Simulation and High-Performance Computing

Part 13: Vectorization

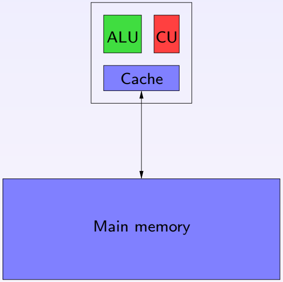
Steffen Börm

Christian-Albrechts-Universität zu Kiel

October 6th, 2020

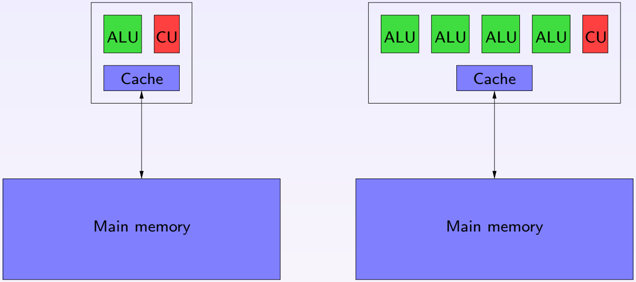
SIMD computing

Task: Reduce the computing time.



SIMD computing

Task: Reduce the computing time.



Idea: Simply increase the number of arithmetic-logic units.

Limitation: Since we have only one control unit, all arithmetic-logic units have to perform the same operation in each cycle:
Single Instruction, Multiple Data, also known as vector computing.

Vector extensions

Vector computers have existed since the construction of the Cray-1 supercomputer in 1976.

Today, vector operations are typically realized by extending a standard processor's registers and instruction set.

- MMX (1997): Intel and AMD; only integers
- AltiVec (1998): Motorola, Apple, IBM; single-precision floats
- SSE (1999): Intel and AMD; single-precision floats
- SSE2 (2000): Intel and AMD; adds double-precision floats to SSE
- NEON (2009): ARM; only integers
- AVX (2011): Intel and AMD; single- and double-precision floats
- AVX2 (2013): Intel and AMD; adds integers to AVX
- AVX512 (2013): Intel; integers, single- and double-precision floats
- SVE (2017): ARM; integers, single- and double-precision floats

Example: Vector addition

Task: Perform $y \leftarrow y + \alpha x$ with n-dimensional vectors using AVX.

```
__m256 v_alpha = _mm256_set1_ps(alpha);
int i;

for(i=0; i+7<n; i+=8)
  _mm256_storeu_ps(y+i,
    _mm256_add_ps(
      _mm256_loadu(y+i),
      _mm256_mul_ps(v_alpha, _mm256_loadu(x+i))));

for(; i<n; i++)
  y[i] += alpha * x[i];
```

- `_mm256_loadu` and `_mm256_storeu` read and write 8 floats ($8 \times 32 = 256$ bits) from and to memory.
- `_mm256_add_ps` and `_mm256_mulps` add and multiply 8 pairs of floats simultaneously.

Memory bandwidth

Problem: In theory, our vector example should run eight times as fast as a simple implementation, since it performs eight multiplications and additions per cycle.
In practice, the speedup is frequently far smaller.

Computational throughput: A modern desktop processor can complete one or even two add and multiplication instructions per cycle.

- At 3600 GHz, $8 \times 1.8 = 14.4$ billion coefficients updated per second.
- This would require 28.8 billion floats per second, i.e., 115.2 GiB/s.

Memory bandwidth: A modern desktop processor can read and write approximately 47 GiB/s with double-channel memory.
A server processor can read and write approximate 119 GiB/s with six-channel or 190 GiB/s with eight-channel memory.

Automatic vectorization

Approach: Instead of using vector instructions "by hand", let the compiler handle the vectorization of our code.

```
float *x, *y;

for(i=0; i<n; i++)
  y[i] += 5.76f * x[i];
```

If we use the GNU C compiler on an Intel or AMD processor, we can enable automatic vectorization using the command-line parameter `-ftree-loop-vectorize` (implied by the optimization parameter `-O3`) and use the parameter `-mavx` to choose the AVX command set.

Helping the compiler: Restricted arrays

Aliasing: In order to vectorize loops, the compiler has to be able to ensure that read and write operations do not interfere. This is not easy, since different pointers can access the same array.

```
void
simple_axpy(int n, const float * restrict x,
                   float * restrict y)
{
  int i;

  for(i=0; i<n; i++)
    y[i] += x[i];
}
```

Here, the `restrict` keyword signals that `x` and `y` point to entirely different arrays.

Helping the compiler: Reordering floating-point operations

Rounding errors depend on the sequence of floating-point operations.

```
float sum = 0.0f;

for(i=1; i<=n; i++)
  sum += i;
```

Without our explicit consent, the compiler can use vectorization only if the result remains exactly the same. In this case, the loop cannot be vectorized, since that would mean rounding differently.

Solution: We can allow the compiler to ignore rounding errors when changing the order of operations, e.g., by using the command-line parameter `-ffast-math` with the GNU C compiler.

SIMD with OpenMP

OpenMP 4.0 introduces additional compiler directives for vectorization.

Vector addition: We can signal that the loop is safe for vectorization by using the *#pragma omp simd* directive.

```
#pragma omp simd
for(i=0; i<n; i++)
  y[i] += alpha * x[i]
```

Reordering: We can signal that we do not mind if the compiler reorders floating-point operations.

```
#pragma omp simd reduction(+:sum)
for(i=1; i<=n; i++)
  sum += i;
```

SIMD functions

Problem: If we call a function from a vectorized loop, the compiler needs a vectorized version of this function.
If the function is defined in a different source file, we have to signal that a vectorized version should be prepared by the compiler.

```
#pragma omp declare simd
float
my_sinf(float x)
{
  const float c[] = { 1.0, -1.0/6.0, 1.0/120.0, -1.0/5040.0,
                      1.0/362880.0, -1.0/39916800.0 };
  float x2 = x * x;

  return x * (c[0] + x2 * (c[1] + x2 * (c[2] + x2 *
             (c[3] + x2 * (c[4] + x2 * c[5])))));
}
```

Nested loops

Example: Horner's algorithms for evaluating polynomials,

$$\sum_{j=0}^{m} a_j x^j = a_0 + x\Big(a_1 + x\big(a_2 + \dots\big)\Big)$$

```
for(j=m; j-->0; )
  #pragma omp simd
  for(i=0; i<n; i++)
    p[i] = a[j] + x[i] * p[i];
```

```
#pragma omp simd
for(i=0; i<n; i++)
  for(j=m; j-->0; )
    p[i] = a[j] + x[i] * p[i];
```

Observation: The *ji* version needs 4.0 seconds, the *ij* version 0.8 seconds, since the latter has to access main memory less frequently.

Array of structures vs structure of arrays

Array of structures: Properties of an object are stored together.

```
typedef struct {
  real x, y, z, m;
} planet;

planet pl[N];
```

Structure of arrays: Properties are stored in separate arrays.

```
typedef struct {
  real x[N], y[N], z[N], m[N];
} planets;
```

In a parallelized code, the second approach allows the processor to access memory more efficiently.

Example: 10.9 seconds for AoS, 6.7 seconds for SoA

Array of structures vs structure of arrays

Array of structures: Properties of an object are stored together.

```
typedef struct {
  real x, y, z, m;
} planet;

planet pl[N];
```

Structure of arrays: Properties are stored in separate arrays.

```
typedef struct {
  real x[N], y[N], z[N], m[N];
} planets;
```

In a parallelized code, the second approach allows the processor to access memory more efficiently.

Example: 10.9 seconds for AoS, 6.7 seconds for SoA (with some cheating).

Data-dependent branches

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

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

With a branch: 1.3 seconds.

```
for(i=0; i<n; i++) {
  if(p[i] >= 0.0)
    x1[i] = p[i] + sqrtf(p[i] * p[i] - q[i]);
  else
    x1[i] = p[i] - sqrtf(p[i] * p[i] - q[i]);
  x2[i] = q[i] / x1[i];
}
```

Data-dependent branches

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

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

With a branch: 1.3 seconds.

```
for(i=0; i<n; i++) {
  if(p[i] >= 0.0)
    x1[i] = p[i] + sqrtf(p[i] * p[i] - q[i]);
  else
    x1[i] = p[i] - sqrtf(p[i] * p[i] - q[i]);
  x2[i] = q[i] / x1[i];
}
```

Without a branch: 0.9 seconds.

```
for(i=0; i<n; i++) {
  x1[i] = p[i] + copysignf(sqrtf(p[i] * p[i] - q[i]), p[i]);
  x2[i] = q[i] / x1[i];
}
```

Summary

SIMD computing: The same operation is applied to multiple sets of input data simultaneously.

Explicit vectorization: We use processor-specific instructions and data types to vectorize our program.

Automatic vectorization: We can assist the compiler in vectorizing our program, e.g., by

- providing restricted arrays,
- allowing it to reorder floating-point operations,
- using OpenMP directives,
- avoiding unnecessary accesses to main memory,
- avoiding data-dependent branches.