DEMYSTIFYING VECTORIZATION

Webinar

Colfax International — *colfaxresearch.com*

June 2017

DISCLAIMER

While best efforts have been used in preparing this training, Colfax International makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The publisher shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. No warranty may be created or extended by sales representatives or written sales materials.

§2. VECTOR OPERATIONS

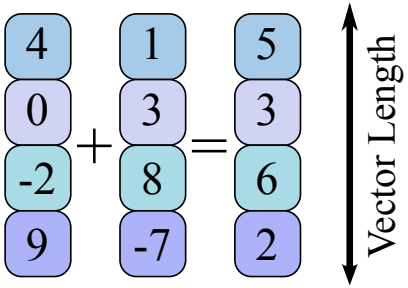
SHORT VECTOR SUPPORT

Vector instructions – one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.

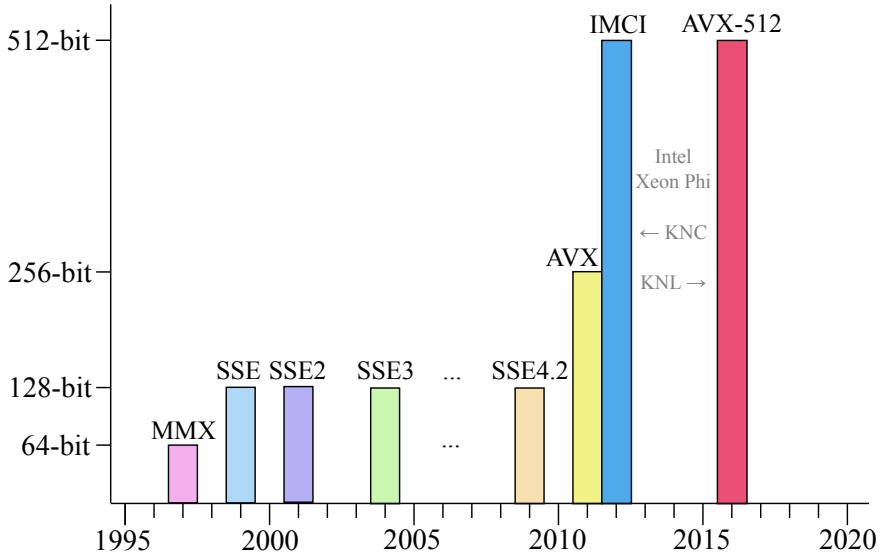
Scalar Instructions

$$4 + 1 = 5$$
$$0 + 3 = 3$$
$$-2 + 8 = 6$$
$$9 + -7 = 2$$

Vector Instructions

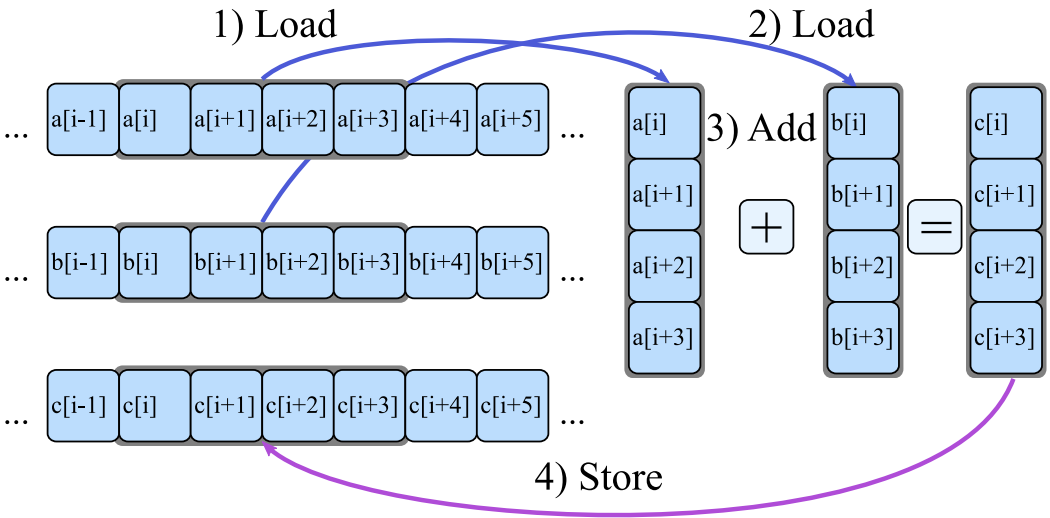


INSTRUCTION SETS IN INTEL ARCHITECTURE



§3. VECTORIZING YOUR CODE

WORKFLOW OF VECTOR COMPUTATION



USING VECTOR INSTRUCTIONS: TWO APPROACHES

Automatic Vectorization →

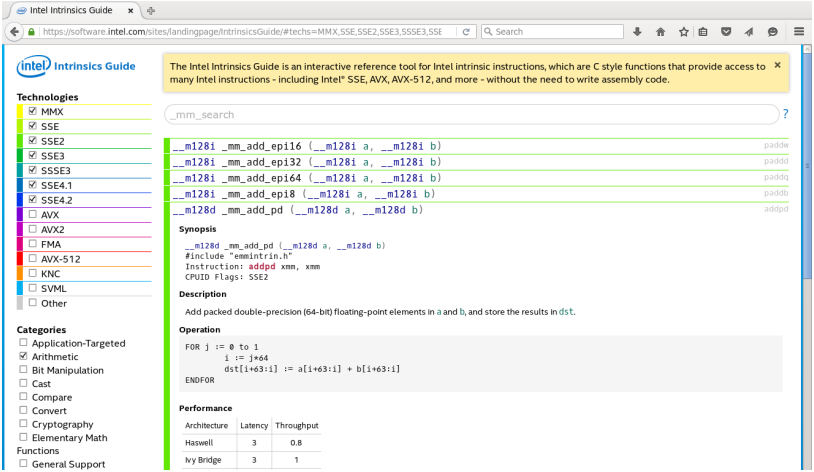
```
1 double A[vec_width], B[vec_width];
2 // ...
3 for(int i = 0; i < vec_width; i++)
4   A[i]+=B[i];
```

```
1 double A[8], B[8];
2 __m512d A_v = _mm512_load_pd(A);
3 __m512d B_v = _mm512_load_pd(B);
4 A_v = _mm512_add_pd(A_v,B_v);
5 _mm512_store_pd(A, A_v);
```

← Explicit Vectorization

INTEL INTRINSICS GUIDE

https://software.intel.com/sites/landingpage/IntrinsicsGuide



§4. AUTOMATIC VECTORIZATION

AUTOMATIC VECTORIZATION OF LOOPS

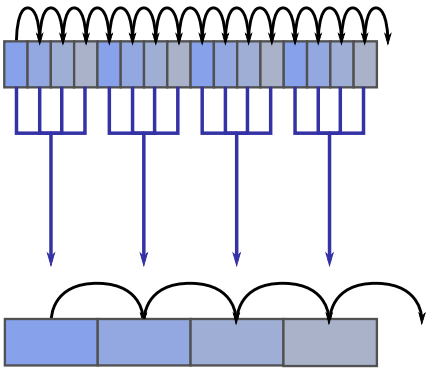
```cpp
#include <cstdio>

int main(){
  const int n=1024;
  int A[n] __attribute__((aligned(64)));
  int B[n] __attribute__((aligned(64)));

  for (int i = 0; i < n; i++)
    A[i] = B[i] = i;

  // This loop will be auto-vectorized
  for (int i = 0; i < n; i++)
    A[i] = A[i] + B[i];

  for (int i = 0; i < n; i++)
    printf("%2d %2d %2d\n",i,A[i],B[i]);
}
```

```
vega@lyra% icpc autovec.cc -qopt-report
vega@lyra% cat autovec.optrpt
...
LOOP BEGIN at autovec.cc(12,3)
remark #15399: vectorization support:
unroll factor set to 2 [autovec.cc(12,3)]
remark #15300: LOOP WAS VECTORIZED
[autovec.cc(12,3)]
LOOP END
...
vega@lyra% ./a.out
0  0  0
1  2  1
2  4  2
3  6  3
4  8  4
...
```

LIMITATIONS ON AUTOMATIC VECTORIZATION

- Innermost loops*
- Known number of iterations
- No vector dependence
- Functions must be SIMD-enabled

* `#pragma omp simd` to override



TARGETING A SPECIFIC INSTRUCTION SET

`-x[code]` to target specific processor architecture

`-ax[code]` for multi-architecture dispatch

`code`	Target architecture
MIC-AVX512	Intel Xeon Phi processors (KNL)
CORE-AVX512	Future Intel Xeon processors
CORE-AVX2	Intel Xeon processor E3/E5/E7 v3, v4 family
AVX	Intel Xeon processor E3/E5 and E3/E5/E7 v2 family
SSE4.2	Intel Xeon processor 55XX, 56XX, 75XX and E7 family
host	architecture on which the code is compiled

AUTO-VECTORIZED LOOPS MAY BE COMPLEX

```
1  for (int i = ii; i < ii + tileSize; i++) { // Auto-vectorized
2
3    // Newton's law of universal gravity
4    const float dx = particle.x[j] - particle.x[i]; // x[j] is a const
5    const float dy = particle.y[j] - particle.y[i]; // x[i] -> vector
6    const float dz = particle.z[j] - particle.z[i];
7    const float rr = 1.0f/sqrtf(dx*dx + dy*dy + dz*dz + softening);
8    const float drPowerN32  = rr*rr*rr;
9
10   // Calculate the net force
11   Fx[i-ii] += dx * drPowerN32;
12   Fy[i-ii] += dy * drPowerN32;
13   Fz[i-ii] += dz * drPowerN32;
14 }
```

§5. VECTOR DEPENDENCE

TRUE VECTOR DEPENDENCE

- True vector dependence – vectorization impossible:

```
1 for (int i = 1; i < n; i++)
2   a[i] += a[i-1]; // dependence on the previous element
```

- Safe to vectorize:

```
1 for (int i = 0; i < n-1; i++)
2   a[i] += a[i+1]; // no dependence on the previous element
```

- May be safe to vectorize:

```
1 for (int i = 16; i < n; i++)
2   a[i] += a[i-16]; // no dependence if vector length <=16
```

ASSUMED VECTOR DEPENDENCE

Not enough information to confirm or rule out vector dependence:

```
1 void AmbiguousFunction(int n, int *a, int *b) {
2   for (int i = 0; i < n; i++)
3     a[i] = b[i];
4 }
```

- If `a`, `b` are not aliased or `b>a`, then safe to vectorize
- If `a`, `b` are aliased (e.g., `b==a-1`), requires scalar computation

MULTIVERSIONING

```
user@host% icpc -c code.cc -qopt-report
user@host% cat code.optrpt
...
LOOP BEGIN at code.cc(4,1)
<Multiversioned v1>
   remark #25228: LOOP WAS VECTORIZED
LOOP END
...
LOOP BEGIN at code.cc(4,1)
<Multiversioned v2>
   remark #15304: loop was not vectorized: non-vectorizable loop instance ....
LOOP END
```

Pointers checked for aliasing *runtime* to choose code path.

POINTER DISAMBIGUATION

Prevent multiversioning or allow vectorization with a directive:

```
1 #pragma ivdep
2   for (int i = 0; i < n; i++)
3     // ...
```

```
user@host% icpc -c code.cc -qopt-report -qopt-report-phase:vec
user@host% cat vdep.optrpt
...
LOOP BEGIN at code.cc(4,1)
   remark #25228: LOOP WAS VECTORIZED
LOOP END
...
```

Alternative: keyword `restrict` – more fine-grained, weaker.

§6. GUIDED AUTOMATIC VECTORIZATION

VECTORIZE MORE LOOPS: `#pragma omp simd`

Used to “enforce vectorization of loops”, which includes:

- Loops with SIMD-enabled functions
- Second innermost loops
- Failed vectorization due to compiler decision
- Where guidance is required (vector length, reduction, etc.)

See OpenMP reference for syntax; `#pragma simd`

EXAMPLE FOR #pragma omp simd

```
1  const int N=128, T=4;
2  float A[N*N], B[N*N], C[T*T];
3
4  for (int jj = 0; jj < N; jj+=T) // Tile in j
5    for (int ii = 0; ii < N; ii+=T) // and tile in i
6  #pragma omp simd   // Vectorize outer loop
7    for (int k = 0; k < N; ++k) // long loop, vectorize it
8      for (int i = 0; i < T; i++) { // Loop between ii and ii+T
9        // Instead of a loop between jj and jj+T, unrolling that loop:
10       C[0*T + i] += A[(jj+0)*N + k]*B[(ii+i)*N + k];
11       C[1*T + i] += A[(jj+1)*N + k]*B[(ii+i)*N + k];
12       C[2*T + i] += A[(jj+2)*N + k]*B[(ii+i)*N + k];
13       C[3*T + i] += A[(jj+3)*N + k]*B[(ii+i)*N + k];
14 }
```

VECTORIZATION DIRECTIVES

- `#pragma omp simd`
- `#pragma vector always`
- `#pragma vector aligned | unaligned`
- `__assume_aligned` keyword
- `#pragma vector nontemporal | temporal`
- `#pragma novector`
- `#pragma ivdep`
- `restrict` qualifier and `-restrict` command-line argument
- `#pragma loop count`

§7. STRIP-MINING

STRIP-MINING FOR VECTORIZATION

- Programming technique that turns one loop into two nested loops.
- Used to expose vectorization opportunities.

Original:

```
1 for (int i = 0; i < n; i++) {
2   // ... do work
3 }
```

Strip-mined:

```
1 const int STRIP=1024;
2 const int nPrime = n - n%STRIP;
3 for (int ii=0; ii<nPrime; ii+=STRIP)
4   for (int i=ii; i<ii+STRIP; i++)
5     // ... do work
6
7 for (int i=nPrime; i<n; i++)
8   // ... do work
```

§8. INTEL MATH KERNEL LIBRARY (MKL)

INTEL MATH KERNEL LIBRARY COMPONENTS

Dense Linear Algebra	BLAS + PBLAS LAPACK + ScaLAPACK Extended eigensolver
Fourier Transform	Multi-threaded Cluster mode 1D and multi-dimensional
Statistics and Probability	Random number generators Convolution and correlation Summary statistics
Sparse Linear Algebra	SpBLAS Iterative, direct solvers Preconditioners
Numerical Analysis	Partial differential equations Nonlinear optimization Data fitting. Vector math.
Machine Learning	New DNN Primitives: Convolution. Inner product. ReLU, LRNC, etc.

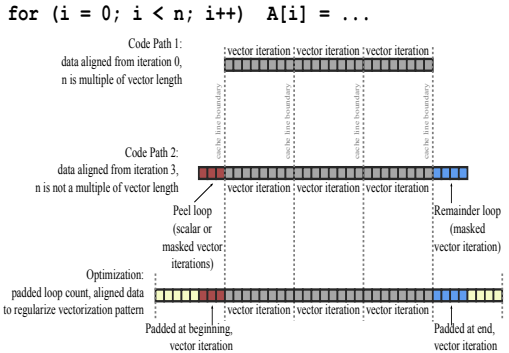
LICENSING OPTIONS FOR MKL

	Community License	Commercial License
Cost	Free	Per developer
Support	Community forum	Intel Premier Support
Use in products	Yes	Yes
Royalty-free	Yes	Yes

§9. MEMORY TRAFFIC

LOOP WAS VECTORIZED, NOW WHAT?

1. Unit-stride access
2. Data alignment
3. Container padding
4. Eliminate peel loops
5. Eliminate multiversioning
6. **Optimize data re-use in caches**



LOOP WAS VECTORIZED, NOW WHAT?

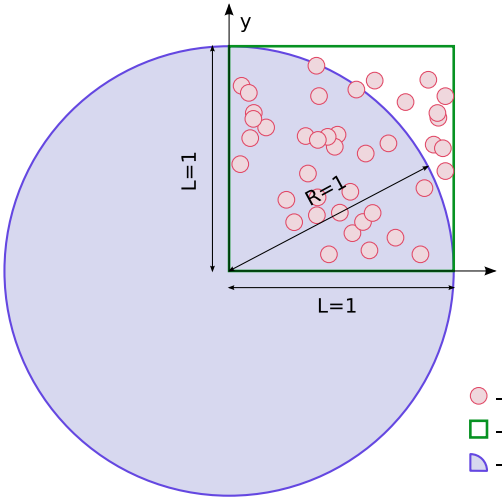
1. Unit-stride access
2. Data alignment
3. Container padding
4. Eliminate peel loops
5. Eliminate multiversioning
6. **Optimize data re-use in caches**

Vector Arithmetics is Cheap, Memory Access is Expensive

If you don't optimize cache usage, vectorization will not matter.
You will be bottlenecked by memory access.

§10. EXAMPLE: MONTE CARLO

COMPUTING PI



$$\pi \approx \frac{4\ (\#\ \text{inside quarter circle})}{\#\ \text{total}}$$

- Monte Carlo trial
- unit square area
- quarter circle area

ORIGINAL IMPLEMENTATION

```
1  float ComputePi(int n) {
2    int hits = 0;
3  #pragma omp parallel reduction(+: hits)
4    { VSLStreamStatePtr rnStream; // Random number generator
5      vslNewStream(&rnStream, VSL_BRNG_MT19937, omp_get_thread_num());
6      float x, y, r; // Temporary variables
7  #pragma omp for
8      for (int i = 0; i < n; i++) {
9        vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream, 1, &x, 0.0f, 1.0f);
10       vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream, 1, &y, 0.0f, 1.0f);
11       r = x*x + y*y;
12       if (r < 1.0f)
13         hits++;
14   } }
15   return 4.0f * (float)hits / (float)n;
16 }
```

SPLIT LOOPS

```
1   float *x = (float*) malloc(sizeof(float)*n);
2   float *y = (float*) malloc(sizeof(float)*n);
3 #pragma omp parallel reduction(+: hits)
4   { VSLStreamStatePtr rnStream; // Random number generator
5     vslNewStream(&rnStream, VSL_BRNG_MT19937, omp_get_thread_num());
6     float r; // Temporary variable
7 #pragma omp for
8     for (int i = 0; i < n; i++) {
9       vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream, 1, x+i, 0.0f, 1.0f);
10      vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream, 1, y+i, 0.0f, 1.0f);
11    }
12 #pragma omp for
13    for (int i = 0; i < n; i++) {
14      r = x[i]*x[i] + y[i]*y[i];
15      if (r < 1.0f)
16        hits++;
```

VECTOR RNG FROM INTEL MKL

```
#pragma omp for
  for (int block = 0; block < n_blocks; block++) {
    vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream,
                              block_size, x+block*block_size, 0.0f, 1.0f);
    vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream,
                              block_size, y+block*block_size, 0.0f, 1.0f);
  }
#pragma omp for
  for (int i = 0; i < n; i++) {
    r = x[i]*x[i] + y[i]*y[i];
// ...
```

STRIP-MINED FOR CACHE USAGE

```
int block_size = 256;
int n_blocks = n/block_size;
int n_padded = n_blocks*block_size;
#pragma omp parallel reduction(+: hits)
{
  float x[block_size], y[block_size];
  // ...
 #pragma omp for
  for (int block = 0; block < n_blocks; block++) {
   vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream, block_size, x, 0.0f, 1.0f);
   vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream, block_size, y, 0.0f, 1.0f);

    for (int i = 0; i < block_size; i++) {
      r = x[i]*x[i] + y[i]*y[i];
      if (r < 1.0f)
        hits++;
// ...
```

AVX2/AVX-512 INSTRUCTIONS

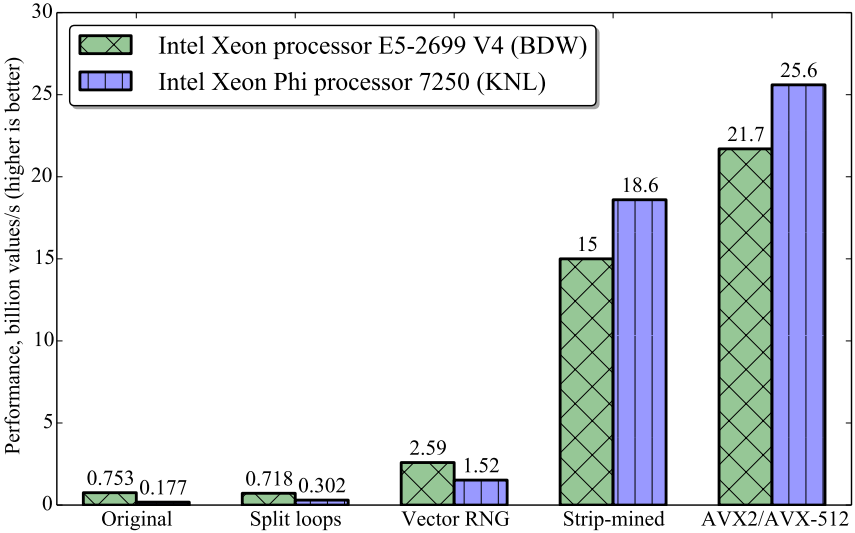
For Intel Xeon Phi processors:

```
GCCFLAGS=-c -fopenmp -O3 -std=c99 $(GCC_MKL_COMPILE) -mavx512f -fopt-info-vec
ICCFLAGS=-c -qopenmp -O3 -std=c99 -qopt-report=5  -xMIC-AVX512
```

For Intel Xeon processors (Haswell/Broadwell architecture):

```
GCCFLAGS=-c -fopenmp -O3 -std=c99 $(GCC_MKL_COMPILE) -mavx2 -fopt-info-vec
ICCFLAGS=-c -qopenmp -O3 -std=c99 -qopt-report=5  -xCORE-AVX2
```

PERFORMANCE



§11. LEARN MORE


THE "HOW" SERIES TRAINING
DEEP DIVE
WITH CODE MODERNIZATION EXPERTS
It's free
→ HOWSERIES.COM
*10x 2-hour sessions | 24-hour 2-weeks remote access to a system