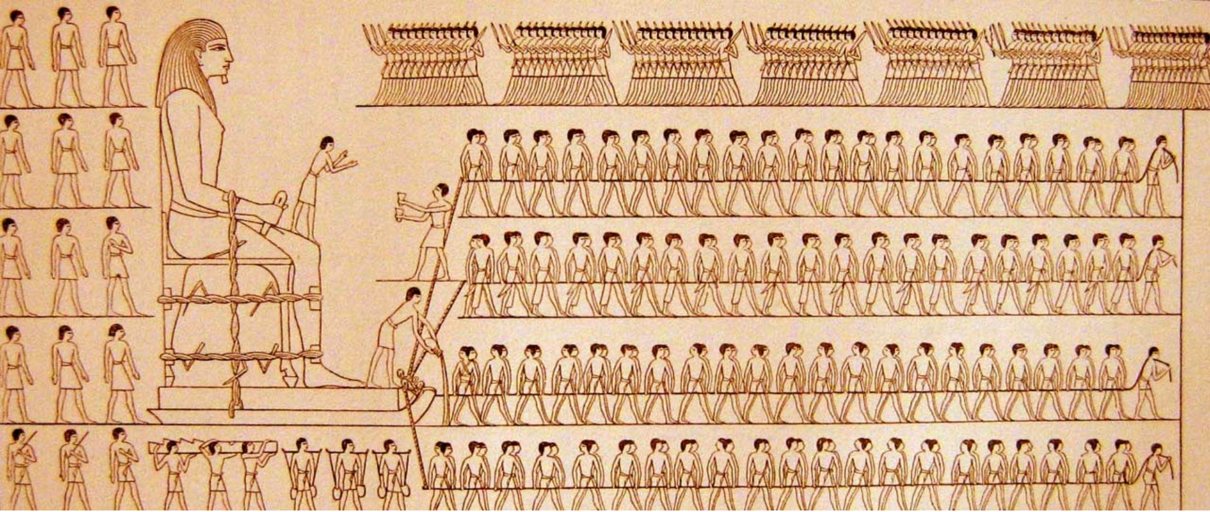
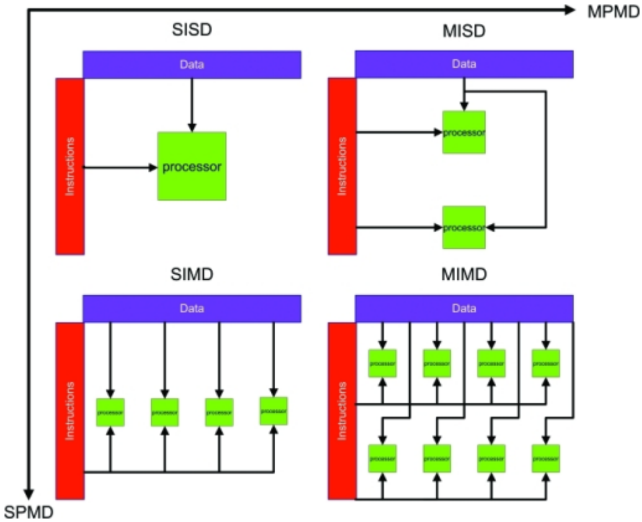


High Performance Co$^{mputing}_{ding}$ (HPC)

Alberto Dassatti - 2023

FLYNN'S TAXONOMY



Single Instruction Multiple Data

What SIMD really means:

- Instruction Set Extension.
- Each producer has its incompatible set(s) (luckily there are similarities).
- Good speed-up, low hardware complexity, but its use is problem dependent.
- Special data types with high parallelism, special instrunctions to process chunks of smaller data in parallel:
  - 64 bit MMX
  - 128 bit SSE2-3-4
  - 256 bit AVX-AVX2
  - 512 bit AVX-512

How SIMD instructions work

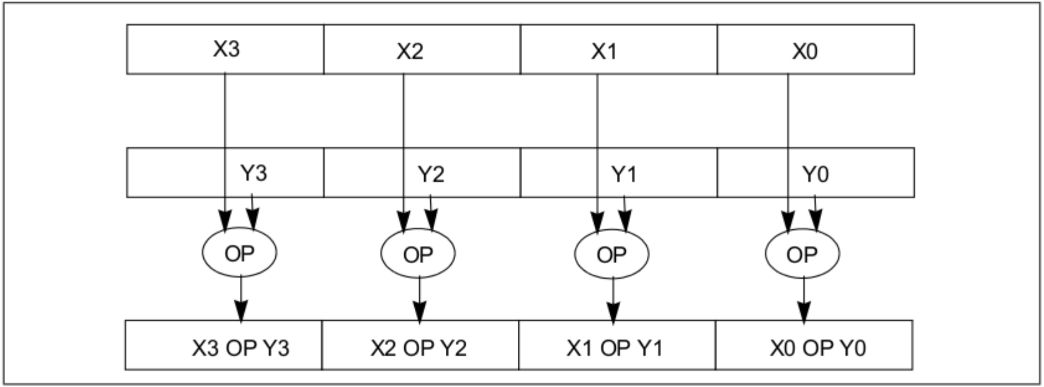


Figure 10-5. Packed Single-Precision Floating-Point Operation

more here.

Data formats in SSE2

128 bit integer

Two 64 bit integers:

64 bit integer	64 bit integer

Four 32 bit integers:

32 bit int.	32 bit int.	32 bit int.	32 bit int.

Eigth 16 bit integers:

16 b.	16 b.	16 b.	16 b.	16 b.	16 b.	16 b.	16 b.

Sixteen 8 bit integers:

8 b	8 b	8 b.	8 b	8 b	8 b.	8 b.	8 b	.8 b.	8 b.	8 b.	8 b	8 b	8 b	8 b	8 b.

`__m128i` (8 bit - 128 bit)

Data formats in SSE2

Two 64 bit floating point numbers:

64 bit floating point	64 bit floating point

double: __m128d (SSE2)

Four 32 bit floating point numbers:

32 bit fl.p.	32 bit fl.p.	32 bit fl.p.	32 bit fl.p.

float: __m128 (SSE)

INSTRUCTIONS

- Memory access (explicit and implicit)
- Basic arithmetic (+, -, *)
- Expensive arithmetic (1/x, sqrt(x), min, max, /, 1/sqrt)
- Logic (and, or, xor, nand)
- Comparison (+, $<$, $>$, $\ldots$)
- Data reorder (shuffling)

Using SIMD

There are several options:

1. Direct or inline Assembly.

USING SIMD

There are several options:

1. Direct or inline Assembly.
2. Compiler provided intrinsic functions.

Using SIMD

There are several options:

1. Direct or inline Assembly.
2. Compiler provided intrinsic functions.
3. Some compilers give you `#pragmas` (`#pragma omp simd` for instance)

Using SIMD

There are several options:

1. Direct or inline Assembly.
2. Compiler provided intrinsic functions.
3. Some compilers give you `#pragmas` (`#pragma omp simd` for instance)
4. Compiler autovectorization (`-ftree-vectorize -msse2 -ftree-vectorizer-verbose=5`).

USING SIMD

There are several options:

1. Direct or inline Assembly.
2. Compiler provided intrinsic functions.
3. Some compilers give you `#pragmas` (`#pragma omp simd` for instance)
4. Compiler autovectorization (`-ftree-vectorize -msse2 -ftree-vectorizer-verbose=5`).
5. Use of a library: usually the best option if it exists; there are many non-trivial optimizations and cosiderations in a specific domain.

Vectorize this function:

```
void loop(float c[], float a[], float b[], int n) {
 int i;

 for (i=0;i<n;i++)
   c[i] = a[i] + b[i];

 }
```

An option:

```
#include <xmmintrin.h>
void vloop(float c[], float a[], float b[], int n)
{
     int i,nb;
     __m128 v1, v2;

     nb = n - (n % 4);
     for (i=0;i<n;i+=4) {
      v1 = _mm_load_ps(&a[i]);
      v2 = _mm_load_ps(&b[i]);
      v2 = _mm_add_ps(v1,v2);
      _mm_store_ps(&c[i],v2);
     }
     for (i=nb;i<n;i++)
      c[i] = a[i] + b[i];
}
```

See it in action here

Vectorize this function:

```
void loop(float c[], float a[], int n) {
 int i;

 for (i=0;i<n;i++)
   c[i] = a[i] + i;

 }
```

An option:

```
#include <xmmintrin.h>
void vloop(float c[], float a[],  int n)
{
     int i,nb;
     __m128 v1;

     nb = n - (n % 4);
     for (i=0;i<n;i+=4) {
       v1 = _mm_load_ps(a+i);
       index = _mm_set_ps(i+3, i+2, i+1, i);
       v1 = _mm_add_ps(v1, index);
       _mm_store_ps(x+i, v1);
     }
     for (i=nb;i<n;i++)
      c[i] = a[i] + i;
}
```

An second option: See them in action here

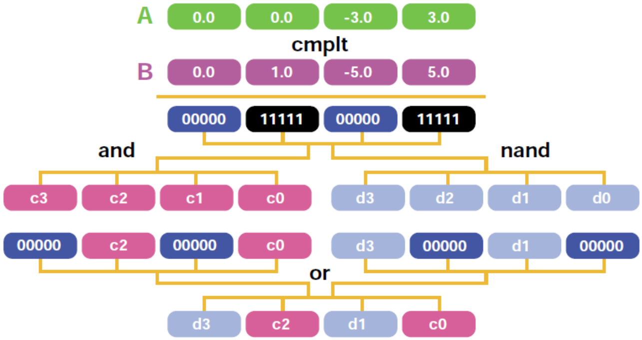
```
#include <xmmintrin.h>
void vloop(float c[], float a[],  int n)
{
     int i,nb;
     __m128 v1, ind, incr;

     ind = _mm_set_ps(3, 2, 1, 0);
     incr = _mm_set1_ps(4);

     nb = n - (n % 4);
     for (i=0;i<n;i+=4) {
       v1 = _mm_load_ps(a+i);
       v1 = _mm_add_ps(v1, ind);
       ind = _mm_add_ps(ind, incr);
       _mm_store_ps(x+i, v1);
     }
     for (i=nb;i<n;i++)
      c[i] = a[i] + i;
}
```

WHAT ABOUT CONDITIONALS?

How can we implement conditionals using SIMD instructions?




```
// R = (A < B)? C : D

F32vec4 mask = cmplt(a, b);
r = (mask & c) | _mm_nand_ps(mask, d);

// OR, using F32vec4 friend function:
r = select_lt(a, b, c, d);
```

What about conditionals?

If still in doubt, read this and this. The second one is more ARM oriented, but the idea is the same.

Sorting is also possible [1, 2], fast, and very instructive.

SIMD are heavily used in Image Processing and Linear algebra. More recently, in ML as well.

GCC

Gcc has some useful flags

```
-msse, -msse2, -msse3, -march=native,
-mfpmath=sse, -mftree-vectorize
```

Always check the produced assembler to check if the vectorization was fine.

- Alisang is a major problem (restrict may help); Compilers can add runtime checks.
- `-fstrict-aliasing`.
- study and help the compiler.
- Mermory alignement can be tricky as well.
- *__attribute__((aligned(8)));*

Vectorization

Disabled by default, regardless of optimization level (but -Ofast).

bash

```
[al@lap ~]# gcc ftree-vectorize -O2
```

SSE by default, for AVX

bash

```
[al@lap ~]# gcc -mavx -march=corei7-avx
```

Vectorization

for a vectorization report

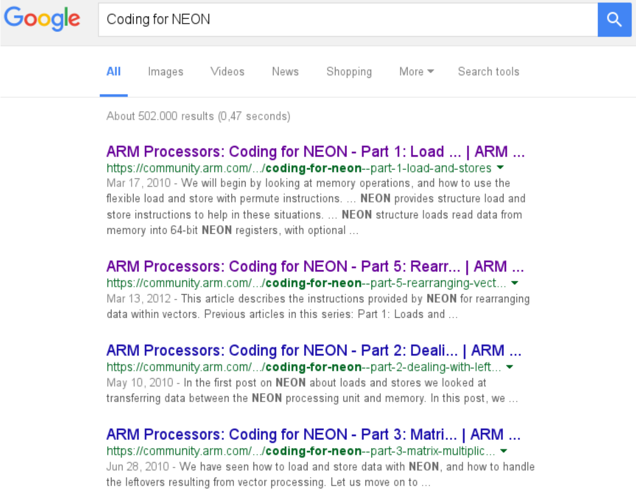
bash

```
[al@lap ~]# gcc -ftree-vectorizer-verbose
```

Resources & References

- MS compiler.
- ethz SIMD class.
- Intrinsic form Intel.
- Tutorial sse.
- Intro Neon.
- ethz on ARM NEON optimizations.
- SSE explained graphically.

TUTORIALS



Coding for NEON

All Images Videos News Shopping More Search tools

About 502.000 results (0,47 seconds)

ARM Processors: Coding for NEON - Part 1: Load ... | ARM ...
https://community.arm.com/.../coding-for-neon--part-1-load-and-stores
Mar 17, 2010 - We will begin by looking at memory operations, and how to use the flexible load and store with permute instructions. ... **NEON** provides structure load and store instructions to help in these situations. ... **NEON** structure loads read data from memory into 64-bit **NEON** registers, with optional ...

ARM Processors: Coding for NEON - Part 5: Rearr... | ARM ...
https://community.arm.com/.../coding-for-neon--part-5-rearranging-vect...
Mar 13, 2012 - This article describes the instructions provided by **NEON** for rearranging data within vectors. Previous articles in this series: Part 1: Loads and ...

ARM Processors: Coding for NEON - Part 2: Deali... | ARM ...
https://community.arm.com/.../coding-for-neon--part-2-dealing-with-left...
May 10, 2010 - In the first post on **NEON** about loads and stores we looked at transferring data between the **NEON** processing unit and memory. In this post, we ...

ARM Processors: Coding for NEON - Part 3: Matri... | ARM ...
https://community.arm.com/.../coding-for-neon--part-3-matrix-multiplic...
Jun 28, 2010 - We have seen how to load and store data with **NEON**, and how to handle the leftovers resulting from vector processing. Let us move on to ...

source

Questions

?