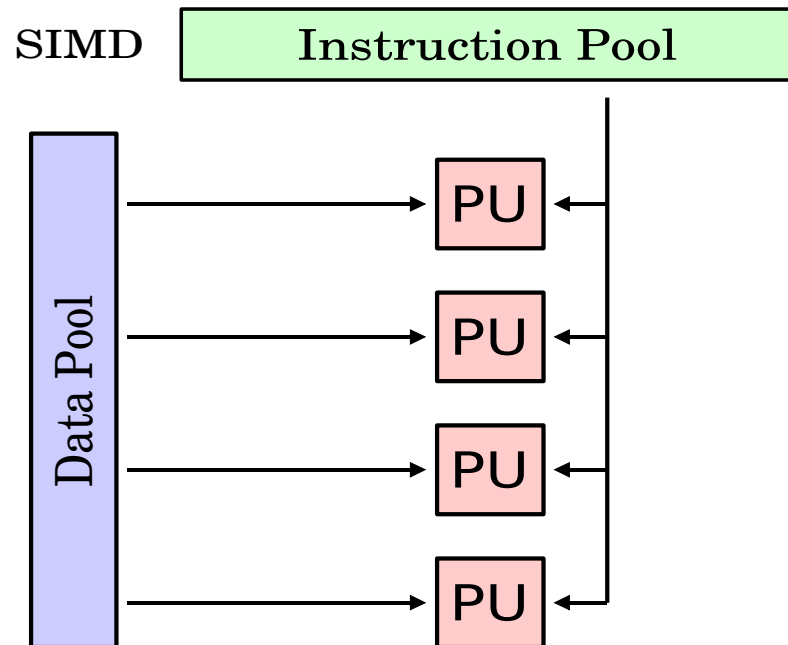


Flynn's taxonomy

- SISD: Single Instruction Single Data
- **SIMD: Single Instruction Multiple Data**
- MISD: Multiple Instruction Single Data
- MIMD: Multiple Instruction Multiple Data

SIMD

- In scalar code, instructions operate on one datum
- In vector code, instructions operate on multiple data (using vector instructions and registers)



SIMD implementations

- Most modern processors support vector instructions:
 - X86: SSE, AVX, ...
 - PowerPC: AltiVec
 - ARM: Neon

Why?

- Vector instructions allow fast implementations of some algorithms, such as:
 - Image and video algorithms (filters, codecs, ...)
 - Cryptography
 - String manipulation

Disadvantages

- Some problems are hard/impossible to vectorize
- Hard to program and debug (as you will see soon enough...)
- Architecture and processor dependent

Streaming SIMD Extensions

- Set of vector instructions and registers available on “recent” x86 processors
 - Several versions: SSE, SSE2, SSE3, Supplemental SSE3 (SSSE3), SSE4, ...
- Note: **ALL** x86_64 processors support at least SSE2

Streaming SIMD Extensions

- 128bits vector registers (also called “XMM registers”)
- ...as well as instructions to operate on them

Using SSE

- Several ways:
 - Compilers can (try to) do it for you, or
 - Use assembly SSE instructions directly, or
 - Use the “wrappers” provided by the compiler (“intrinsics”)

Compiler intrinsics

- Different vector types:
 - `__m128i` (integral types)
 - `__m128` (single precision floats)
 - `__m128d` (double precision floats)
- SSE instructions wrappers.
 - Format: `_mm_<op>_<type>`
 - Example:
`_mm_add_ps(__m128 a, __m128 b)`

Compiler intrinsics

- Include the necessary headers (Example: `<emmintrin.h>` for SSE2)
- And the necessary compiler flags (Example: `-msse2` for SSE2)

Load, set & store operations

- SSE programming requires explicit loads and stores
 - Use intrinsics to load/store 128 bits at once.
Example:

```
_mm_loadu_si128();  
_mm_store_pd();
```

- Beware the data alignment!

Load, set & store operations

- Set operations: Used to load a value into a XMM register. Example:

```
_mm_set1_epi32(10);
```

→ Load the constant '10' four times in a vector register

Load, set & store operations

- 3 classes of load & store operations
 - Unaligned: No alignment requirement, but slower
 - Aligned: Faster, but requires 16-byte alignment
 - Streaming: Avoid cache pollution. Requires 16-byte alignment
- Runtime error if alignment not respected
- Note: Streaming load requires SSE4

Data alignment

- Dynamic allocation:
 - `_mm_malloc()/_mm_free()` (Compiler intrinsics)
 - `posix_memalign()` (POSIX only)
 - `aligned_alloc()` (introduced in C11)
- Static allocation: Compiler attributes

Loop unrolling

- First step to vectorize a loop
- Type int is 32-bits. XMM registers are 128-bits wide
 - each XMM registers will hold 4 int's
 - Each loop iteration will operate on 4 int's at a time
- Question: What about char's? Double's?

Loop unrolling

```
int array[4096];  
for (int i = 0; i < 4096; ++i)  
    array[i] = 10;
```

Becomes...

```
for (int i = 0; i < 4096; i += 4)  
{  
    array[i] = 10;  
    array[i + 1] = 10;  
    array[i + 2] = 10;  
    array[i + 3] = 10;  
}
```


Demo

- Add 42 to every element in an int array
- Works as follow:
 - Load the constant '42' four times in a XMM register
 - Unroll the loop four times
 - For each iteration:
 - Load 128 bits from the array in a XMM register
 - Perform 4 additions at a time
 - Store the result back into the array

Your tasks

- You will implement with SSE:
 - Convert a string to lower case
 - Matrix-vector multiplication
 - Matrix-matrix multiplication
- Bonus: Blocked matrix-matrix multiplication