Vectorization

Ned Nedialkov

McMaster University

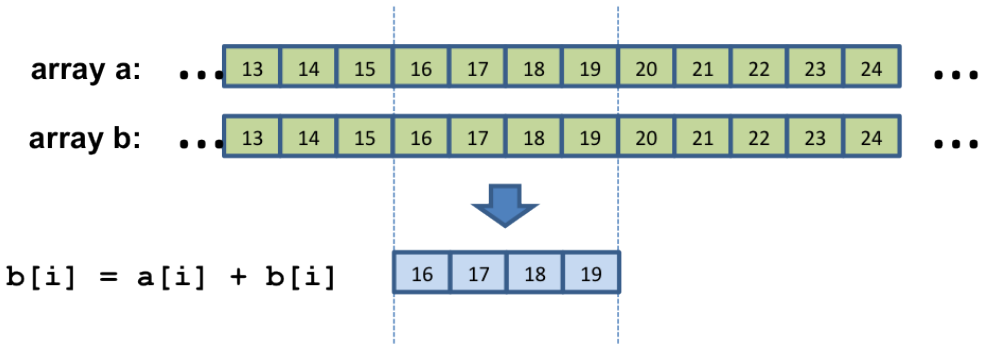
17 January 2023

Outline

SIMD basics

SIMD: Single Instruction Multiple Data

Single instruction is applied to multiple data elements.



- Intel: SSE (Streaming SIMD Extensions), AVX (Advanced Vector Extensions), AVX2, AVX-512.
- AMD: SSE and AVX.
- ARM: SIMD instruction sets through NEON technology.
- NVIDIA: SIMD through CUDA technology.
- IBM: POWER processors SIMD through Vector Scalar Extension (VSX) technology.

SSE, AVX

Streaming SIMD Extensions (SSE)

- SSE, 64-bit registers, single precision (1999)
- SSE2, 128-bit registers, single and double precision (2000)
- SSE3 adds string and integer instructions (2004)
- SSE4 adds instructions for multimedia and gaming (2006)

Advanced Vector Extensions (AVX)

- AVX, 256-bit registers (2011)
- AVX2, 256-bit registers (2013)

  Adds integer operations, gather and scatter, fused multiply-add (FMA) ...
- AVX-512, 512-bit registers (2013)

  Adds instructions for cryptography and compression/decompression, for improving performance in machine learning ...

Example

```
for (i = 0; i < n; i++)
   c[i] = a[i] + b[i]
```

Loop unrolling:

```
for (i = 0; i < n; i+=4)
  {
   c[i] = a[i] + b[i];
   c[i+1] = a[i+1] + b[i+1];
   c[i+2] = a[i+2] + b[i+2];
   c[i+3] = a[i+3] + b[i+3];
   }
 // handle remainder of the loop
```

Vectorization: execute the 4 statements in parallel.

Aliasing

Consider

```
// add.c
void add(int n, double *a, double *b, double *c) {
  for (int i = 0; i < n; i++)
    c[i] = a[i] + b[i];
}
```

If we call for example

```
add(n, a, b, a+1);
```

inside the loop we have `a[i+1] = a[i]+b[i]`

Aliasing and data dependence. Not really vectorizable.

Data dependency

This loop can't be vectorized as `b[i]` depends on `b[i-1]`

```
for (int i = 1; i < n; i++)
      b[i] += a[i] + b[i - 1];
```

Utilizing SIMD

In increasing programming effort:

- Vectorized libraries
- Compiler auto vectorization
  - GCC: `-O2 -ftree-vectorize` or just `-O3`
    To disable vectorization `-fno-vectorize`
  - Intel compilers: `-O2` or `-O3`
    To enable extra levels of optimizations
    `-x<host architecture>`, e.g. `-xskylake-avx512`
    `-xhost` enables all optimizations and advanced vectorization for the processor
    To disable vectorization `-no-vec`.
- Pragmas
- Intrinsics (C/C++), see ▸ Intel Intrinsics Guide
- Assembly

Compiler reports

GCC

Consider

```
1  // add.c
2  void add(int n, double *a, double *b, double *c) {
3    for (int i = 0; i < n; i++)
4      c[i] = a[i] + b[i];
5  }
```

`gcc -c -O3 -fopt-info-vec add.c`

produces (all reports on Intel Core i9-10850K)

```
add.c:3:16: optimized: loop vectorized using 16 byte vectors
add.c:3:16: optimized:  loop versioned for vectorization because
of possible aliasing
```

- 16 byte vectors = 128 bit vectors, 2 doubles

"Loop versioned for vectorization because of possible aliasing" means that (from ChatGPT)

- a loop-based algorithm has been modified or rewritten to take advantage of vectorization while also addressing the issue of aliasing
- the compiler has modified the code of a loop to make it amenable to vectorization

Compiler reports

GCC

`-march=native` produce instructions for the machine on which the code is compiled

```
gcc -c -march=native -O3 -fopt-info-vec add.c
```

produces

```
add.c:3:16: optimized: loop vectorized using 32 byte vectors
add.c:3:16: optimized:  loop versioned for vectorization because
of possible aliasing
add.c:3:16: optimized: loop vectorized using 16 byte vectors
```

- 32 byte vectors = 256 bit vectors
  Without `-march=native`: 16 byte vectors
- 16 byte vectors = 128 bit vectors

Intel compilers

```
icx -O3 -qopt-report=2 -qopt-report-file=stderr -c add.c
```

produces

```
LOOP BEGIN at add.c (3, 1)
<Multiversioned v2>
    remark #15319: Loop was not vectorized: novector directive used
LOOP END

LOOP BEGIN at add.c (3, 1)
<Multiversioned v1>
    remark #25228: Loop multiversioned for Data Dependence
    remark #15436: loop was not vectorized:
    remark #25439: Loop unrolled with remainder by 8
LOOP END

LOOP BEGIN at add.c (3, 1)
<Remainder loop>
    remark #25585: Loop converted to switch
LOOP END
```

Versioning

- Modern compilers can produce multiversioned code containing vectorized and unvectorized versions.
- The correct version may be chosen at runtime.
- If there are too many manipulations of addresses or/and indexing, the vectorized version may not be chosen when it is safe.

Pragmas

GCC

`#pragma GCC ivdep`

- Ignore Vector DEPendencies
- Tells the compiler data dependencies are safe to ignore.

Consider

```
1  // add2.c
2  void add2(int n, double *a, double *b, double *c) {
3  #pragma GCC ivdep
4    for (int i = 0; i < n; i++)
5      c[i] = a[i] + b[i];
6  }
```

`gcc -c -O3 -fopt-info-vec add2.c`

produces

```
add2.c:3:3: optimized: loop vectorized using 16 byte vectors
```

Intel compilers

`#pragma vector always`

is like

`#pragma GCC ivdep`

Consider

```
1  // add3.c
2  void add3(int n, double *a, double *b, double *c){
3  #pragma vector always
4     for (int i = 0; i < n; i++)
5        c[i] = a[i] + b[i];
6  }
```

`icx -O3 -qopt-report=2 -qopt-report-file=stderr -c add3.c`

produces

```
LOOP BEGIN at add3.c (4, 5)
<Multiversioned v2>
    remark #15319: Loop was not vectorized: novector directive used
LOOP END

LOOP BEGIN at add3.c (4, 5)
<Multiversioned v1>
    remark #25228: Loop multiversioned for Data Dependence
    remark #15300: LOOP WAS VECTORIZED
    remark #15305: vectorization support: vector length 2
LOOP END

LOOP BEGIN at add3.c (4, 5)
<Remainder loop for vectorization>
LOOP END
```

`#pragma omp simd`

- indicates a loop can be vectorized, from OpenMP 4.0

Consider

```
1  // vecdep.c
2  void vec_dep1(int n, double *a, double alpha, int k) {
3    for (int i = 0; i < n; i++)
4      a[i] = a[i + k] * alpha;
5  }
6  void vec_dep2(int n, double *a, double alpha, int k) {
7  #pragma omp simd
8    for (int i = 0; i < n; i++)
9      a[i] = a[i + k] * alpha;
10 }
```

```
gcc -c -O3 -fopt-info-vec -fopenmp vecdep.c
```

```
vecdep.c:3:23: optimized: loop vectorized using 16 byte vectors
vecdep.c:3:23: optimized:  loop versioned for vectorization because
 of possible aliasing
vecdep.c:9:20: optimized: loop vectorized using 16 byte vectors
```

`restrict` keyword

`restrict` (C99) tells the compiler no aliasing.

```
1  // add4.c
2  void add4(int n, double *restrict a, double *restrict b, double *restrict
           c)
3  {
4     for (int i = 0; i < n; i++)
5        c[i] = a[i] + b[i];
6  }
```

`icx -O3 -mavx -qopt-report=2 -qopt-report-file=stderr -c add4.c`

produces

```
LOOP BEGIN at add4.c (3, 5)
    remark #15300: LOOP WAS VECTORIZED
    remark #15305: vectorization support: vector length 4
LOOP END

LOOP BEGIN at add4.c (3, 5)
<Remainder loop for vectorization>
LOOP END
```

Some guidelines

Loops

- Number of iteration must be known at runtime.
  - Single control flow within the loop.
  - No `break` statements.
  - No `if` and `switch` statements.
    In some cases the compiler can get around them and vectorize.
- No function calls.
  - Unless functions are inlined.
  - Except vectorized functions `sin`, `sqrt`,... e.g. from Intel Vector Math Library
- No indirect indexing.
  - e.g. `a[b[i]]`
  - data needs to be aligned and sequential in memory
  - In nested loops, the compiler will normally try to vectorize only the innermost loop.
    The compiler may be able to rearrange the order of the loops.

Some guidelines

Intel's recommendations

From ▸ Avoid Manual Loop Unrolling:

- "The Intel Compiler can typically generate efficient vectorized code if a loop structure is not manually unrolled.
- It is better to let the compiler do the unrolls, and you can control unrolling using `#pragma unroll(n)`.
- Vector-alignment, loop-collapsing, interactions with other loop optimizations become much more complex if the compiler has to "undo" the manual unrolling.

- In all but the simplest of cases, this refactoring has to be done by the user to get the best performing vector-code.
- ... manual loop unrolling tends to tune a loop for a particular processor or architecture, making it less optimal for some future port of the application.
- Generally, it is good advice to write code in the most readable, straightforward manner. This gives the compiler the best chance of optimizing a given loop structure."