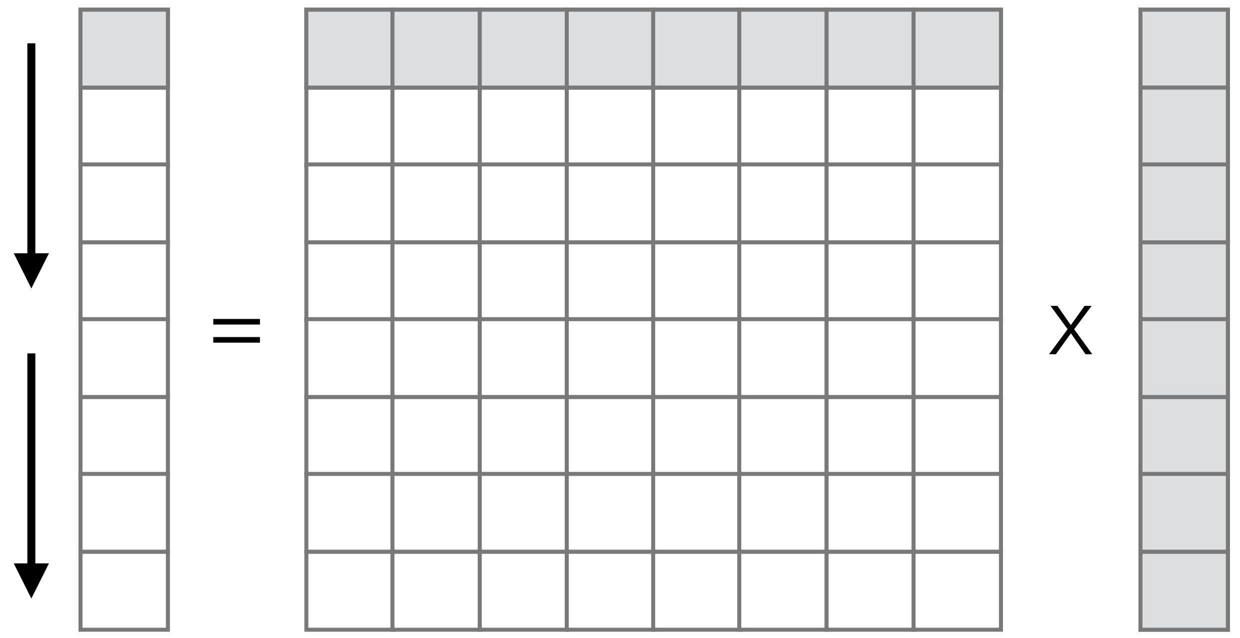
Parallelism in matrix-vector multiplication



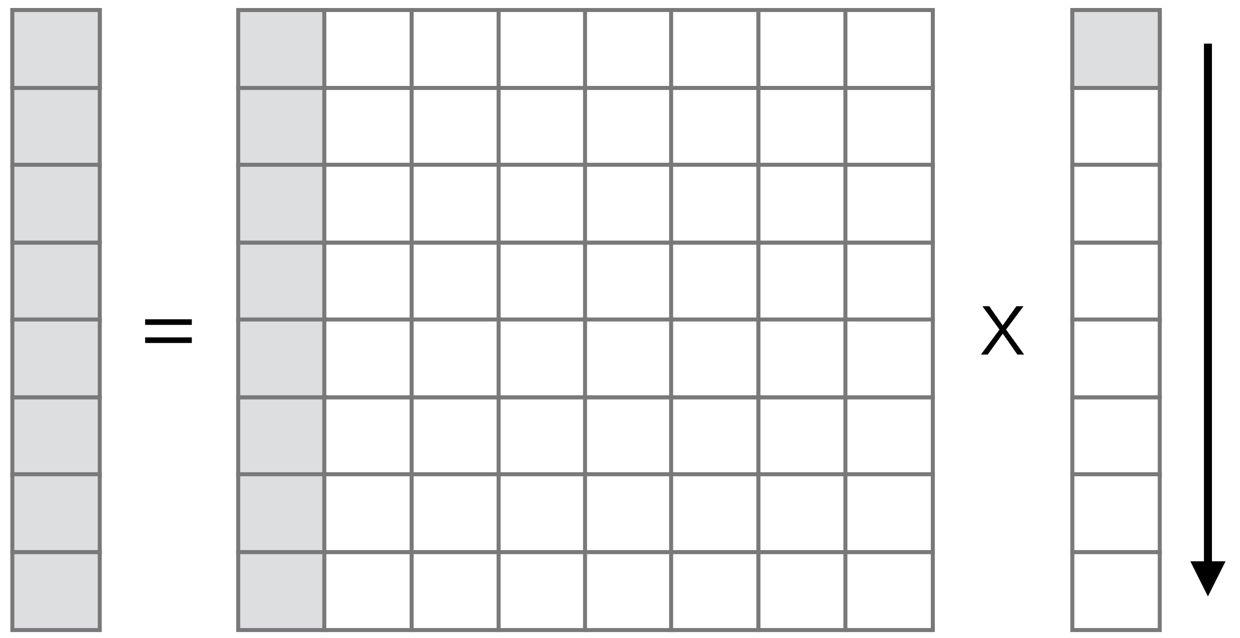

```
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    y[i] += A[i*n+j] * x[j];
```




```
#pragma omp parallel for
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    y[i] += A[i*n+j] * x[j];
```

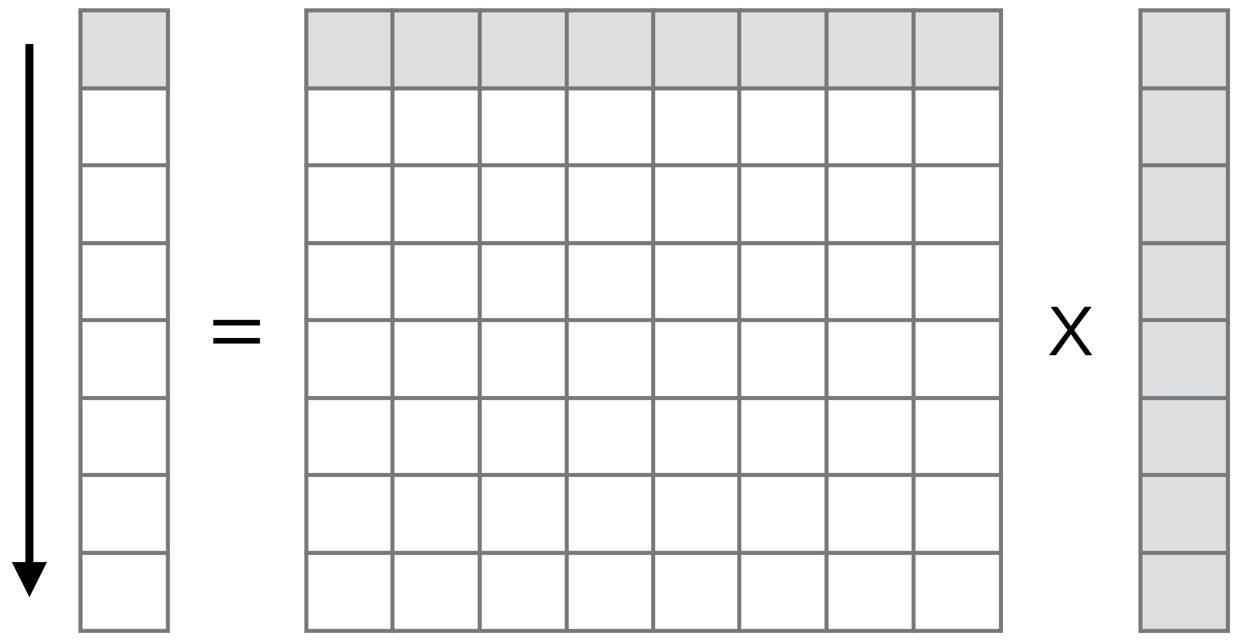



```
#pragma omp parallel for
for (int k=0; k<m; k+=4)
  for (int i=k; i<k+4; i++)
    for (int j=0; j<n; j++)
      y[i] += A[i*n+j] * x[j];
```

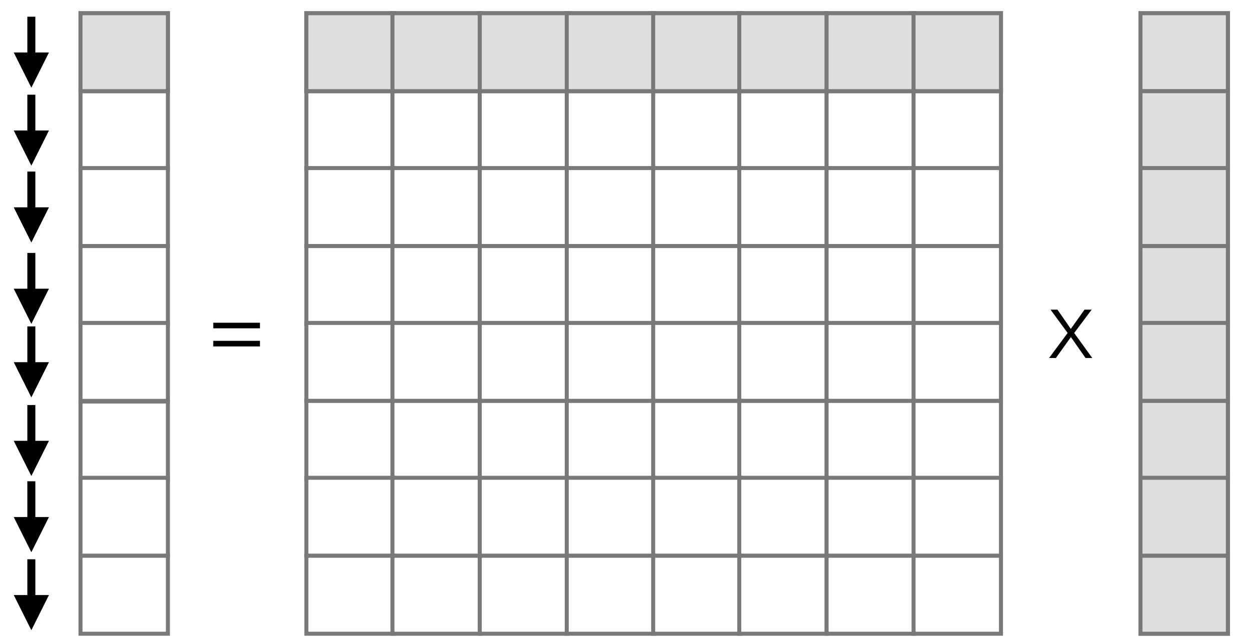



```
for (int j=0; j<n; j++)
  for (int i=0; i<m; i++)
    y[i] += A[i*n+j] * x[j];
```

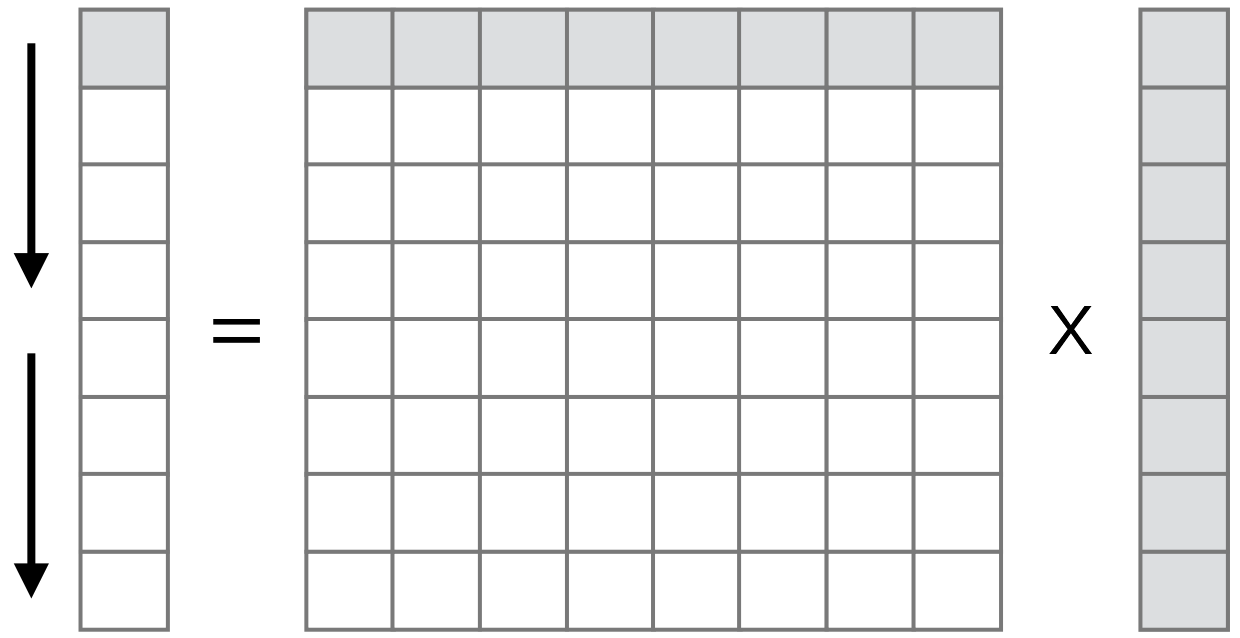
Parallelism in matrix-vector multiplication



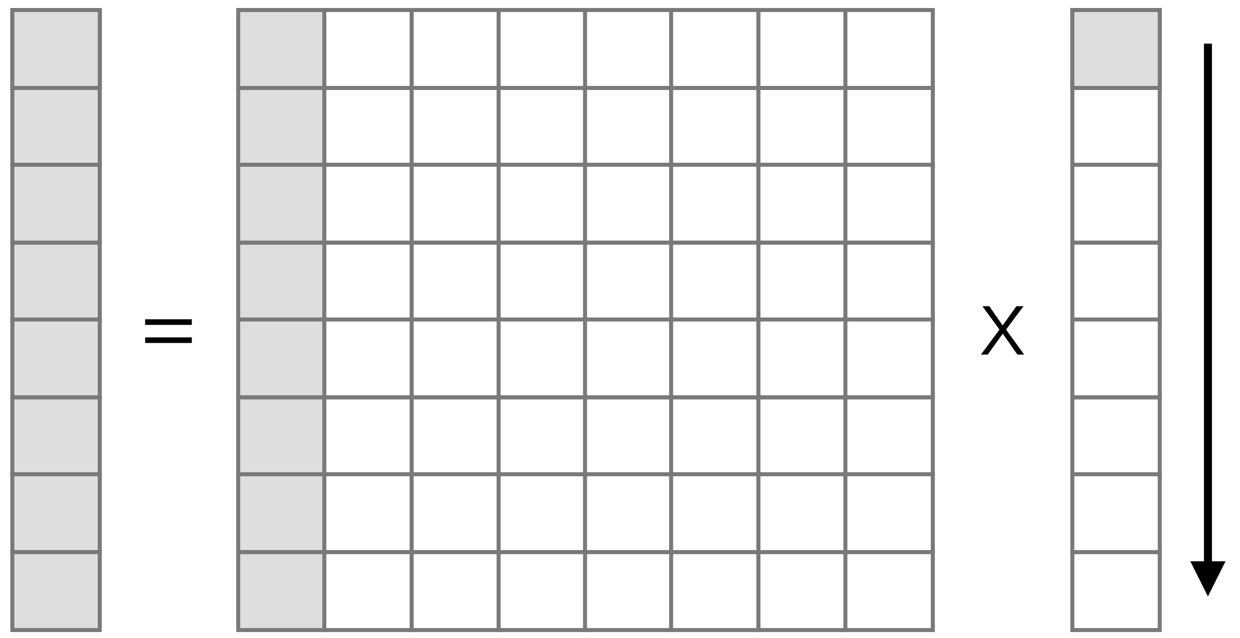

```
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    y[i] += A[i*n+j] * x[j];
```



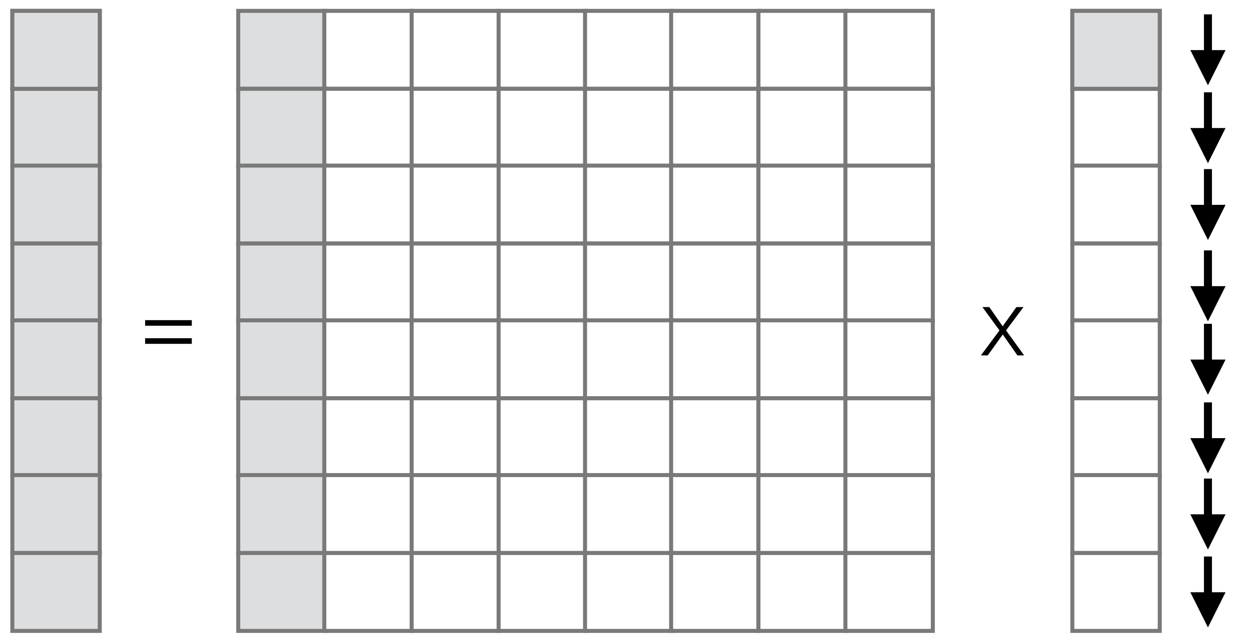

```
#pragma omp parallel for
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    y[i] += A[i*n+j] * x[j];
```




```
#pragma omp parallel for
for (int k=0; k<m; k+=4)
  for (int i=k; i<k+4; i++)
    for (int j=0; j<n; j++)
      y[i] += A[i*n+j] * x[j];
```

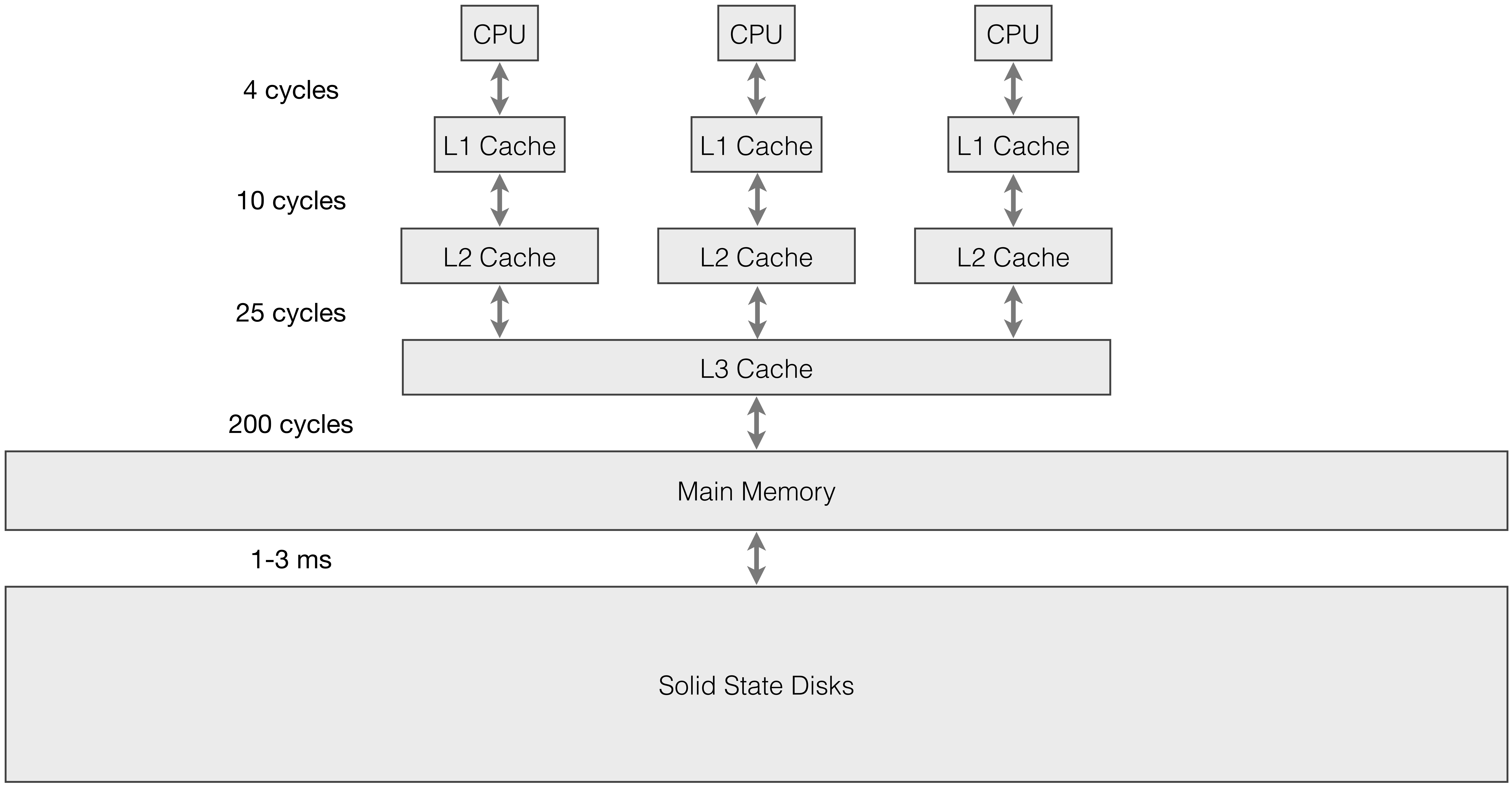



```
for (int j=0; j<n; j++)
  for (int i=0; i<m; i++)
    y[i] += A[i*n+j] * x[j];
```

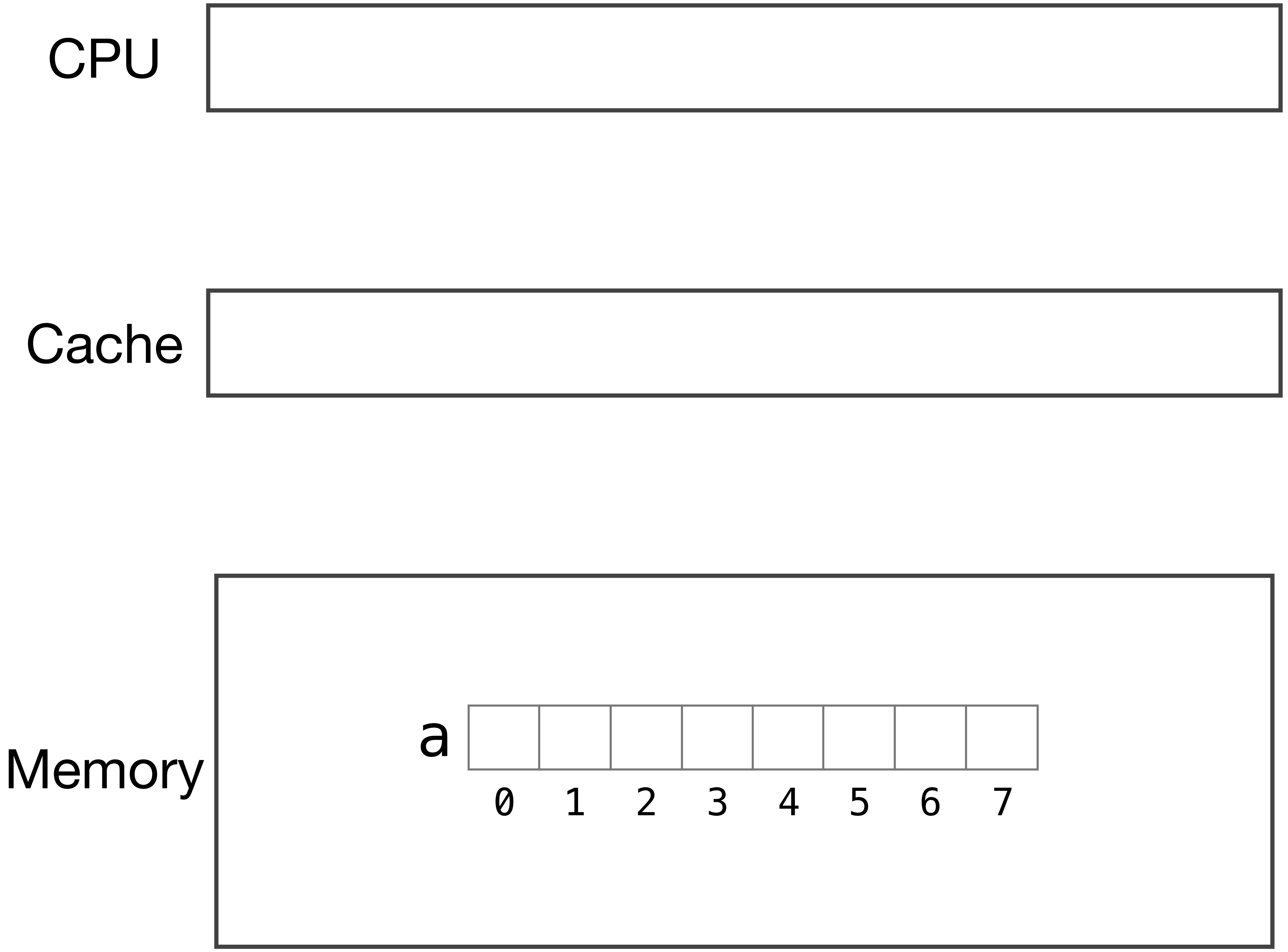



```
#pragma omp parallel for
for (int j=0; j<n; j++)
  for (int i=0; i<m; i++)
    #pragma omp atomic
    y[i] += A[i*n+j] * x[j];
```

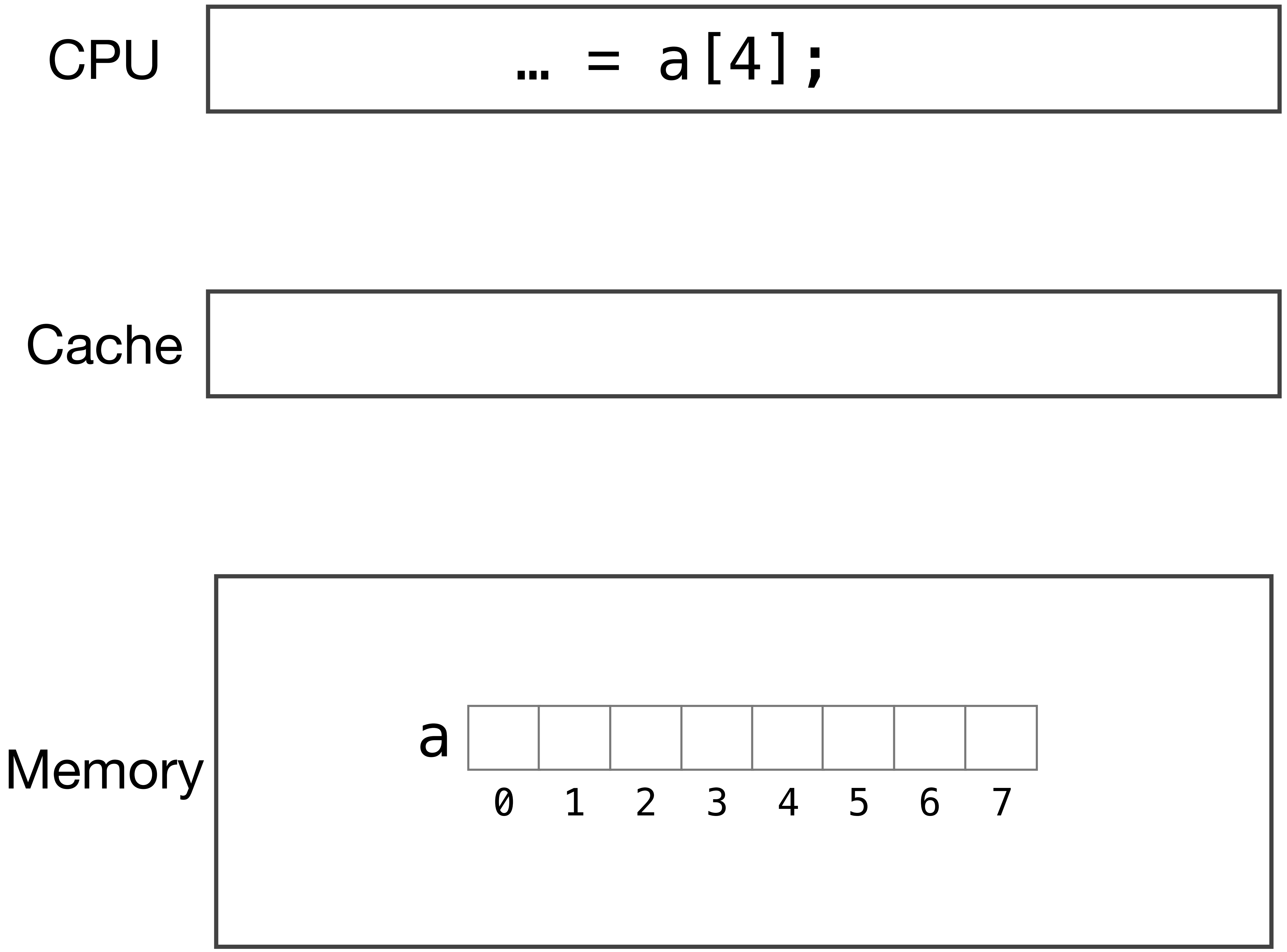
Cache Hierarchies with typical latencies



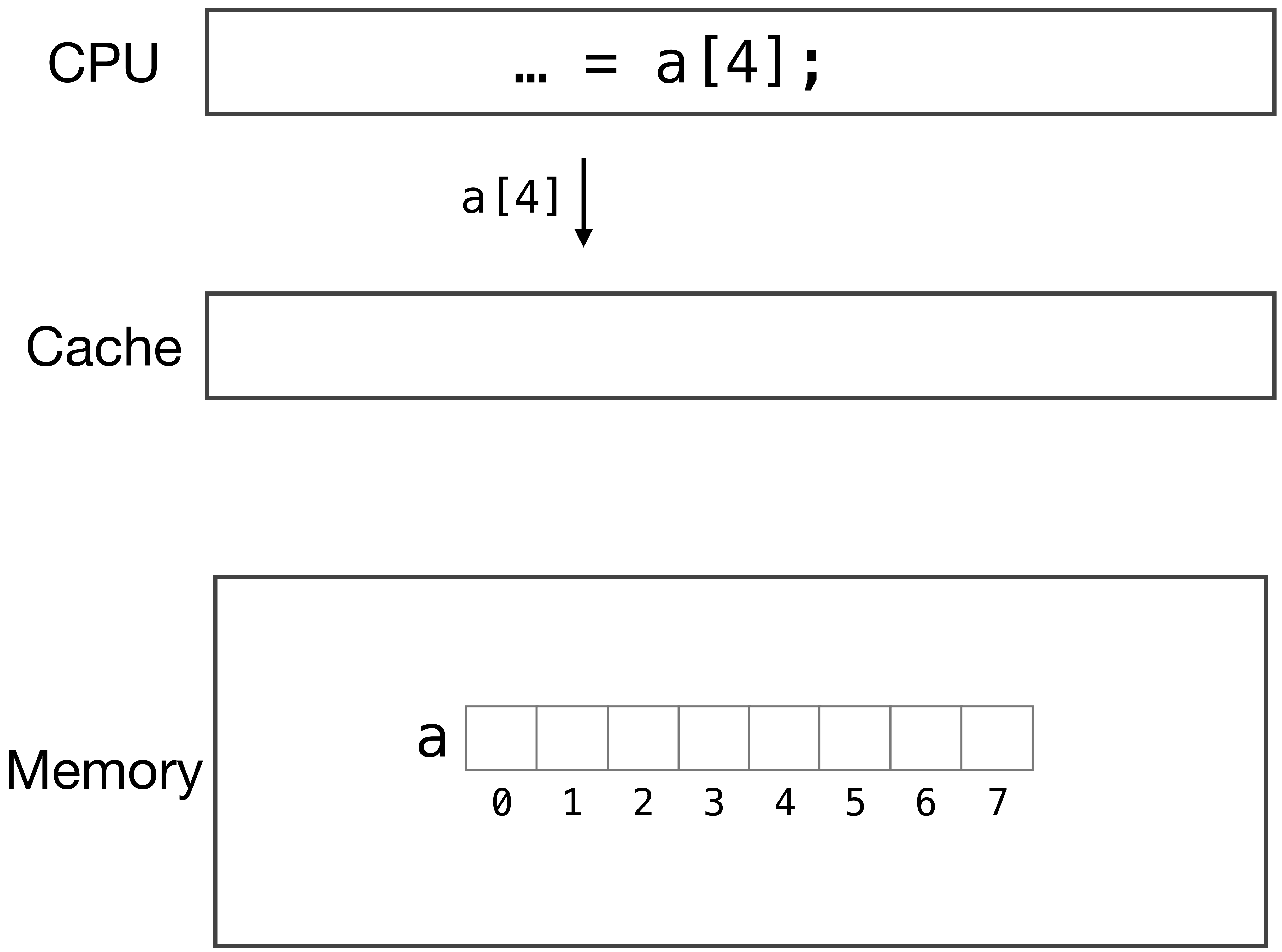
Spatial locality



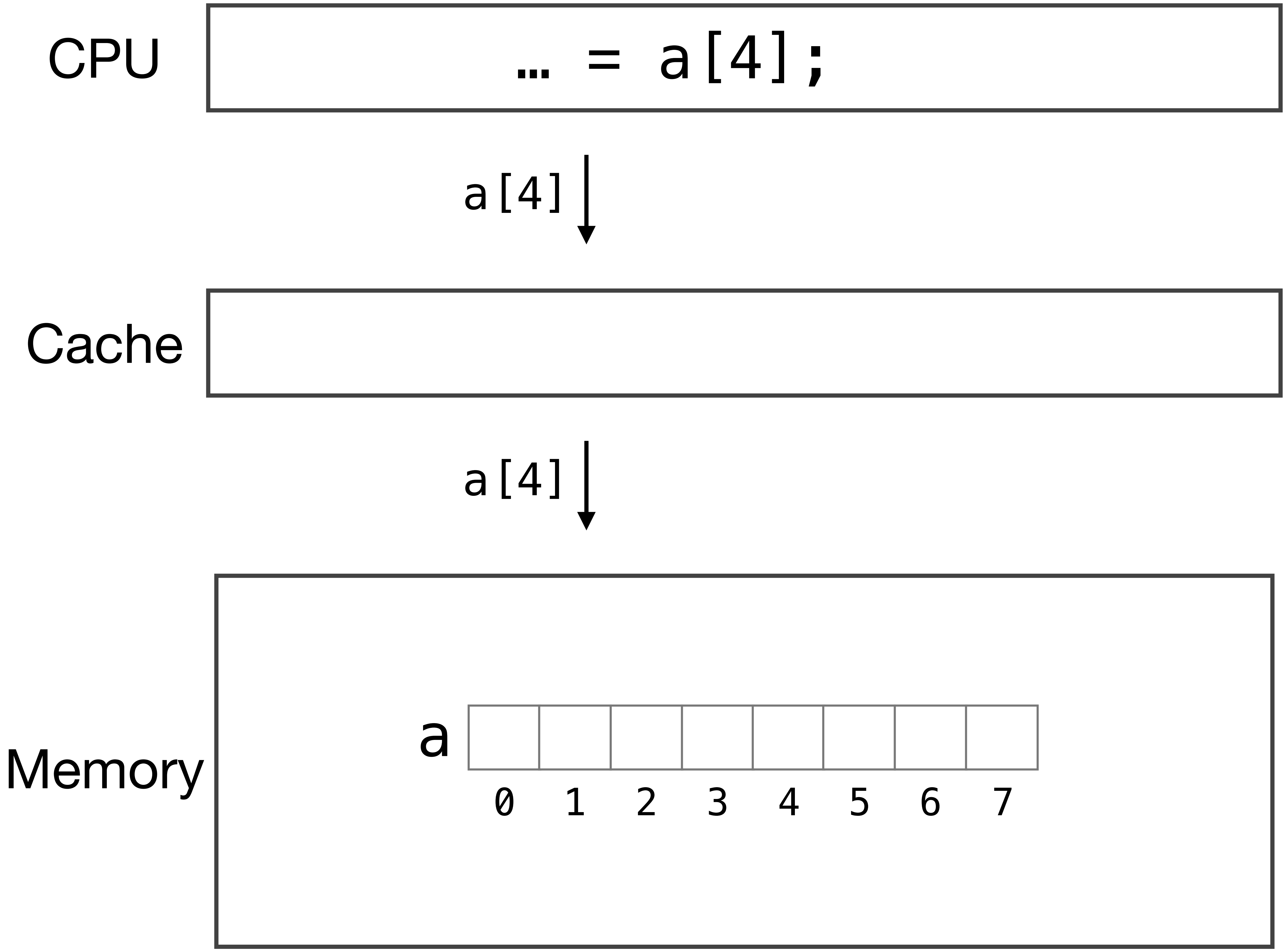
Spatial locality



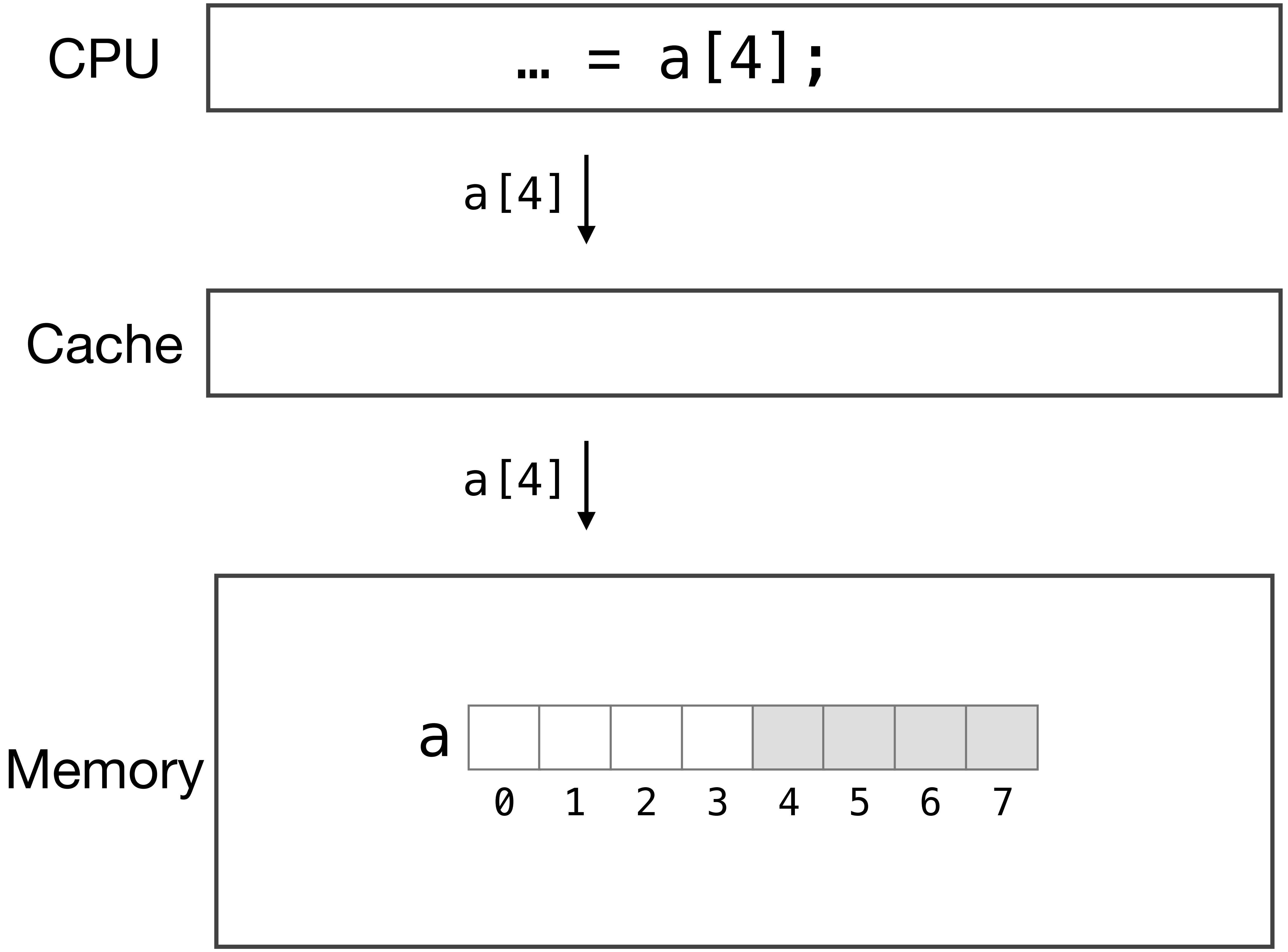
Spatial locality



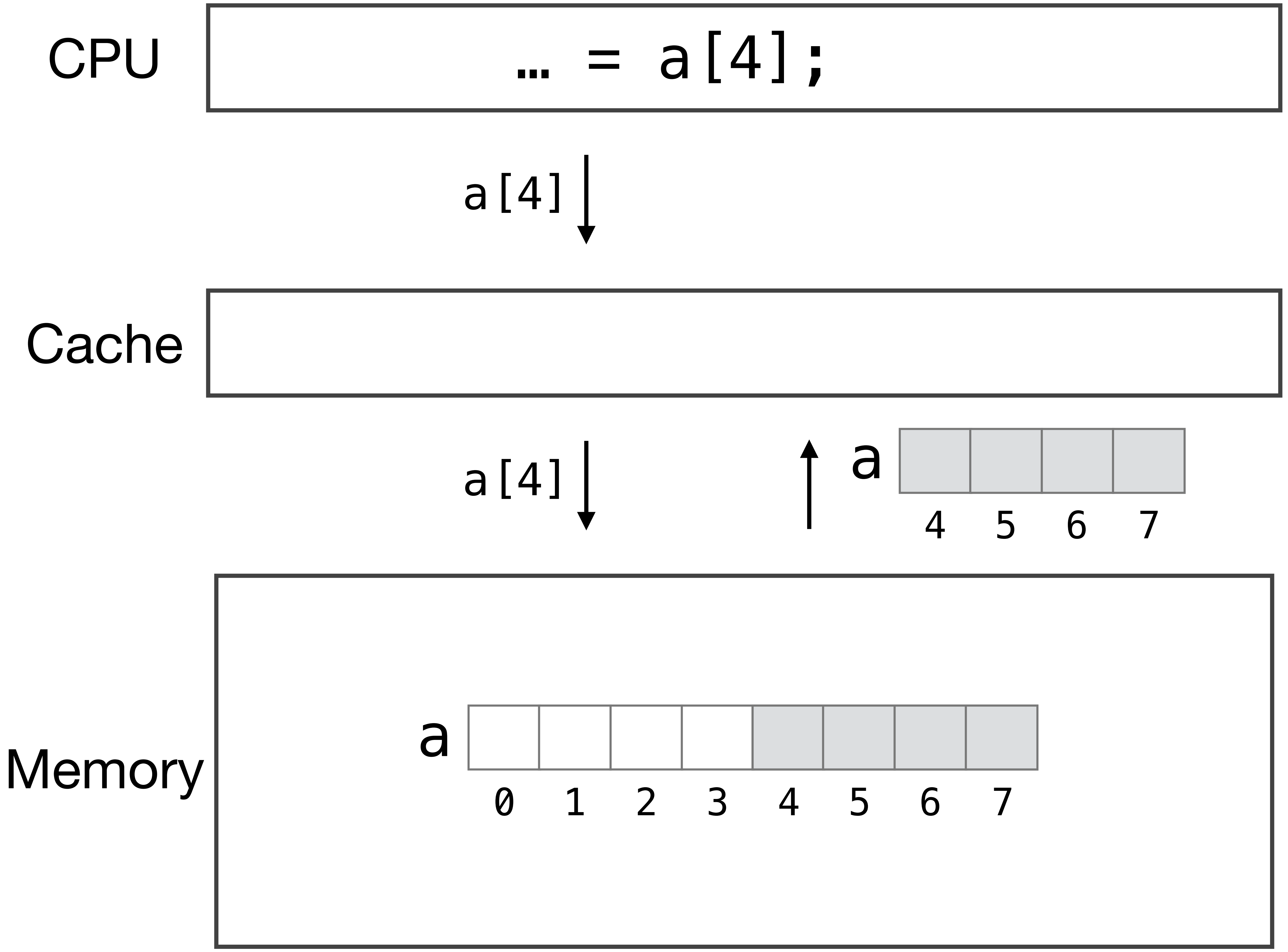
Spatial locality



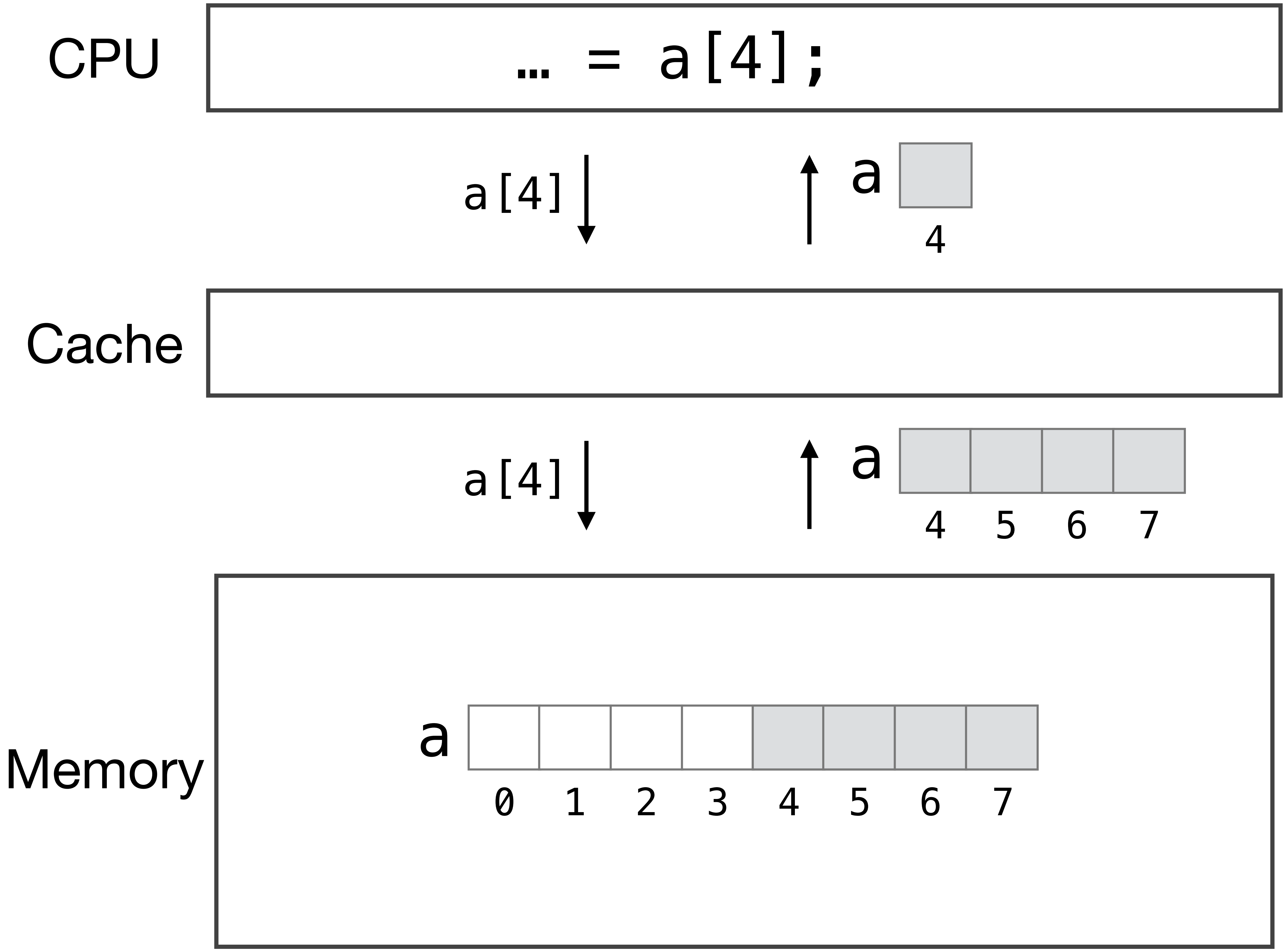
Spatial locality



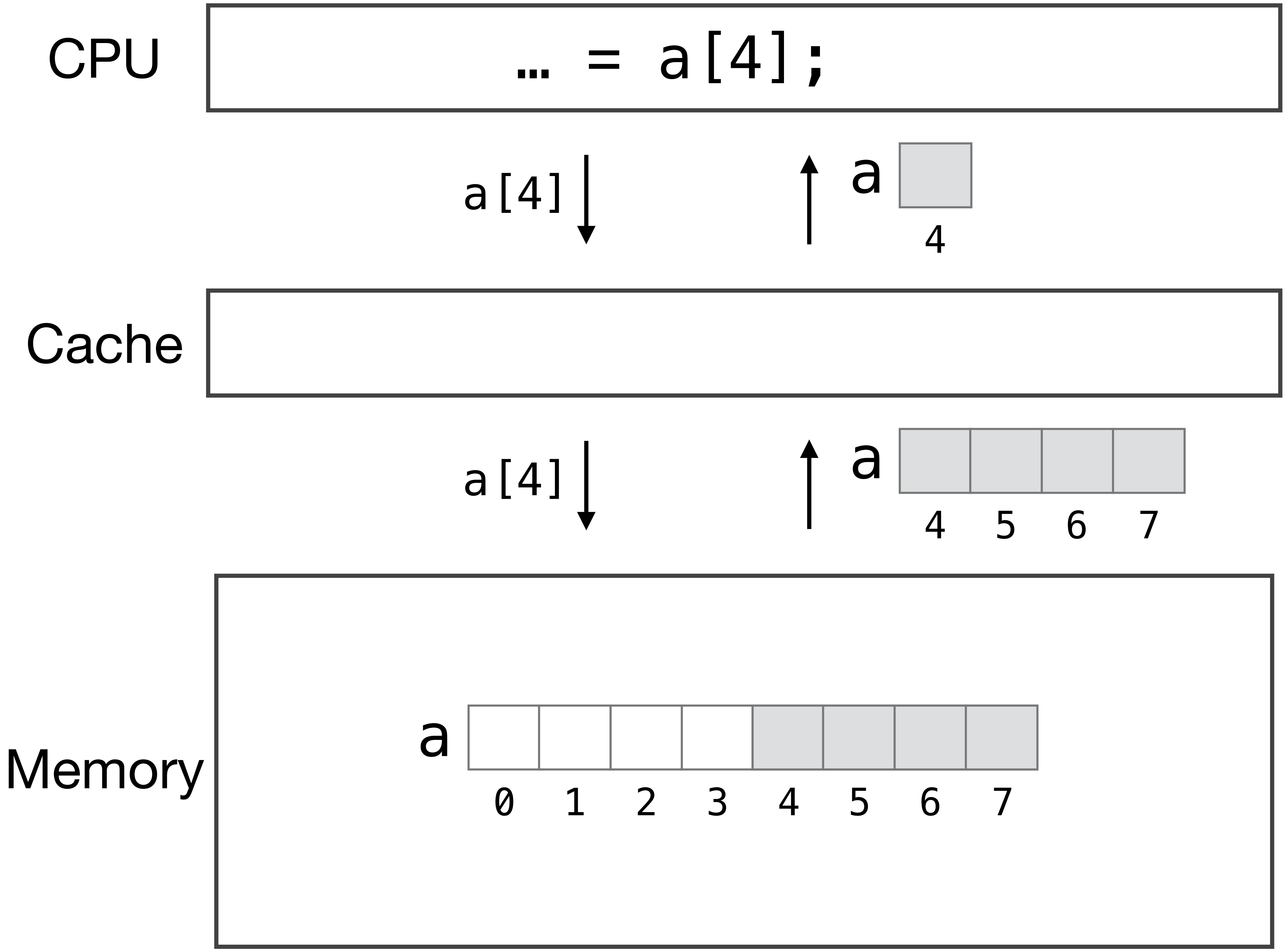
Spatial locality

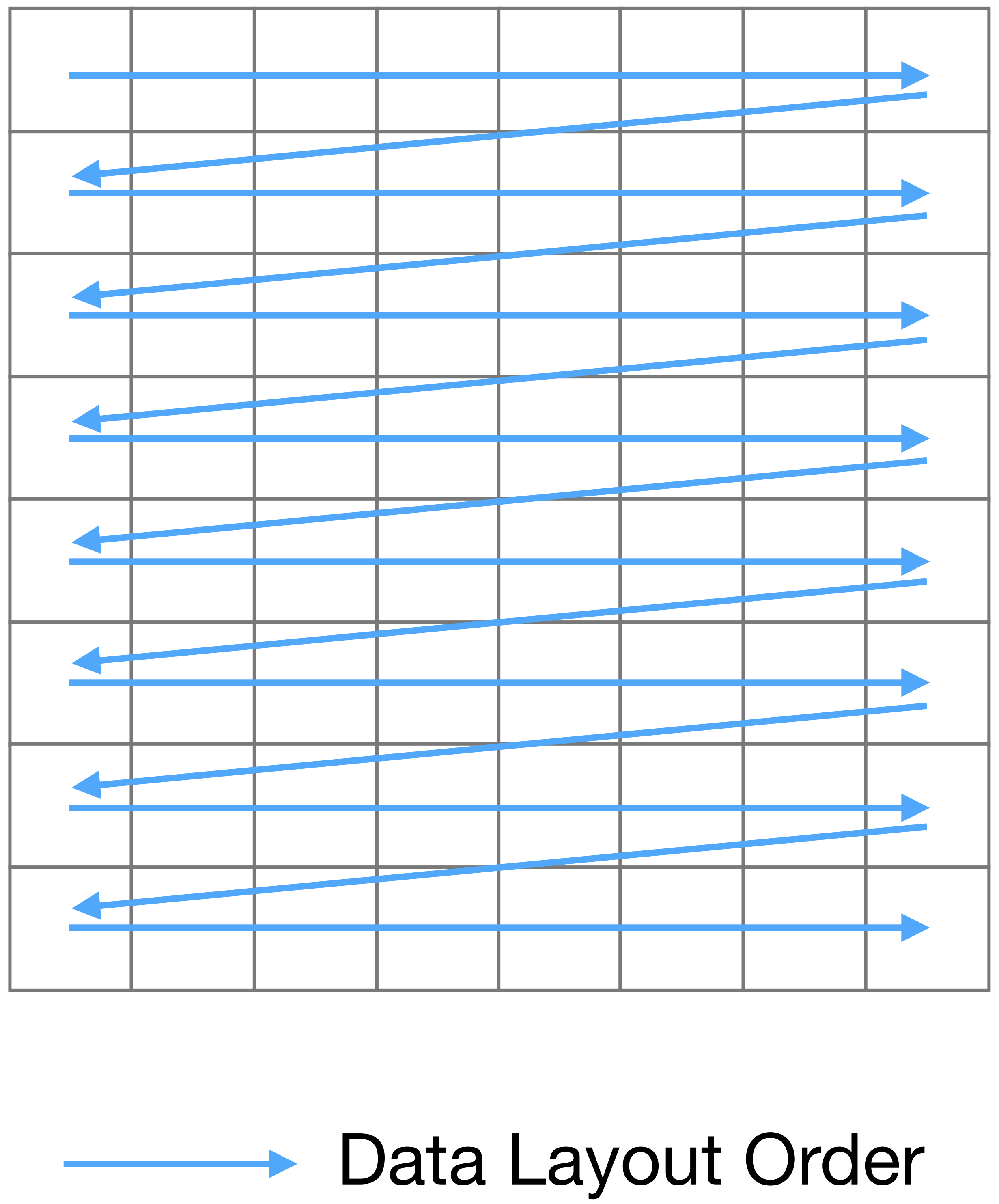


Spatial locality

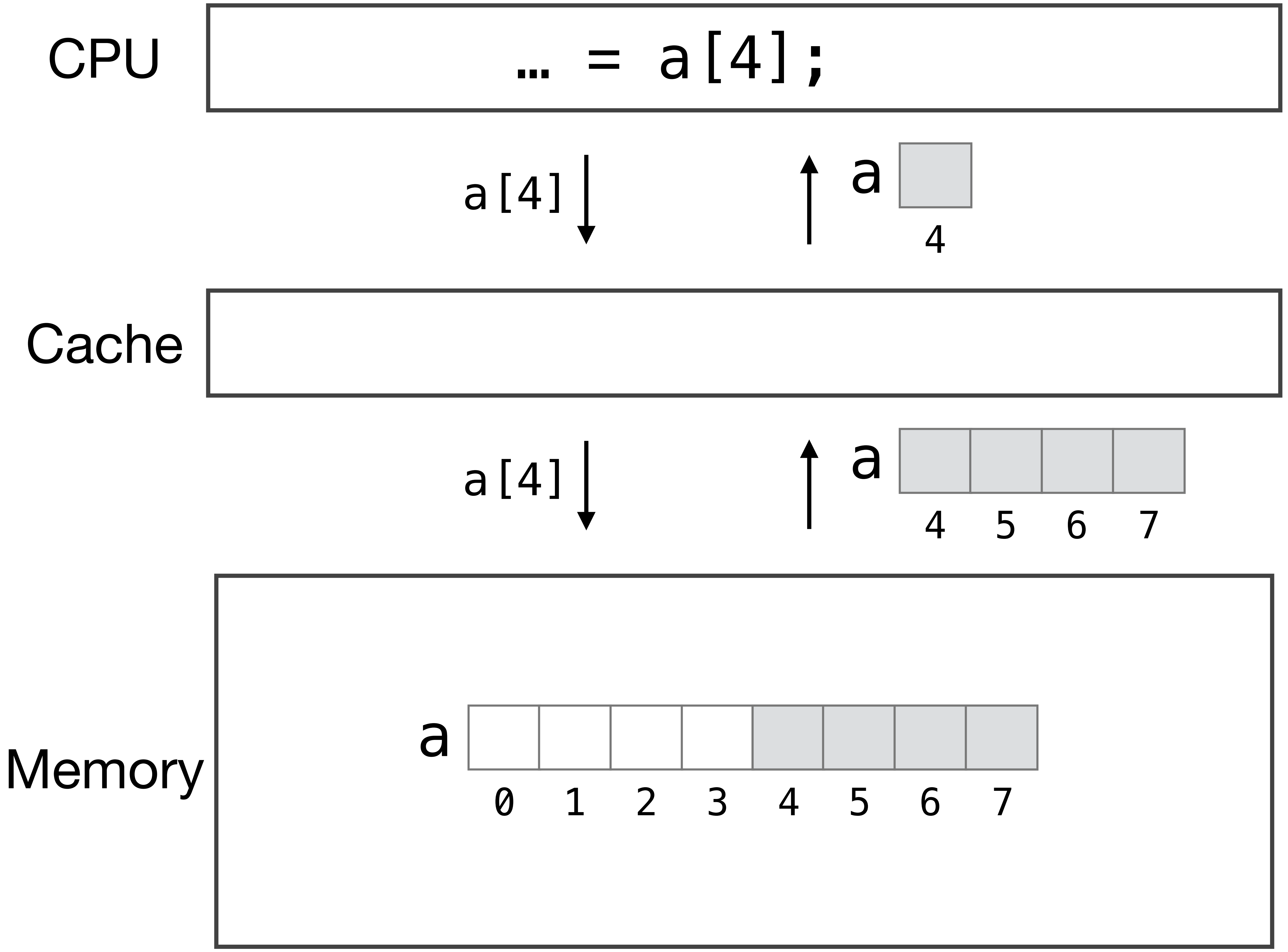


Spatial locality



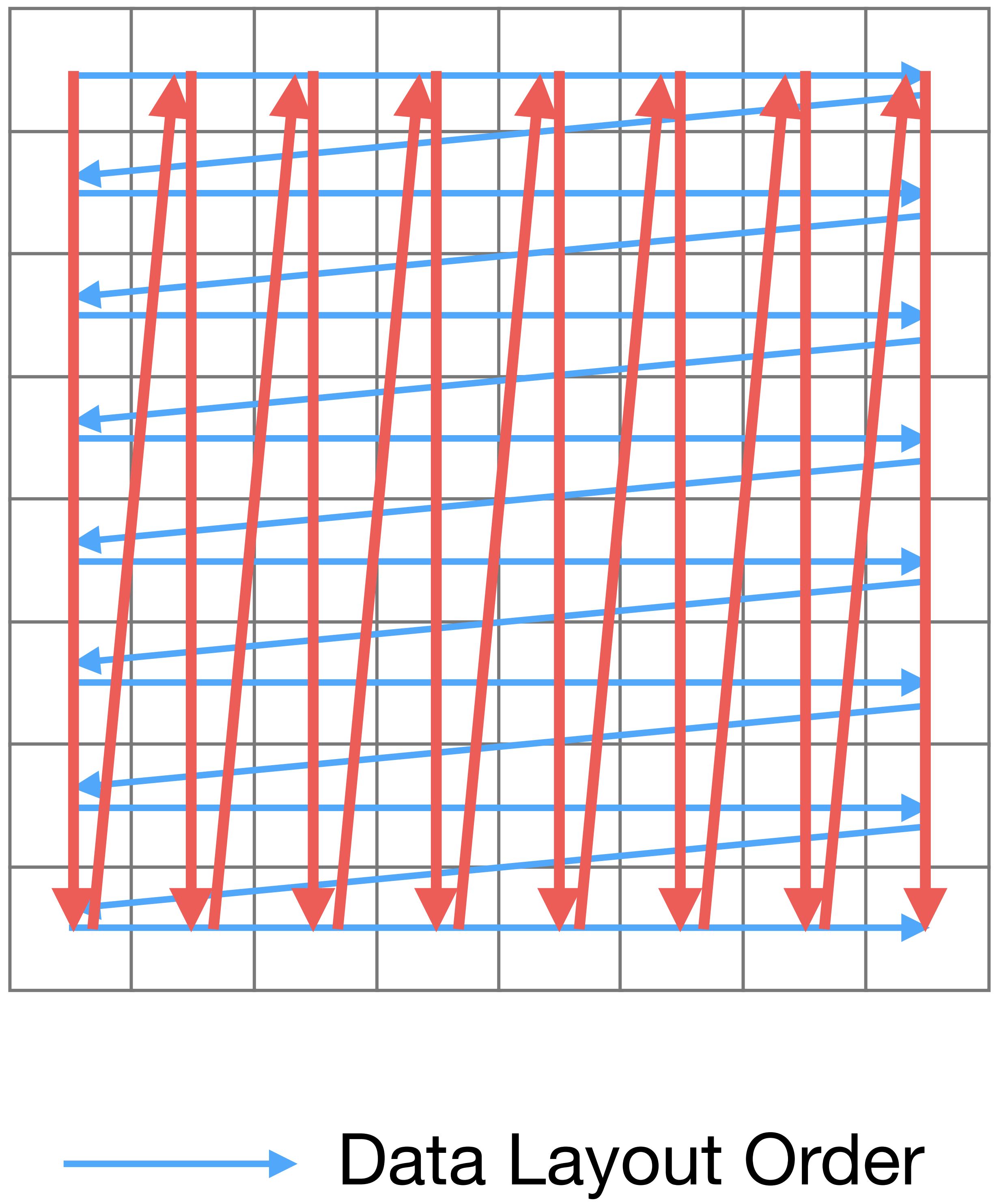


Spatial locality



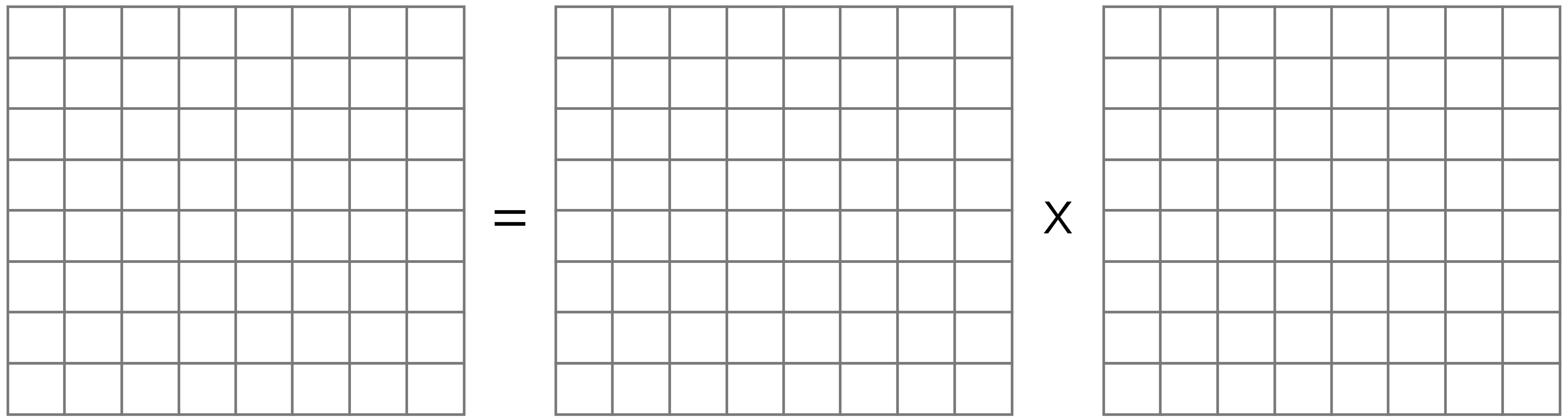
Avoid jumping around the address space

by not iterating along the data layout



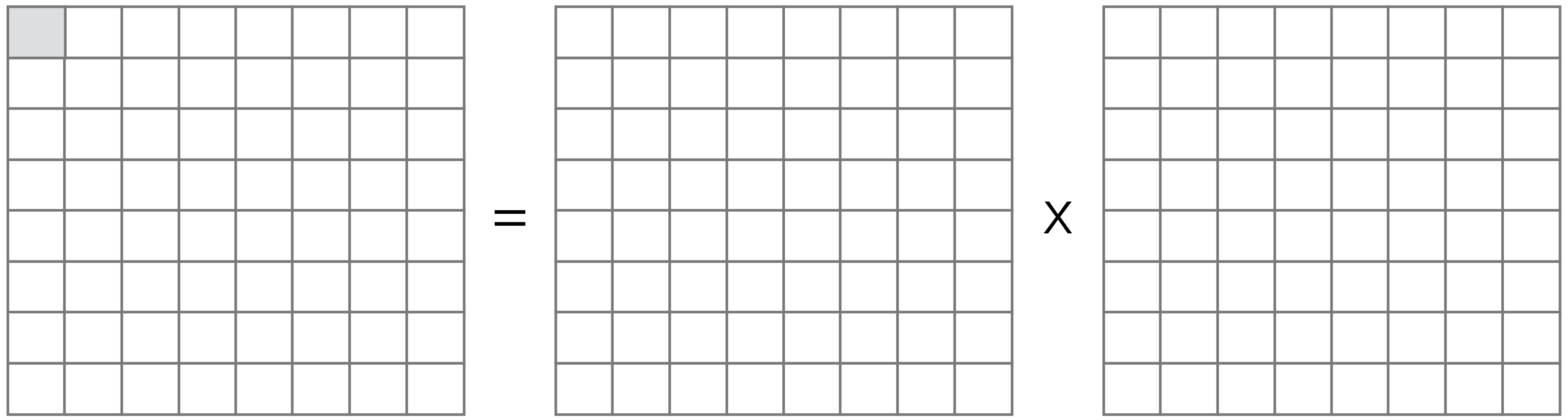
Temporal locality in matrix-matrix multiplication

$$A_{ij} = B_{ik}C_{kj}$$



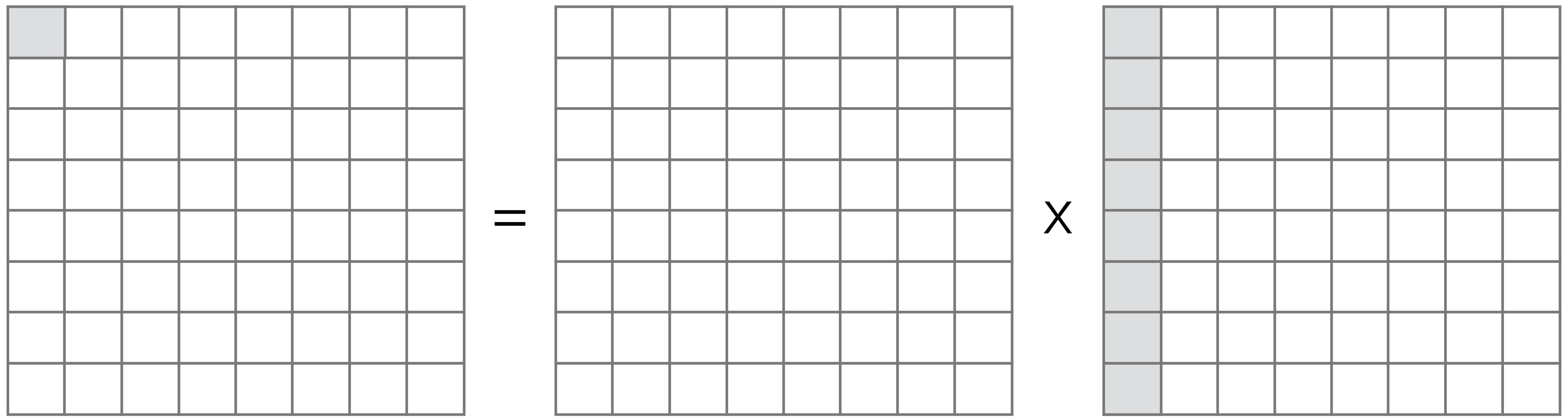
Temporal locality in matrix-matrix multiplication

$$A_{ij} = B_{ik}C_{kj}$$



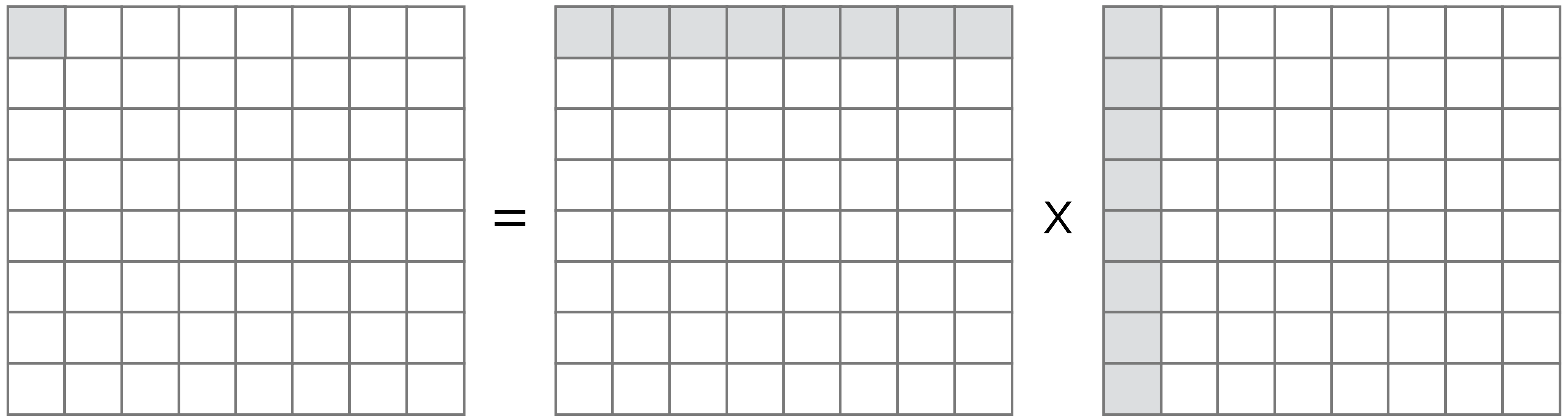
Temporal locality in matrix-matrix multiplication

$$A_{ij} = B_{ik}C_{kj}$$



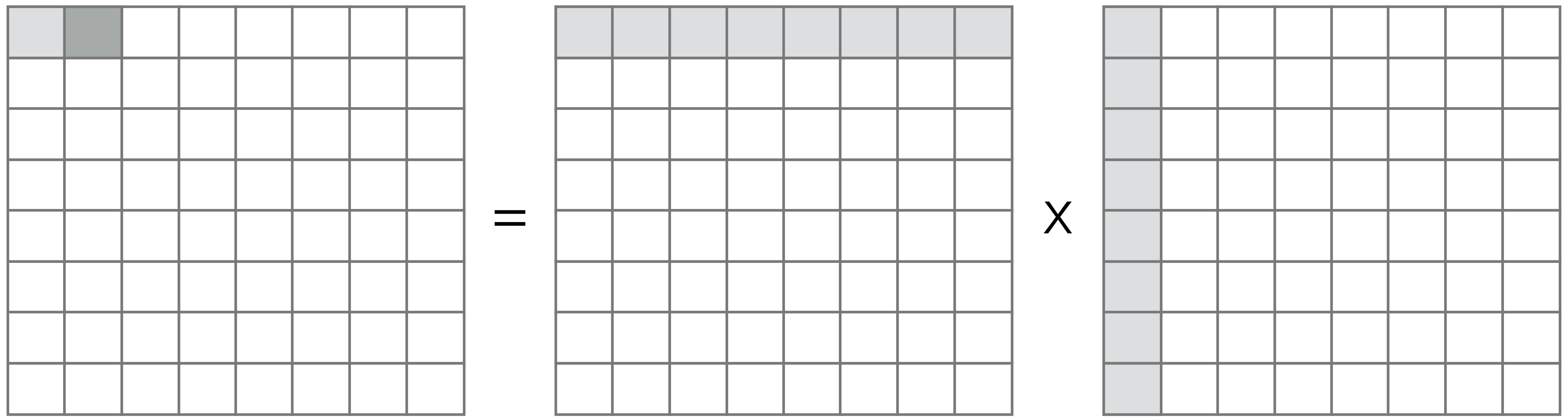
Temporal locality in matrix-matrix multiplication

$$A_{ij} = B_{ik}C_{kj}$$



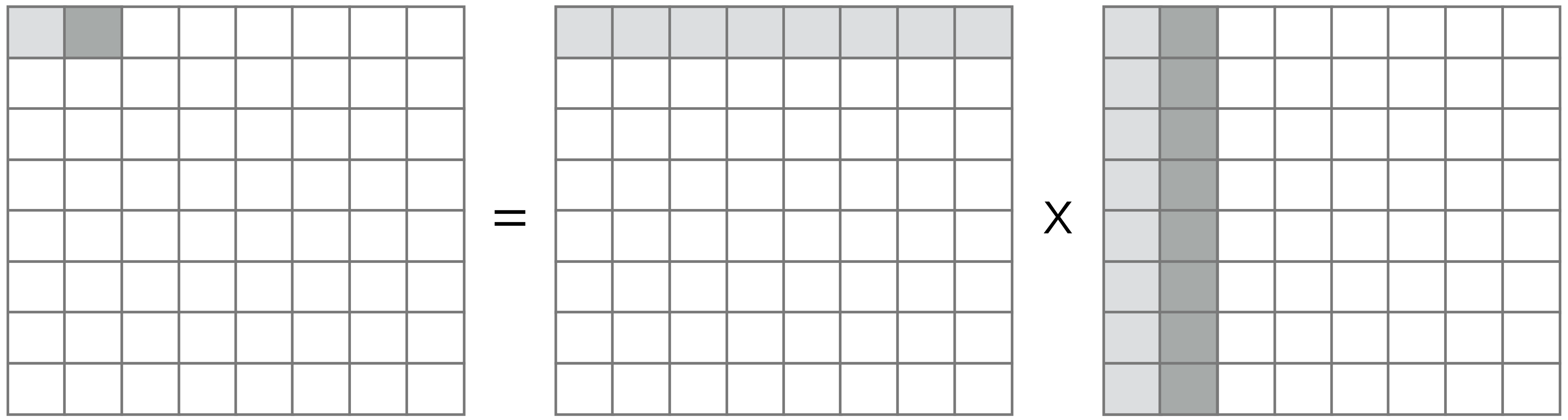
Temporal locality in matrix-matrix multiplication

$$A_{ij} = B_{ik}C_{kj}$$



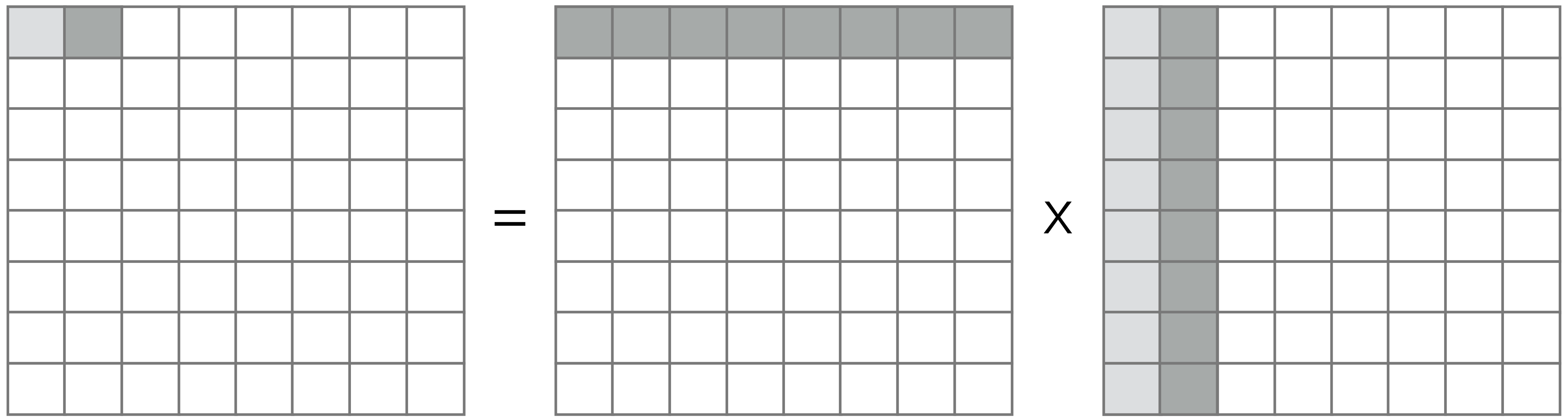
Temporal locality in matrix-matrix multiplication

$$A_{ij} = B_{ik}C_{kj}$$



Temporal locality in matrix-matrix multiplication

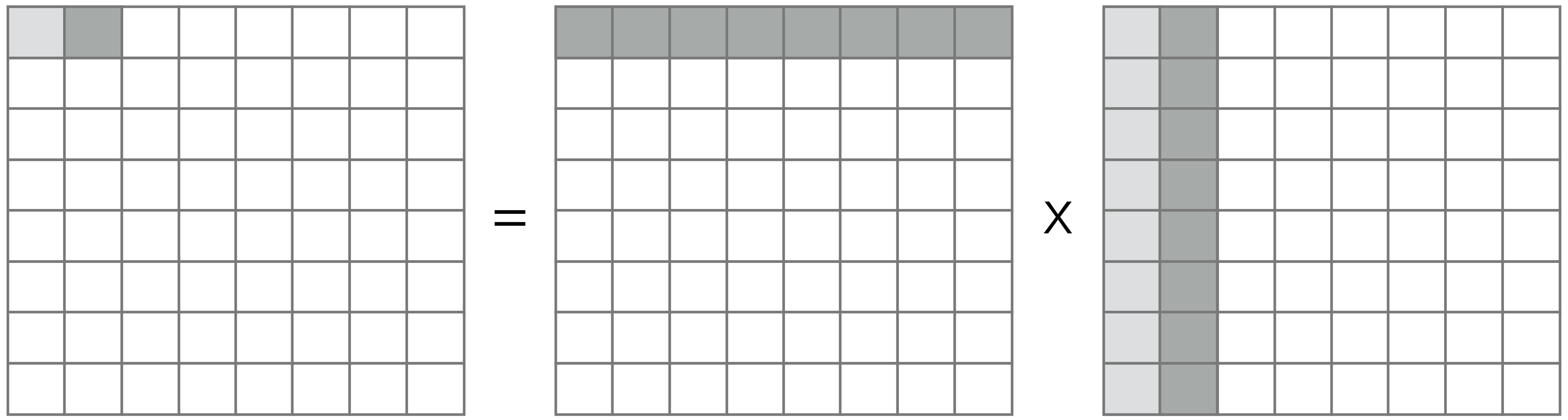
$$A_{ij} = B_{ik}C_{kj}$$



Temporal locality in matrix-matrix multiplication

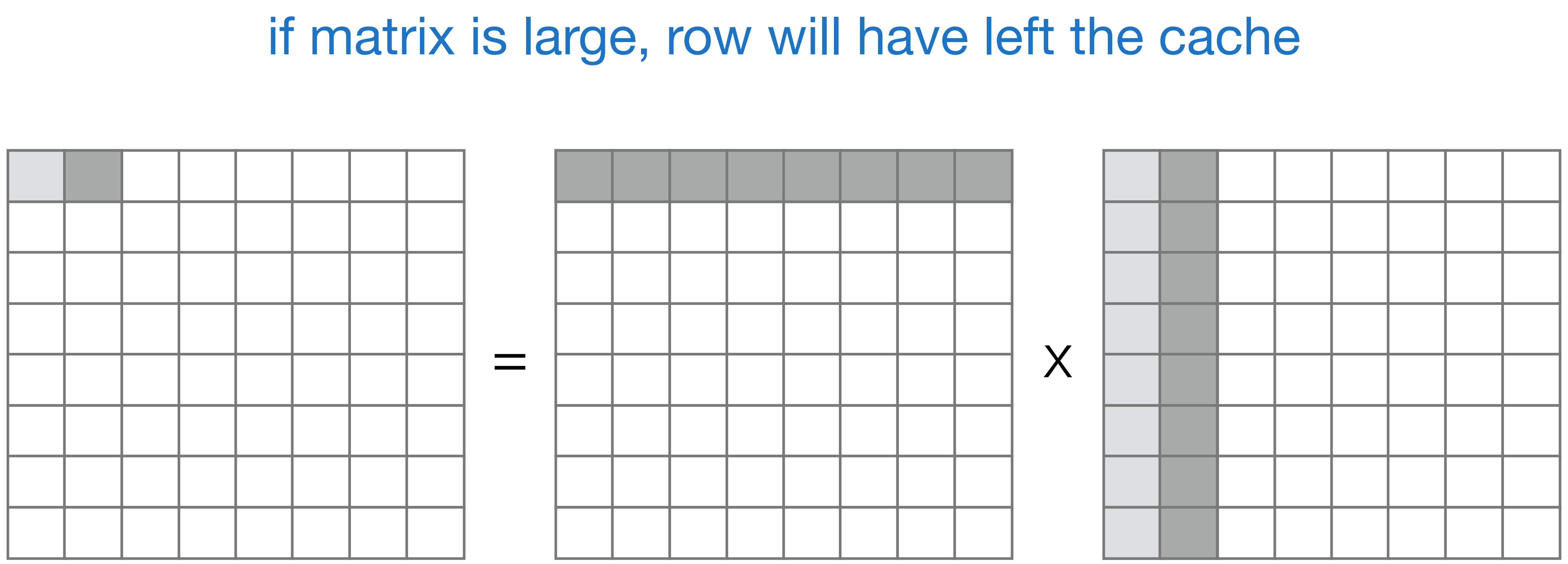
$$A_{ij} = B_{ik}C_{kj}$$

if matrix is large, row will have left the cache

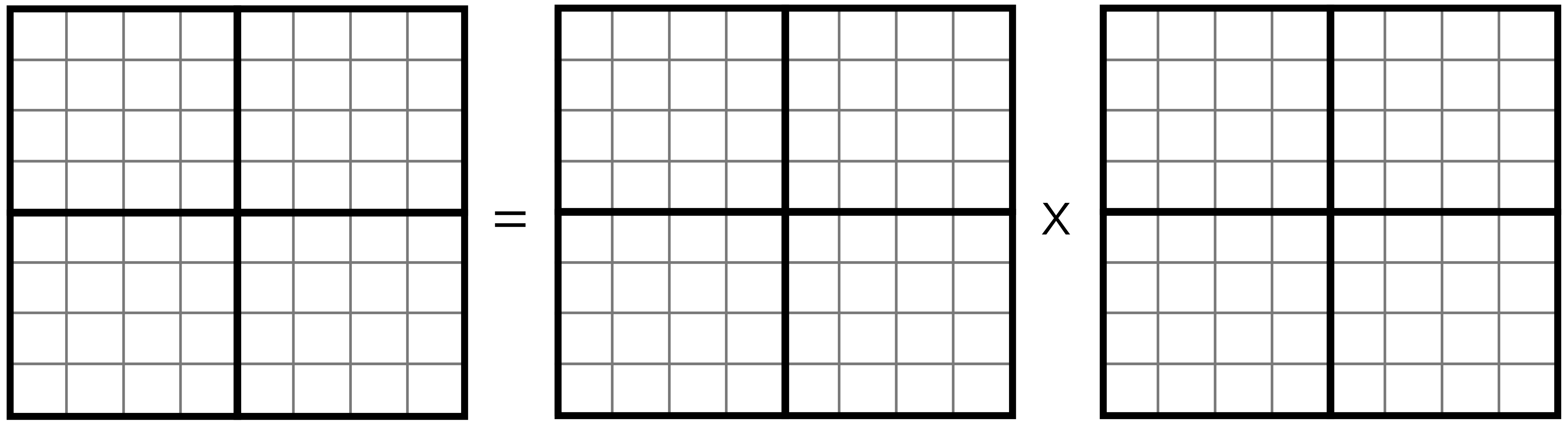


Temporal locality in matrix-matrix multiplication

$$A_{ij} = B_{ik}C_{kj}$$



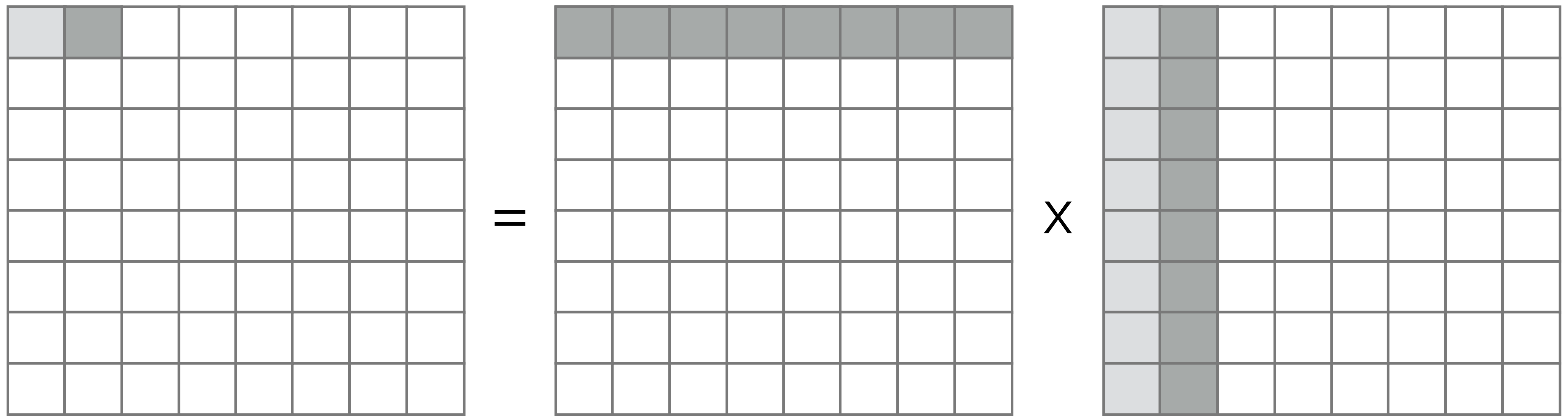
2x2 matrix multiply,
where the operations are
4x4 matrix multiplies



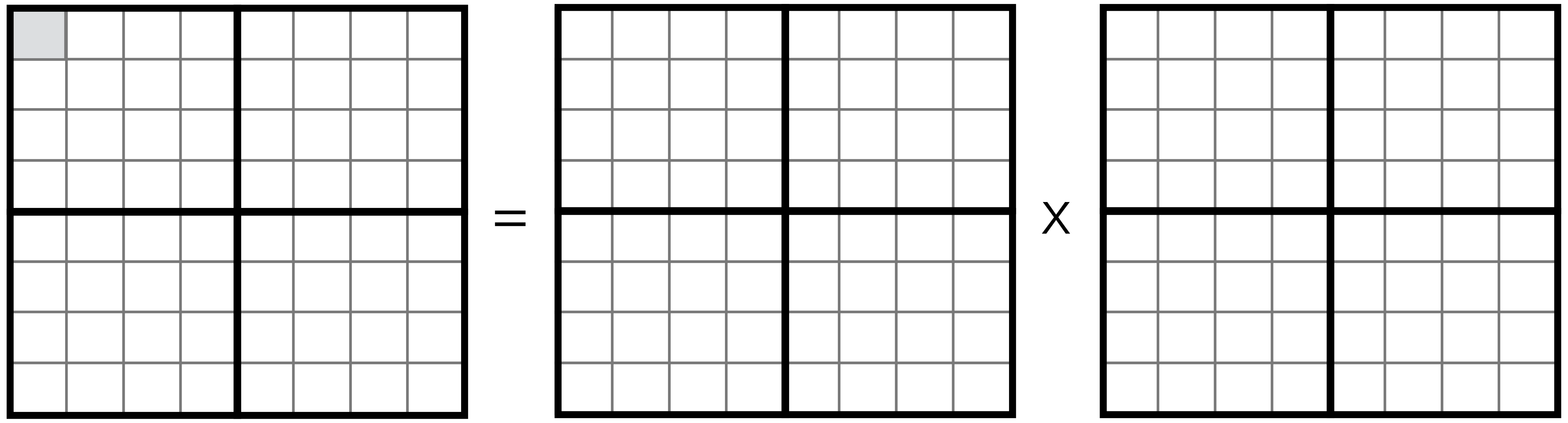
Temporal locality in matrix-matrix multiplication

if matrix is large, row will have left the cache

$$A_{ij} = B_{ik}C_{kj}$$



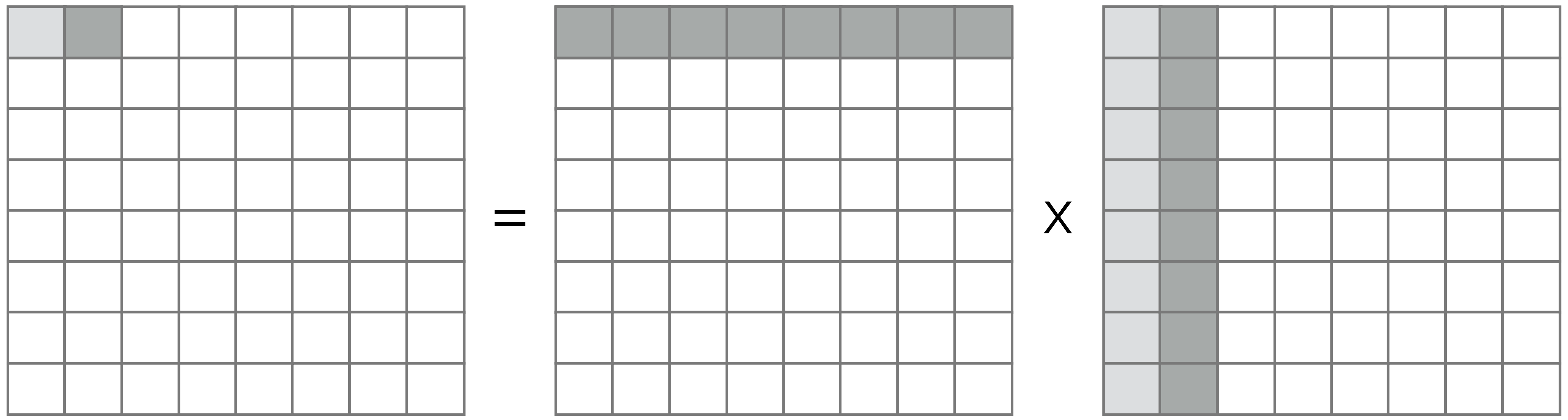
2x2 matrix multiply,
where the operations are
4x4 matrix multiplies



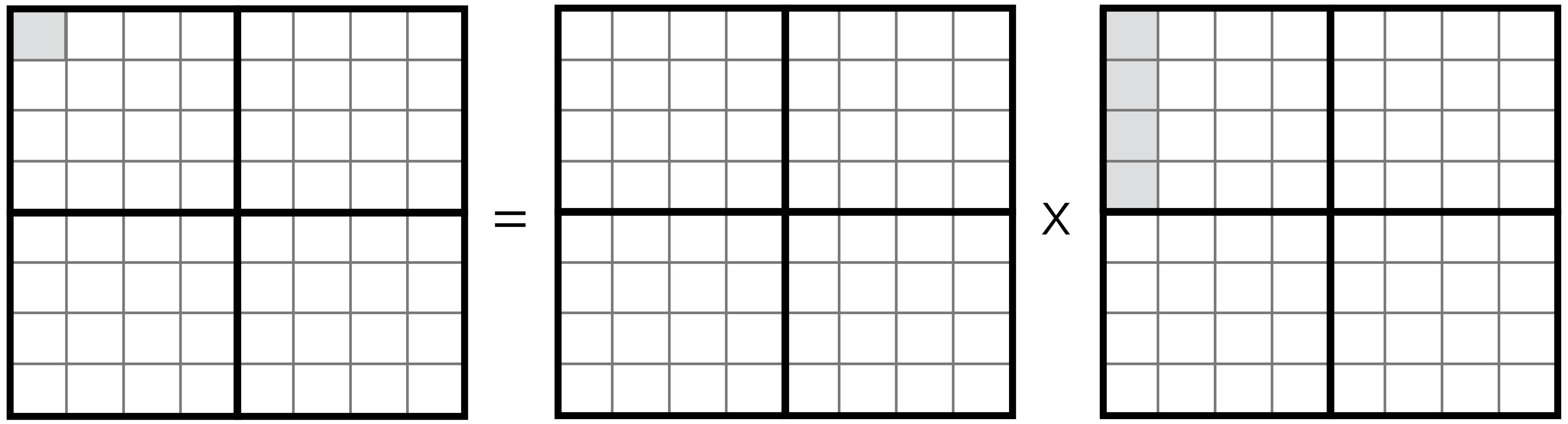
Temporal locality in matrix-matrix multiplication

if matrix is large, row will have left the cache

$$A_{ij} = B_{ik} C_{kj}$$



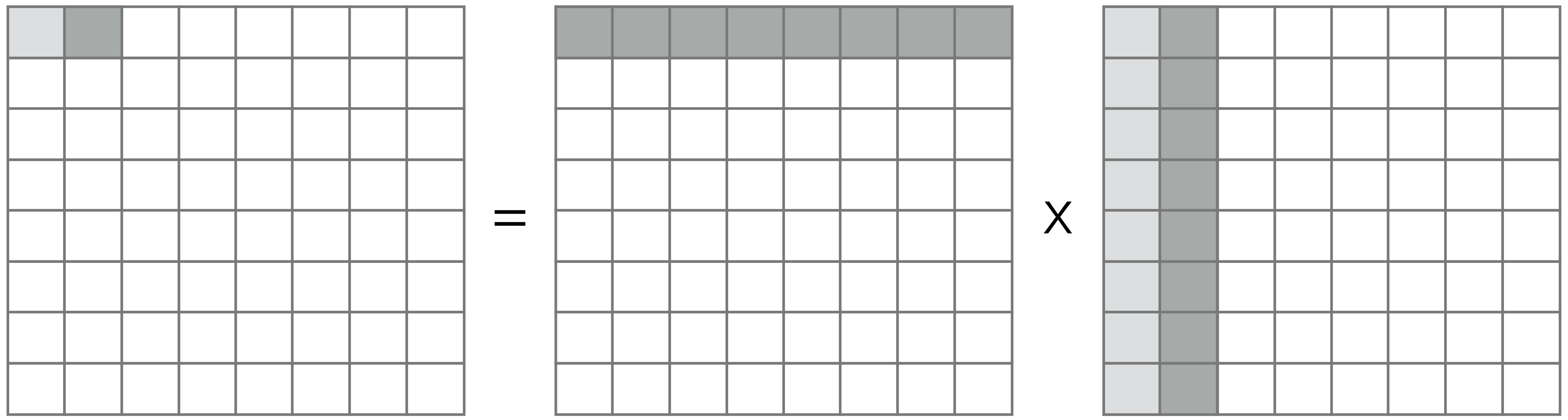
2x2 matrix multiply,
where the operations are
4x4 matrix multiplies



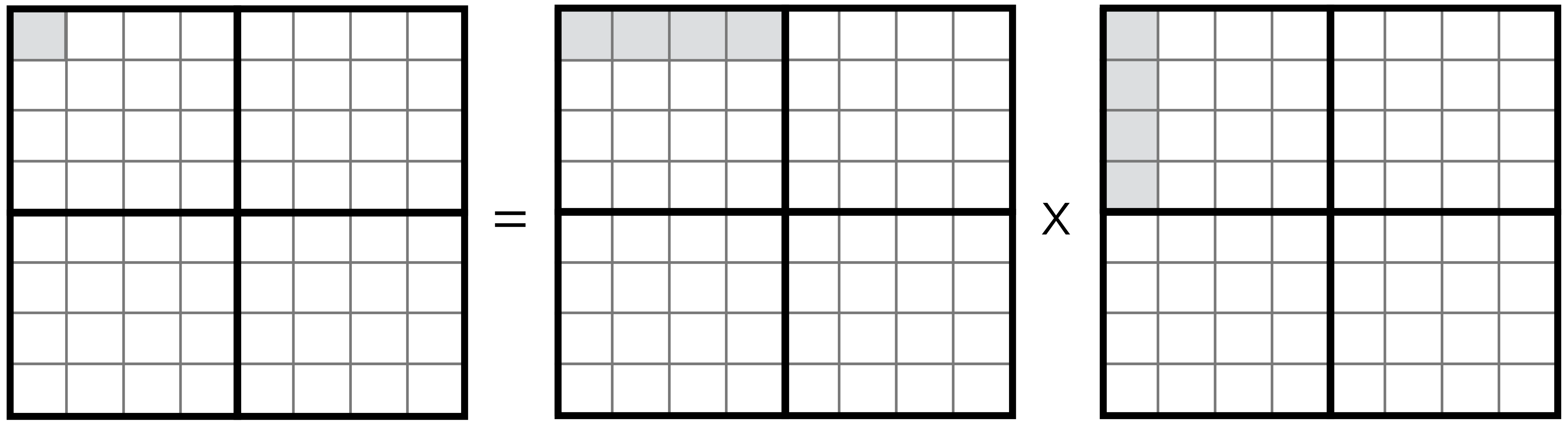
Temporal locality in matrix-matrix multiplication

if matrix is large, row will have left the cache

$$A_{ij} = B_{ik}C_{kj}$$



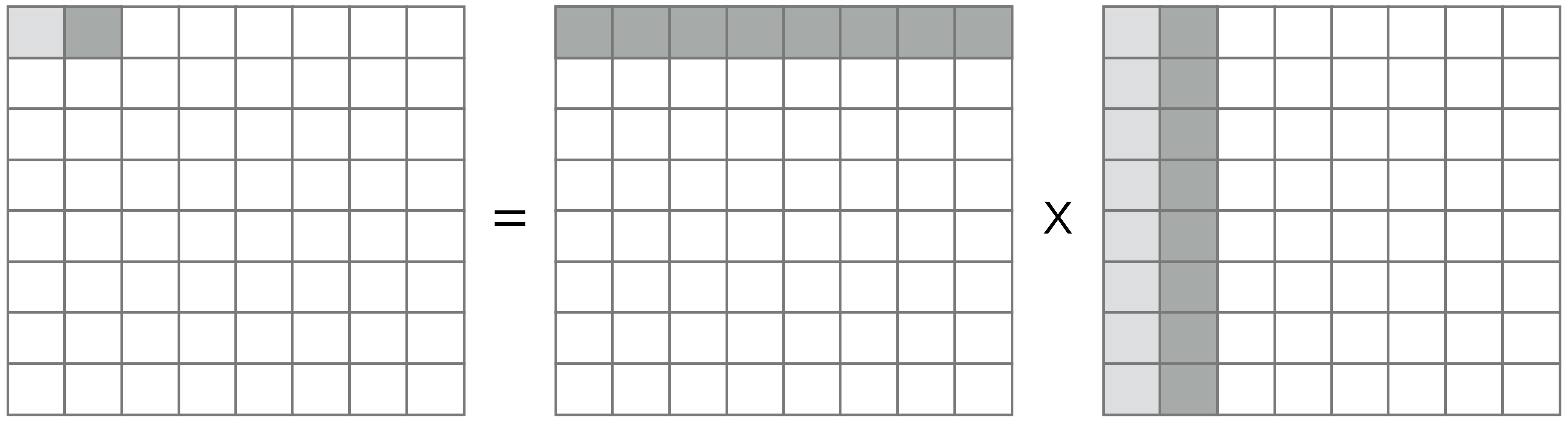
2x2 matrix multiply,
where the operations are
4x4 matrix multiplies



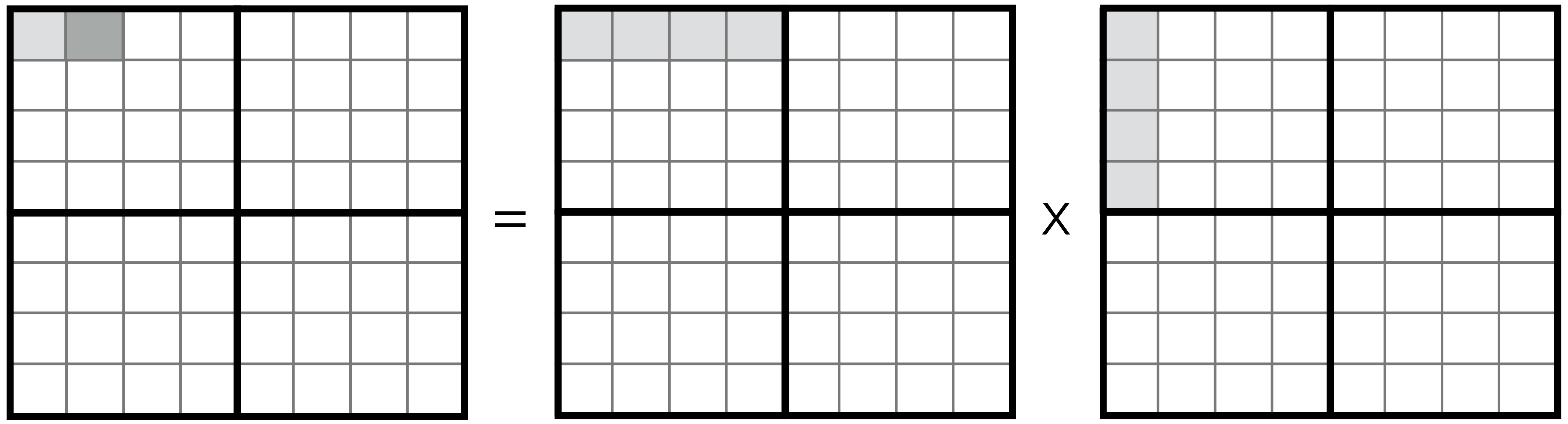
Temporal locality in matrix-matrix multiplication

if matrix is large, row will have left the cache

$$A_{ij} = B_{ik}C_{kj}$$

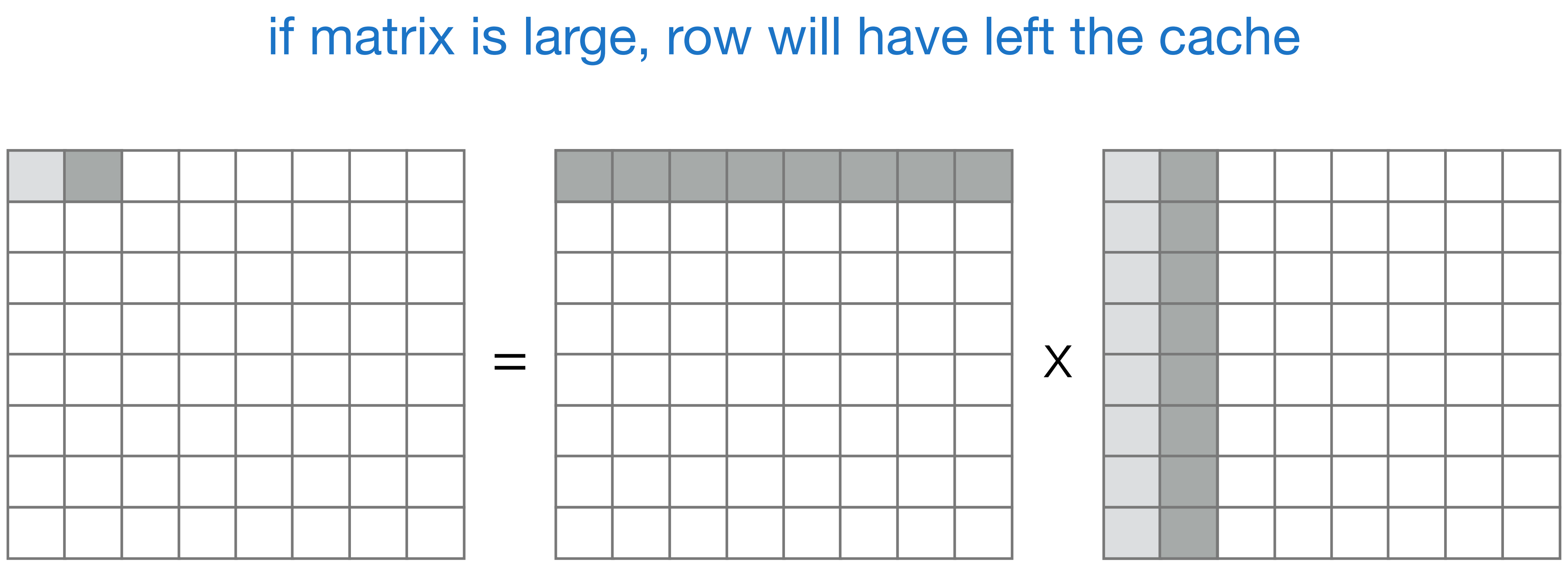


2x2 matrix multiply,
where the operations are
4x4 matrix multiplies

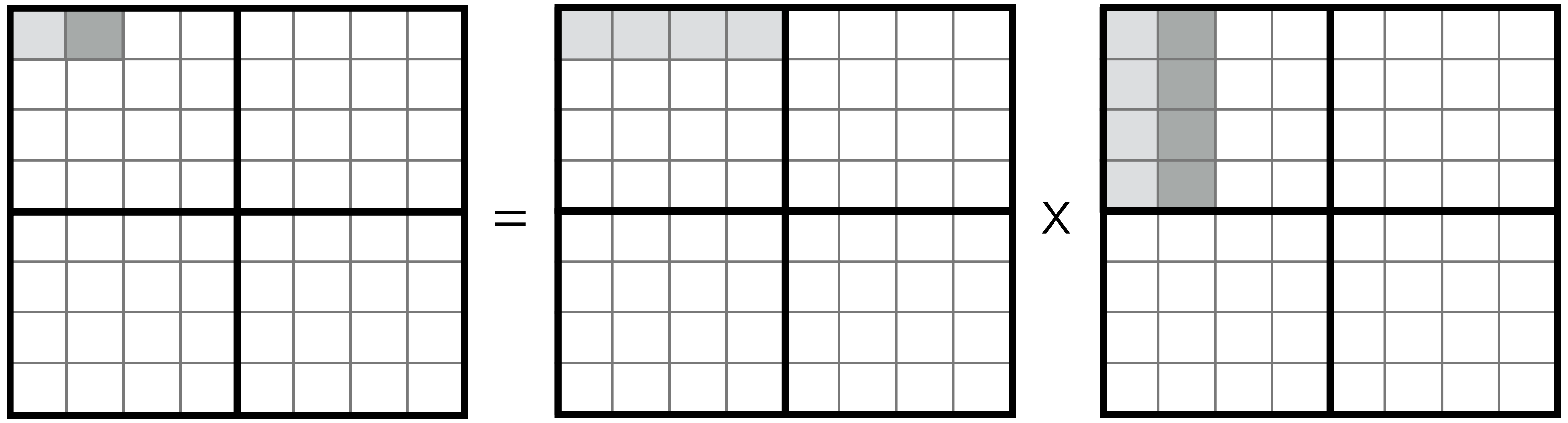


Temporal locality in matrix-matrix multiplication

$A_{ij} = B_{ik}C_{kj}$



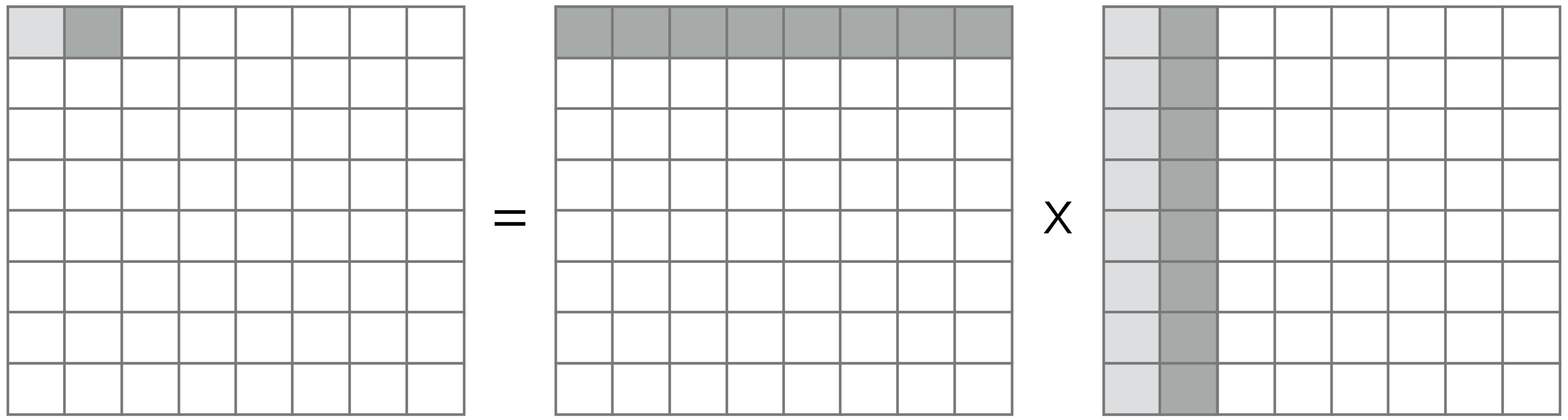
2x2 matrix multiply,
where the operations are
4x4 matrix multiplies



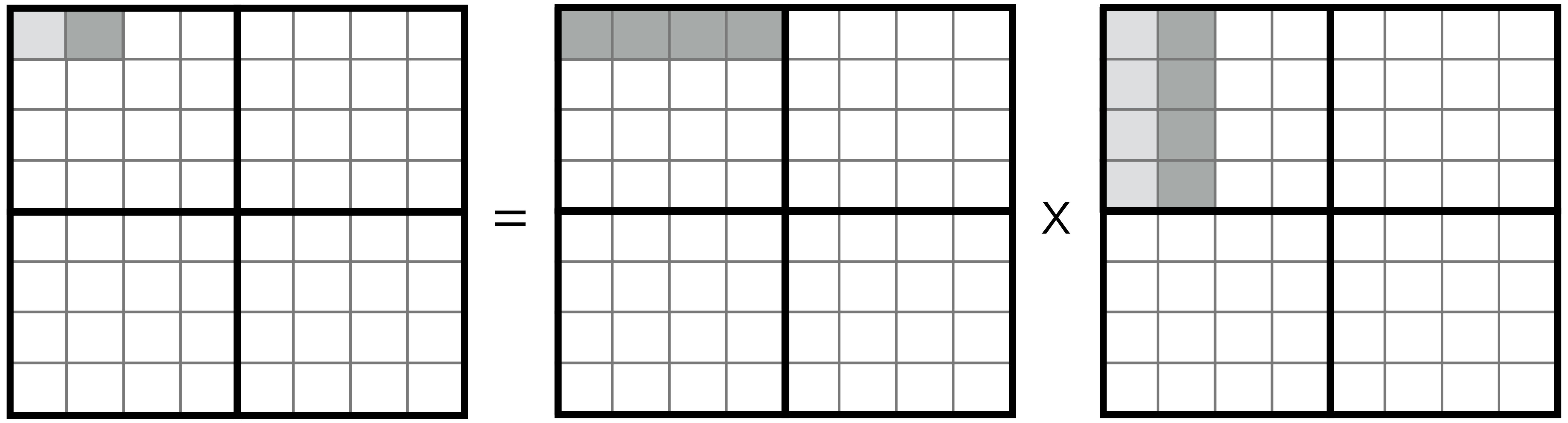
Temporal locality in matrix-matrix multiplication

if matrix is large, row will have left the cache

$$A_{ij} = B_{ik}C_{kj}$$



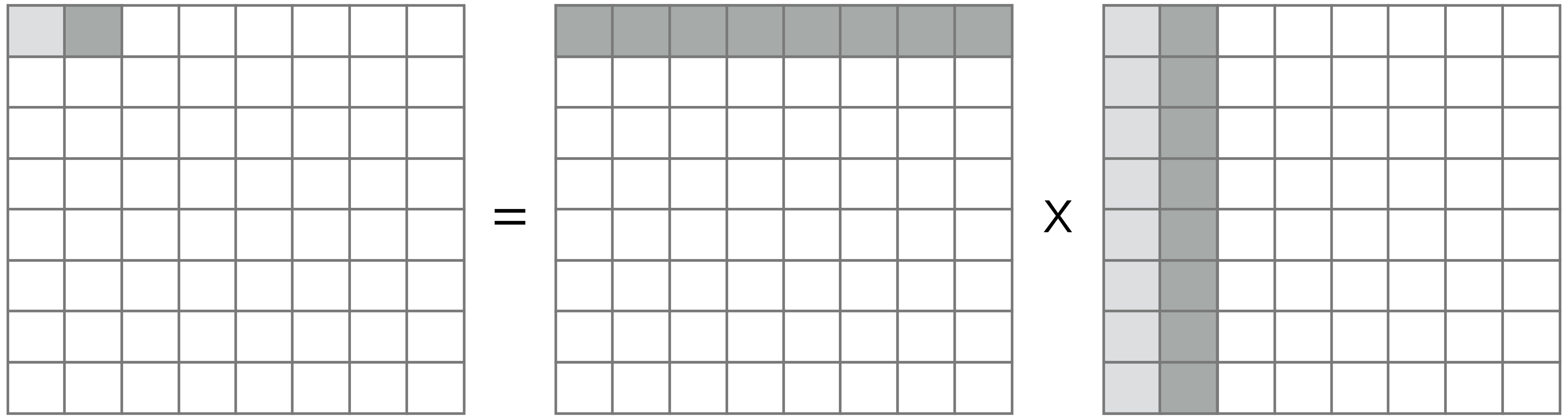
2x2 matrix multiply,
where the operations are
4x4 matrix multiplies



Temporal locality in matrix-matrix multiplication

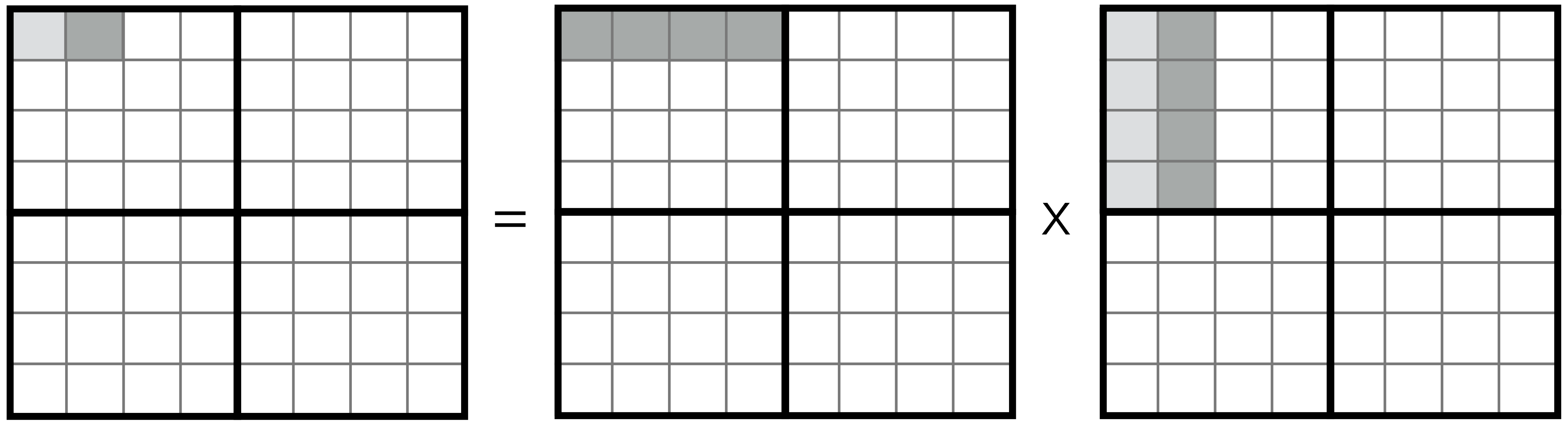
$$A_{ij} = B_{ik}C_{kj}$$

if matrix is large, row will have left the cache



shorter reuse distance

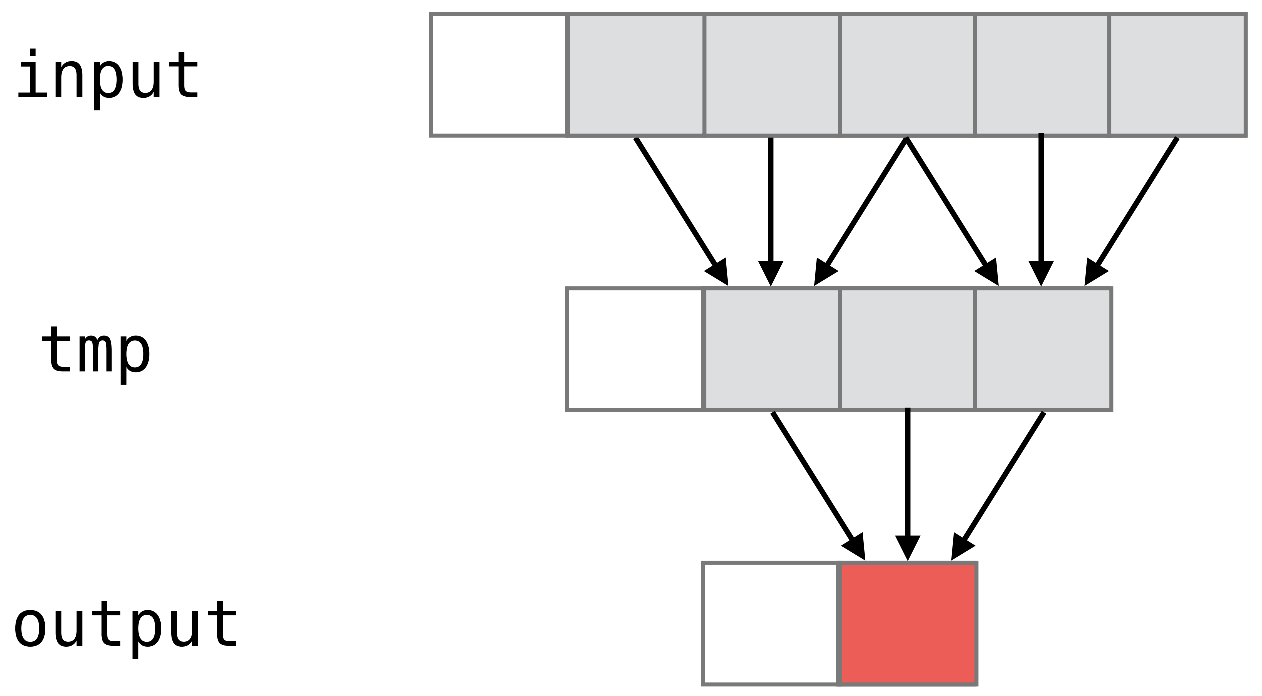
2x2 matrix multiply,
where the operations are
4x4 matrix multiplies



Buying locality with redundant work in fused stencils

Stencil loops

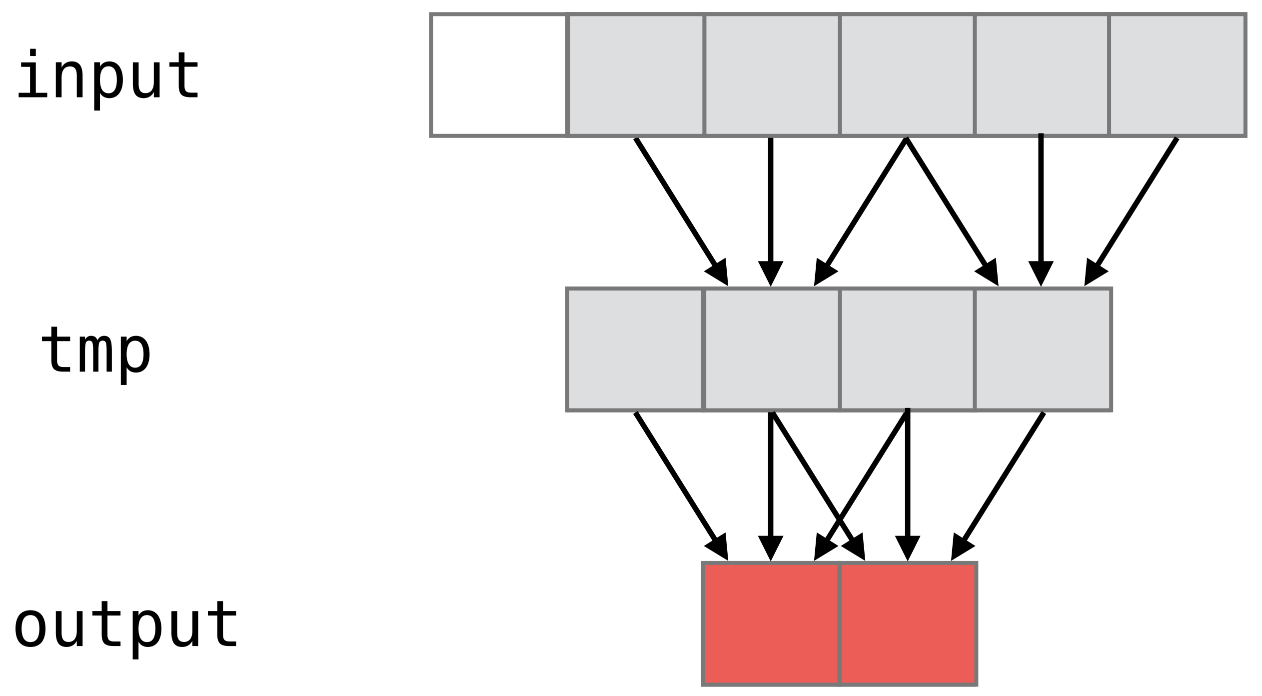
```
for (int j=0; j<4; i++)
  tmp[j] = (input[j-1] + input[j] + input[j+1]) / 3;

for (int i=1; i<3; i++)
  output[i] = (tmp[i-1] + tmp[i] + tmp[i+1]) / 3;
```



Buying locality with redundant work in fused stencils

Stencil loops

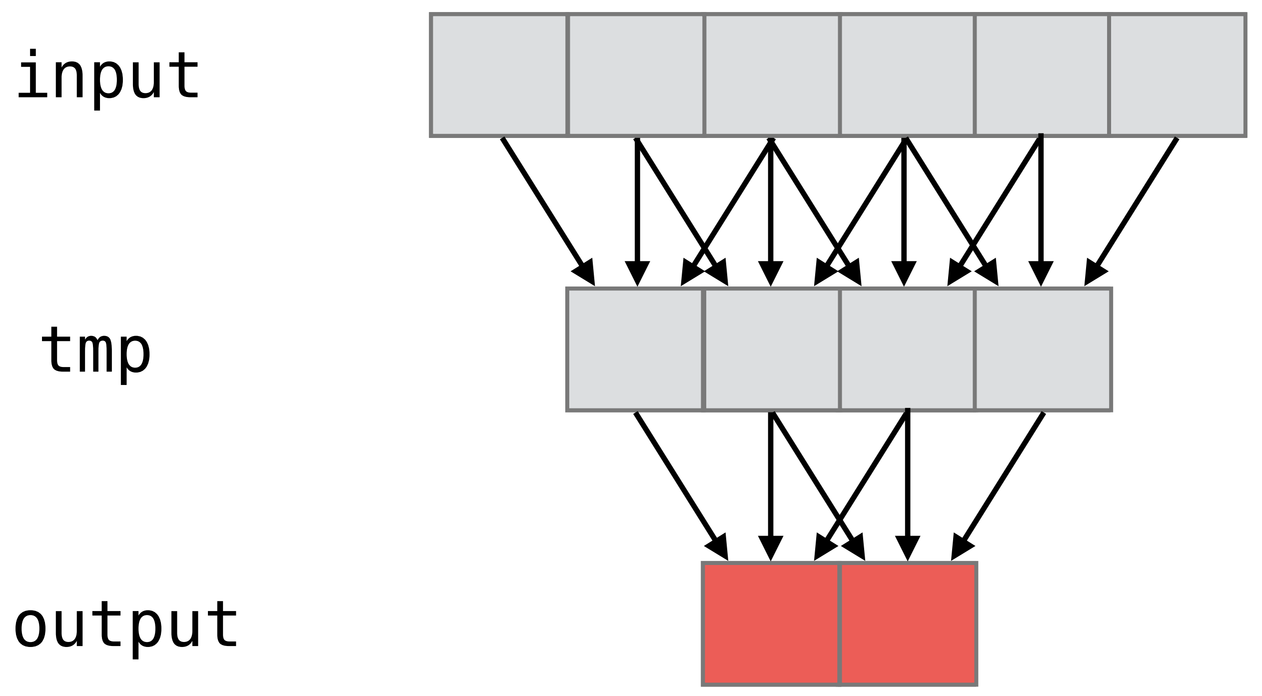
```
for (int j=0; j<4; i++)
  tmp[j] = (input[j-1] + input[j] + input[j+1]) / 3;

for (int i=1; i<3; i++)
  output[i] = (tmp[i-1] + tmp[i] + tmp[i+1]) / 3;
```



Buying locality with redundant work in fused stencils

Stencil loops

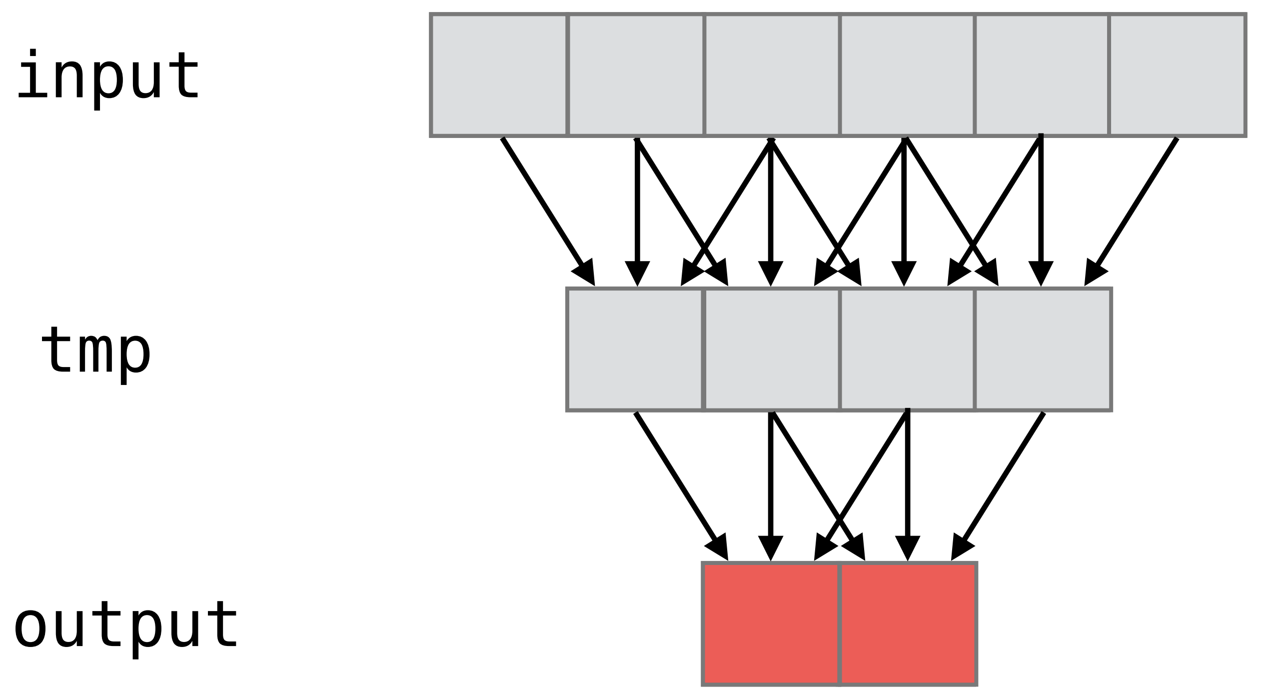
```
for (int j=0; j<4; i++)
  tmp[j] = (input[j-1] + input[j] + input[j+1]) / 3;

for (int i=1; i<3; i++)
  output[i] = (tmp[i-1] + tmp[i] + tmp[i+1]) / 3;
```



Buying locality with redundant work in fused stencils

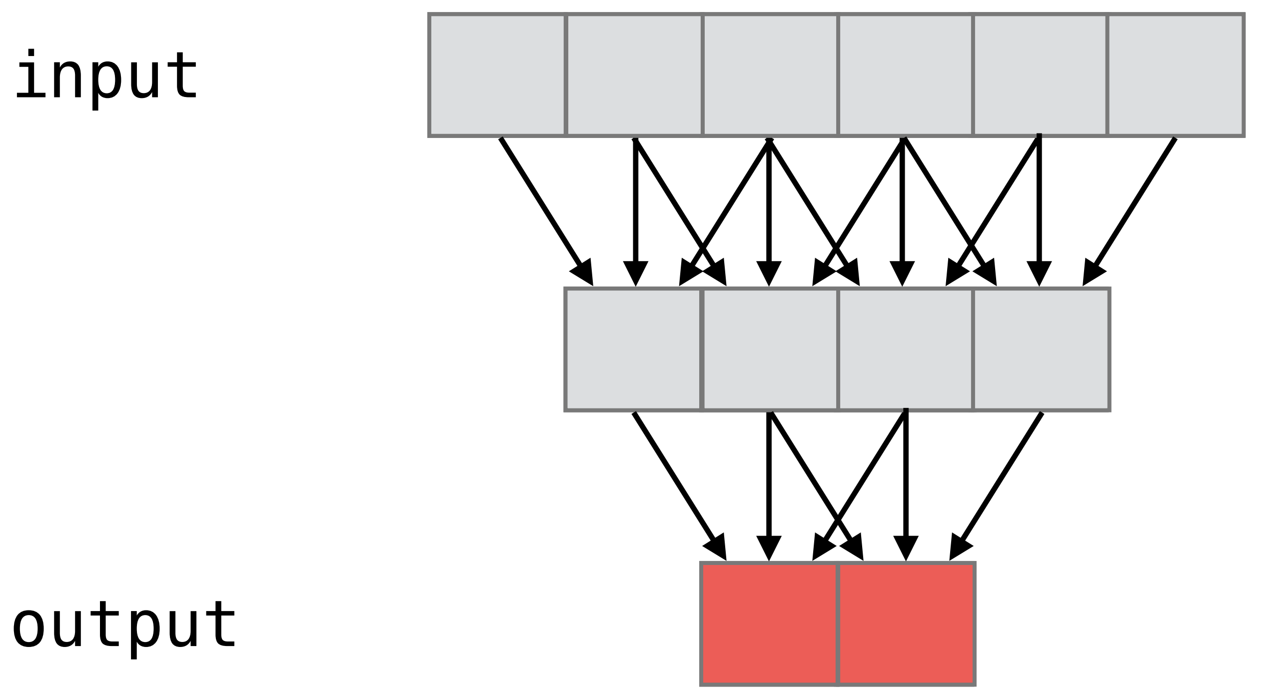
Stencil loops

```
for (int j=0; j<4; i++)
  tmp[j] = (input[j-1] + input[j] + input[j+1]) / 3;

for (int i=1; i<3; i++)
  output[i] = (tmp[i-1] + tmp[i] + tmp[i+1]) / 3;
```



Fused stencil loops

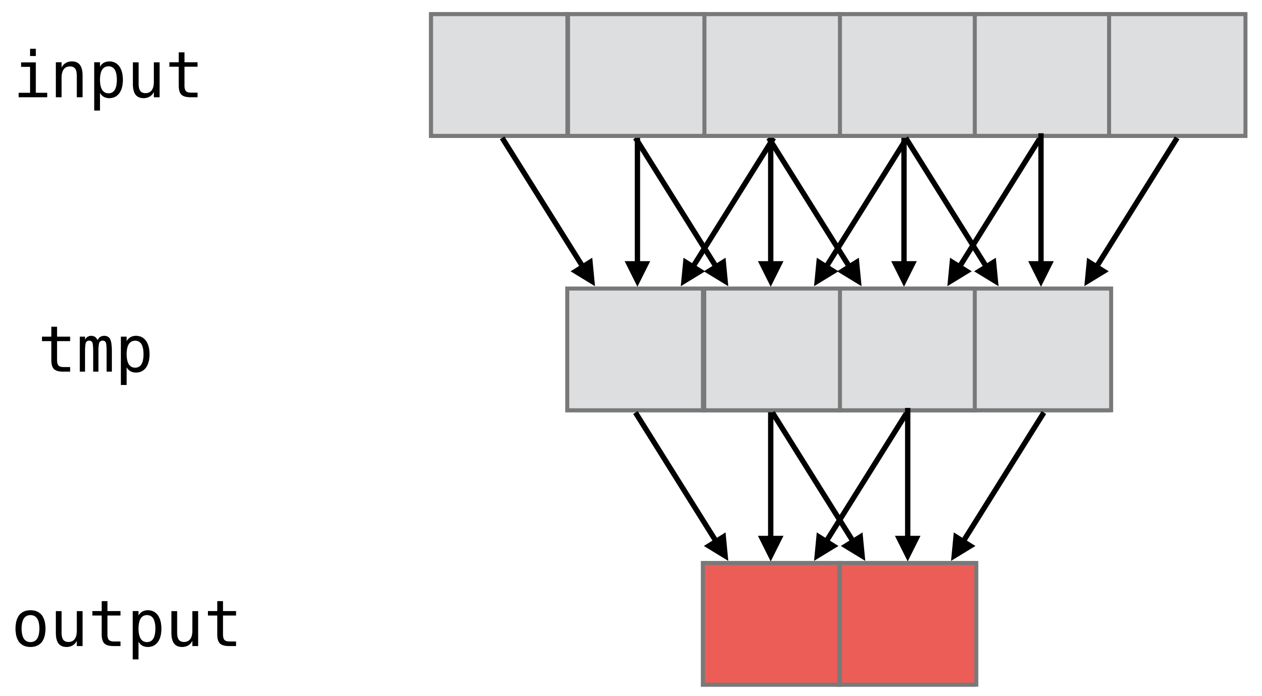
```
for (int i=1; i<3; i++)
  output[i] = (   (input[i-2] + input[i-1] + input[i]  ) / 3
                + (input[i-1] + input[i]   + input[i+1]) / 3
                + (input[i]   + input[i+1] + input[i+2]) / 3
              ) / 3;
```



Buying locality with redundant work in fused stencils

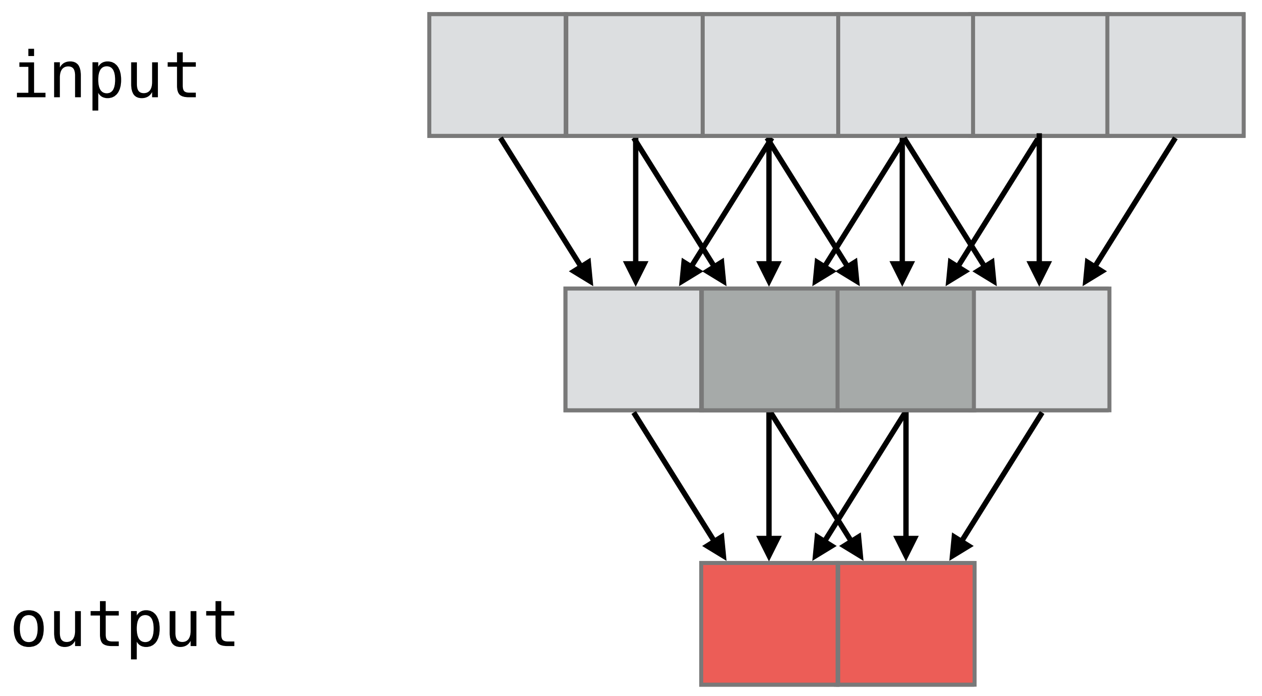
Stencil loops

```
for (int j=0; j<4; i++)
  tmp[j] = (input[j-1] + input[j] + input[j+1]) / 3;

for (int i=1; i<3; i++)
  output[i] = (tmp[i-1] + tmp[i] + tmp[i+1]) / 3;
```



Fused stencil loops

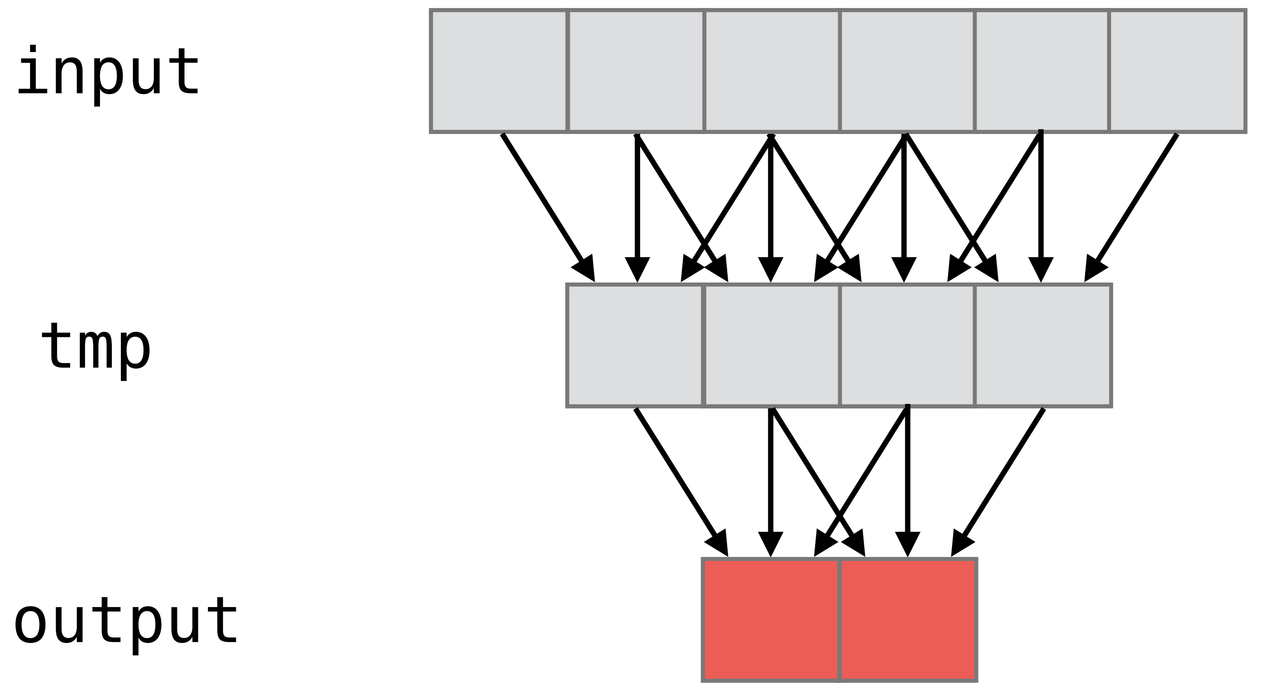
```
for (int i=1; i<3; i++)
  output[i] = (   (input[i-2] + input[i-1] + input[i]   ) / 3
                + (input[i-1] + input[i]   + input[i+1]) / 3
                + (input[i]   + input[i+1] + input[i+2]) / 3
              ) / 3;
```



Buying locality with redundant work in fused stencils

Stencil loops

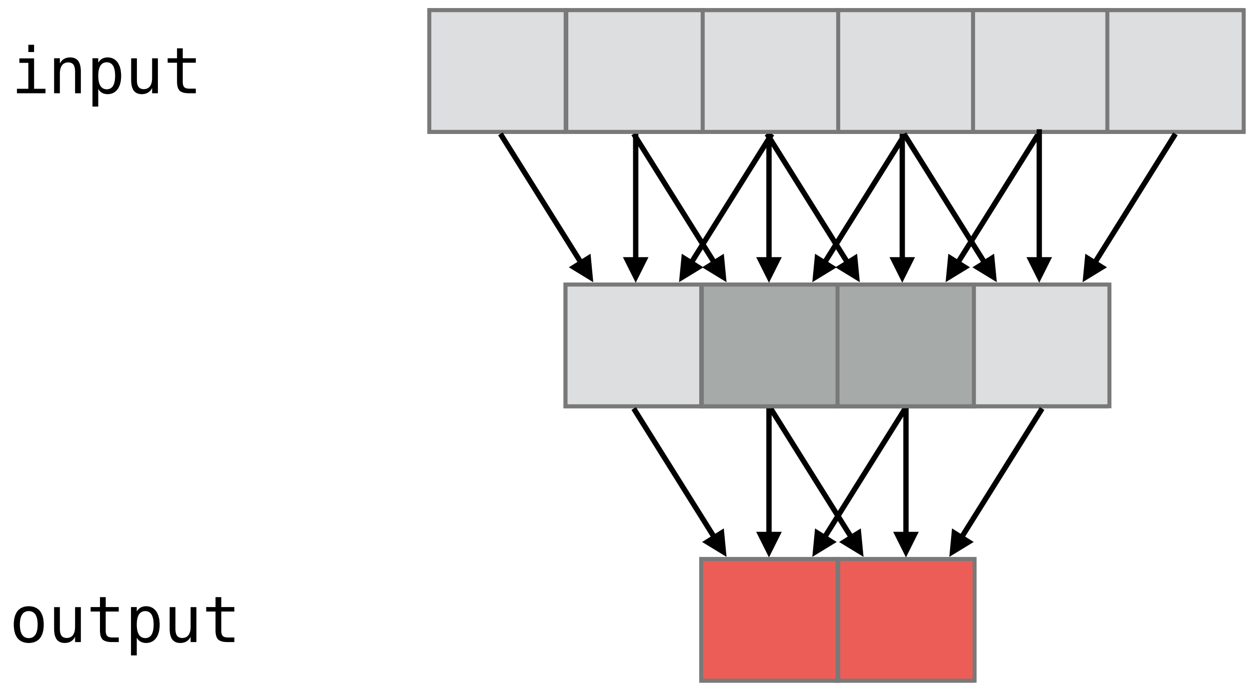
```
for (int j=0; j<4; i++)
  tmp[j] = (input[j-1] + input[j] + input[j+1]) / 3;

for (int i=1; i<3; i++)
  output[i] = (tmp[i-1] + tmp[i] + tmp[i+1]) / 3;
```



8 additions and
4 divides

Fused stencil loops

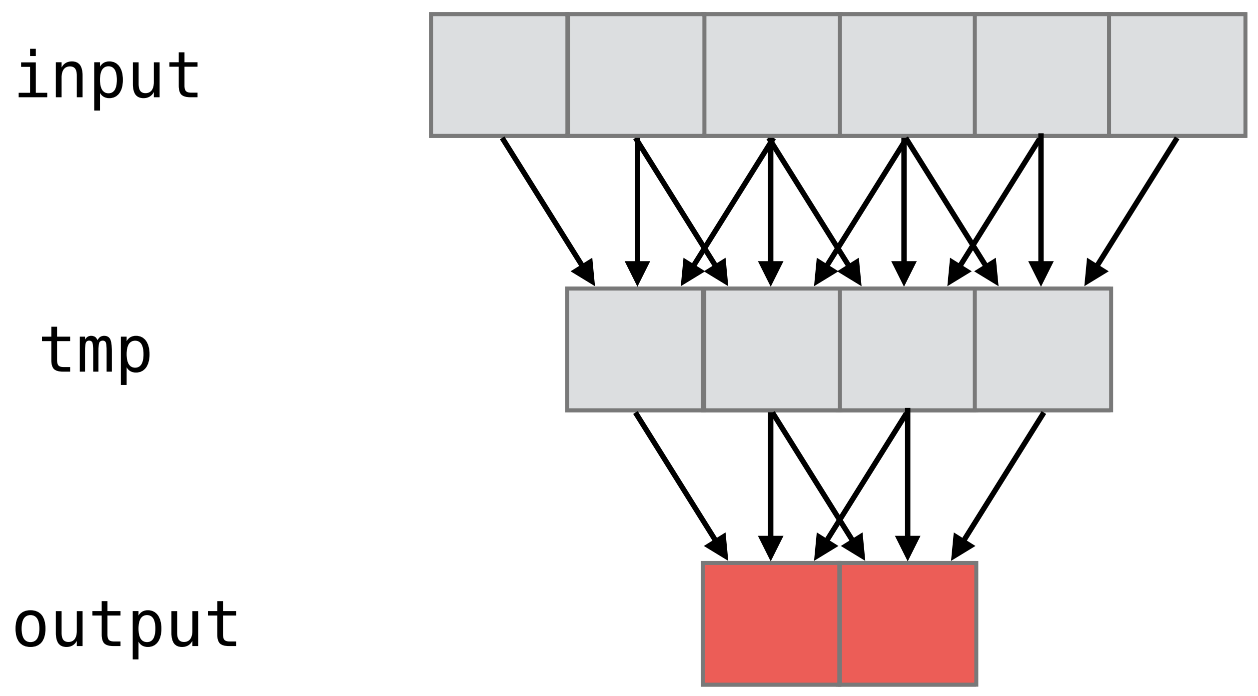
```
for (int i=1; i<3; i++)
  output[i] = (   (input[i-2] + input[i-1] + input[i]   ) / 3
              + (input[i-1] + input[i]   + input[i+1]) / 3
              + (input[i]   + input[i+1] + input[i+2]) / 3
              ) / 3;
```



Buying locality with redundant work in fused stencils

Stencil loops

```
for (int j=0; j<4; i++)
  tmp[j] = (input[j-1] + input[j] + input[j+1]) / 3;

for (int i=1; i<3; i++)
  output[i] = (tmp[i-1] + tmp[i] + tmp[i+1]) / 3;
```

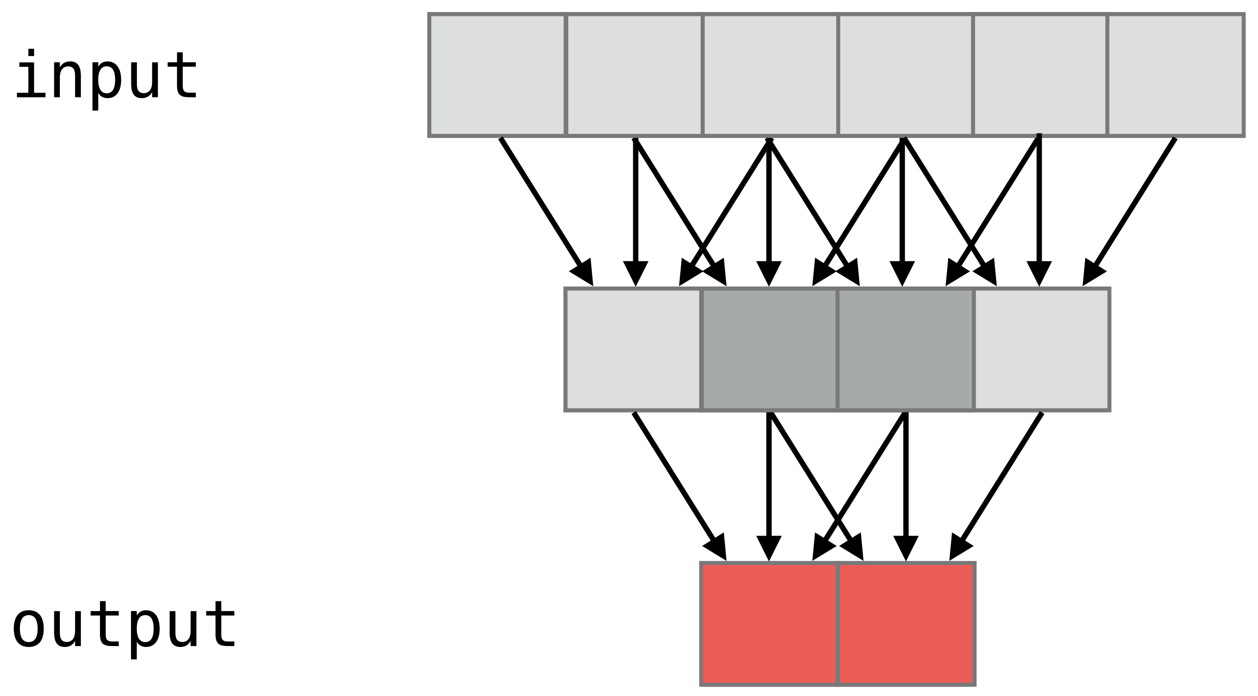


8 additions and
4 divides

4 additions and
2 divides

Fused stencil loops

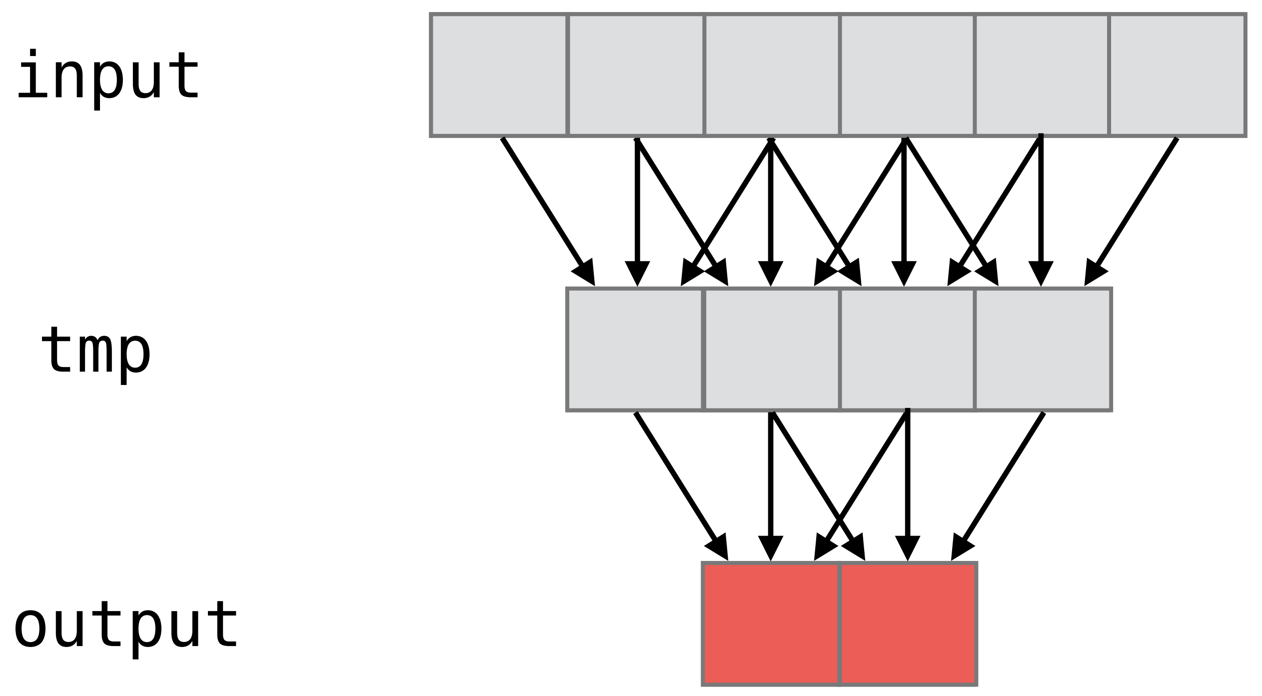
```
for (int i=1; i<3; i++)
  output[i] = (  (input[i-2] + input[i-1] + input[i]  ) / 3
               + (input[i-1] + input[i]   + input[i+1]) / 3
               + (input[i]   + input[i+1] + input[i+2]) / 3
              ) / 3;
```



Buying locality with redundant work in fused stencils

Stencil loops

```
for (int j=0; j<4; i++)
  tmp[j] = (input[j-1] + input[j] + input[j+1]) / 3;

for (int i=1; i<3; i++)
  output[i] = (tmp[i-1] + tmp[i] + tmp[i+1]) / 3;
```

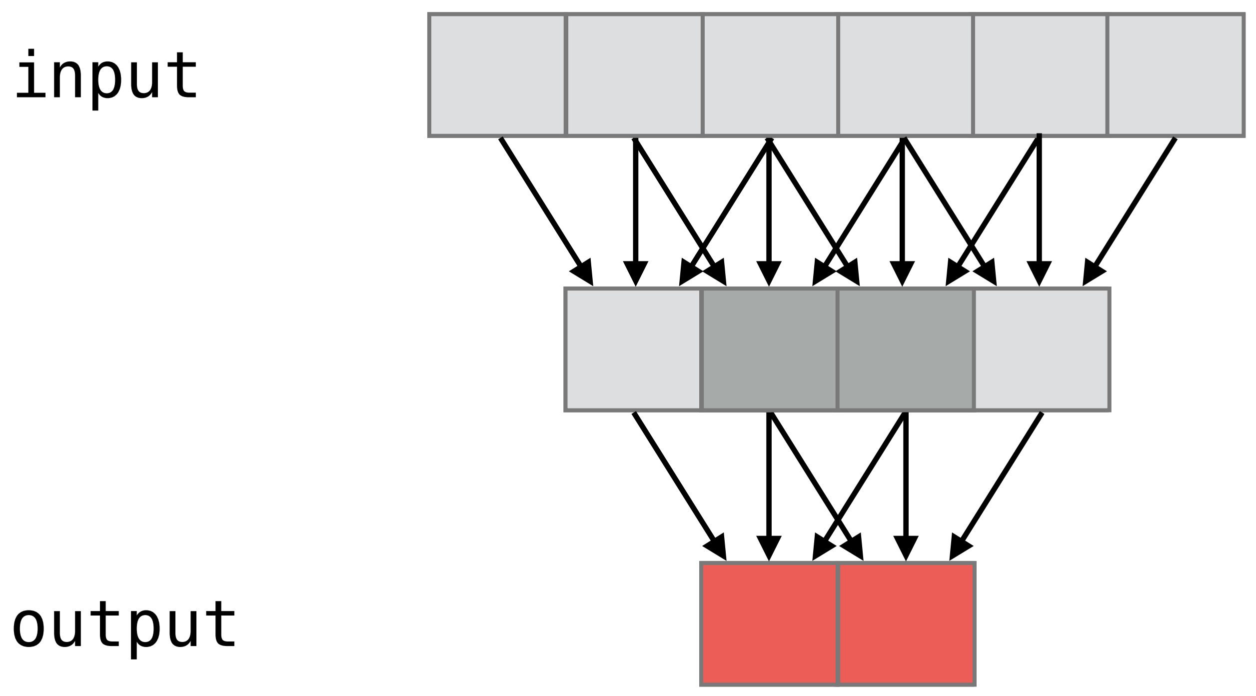


8 additions and
4 divides

4 additions and
2 divides

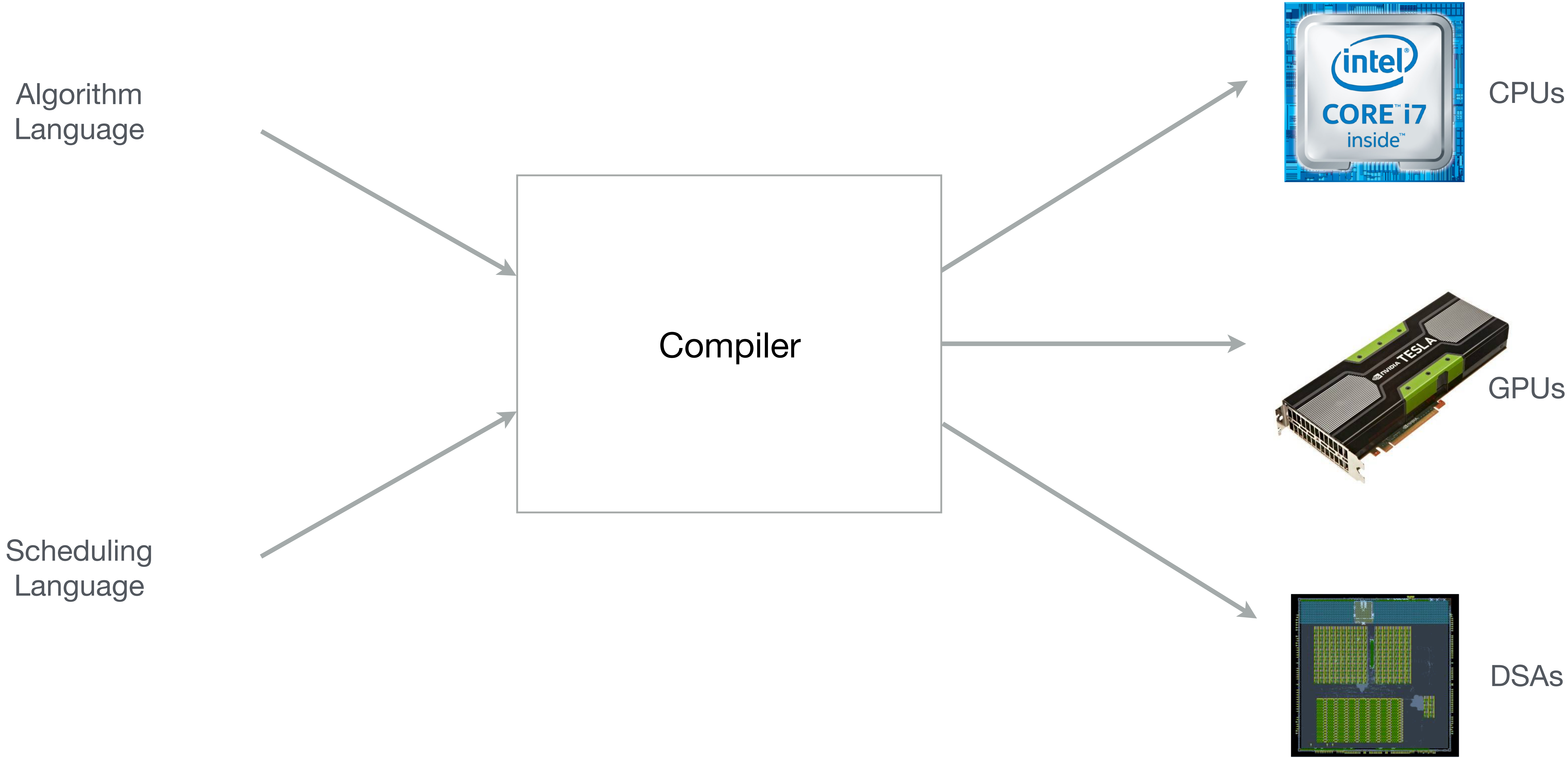
Fused stencil loops

```
for (int i=1; i<3; i++)
  output[i] = (   (input[i-2] + input[i-1] + input[i]   ) / 3
                + (input[i-1] + input[i]   + input[i+1]) / 3
                + (input[i]   + input[i+1] + input[i+2]) / 3
              ) / 3;
```



16 additions and
8 divides

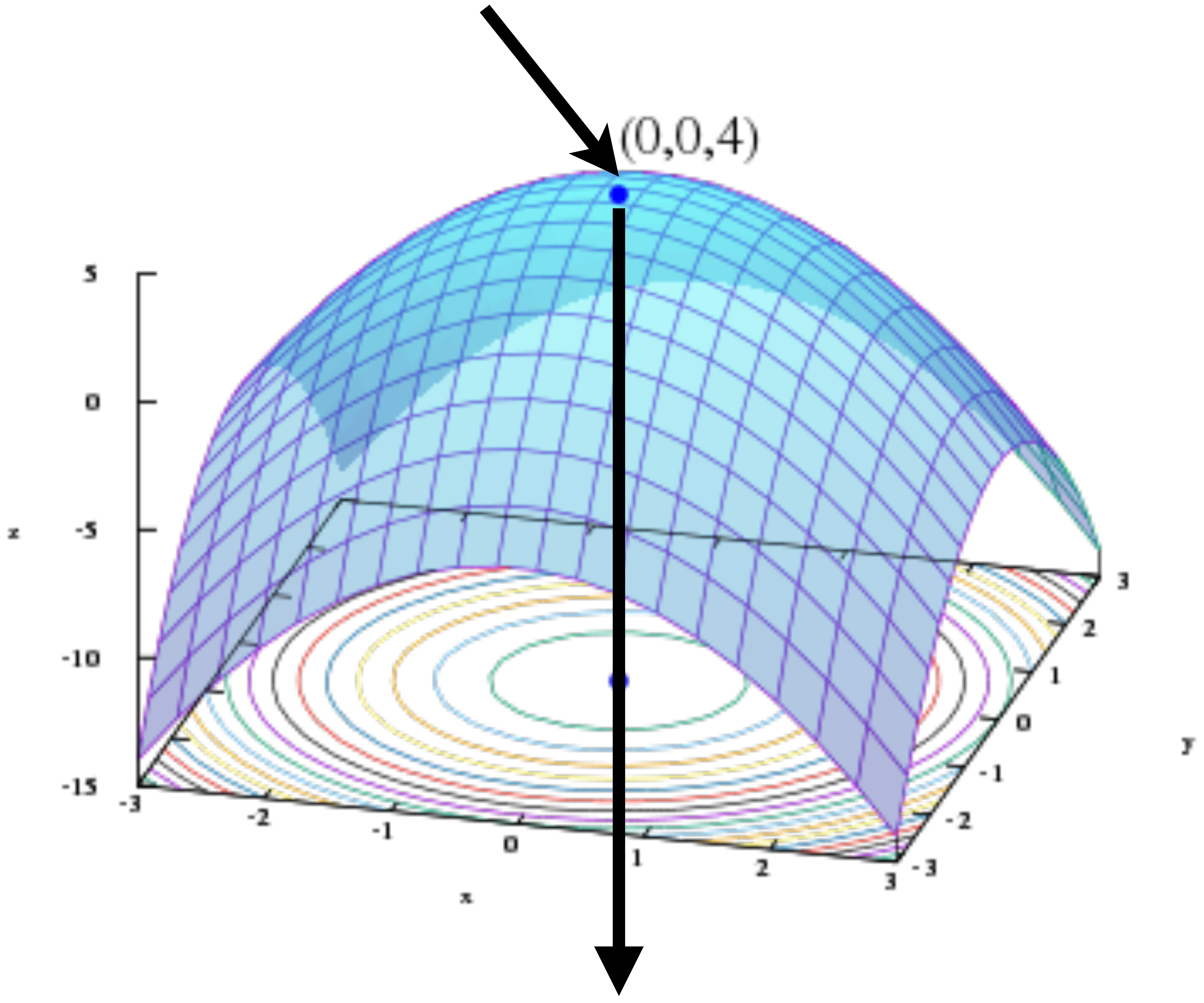
Separation of algorithm from schedules



This idea was most clearly demonstrated in the Halide system

General Principle: Separation of policy and mechanism

Policy is deciding what to do
(decide what transformations to apply)



Mechanism is doing it
(generate code)

Separate by a clean API/language to:

- Solve one complex problem at a time
- Experiment with automatic policy systems without reimplementing mechanism
- Allow users to override default decisions with their own
- Policy tends to evolve faster than mechanism

The Nucleus of a Multiprogramming System. *P. Brinch Hansen* (1970)

Optimization strategies in compilers

Optimization strategies in compilers

1. Greedy or heuristic rewrites

Optimization strategies in compilers

1. Greedy or heuristic rewrites
2. Integer-linear programming

Optimization strategies in compilers

1. Greedy or heuristic rewrites
2. Integer-linear programming
3. Beam search combined with ML

Optimization strategies in compilers

1. Greedy or heuristic rewrites
2. Integer-linear programming
3. Beam search combined with ML
4. Autotuning with hill climbing, genetic algorithms, etc.

Optimization strategies in compilers

1. Greedy or heuristic rewrites
2. Integer-linear programming
3. Beam search combined with ML
4. Autotuning with hill climbing, genetic algorithms, etc.
5. Or pick your favorite optimization strategy and
    - Define an optimization space and a cost function
    - Implement a search procedure

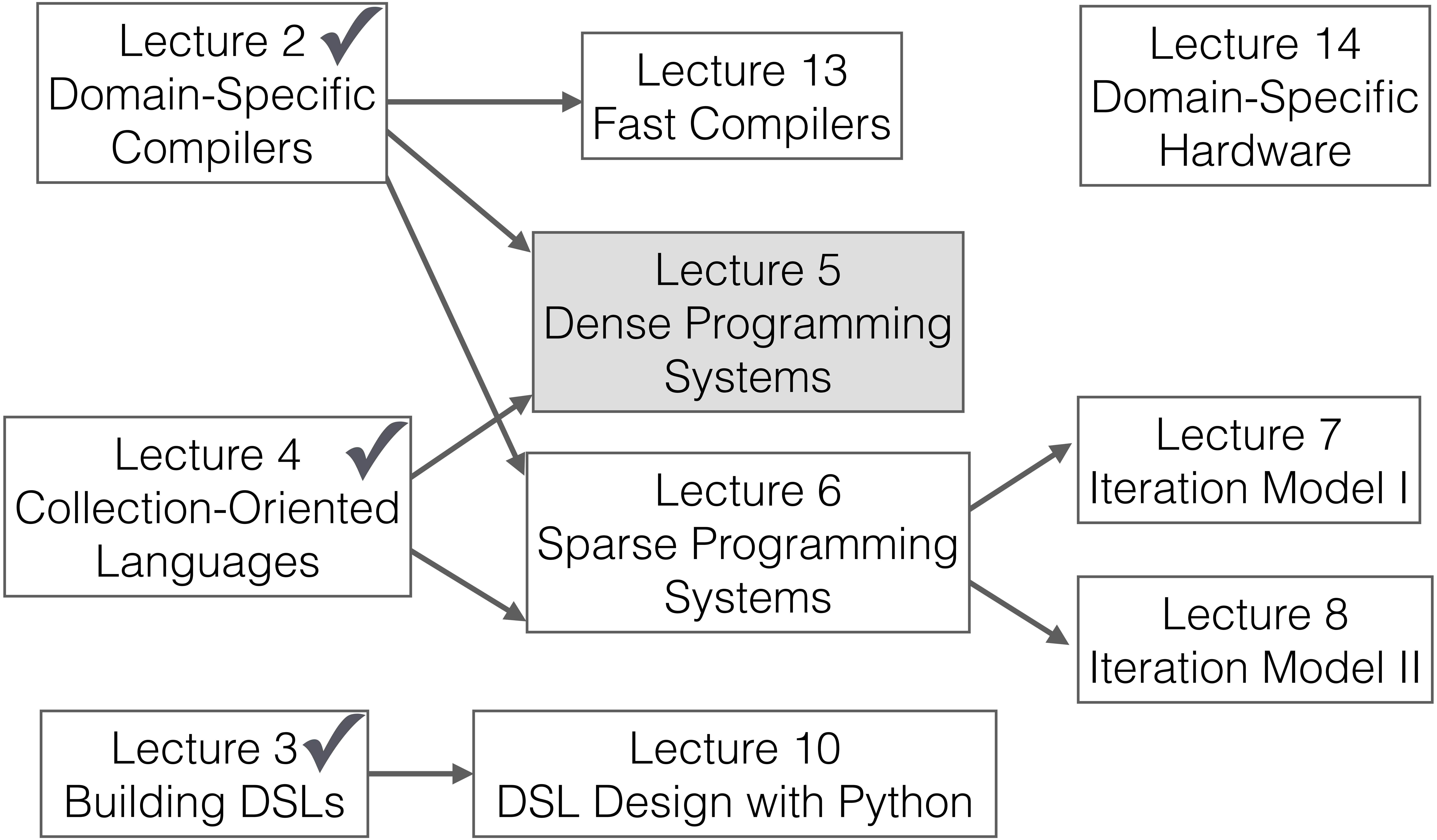
Example: Halide

```
Func halide_blur(Func in) {
 Func tmp, blurred;
 Var x, y, xi, yi;

 // The algorithm
 tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

 // The schedule
 blurred.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
 tmp.chunk(x).vectorize(x, 8);

 return blurred;
}
```

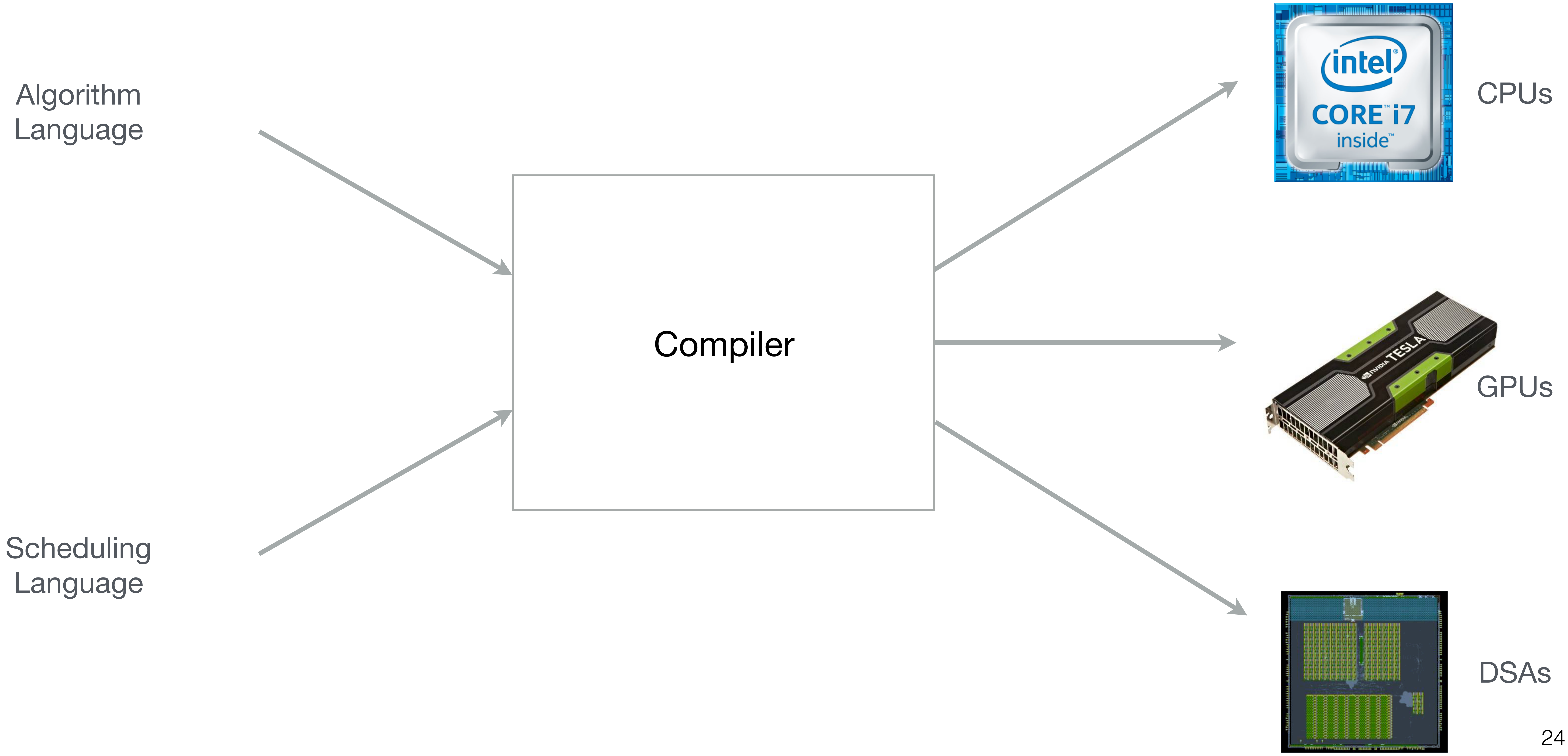
Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *Ragan-Kelley et al.* (2012)

Several auto-schedulers have been developed; a recent autoscheduler uses beam-search

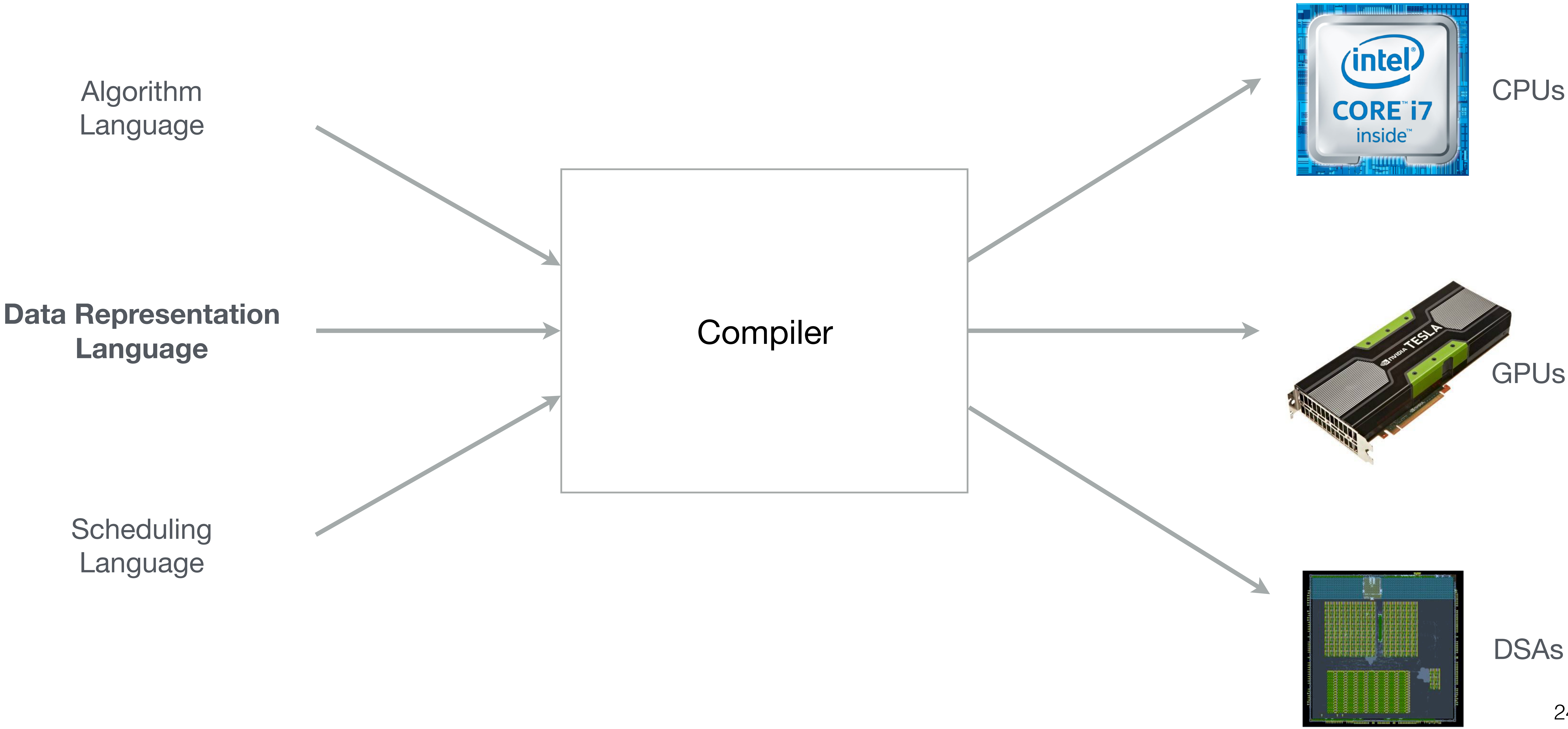
Lecture Overview



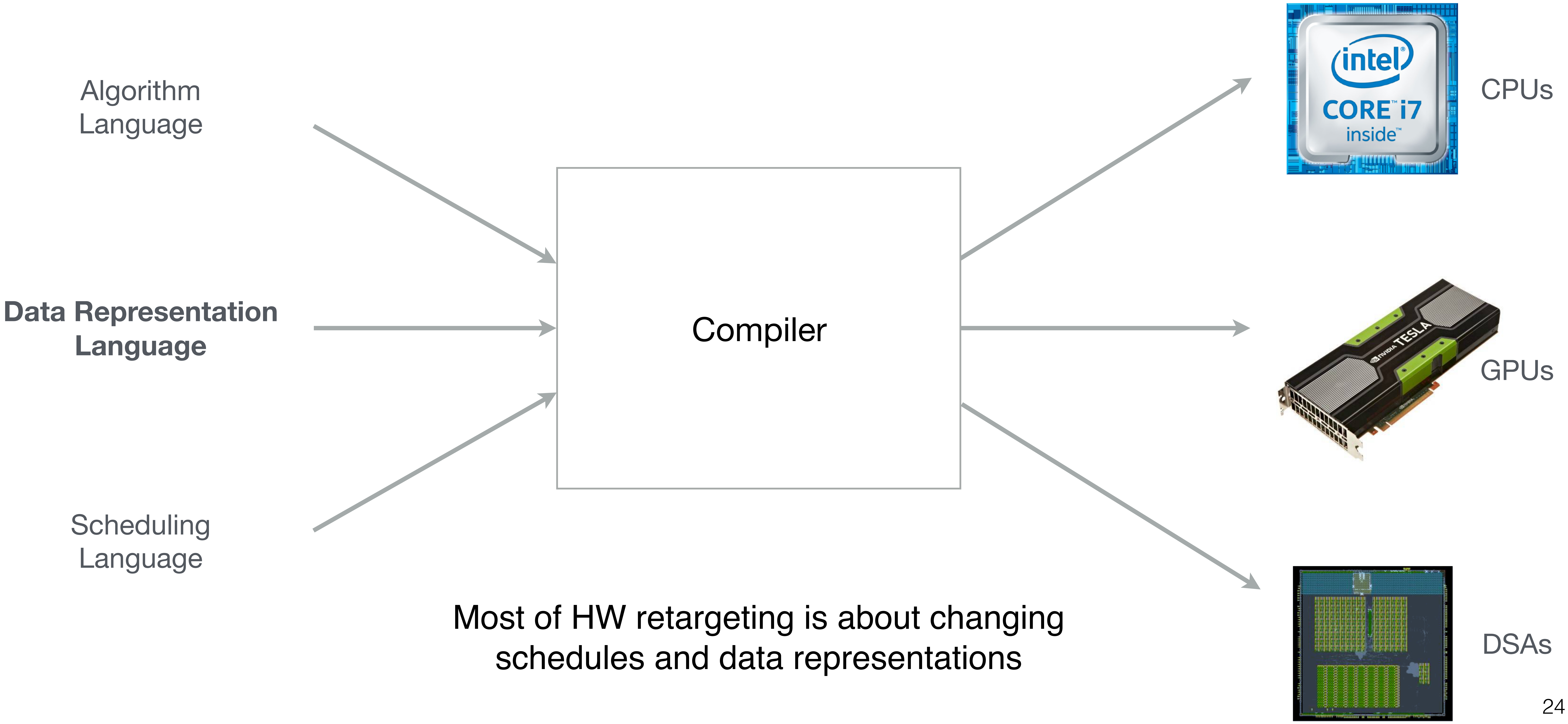
Next up: separation of Algorithm, Schedule, and Data Representation



Next up: separation of Algorithm, Schedule, and Data Representation



Next up: separation of Algorithm, Schedule, and Data Representation



Most of HW retargeting is about changing schedules and data representations