# Lecture 8

# Vector Processing

## 8.1 Flynn's taxonomy

M. J. Flynn proposed a categorization of parellel computer systems in 1966.

- Single Instruction, Single Data stream (SISD)
- Single Instruction, Multiple Data stream (SIMD)
- Multiple Instruction, Single Data stream (MISD)
- Multiple Instruction, Multiple Data stream (MIMD)

## 8.2 SIMD

SIMD focuses on performing one operation on multiple values, for example, adding two vectors,

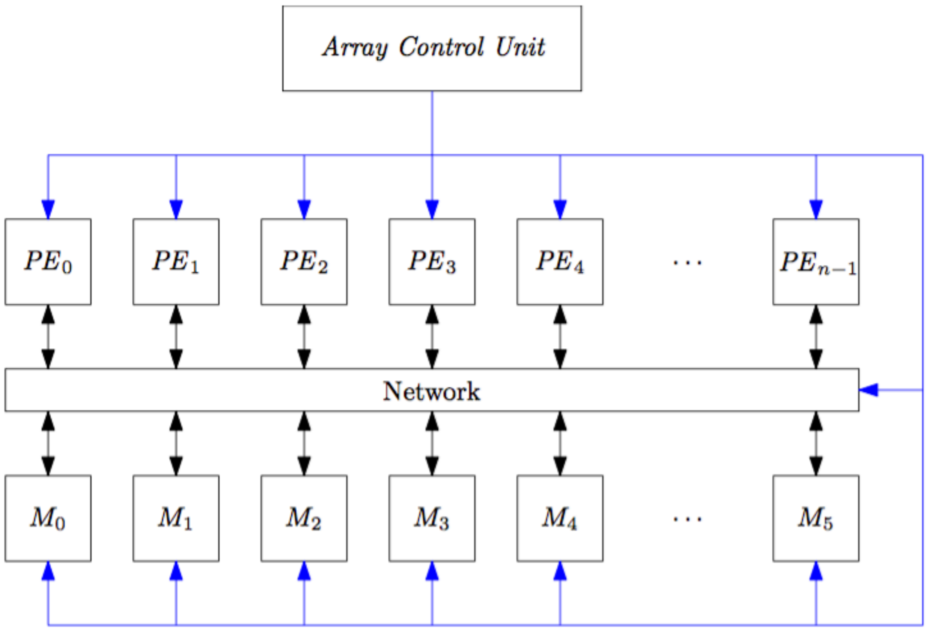
$$\begin{bmatrix}2\\4\\8\\1\\5\\7\\9\\3\end{bmatrix} + \begin{bmatrix}1\\9\\3\\2\\6\\7\\8\\5\end{bmatrix} =?$$

This vector addition can be simply done in a SISD computer system by using repetitions.

```
#include<stdio.h>
#define N 8

int main() {
    float a[] = {2, 4, 8, 1, 5, 7, 9, 3};
    float b[] = {1, 9, 3, 2, 6, 7, 8, 5};
    float c[N];
    int i;

    for(i=0; i<N; i++) {
        c[i] = a[i] + b[i];
    }

    for(i=0; i<N; i++) {
        printf("a[%d]+b[%d] = %4.1f\n", i, i, c[i]);
    }

    return 0;
}
```

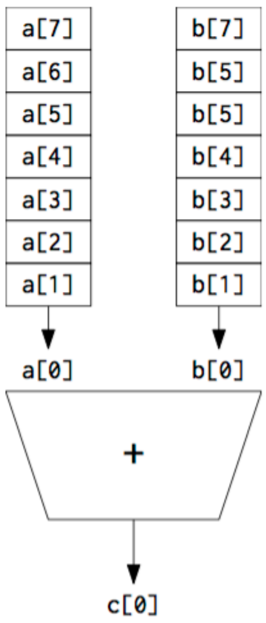
### 8.2.1 Array Architecture

*Array architecture* improves the performance by using multiple *processing elements*. These PEs are controlled by one *array control unit*.

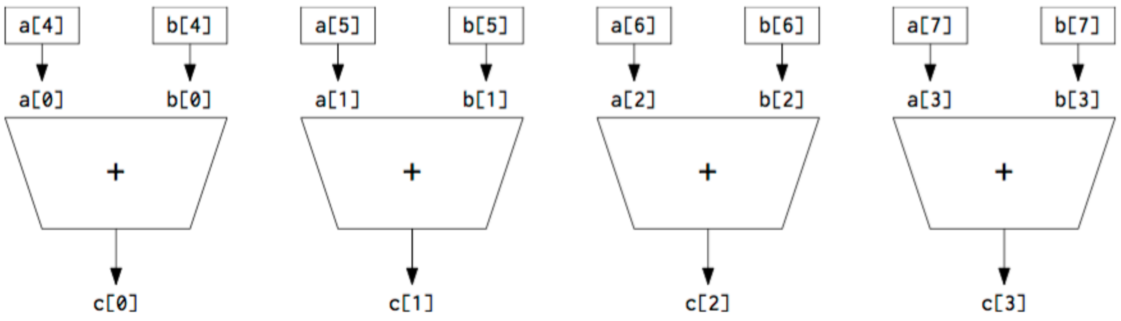


### 8.2.2 Vector Architecture

*Vector Architecture* proposed by Seymour Cray in 1970s. It uses one or more *pipelined execution units* to conduct the vector operations.



The performance can be improved by adding more *vector lanes*.



This vector architecture is designed to be flexible so that any number of vector elements can be processed.

**Exercise 8.1** A program running on a microprocessor spends 20% of its execution time on scalar operations and 80% on vector operations. A vector processor provides an average speed up of 4 for the vector operations. However, it slows down the scalar parts by a factor of 2.

1. What is the expected speedup of the vector processor over the conventional microprocessor?

2. What mix of scalar and vector operation times would lead to no performance improvement on the vector processor?

### 8.2.3 Multimedia Extensions

Intel has provided a set of instructions for vector processing in form of extension to a SISD processor. It is originally called *MMX (MultiMedia eXtension)*. It is then extended to *SSE (Streaming SIMD Extension)*, *SSE2*, *SSE3*, *SSE4*. It then becomes *AVX (Advanced Vector eXtensions)* and *AVX2*.

Different from the vector architecture, the mulmedia extensions provide a number of 256-bit registers called *YMM*. Each YMM register can be packed into vectors of different element types and sizes, i.e.

- Eight 32-bit single-precision floating point numbers (**`float`**)
- Four 64-bit double-precision floating point numbers (**`double`**)

## C Functions for AVX

Let $\mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$,

| Instruction/Function | Description |
|---|---|
| **`_mm256_load_pd(A)`** | loads an array **`A`** into a YMM register, and returns the value of the register |
| **`_mm256_store_pd(A, b)`** | stores the value of a register (**`b`**) into an array **`A`** |
| **`_mm256_add_pd(a, b)`** | returns $\begin{bmatrix} a_0 + b_0 \\ a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{bmatrix}$ |
| **`_mm256_addsub_pd(a, b)`** | returns $\begin{bmatrix} a_0 + b_0 \\ a_1 - b_1 \\ a_2 + b_2 \\ a_3 - b_3 \end{bmatrix}$ |

| Instruction/Function | Description |
|---|---|
| `_mm256_hadd_pd(a, b)` | returns $\begin{bmatrix} a_0 + a_1 \\ b_0 + b_1 \\ a_2 + a_3 \\ b_2 + b_3 \end{bmatrix}$ |
| `_mm256_sub_pd(a, b)` | returns $\begin{bmatrix} a_0 - b_0 \\ a_1 - b_1 \\ a_2 - b_2 \\ a_3 - b_3 \end{bmatrix}$ |
| `_mm256_hsub_pd(a, b)` | returns $\begin{bmatrix} a_0 - a_1 \\ b_0 - b_1 \\ a_2 - a_3 \\ b_2 - b_3 \end{bmatrix}$ |
| `_mm256_mul_pd(a, b)` | returns $\begin{bmatrix} a_0 b_0 \\ a_1 b_1 \\ a_2 b_2 \\ a_3 b_3 \end{bmatrix}$ |
| `_mm256_div_pd(a, b)` | returns $\begin{bmatrix} a_0/b_0 \\ a_1/b_1 \\ a_2/b_2 \\ a_3/b_3 \end{bmatrix}$ |
| `_mm256_sqrt_pd(a)` | returns $\begin{bmatrix} \sqrt{a_0} \\ \sqrt{a_1} \\ \sqrt{a_2} \\ \sqrt{a_3} \end{bmatrix}$ |
| `_mm256_max_pd(a, b)` | returns $\begin{bmatrix} \max(a_0, b_0) \\ \max(a_1, b_1) \\ \max(a_2, b_2) \\ \max(a_3, b_3) \end{bmatrix}$ |

| Instruction/Function | Description |
|---|---|
| `_mm256_min_pd(a, b)` | returns $\begin{bmatrix}\min(a_0,b_0)\\ \min(a_1,b_1)\\ \min(a_2,b_2)\\ \min(a_3,b_3)\end{bmatrix}$ |
| `_mm256_permute_pd(a, ctrl)` | changes the order of $(a_0,a_1)$ and $(a_2,a_3)$ based on 4-bit `ctrl`: `0` selects the first element of the pair, `1` selects the second element of the pair |
| `_mm256_permute_pd(a, 5)` | returns $\begin{bmatrix}a_1\\ a_0\\ a_3\\ a_2\end{bmatrix}$ |
| `_mm256_permute2f128_pd(a, a, 1)` | returns $\begin{bmatrix}a_2\\ a_3\\ a_0\\ a_1\end{bmatrix}$ |

**Example 8.1** Adding two vectors of 8 `double` values.

```
#include<stdio.h>
#include<stdlib.h>
#include<immintrin.h>
#include<x86intrin.h>

#define N 8
#define ALIGN __attribute__ ((aligned (32)))

int main() {
    double ALIGN arrA[] = {2, 4, 8, 1, 5, 7, 9, 3};
    double ALIGN arrB[] = {1, 9, 3, 2, 6, 7, 8, 5};
    double ALIGN arrC[N];

    __m256d a, b, c;

    int i, r;

    for(r=0; r<N; r+=4) {
        a = _mm256_load_pd(arrA+r);
        b = _mm256_load_pd(arrB+r);
        c = _mm256_add_pd(a, b);

        _mm256_store_pd(arrC, c);

        for(i=0; i<4; i++) {
            printf("a[%d]+b[%d] = %4.1lf\n", i+r, i+r, arrC[i]);
        }
    }

    return 0;

}
```

**Example 8.2** Adding two vectors of 8 `float` values.

```
#include<stdio.h>
#include<stdlib.h>
#include<immintrin.h>
#include<x86intrin.h>

#define N 8
#define ALIGN __attribute__ ((aligned (32)))

int main() {
    float ALIGN arrA[] = {2, 4, 8, 1, 5, 7, 9, 3};
    float ALIGN arrB[] = {1, 9, 3, 2, 6, 7, 8, 5};
    float ALIGN arrC[N];

    __m256d a, b, c;

    int i;

    a = _mm256_load_ps(arrA);
    b = _mm256_load_ps(arrB);

    c = _mm256_add_ps(a, b);

    _mm256_store_ps(arrC, c);

    for(i=0; i<N; i++) {
        printf("a[%d]+b[%d] = %4.1f\n", i, i, arrC[i]);
    }
    return 0;

}
```

**Exercise 8.2** Write a C program using AVX to find the maximum value of an 8-element array of `double [10, 5, 9, 4, 1, 6, 9, 2]`

**Exercise 8.3** Write a C program using AVX to approximate the value of $\pi$ using Euler's formula

$$\sum_{i=1}^{\infty} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots = \frac{\pi^2}{6}$$