REGIONALES HOCHSCHUL-
RECHENZENTRUM KAISERSLAUTERN

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

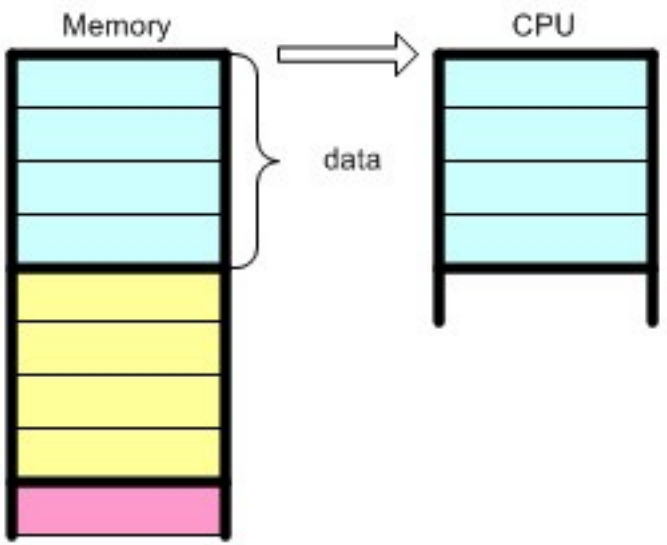
# RHRK-Tutorial

## Vectorization: Alignment

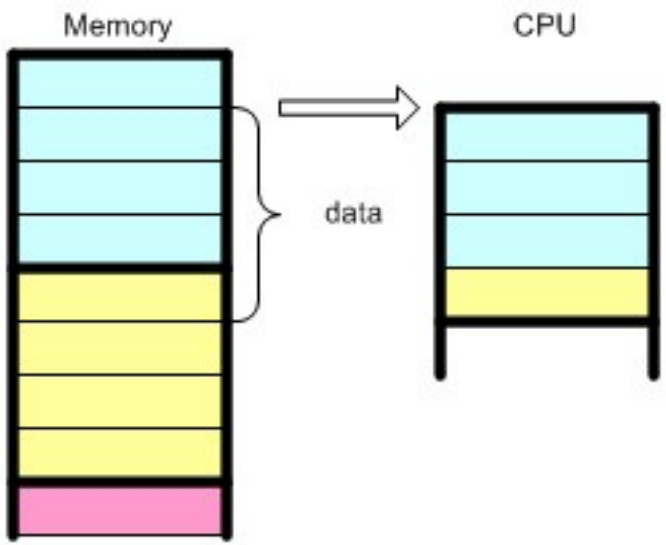
Course instructor: Gabriele Monaco
In charge: Dr. Josef Schüle, RHRK

# Data alignment

- **For performance reasons, data are aligned based on their type**




4-byte memory access for aligned data




4-byte memory access for misaligned data

# Data alignment

- **For performance reasons, data are aligned based on their type**
  - A float type has size 4 bytes, the resulting address must be multiple of 4. In this way loading an array from memory can usually be faster
  - The alignment of a variable/type, can be verified with the keyword
    - `__alignof__`
  - Memory can be allocated with a defined alignment using
    - `void *aligned_alloc(size_t alignment, size_t size)`
  - Or for statically defined variables by adding the attribute aligned
    - `double R __attribute__((aligned(64)))`

```
double d __attribute__((aligned(64)));
double *a = aligned_alloc(64, 4*sizeof(double));
printf("%ld, %ld\n", sizeof(d), __alignof__(d)); //8, 64
printf("%ld\n", (long)&a[0] % 64); //0
free(a);
```

# Alignment for structures (AoS)

| code | offset | size |
|---|---|---|
| `struct Particle {` | | |
| `  char type;` | 0 | 1 |
| `  double x,y;` | 8,16 | 8,8 |
| `  float m;` | 24 | 4 |
| `  double vx,vy;` | 32,40 | 8,8 |
| `};` | | 37 |
| `struct Particle *particle;` | | |
| | | **48** |

```
for (int p_=0;p_<n;p_++) {
  particle[p_].y=c[p_];
}
```

**All fields are close to each others and the structure is well organized, but iterating over a field has huge stride of 48 bytes!**

# Alignment for structures (SoA)

```
struct Particle {
  char *type;
  double *x,*y;
  float *m;
  double *vx,*vy;
};
struct Particle particle;

for (int p_=0;p_<n;p_++) {
  particle->y[p_]=c[p_];
}
```

Worse encapsulation, but the assembly code produced for iterating over a field is way simpler.

The memory accesses are contiguous and vectorization produces a very efficient result here.

This structure is on purpose inefficient (8 bytes could be saved by moving m before), but this is not affecting the loop for the SoA

The AVX512 typically loads 64 bytes of memory contiguously, however an extension can allow access with stride, still better to avoid it though!

# Data alignment and vectors

- Vector memory loads try to fill a full register (64 bytes for the ZMM)
- Data must be aligned to that boundary, otherwise multiple (partial) loads may be required
- What happens if our data isn't aligned?
  - The compiler may restructure the code with *peeled* loops to handle the data until the alignment boundaries, this may not be vectorized
  - Later it will start to vectorize properly
- Still the compiler may need to be informed that pointers are aligned (just like we did for non overlapping)
  - `__builtin_assume_aligned(ptr, size) //on gcc`
  - `#pragma vector aligned //on intel compiler`

# Compiler flags

- Besides pragmas, the compiler can be configured while calling it from the shell
- It can be instructed to optimize the code in various ways, sometimes it can restructure our code to make it easier to vectorize (e.g. reorder or skip function calls)
- By default the compiler doesn't really vectorize, it needs to be instructed explicitly (usually the optimization levels 2 and 3 also vectorize the code when possible)
- We should also inform the capabilities of the target system, otherwise it may suppose AVX instructions are not supported
- Finally we can force it to produce some optimization reports, to see for instance which loops where vectorized or why others were not

# Compiler flags

| Meaning | Intel compiler | GNU compiler |
|---|---|---|
| compiler invocation | icc (C) icpc (C++) | gcc(C) g++ (C++) |
| Basic optimization | -O | -O |
| compile only | -c | -c |
| Vectorizing reductions | | -ffast-math |
| Vectorizing AVX (256-bit) | -xavx | -mavx |
| Vectorizing AVX (512-bit) | -xCOMMON-AVX512 | -mavx512f -mavx512cd |
| Reports | -qopt-report=1<br>-qopt-report-phase=vec | -ftree-vectorize<br>-fopt-info-loop-optimized |
| Assembler code | -S | -S |
| Optimizing function calls | -ipo | -flto |
| OpenMP | -qopenmp | -fopenmp |

# Compiler flags (AVX specific)

- Instruction set specific features are enabled in the ICC compiler through the -xCODE or -axCODE, where CODE can be an extension (AVX) or family. The -ax version includes multiple execution paths and may produces more portable code
- Use -axCORE-AVX512 to allow inclusion for most basic instructions in avx512, COMMON-AVX512 also allows more advanced instructions like conflict detection (avx512cd), both are supported on cores from the Skylake family

```
-O3 -axCOMMON-AVX512
```

- In GCC use the -mMACHINE option to specify capabilities, can use the one present in cpuinfo (e.g. -mavx512cd), processor families can be specified too

```
-O3 -ftree-vectorize -mavx512f -mavx512cd -ffast-math -march=skylake-avx512 -flto
```

- **High Performance Computing on Elwetritsch**

## Vielen Dank
## Thank You