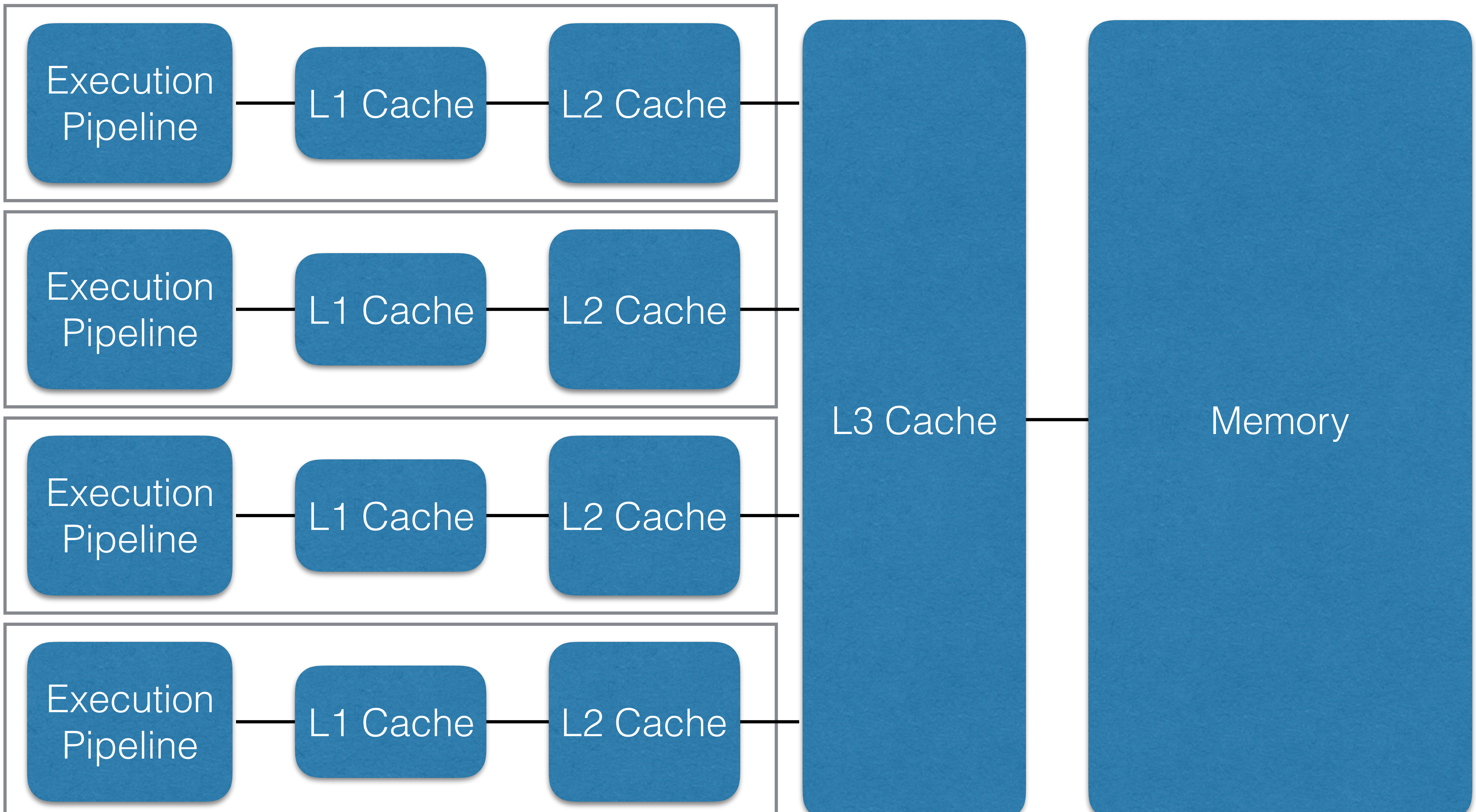
Lecture #08 - Performance Considerations

AMath 483/583

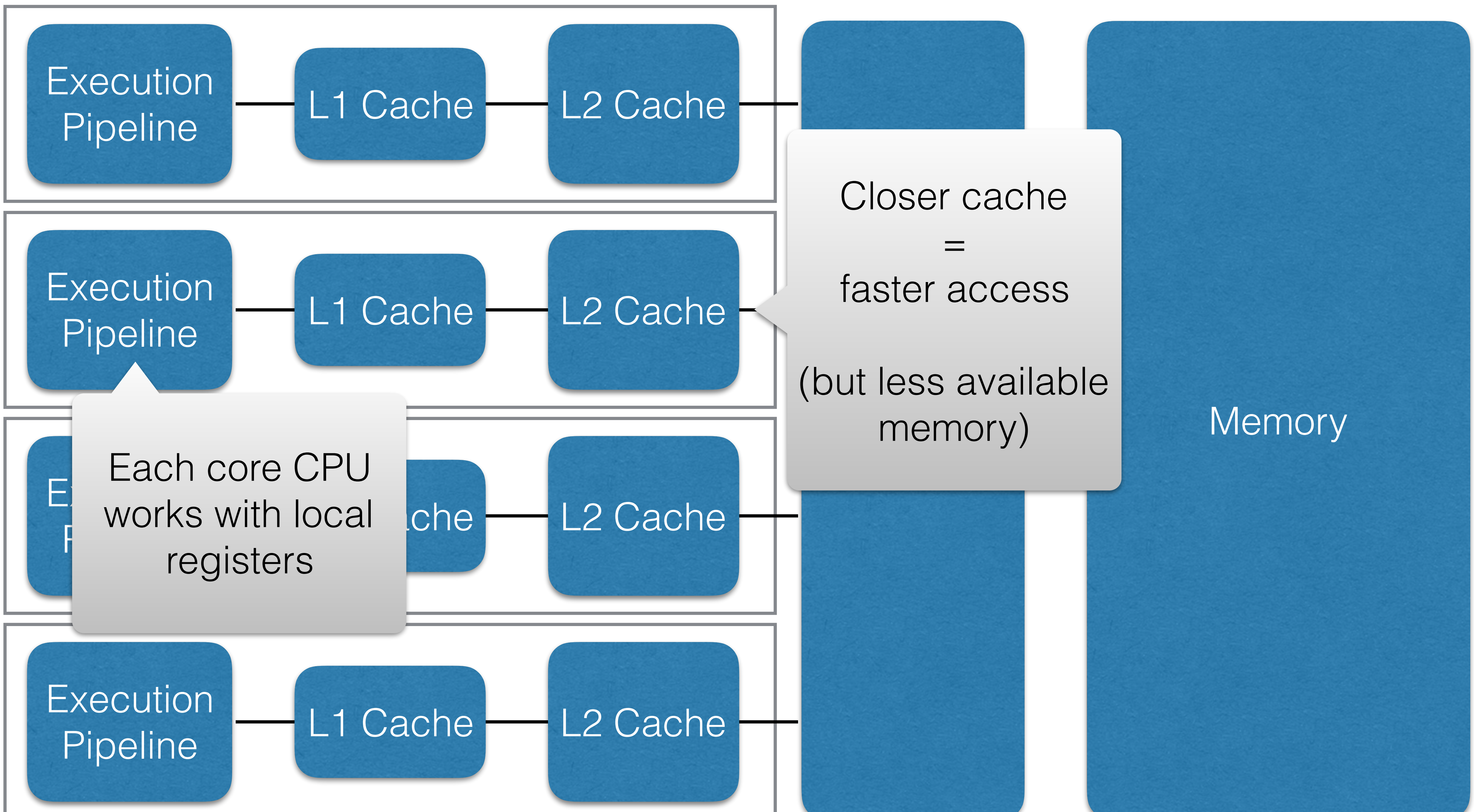
Announcements

- Primary references online
- Next week: parallelization

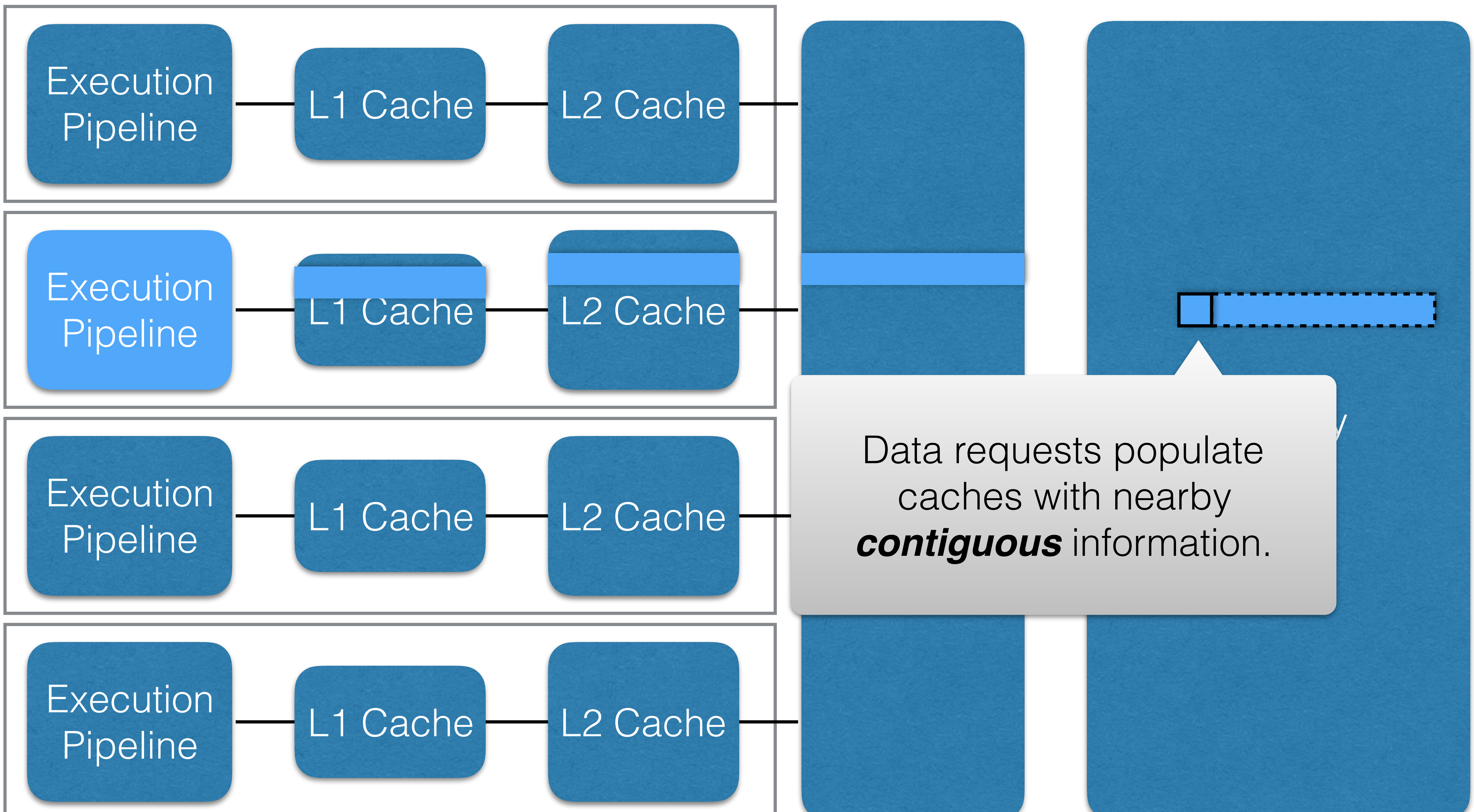
Memory Layout



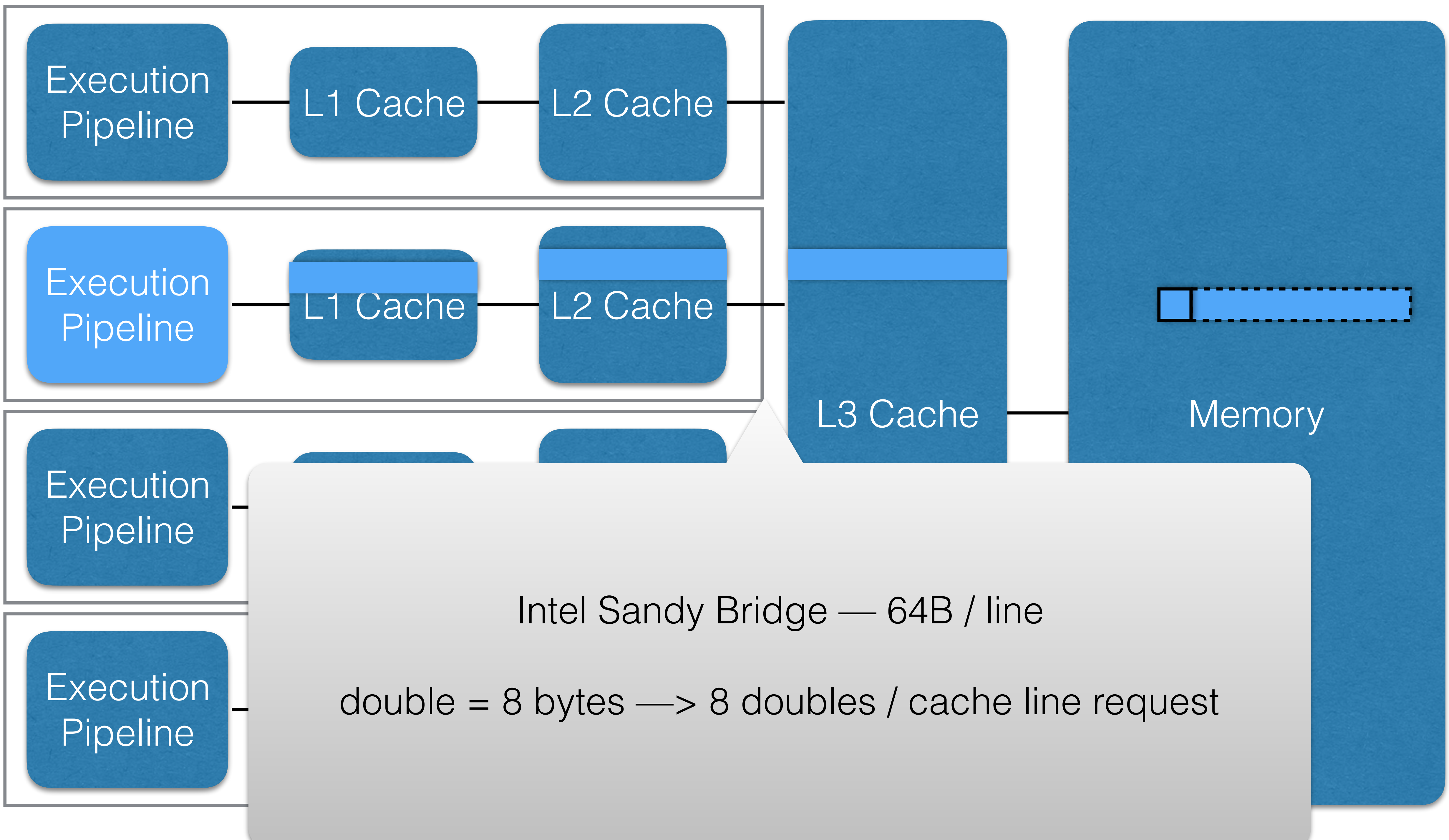
Memory Layout



Memory Layout



Memory Layout



Intel Sandy Bridge — 64B / line

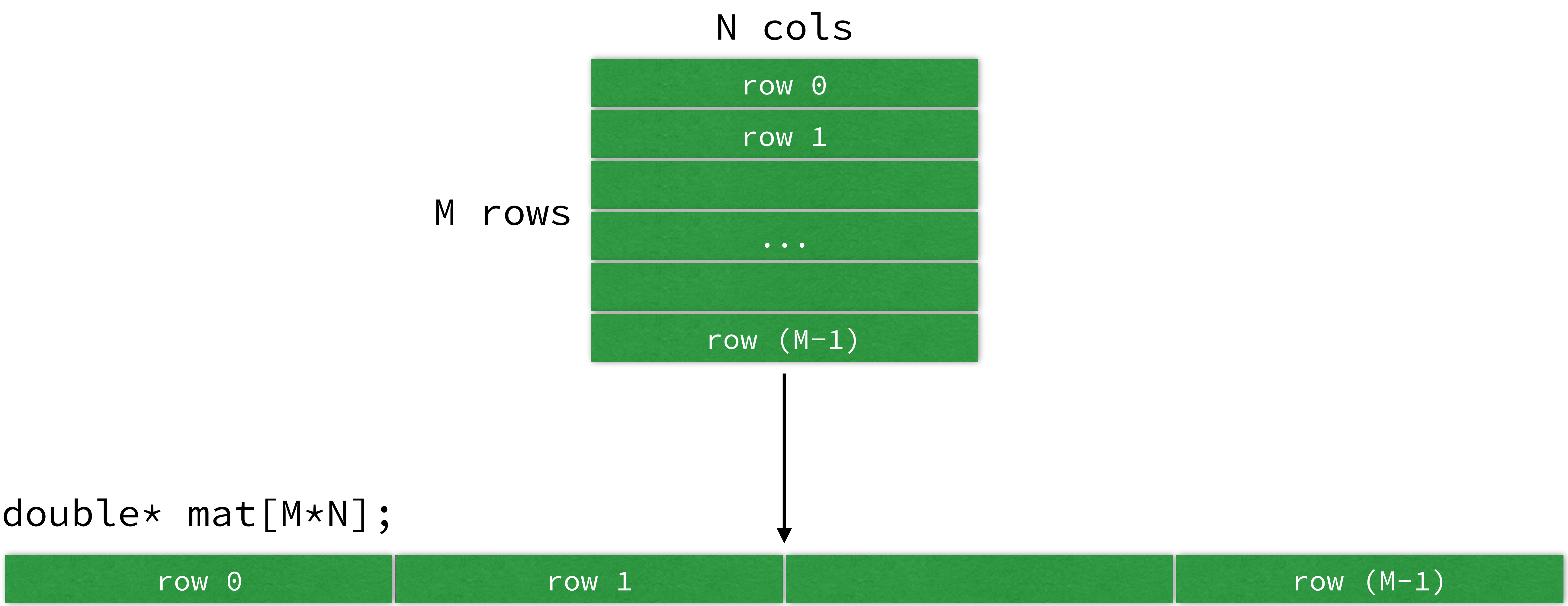
double = 8 bytes —> 8 doubles / cache line request

Main Takeaway

- Data contiguity is essential to fast code

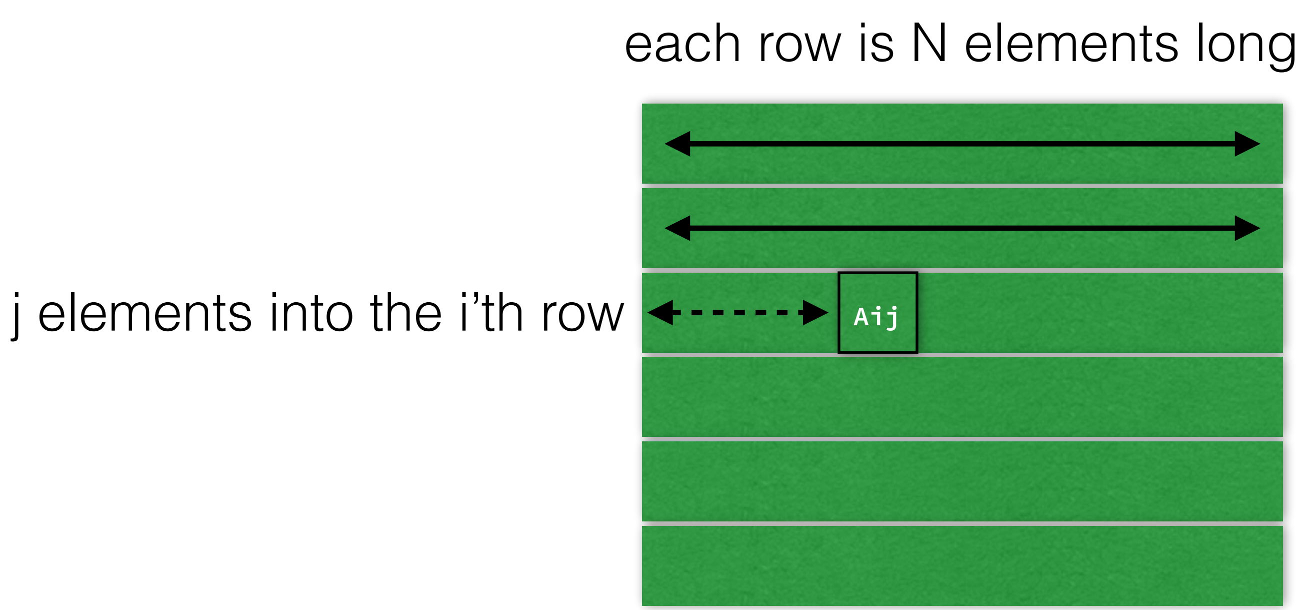
Example: MatMul

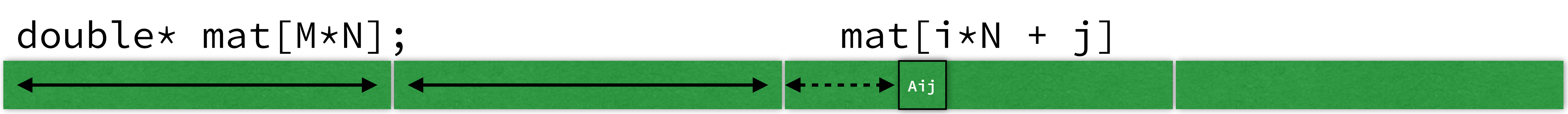
- Matrices typically stored in single array: (row-ordered)



Example: MatMul

- How do you access the (i,j) element?



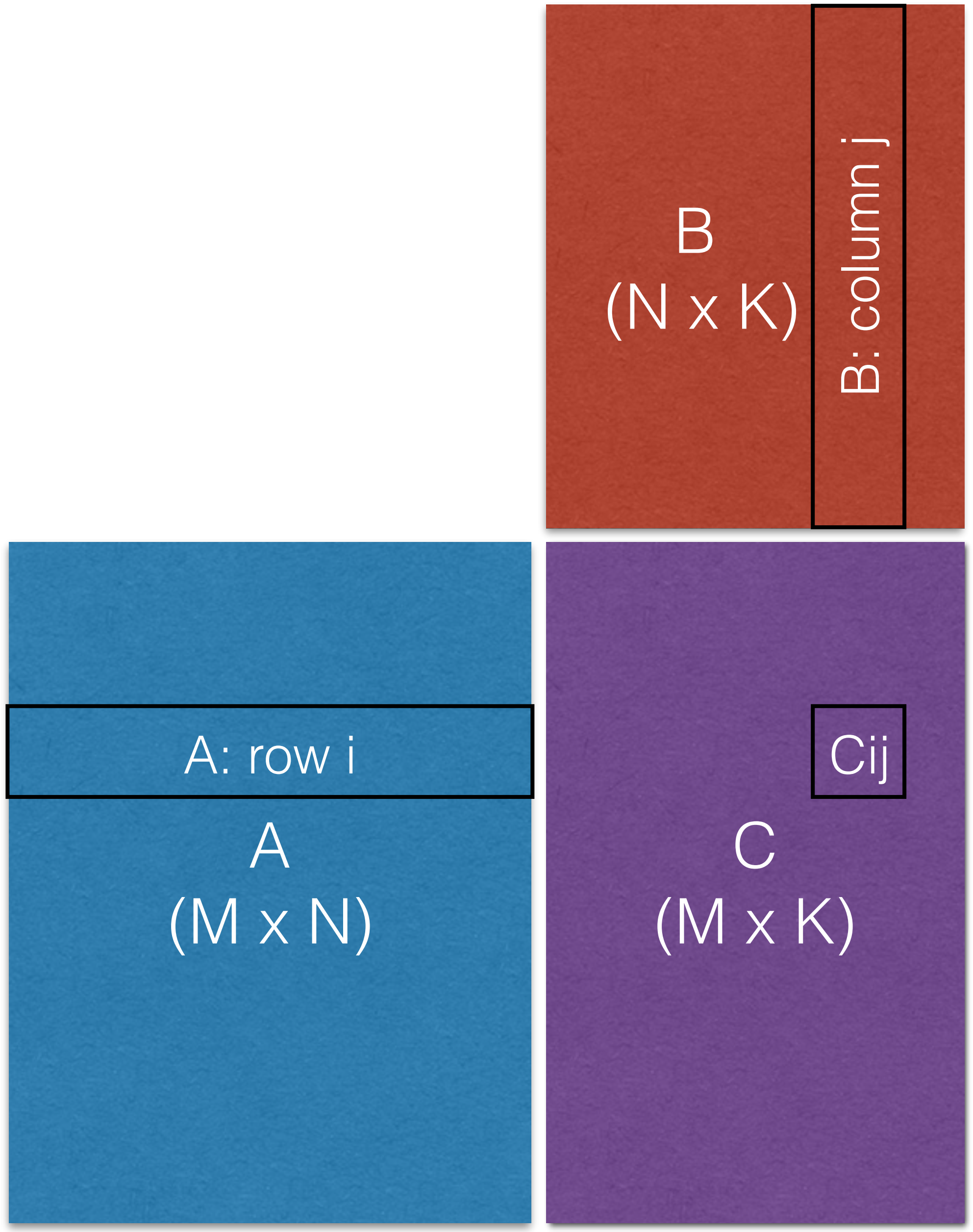


Example: MatMul

- Compute AB = C

$$C_{i,j} = \sum_{k=0}^{N} A_{i,k} B_{k,j}$$

```
// for each row of C
for (int i=0; i<M; ++i) {
  // for each column of C
  for (int j=0; j<K; ++j) {
    // compute the inner product
    for (int k=0; k<N; ++k)
      C[i*K + j] += A[i*N + k]*B[k*K + j];
  }
}
```



Example: MatMul

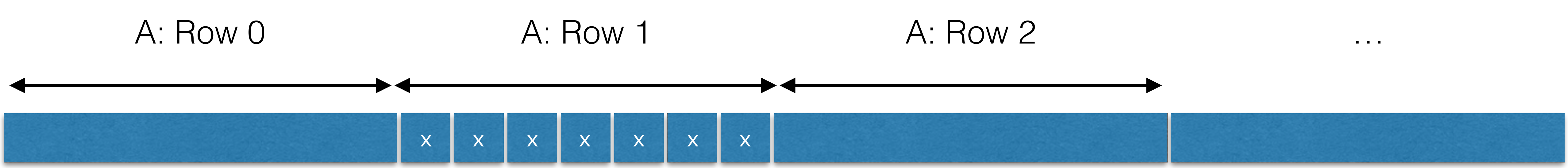
```
for (int i=0; i<M; ++i) {
  for (int j=0; j<K; ++j) {

    for (int k=0; k<N; ++k)
      C[i*K + j] += A[i*N + k]*B[k*K + j];

  }
}
```

How are elts of A accessed? —>

Example: MatMul

How are elts of A accessed? —>

```
for (int i=0; i<M; ++i) {
  for (int j=0; j<K; ++j) {

    for (int k=0; k<N; ++k)
       C[i*K + j] += A[i*N + k]*B[k*K + j];

  }
}
```




```
A[i*N + 0]
A[i*N + 1]
A[i*N + 2]
    …
A[i*N + (N-1)]
```

Example: MatMul

```
for (int i=0;
  for (int

    fo
                                                   j];

  }
}
```

How are elts of A accessed? —>

Good access pattern:

A cache line always contains contiguous memory.

The next element used in the loop is already in the cache.

A: Row 0 …


```
 A[i*N + 0]
 A[i*N + 1]
 A[i*N + 2]
     …
A[i*N + (N-1)]
```

Example: MatMul

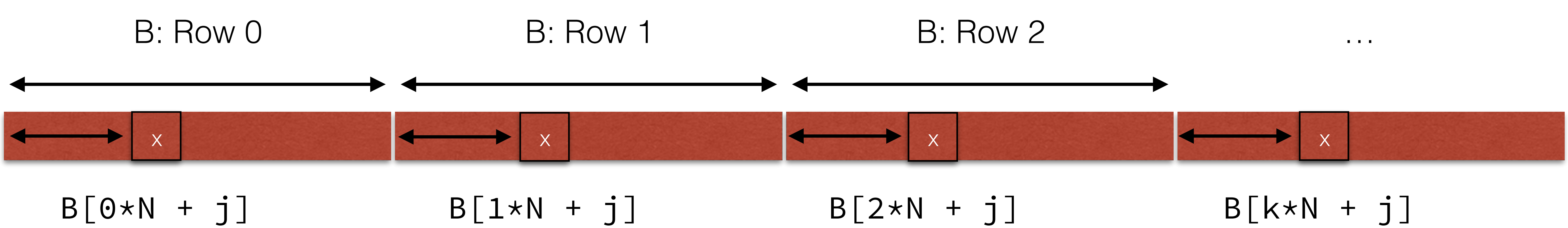
```
for (int i=0; i<M; ++i) {
  for (int j=0; j<K; ++j) {

    for (int k=0; k<N; ++k)
      C[i*K + j] += A[i*N + k]*B[k*K + j];

  }
}
```

<— How are elts of B accessed?

Example: MatMul

```
for (int i=0; i<M; ++i) {
  for (int j=0; j<K; ++j) {

    for (int k=0; k<N; ++k)
      C[i*K + j] += A[i*N + k]*B[k*K + j];

  }
}
```

<— How are elts of B accessed?



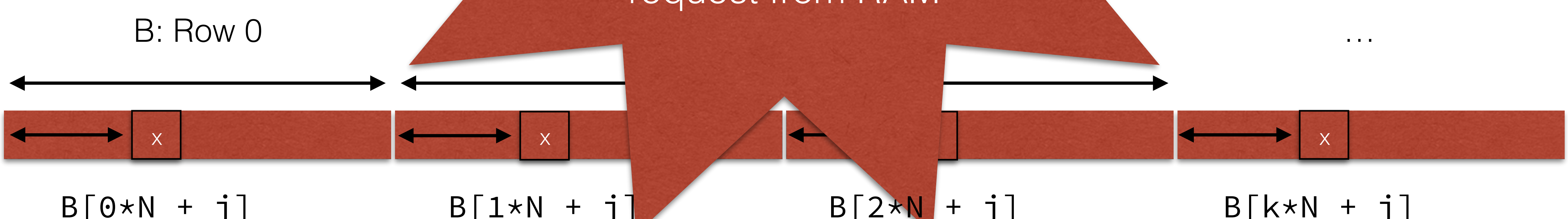
Example: MatMul

```
for (int  =0; i<M;
  for (ir        j<        {

    for (i
      C[i:        k*N + j];

  }
}
```

elts of B accessed?

<u>Bad access pattern:</u>

The next element needed in the loop is not close to the current element —> need to request from RAM



Example: MatMul

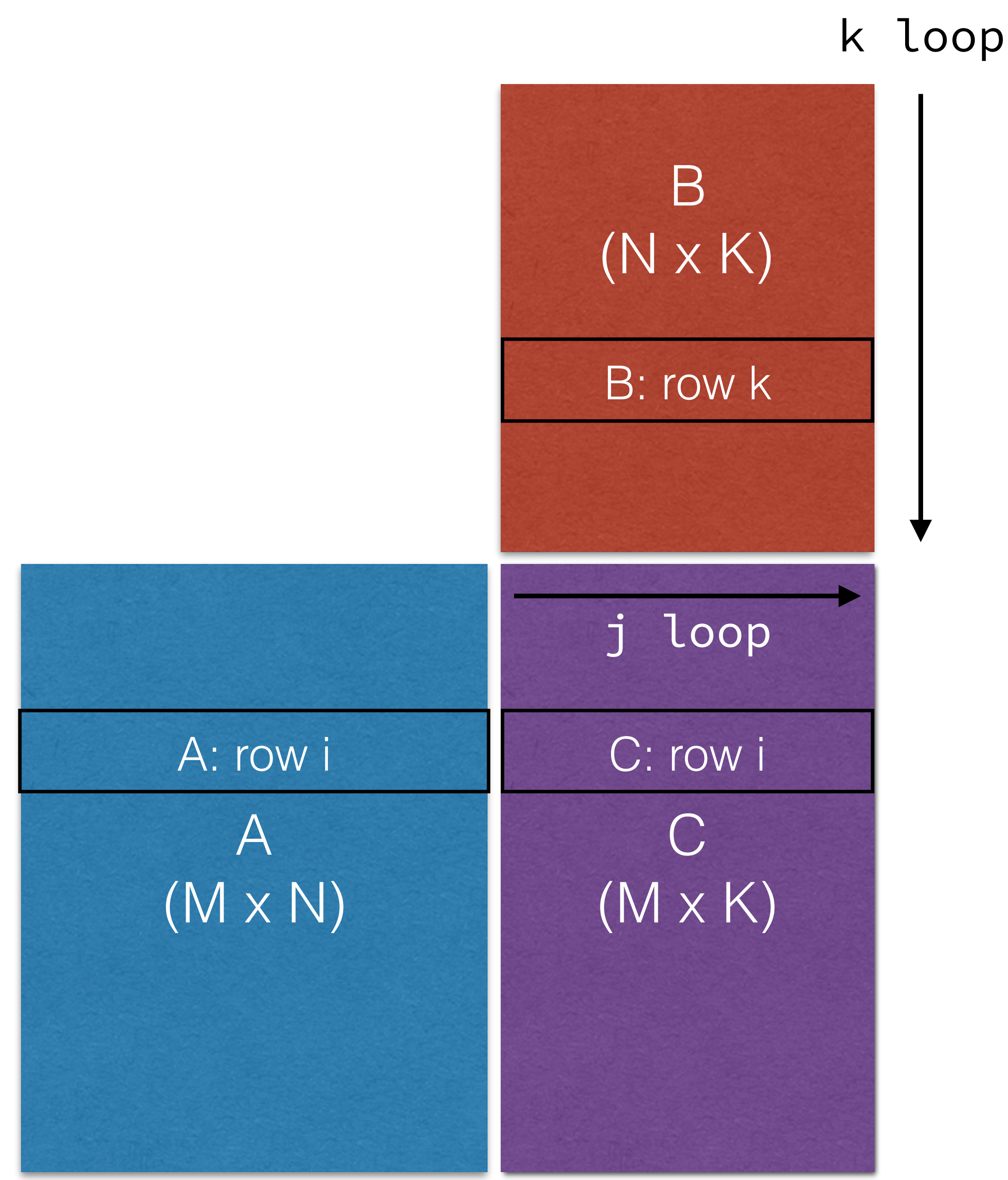
- **One solution**: add each row of B's contribution to all inner products in row i of C

$$C_{i,j} = \sum_{k=0}^{N} A_{i,k} B_{k,j}$$

Swapped loops

```
// for each row
for (int i=0; i<M; ++i) {
  // for each row of B
  for (int k=0; k<N; ++k) {
    // add that row's contrib. to C
    for (int j=0; j<K; ++j)
      C[i*K + j] += A[i*N + k]*B[k*K + j];
  }
}
```

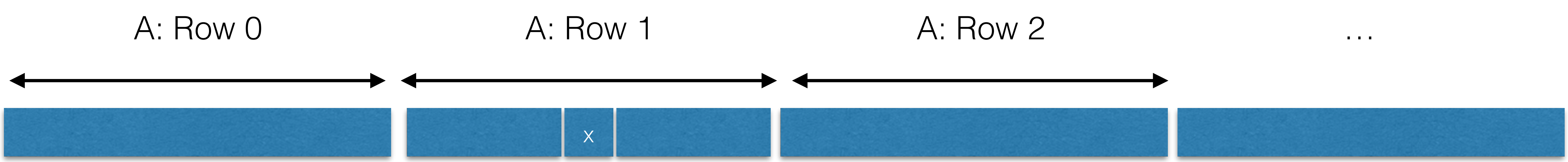
Same expression



Example: MatMul

```
for (int i=0; i<M; ++i) {
  for (int k=0; k<N; ++k) {

    for (int j=0; j<K; ++j)
      C[i*K + j] += A[i*N + k]*B[k*K + j];
  }
}
```

How are elts of A accessed, now? —>



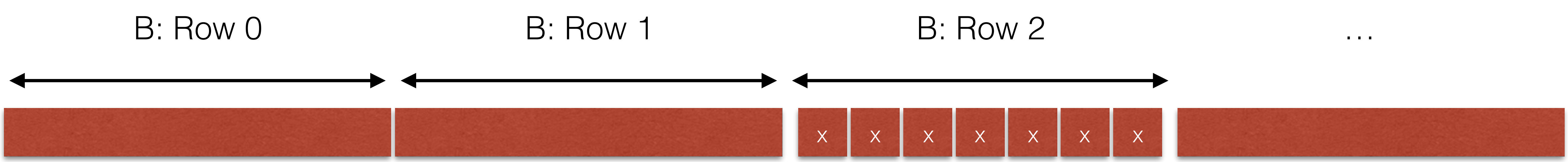
`A[i*N + k]`

`(outer k-loop is contiguous)`

Example: MatMul

```
for (int i=0; i<M; ++i) {
  for (int k=0; k<N; ++k) {

    for (int j=0; j<K; ++j)
      C[i*K + j] += A[i*N + k]*B[k*K + j];

  }
}
```

<— How are elts of B accessed, now?




```
B[k*N + 0]
B[k*N + 1]
B[k*N + 2]
…
B[k*N + (N-1)]
```

Demo

MatMul and speed improvements

Memory Optimization

- Access patterns are key to performance
- *Locality of reference*
    - spatial locality - if a memory location is referenced will nearby locations be ref'd as well?
    - temporal locality - if a memory location is referenced will it be ref'd again soon?
    - branch locality - are there only a few / infrequently occurring possible alternatives in program loop?

Memory Optimization

- General strategies
  - **rearrange** - change layout to increase spatial locality
  - **reuse** - group data accesses to increase temporal locality
  - **reduce** - smaller formats / compression to minimize number of cache lines read

Memory Optimization

- Goal is to reduce / hide “latency” of memory requests
  - (in this case: reduce total number of requests)
- Herb Sutter: “*increasing bandwidth is easy, but we can't buy our way out of latency*"

Operation Costs

- comparisons
- `(u)int` addition, subtraction, bitops, shifting
- `float` addition, subtraction
- indexed array access (subject to cache)
- `(u)int` multiplication
- `float` multiplication
- `float` division, remainder
- `(u)int` division, remainder

Example: Digit Count

- Count number of digits in an `unsigned long`.
- Integer division = “divide + floor”

```
1234 / 10 = 123
123 / 10 = 12
12 / 10 = 1
1 / 10 = 0
```

- Divide while non-zero

Example: Digit Count

```
unsigned int digits_naive(unsigned long v)
{
  unsigned int result = 0;
  do
    {
      ++result;
      v /= 10U;   // very costly
    } while (v);
  return result;
}
```

Example: Digit Count

- *Key observation*: integer division is equally expensive no matter the input. Comparison is cheap.

```
while (1)
{
  if (v < 10) return result;
  if (v < 100) return result + 1;
  if (v < 1000) return result + 2;
  if (v < 10000) return result + 3;

  v /= 10000U;  // same cost as /= 10U
  result += 4;
}
```

Demo

Digit count and expensive operations

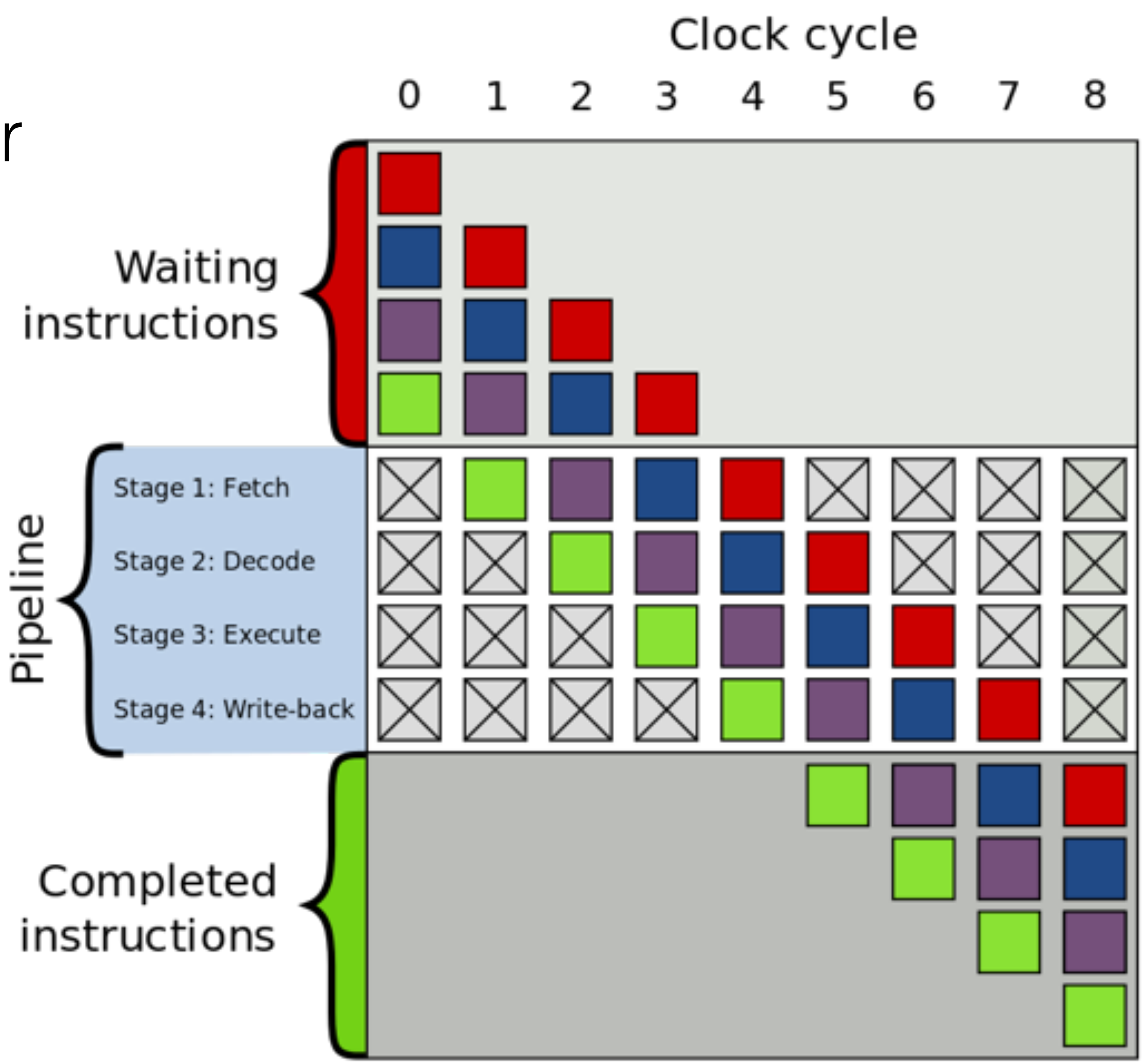
Branch Prediction

- Branch locality - take advantage of processor pipelining (parallelism)

- modern processors perform “branch prediction”

```
if (cond)
{ … }
else
{ … }
```

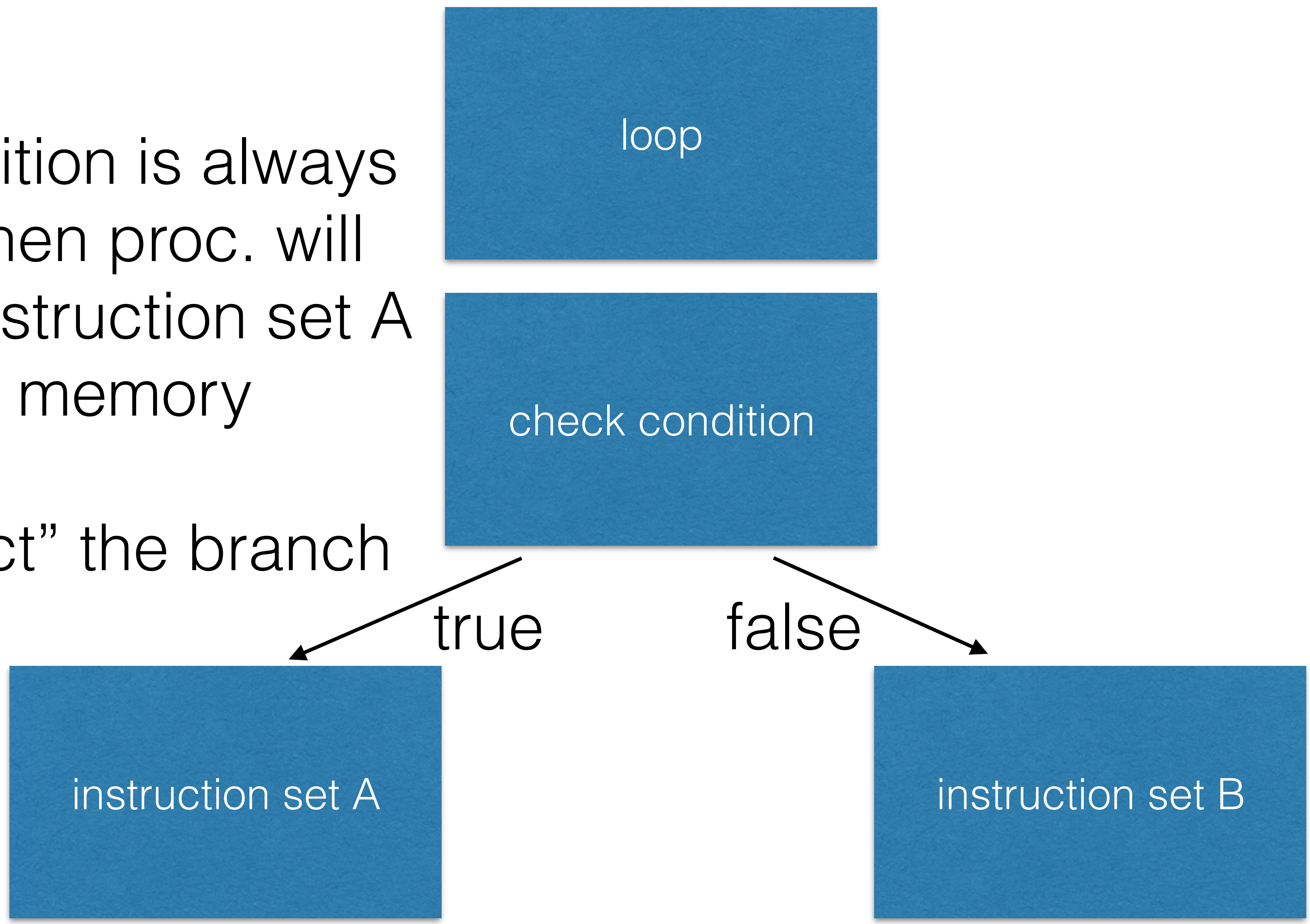
if this was true,
assume still true (check later)



Example: Array Counting

if condition is always true then proc. will keep instruction set A in memory

“predict” the branch



Example: Array Counting

```
long array[size]; // random data
for (size_t n=0; n<size; ++n)
    if (array[n] <= 128) // branch
      {
        threshold += 1;
        sum += array[n];
      }
printf("no. lets < 128 = %d", sum);
```

Example: Array Counting

```
long array[size]; // random data
for (size_t n=0; n<size; ++n)
    if (array[n] <= 128) // branch
        {


        }
printf(                                sum);
```

Branch prediction fail

Data is random —> random branch at each iteration

T F T T F F T F F F T F T

Example: Array Counting

```
long array[size]; // sorted data
for (size_t n=0; n<size; ++n)
    if (array[n] <= 128) // branch
      {


      }
printf(                                  sum);
```

Branch prediction success

sorted data—> high occurrence of correct prediction

T T T T T T T F F F F F F F F

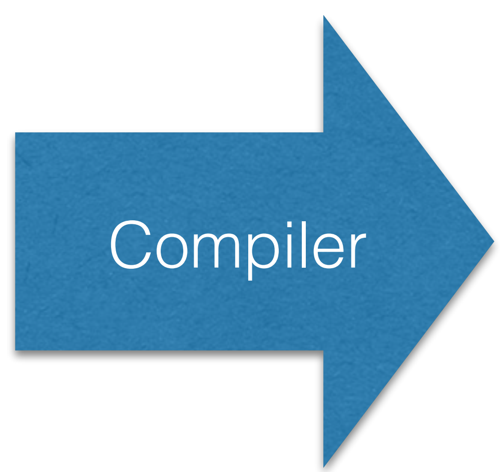
Demo

Branch prediction

Inline Functions

- Functions live in memory — CPU spends time looking for it
- Inlined functions — literally copy contents of function into calling function
- Occasionally effective, depending on situation.

```
inline void swap(int& m, int& n)
{
  int temp = m;
  m = n;
  n = temp;
}

int main()
{
  int a=1,b=2;
  swap(&a,&b);
}
```




```
int main()
{
  int a=1,b=2;
  int temp = a;
  a = b;
  b = temp;
}
```

Compiler Optimizations

- Compilers can optimize code for you (to an extent). More time compiling —> (usually) faster code

```
$ gcc -O1 …  # (or -O2, or -O3, etc.)
```

- See gcc optimizations for details.
- Example:

```
-fdelayed-branch
```

> *If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.*
>
> *Enabled at levels -O, -O2, -O3, -Os.*

Compiler Optimizations

- Still need to manually handle memory contiguity
- Compiler optimizations — transformations that produce provably equivalent code
  - reordering data is not (in general) provably equivalent