COMP90025 Parallel and Multicore Computing

Vectorization

Lachlan Andrew

School of Computing and Information Systems
The University of Melbourne

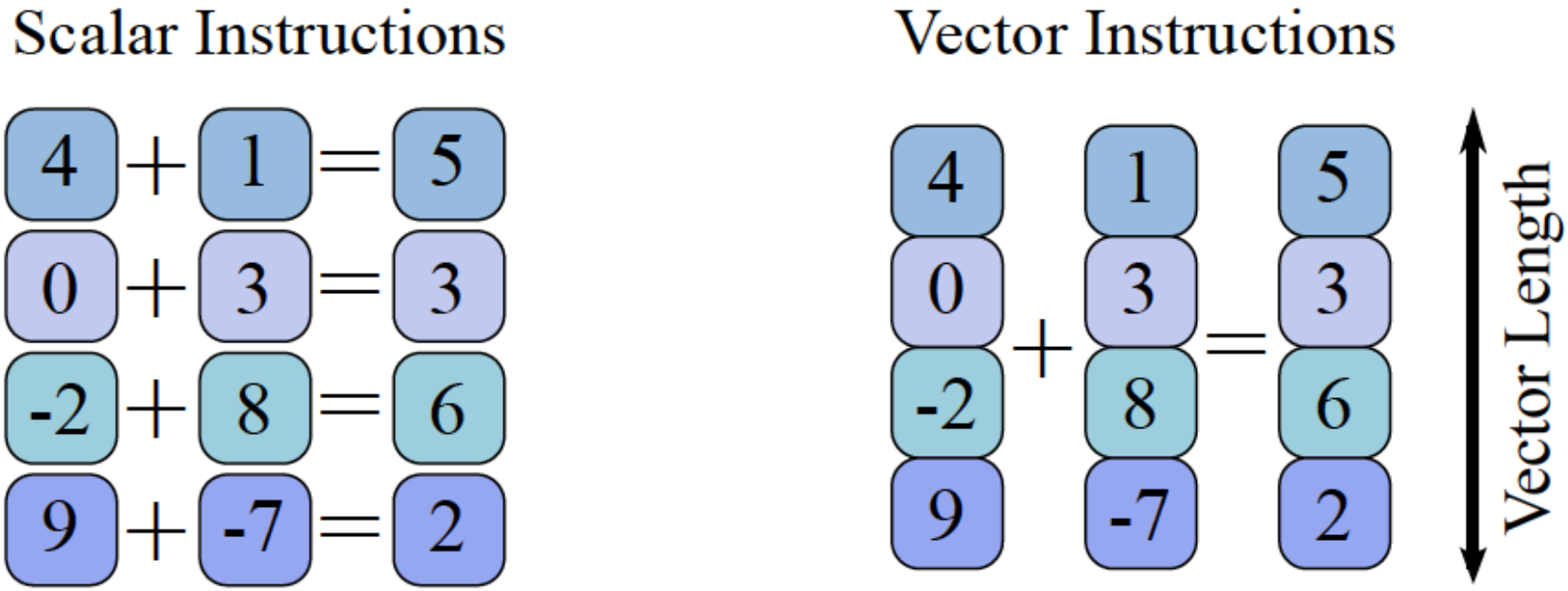
2019 Semester II

Agenda

- Vectorization
  - Automatic vectorization
  - SIMD construct
  - Data alignment

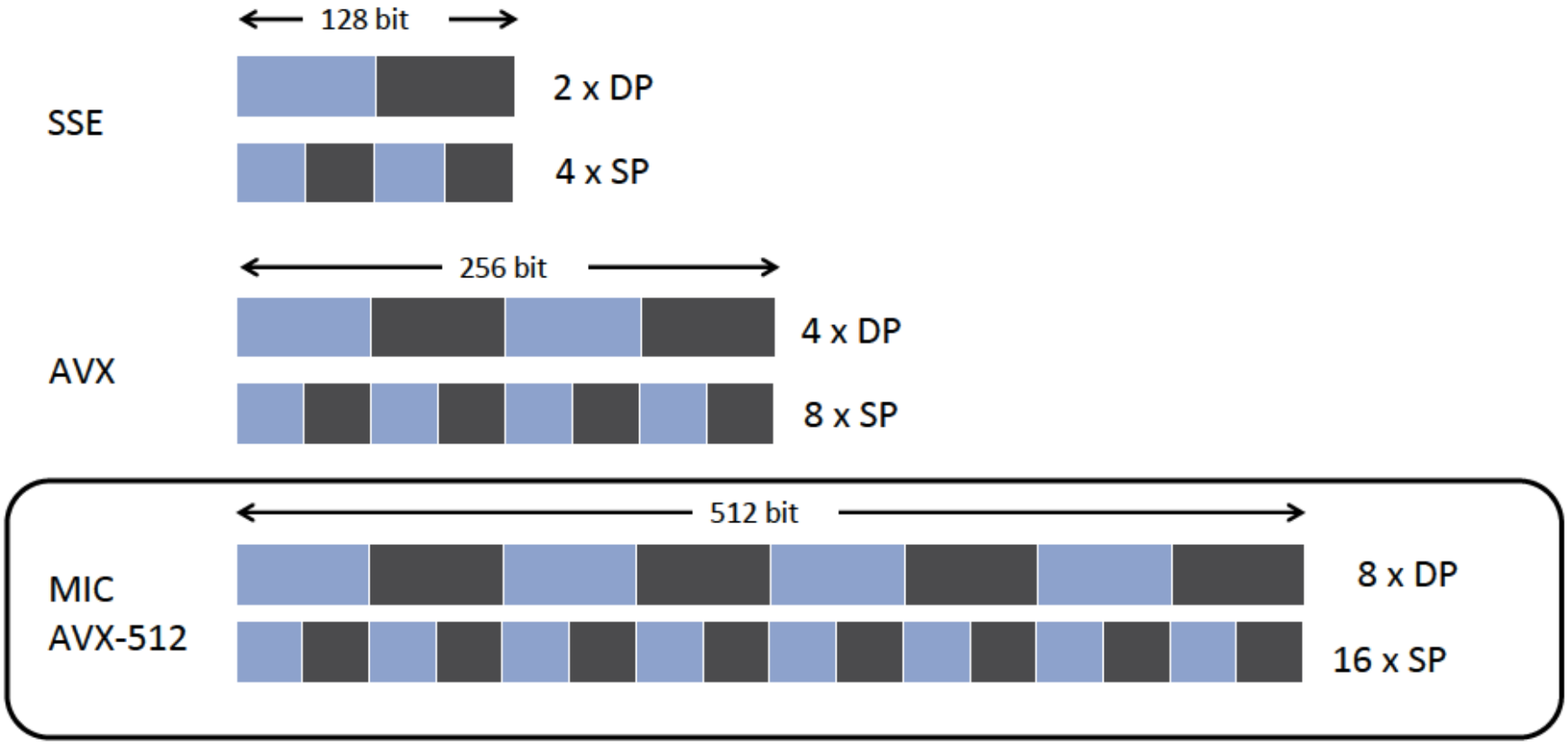
Vector Support

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



SIMD on Intel Architecture

Width of SIMD registers has been growing:



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]);
}
```

Automatic Vectorization of Loops

```
icpc autovec.cpp -qopt-report
cat autovec.optrpt
```

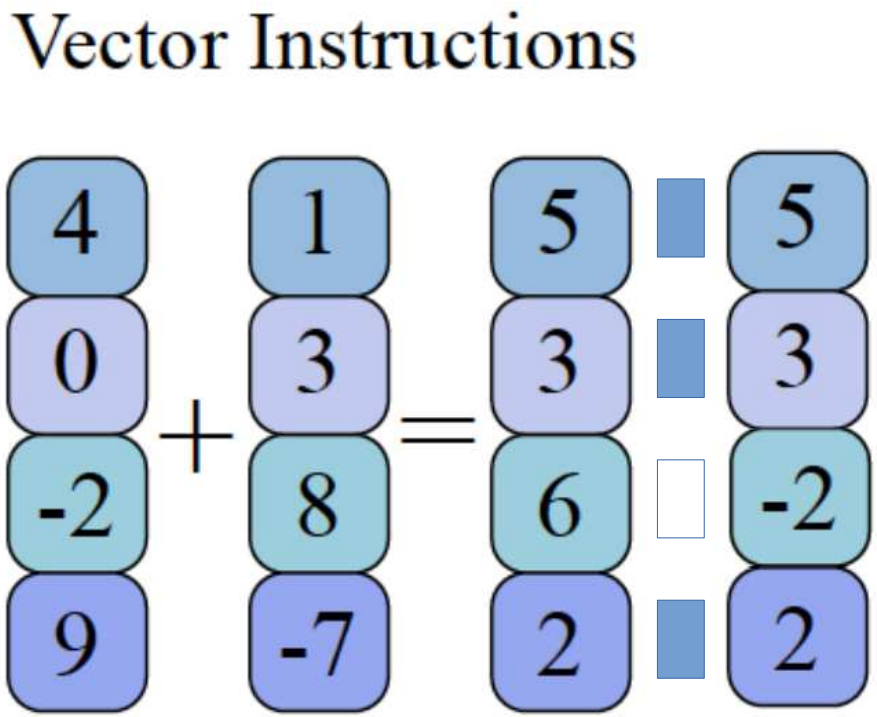
...
LOOP BEGIN at autovec.cpp(12,3)
remark#15300: LOOP WAS VECTORIZED
LOOP END
...

Automatic Vectorization of Loops with conditions

```
#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++)
    if (B[i] % 2 == 1)
        A[i] = A[i] + B[i];

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

Masked instructions

- Each SIMD PE computes the boolean condition → Mask
- Only perform operation on components for which mask is set
- Not all operations can use the mask
- Minimum requirement: Store
  - everyone computes result, only the chosen record it



Vectorizing with Unit-Stride Memory Access

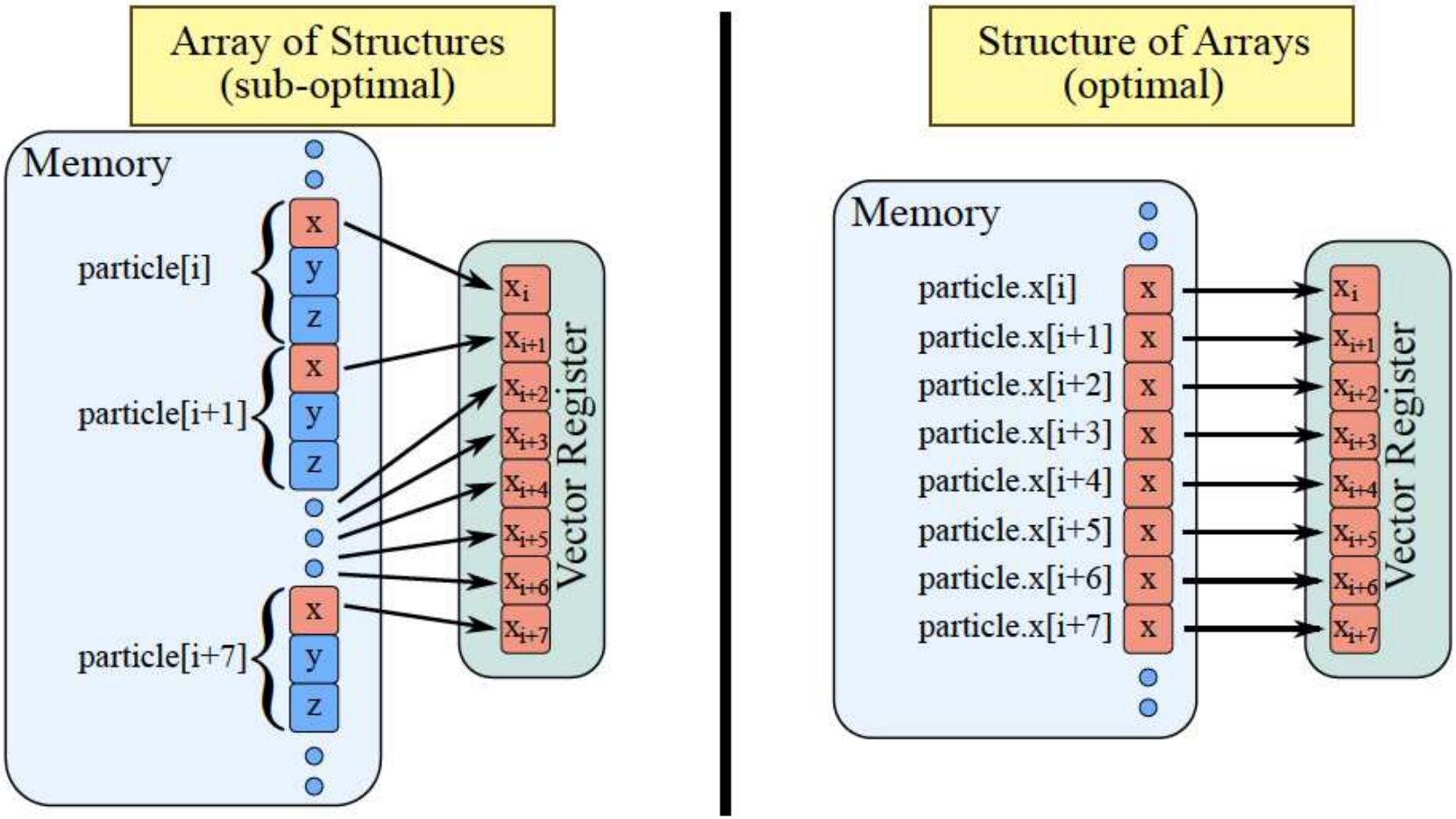
Before:

```
struct ParticleType {
 float x, y, z, vx, vy, vz;
 }; // ...
 const float dx = particle[j].x - particle[i].x;
 const float dy = particle[j].y - particle[i].y;
 const float dz = particle[j].z - particle[i].z;
```

After:

```
struct ParticleSet {
 float *x, *y, *z, *vx, *vy, *vz;
 }; // ...
 const float dx = particle.x[j] - particle.x[i];
 const float dy = particle.y[j] - particle.y[i];
 const float dz = particle.z[j] - particle.z[i];
```

Why AoS to SoA Conversion Helps: Unit Stride



Assumed Vector Dependence

True vector dependence vectorization impossible:

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

Assumed vector dependence compiler suspects dependence

```
void mycopy(int n, float* a, float* b) {
 for (int i=0; i<n; i++)
   a[i] = b[i];
}
```

Resolving Assumed Dependency

Restrict: Keyword indicating that there is no pointer aliasing (C++11)

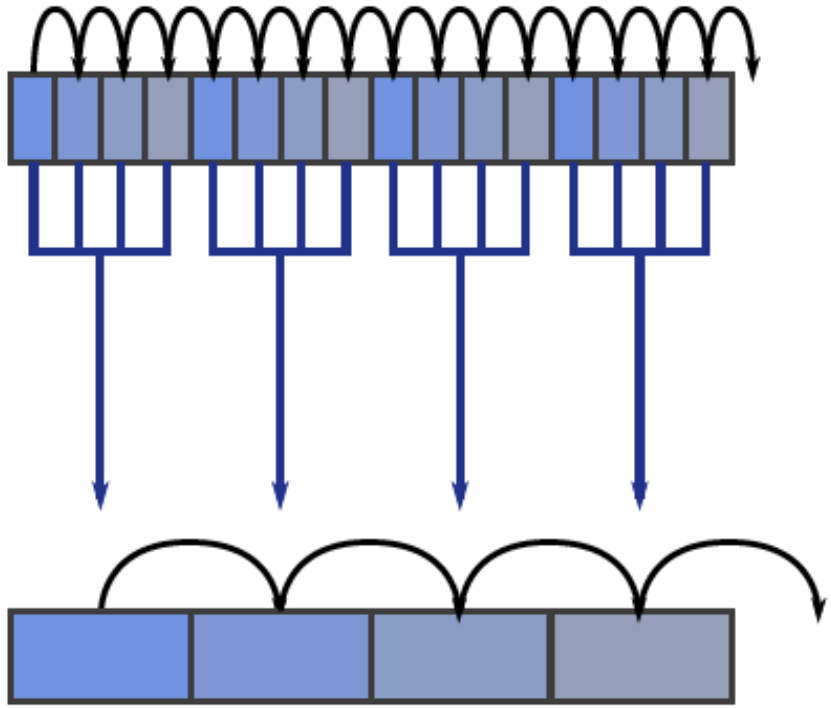
```
void mycopy(int n, float* restrict a,
            float* restrict b) {
 for (int i=0; i<n; i++)
 a[i] = b[i];
}
```

#pragma ivdep : ignores assumed dependency for a loop (Intel Compiler)

```
void mycopy(int n, float* a, float* b) {
 #pragma ivdep
 for (int i=0; i<n; i++)
 a[i] = b[i];
}
```

Limitations on Automatic Vectorization

- Only for-looops can be auto-vectorized. Number of iterations must be known.
- Memory access in the loop must have a regular pattern, ideally with unit stride.
- Non-standard loops that cannot be automatically vectorized:
  - calculations with vector dependence
  - while-loops, for-loops with undetermined number of iterations
  - outer loops (unless `#pragma simd` overrides this restriction)
  - loops with complex branches (i.e., if-conditions)



SIMD Loop Construct

Vectorize a loop nest

- Cut loop into chunks that fit a SIMD vector register
- No parallelization of the loop body

Syntax:

```
#pragma omp simd [clause[[,] clause], ...]
 for-loops
```

Simultaneous Threading and Vectorization

```
#pragma omp parallel for
for (int i = 0; i < n; i++)
// Thread parallelism in outer loop
#pragma simd
 for (int j = 0; j < m; j++)
 // Vectorization in inner loop
  DoSomeWork(A[i][j]);

#pragma omp parallel for simd
for (int i = 0; i < n; i++)
// If the problem is all data-parallel
 DoSomeWork(A[i]);
```

SIMD Function Vectorization

Declare one or more functions to be compiled for calls from a SIMD-parallel loop.

```
#pragma omp declare simd [clause[[,] clause], ...]
```

SIMD Function Vectorization

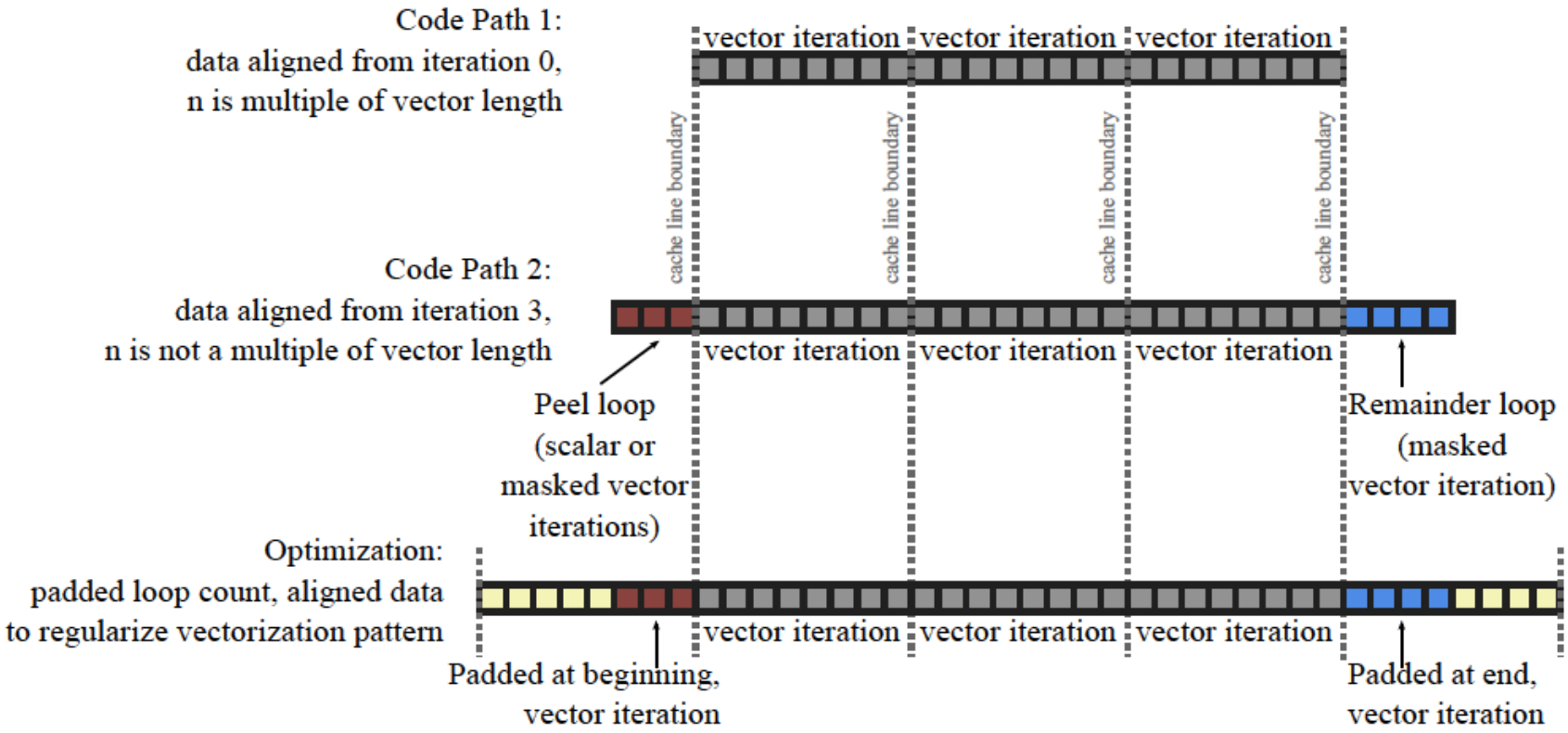
Declare one or more functions to be compiled for calls from a SIMD-parallel loop.

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

#pragma omp declare simd
float distsq(float x, float y) {
   return (x-y)*(x-y);
}

void example(){
#pragma omp parallel for simd
 for (i=0;i<N; i++){
   d[i] = min(distsq(a[i],b[i]),c[i]);
 }
```

Data Alignment

Compiler may implement peel and remainder loops:

```
for (i = 0; i < n; i++) A[i] = ...
```



Creating Aligned Data Containers

Data alignment on the stack

```
float A[n] __attribute__((aligned(64)));
// 64-byte alignment applied
```

Data alignment on the heap

```
float *A = (float*) _mm_malloc(sizeof(float)*n, 64)
```

A[0] is aligned on a 64-byte boundary.

Padding Multi-Dimensional Containers for Alignment

To use aligned instructions, you may need to pad inner dimension of multi-dimensional arrays to a multiple of 16 (in SP) or 8 (DP) elements.

No padding:

```
// A - matrix of size (n x n)
// n is not a multiple of 16
float* A =
  _mm_malloc(sizeof(float)*n*n, 64);

for (int i = 0; i < n; i++)
  // A[i*n + 0] may be unaligned
  for (int j = 0; j < n; j++)
    A[i*n + j] = ...
```

Padding Multi-Dimensional Containers for Alignment

Padding:

```
// ... Padding inner dimension
int lda=n + (16-n%16); // lda%16==0
float* A =
  _mm_malloc(sizeof(float)*n*lda, 64);

for (int i = 0; i < n; i++)
  // A[i*lda + 0] aligned for any i
  for (int j = 0; j < n; j++)
    A[i*lda + j] = ...
```

Reference

http://www.colfax-intl.com/nd/resources/slides.aspx
http://www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html/