Vectorization and tools

Fifth International Workshop for Future Challenges in Tracking and Trigger Concepts

› 13 May 2014

Georgios Bitzes, CERN openlab
Andrzej Nowak, CERN openlab

CERN openlab

The ever-increasing vector width

- **1996: MMX**
- **1999: SSE**
  - Instructions only for four 32-bit single-precision floating point
- **2001: SSE2**
- **2004: SSE3**
- **2006: SSE4**
- **2011: AVX, starting with “Sandy Bridge” and a vector width of 256 bits**
- **2011: Xeon Phi with custom vector instructions operating on 512-bit vectors**
- **2013: AVX2 with Fused Multiplication and Addition instructions (FMA)**
- **Upcoming: AVX-512 on Intel “Skylake” microarchitecture and an increased vector width of 512 bits on x86-64**

Some downsides..

- **Increases in vector size not always correspond to equivalent increases in efficiency**
  - Not all operations parallelized in hardware
    - AVX division has around twice the latency as an SSE division…
- **This increase cannot go on forever..**
  - Likely to have a stop at 1024 bits – memory bandwidth becoming a limiting factor
- **Vectorization attempted in HEP during the 80s, did not continue**

How to use in our software?

- Today, an abundance of SIMD instructions in all commodity hardware
- Still, relatively low usage despite large potential benefits
- Retrofitting SIMD in existing software difficult or impossible to do – often requires changes in fundamental data structures
- Many subtle things to take into account: data layout, alignment, dependencies among loop iterations, pointer aliasing…

Candidates for vectorization

- Not all problems suitable for vectorization, but a lot are
- Example: No way to vectorize this exact loop

```
fib[0] = 0;
fib[1] = 1;
for(i = 2; i < N; i++) {
    fib[i] = fib[i-1] + fib[i-2];
}
```

Dependency among loop iterations

Autovectorization

- **“Well, it’s the compiler’s job to emit the correct instructions, why should we care?”**
- **Autovectorization can do the right thing in many cases, but not all – often needs help with directives**

```
int add(float *a, float *b, float *c, int N)
{
    int i;
    for(i = 0; i < N; i++) {
        c[i] = a[i] + b[i]
    }
}
```

- **Will it vectorize?**

taken from slides of Andrzej Nowak

Pointer aliasing

- **Pointer aliasing: What if c-1 and b actually point to the same location in memory?**
  - Autovectorization: The compiler could place values from **a** and **b** in a vector register.. Calculating c[0], c[1], c[2], c[3] in a single instruction.
  - But the value of c[1] depends on b[1], which is actually c[0]…
  - Hidden dependency among loop iterations!
- **A compiler has to be conservative and assume this is happening**

Pointer aliasing (2)

- **C99: restrict keyword**
  - A promise to the compiler that memory accesses through a pointer will not alias memory accesses from **any** other pointer
  - Not in C++ standard but compiler specific extensions exist
    - __restrict__ in GCC
  - Compiler is then free to apply further optimizations that would be unsafe under the assumption of pointer aliasing
  - If there actually **is** aliasing when we promise there's not.. undefined behavior
    - Never lie to the compiler!

Alignment

Alignment

- Two categories of load/store vector instructions – aligned and unaligned
- An unaligned instruction can be many times slower than its aligned equivalent – especially if it spans over two cache lines..
- If striving for performance, we need to use aligned data structures AND make sure the compiler knows they're aligned

Alignment (2)

- **Many options**
- **Aligning on the stack**
  - New C++11 feature: `alignas` specifier with which you can declare the (minimum) desired alignment
  - Example:
    - `alignas(32) int array[N];`
  - Guarantees that `array % 32 == 0`
  - Can also be used to align data member inside an object

Alignment (3)

```
class A {
    public:
        int N;
        alignas(32) int array[ .. ];
}
```

- Other compiler-specific extensions for the same purpose: __declspec(align(32)) and __attribute__((aligned(32)))

Aligning on the heap

- Many malloc functions that take the alignment as an argument: _mm_malloc, _aligned_malloc..
- std::align: Over-allocate – then pass a pointer and a size, will truncate and give an aligned pointer inside that container

Alignment (4)

- After allocating the raw memory, possible to create objects on it with placement new

```
void *ptr = _mm_alloc(size, alignment);
MyAlignedObject *obj = new (ptr) MyAlignedObject();
```

- Or override operators new and delete of the target object:

```
void * operator new(std::size_t sz) {
  void *aligned_buffer=_mm_malloc( sizeof(*this), 32 );
  return ::operator new(sz, aligned_buffer);
}
void operator delete(void * ptr) {
  _mm_free(ptr);
}
```

Code snippet courtesy of Sandro Wenzel

Alignment (5)

- **Still need to tell the to-be-vectorized code that the data structures are aligned!**
  - Just declaring them as aligned during allocation/construction is not enough..
  - If the data structures are defined in a different compilation unit than the one we hope to auto-vectorize, the compiler cannot know they're aligned
    - An aligned instruction operating on unaligned data will crash our program.. again, the compiler has to be conservative
  - __assume_aligned(ptr, 32) will do the trick
    - (or __builtin_assume_aligned …)

Alignment (6)

- **How much alignment should there be?**
  - Equal to the size of the vector registers
  - 16 bytes on 128-bit SSE
  - 32 bytes on 256-bit AVX / AVX2
  - 64 bytes on 512-bit Xeon Phi / AVX-512
- **Easy to get wrong – if you're segfaulting, this is the first thing to check**
  - Never hardcode the alignment! Use a #define

A better attempt at autovectorization

```
int add(float *a, float *b, float *restrict c, int N) {
    __builtin_assume_aligned(a, 32);
    __builtin_assume_aligned(b, 32);
    __builtin_assume_aligned(c, 32);
    int i;
    for(i = 0; i < N; i++) {
        c[i] = a[i] + b[i]
    }
}
```

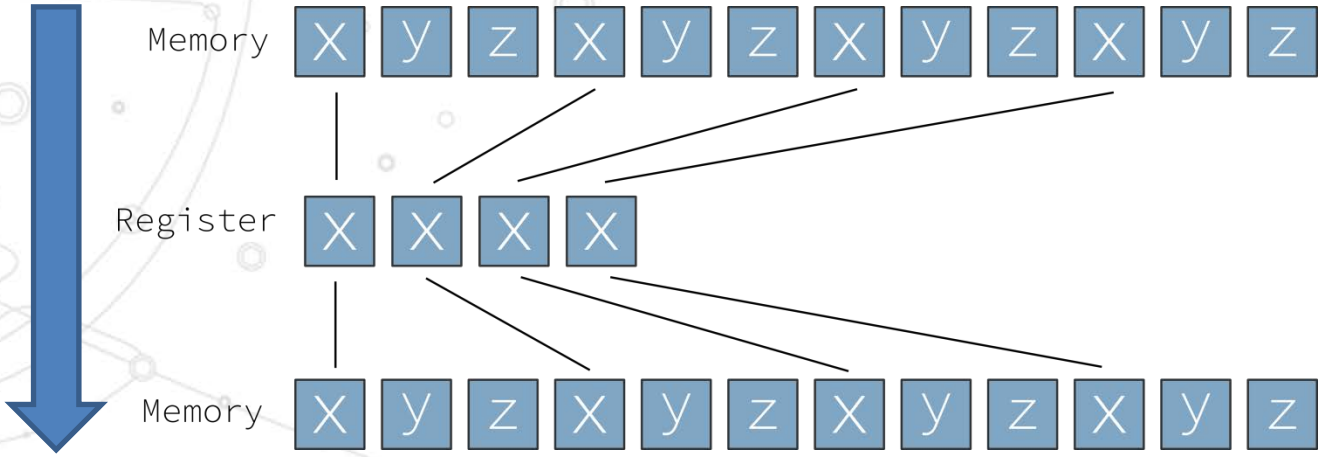
Array size

- **If size of the array not multiple of vector size, there are leftover elements to be processed sequentially**
  - Example: N == 10, vector size of 8.. Two more elements to process sequentially
- **If we know array size is a multiple of the vector size, we can tell the compiler:**
  - `__assume(N%8==0);` (ICC only)
  - Now compiler doesn't need to generate code to handle leftover elements

Data layout

- What's going on here?

```
for(i = 0; i < N; i++) {
    a[i].x = k * a[i].x
}
```



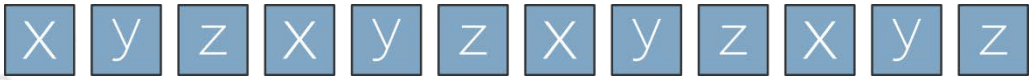
Data layout (2)

- **Gather and scatter instructions supported by the hardware – slower, though**
  - Processor exchanges memory using entire cache lines – would need to load/store whole array, even if not using all
    - More memory bandwidth, more operations
    - … and crucially, more cache misses!
    - If having a cache miss on each access, vectorization only shortens the time between cache misses..
  - Diminished benefits from vectorization

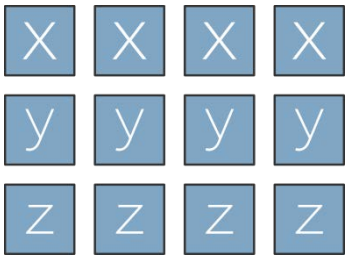
Data layout (3)

- **Suggestion: Prefer contiguous memory layouts**
  - Array of Structures (AOS) vs Struct of Arrays (SOA)

```
struct point_t {
    double x, y, z;
} points[4];
```




```
struct points_t {
    double x[4], y[4], z[4];
} points;
```



Branches?

- **Can be problematic, but not fatal**
  - Masks: Essentially an array of booleans, one value for each element in a vector register
  - Instructions that operate only on the elements whose corresponding bit in the mask is on
  - Example: Loop can still vectorize, despite the branch

```
for(i = 0; i < N; i++) {
    if(a[i] >= 0) a[i] = sqrt(a[i]);
}
```

Increase in memory traffic

- **The faster you process the data, the more often you need to fetch again**
  - Example: Without vectorization, a cache line worth of data might take 200 cycles to process – with vectorization only 70
- **Impact of cache misses more visible**
  - If your loop generates many cache misses, vectorization will just shorten the time from one cache miss to another
- **Prefer to do as many calculations as possible on a set of data**
  - Doing 10-20 operations with some data much better than doing 2-3 before writing back to memory
    - c[i] = a[i]*a[i] + b[i]*b[i] + b[i]*a[i] + b[i]/a[i] would likely benefit more than c[i] = a[i] + b[i]
- **Memory bandwidth becoming a limiting factor**

ICC specific extensions

#pragma ivdep

- Put before a loop – tells the compiler to ignore assumed vector dependencies

```
for (int i = 0; i < m; i++)
        a[i] = a[i + k] * c;
```

  - Example: Vectorization safe only if k >= number of elements of a that can fit in a vector register

```
#pragma ivdep
for (int i = 0; i < m; i++)
        a[i] = a[i + k] * c;
```

  - “#pragma ivdep” => If unsure about a dependency, assume it doesn’t happen

ICC specific extensions (2)

- **#pragma vector**
  - Extra directives to control vectorization
  - Possible arguments:
    - aligned/unaligned: Use aligned/unaligned instructions, ignoring what you know about the alignment of the data structures involved
    - temporal/nontemporal: Controls use of streaming stores
    - always: “Even if you don’t think vectorization will help performance, do it anyway”
    - assert: If vectorization not possible, throw compiler error

ICC specific extensions (3)

- **#pragma simd**
  - The nuclear option – enforce vectorization if at all possible
  - Now in GCC 4.9 as part of Cilk+
  - Lots of knobs and options.. Too many to describe here
    - vectorlength: Specify safe length of vector registers (in number of elements) – useful to specify there's a dependency between i-th and i-5-th elements
    - reduction

```
int x = 0;
#pragma simd reduction(+:x)
for(int i = 0; I < N; i++) {
    x += a[i];
}
```

Vc library by Matthias Kretz

- **Explicit vectorization: Fine-tuned control, ability to directly manipulate vector registers and elements**
- **Requires C++11**
- **No overhead**
- **Operator overloading for easy to understand and intuitive code**
- **Vector types: double_v, float_v, int_v ..**
  - Size depends on target instruction set .. for AVX, double_v contains 4 elements, 2 for SSE
- Convenience containers to handle alignment and looping for us

```
Vc::Memory<double_v, 100> mem;
...
for(int i = 0; i < mem.vectorsCount(); i++) {
    double_v vec = mem.vector(i);
    ... handle vec ...
}
```

- **http://code.compeng.uni-frankfurt.de/projects/Vc**
- **Very good experience with it**

Vc library (2)

What about branches?

- Neat syntax to only operate on specific elements of the vector

```
double_v a, c;
...
c(a > 0) = a + ...;
```

only touch those elements for which a > 0
No, not a function call, just C++ magic

- double_m, int_m, float_m … essentially arrays of booleans with as many number of elements as their corresponding double_v, int_v, float_v …

Intel Cilk+ extensions

- **Not just a library: language extensions to the C/C++ syntax**
  - Requires support from the compiler
    - Starting from ICC 12.0
    - Added to GCC 4.9
    - Clang support coming up
- **Still need to care about aliasing, alignment, data layout …**
  - No escape
  - Likely to improve in the future

CERN openlab

Cool features

Simple assignments	`A[:] = 5;`
Range assignment	`A[0:7] = 5;`
Assignment w/ stride	`A[0:5:2] = 5;`
Increments	`A[:] = B[:] + 5;`
2D arrays	`C[:][:] = 12;`
	`C[0:5:2][:] = 12;`
Function calls	`func (A[:]);`
	`A[:] = pow(c, B[:])`
	operators
Conditions	`if (5 == a[:])`
	`  results[:] = „y"`
	`else`
	`  results[:] = „n"`
Reductions	`__sec_reduce_mul (A[:])`
Gather	`C[:] = A[B[:]]`
Scatter	`A[B[:]] = C[:]`

OpenMP 4.0

- **An update to the widely supported standard**
  - #pragma omp simd
    - Similar, but not identical to Cilk+ #pragma simd
    - Some options
      - reduction
      - safelen
  - Function vectorization
    - Declares function to be used inside SIMD loop

OpenMP 4.0 (2)

- Two versions of the function generated: A normal and one that takes vectors as parameters

```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}

void example() {
    #pragma omp simd
    for( … ) {
        c[i] = min(a[i], b[i]);
    }
}
```

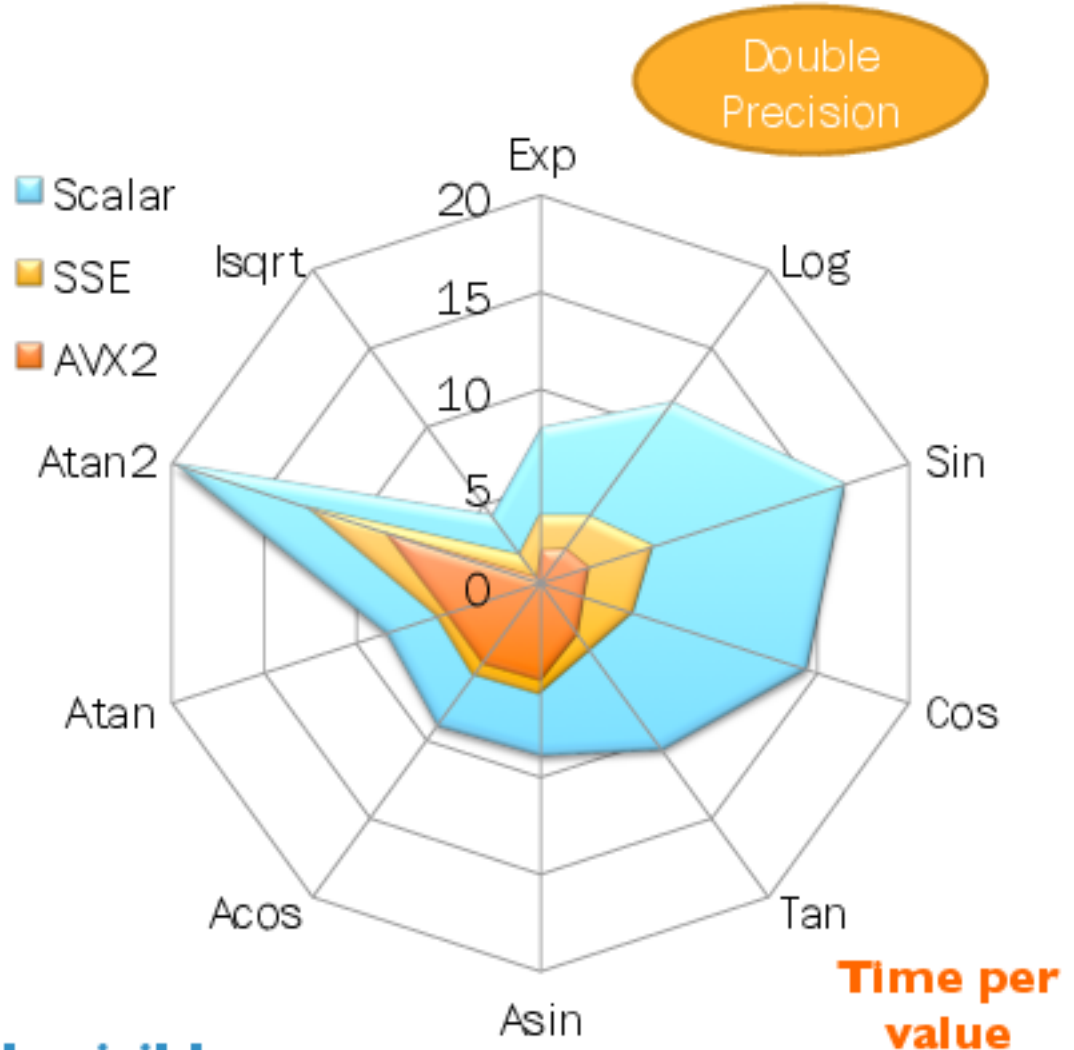
Math libraries

- **In addition, many high performance libraries use vectorization internally**
  - VDT: glibm replacement – exp, log, sin, cos, tan, …
    - Open source
    - Vectorization provides excellent performance – 5x speedup compared to glibm not uncommon
    - Very accurate
      - No error in most cases (double precision)
      - 1-2 bits of error in a few cases

Speed: VDT Vectorisation

Fnc.	Scalar	SSE	AVX2
Exp	8	3.5	1.7
Log	11.5	4.3	2.2
Sin	16.5	6.2	2.6
Cos	14.4	5.1	2.3
Tan	10.6	4.4	3.2
Asin	8.9	5.8	5
Acos	9.1	5.9	5.1
Atan	8.4	5.6	5.1
Atan2	19.9	12.7	8.4
Isqrt	4.3	1.8	0.4

Time in ns per value calculated



- **Effect of vectorisation clearly visible**

Slide courtesy of Danilo Piparo (PH-SFT, CERN)

Math libraries (2)

- **Intel MKL**
  - Extensive: Covers many domains
    - Linear algebra, FFTs, vector math, statistics …
  - New in latest beta: Features and optimizations specific to small matrices (inlining, batch processing)

Thanks

georgios.bitzes@cern.ch