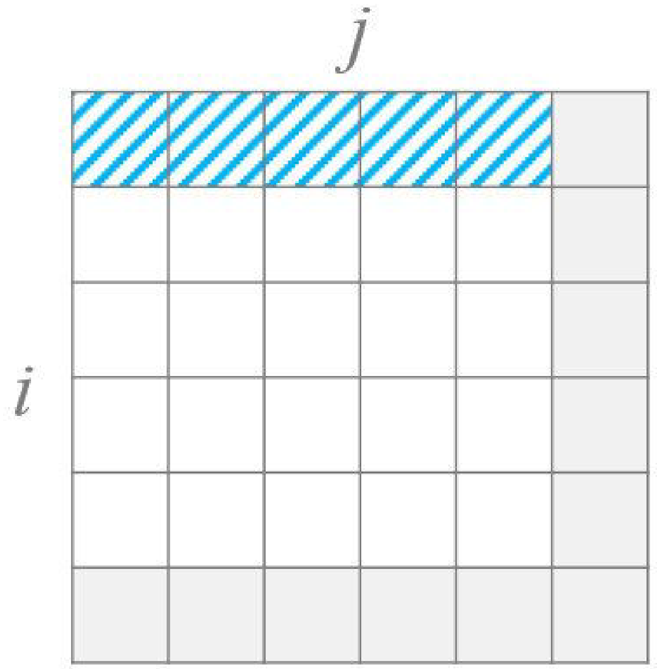
Alignment - Multi-dimensional data structures

```
constexpr int N = 100;
float A[N * N] __attribute__((aligned(64)));
float y[N]     __attribute__((aligned(64)));
float x[N]     __attribute__((aligned(64)));
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j)
    y[i] += A[i * N + j] * x[j];
}
```

Alignment - cont.

- often found in ND-structure (e.g. matrix, N=2) traversal
- first element of subsequent rows/columns not aligned to 64 byte boundary!
- this is ignored when #pragma omp simd is applied!
  → can lead to crash
- solution: apply padding to the row to make it a multiple of 64 bytes in size
- total number of elements allocated for one row/column is called the leading dimension

Padding

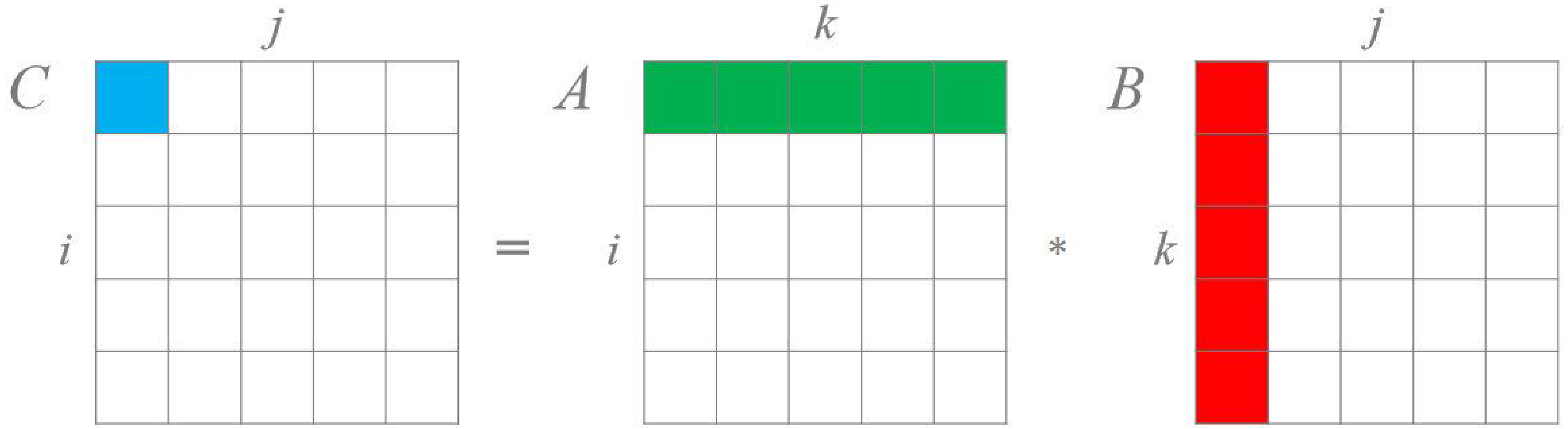


- well known technique in practice
- extra dummy space to speed-up computation (see false-sharing)
- “wasted”-space can sometimes be used to accommodate for in-loop variables

Strided Access

```
constexpr int N = 100;
float A[N * N] __attribute__((aligned(64)));
float y[N]     __attribute__((aligned(64)));
float x[N]     __attribute__((aligned(64)));
for (int j = 0; j < N; ++j) {
  for (int i = 0; i < N; ++i)
    y[i] += A[i * N + j] * x[j];
}
```

Non-unit stride



Strided access

- best performance is achieved when data is accessed with unit (=1) stride, i.e. consecutive elements in an array
- a single packed load/store instruction will be generated
- when facing a fixed non-unit stride the compiler might decide to generate gather/scatter instructions (slow)
  or
  non-vectorized code instead (if N is too large)
- rewrite code if necessary to avoid non-unit stride

Streaming Stores

- data that is the CPU reads/writes needs to be fetched from memory into cache first
- when a memory region is never read, but only written to, these elements consume precious cache space
- cache is for temporal locality, but there is no reuse in sight!
- streaming stores are a special feature of the Xeon Phi vector units (and upcoming CPUs)
- data that is written and not read is not fetched to cache
- writes go straight to memory over separate buffer
- safes cache space and memory bandwidth

Example

- z is never read, only written
- add compiler flag -opt-streaming-stores [auto|never|always]
- use compiler directive before loop: #pragma vector nontemporal

```
constexpr int N = 100;
float z[N], float x[N], float y[N];
#pragma vector nontemporal
for (int i = 0; i < N; ++i)
  z[i] = x[i] + y[i];
```

Structure of Arrays vs. Array of Structures

- complex data structures usually have multiple attributes combined in a single struct/class → *Object Oriented Programming* (OOP) / *data encapsulation*
- this can lead to performance penalties due to non-unit stride access
- even if all elements within the structure are processed at the same time (spatial locality), vector units can be left partially empty

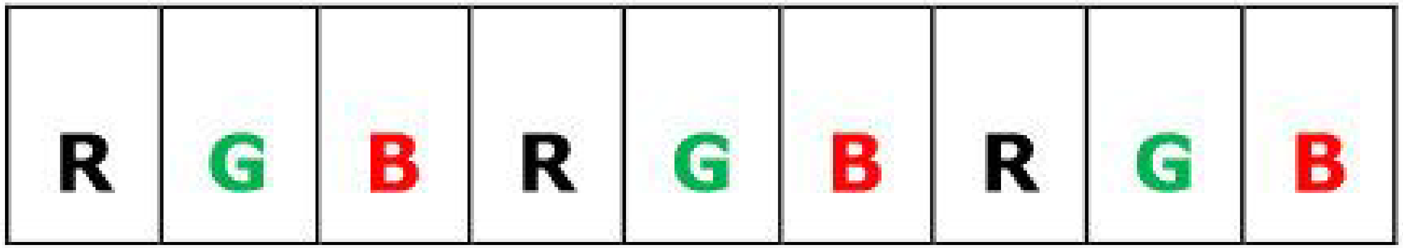
SOA vs. AOS

```
constexpr int NUM_PIXELS; = 100;
struct Pixel {
  float r;
  float g;
  float b;
};
// AOS-approach
std::vector<Pixel> pixels(NUM_PIXELS);
// SOA-approach
struct Pixels {
  float r[NUM_PIXELS];
  float g[NUM_PIXELS];
  float b[NUM_PIXELS];
};
```

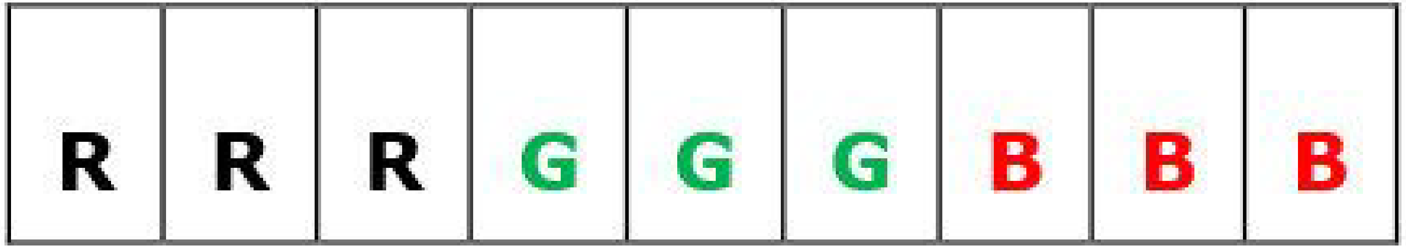
- if all components of point are visited before going to the next → good locality of reference
- but even then, we either pay for expensive loads due to unaligned structs
  or
  the structs are aligned but the vector units are only filled to 75% (at max)
- in case a single member within a struct is visited in a loop, SoA provides better locality of reference AND unit-stride access
- however, it exhibits bad locality of reference when all components of a single struct are required at the same time

SOA vs. AOS - Cont.

Array of Structures



Structure of Arrays



Cache-Blocking

- class of algorithmic techniques
- block data structures to fit in cache by reorganizing memory accesses
- idea: load small subset of larger data set into cache,
  then work on this block while it's in cache
- two types of blocking:
  using: blocking supports spatial locality
  reusing: blocking supports temporal locality
- like all cache-aware techniques: reduces pressure on the main memory bandwidth → speed-up

Blocking-basics

- can be performed both on 1D-data structures or multidimensional data structures
- many applications exhibit multiple accesses to the same memory location
- a cache miss penalty occurs when the data has been evicted from cache when subsequent accesses happen
- blocking usually involves loop-splitting and loop-interchange
- hard for the compiler to do:
  due to strive for correctness → wants to remain loop-iteration order

Blocking-Principle

```
for (int i = 0; i < M; ++i)
 for (int j = 0; j < N; ++j)
  compute(i, j);  // some routine using large data
                // indexed by i and j
```

- for every value of i, compute iterates over the same 0 <= j < N part of the data
- if that part is too big for the machines cache, it will be read from main memory M times
- use blocking, if the order of the M * N iterations doesn't matter

Loop-Splitting (1)

```
constexpr int BLOCK_SIZE = 1;
for (int ii = 0; ii < M; ii+=BLOCK_SIZE)
  for (int i = ii; i < ii + BLOCK_SIZE; ++i)
    for (int jj = 0; jj < N; jj+=BLOCKS_SIZE)
      for (int j = jj; j < jj + BLOCK_SIZE; ++j)
        compute(i, j);
```

Loop-Splitting (2)

```
constexpr int BLOCK_SIZE = 128;
using namespace std;
for (int ii = 0; ii < M; ii+=BLOCK_SIZE)
  for (int i = ii; i < min(ii + BLOCK_SIZE, M); ++i)
    for (int jj = 0; jj < N; jj+=BLOCKS_SIZE)
      for (int j = jj; j < min(jj + BLOCK_SIZE, N); ++j)
        compute(i, j);
```

Loop-Interchange

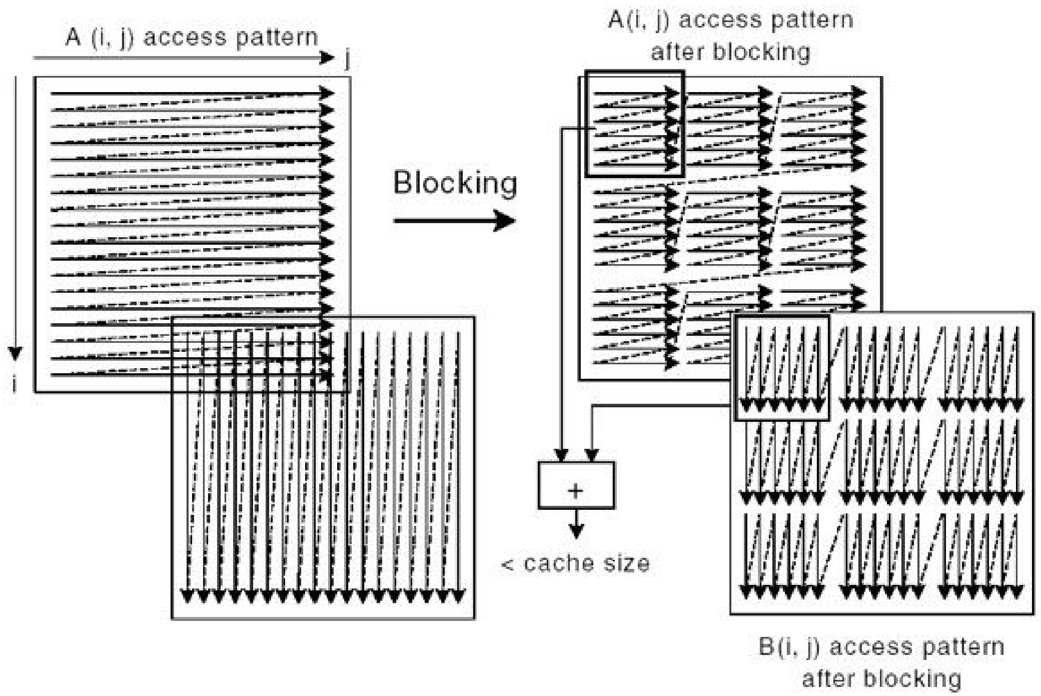
```
constexpr int BLOCK_SIZE = 128;
using namespace std;
for (int ii = 0; ii < M; ii+=BLOCK_SIZE)
 for (int jj = 0; jj < N; jj+=BLOCKS_SIZE)
  for (int i = ii; i < min(ii + BLOCK_SIZE, M); ++i)
   for (int j = jj; j < min(jj + BLOCK_SIZE, N); ++j)
    compute(i, j);
```

- now the access pattern changed
- i and j are now confined to the respective ii-jj-block defined by BLOCK_SIZE

About BLOCK_SIZE

- the BLOCK_SIZE parameter needs to be tuned for the particular application and target machine
- there can be multiple levels of blocking for multiple memory hierarchy transitions, e.g. blocking on L2 cache and blocking on L1 cache
- the BLOCK_SIZE does not have to be the same for every dimension → blocks can be rectangular: this might help for better cache utilization

Blocking vs Non-Blocking

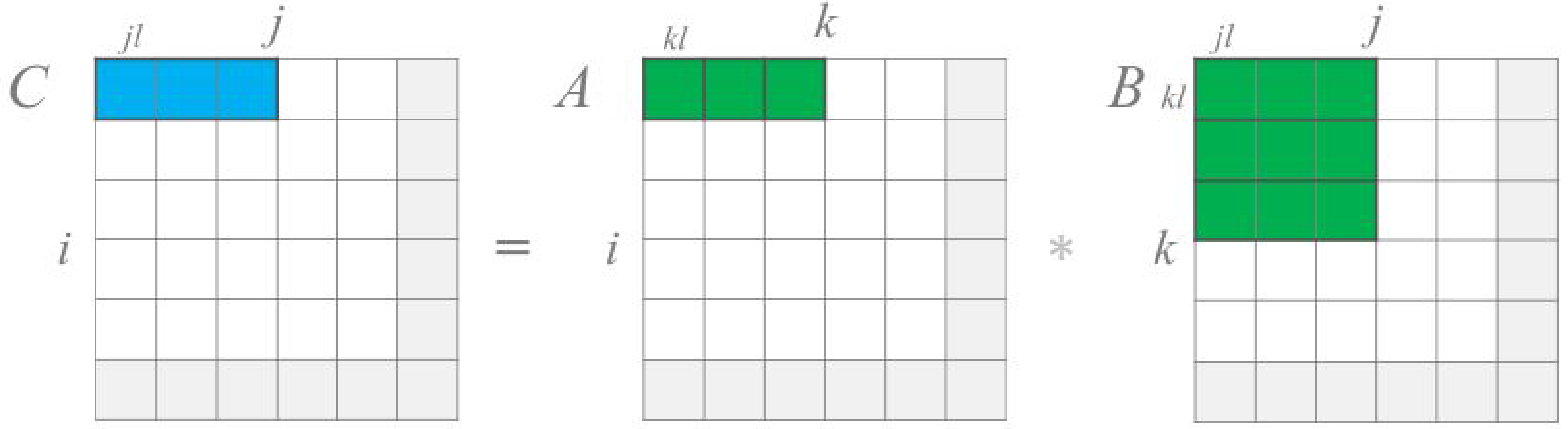


Example: Matrix Transposition

```
// assume square matrix for simplicity
void transpose(double * m, const int N) {
 for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
   std::swap(m[i * N + j], m[j * N + i]);
}
```

- every element is accessed at most once → no temporal reuse possible
- big rows/columns might not fit into cache and evict cache lines containing next element
- improve spatial locality

Example: Matrix Multiplication



- conceptually, B is looped over m times
- If B is too big for the machines cache, B will have to be read m times from main memory
- Blocking can improve temporal locality

Vectorization guidelines

- develop a habit to think about your access pattern even before you start to vectorize
- make your data-structures vector-friendly:
  usually static/dynamic arrays
- make sure your inner-most loop-index corresponds to unit stride access index of array
- prefer structure of array (SoA) over array of structures (AoS), if(!) your access pattern would benefit from vectorization