SESSION 8

VECTORISATION & DATA LAYOUT

MASSIMILIANO FASI

Durham University

Scalar and vector operations

$$z \leftarrow x + y$$

Scalar operation

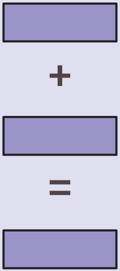
$$\begin{bmatrix} z_0 \\ \vdots \\ z_n \end{bmatrix} \leftarrow \begin{bmatrix} x_0 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix}$$

Vector operation

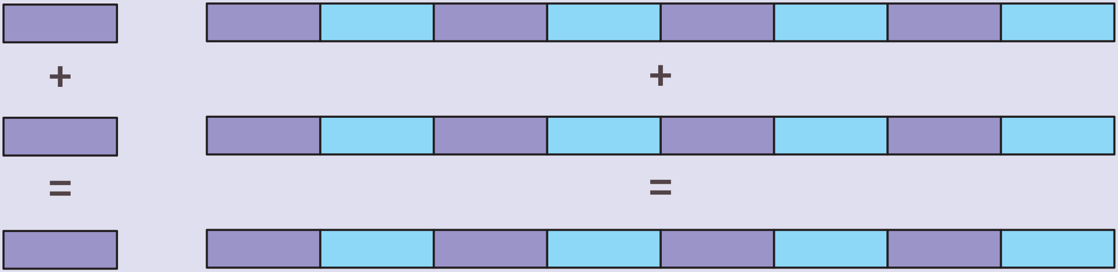
Two realizations

- lockstepping (GPUs SIMT)
- large vector registers (x86 extensions)

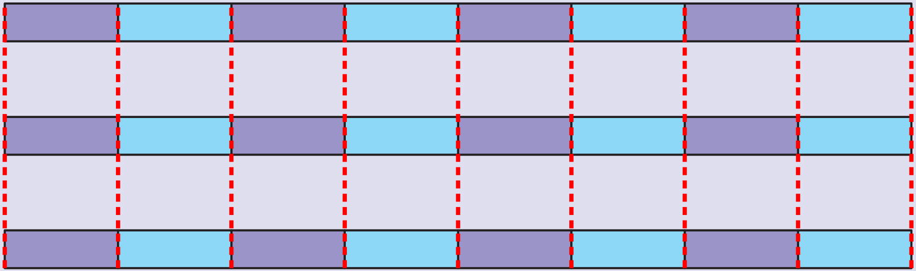
Large vector registers



Large vector registers

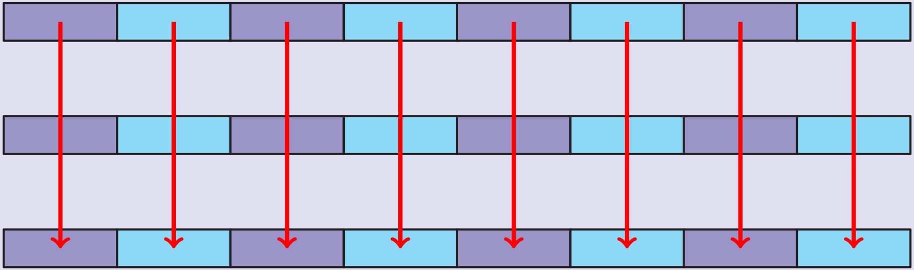


Large vector registers



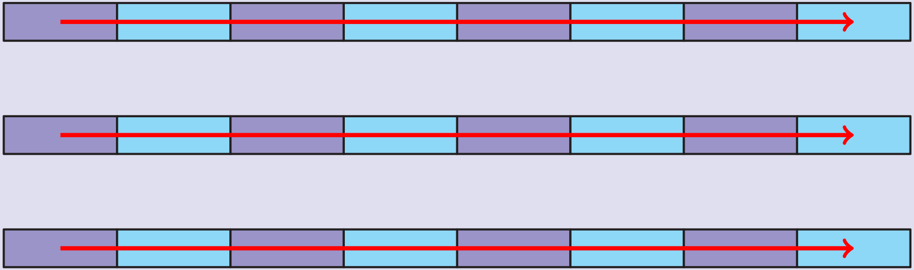
- SIMD lanes

Large vector registers



- SIMD lanes
- vertical operation

Large vector registers



- SIMD lanes
- vertical operation
- horizontal operation

Vector extensions

Arch.	Extension	bits	binary32	binary64
x86	SSE	128	4	–
x86	SSE2	128	4	2
x86	AVX	256	8	4
x86	AVX2 (FMA)	256	8	4
x86	AVX512	512	16	8
ARM	SVE	128–2048	4–64	2–32

- **SSE** Streaming SIMD Extension
- **AVX** Advanced Vector eXtension
- **SVE** Scalable Vector Extension

Compiler x86 options

Architectures

- `-march=x86-64`
- `-march=core-avx2`
- `-march=skylake-avx512`
- `-march=znver2`
- `-march=native`

Extensions

- `-mmmx`
- `-msse`
- `-msse4.2`
- `-mavx2`
- `-mavx512f` (Foundation)

The GCC flag `--help=target` shows **all** target-specific options.

Vectorising C and C++ code

Vectorisation in practice

1. Automatic optimisation
   - `g++ -fopt-info` ?
   - `icpc -qopt-report` ?
2. Compiler loop-specific `#pragma` directives[1]
3. OpenMP[2] vectorisation `#pragma` directives
4. Compiler built-in (intrinsic) functions ?
5. Hand-written assembly

---

[1] Pragmas are used to give additional information to the compiler.
[2] Open Multi-Processing.

Unrolling a for loop

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

Unrolling a for loop

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

```
for (int i = 0; i < 4 * (N / 4); i += 4) {
    a[i]   = b[i]   + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
    a[i+3] = b[i+3] + c[i+3];
}
for (; i < N; ++i)
    a[i]   = b[i]   + c[i];
```

Requirements for automatic vectorisation

1. iteration count known beforehand
2. no jumps (`break`/`continue`)
3. no exceptions
4. no loop carried dependency
5. no nested loops
6. no if statements (almost)
7. no function calls (almost)

This requires `-ftree-vectorize` (included in `-O3`).

Will the compiler vectorise this?

```
double A* = (double *) malloc(N * N * sizeof *A);
double B* = (double *) malloc(N * N * sizeof *B);
double C* = (double *) malloc(N * N * sizeof *C);
```

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

Will the compiler vectorise this?

```
double A* = (double *) malloc(N * N * sizeof *A);
double B* = (double *) malloc(N * N * sizeof *B);
double C* = (double *) malloc(N * N * sizeof *C);
```

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

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

Will the compiler vectorise this?

```
double A* = (double *) malloc(N * sizeof *A);
```

```
for (int i = 0; i != N; ) {
    tmp = N;
    N = A[i];
    A[i] = tmp;
}
```

Will the compiler vectorise this?

```
double A* = (double *) malloc(N * sizeof *A);
```

```
for (int i = 0; i != N; ) {
    tmp = N;
    N = A[i];
    A[i] = tmp;
}
```

```
for (int i = 0; i <= N; i++) {
    if (A[i] > 0)
        sum += A[i];
}
```

Will the compiler vectorise this?

```c
double A* = (double *) malloc(N * sizeof *A);
```

```c
for (int i = 2; i < N; i++)
    A[i] = (A[i-1] + A[i-2]) / 2;
```

Will the compiler vectorise this?

```
double A* = (double *) malloc(N * sizeof *A);
```

```
for (int i = 2; i < N; i++)
    A[i] = (A[i-1] + A[i-2]) / 2;
```

```
void foo(double *A, double *B, double *C, int N)
{
    for (int i = 0; i < N; i++)
        A[i] = (B[i] + C[i]) / 2;
}
```

What are `B` and `C`?

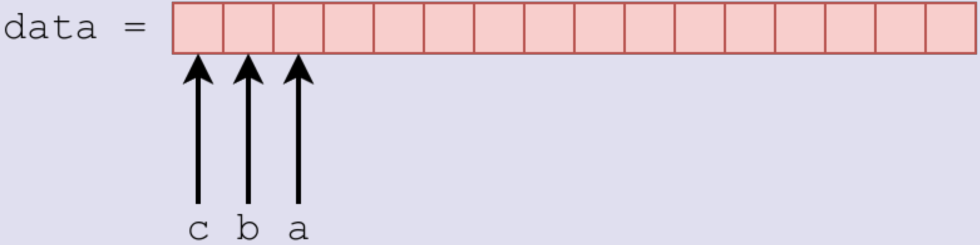
Pointer aliasing (C/C++)

```c
void foo(double *A, double *B, double *C, int N)
{
    for (int i = 0; i < N; i++)
        A[i] = (B[i] + C[i]) / 2;
}
```

```c
void bar(double *c, int n)
{
    double *a = c + 2;
    double *b = c + 1;
    foo(a, b, c, n-3);
}
```

Pointer aliasing (C/C++)

```
void bar(double *c, int n)
{
    double *a = c + 2;
    double *b = c + 1;
    foo(a, b, c, n-3);
}
```



Solutions to pointer aliasing

1. Some compilers can handle it on their own
   - Multiple versions of the loop are generated
   - Run-time check for aliasing
   - Appropriate version is used

Solutions to pointer aliasing

1. Some compilers can handle it on their own
   - Multiple versions of the loop are generated
   - Run-time check for aliasing
   - Appropriate version is used
2. Tell the compiler with `-fno-alias`

Solutions to pointer aliasing

1. Some compilers can handle it on their own
   - Multiple versions of the loop are generated
   - Run-time check for aliasing
   - Appropriate version is used
2. Tell the compiler with `-fno-alias`
3. We can guarantee pointers will not alias
   - `double * restrict` (in C99 or newer)
   - `double * __restrict__` (in C++)

Compiler loop-specific `#pragma` directives

```
#pragma <directive>\n
<for_loop>
```

Ignore vector dependencies

- g++ : `#pragma GCC ivdep`
- icpc : `#pragma ivdep` (GCC ignored)

Force loop unrolling factor

- g++ : `#pragma GCC unroll(<factor>)`
- icpc : `#pragma unroll(<factor>)`

OpenMP vectorisation `#pragma` directives

```
#pragma omp simd [<clause>[[,]<clause>]]...]\n
<for_loop>
```

For **vertical** operations, `<clause>` can be

- `safelen(<length>)`: unrolling factor safe to use.
- `simdlen(<length>)`: number of SIMD lanes to use.
- `linear(<list>[:<step>])`: step for variables in `<list>`.
- `if([simd :] <expr>)`: vectorise only if `<expr>` is true.
- `collapse(<num>)`: collapse `<num>` levels of nested loops.

Reduction

For **horizontal** operations, `<clause>` can be

```
reduction([<modifier>,]<identifier>:<list>)
```

where `<identifier>` can be

- an arithmetic operation: `+`, `*`, `-`, `max`, `min`
- a logical or bitwise operation: `&`, `&&`, `|`, `||`, `^`

and `<list>` is a list of variables. For `<modifier>`s, see ?.

Vectorised functions

```
#pragma omp declare simd [<clause>[[,]<clause>]...]\n
[#pragma omp declare simd [<clause>[[,]<clause>]\n]
[...]
<function_definition_or_declaration>
```

- Generates multiple (vectorised) versions of the function.
- **However**, compilers will often inline, then vectorise.
- Use `-fno-inline` (g++) or `-qno-inline` (icpc) to check.

Some examples

- Ignoring vector dependencies ⚙ $\Rightarrow$ ⚙
- Safe forward dependencies ⚙ $\Rightarrow$ ⚙ $\Rightarrow$ ⚙
- Reduction in inner product ⚙ $\Rightarrow$ ⚙
- Declare SIMD function ⚙ $\Rightarrow$ ⚙

Some examples

- Ignoring vector dependencies ⚙ $\Rightarrow$ ⚙
- Safe forward dependencies ⚙ $\Rightarrow$ ⚙ $\Rightarrow$ ⚙
- Reduction in inner product ⚙ $\Rightarrow$ ⚙
- Declare SIMD function ⚙ $\Rightarrow$ ⚙

USEFUL REMINDERS

- Don't forget `-fopenmp` (`g++`) or `-qopenmp` (`icpc`)!
- `-fopt-info` (`g++`) and `-qopt-report` (`icpc`) can help.
- Memory **must be contiguous** ⚙ $\Rightarrow$ ⚙