CS60203: Design Optimization of Computing Systems *August 25, 2024*

SIMD-part-1

Instructor: Mainack Mondal *Scribe-By: Parthiv Reddy Anuguru*

1 Why parallelize ?

- Simply because it increases CPU performance
- Let's say P fraction of code can be parallelized and S be the fraction that can't be parallelized, If there are n processors then speed up form **amdhal's law** is given by :

$$\text{Speed up} = \frac{1}{\frac{P}{n} + S} \tag{1}$$

$$\text{Speed up} = \frac{1}{\frac{P}{n} + 1 - P} = \frac{1}{1 - P(1 - \frac{1}{n})} \tag{2}$$

- As the number of processing units(n) increases, a system that maintains substantial performance gains demonstrates **higher scalability**.

2 What exactly is parallelized ?

- We will be talking about different types of parallelism based on what are we parallelizing in a task and how are we doing that ?

2.1 Software approach

- Here we talk about software Ideas to parallelize a task

2.1.1 Task level parallelism

- Simply it is **Different** operations occur in parallel.
- Multiple threads / instruction sequences from same application can be executed concurrently
- These threads are inherently parallel i.e, run independent operations across different threads
- This approach will work across mutliple processes

Cilk:

- Parallel programming is hard because of task partitioning and synchronization, which demanded a more abstract programming language

- Extends C language with few key words, can run without rewriting on any number of processors here compiler and runtime system will schedule the task to run on the given platform

OpenMP:

- Offers compiler side directives and API for parallel programming
- If the compiler does not recognize OpenMP directives, the code remains functional (though single-threaded)
- Offers wider support from more vendors and more control to developer regarding parallelizing

2.1.2 Instruction level parallelism (ILP)

- Multiple instructions from same instruction stream can be executed concurrently
- Works within single process, basically pipelining
- Reordering, out of order execution, branch prediction are some approaches to make the pipelining efficient

2.2 Hardware approach

Here changes are made to the architecture of the computer for exploiting parallelism of a task, like registers, instruction set.

2.2.1 SIMD : Single Instruction Multiple Data streams

- Same instruction is executed in multiple processing units with different data
  SIMD processing:
  For this SIMD uses specialized vector registers. Also SIMD significantly speeds up when there is data alignment (architecture details)
  - **Array processor**: Instruction operates on multiple data elements at the same time
  - **Vector processor**: Instruction operates on multiple data elements in consecutive time steps
- In VLIW multiple instructions are packed together in instruction memory and each of them is sent to different processing units, this way it is different from SIMD
- SIMD exploits ILP as part of vector processing.

3 How to use SIMD ?

3.1 Some note worthy points before start:

- Compiler aggressively tries to auto vectorise the instructions when -O3 flag is used
- Compiler can't distinguish between the data types of SIMD they are just for type checking
- Compile your SIMD program with flags -mavx or -mavx2

3.2 What changed with time ?

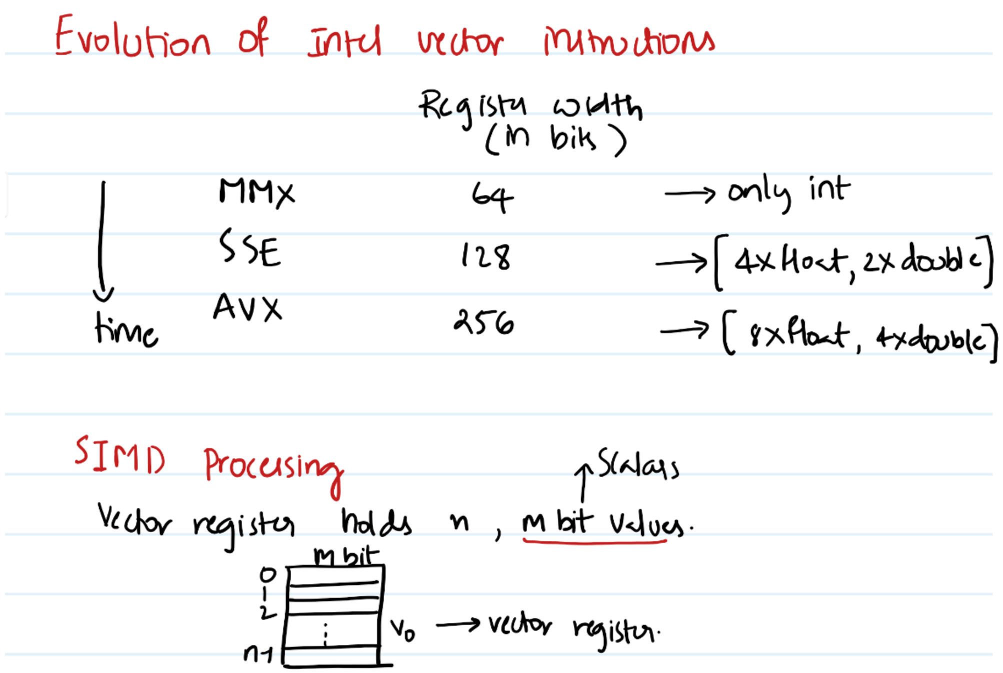

Figure 1: Evolution of Intel vector instructions and SIMD vector register

Here SSE3 has 16, 128b xmm registers and AVX has 16, 256b ymm registers numbered 0 to 15, %xmm ⟨*number*⟩ key word in assembly of SIMD is nothing but SSE register

3.3 SSE3 Registers

These are 128 bit registers and shown in Figure 1

- **Different data types and associated instructions**
- **Integer vectors**
  - 16-way byte
  - 8-way 2 bytes
  - 4-way 4 bytes
  - 2-way 8 bytes
- **Floating point vectors**
  All the 128b are used and all the partitions as per data types are affected in a SIMD operation
  - 4-way single (since SSE)
  - 2-way double (since SSE2)
- **Floating point scalars**
  Here not all 128 bits are used only right most 32b are used in case of single, and 64b are used in case of double scalars. The remaining will be unaffected in any vectorized operations.

- single (since SSE)
- double (since SSE2)

3.4 Data types and packing

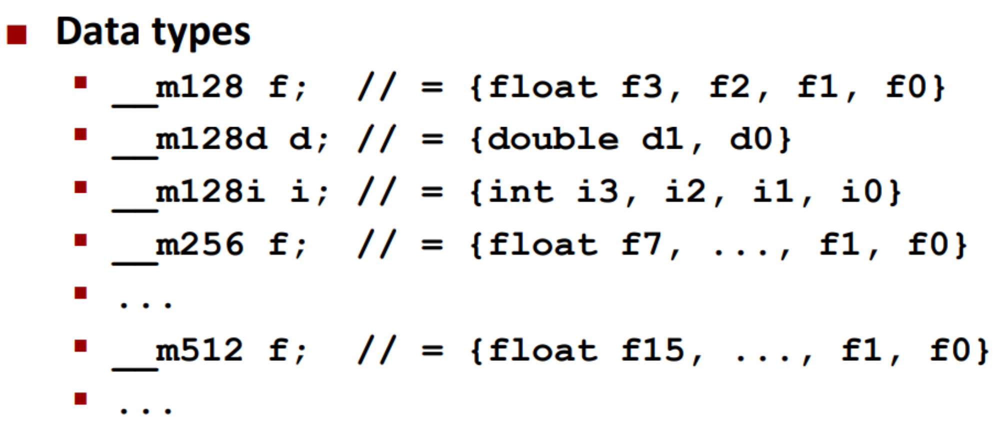

Figure 2: Data types

Packed represents vector and single slot represents scalar, single precision means floats and double precision means doubles

decoding addps: add=$\rangle operation, p => packed, s => singleprecision$
decoding addss : $add => operation, s => scalar, s => singleprecision$
decoding vaddpd : $v => 3operands, add => operation, p => packed, d => doubleprecision$

3.5 Instructions

SIMD instructions are of the form _mm_⟨operation⟩_⟨suffix⟩, suffix talks about packing and precision

3.5.1 Load, Set and Store

Intrinsic Name	Operation	Corresponding SS
_mm256_broadcast_pd	Broadcast 128 bits from memory to all elements of dst	vbroadcastf128
_mm_i32gather_epi32	Gather 32-bit integers from memory using 32-bit indices	vpgatherdd
_mm_load_ss	Load the low value and clear the three high values	movss
_mm256_loadu2_m128i	Load two 128-bit values from memory, and combine them into dst	composite
_mm256_load_ps	Load eight values, address aligned	vmovaps
_mm_loadu_ps	Load four values, address unaligned	movups
_mm_maskload_pd	Load packed double-precision elements from memory using mask	vmaskmovpd

Table 1: Load Instructions

Intrinsic Name	Operation	Corresponding SSE/AVX Instructio
_mm_set_ss	Set the low value and clear the three high values	Composite
_mm_set1_ps	Set all four words with the same value	Composite
_mm_set_ps	Set four values	Composite
_mm_setr_ps	Set four values, in reverse order	Composite
_mm_setzero_ps	Clear all four values	Composite
_mm256_set1_pd	Set all four words with the same value	Composite
_mm256_set_ps	Set eight values	Composite
_mm256_set1_epi64x	Broadcast 64-bit integer a to all elements of dst	Composite
_mm256i_setzero_si256	Clear all 256 bits	Composite

Table 2: Set Instructions

Intrinsic Name	Operation	Corresponding S
_mm256_store_pd	Store 4 doubles to aligned memory	vmovapd
_mm_store_pd	Store 2 doubles to aligned memory	movapd
_mm256_maskstore_ps	Store single-precision elements from a into memory using mask	vmaskmovps
_mm256_stream_si256	Non-temporal store	vmovntdq
_mm256_storeu2_m128d	Store the high and low 128-bit into memory two different locations	composite
_mm_storel_epi64	Store 64 bit of XMM register to memory	movq
_mm_store1_pd	Store lowest double to memory	composite

Table 3: Store Instructions

3.5.2 Arithmetic

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_add_ss	Addition	ADDSS
_mm_add_ps	Addition	ADDPS
_mm_sub_ss	Subtraction	SUBSS
_mm_sub_ps	Subtraction	SUBPS
_mm_mul_ss	Multiplication	MULSS
_mm_mul_ps	Multiplication	MULPS
_mm_div_ss	Division	DIVSS
_mm_div_ps	Division	DIVPS
_mm_sqrt_ss	Squared Root	SQRTSS
_mm_sqrt_ps	Squared Root	SQRTPS
_mm_rcp_ss	Reciprocal	RCPSS
_mm_rcp_ps	Reciprocal	RCPPS
_mm_rsqrt_ss	Reciprocal Squared Root	RSQRTSS
_mm_rsqrt_ps	Reciprocal Squared Root	RSQRTPS
_mm_min_ss	Computes Minimum	MINSS
_mm_min_ps	Computes Minimum	MINPS
_mm_max_ss	Computes Maximum	MAXSS
_mm_max_ps	Computes Maximum	MAXPS

For implementing a solution in some real problems look at this blog. Also all the **references** links are already provided as hrefs.