Programming Parallel Computers

Jukka Suomela · Aalto University · ppc.cs.aalto.fi

Part 2B: Vector operations

What can you do fast with one machine-language instruction?

- **40 years ago:** addition of 8-bit integers — $111 + 22$
- **20 years ago:** multiplication of floating-point numbers — 11.1111×22.2222
- **Today:** multiplication of ***vectors*** of floating-point numbers

$$\begin{bmatrix} 11.1111 \times 22.2222 \\ 33.3333 \times 44.4444 \\ 55.5555 \times 66.6666 \\ 77.7777 \times 88.8888 \end{bmatrix}$$

Modern CPUs are vector processors

- **Even if your code is only doing scalar operations:**

```
float a = ...
float b = ...
float c = a * b;
```

- **CPU will run your code using its vector unit:**
  - “store a to the first element of vector register 0”
  - “store b to the first element of vector register 1”
  - “multiply the first elements of vector registers 0 and 1”

Modern CPUs are vector processors

- Modern Intel CPUs have two kinds of registers:
  - `%rax`, `%rbx`, ….: *64-bit integers*
  - `%ymm0`, `%ymm1`, ….: *256-bit vectors*
- How compilers typically use these:
  - **integer registers:** memory addresses, array indexing, loop counters, integer arithmetic …
  - **vector registers:** floating point arithmetic
- But you can do much more with vector registers!

“Vector”: 4 × double or 8 × float

- `float` (single-precision floating-point number): ***32 bits***
- `double` (double-precision floating-point number): ***64 bits***
- Vector registers: ***256 bits***
  - enough space for 4 × double
  - enough space for 8 × float
  - enough space for 32 × byte

"Vector": 4 × double or 8 × float

- `float` (single-precision floating-point number): ***32 bits***
- `double` (double-precision floating-point number): ***64 bits***
- Vector registers: ***256 bits***
  - enough space for 4 × double
  - enough space for 8 × float
  - enough space for 32 × byte

This text fits in one register!

Vector operations in CPU

- Example: `vaddps %ymm0, %ymm1, %ymm2`
- Behaves like this:

```
z[0] = x[0] + y[0];
z[1] = x[1] + y[1];
z[2] = x[2] + y[2];
z[3] = x[3] + y[3];
z[4] = x[4] + y[4];
z[5] = x[5] + y[5];
z[6] = x[6] + y[6];
z[7] = x[7] + y[7];
```

```
float x[8] ≈ %ymm0
float y[8] ≈ %ymm1
float z[8] ≈ %ymm2
```

Vector operations in C++

- **Hard way:**
  - use “intrinsic functions”
  - code looks like this: `z = _mm256_add_ps(x, y);`
- **Easy way:**
  - use “vector types”
  - code looks like this: `z = x + y;`

Vector types

GCC syntax for saying that “`float8_t`” = vector of 8 × float:

```
typedef float float8_t
  __attribute__ ((vector_size (8 * sizeof(float))));
```

Just copy & paste it whenever you need it…

Vector types

```
float8_t a, b, c;

a = ...;
b = ...;
c = a + b;
```

≈

```
float a[8], b[8], c[8];

a = ...;
b = ...;
c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
c[2] = a[2] + b[2];
c[3] = a[3] + b[3];
c[4] = a[4] + b[4];
c[5] = a[5] + b[5];
c[6] = a[6] + b[6];
c[7] = a[7] + b[7];
```

Similar behavior, but much more efficient code: one vector addition

Vector types

- You can refer to entire vectors — compiler will generate efficient code in which you do element-wise operations:

```
x = (a + b) * c;
```

- You can mix scalars and vectors:

```
x = 3 * a + 2;
```

- You can also refer to individual vector elements if needed, but don’t expect this to generate efficient code:

```
x[0] = 3 * a[1] + 2;
```

Vector types

- You can imagine that vector types are a class or struct that contains 8 floats
  - happens to support convenient overloaded “+”, “*”, etc. operations
- You can freely pass vectors around in ***function parameters***, ***return values***, etc.
  - they are typically kept in registers
- You can allocate ***small arrays of vectors in stack***

Vector types

```
float8_t example(float8_t a, float8_t b) {
    float8_t c[2];
    c[0] = a + b;
    c[1] = a - b;
    float8_t d = c[0] * c[1];
    return d;
}
```

Works fine!

```
float8_t example(float8_t a, float8_t b) {
    float8_t c[2];
    c[0] = a + b;
    c[1] = a - b;
    float8_t d = c[0] * c[1];
    return d;
}
```

```
vaddps %ymm1, %ymm0, %ymm2
vsubps %ymm1, %ymm0, %ymm0
vmulps %ymm0, %ymm2, %ymm0
ret
```

Efficient code!

Memory alignment

- Just one complication: care needed with memory allocation!
- Any *pointer* to **`float8_t`** has to be *properly aligned*
  - memory address has to be a multiple of 32 bytes
  - `malloc`, new, etc. do not guarantee that!
- All of these are ***seriously broken***:
  - `float8_t* p = (float8_t*)malloc(n * sizeof(float8_t));`
  - `float8_t* p = new float8_t[n];`
  - `std::vector<float8_t> p(n);`

Program might crash with 50% probability!

Memory alignment

- Always use `posix_memalign` for dynamic memory allocation
  - instead of `malloc`, `new`, `std::vector`, etc.
- See the course material for more details & examples
  - our code templates also contain memory allocation functions that you can directly use
- Remember that local variables, small arrays in stack, function parameters, return values etc. do not need any special care
  - compiler knows about their alignment requirements and does the right job (and often keeps those in registers anyway)