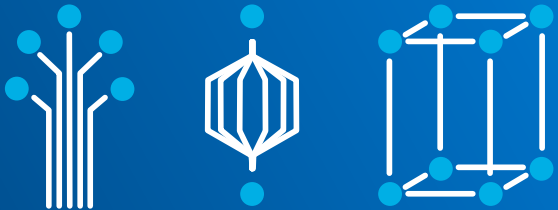
Fundamentals of Parallelism on Intel® Architecture

Week 2
Vectorization



COLFAX RESEARCH

May 2017

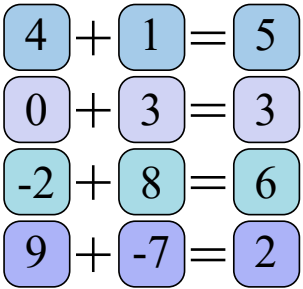
intel

§1. Vector Operations

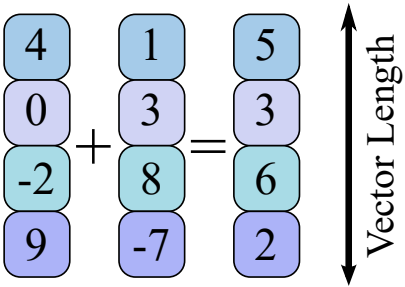
Short Vector Support

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

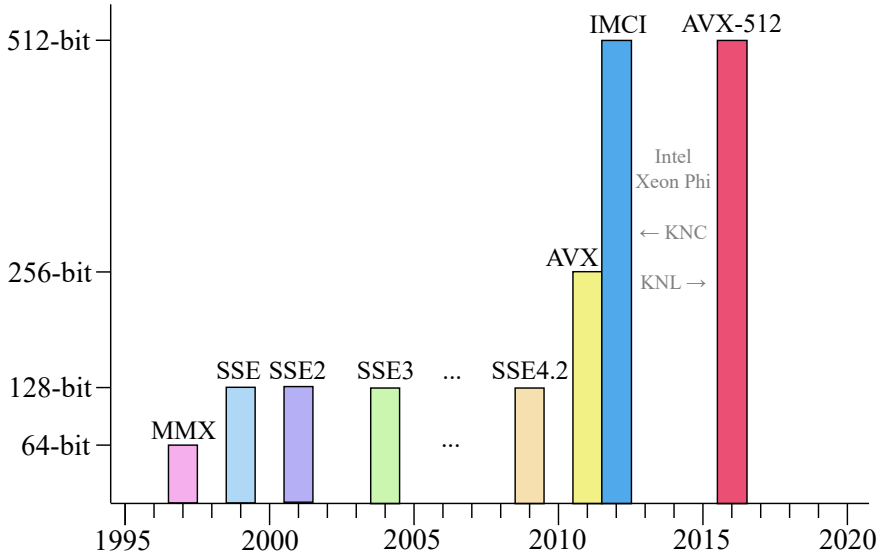
Scalar Instructions



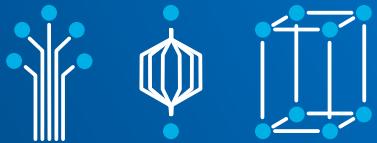
Vector Instructions



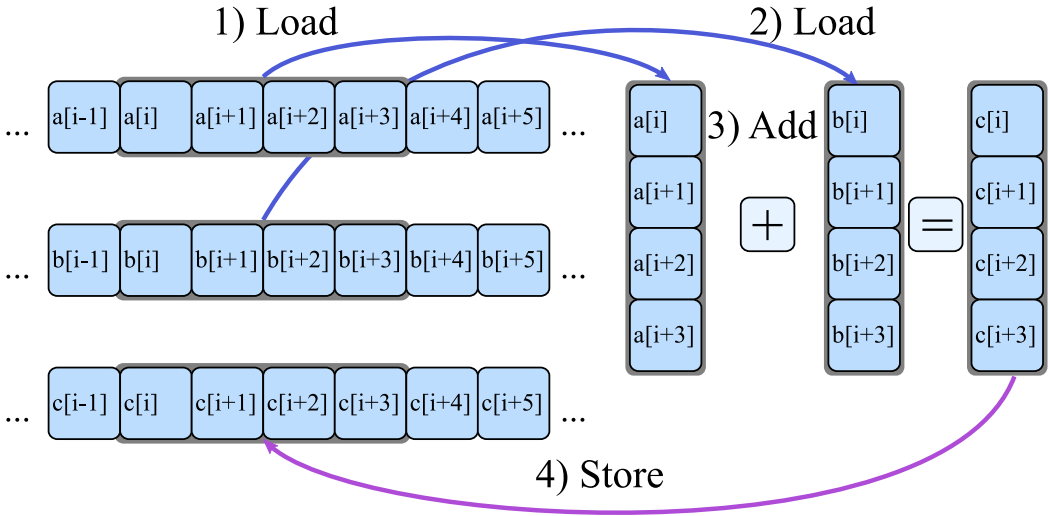
Instruction Sets in Intel Architecture



§2. Vectorizing your Code



Workflow of Vector Computation



Using Vector Instructions: Two Approaches

Automatic Vectorization →

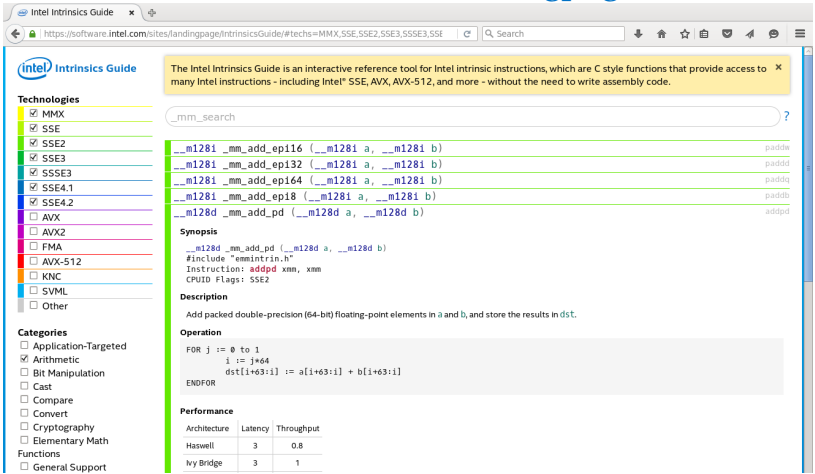
```
double A[vec_width], B[vec_width];
// ...
for(int i = 0; i < vec_width; i++)
  A[i]+=B[i];
```

```
double A[8], B[8];
__m512d A_v = _mm512_load_pd(A);
__m512d B_v = _mm512_load_pd(B);
A_v = _mm512_add_pd(A_v,B_v);
_mm512_store_pd(A, A_v);
```

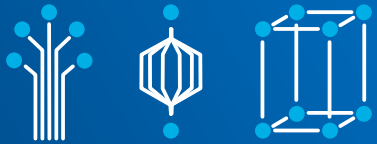
← Explicit Vectorization

Intel Intrinsics Guide

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



§3. Automatic Vectorization



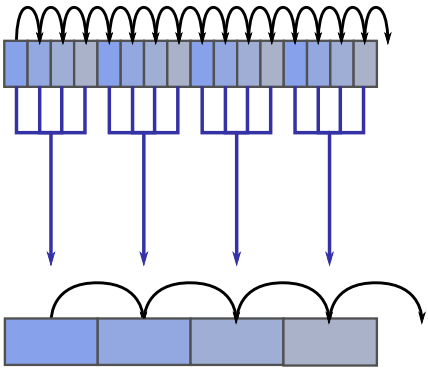
Automatic Vectorization of Loops

```
#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

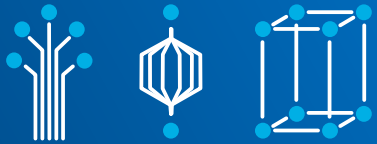
`-x[code]` to target specific processor architecture
`-ax[code]` for multi-architecture dispatch

`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 }
```


§4. 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`

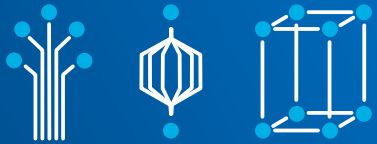
Example for #pragma omp simd

```
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`

§5. SIMD-enabled Functions



SIMD-Enabled Functions

Define function in one file (e.g., library), use in another

```
// Compiler will produce 3 versions:
#pragma omp declare simd
float my_simple_add(float x1, float x2){
  return x1 + x2;
}
```

```
// May be in a separate file
#pragma omp simd
for (int i = 0; i < N, ++i) {
  output[i] = my_simple_add(inputa[i], inputb[i]);
}
```


SIMD-enabled Functions May Be Complex

```
#pragma omp declare simd
float MyErfElemental(const float inx){
    const float x = fabsf(inx); // Absolute value (in each vector lane)
    const float p = 0.3275911f; // Constant parameter across vector lanes
    const float t = 1.0f/(1.0f+p*x); // Expression in each vector lanes
    const float l2e = 1.442695040f; // log2f(expf(1.0f))
    const float e = exp2f(-x*x*l2e); // Transcendental in each vector lane
    float res = -1.453152027f + 1.061405429f*t; // Computing a polynomial
    res = 1.421413741f + t*res;                 // in each vector lane
    res =-0.284496736f + t*res;
    res = 0.254829592f + t*res;
    res *= e;
    res = 1.0f - t*res; // Analytic approximation in each vector lane
    return copysignf(res, inx); // Copy sign in each vector lane
}
```


§6. 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

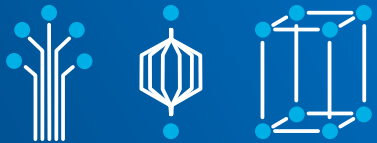
Prevent multiversioning or allow vectorization with a directive:

```
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.

§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. Example: Stencil Code

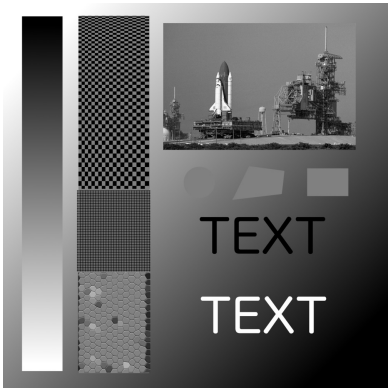
Stencil Operators

- Linear systems of equations
- Partial differential equations

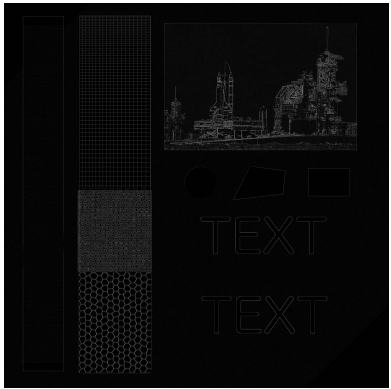
$$
\begin{aligned}
Q_{x,y} = \; & c_{00}P_{\mathrm{x-1,y-1}} + c_{01}P_{\mathrm{x,y-1}} + c_{02}P_{\mathrm{x+1,y-1}} + \\
& c_{10}P_{\mathrm{x-1,y}} + c_{11}P_{\mathrm{x,y}} + c_{12}P_{\mathrm{x+1,y}} + \\
& c_{20}P_{\mathrm{x-1,y+1}} + c_{21}P_{\mathrm{x,y+1}} + c_{22}P_{\mathrm{x+1,y+1}}
\end{aligned}
$$

Fluid dynamics, heat transfer, image processing (convolution matrix), cellular automata.

Edge Detection



$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \rightarrow$$



Basic Implementation

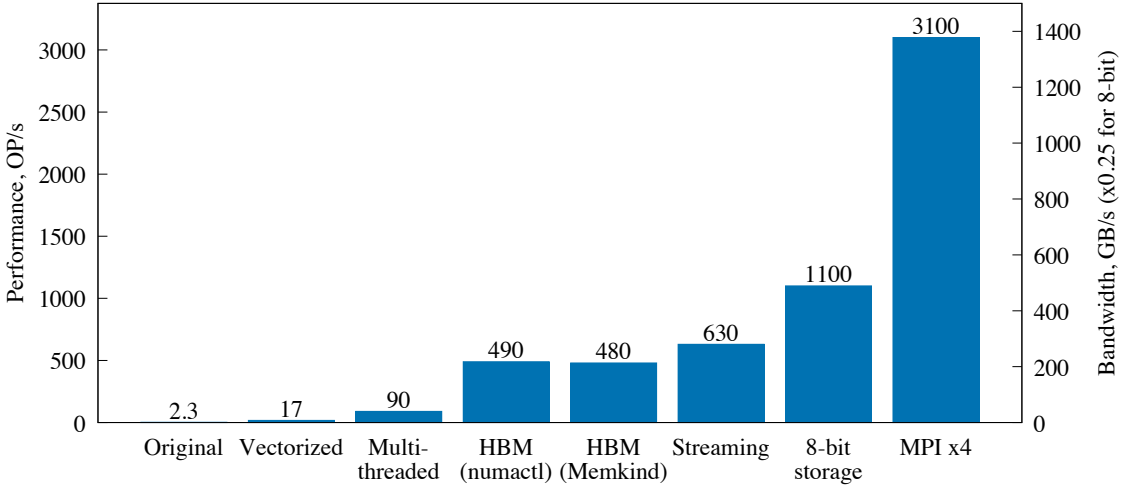
```
1  float *in, *out; // Input and output images
2  int width, height; // Dimensions of the input image
3
4  // Image convolution with the edge detection stencil kernel:
5  for (int i = 1; i < height-1; i++)
6   for (int j = 1; j < width-1; j++)
7    out[i*width + j] =
8    -in[(i-1)*width + j-1] -   in[(i-1)*width + j] - in[(i-1)*width + j+1]
9    -in[(i  )*width + j-1] + 8*in[(i  )*width + j] - in[(i  )*width + j+1]
10   -in[(i+1)*width + j-1] -   in[(i+1)*width + j] - in[(i+1)*width + j+1];
```


Vectorization

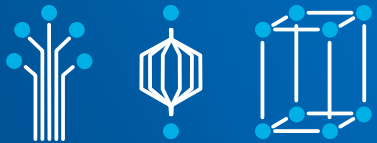
```
user@vega% icpc -c -qopt-report=5 -xMIC-AVX512 stencil.cc
```

```
1 for (int i = 1; i < height-1; i++)
2 #pragma omp simd
3  for (int j = 1; j < width-1; j++)
4   out[i*width + j] =
5   -in[(i-1)*width + j-1] -   in[(i-1)*width + j] - in[(i-1)*width + j+1]
6   -in[(i  )*width + j-1] + 8*in[(i  )*width + j] - in[(i  )*width + j+1]
7   -in[(i+1)*width + j-1] -   in[(i+1)*width + j] - in[(i+1)*width + j+1];
```

Performance



§9. Example: Numerical Integration



Midpoint Rectangle Method

$$I(a,b) = \int_0^a f(x)\mathrm{d}x \approx \sum_{i=0}^{n-1} f\left(x_{\mathrm{i}+\frac{1}{2}}\right)\Delta x,$$

where

$$\Delta x = \frac{a}{n}, \qquad x_{\mathrm{i}+\frac{1}{2}} = \left(i+\frac{1}{2}\right)\Delta x.$$

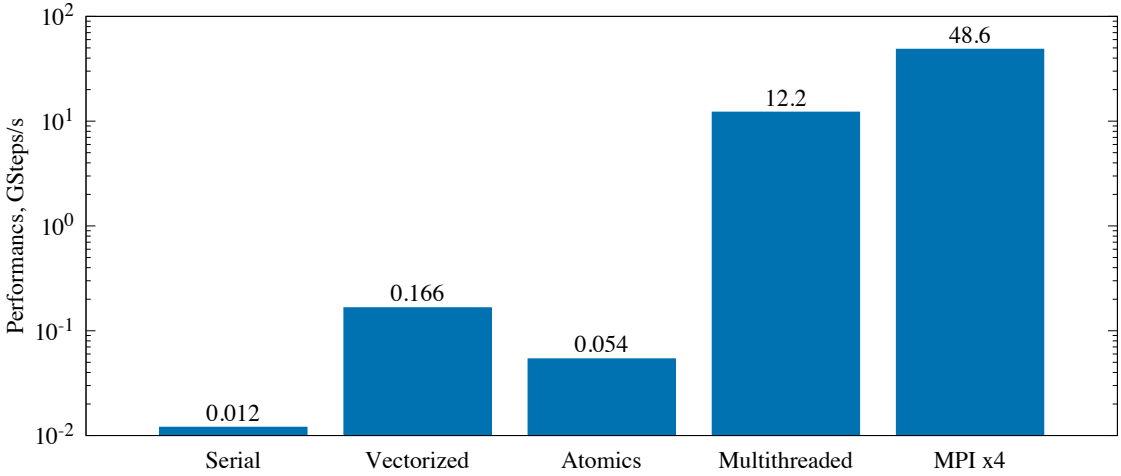
Scalar Implementation

```
1  double BlackBoxFunction(double x);
2  ...
3  const double dx = a/(double)n;
4  double integral = 0.0;
5
6  for (int i = 0; i < n; i++) {
7    const double xip12 = dx*((double)i + 0.5);
8    const double dI = BlackBoxFunction(xip12)*dx;
9    integral += dI;
10 }
```

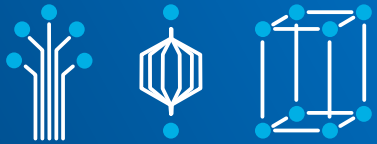

Vectorized Implementation

```
1  #pragma omp declare simd
2  double BlackBoxFunction(double x);
3  ...
4  const double dx = a/(double)n;
5  double integral = 0.0;
6
7  #pragma omp simd reduction(+: integral)
8  for (int i = 0; i < n; i++) {
9    const double xip12 = dx*((double)i + 0.5);
10   const double dI = BlackBoxFunction(xip12)*dx;
11   integral += dI;
12 }
```


Performance

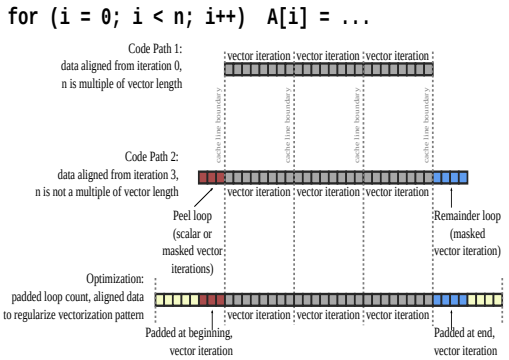


§10. Learn More



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.