Durham University
Department of Computer Science

Core I—Introduction to HPC

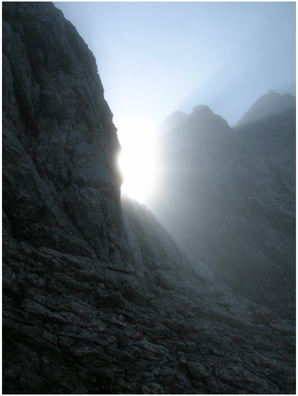
Session IV: Algorithms and alignment

Dr. Weinzierl

Michaelmas term 2018

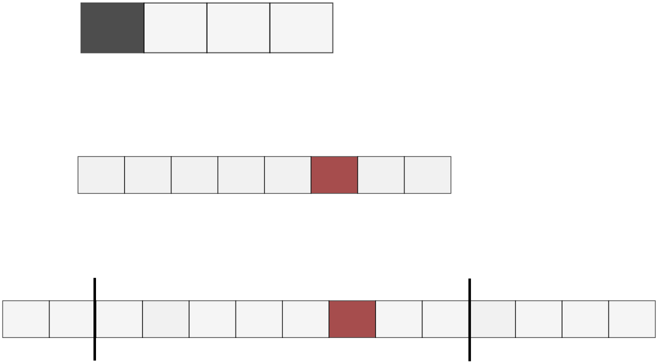
Outline

Durham University
Department of Computer Science



Alignment
Padding and AoS vs. SoA

Individual data loads

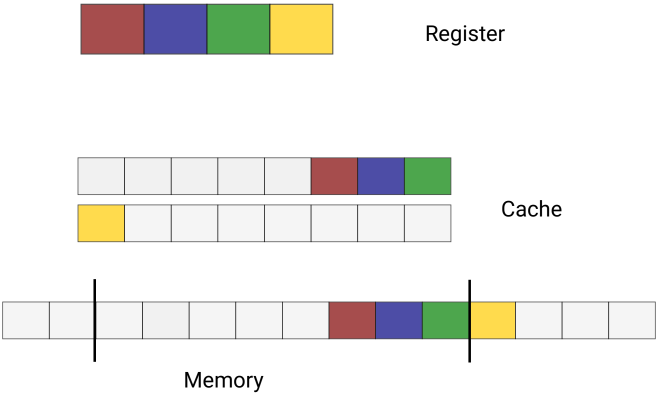


You can write programs that appear to load/inspect individual variables (1-8 bytes), but

- the memory subsystem (RAM, virtual memory, caches) physically works with larger blocks
- typically 64 bytes for cache lines
- 4KB up to 2MB typical for memory pages
- accessing a memory address loads the *cache line* that contains it.
- this is 64 byte aligned

AVX/SSE data loads

Durham University
Department of Computer Science



You can write programs that appear to load/inspect individual variables (1-8 bytes), but
Unaligned load

1. load first cache line
2. bring in first three values (red, blue, green)—one by one
3. load second cache line
4. bring in yellow datum
5. compute

Alignment

Durham University
Department of Computer Science

```
void * malloc(size_t size);
```

Default memory allocation: The `malloc()` function allocates `size` bytes of memory and returns a pointer to the allocated memory.

- The address of a block returned by malloc or realloc in the GNU system is always a multiple of eight (or sixteen on 64-bit systems).
- Malloc alignment guarantees that loading a “builtin” data type will never load more than one cache line.
- To vectorise efficiently, we’d need 64 bit alignment

Using aligned loads and stores

Durham University
Department of Computer Science

```
double *c = malloc(4*sizeof(double));
/* Load 4 doubles into
 * vector register (aligned) */
__m256d c_ = _mm256_load_pd(c);
```

```
double *c = malloc(4*sizeof(double));
/* Load 4 doubles into
 * vector register (unaligned) */
__m256d c_ = _mm256_loadu_pd(c);
```

- Aligned load on unaligned address $\Rightarrow$ segfault
- Compilers will therefore use unaligned load instructions *unless* they can prove the addresses are aligned

$\Rightarrow$ If we know data are aligned, *tell* compiler

Aligned allocations on stack

Durham University
Department of Computer Science

- Straightforward allocation

```
double foo[10];
int bar[4];
```

- C/C++ 2011 and later:

```
#include <stdalign.h>
/* 64 byte alignment of b */
alignas(64) float b[4];
```

- GNU:

```
/* 64 byte alignment of b */
float b[4] __attribute__((aligned(64)));
```

- Intel:

```
/* Intel-specific */
__declspec(align(64)) float b[4];
```

Aligned allocations on the heap

- Straightforward allocation

```
double *foo = malloc(10 * sizeof(double));
int *bar = malloc(4 * sizeof(int));
```

- POSIX (Linux, Mac, BSD):

```
#include <stdlib.h>

double *a = NULL;
/* Allocate space for 100 doubles
 * aligned to 64 byte boundary */
posix_memalign(&a, 64,
               100*sizeof(double));
```

- Windows

```
#include <malloc.h>
double *a = NULL;
a = _aligned_malloc(100*sizeof(double),
                    64);
```

- Intel:

```
double *a = NULL;
/* Intel only */
a = _mm_malloc(100*sizeof(double), 64);
```

Instruct compiler about alignment

Durham University
Department of Computer Science

- Having controlled the allocation of variables to be appropriately aligned
- Also need to inform compiler *at point of use*
- Use (compiler-specific) builtins to provide information

```
void foo (float * a, ...) {
   /* a is aligned to a 64
    * byte boundary */
   __assume_aligned(a, 64);
   ...
}
```

- Intel's solution:

```
#pragma vector aligned
for (i = 0; i < n; i++)
  X[i] += a[i] + a[i+n1] + a[i-n1]+ a[i+n2] + a[i-n2];
```

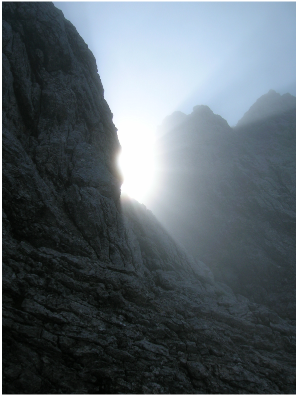
Concept of building block

Durham University
Department of Computer Science

- Content
  - Aligned and non-aligned loads and stores
  - Create aligned data structures
  - Make compiler exploit alignment
  - Padding
- Expected Learning Outcomes
  - The student can explain aligned/unaligned loads/stores and implications
  - The student can use aligned data structures (with Google)
  - The student can explain and use padding

Outline

Durham University
Department of Computer Science



Alignment
Padding and AoS vs. SoA

Observations

Durham University
Department of Computer Science

- Codes run best when they use streams that are properly aligned
- Gains importance for ARM chips
- Very important for GPGPUs

Image blur/finite differences

Durham University
Department of Computer Science

```
for (int x=1; x<N-1; x++)
for (int y=1; y<N-1; y++) {
  b[x,y] = -1.0 * a[x-1,y] - 1.0 * a[x+1,y]
           -1.0 * a[x,y-1] - 1.0 * a[x,y+1]
           +4.0 * a[x,y];
}
```

- C maps multidimensional arrays into one long array
- Alignment of `a` makes `a[0][i]` accesses aligned
- Alignment of `a` does not align `a[j][i]` accesses

Padding

Durham University
Department of Computer Science

> **Padding:** Insert additional byte into an array to ensure that all accesses are aligned.

Pros and cons

Durham University
Department of Computer Science

- Better performance
- Higher memory footprint
- Machine-specific
- One padding for one program phase might be the wrong one for the other phase (transpose challenge)

AoS vs. SoA

Durham University
Department of Computer Science

NVidia: Use struct of arrays, not array of structs for data layout.

Array of Structs (AoS)

```
struct Point {
  double x, y, z;
};

struct Point *points = ...;
```

Struct of Arrays (SoA)

```
struct Points {
  double *x, *y, *z;
};

struct Points points = ...;
```

Pros and cons

Durham University
Department of Computer Science

AoS:

- Proper code design/logical view (each record has all its fields together)
- Good cache usage when modifying struct
- “Easy” to insert/remove structs from array
- “Easy” to send out particular structs (cmp. MPI session)
- Alignment tricky
- Vectorisation tricky

SoA:

- Data structure does not represent physical concept 1:1
- Modifying individual struct introduces cache misses
- Can’t remove/insert particular structs straightforwardly
- Alignment great
- Vectorisation efficient

Tiling

Durham University
Department of Computer Science

```
for (int x=1; x<N-1; x++)
for (int y=1; y<N-1; y++) {
  b[x,y] = -1.0 * a[x-1,y] - 1.0 * a[x+1,y]
           -1.0 * a[x,y-1] - 1.0 * a[x,y+1]
           +4.0 * a[x,y];
}
```

- When we compute `b[1,1]=b[1+N]`, we load `a[1,2]=b[1+2*N]`
- When we compute `b[1,2]`, we need this value once more
- Very likely removed from cache for reasonably big `N`

Tiling

Durham University
Department of Computer Science

> Tiling: Reorder multidimensional traversal such that cache misses are reduced.

- Makes code more complicated
- Machine-specific
- To be combined with padding
- Yields massive speed-ups

Concept of building block

Durham University
Department of Computer Science

- Content
  - Padding
  - AoS
  - SoA
  - Tiling
- Expected Learning Outcomes
  - The student can explain ideas/rationale behind all techniques
  - The student can write codes employing the ideas