Making Use of SIMD Vectorisation to Improve Code Performance

Compute Ontario Colloquium

James Willis (SciNet)

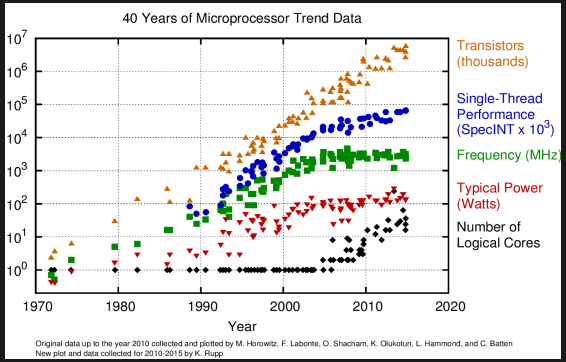
February 15, 2023

Outline

- Motivation
- What is Vectorisation?
- Addition Example
- Vectorisation Pitfalls
- Analysis Tools
- Vector Intrinsics

End of Moore's Law

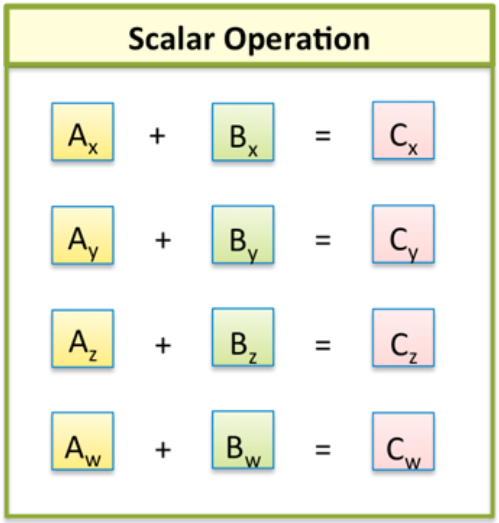
- CPU clock rates peaking
- No. of cores per chip increasing
- New AMD EPYC Bergamo CPU can have up to 128 cores!
- Need to parallelise applications to get anywhere near peak theoretical performance
- We also need to make use of SIMD vector units inside CPUs
- Vector units can give up to **8x** (AVX/AVX2) or **16x** (AVX512) **speed-up** improvements to code!

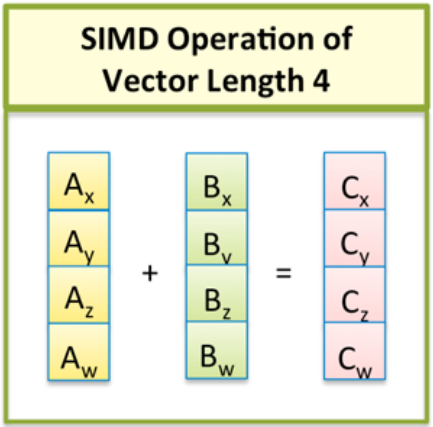


What is SIMD Vectorisation?

- SIMD: **S**ingle **I**nstruction **M**ultiple **D**ata
- Allows us to perform one operation (add/substract/multiply/divide etc.) on multiple data at the same time
- So what would previously involve 8 separate *scalar* instructions on the CPU, can now be done with 1 AVX *vector* instruction
- Resulting in a 8x speed increase

What is SIMD Vectorisation?

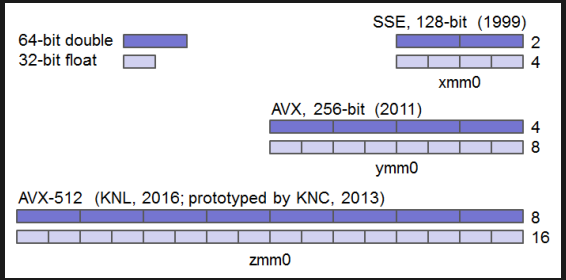




Intel® Architecture currently has SIMD operations of vector length 4, 8, 16

Vector Lengths

- Vector lengths are set by the size of the vector *registers* on the CPU:
  - 128-bit register SSE instructions (`xmm`)
  - 256-bit register AVX/AVX2 instructions (`ymm`)
  - 512-bit register AVX512 instructions (`zmm`)
- For example a 512-bit register can perform operations on either:
  - 16x 32-bit numbers, e.g. floats; or
  - 8x 64-bit numbers, e.g. doubles



- Lets look at a simple example adding two arrays together

Addition Example

- Imagine we want to add two `float` (32-bit) arrays `a` and `b` and store the result in `c`
- Declare arrays with the same size as the vector length, 4 in the case of SSE (4x 32-bit)
- Initialise with values
- Add elements `a` and `b` and store result in `c`

add.c

```
#define VEC_LENGTH 4

float a[VEC_LENGTH], b[VEC_LENGTH],
      c[VEC_LENGTH];

for(int i=0; i<VEC_LENGTH; i++) {
    a[i] = i;
    b[i] = i + 1;
}

for(int i=0; i<VEC_LENGTH; i++) {
    c[i] = a[i] + b[i];
}
```

Addition Example - Compilation

- *Auto-vectorisation* is when a compiler unrolls a loop and generates vector instructions in its place
- Specific flags are needed to perform auto-vectorisation
- The code should also be compiled using the highest available instruction set on the CPU, `-xHost` (ICC) or `-march=native` (GCC)
  - Intel compiler vectorises at default optimisation `-O2` or higher:

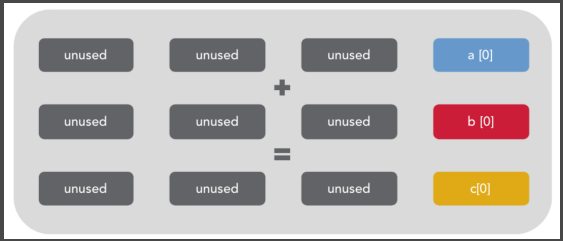
```
icc -O2 -xHost add.c
```

  - GNU compiler vectorises at `-O3` or with `-ftree-vectorize`:

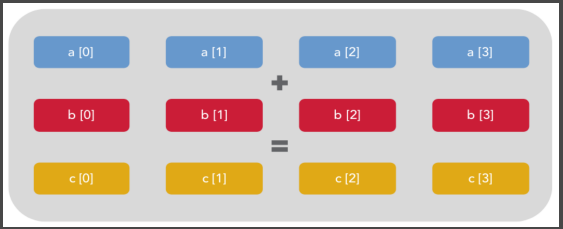
```
gcc -O2 -march=native -ftree-vectorize add.c
```

Addition Example - Scalar vs Vector Execution

Scalar



Vector



Vectorisation Pitfalls

- Not all loops are suitable for vectorisation
- Some loops may not vectorise efficiently
- Others may not vectorise at all
- Lets look at some of the main reasons behind this and how to remedy them

Loop Dependencies

- The biggest reason why a loop won't vectorise is a ***loop dependency***
- This happens when one iteration of a loop depends on another iteration
- For example, look at this loop:

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

- `a[i]` depends on the result of the previous iteration `a[i-1]`
- As we now know vectorisation performs 8 iterations in parallel there is no way for `a[i-1]` to be known before the ith iteration
- This is know as a ***read-after-write*** dependency, one variable is written in one iteration and read in a subsequent iteration

Memory Access Patterns

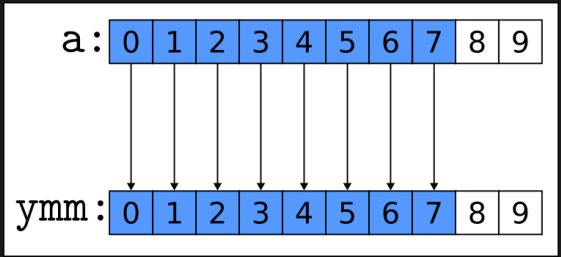
- Vectorisation is very sensitive to **memory access** patterns
- There are 3 types of memory access:
  - *Unit stride*
  - *Constant stride*
  - *Random access*
- Lets look at each case

Memory Access Patterns - Unit Stride

- Occurs when the data loaded is contiguous in memory:

```
for(int i=0; i<count; i++) {
    c[i] = a[i] + b[i];
}
```

- **Gives the best vectorisation performance**

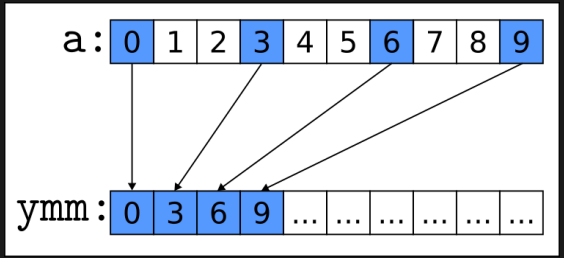


Memory Access Patterns - Constant Stride

- Occurs when the data loaded has a **fixed offset** in memory:

```
int offset = 3;
for(int i=0; i<count; i++) {
    c[i] = a[offset * i] + b[i];
}
```

- Lower vectorisation performance than **unit stride**

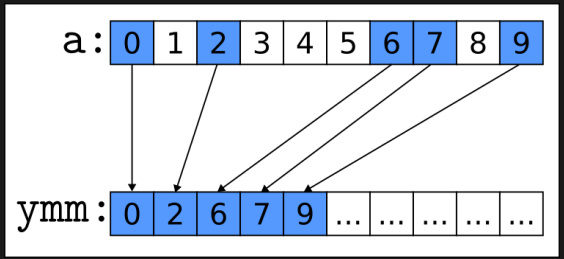


Memory Access Patterns - Random Access

- Occurs when the data is loaded from memory *randomly* or in an *unpredictable* way:

```
for(int i=0; i<count; i++) {
    c[i] = a[index[i]] + b[i];
}
```

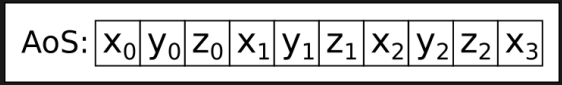
- Where the `index` array is populated at runtime, meaning the compiler can't optimise the memory load
- **Poorest** vectorisation performance of the 3 access patterns
- Always try to **avoid** this access pattern



SoA vs AoS

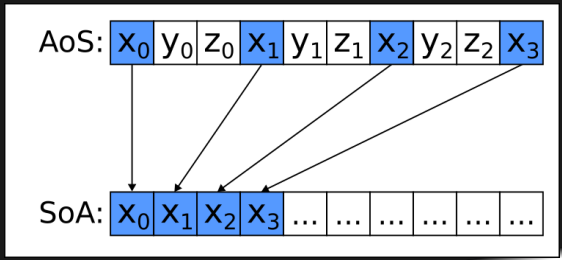
- Codes typically store their data in an **Array-Of-Structures** (AoS):

```
struct pos {
    float x; float y; float z;
}
struct pos part_pos[1000];
```



- One way to improve memory access is to make use of a data format called a **Structure-Of-Arrays** (SoA):

```
struct pos {
    float x[1000]; float y[1000];
    float z[1000];
}
struct pos part_pos;
```



- This converts the memory access from **constant** stride to **unit** stride

Conditional Statements

- If there is a branch in the loop:

```
for(int i=0; i<count; i++) {
    if(i % 2 == 0) {
        c[i] = a[i] + b[i];
    }
}
```

- Where every other element of `a` and `b` are added together
- This may prevent the compiler from vectorising the loop
- The compiler will attempt to vectorise using a logical mask
- It will perform addition of all elements and mask the result to get the even-indexed elements
- Even if the compiler succeeds in vectorising the loop it will be **less efficient** than a loop which does not contain a branch

Data Alignment

- Moving data to/from memory addresses which are **aligned** on specific byte boundaries is more efficient than **unaligned** addresses
- AVX512 instructions for example prefer data which is aligned on a **64-byte** boundary
- We can improve vectorisation performance by:
  - Aligning data with the `aligned(64)` *attribute* and
  - Giving a *hint* to the compiler with `__assume_aligned(a, 64)` in order to generate aligned load and store instructions

```
float a[COUNT] __attribute__((aligned(64)));
float b[COUNT] __attribute__((aligned(64)));
float c[COUNT] __attribute__((aligned(64)));

__assume_aligned(a, 64); __assume_aligned(b, 64); __assume_aligned(c, 64);
for(int i=0; i<COUNT; i++) {
    c[i] = a[i] + b[i];
}
```

Vectorisation Pragmas

- The compiler will **not** vectorise a loop if it thinks:
  - There is a data dependency; or
  - The loop will be faster executing with scalar instructions vs vector instructions
- You can override the compiler and **force** a loop to be vectorised with the use of a `#pragma`
- If you are confident there are no vector dependencies use `#pragma ivdep`:

```
#pragma ivdep
for(int i=0; i<count; i++) {
```

- **Beware, if there is an actual dependency the result will be incorrect!**
- `#pragma vector always` tells the compiler to vectorise a loop if it has no dependencies and ignore any cost metrics which may prevent vectorisation

Optimisation Report

- There are tools which can aid in code vectorisation
- Compilers have the ability to generate what is called an *optimisation report*
- These reports will show the **vectorisation eligibility** and **estimated vectorisation efficiency** of each loop
- Add `-qopt-report=5` to the compiler flag list for Intel and `-fopt-info-vec-all=gcc.optrpt` for GNU

Optimisation Report - Intel

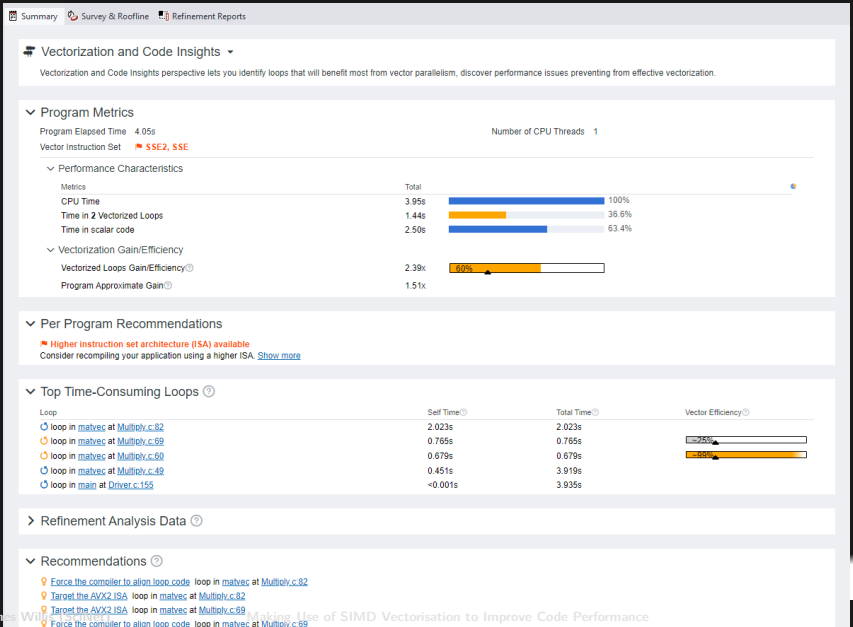
- Generated with: `icc -O2 -xHost add.c -qopt-report=5`

```
LOOP BEGIN at add.c(7,3) inlined into add.c(22,3)
   remark #15388: vectorization support: reference c[i] has aligned access [ add.c(22,13) ]
   remark #15388: vectorization support: reference a[i] has aligned access [ add.c(22,7) ]
   remark #15388: vectorization support: reference b[i] has aligned access [ add.c(22,10) ]
   remark #15305: vectorization support: vector length 8
   remark #15427: loop was completely unrolled
   remark #15399: vectorization support: unroll factor set to 10
   remark #15300: LOOP WAS VECTORIZED
   remark #26013: Compiler has chosen to target XMM/YMM vector.
                  Try using -qopt-zmm-usage=high to override
   remark #15448: unmasked aligned unit stride loads: 2
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 6
   remark #15477: vector cost: 0.620
   remark #15478: estimated potential speedup: 9.600
   remark #15488: --- end vector cost summary ---
LOOP END
```

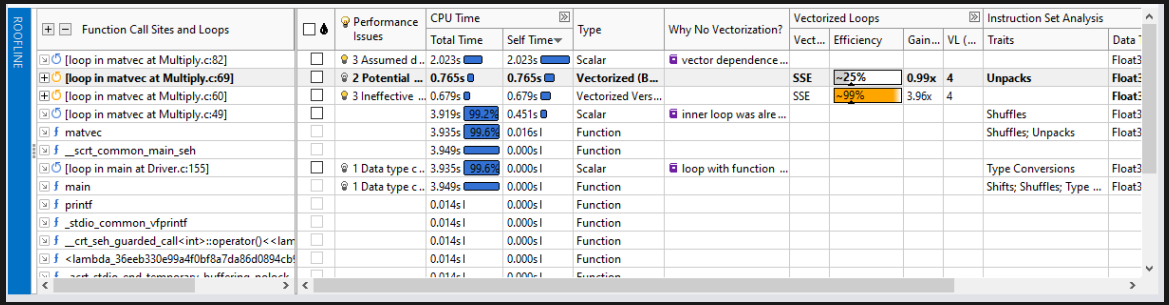
Intel Advisor

- Intel also provides an analysis tool called *Intel Advisor*
- GUI interface which profiles code and records metrics on code performance
- Great for analysing vectorisation efficiency of your code
- Provides hints and tips on how to vectorise loops and improve vectorisation efficiency
- Tutorial: www.intel.com/content/www/us/en/develop/documentation/get-started-with-advisor

Intel Advisor



Intel Advisor



Function Call Sites and Loops	Performance Issues	CPU Time: Total Time	CPU Time: Self Time	Type	Why No Vectorization?	Vectorized Loops: Vect...	Vectorized Loops: Efficiency	Vectorized Loops: Gain...	Vectorized Loops: VL (...	Instruction Set Analysis: Traits	Instruction Set Analysis: Data
[loop in matvec at Multiply.c:82]	3 Assumed d ...	2.023s	2.023s	Scalar	vector dependence ...						Float3
[loop in matvec at Multiply.c:69]	**2 Potential ...**	**0.765s**	**0.765s**	**Vectorized (B...**		**SSE**	~25%	**0.99x**	**4**	**Unpacks**	**Float3**
[loop in matvec at Multiply.c:60]	3 Ineffective ...	0.679s	0.679s	Vectorized Vers...		SSE	~99%	3.96x	4		**Float3**
[loop in matvec at Multiply.c:49]		3.919s (99.2%)	0.451s	Scalar	inner loop was alre ...					Shuffles	Float3
matvec		3.935s (99.6%)	0.016s	Function						Shuffles; Unpacks	Float3
__scrt_common_main_seh		3.949s	0.000s	Function							
[loop in main at Driver.c:155]	1 Data type c ...	3.935s (99.6%)	0.000s	Scalar	loop with function ...					Type Conversions	Float3
main	1 Data type c ...	3.949s	0.000s	Function						Shifts; Shuffles; Type ...	Float3
printf		0.014s	0.000s	Function							
_stdio_common_vfprintf		0.014s	0.000s	Function							
__crt_seh_guarded_call<int>::operator()<lam		0.014s	0.000s	Function							
<lambda_36eeb330e99a4f0bf8a7da86d0894cb		0.014s	0.000s	Function							

Vector Intrinsics

- Auto-vectorisation is the easiest form of vectorisation with little to no code changes
- However, it may not give the most **optimal** performance
- Vectorisation can also be achieved through the use of ***intrinsics***
- Vector intrinsics operate on vector registers directly in the source code
- Lowers **portability** of code
- Intel Intrinsics Guide: **www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html**

- Addition example with intrinsics:

```
for(int i=0; i<COUNT; i+=VEC_LENGTH) {
  // Load elements into vector registers
  __m256 v_a = _mm256_load_ps(&a[i]);
  __m256 v_b = _mm256_load_ps(&b[i]);

  // Add vectors together
  __m256 v_c = _mm256_add_ps(v_a, v_b);

  // Store result in c
  _mm256_store_ps(&c[i], v_c);
}
```

Caveat

- At longer instruction sets, specifically AVX512, the CPU may **downclock** if all cores are running
- This is because these instructions draw more power from the CPU
- Therefore the clock frequency will decrease to maintain the same power usage
- For example the Intel Xeon Gold 5115 CPU's frequency behaves as:

Mode	Base	Turbo Frequency/Active Cores									
		1	2	3	4	5	6	7	8	9	10
Normal	2,400 MHz	3,200 MHz	3,200 MHz	3,000 MHz	3,000 MHz	2,900 MHz	2,900 MHz	2,900 MHz	2,900 MHz	2,800 MHz	2,800 MHz
AVX2	2,000 MHz	3,100 MHz	3,100 MHz	2,900 MHz	2,900 MHz	2,600 MHz	2,600 MHz	2,600 MHz	2,600 MHz	2,400 MHz	2,400 MHz
AVX512	1,200 MHz	2,900 MHz	2,900 MHz	2,200 MHz	2,200 MHz	1,700 MHz	1,700 MHz	1,700 MHz	1,700 MHz	1,600 MHz	1,600 MHz

Caveat

- Take this into account when deciding which vector instruction set to use
- Benchmark your code with AVX/AVX2 and compare the results against AVX512
- To compile with a specific vector instruction set with the Intel compiler use:
  - `-xAVX` (AVX)
  - `-xCORE-AVX2` (AVX2)
  - `-xCORE-AVX512` (AVX512)

Summary

- Vectorisation can lead to **16x** improvement in peformance
- Compilers auto-vectorise loops if no dependencies are present
- Vectorisation efficiency can be increased by:
  - Optimising memory access patterns
  - Aligning data on a 64-byte boundary
  - Making use of SoA instead of AoS
- Code analysis tools such as Intel Advisor and optimisation reports
- Good introduction to vectorisation: https://www.intel.com/content/dam/develop/external/us/en/documents/vectorization-performance-quantifi-755040.pdf
- Please email any questions to: **support@scinet.utoronto.ca**

References

- Slide 3: www.karlrupp.net/2015/06
- Slide 5: https://medium.com/pixelstab/the-simd-experience-data-parallelism-on-my-game-engine-13711054ed6e
- Slide 6: https://cvw.cac.cornell.edu/vector/hw_registers
- Slide 9: https://www.intel.com/content/dam/develop/external/us/en/documents/vectorization-performance-quantifi-755040.pdf
- Slide 22: https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-advisor/top/discover-where-vectorization-will-pay-off.html#discover-where-vectorization-will-pay-off
- Slide 26: https://en.wikichip.org/wiki/intel/xeon_gold/5115