

Vectorisation

James Briggs

¹COSMOS DiRAC

April 28, 2015

Session Plan

- 1 Overview
- 2 Implicit Vectorisation
- 3 Explicit Vectorisation
- 4 Data Alignment
- 5 Summary

Section 1

Overview

What is SIMD?

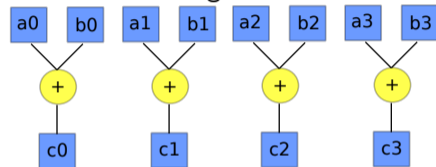
Scalar Code

- Executes one element at a time.

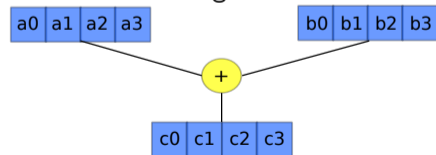
Vector Code

- Executes on multiple elements at a time in hardware.
- **Single Instruction Multiple Data.**






Scalar Processing:



Vector Processing:







A Brief History

- Pentium (1993):
32 bit: 
- MMX (1997):
64 bit: 
- Streaming SIMD Extensions (SSE in 1999,..., SSE4.2 in 2008):
128 bit: 
- Advanced Vector Extensions (AVX in 2011, AVX2 in 2013):
256 bit: 
- Intel MIC Architecture (Intel Xeon Phi in 2012):
512 bit: 

Why you should care about SIMD (1/2)

- Big potential performance **speed-ups** per core.

E.g. for **Double Precision FP** vector width vs theoretical speed-up over scalar:

- 128 bit:  **2**× potential for SSE.
- 256 bit:  **4**× potential for AVX.
- 256 bit:  **8**× potential for AVX2 (FMA).
- 512 bit:  **16**× potential for Xeon Phi (FMA).

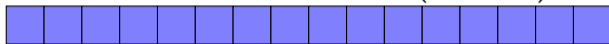
- Wider vectors allow for higher potential performance gains.
- Little programmer effort can often unlock hidden 2-8× in code!

Why you should care about SIMD (2/2)

The Future:

- Chip designers like SIMD – low cost, low power, big gains.
- Next Generation Intel Xeon and Xeon Phi (AVX-512):

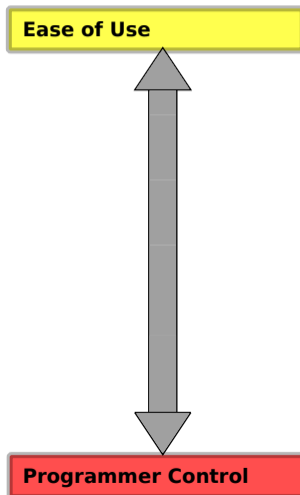
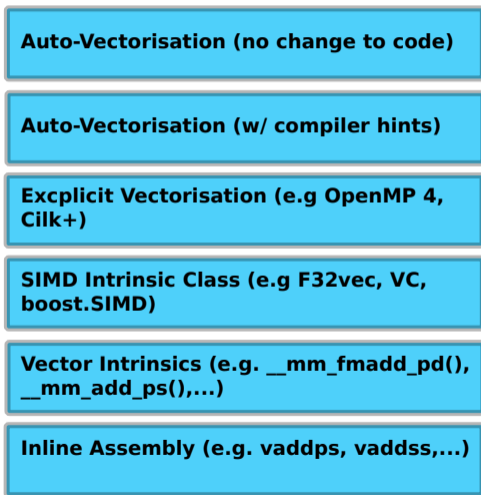
512 bit:



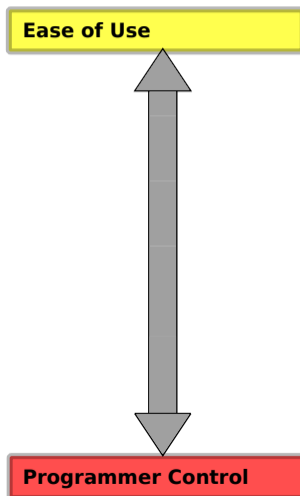
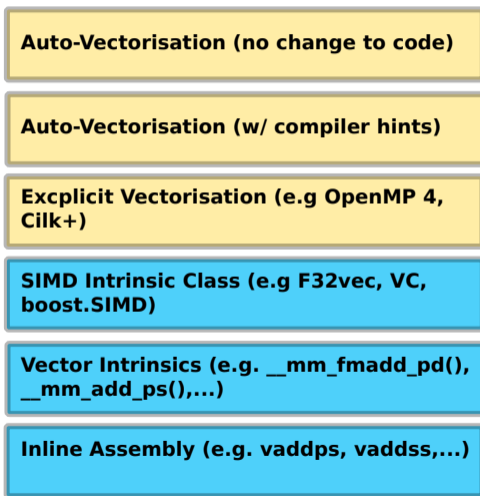
Not just Intel:

- ARM Neon - 128 bit SIMD.
- IBM Power8 - 128 bit (VMX)
- AMD Piledriver - 256 bit SIMD (AVX+FMA).

Many Ways to Vectorise



Many Ways to Vectorise



Section 2

Implicit Vectorisation

Auto-Vectorisation

- Compiler will analyse your loops and generate vectorised versions of them at the optimisation stage.
- Intel Compiler required flags:
Xeon: `-O2 -xHost`
Mic Native: `-O2 -mmic`
- On Intel use `qopt-report=[n]` to see if loop was auto-vectorised.
- Powerful, but the compiler cannot make unsafe assumptions.

Auto-Vectorisation

What does the compiler check for:

```
int *g_size;
void not_vectorisable(float *a, float *b, float *c, int *ind) {
    for (int i=0; i < *g_size; ++i) {
        int j = ind[i];
        c[j] = a[i] + b[i];
    }
}
```

- Is `*g_size` loop-invariant?
- Do `a`, `b`, and `c` point to different arrays? (Aliasing)
- Is `ind[i]` a one-to-one mapping?

Auto-Vectorisation

This will now auto-vectorise:

```
int *g_size;
void vectorisable(float * restrict a, float * restrict b, float *
  restrict c, int * restrict ind) {
  int n = *g_size;
  #pragma ivdep
  for (int i=0; i < n; ++i) {
    int j = ind[i];
    c[i] = a[i] + b[i];
  }
}
```

- Dereference *g_size outside of loop.
- restrict keyword tells compiler there is no aliasing.
- ivdep tells compiler there are no data dependencies between iterations.

Auto-Vectorisation Summary

- Minimal programmer effort. May require some compiler hints.
- Compiler can decide if scalar loop is more efficient.
- Powerful, but cannot make unsafe assumptions.
- Compiler will **always** choose correctness over performance.

Section 3

Explicit Vectorisation

Explicit Vectorisation

There are more involved methods for generating the code you want. These can give you:

- Fine-tuned performance.
- Advanced things the auto-vectoriser would never think of.
- Greater performance portability.

This comes at a price of increased programmer effort and possibly decreased portability.

Explicit Vectorisation

Compiler's Responsibilities

- Allow programmer to declare that code **can** and **should** be run in SIMD.
- Generate the code that the programmer asked for.

Programmer's Responsibilities

- Correctness (e.g. no dependencies or incorrect memory accesses)
- Efficiency (e.g. alignment, strided memory access)

Vectorise with OpenMP4.0 SIMD

- OpenMP 4.0 ratified July 2013.
- Specifications: <http://openmp.org/wp/openmp-specifications/>
- Industry standard.

- OpenMP 4.0 new feature: **SIMD pragmas!**

OpenMP – Pragma SIMD

- Pragma SIMD:

“The simd construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions).” - OpenMP 4.0 Spec.

- Syntax in C/C++:

```
#pragma omp simd [clause [, clause]...]  
for (int i=0; i<N; ++i)
```

- Syntax in Fortran:

```
!omp$ simd [clause [, clause]...]
```

OpenMP – Pragma SIMD Clauses

- `safelen(len)`
len must be a power of 2: The compiler can assume a vectorization for a vector length of len to be safe.
- `private(v1, v2, ...)`: Variables private to each lane.
- `linear(v1:step1, v2:step2, ...)`
For every iteration of original scalar loop v1 is incremented by step1,... etc. Therefore it is incremented by `step1 * vector length` for the vectorised loop.
- `reduction(operator:v1,v2,...)`:
Variables v1, v2,...etc. are reduction variables for operation operator.
- `collapse(n)`: Combine nested loops.
- `aligned(v1:base,v2:base,...)`: Tell compiler variables v1, v2,... are aligned.

OpenMP – SIMD Example 1

The old example that wouldn't auto-vectorise will do so now with SIMD:

```
int *g_size;
void vectorisable(float *a, float *b, float *c, int *ind) {
    #pragma omp simd
    for (int i=0; i < *g_size; ++i) {
        int j = ind[i];
        c[j] = a[i] + b[i];
    }
}
```

- The programmer **asserts** that there is no aliasing or loop variance.
- Explicit SIMD lets you express what you want, but correctness is your responsibility.

OpenMP – SIMD Example 2

An example of SIMD reduction:

```
int *g_size;
void vec_reduce(float *a, float *b, float *c) {
    float sum=0;
    #pragma omp simd reduction(+:sum)
    for (int i=0; i < *g_size; ++i) {
        int j = ind[i];
        c[j] = a[i] + b[i];
        sum += c[j];
    }
}
```

- sum should be treated as a reduction.

OpenMP – SIMD Example 3

An example of SIMD reduction with `linear` clause.

```
float sum = 0.0f;
float *p = a;
int step = 4;
#pragma omp simd reduction(+:sum) linear(p:step)
for (int i = 0; i < N; ++i) {
    sum += *p;
    p += step;
}
```

- `linear` clause tells the compiler that `p` has a linear relationship w.r.t the iterations space. i.e. it is computable from the loop index – $p_i = p_0 + i * \text{step}$.
- It also means that `p` is **SIMD lane private**.
- Its initial value is the value before the loop.
- After the loop `p` is set to the value it was in the sequentially last iteration.

SIMD Enabled Functions

- SIMD-enabled functions allow user defined functions to be vectorised when they are called from within vectorised loops.
- The vector declaration and associated modifying clauses specify the vector and scalar nature of the function arguments.
- Syntax C/C++:

```
#pragma omp declare simd [clause [,clause]...]  
function definition or declaration
```

- Syntax Fortran:

```
!$omp declare simd(proc-name) [clause [[,] clause] ...]  
function definition or declaration
```


SIMD-Enabled Function Clauses

- `simdlen(len)`
len must be a power of 2: generate a function that works for this vector length.
- `linear(v1:step1, v2:step2, ...)`
For every iteration of original scalar loop v1 is incremented by step1,... etc.
Therefore it is incremented by `step1 * vector length` for the vectorised loop.
- `uniform(a1,a2,...)`
Arguments a1, a2,... etc are not treated as vectors (constant values across SIMD lanes).
- `inbranch, notinbranch`: SIMD-enabled function called only inside branches or never.
- `aligned(v1:base,v2:base,...)`: Tell compiler variables v1, v2,... are aligned.

SIMD-Enabled Functions – Example

- Write a function for one element and add `pragma` as follows:

```
#pragma omp declare simd
float foo(float a, float b, float c, float d) {
    return a*b + c*d;
}
```

- You can call the scalar version as per usual:

```
e = foo(a, b, c, d);
```

- Call vectorised version in a SIMD loop:

```
#pragma omp simd
for (i=0; i < n; ++i) {
    E[i] = foo(A[i], B[i], C[i], D[i]);
}
```

SIMD-Enabled Functions – Recommendations

- SIMD-enabled functions still incur **overhead**.
- **Inlining is always better, if possible.**

Explicit Vectorisation – CilkPlus Array Notation

- An extension to C/C++.
- Perform operations on sections of arrays in parallel.

- Example, vector addition:

```
A[:] = B[:] + C[:];
```

- Looks like matlab/numpy/fortran,... but in C/C++!

Explicit Vectorisation – CilkPlus Array Notation

- Syntax:

```
A[:]  
A[start_index : length]  
A[start_index : length : stride]
```

- Use “:” for all elements
- “length” specifies the number of elements of a subset. **N.B. Not like F90.**
- “stride” is the distance between elements for subset.

Explicit Vectorisation – CilkPlus Array Notation

- Array notation also works with SIMD-enabled functions:

```
A[:] = mysimdfn(B[:], C[:]);
```

- Reductions on vectors done via predefined functions e.g.:

```
__sec_reduce_add , __sec_reduce_mul ,  
__sec_reduce_all_zero , __sec_reduce_all_nonzero ,  
__sec_reduce_max , __sec_reduce_min , ...
```

Array Notation Performance Issues

Long Form

```
C[0:N] = A[0:N] + B[0:N];  
D[0:N] = C[0:N] * C[0:N];
```

Short Form

```
for (i=0; i<N; i+=V) {  
    C[i:V] = A[i:V] + B[i:V];  
    D[i:V] = C[i:V] * C[i:V];  
}
```

- Long form is more elegant, but the short form will actually have better performance.
- If we expand the expressions back into for loops:
 - For large N , the long form will kick C out of cache. No reuse in next loop.
 - For appropriate V in the short form C can even be kept in registers.
- This is applicable for Fortran as well as Cilk Plus.

CilkPlus Availability

The following support Cilk Plus array notation (as well as its other features):

- GNU GCC 4.9+:
 - Enable with `-fcilkplus`
- clang/LLVM 3.5:
 - Not official branch yet but development branch exists:
<http://cilkplus.github.io/>
 - Enable with `-fcilkplus`
- Intel C/C++ compiler since version 12.0.

Implicit vs Explicit Vectorisation

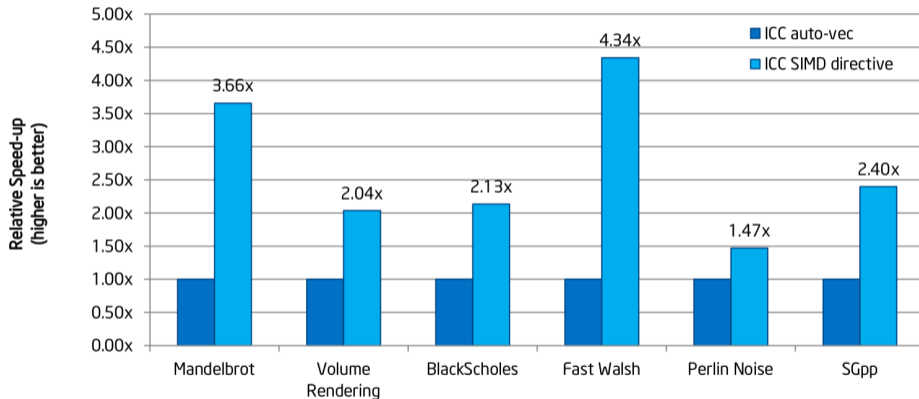
Implicit

- Automatic dependency analysis (e.g. reductions).
- Recognises idioms with data dependencies.
- Non-inline functions are scalar.
- Limited support for outer-loop vectorisation (possible in `-O3`).
- Relies on the compiler's ability to recognise patterns/idioms it knows how to vectorise.

Explicit

- No dependency analysis (e.g. reductions declared **explicitly**).
- Recognises idioms without data dependencies.
- Non-inline functions can be vectorised.
- Outer loops can be vectorised.
- May be more cross-compiler portable.

Implicit vs Explicit Vectorisation

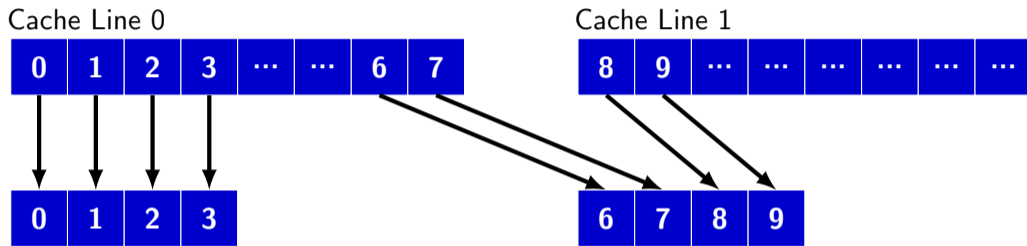


M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell, "Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP", pages 59-72, Rome, Italy, June 2012. LNCS 7312.

Section 4

Data Alignment

Data Alignment – Why it Matters



Aligned Load

- Address is aligned.
- One cache line.
- One instruction.
- 2-version vector/remainder.

Unaligned Load

- Address is not aligned.
- Potentially multiple cache lines.
- Potentially multiple instructions.
- 3-version peel/vector/remainder.

Data Alignment – Workflow

- 1 Align your data.
- 2 Access your memory in an aligned way.
- 3 Tell the compiler the data is aligned.

1. Align Your Data

- Automatic / free-store arrays in C/C++:

```
float a[1024] __attribute__((aligned(64)));
```

- Heap arrays in C/C++:

```
float *a = _mm_malloc(1024*sizeof(*a), 64); // on Intel/GNU  
_mm_free(a); // need this to free!
```

(For non-Intel there is also `posix_memalign` and `aligned_alloc` (C11)).

- In Fortran:

prettyc

```
real :: A(1024)  
!dir$ attributes align: 64 :: A  
real, allocatable :: B(512)  
!dir$ attributes align: 64 :: B
```

2. Access Memory in Aligned Way

- Example:

```
float a[N] __attribute__((aligned(64)));  
...  
for (int i=0; i<N; ++i)  
    a[i] = ...;
```

- Starting from an aligned boundary e.g. $a[0]$, $a[16]$, ...

3. Tell the Compiler

In C/C++:

- `#pragma vector aligned`
- `#pragma omp simd
aligned(p:64)`
- `__assume_aligned(p, 16)`
- `__assume(i%16==0)`

In Fortran:

- `!dir$ vector aligned`
- `!omp$ simd aligned(p:64)`
- `!dir$ assume_aligned(p, 16)`
- `!dir$ assume
(mod(i,16).eq.0)`

Alignment Example

```
float *a = _mm_malloc(n*sizeof(*a), 64);  
float *b = _mm_malloc(n*sizeof(*b), 64);  
float *c = _mm_malloc(n*sizeof(*c), 64);  
  
#pragma omp simd aligned(a:64,b:64,c:64)  
for (int i=0; i<n; ++i) {  
    a[i] = b[i] + c[i];  
}
```

Aligning Multi-dimensional Arrays 1/2

- Consider a 15×15 sized array of doubles. If we do:

```
double* a = _mm_malloc(15*15*sizeof(*a), 64);
```

- `a[0]` **is** aligned.
- `a[i*15+0]` for $i > 0$ are **not** aligned.
- The following may seg-fault:

```
for (int i=0; i<n; ++i) {  
    #pragma omp simd aligned(a:64)  
    for (int j=0; j<n; ++j) {  
        b[j] += a[i*n+j];  
    }  
}
```

Aligning Multi-dimensional Arrays 2/2

- We need add padding to every row of the array so each row starts on a 64 byte boundary.
- For 15×15 we should alloc 15×16 .

- **Useful code:**

```
int n_pad = (n+7) & ~7;
double* a = _mm_malloc(n*n_pad*sizeof(*a), 64);
```

- The following is now valid:

```
for (int i=0; i<n; ++i) {
    __assume(n_pad % 8 == 0);
    #pragma omp simd aligned(a:64)
    for (int j=0; j<n; ++j) {
        b[j] += a[i*n_pad+j];
    }
}
```

Section 5

Summary

Summary

- What we have learned:
 - Why vectorisation is important.
 - How the vector units on modern processors can provide big speed-ups with often small effort.
 - Auto-vectorisation in modern compilers.
 - Explicit vectorisation with OpenMP4.0, and array notation.
 - SIMD-Enabled functions.
 - How to align data and why it helps SIMD performance.