Programming Parallel Computers

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

Part 3A: All three forms of parallelism in action

Parallel computing resources

Intel Core i5-6500

single-precision floating-point multiplication

- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors

Parallel computing resources

- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors

OpenMP
(part 2A)

Parallel computing resources

- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors

Instruction-level parallelism (part 1D)

Parallel computing resources

- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors

Vector instructions (part 2B)

OpenMP parallel for loop

part 2A

```
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
```

thread 0: `c(0)` `c(1)` `c(2)`

thread 1: `c(3)` `c(4)` `c(5)`

thread 2: `c(6)` `c(7)`

thread 3: `c(8)` `c(9)`

Instruction-level parallelism

part 1D

Bad: dependent

```
a1 *= a0;
a2 *= a1;
a3 *= a2;
a4 *= a3;
a5 *= a4;
```

Good: independent

```
b1 *= a1;
b2 *= a2;
b3 *= a3;
b4 *= a4;
b5 *= a5;
```

Vector types

part 2B

```
float8_t a, b, c;

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

$\approx$

```
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

Is this enough?

Example

- *“Mandelbrot iteration”*:
  - `c = input`
  - `x = 0`
  - ***repeat N times:*** `x = x * x + c`
  - `result = x`

c = 0.2:

$$N = 5 \left\{ \begin{array}{l} x = 0.000^2 + 0.2 = 0.200 \\ x = 0.200^2 + 0.2 = 0.240 \\ x = 0.240^2 + 0.2 \approx 0.258 \\ x \approx 0.258^2 + 0.2 \approx 0.266 \\ x \approx 0.266^2 + 0.2 \approx 0.271 \end{array} \right.$$

c = 0.3:

$x = 0.000^2 + 0.3 = 0.300$

$x = 0.300^2 + 0.3 = 0.390$

$x = 0.390^2 + 0.3 \approx 0.452$

$x \approx 0.452^2 + 0.3 \approx 0.504$

$x \approx 0.504^2 + 0.3 \approx 0.554$

Example

- "*Mandelbrot iteration*" for 512 values, for a very large *N*:
  - `c = input[i]`
  - `x = 0`
  - ***repeat N times***: `x = x * x + c`
  - `result[i] = x`
- Calculation of `result[0]`:
  - very long dependency chain, cannot parallelize
- Calculation of `result[0]` and `result[1]`:
  - ***independent of each other!***

```
for (int i = 0; i < 512; ++i) {

    float x = 0.0;
    float c = input[i];

    for (long long n = 0; n < N; ++n) {

        x = x * x + c;

    }

    result[i] = x;

}
```

Naive sequential version

```
#pragma omp parallel for
for (int i = 0; i < 512; ++i) {

    float x = 0.0;
    float c = input[i];

    for (long long n = 0; n < N; ++n) {

        x = x * x + c;

    }

    result[i] = x;

}
```

OpenMP

```
#pragma omp parallel for
for (int i = 0; i < 64; ++i) {
    float c[8], x[8];
    for (int j = 0; j < 8; ++j) {
        x[j] = 0.0; c[j] = input[i][j];
    }
    for (long long n = 0; n < N; ++n) {
        for (int j = 0; j < 8; ++j) {
            x[j] = x[j] * x[j] + c[j];
        }
    }
    for (int j = 0; j < 8; ++j) {
        result[i][j] = x[j];
    }
}
```

Instruction-level parallelism

```
#pragma omp parallel for
for (int i = 0; i < 8; ++i) {
    float8_t c[8], x[8];
    for (int j = 0; j < 8; ++j) {
        x[j] = float8_0; c[j] = input[i][j];
    }
    for (long long n = 0; n < N; ++n) {
        for (int j = 0; j < 8; ++j) {
            x[j] = x[j] * x[j] + c[j];
        }
    }
    for (int j = 0; j < 8; ++j) {
        result[i][j] = x[j];
    }
}
```

Vector operations

```
#pragma omp parallel for
for (int i = 0; i < 8; ++i) {
    float8_t c[8], x[8];
    for (int j = 0; j < 8; ++j) {
        x[j] = float8_0; c[j] = input[i][j];
    }
    for (long long n = 0; n < N; ++n) {
        for (int j = 0; j < 8; ++j) {
            x[j] = x[j] * x[j] + c[j];
        }
    }
    for (int j = 0; j < 8; ++j) {
        result[i][j] = x[j];
    }
}
```

Using 4 threads evenly

Plenty of room for instruction-level parallelism here

8-wide vector operations

“input” and “result” are here 8×8×8 arrays

Performance?

- N = 1 billion
  - we do **1024 billion** arithmetic operations
  - running time on **3.3 GHz 4-core** Skylake CPU: **2.44** seconds
- Got: **420** billion single-precision arithmetic operations / second

Happy?

Performance!

- N = 1 billion
  - we do **1024 billion** arithmetic operations
  - running time on **3.3 GHz 4-core** Skylake CPU: 2.44 seconds
- Got: **420** billion single-precision arithmetic operations / second
- Theoretical maximum for this CPU: ≈ **422** billion / second

Yes!

Cheating?

CPUs are also very good at this kind of operations

- key operation: FMA (fused multiply and add)
- single instruction for `d = a * b + c`

- Tiny input, tiny output
- Everything in inner loops fits in CPU registers
- ***No memory accesses in inner loops***
- It would be much slower if we had any memory accesses in the performance-critical parts
- What to do if you must read some input in your inner loops?