Vector instructions

lots of computing power, just be careful with memory alignment...
Vector instructions

• **1997**: MMX, 64-bit registers MM0 … MM7
• **1999**: SSE, 128-bit registers XMM0 … XMM7
• **2011**: AVX, 256-bit registers YMM0 … YMM15
• **soon**: AVX-512, 512-bit registers ZMM0…ZMM31
Vector instructions

• Hundreds of machine-language instructions
  • interpret AVX registers as vectors
  • “horizontal add with saturation”
  • “conditional dot product”
  • “sum of absolute differences”
  • “fused multiply and add” …
Vector instructions

- Hundreds of machine-language instructions
- Directly available via compiler intrinsics and built-in functions, if needed
  - \( c = \_\text{mm256\_hadd\_pd}(a, b); \)
  - \( c = \_\_\text{builtin\_ia32\_haddpd256}(a, b); \)
- see the course web site for pointers
Vector instructions

- Hundreds of machine-language instructions

- Directly available via compiler intrinsics and built-in functions, \textit{if needed}

- In many cases we do not need to worry about low-level details!
Vector types in GCC

typedef double double4_t __attribute__
  ((__vector_size__) (4*sizeof(double)));

// Now these are almost equivalent:
double4_t a;
double a[4];
Vector types in GCC

typedef float float8_t __attribute__((__vector_size__ (8*sizeof(float))));

// Now these are almost equivalent:
float8_t a;
float a[8];
Vector types in GCC

// Can address individual elements as usual:
float8_t a;
for (int i = 0; i < 8; ++i) {
    a[i] = 123.0 + i;
}

Vector types in GCC

// Can address individual elements as usual:
float8_t a[3];
for (int j = 0; j < 3; ++j) {
    for (int i = 0; i < 8; ++i) {
        a[j][i] = 123.0 + i;
    }
}
Vector types in GCC

// Operations on entire vectors:
float8_t a, b, c;
a += b * c;

// Same as:
for (int i = 0; i < 8; ++i) {
    a[i] += b[i] * c[i];
}
// Operations on entire vectors, also with scalars:
float8_t a, b, c;
a += b * c / 3 + 2;

// Same as:
for (int i = 0; i < 8; ++i) {
    a[i] += b[i] * c[i] / 3 + 2;
}
Vector types in GCC

• Always available

• Compiler uses special vector registers and instructions whenever possible

• Remember to specify the architecture
  • `g++ -march=native`
Memory alignment

- CPU vector instructions: require proper alignment of all data
- Memory addresses must be divisible by sizeof(vector)
- Easier for hardware: wide memory bus, no need to “shift” data after memory access
Memory alignment

• CPU vector instructions: *require* proper alignment

• Vector types in C code: *promise* of proper alignment

• If you use vector types *(correctly!)*, compiler can safely generate vector instructions
Memory alignment

- Vector data must be properly aligned
- **Compiler** takes care of this for local variables allocated from *stack*
- **You** take care of this for arrays allocated from *heap*
Memory alignment

- `malloc()` not necessarily good enough!
- `posix_memalign()` to allocate memory, `free()` to release
- See `common/vector.*` for helper functions
  - `float8_alloc()`, `double4_alloc()`
double4_t* x = double4_alloc(n);
double4_t* y = double4_alloc(n);

for (int i = 0; i < n; ++i) {
    for (int j = 0; j < 4; ++j) {
        x[i][j] = ...;
    }
}

for (int i = 0; i < n; ++i) {
    double4_t z = x[i];
    y[i] = z * z;
}

free(x);
free(y);
for (int i = 0; i < n; ++i) {
    double4_t z = x[i];
    y[i] = z * z;
}

L42:

```
vmovapd (%rbx,%rax), %ymm0
vmulpd %ymm0, %ymm0, %ymm0
vmovapd %ymm0, (%r12,%rax)
addq $32, %rax
cmpq %rdx, %rax
jne L42
```

“…pd” = packed doubles = vector of doubles
“ymm…” = 256-bit register
How to exploit vector extensions

some creativity required
How to exploit vector instructions?

- Needs some creativity!

- Design your algorithm so that you can do the same operation for many items, in parallel

- Often some preprocessing & postprocessing needed: convert input data to suitable vectors and back
// **Goal:** sum of squares
s = x[0]*x[0] + ... + x[n-1]*x[n-1];

// **Preprocessing:** pack to vectors
v[0] = { x[0], x[1], x[2], x[3] };
v[1] = { x[4], x[5], x[6], x[7] };
...
// Pad last vector with zeroes if needed
v[m-1] = { x[n-2], x[n-1], 0, 0 };

// **Calculation:** each component independently in parallel
y = v[0]*v[0] + ... + v[m-1]*v[m-1];

// **Postprocessing:** combine components
Other examples

• **Interleave input rows:**
  in each vector, element $i$ comes from row $i$,
  we can then process multiple rows in parallel

• **Multidimensional input:**
  ($red$, $green$, $blue$)-triples in digital images,
  multiple channels in digital audio
Efficient use of vector instructions
— instruction-level parallelism
— memory hierarchy
Toy example: sum of squares

// Repeatedly do multiply-and-add
// for an array with “size” vectors
for (int j = 0; j < iter; ++j) {
    for (int i = 0; i < size; ++i) {
        double4_t y = v[i];
        x += y * y;
    }
}

How well does this perform?
nanoseconds/vector multiplication

array size in bytes

1K  32K  1M  32M  1G
nanoseconds/vector multiplication

disappointing?
Instruction-level parallelism

- Good: parallelism in *vector operations*

- Bad: very little opportunities for *instruction-level parallelism*

- Inherently sequential:

  ```
  x += y[0]*y[0]; x += y[1]*y[1];
  x += y[2]*y[2]; x += y[3]*y[3]; ...
  ```
Bad

x  +=  y[0] * y[0];
x  +=  y[1] * y[1];
x  +=  y[2] * y[2];
x  +=  y[3] * y[3];
x  +=  y[4] * y[4];
x  +=  y[5] * y[5];
x  +=  y[6] * y[6];
x  +=  y[7] * y[7];
x  +=  y[8] * y[8];
...

Better

t[0]  +=  y[0] * y[0];
t[1]  +=  y[1] * y[1];
t[0]  +=  y[4] * y[4];
t[0]  +=  y[8] * y[8];
...
More opportunities for instruction-level parallelism
(assuming here that “size” is a multiple of 8)

```c
double4_t t[8];
...
for (int j = 0; j < iter; ++j) {
    for (int i = 0; i < size; i += 8) {
        for (int k = 0; k < 8; ++k) {
            double4_t y = v[i + k];
            t[k] += y * y;
        }
    }
}
for (int k = 0; k < 8; ++k) {
    x += t[k];
}
```

Any improvements?
nanoseconds/vector multiplication

array size in bytes

naive

ILP
Bottlenecks

- **Naive version:** *latency* of vector operations
- **ILP-friendly version, small data:** *throughput* of vector operations
- **ILP-friendly version, large data:** getting data from the *memory*
Caches
How do caches work?

• CPU ↔ L1 ↔ L2 ↔ L3 ↔ memory

• Whenever you read memory:
  • CPU reads the full *cache line* (64 bytes) from the nearest cache that contains it
  • stores it in all intermediate caches, makes room by throwing away older data
Some rules of thumb

• Repeatedly work with a small chunk of $\ll 32$KB of data:
  • all data remains in L1
  • small latency (order of 1 ns)
  • large bandwidth
Some rules of thumb

- Random reads in >> 8MB of data:
  - most memory lookups are cache misses
  - large latency (order of 100 ns)
  - small bandwidth
Some rules of thumb

• **Ideal**: linear read of L1
• **Good**: random access of L1, linear read of L2–L3
• **Tolerable**: linear read of main memory
• **Bad**: random access of main memory
• **Horrible**: random reads with *dependencies*
Some rules of thumb

• You can do useful work while you wait for data from main memory

• *Instruction-level parallelism* does it automatically, if there are some other *independent* operations that you can run
  • following a linked list: expensive
Optimising cache usage

cache blocking in matrix multiplication
Arithmetic intensity

• Throughput of arithmetic operations larger than main memory bandwidth

• Whenever you read data from main memory to caches (or from caches to registers), try to do many arithmetic operations with the same data
Example: matrix multiplication

- Multiplying $n \times n$ matrixes: $O(n^2)$ data, $O(n^3)$ operations

- Naive algorithm: each operation needs to fetch new data from memory

- Better algorithm: most operations use data that is already in cache or registers
Matrix multiplication

\[
\begin{array}{ccc}
A & \times & B \\
\times & \times & = \\
& & C
\end{array}
\]
Naive solution: poor locality

\[ A \times B = C \]
Cache blocking: better locality

\[
\begin{align*}
A \times B &= C \\
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\end{array}
\end{array}
\end{array} & \times \\
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\end{array}
\end{array}
\end{array} & = \\
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\end{array}
\end{array}
\end{array}
\end{align*}
\]
\[ \begin{array}{c c c}
A & \times & B \\
\times & = & C \\
\end{array} \] \\
\[ \begin{array}{c c c}
C_{11} & \times & \ \\
\times & = & \ \\
\end{array} \] \\
\[ \begin{array}{c c c}
X_{11} & \times & \ \\
\times & = & \ \\
\end{array} \] \\
\[ \begin{array}{c c c}
Y_{11} & \times & \ \\
\times & = & \ \\
\end{array} \] \\
\[ C_{11} = X_{11} + Y_{11} \]
Reuse data in all scales

- Reused data that you read to L3 cache
- Reused data that you read to L2 cache
- Reused data that you read to L1 cache
  - L1 cache size $\approx$ few rows of data
- Reused data that you read to CPU registers!
Reusing data in registers

- Naive: calculate 1 dot product $x \cdot y$
  - each iteration reads $1+1$ elements, does 1 multiplication and 1 addition

  $a = x[i]$;
  $b = y[i]$;
  $p = a \ast b$;
  $s += p$;
Reusing data in registers

- Naive: calculate 1 dot product $x \cdot y$
  - each iteration reads $1+1$ elements, does 1 multiplication and 1 addition

- Better: calculate simultaneously 4 dot products $x_1 \cdot y_1$, $x_1 \cdot y_2$, $x_2 \cdot y_1$, $x_2 \cdot y_2$
  - each iteration reads $2+2$ elements, does $2 \times 2$ multiplications and $2 \times 2$ additions
Reusing data in registers

• Naive: calculate 1 dot product $x \cdot y$

• Better: calculate simultaneously 4 dot products $x_1 \cdot y_1$, $x_1 \cdot y_2$, $x_2 \cdot y_1$, $x_2 \cdot y_2$
  • read 2 times as much data
  • produce 4 times as many results
  • better *arithmetic intensity*
Reusing data in registers

• Naive: calculate 1 dot product $x \cdot y$

• Better: calculate simultaneously 9 dot products $x_1 \cdot y_1, \ldots, x_3 \cdot y_3$
  • read 3 times as much data
  • produce 9 times as many results
  • still enough registers to keep everything...?
Summary

• Use *vector instructions* to better exploit parallel processing units in modern CPUs

• Pay attention to *caches*: reuse data

• Do not forget *instruction-level parallelism*

• Do not forget *using multiple threads*
Wide operands

CP1: hardware can multiply 64-bit floating-point numbers
Pipelining: can start the next independent operation before the previous result is available.

CP4: better performance for independent operations (instruction-level parallelism)
CP4: clever use of caches helps us to get data from memory to processor faster
Multiple cores

CP2: OpenMP makes it possible to use multiple cores
Many execution ports

CP4: better performance for independent operations (instruction-level parallelism)
Many execution ports

CP2: OpenMP and hyper-threading may also help if some execution ports would be idle
CP3: vector types let us perform many independent operations in parallel