Simulation and High-Performance Computing

Part 14: Explicit Vectorization

Steffen Börm

Christian-Albrechts-Universität zu Kiel

October 6th, 2020

Intrinsics

Approach: Compilers provide special functions that are not implemented in a library, but directly translated into machine code.

Example: AVX vectorization

- Definitions can be obtained via `#include <immintrin.h>`
- Data type `__m256` for 256-bit vectors
- Intrinsic functions `_mm256_set1_ps`, `_mm256_loadu_ps`, `_mm256_storeu_ps` to set, load, and store vectors
- Intrinsic functions `_mm256_add_ps`, `_mm256_sub_ps`, `_mm256_mul_ps`, `_mm256_div_ps` for addition, subtraction, multiplication, and division

Intrinsics

Approach: Compilers provide special functions that are not implemented in a library, but directly translated into machine code.

Example: AVX vectorization

- Definitions can be obtained via `#include <immintrin.h>`
- Data type `__m256` for 256-bit vectors
- Intrinsic functions `_mm256_set1_ps`, `_mm256_loadu_ps`, `_mm256_storeu_ps` to set, load, and store vectors
- Intrinsic functions `_mm256_add_ps`, `_mm256_sub_ps`, `_mm256_mul_ps`, `_mm256_div_ps` for addition, subtraction, multiplication, and division
- Intrinsic function `_mm256_cmp_ps`, `_mm256_blendv_ps` for comparisons and avoiding branches
- Intrinsic functions `_mm256_and_ps`, `_mm256_or_ps`, `_mm256_xor_ps` for bitwise logical operations

Example: Reciprocal square root

Goal: Evaluate $x = 1/\sqrt{y}$ efficiently, e.g., for gravitational fields.

Approach: The intrinsic function `_mm256_rsqrt_ps` provides us with a rough approximation of x.
Improve by Newton's iteration for $f(x) = y - \frac{1}{x^2}$.

$$x \leftarrow x - \frac{f(x)}{f'(x)} = x - \frac{y - x^{-2}}{-2x^{-3}} = \frac{x}{2}\left(3 - yx^2\right)$$

```
const _mm256 c05 = _mm256_set1_ps(0.5f);
const _mm256 c3 = _mm256_set1_ps(3.0f);

x = _mm256_rsqrt_ps(y);
x = _mm256_mul_ps(_mm256_mul_ps(c05, x),
      _mm256_sub_ps(c3,
        _mm256_mul_ps(y, _mm256_mul_ps(x, x))));
```

Example: Reciprocal square root

Goal: Evaluate $x = 1/\sqrt{y}$ efficiently, e.g., for gravitational fields.

Approach: The intrinsic function `_mm256_rsqrt_ps` provides us with a rough approximation of x.
Improve by Newton's iteration for $f(x) = y - \frac{1}{x^2}$.

$$x \leftarrow x - \frac{f(x)}{f'(x)} = x - \frac{y - x^{-2}}{-2x^{-3}} = \frac{x}{2}\left(3 - yx^2\right)$$

```
const _mm256 c05 = _mm256_set1_ps(0.5f);
const _mm256 c3 = _mm256_set1_ps(3.0f);

x = _mm256_rsqrt_ps(y);
x = _mm256_mul_ps(_mm256_mul_ps(c05, x),
      _mm256_sub_ps(c3,
        _mm256_mul_ps(y, _mm256_mul_ps(x, x))));
```

Result: 0.5 instead of 4.2 seconds, maximal error 2.98_{-8}

Example: Gravitation

Goal: Evaluate gravitational potentials

$$\varphi_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{m_j}{\|x_j - x_i\|}.$$

```
d0 = _mm256_sub_ps(_mm256_loadu_ps(y0+j), y0i);
d1 = _mm256_sub_ps(_mm256_loadu_ps(y1+j), y1i);
d2 = _mm256_sub_ps(_mm256_loadu_ps(y2+j), y2i);
norm2 = _mm256_add_ps(
            _mm256_add_ps(_mm256_mul_ps(d0, d0),
                          _mm256_mul_ps(d1, d1)),
            _mm256_mul_ps(d2, d2));
force = _mm256_mul_ps(_mm256_loadu_ps(m+j),
                      avx_rsqrt(norm2));
sum = _mm256_add_ps(sum, force);
```

Example: Gravitation

Goal: Evaluate gravitational potentials

$$\varphi_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{m_j}{\|x_j - x_i\|}.$$

```
d0 = _mm256_sub_ps(_mm256_loadu_ps(y0+j), y0i);
d1 = _mm256_sub_ps(_mm256_loadu_ps(y1+j), y1i);
d2 = _mm256_sub_ps(_mm256_loadu_ps(y2+j), y2i);
norm2 = _mm256_add_ps(
            _mm256_add_ps(_mm256_mul_ps(d0, d0),
                          _mm256_mul_ps(d1, d1)),
            _mm256_mul_ps(d2, d2));
force = _mm256_mul_ps(_mm256_loadu_ps(m+j),
                      avx_rsqrt(norm2));
sum = _mm256_add_ps(sum, force);
```

Result: 0.9 seconds instead of 6.9, relative error 1.50_{-5}.

Example: Sine

Goal: Evaluate $y = \sin(x)$ efficiently, e.g., for wave simulations.

Approach: Taylor expansion (Horner's method), maximal error 5.69_{-8}.

$$\sin(x) \approx x \left(1 - x^2 \left(\frac{1}{3!} - x^2 \left(\frac{1}{5!} - x^2 \left(\frac{1}{7!} - x^2 \left(\frac{1}{9!} - \frac{x^2}{11!}\right)\right)\right)\right)\right)$$

```
const float c[] = { 1.0, 1.0/6.0, 1.0/120.0, -.0/5040.0,
                    1.0/362880.0, 1.0/39916800.0 };
int j = sizeof(c) / sizeof(float);

y = _mm256_set1_ps(c[--j]);
x2 = _mm256_mul_ps(x, x);
for(; j-->0; )
  y = _mm256_sub_ps(_mm256_set1_ps(c[j]),
                    _mm256_mul_ps(x2, y));
```

Result: 0.3 instead of 1.4 seconds, maximal error 1.79_{-7}

Example: Sine

Goal: Evaluate $y = \sin(x)$ efficiently, e.g., for wave simulations.

Approach: Taylor expansion (Horner's method), maximal error 5.69_{-8}.

$$\sin(x) \approx x \left(1 - x^2 \left(\frac{1}{3!} - x^2 \left(\frac{1}{5!} - x^2 \left(\frac{1}{7!} - x^2 \left(\frac{1}{9!} - \frac{x^2}{11!}\right)\right)\right)\right)\right)$$

```
const float c[] = { 1.0, 1.0/6.0, 1.0/120.0, -.0/5040.0,
                    1.0/362880.0, 1.0/39916800.0 };
int j = sizeof(c) / sizeof(float);

y = _mm256_set1_ps(c[--j]);
x2 = _mm256_mul_ps(x, x);
for(; j-->0; )
  y = _mm256_sub_ps(_mm256_set1_ps(c[j]),
                    _mm256_mul_ps(x2, y));
```

Result: 0.3 instead of 1.4 seconds, maximal error 1.79_{-7}

Example: Quadratic equation

Goal: Stable computation of the solutions of $x^2 + 2\,p\,x + q = 0$.

$$x_1 = \begin{cases} p + \sqrt{p^2 - q} & \text{if } p \geq 0, \\ p - \sqrt{p^2 - q} & \text{otherwise,} \end{cases} \qquad x_2 = \frac{q}{x_1}.$$

Idea: Copy the sign of p to the square root.

```
mask = _mm256_castsi256_ps(_mm256_set1_epi32(0x80000000));
sgn = _mm256_and_ps(p, mask);
x1 = _mm256_add_ps(p,
       _mm256_or_ps(sgn,
         _mm256_sqrt_ps(
           _mm256_sub_ps(_mm256_mul_ps(p, p), q))));
x2 = _mm256_div_ps(q, x1);
```

Example: Quadratic equation

Goal: Stable computation of the solutions of $x^2 + 2\,p\,x + q = 0$.

$$x_1 = \begin{cases} p + \sqrt{p^2 - q} & \text{if } p \geq 0, \\ p - \sqrt{p^2 - q} & \text{otherwise,} \end{cases} \qquad x_2 = \frac{q}{x_1}.$$

Idea: Copy the sign of p to the square root.

```
mask = _mm256_castsi256_ps(_mm256_set1_epi32(0x80000000));
sgn = _mm256_and_ps(p, mask);
x1 = _mm256_add_ps(p,
       _mm256_or_ps(sgn,
         _mm256_sqrt_ps(
           _mm256_sub_ps(_mm256_mul_ps(p, p), q))));
x2 = _mm256_div_ps(q, x1);
```

Result: 0.7 seconds instead of 1.3, maximal error 2.28_{-7}.

Branches

Problem: How can we handle data-dependent branches, if we can only perform operations on entire vectors?

Idea: Execute all branches and combine the results.

- `_mm256_cmp_ps` compares all components of two vectors and creates a bitmask:
  If the comparison was successful, the component is set to `~0`.
  If it was not successful, the component is set to `0`.
- `_mm256_and_ps`, `_mm256_andnot_ps`, `_mm256_or_ps` can be used to combine bitmasks or to merge results according to bitmasks.
- `_mm256_blendv_ps` merges results of two branches according to a bitmask.

Example: Data-dependent branch

Goal: Evaluate $\sin(\pi x)$ efficiently for $x \in [0, 1]$. Our Taylor expansion converges more quickly in $[0, \frac{1}{2}]$. Let's use the sine function's symmetry:

$$\sin(\pi x) = \begin{cases} \sin(\pi(1-x)) & \text{if } x \in [\frac{1}{2}, 1], \\ \sin(\pi x) & \text{otherwise.} \end{cases}$$

```
const __m256 c05 = _mm256_set1_ps(0.5f);
const __m256 c1 = _mm256_set1_ps(1.0f);

msk = _mm256_cmp_ps(x, c05, _CMP_GT_OQ);
x = _mm256_blendv_ps(x, _mm256_sub_ps(c1, x), msk);
y = avx_sin_pi(x);
```

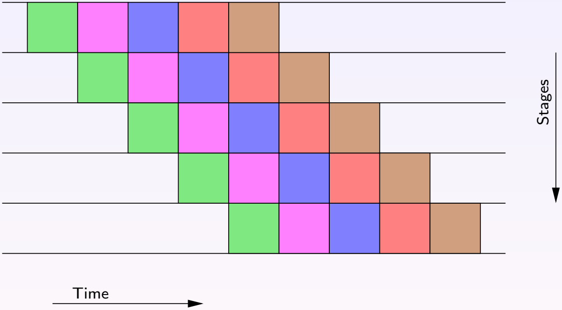
The bitmask `msk` for each component is `~0` if $x > \frac{1}{2}$ and `0` otherwise. In the first case, the blend functions chooses $1 - x$, in the second it chooses x.

Latency and throughput

Latency: Time from the start to the end of an operation.

Throughput: Time from the start of one operation to the start of the next.

Pipelined processors: The throughput can be far smaller than the latency.

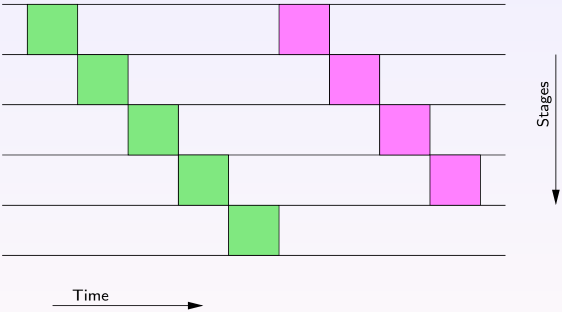


Latency and throughput

Latency: Time from the start to the end of an operation.

Throughput: Time from the start of one operation to the start of the next.

Pipelined processors: The throughput can be far smaller than the latency.



Condition: The operations must be independent.

Improving the throughput

Example: Taylor expansion by Horner's algorithm.

$$
\begin{aligned}
y &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 \\
&= a_0 + x(a_1 + x(a_2 + x(a_3 + x(a_4 + x a_5))))
\end{aligned}
$$

Improving the throughput

Example: Taylor expansion by Horner's algorithm.

$$
\begin{aligned}
y &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 \\
&= a_0 + x(a_1 + x(a_2 + x(a_3 + x(a_4 + x a_5)))) \\
&= (a_0 + x^2(a_2 + x^2 a_4)) + x(a_1 + x^2(a_3 + x^2 a_5)).
\end{aligned}
$$

Improving the throughput

Example: Taylor expansion by Horner's algorithm.

$$
\begin{aligned}
y &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 \\
&= a_0 + x(a_1 + x(a_2 + x(a_3 + x(a_4 + x a_5)))) \\
&= (a_0 + x^2(a_2 + x^2 a_4)) + x(a_1 + x^2(a_3 + x^2 a_5)).
\end{aligned}
$$

Mathematically, both formulations are equivalent.
The second formulation allows the left and the right term to be evaluated independently, thus improving the throughput.

Improving the throughput

Example: Taylor expansion by Horner's algorithm.

$$\begin{aligned} y &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 \\ &= a_0 + x(a_1 + x(a_2 + x(a_3 + x(a_4 + x a_5)))) \\ &= (a_0 + x^2(a_2 + x^2 a_4)) + x(a_1 + x^2(a_3 + x^2 a_5)). \end{aligned}$$

Mathematically, both formulations are equivalent.
The second formulation allows the left and the right term to be evaluated independently, thus improving the throughput.

Problem: The compiler is not allowed to rearrange floating-point operations if it results in different rounding errors.
$\rightarrow$ It is up to us.

Filling the gaps

Idea: We can use the latency gaps of one computation for another one.

Example: Sine and cosine computed simultaneously.

```
const float sc[] = { 1.0, 1.0/6.0, 1.0/120.0, 1.0/5040.0,
                     1.0/362880.0, 1.0/39916800.0 };
const float cc[] = { 1.0, 1.0/2.0, 1.0/24.0, 720.0,
                     1.0/40320.0, 1.0/3628800.0 };
sx = _mm256_set1_ps(sc[5]);
cx = _mm256_set1_ps(cc[5]);
x2 = _mm256_mul_ps(x, x);
for(j=5; j-->0; ) {
  sx = _mm256_mul_ps(x2, sx);
  cx = _mm256_mul_ps(x2, cx);
  sx = _mm256_sub_ps(_mm256_set1_ps(sc[j]), sx);
  cx = _mm256_sub_ps(_mm256_set1_ps(cc[j]), cx);
}
```

Filling the gaps

Idea: We can use the latency gaps of one computation for another one.

Example: Sine and cosine computed simultaneously.

```
const float sc[] = { 1.0, 1.0/6.0, 1.0/120.0, 1.0/5040.0,
                     1.0/362880.0, 1.0/39916800.0 };
const float cc[] = { 1.0, 1.0/2.0, 1.0/24.0, 720.0,
                     1.0/40320.0, 1.0/3628800.0 };
sx = _mm256_set1_ps(sc[5]);
cx = _mm256_set1_ps(cc[5]);
x2 = _mm256_mul_ps(x, x);
for(j=5; j-->0; ) {
  sx = _mm256_mul_ps(x2, sx);
  cx = _mm256_mul_ps(x2, cx);
  sx = _mm256_sub_ps(_mm256_set1_ps(sc[j]), sx);
  cx = _mm256_sub_ps(_mm256_set1_ps(cc[j]), cx);
}
```

Result: Sine takes 0.23 seconds, sine and cosine take 0.34 seconds.

Prefetching

Problem: Read operations from main memory can have very long latencies.

Idea: If we know the algorithm well enough, we can provide the processor with hints about what data we will need soon.

```
for(i=0; i+7<n; i+=8) {
  _mm_prefetch(x+i+64);

  vx = _mm256_loadu_ps(x+i);

  _mm256_stream_ps(y+i, _mm256_mul_ps(vx, vx));
}
```

- The `_mm_prefetch` intrinsic lets the processor know that we may soon need information from an address.
- The `_mm256_stream_ps` intrinsic lets the processor know that we do not expect to use the written data again soon, so it does not have to be kept in the cache.

Summary

Intrinsics allow us to directly work with the processor's low-level instructions.
Learning how to use intrinsics takes time, but the performance of programs can be significantly improved in situations that are too complicated for the compiler's automatic vectorization.

Data-dependent Branches require special treatment, e.g., by constructing bitmasks and using bit manipulation or blending to combine partial results.

Latencies during the execution of instructions should be avoided, e.g., by interleaving independent parts of a computation or even completely different computations.

Prefetching can help avoid memory latencies, particularly if the access pattern is too irregular for the processor to predict.