Part 3A:
All three forms of parallelism in action
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

Intel Core i5-6500
single-precision floating-point multiplication
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
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 parallel for loop

#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

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

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];
Is this enough?
Example

• “Mandelbrot iteration”:
  • \(c = \text{input}\)
  • \(x = 0\)
  • repeat \(N\) times: \(x = x \times x + c\)
  • result = \(x\)

\[c = 0.2:\]
\[
\begin{align*}
x &= 0.000^2 + 0.2 = 0.200 \\
x &= 0.200^2 + 0.2 = 0.240 \\
x &\approx 0.240^2 + 0.2 \approx 0.258 \\
x &\approx 0.258^2 + 0.2 \approx 0.266 \\
x &\approx 0.266^2 + 0.2 \approx 0.271
\end{align*}
\]

\[N = 5\]

\[c = 0.3:\]
\[
\begin{align*}
x &= 0.000^2 + 0.3 = 0.300 \\
x &= 0.300^2 + 0.3 = 0.390 \\
x &\approx 0.390^2 + 0.3 \approx 0.452 \\
x &\approx 0.452^2 + 0.3 \approx 0.504 \\
x &\approx 0.504^2 + 0.3 \approx 0.554
\end{align*}
\]
Example

• “Mandelbrot iteration” for 512 values, for a very large $N$:
  • $c = \text{input}[i]$
  • $x = 0$
  • \textit{repeat N times}: $x = x \times x + c$
  • $\text{result}[i] = x$

• Calculation of $\text{result}[0]$:  
  • very long dependency chain, cannot parallelize

• Calculation of $\text{result}[0]$ and $\text{result}[1]$:  
  • \textit{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;
}
#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];
    }
}
#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];
    }
}
#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
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: $\approx 422$ billion / second

Yes!
Cheating?

- 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?

CPUs are also very good at this kind of operations

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