Tutorial: Programming with Intel® Knights Landing (KNL) Manycore Processors

Dr. Sunghwan Choi

(E: sunghwanchoi@kisti.re.kr)

Dr. Chan Park

(E: iamparkchan@gmail.com)

Intel® Parallel Computing Center (IPCC)
Korea Institute of Science and Technology Information (KISTI)

KiSTi www.kisti.re.kr 한국과학기술정보연구원 Korea Institute of Science and Technology Information

국가슈퍼컴퓨팅연구소 National Institute of Supercomputing and Networking

IPCC KISTI

Contents

Sunghwan Choi

- Summary of features of Knights Landing (15m)
- Performance optimization
  - matrix-matrix multiplication (25m)
  - Sparse vector-dense vector dot (15m)
  - Histogram (15m)

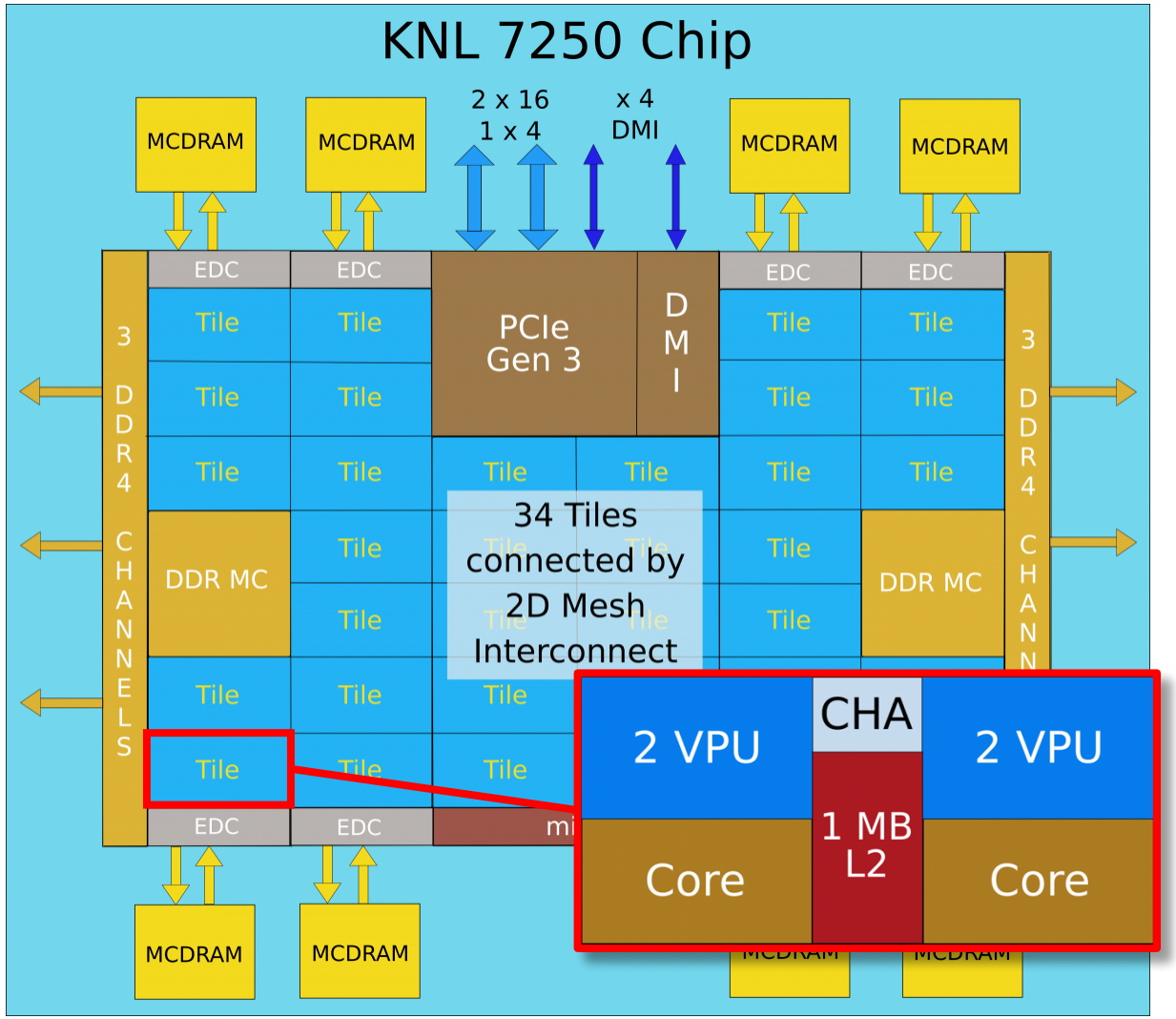
Chan Park

- Vectorization Practice
  - Loop Dependency: RAW vs WAR
  - Striding: SOA vs AOS
  - Application: Pi calculation

Connection / File transfer

- tar -xvzf WinterSchool2018_KNLtutorial.tar.gz

Overview of KNL architecture



- Up to 36 tiles (64 cores / 256 threads)
  - 2 cores / tile
  - Shared mesh connection
  - 1MB shared L2 cache / tile
  - 2 512-bit VPUs / core
  - Based on Intel Atom Silvermont architecture
- 2D mesh interconnect
- 2 DDR memory controllers
- 6 channels DDR4
  - Up to 90GB/s
- 16GB MCDRAM
  - 8 embedded DRAM controllers
  - Up to 450 GB/s

Summary of KNL Spec

1) More and Wider vector processing unit – **2 512-bit VPUs (AVX512) per core (4 threads)**

2) Higher memory bandwidth – **up to 450GB/s 16GB MCDRAM**

3) More cores / threads – **up to 64 cores / 256 threads (with 4-way Hyper-Threading)**

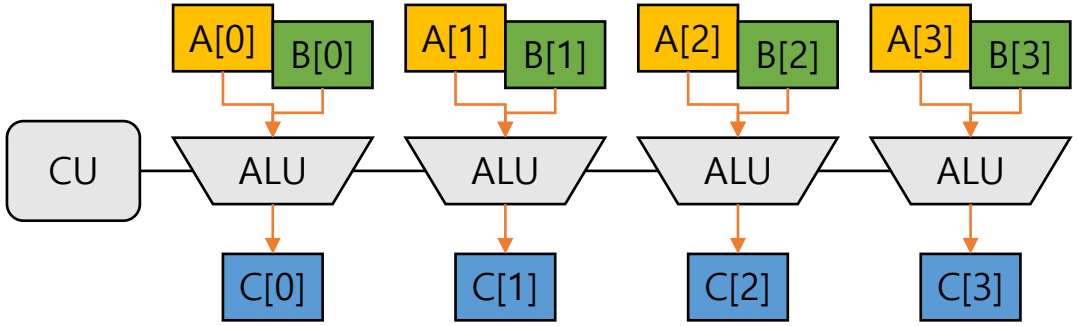
Vectorization vs. たこ焼き調理する

SIMD (Single Instruction Multiple Data) Parallel Strategy

Vectorized Operation

- Cast → vector register
- Heating → vector operation

Ex) C[:]=A[:]+B[:]



- 512-bit VPUs (AVX512) per core
  - vector register size: 512bit
  - 8 or 16 data can be treated at once
    64Byte (double) 32Byte type (float, int)

Vectorization vs. たこ焼き調理する

SIMD (Single Instruction Multiple Data) Parallel Strategy

- If most of time consuming comes from flipping process, performance gain from parallelization become negligible.

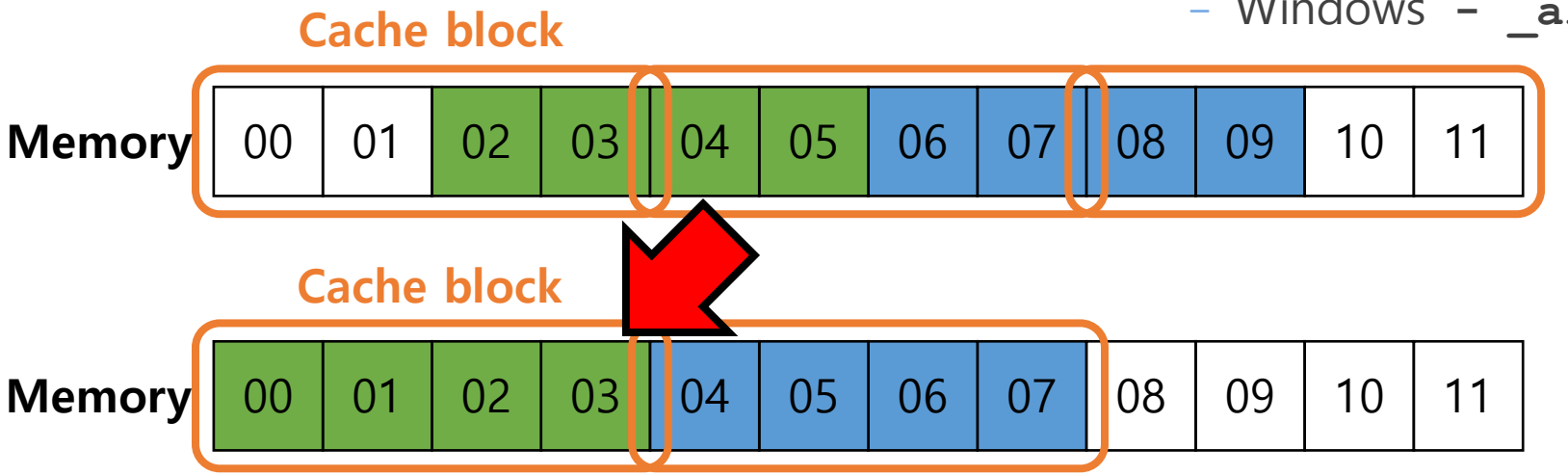
High Vectorization

Memory alignment

- Conditions for high Vectorization
  1. Memory alignment
  2. Memory access pattern
  3. Loop data dependency

- Memory align function
  - **`_mm_malloc`**
  - **`_mm_free`**
  - **`hbw_posix_memalign`** for HBM

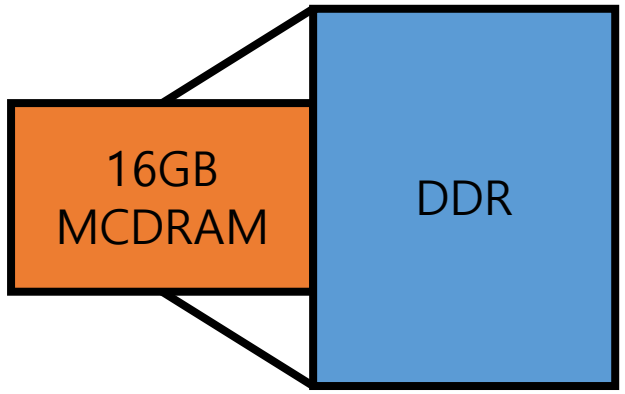
  - POSIX – **`posix_memaglign`**
  - C11 – **`algined_alloc`**
  - Windows - **`_aligned_malloc`**



MCDRAM memory modes

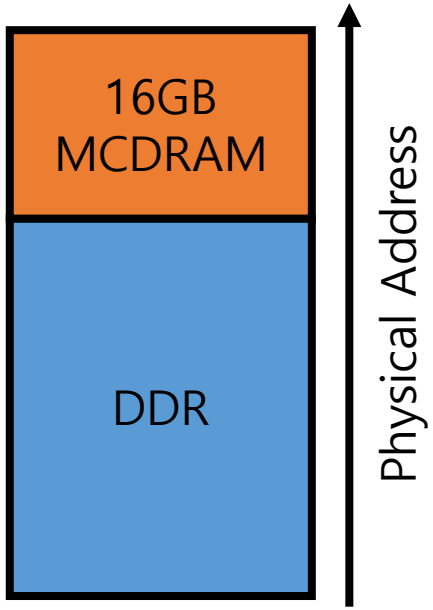
Three modes. Selected at boot

Cache Mode



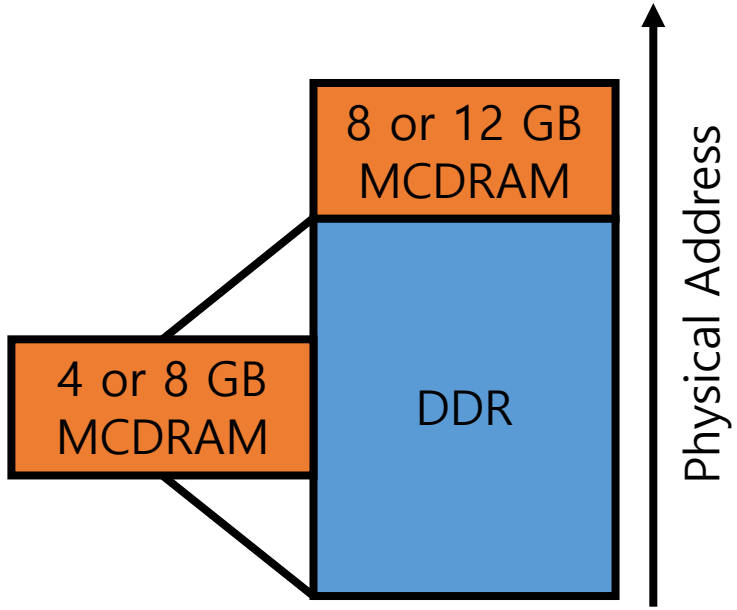
- MCDRAM is used as a L3 cache

Flat Mode



- MCDRAM is used as a DDR
  - numactl command
  - memkind library

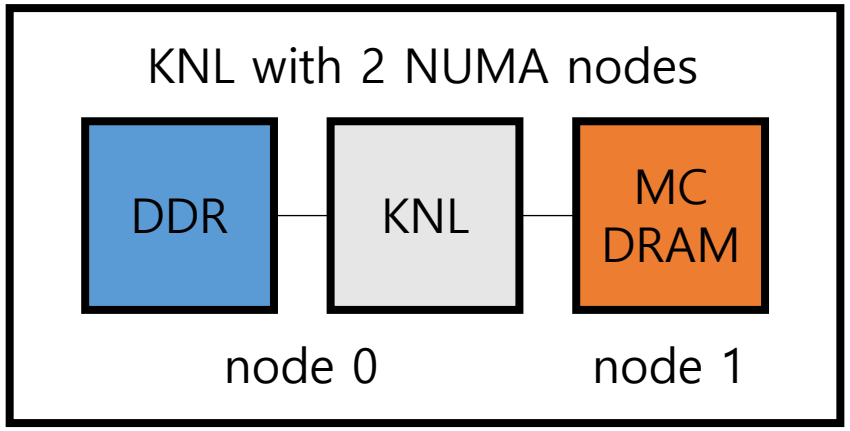
Hybrid Mode



- MCDRAM is used
  - as a L3 cache
  - as a DDR

MCDRAM using by numactl command

- Check memory details using numactl command
  - **`$ numactl --hardware`**




```
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 1
05 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 1
32 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 1
59 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 1
86 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 2
13 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 2
40 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255
node 0 size: 98178 MB
node 0 free: 94749 MB
node 1 cpus:
node 1 size: 16384 MB
node 1 free: 15929 MB
node distances:
node   0   1
  0:  10  31
  1:  31  10
```

- We can simply use MCDRAM with numactl command with membind option
  - **`$ numactl --membind 1 ./myapp.ex`**

MCDRAM using by memkind library

- Use **hbw_malloc** / **hbw_free** function, instead of **malloc** / **free** function
- Add **memkind** library to your compile option
  - **CFLAGS = -O3 -std=c11 -qopenmp -qop-report=5 -xMIC-AVX512 -lmemkind**
- Add a header file **<hbwmalloc.h>** in your source code
  - **#include <hbwmalloc.h>**

```
HBWMALLOC(3)                                HBWMALLOC                                HBWMALLOC(3)

NAME
       hbwmalloc - The high bandwidth memory interface
       Note: hbwmalloc.h functionality is considered as stable API (STANDARD API).

SYNOPSIS
       #include <hbwmalloc.h>

       Link with -lmemkind

       int hbw_check_available(void);
       void* hbw_malloc(size_t size);
       void* hbw_calloc(size_t nmemb, size_t size);
       void* hbw_realloc (void *ptr, size_t size);
       void hbw_free(void *ptr);
       int hbw_posix_memalign(void **memptr, size_t alignment, size_t size);
       int hbw_posix_memalign_psize(void **memptr, size_t alignment, size_t size, hbw_pagesize_t pagesize);
       hbw_policy_t hbw_get_policy(void);
       int hbw_set_policy(hbw_policy_t mode);
```

https://github.com/memkind/memkind

64 Physical Cores & 256 Logical Cores

- $ vi /proc/cpuinfo

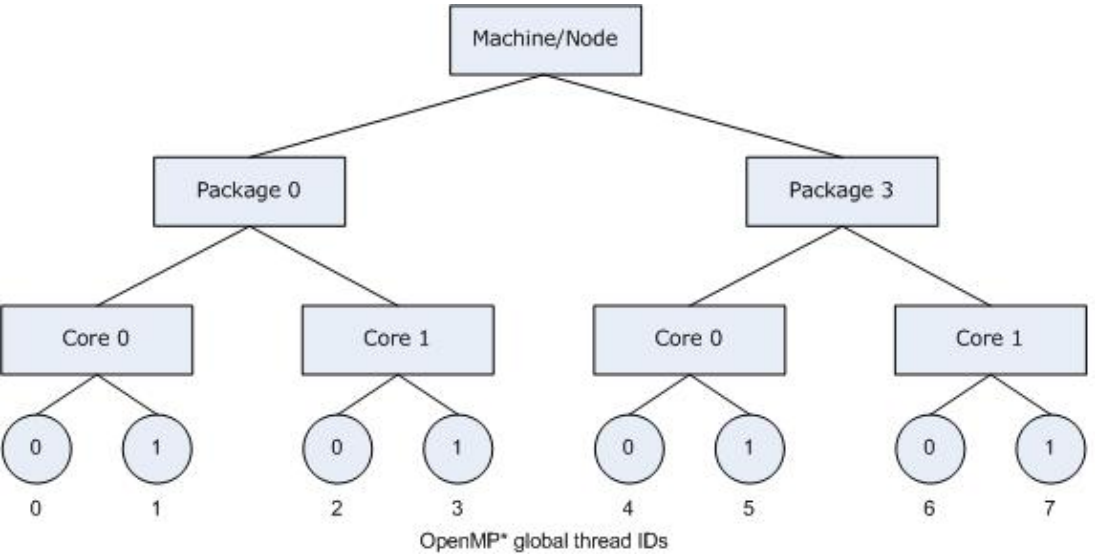
```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 87
model name      : Intel(R) Xeon Phi(TM) CPU 7210 @ 1.30GHz
stepping        : 1
microcode       : 0x130
cpu MHz         : 1300.000
cache size      : 1024 KB
physical id     : 0
siblings        : 256
core id         : 0
cpu cores       : 64
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
```

Thread Management

Tread Binding

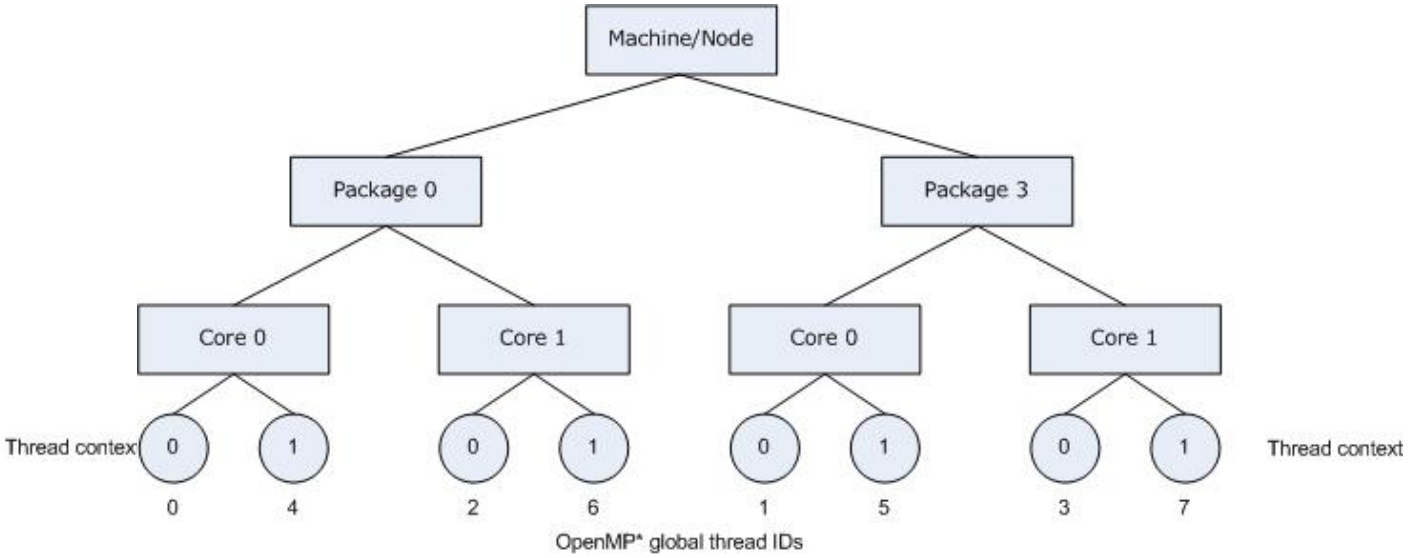
- Allocation of threads may affect performance seriously especially for computation with many threads
- export KMP_AFFINITY=compact,verbose

Compact



- Threads are allocated to be close to each other

Scatter



- Threads are allocated to be close to each other

Set KMP_AFFINITY

Can you guess the env. option of process?

OMP: Info #156: KMP_AFFINITY: 256 available OS procs

OMP: Info #157: KMP_AFFINITY: Uniform topology

OMP: Info #179: KMP_AFFINITY: 1 packages x 64 cores/pkg x 4 threads/core (64 total cores)

OMP: Info #206: KMP_AFFINITY: OS proc to physical thread map:

OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0

OMP: Info #171: KMP_AFFINITY: OS proc 64 maps to package 0 core 0 thread 1

……

OMP: Info #242: KMP_AFFINITY: pid 4393 thread 0 bound to OS proc set {0}

OMP: Info #242: KMP_AFFINITY: pid 4660 thread 1 bound to OS proc set {64}

Set KMP_AFFINITY

Can you guess the env. option of process?

OMP: Info #156: KMP_AFFINITY: 256 available OS procs

OMP: Info #157: KMP_AFFINITY: Uniform topology

OMP: Info #179: KMP_AFFINITY: 1 packages x 64 cores/pkg x 4 threads/core (64 total cores)

OMP: Info #206: KMP_AFFINITY: OS proc to physical thread map:

OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0

OMP: Info #171: KMP_AFFINITY: OS proc 64 maps to package 0 core 0 thread 1

……

OMP: Info #242: KMP_AFFINITY: pid 4393 thread 0 bound to OS proc set {0}

OMP: Info #242: KMP_AFFINITY: pid 4660 thread 1 bound to OS proc set {64}

export OMP_NUM_THREADS

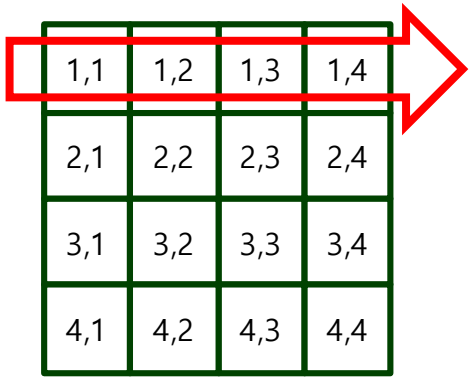
export KMP_AFFINITY=compact,verbose

Example 1: Dense Matrix multiplication

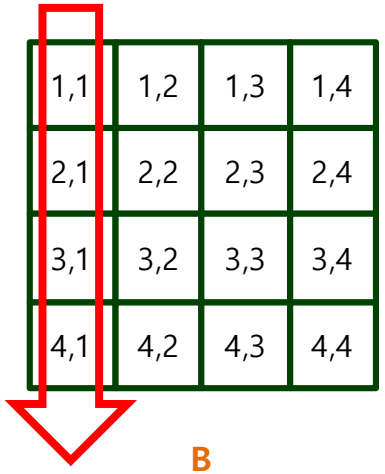
Human friendly code

```
for(int i=0; i<SIZE; i++) {
    for(int j=0; j<SIZE; j++) {
        double sum = 0;
        for(int k=0; k<SIZE; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

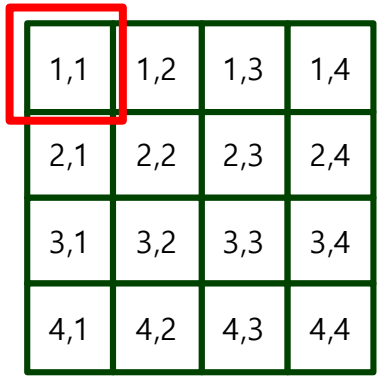
- For a 4 x 4 case
  - # of cache miss
    → 4 + 16 + 4 = 24
- For a general case of SIZE x SIZE
  - # of cache miss
    → SIZE + SIZE * SIZE + SIZE
    = SIZE * (SIZE+2)



A
Row-wise access
(i, *)



B
Colum-wise access
(*, j)



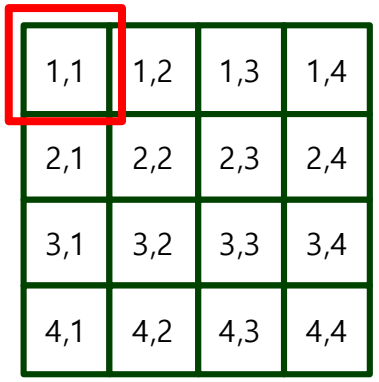
C
Fixed value access
(i, j)

Example 1: Dense Matrix multiplication

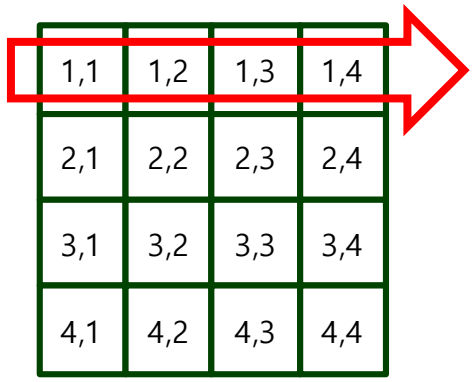
Cache & Vectorization friendly code

```
for(int i=0; i<SIZE; i++) {
    for(int k=0; k<SIZE; k++) {
        double A_val = A[i][k];
        for(int j=0; j<SIZE; j++) {
            C[i][j] += A_val * B[k][j];
        }
    }
}
```

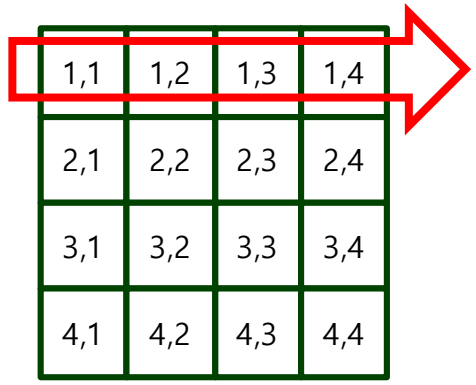
- For a 4 x 4 case
  - # of cache miss
    → 4 + 4 + 4 = 12
- For a general case of SIZE x SIZE
  - # of cache miss
    → SIZE + SIZE + SIZE
    = 3 * SIZE



A
Fixed value access
(i, k)



B
Row-wise access
(k, *)



C
Row-wise access
(i, *)

Example 1: Dense Matrix multiplication

Source code – MMmul.c

```
01  #include <stdio.h>
02  #include <string.h>
03  #include <omp.h>
04  #define SIZE 4096
05
06  int main(int argc, char *argv[]) {
07      double time;
08      double *A = (double*)_mm_malloc(sizeof(double)*SIZE*SIZE, 64);
09      double *B = (double*)_mm_malloc(sizeof(double)*SIZE*SIZE, 64);
10      double *C = (double*)_mm_malloc(sizeof(double)*SIZE*SIZE, 64);
11
12      #pragma omp parallel for
13      for(int i=0; i<SIZE; i++) {
14          #pragma vector aligned
15          #pragma omp simd
16          for(int j=0; j<SIZE; j++) {
17              A[i*SIZE+j] = (double)(i + j);
18              B[i*SIZE+j] = (double)(j - i);
19          }
20      }
```

Example 1: Dense Matrix multiplication

Source code – MMmul.c

```
21  ////////////////////////////////////////////////
22      memset(C, 0, sizeof(double)*SIZE*SIZE);
23
24      time = -omp_get_wtime();
25      #pragma omp parallel for
26      for(int i=0; i<SIZE; i++) {
27          #pragma omp simd
28          #pragma vector aligned
29          for(int j=0; j<SIZE; j++) {
30              double sum = 0;
31              for(int k=0; k<SIZE; k++) {
32                  sum += A[i*SIZE+k] * B[k*SIZE+j];
33              }
34              C[i*SIZE+j] = sum;
35          }
36      }
37      time += omp_get_wtime();
38      printf("\ti-j-k MMmul time: %lf (secs)\n", time);
39      printf("\t\tlast element: %lf\n\n", C[(SIZE-1)*SIZE+SIZE-1]);
```

```
40  ////////////////////////////////////////////////
41      memset(C, 0, sizeof(double)*SIZE*SIZE);
42
43      time = -omp_get_wtime();
44      #pragma omp parallel for
45      for(int i=0; i<SIZE; i++) {
46          for(int k=0; k<SIZE; k++) {
47              double A_val = A[i*SIZE+k];
48              #pragma omp simd
49              #pragma vector aligned
50              for(int j=0; j<SIZE; j++) {
51                  C[i*SIZE+j] += A_val * B[k*SIZE+j];
52              }
53          }
54      }
55      time += omp_get_wtime();
56      printf("\ti-k-j MMmul time: %lf (secs)\n", time);
57      printf("\t\tlast element: %lf\n\n", C[(SIZE-1)*SIZE+SIZE-1]);
58  ////////////////////////////////////////////////
59      _mm_free(A);
60      _mm_free(B);
61      _mm_free(C);
62
63      return 0;
64  }
```

Example 1: Dense Matrix multiplication

Results – Auto vectorization? Remove –no-vec option and simd directives

```
[root@node01 02_MMmul]# ./MMmul_O2.ex
        i-j-k MMmul time: 27.177551 (secs)
                last element: 45787822080.000000

        i-k-j MMmul time: 2.339471 (secs)
                last element: 45787822080.000000
[root@node01 02_MMmul]# ./MMmul_O2_AVX512.ex
        i-j-k MMmul time: 12.978245 (secs)
                last element: 45787822080.000000

        i-k-j MMmul time: 2.041360 (secs)
                last element: 45787822080.000000
[root@node01 02_MMmul]# ./MMmul_O3_AVX512.ex
        i-j-k MMmul time: 11.755714 (secs)
                last element: 45787822080.000000

        i-k-j MMmul time: 1.934455 (secs)
                last element: 45787822080.000000
```

Compile with
–no-vec option and simd directive

```
[root@node01 02_MMmul]# ./MMmul_O2_autovec.ex
        i-j-k MMmul time: 56.064762 (secs)
                last element: 45787822080.000000

        i-k-j MMmul time: 2.318266 (secs)
                last element: 45787822080.000000
[root@node01 02_MMmul]# ./MMmul_O2_AVX512_autovec.ex
        i-j-k MMmul time: 57.914773 (secs)
                last element: 45787822080.000000

        i-k-j MMmul time: 2.041022 (secs)
                last element: 45787822080.000000
[root@node01 02_MMmul]# ./MMmul_O3_AVX512_autovec.ex
        i-j-k MMmul time: 28.743565 (secs)
                last element: 45787822080.000000

        i-k-j MMmul time: 2.178874 (secs)
                last element: 45787822080.000000
```

Compile without
–no-vec option and simd directive

Example 1: Dense Matrix multiplication

Results – Auto vectorization? Remove –no-vec option and simd directives

```
87  LOOP BEGIN at MMmul.c(29,12)
88     remark #15388: vectorization support: reference B has aligned access   [ Mmul
89     remark #15388: vectorization support: reference C has aligned access   [ Mmul
90     remark #15305: vectorization support: vector length 16
91     remark #15309: vectorization support: normalized vectorization overhead 333
92     remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
93     remark #15448: unmasked aligned unit stride loads: 1
94     remark #15449: unmasked aligned unit stride stores: 1
95     remark #15475: --- begin vector cost summary ---
96     remark #15476: scalar cost: 14
97     remark #15477: vector cost: 1.680
98     remark #15478: estimated potential speedup: 8.280
99     remark #15488: --- end vector cost summary ---
100    remark #25015: Estimate of max trip count of loop=256
```

Let's Check this code line

Compile with
–no-vec option and simd directive

```
87  LOOP BEGIN at MMmul.c(29,3)
88     remark #15542: loop was not vectorized: inner loop was already vectorized
89
90     LOOP BEGIN at MMmul.c(31,4)
91     <Peeled loop for vectorization>
92        remark #15389: vectorization support: reference A has unaligned access   [
93        remark #15381: vectorization support: unaligned access used inside loop bo
94        remark #15335: peel loop was not vectorized: vectorization possible but se
95        remark #15305: vectorization support: vector length 8
96        remark #15309: vectorization support: normalized vectorization overhead 1.
97        remark #25015: Estimate of max trip count of loop=15
98     LOOP END
99
100    LOOP BEGIN at MMmul.c(31,4)
101       remark #15388: vectorization support: reference A has aligned access   [
102       remark #15415: vectorization support: non-unit strided load was generate
103       remark #15305: vectorization support: vector length 16
104       remark #15399: vectorization support: unroll factor set to 2
105       remark #15309: vectorization support: normalized vectorization overhead
106       remark #15300: LOOP WAS VECTORIZED
107       remark #15442: entire loop may be executed in remainder
108       remark #15448: unmasked aligned unit stride loads: 1
109       remark #15452: unmasked strided loads: 1
110       remark #15475: --- begin vector cost summary ---
111       remark #15476: scalar cost: 8
112       remark #15477: vector cost: 2.500
113       remark #15478: estimated potential speedup: 3.130
114
115       remark
116    LOOP END
117
118    LOOP BEGI
119    <Remainde
120       remark #15389: vectorization support: reference A has unaligned access   [
121       remark #15381: vectorization support: unaligned access used inside loop bo
122       remark #15305: vectorization support: vector length 8
123       remark #15309: vectorization support: normalized vectorization overhead 1.
124       remark #15301: REMAINDER LOOP WAS VECTORIZED
125    LOOP END
```

Compile without
–no-vec option and simd directive

Example 1: Dense Matrix multiplication

Let's use MCDRAM (Multi Channel DRAM)

```
#define SIZE 6144
```

```
$ export OMP_NUM_THREADS=256
```

```
[root@node01 02_MMmul]# ./MMmul_O3_AVX512.ex
        i-j-k MMmul time: 33.214267 (secs)
                last element: 154562204672.000000

        i-k-j MMmul time: 12.798154 (secs)
                last element: 154562204672.000000
```

```
[root@node01 02_MMmul]# numactl --membind 1 ./MMmul_O3_AVX512.ex
        i-j-k MMmul time: 27.525452 (secs)
                last element: 154562204672.000000

        i-k-j MMmul time: 6.070123 (secs)
                last element: 154562204672.000000
```

```
$ export OMP_NUM_THREADS=16
```

```
[root@node01 02_MMmul]# ./MMmul_O3_AVX512.ex
        i-j-k MMmul time: 100.921259 (secs)
                last element: 154562204672.000000

        i-k-j MMmul time: 26.320423 (secs)
                last element: 154562204672.000000
```

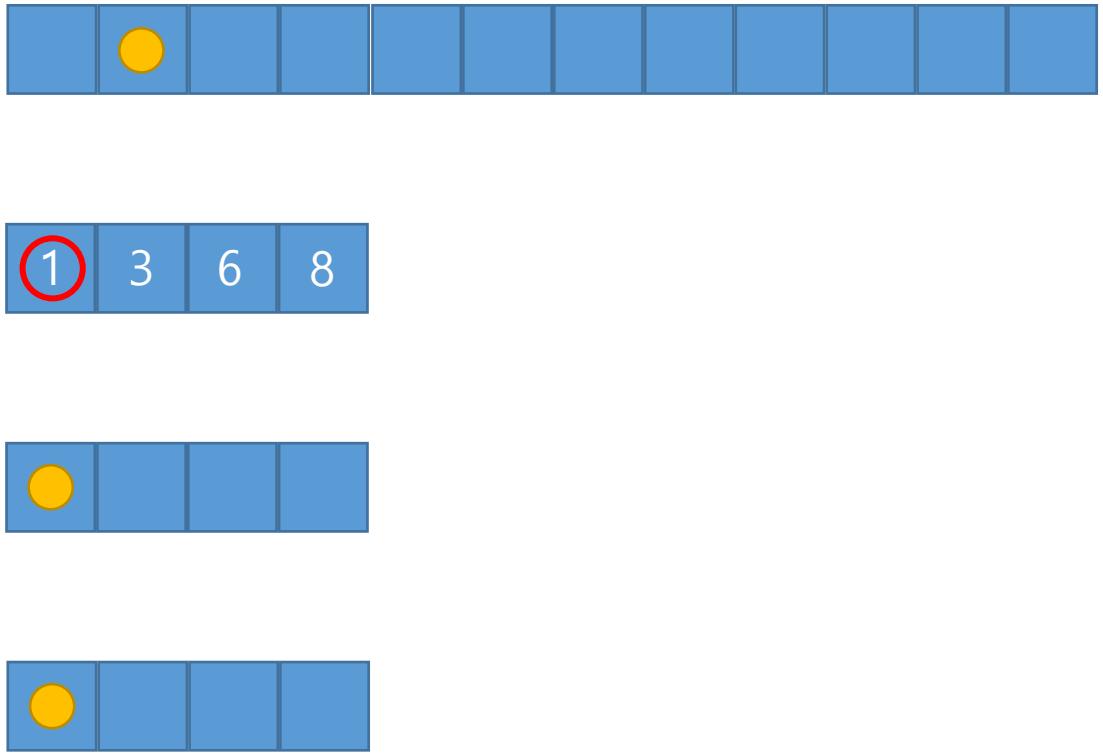
```
[root@node01 02_MMmul]# numactl --membind 1 ./MMmul_O3_AVX512.ex
        i-j-k MMmul time: 37.283408 (secs) ???
                last element: 154562204672.000000

        i-k-j MMmul time: 27.557915 (secs) ???
                last element: 154562204672.000000
```

Example 2: Dot Product (Prefetch)

Dot Product Between Sparse Vector and Dense Vector

Code

```
double A   = malloc(sizeof *A     * N);
double B   = malloc(sizeof *B     * M);
double B_  = malloc(sizeof *B_    * N);
double C   = malloc(sizeof *C     * N);
int index = malloc(sizeof *index * N);

for (int i = 0; i < N; i++)
    C[i] = A[i] * B[index[i]];

for (int i = 0; i < N; i++)
    B_[i] = B[index[i]];
for (int i = 0; i < N; i++)
    C[i] = A[i] * B_[i];
```

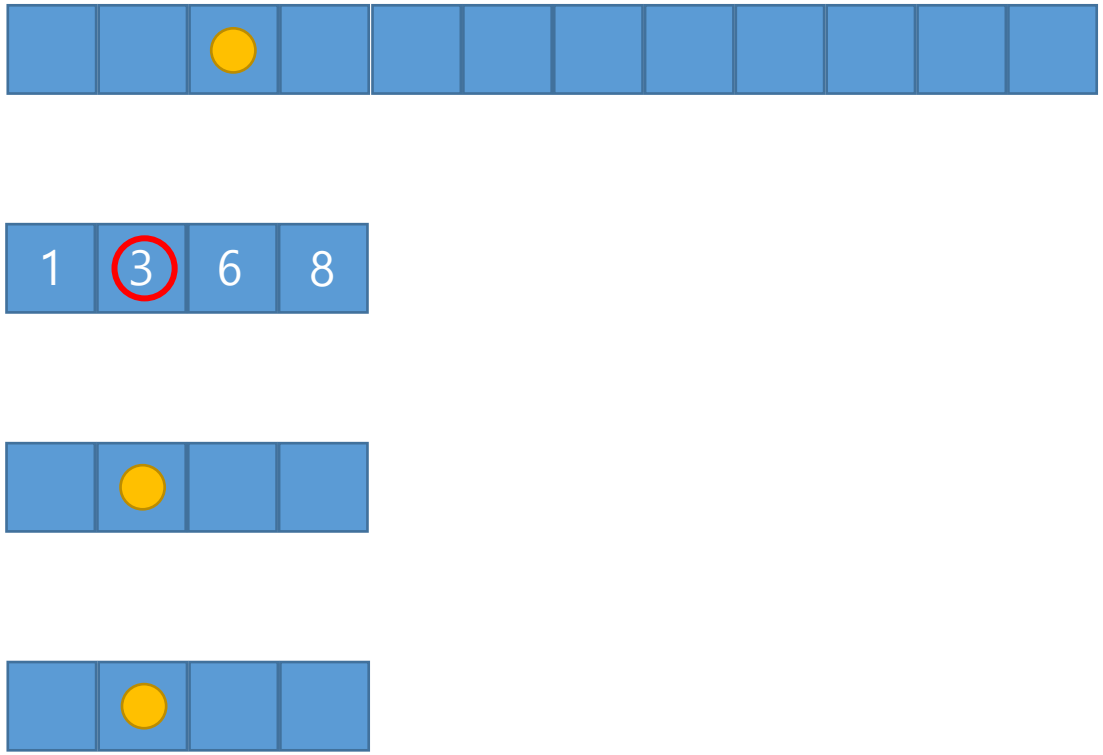


Example 2: Dot Product (Prefetch)

Dot Product Between Sparse Vector and Dense Vector

Code

```
double A  = malloc(sizeof *A     * N);
double B  = malloc(sizeof *B     * M);
double B_ = malloc(sizeof *B_    * N);
double C  = malloc(sizeof *C     * N);
int index = malloc(sizeof *index * N);

for (int i = 0; i < N; i++)
    C[i] = A[i] * B[index[i]];

for (int i = 0; i < N; i++)
    B_[i] = B[index[i]];
for (int i = 0; i < N; i++)
    C[i] = A[i] * B_[i];
```

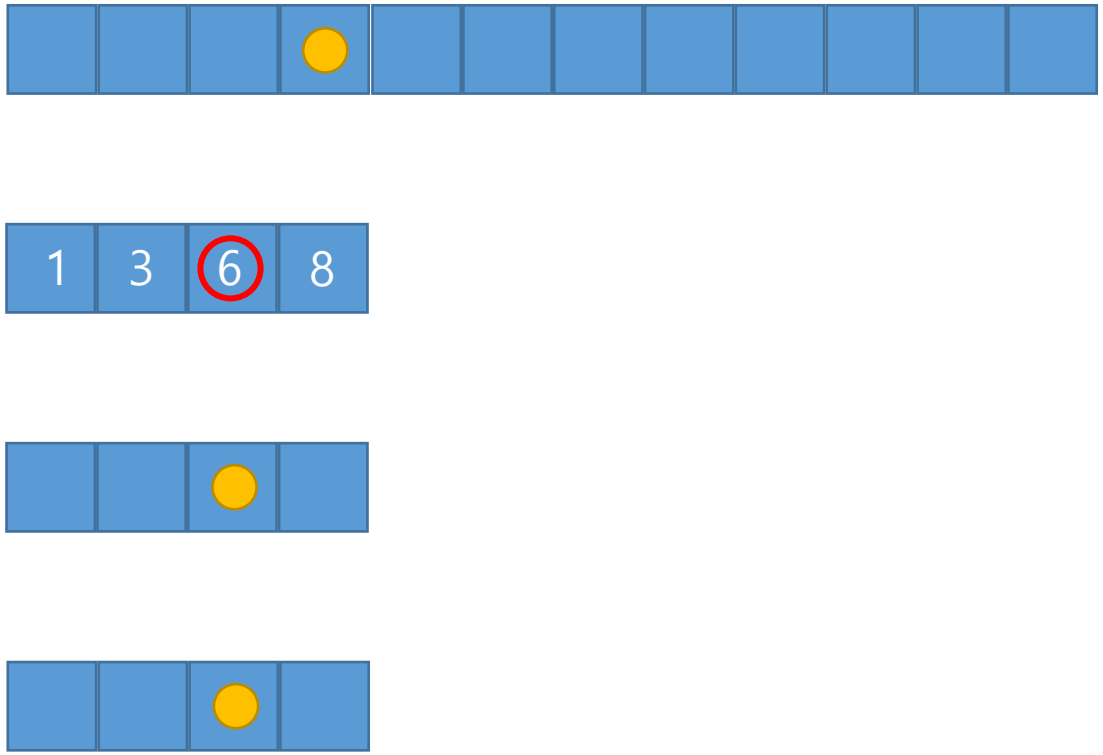


Example 2: Dot Product (Prefetch)

Dot Product Between Sparse Vector and Dense Vector

Code

```
double A  = malloc(sizeof *A     * N);
double B  = malloc(sizeof *B     * M);
double B_ = malloc(sizeof *B_    * N);
double C  = malloc(sizeof *C     * N);
int index = malloc(sizeof *index * N);

for (int i = 0; i < N; i++)
    C[i] = A[i] * B[index[i]];

for (int i = 0; i < N; i++)
    B_[i] = B[index[i]];
for (int i = 0; i < N; i++)
    C[i] = A[i] * B_[i];
```



Example 2: Dot Product (Prefetch)

Dot Product Between Sparse Vector and Dense Vector

Code

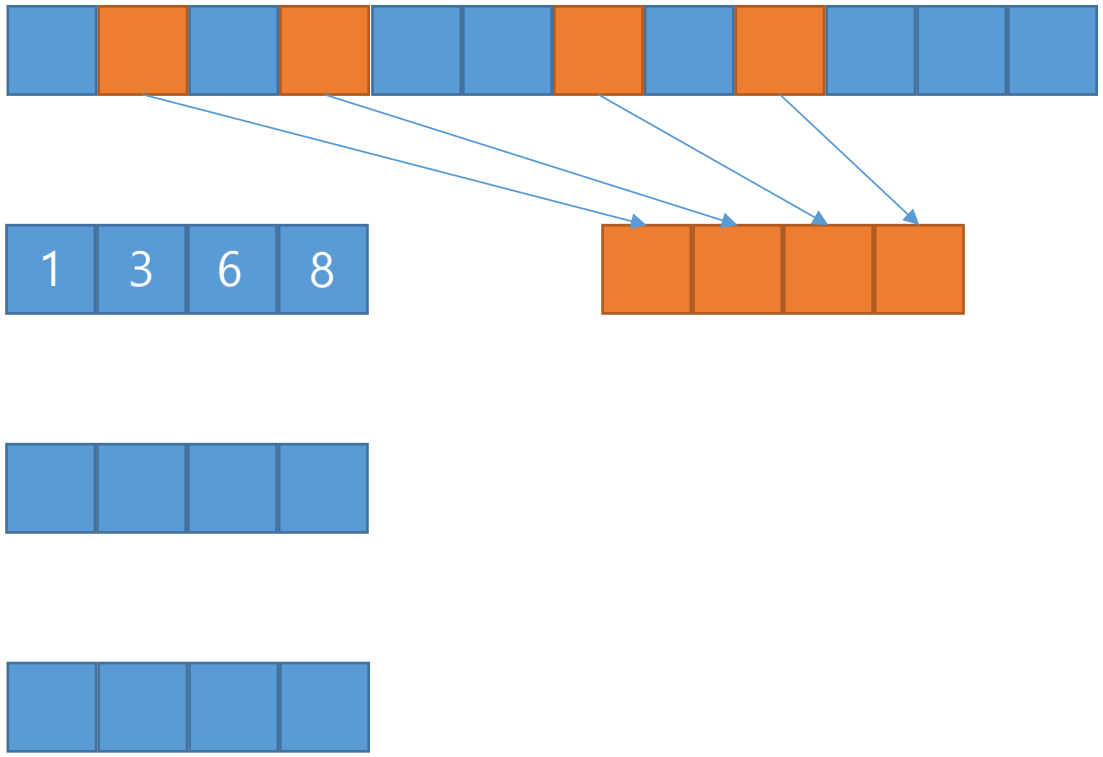
```
double A  = malloc(sizeof *A     * N);
double B  = malloc(sizeof *B     * M);
double B_ = malloc(sizeof *B_    * N);
double C  = malloc(sizeof *C     * N);
int index = malloc(sizeof *index * N);

for (int i = 0; i < N; i++)
    C[i] = A[i] * B[index[i]];

for (int i = 0; i < N; i++)
    B_[i] = B[index[i]];
for (int i = 0; i < N; i++)
    C[i] = A[i] * B_[i];
```

Example 2: Dot Product (Prefetch)

Dot Product Between Sparse Vector and Dense Vector

Code

```
double A  = malloc(sizeof *A     * N);
double B  = malloc(sizeof *B     * M);
double B_ = malloc(sizeof *B_    * N);
double C  = malloc(sizeof *C     * N);
int index = malloc(sizeof *index * N);

for (int i = 0; i < N; i++)
    C[i] = A[i] * B[index[i]];

for (int i = 0; i < N; i++)
    B_[i] = B[index[i]];
for (int i = 0; i < N; i++)
    C[i] = A[i] * B_[i];
```

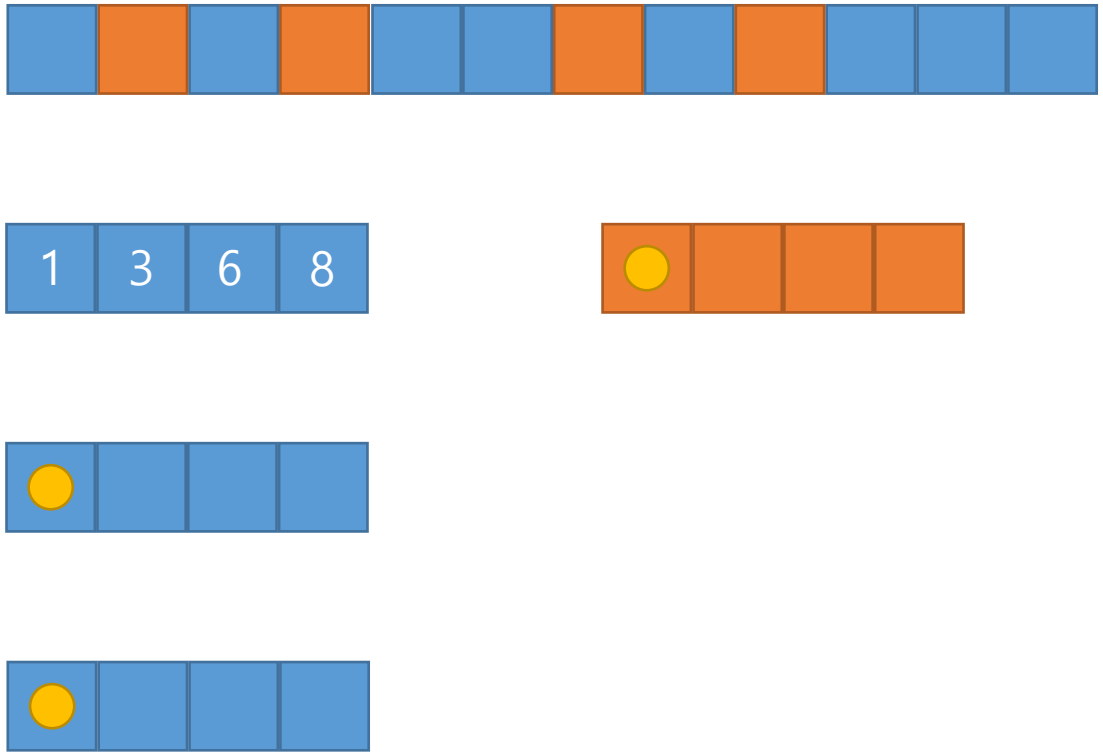


Example 2: Dot Product (Prefetch)

Dot Product Between Sparse Vector and Dense Vector

Code

```
double A  = malloc(sizeof *A     * N);
double B  = malloc(sizeof *B     * M);
double B_ = malloc(sizeof *B_    * N);
double C  = malloc(sizeof *C     * N);
int index = malloc(sizeof *index * N);

for (int i = 0; i < N; i++)
    C[i] = A[i] * B[index[i]];

for (int i = 0; i < N; i++)
    B_[i] = B[index[i]];
for (int i = 0; i < N; i++)
    C[i] = A[i] * B_[i];
```

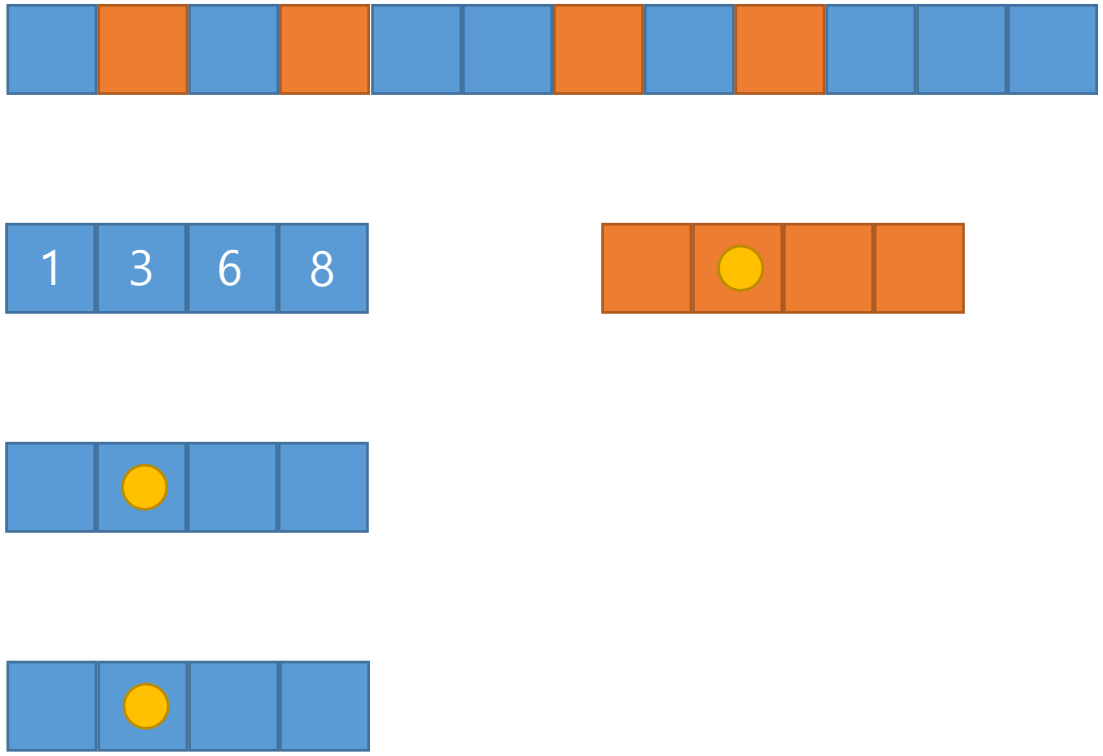


Example 2: Dot Product (Prefetch)

Dot Product Between Sparse Vector and Dense Vector

Code

```
double A  = malloc(sizeof *A     * N);
double B  = malloc(sizeof *B     * M);
double B_ = malloc(sizeof *B_    * N);
double C  = malloc(sizeof *C     * N);
int index = malloc(sizeof *index * N);

for (int i = 0; i < N; i++)
    C[i] = A[i] * B[index[i]];

for (int i = 0; i < N; i++)
    B_[i] = B[index[i]];
for (int i = 0; i < N; i++)
    C[i] = A[i] * B_[i];
```

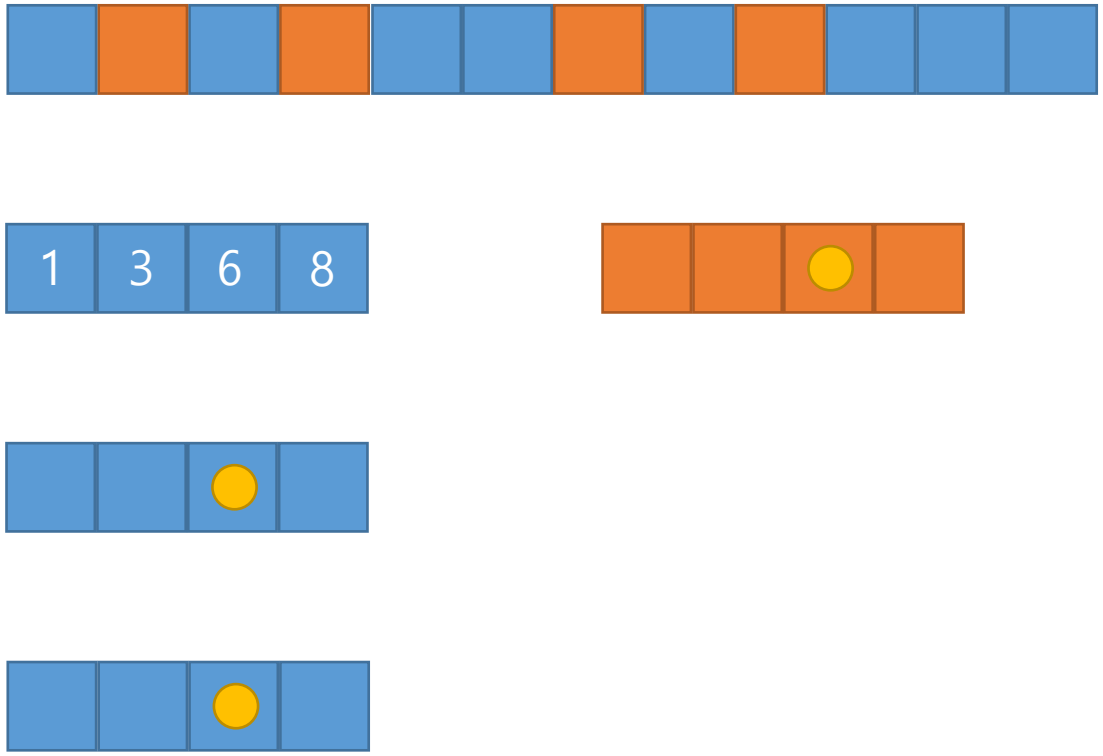


Example 2: Dot Product (Prefetch)

Dot Product Between Sparse Vector and Dense Vector

Code

```
double A  = malloc(sizeof *A     * N);
double B  = malloc(sizeof *B     * M);
double B_ = malloc(sizeof *B_    * N);
double C  = malloc(sizeof *C     * N);
int index = malloc(sizeof *index * N);

for (int i = 0; i < N; i++)
    C[i] = A[i] * B[index[i]];

for (int i = 0; i < N; i++)
    B_[i] = B[index[i]];
for (int i = 0; i < N; i++)
    C[i] = A[i] * B_[i];
```

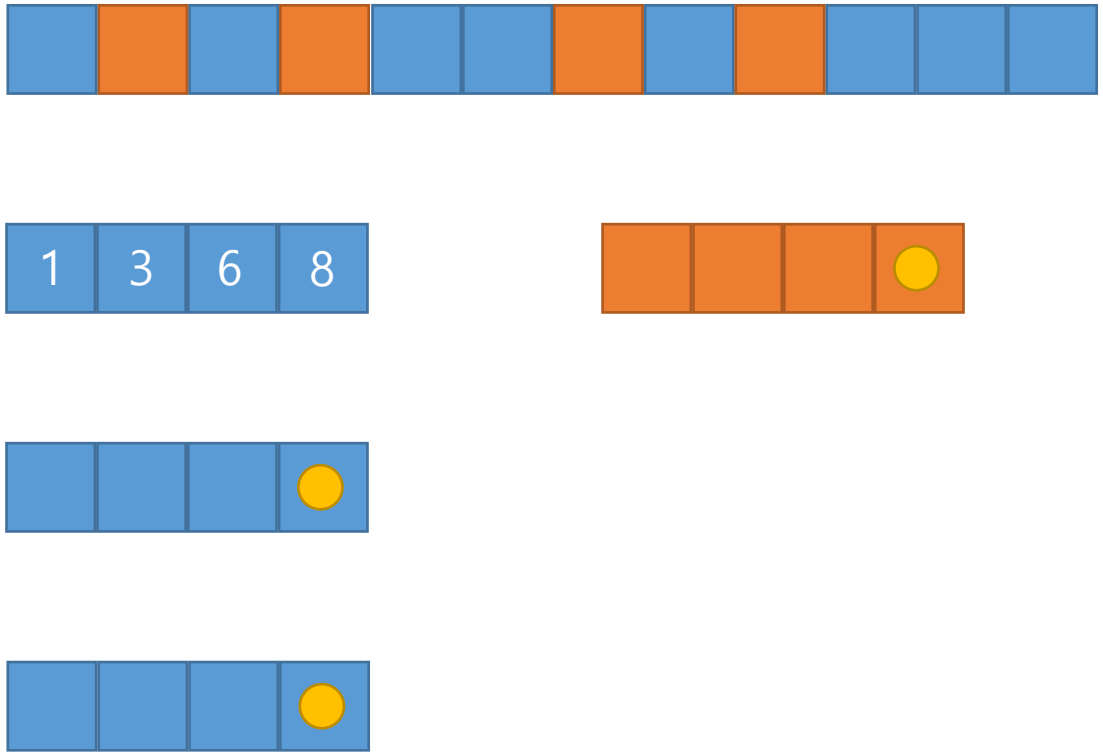


Example 2: Dot Product (Prefetch)

Dot Product Between Sparse Vector and Dense Vector

Code

```
double A  = malloc(sizeof *A     * N);
double B  = malloc(sizeof *B     * M);
double B_ = malloc(sizeof *B_    * N);
double C  = malloc(sizeof *C     * N);
int index = malloc(sizeof *index * N);

for (int i = 0; i < N; i++)
    C[i] = A[i] * B[index[i]];

for (int i = 0; i < N; i++)
    B_[i] = B[index[i]];
for (int i = 0; i < N; i++)
    C[i] = A[i] * B_[i];
```



Example 2: Dot Product (Prefetch)

Source Code of Dot Product

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <math.h>
04: #include <omp.h>
05:
06: int main(int argc, char **argv){
07:    int N = 10000000;
08:    int Nnnz = atoi(argv[1]);
09:    double time;
10:    int    *index       = malloc(sizeof *index       * Nnnz);
11:    double *svector_in  = malloc(sizeof *svector_in  * Nnnz);
12:    double *fvector_in  = malloc(sizeof *fvector_in  * N   );
13:    double *svector_out = malloc(sizeof *svector_out * Nnnz);
14:    double *temp        = malloc(sizeof *temp        * Nnnz);
15:    for (int i = 0; i < N; i++)
16:        fvector_in[i] = (double)(i);
17:    for (int i = 0; i < Nnnz; i++) {
18:        svector_in[i] = (double)(i);
19:        index[i] = i * (int)(N / Nnnz);
20:        svector_out[i] = 0.;
21:        temp[i] = fvector_in[index[i]];
22:    }
```

```
23:    time = -omp_get_wtime();
24:    for (int j = 0; j < 100000; j++)
25:        #pragma simd
26:        for (int i = 0; i < Nnnz; i++)
27:            svector_out[i] = svector_in[i] * fvector_in[index[i]];
28:    time += omp_get_wtime();
29:    printf("call-core: %e (secs)\n", time);
30:
31:    time = -omp_get_wtime();
32:    for (int j = 0; j < 100000; j++)
33:        #pragma simd
34:        for (int i = 0; i < Nnnz; i++)
35:            svector_out[i] = svector_in[i] * temp[i];
36:    time += omp_get_wtime();
37:    printf("call-core: %e (secs)\n", time);
38:
39:    free(index); free(svector_in);
40:    free(svector_out); free(fvector_in);
41:    return 0;
42: }
```

Example 2: Dot Product (Prefetch)

without MCDRAM

64 threads w/ scatter

```
VVdot_O0.ex is running
        omp_num_thread: 64
        1 VVdot time: 41.577527 (secs)
                sum: 218448213359909056.000000

        2 VVdot time: 20.869930 (secs)
                sum: 218448213359909056.000000
VVdot_O1.ex is running
        omp_num_thread: 64
        1 VVdot time: 12.892845 (secs)
                sum: 218448213359909056.000000

        2 VVdot time: 2.736105 (secs)
                sum: 218448213359909056.000000
VVdot_O2.ex is running
        omp_num_thread: 64
        1 VVdot time: 11.400964 (secs)
                sum: 218448213360000000.000000

        2 VVdot time: 1.946798 (secs)
                sum: 218448213360000000.000000
VVdot_O2_AVX512.ex is running
        omp_num_thread: 64
        1 VVdot time: 9.843138 (secs)
                sum: 218448213360000000.000000

        2 VVdot time: 1.388379 (secs)
                sum: 218448213360000000.000000
VVdot_O3_AVX512.ex is running
        omp_num_thread: 64
        1 VVdot time: 9.855993 (secs)
                sum: 218448213360000000.000000

        2 VVdot time: 1.387991 (secs)
                sum: 218448213360000000.000000
```

256 threads w/ scatter

```
VVdot_O0.ex is running
        omp_num_thread: 256
        1 VVdot time: 41.427260 (secs)
                sum: 218448213359909056.000000

        2 VVdot time: 20.880240 (secs)
                sum: 218448213359909056.000000
VVdot_O1.ex is running
        omp_num_thread: 256
        1 VVdot time: 13.065071 (secs)
                sum: 218448213359909056.000000

        2 VVdot time: 2.871832 (secs)
                sum: 218448213359909056.000000
VVdot_O2.ex is running
        omp_num_thread: 256
        1 VVdot time: 11.219486 (secs)
                sum: 218448213360000000.000000

        2 VVdot time: 1.902991 (secs)
                sum: 218448213360000000.000000
VVdot_O2_AVX512.ex is running
        omp_num_thread: 256
        1 VVdot time: 9.992419 (secs)
                sum: 218448213360000000.000000

        2 VVdot time: 1.447545 (secs)
                sum: 218448213360000000.000000
VVdot_O3_AVX512.ex is running
        omp_num_thread: 256
        1 VVdot time: 9.941504 (secs)
                sum: 218448213360000000.000000

        2 VVdot time: 1.367001 (secs)
                sum: 218448213360000000.000000
```

256 threads w/ compact

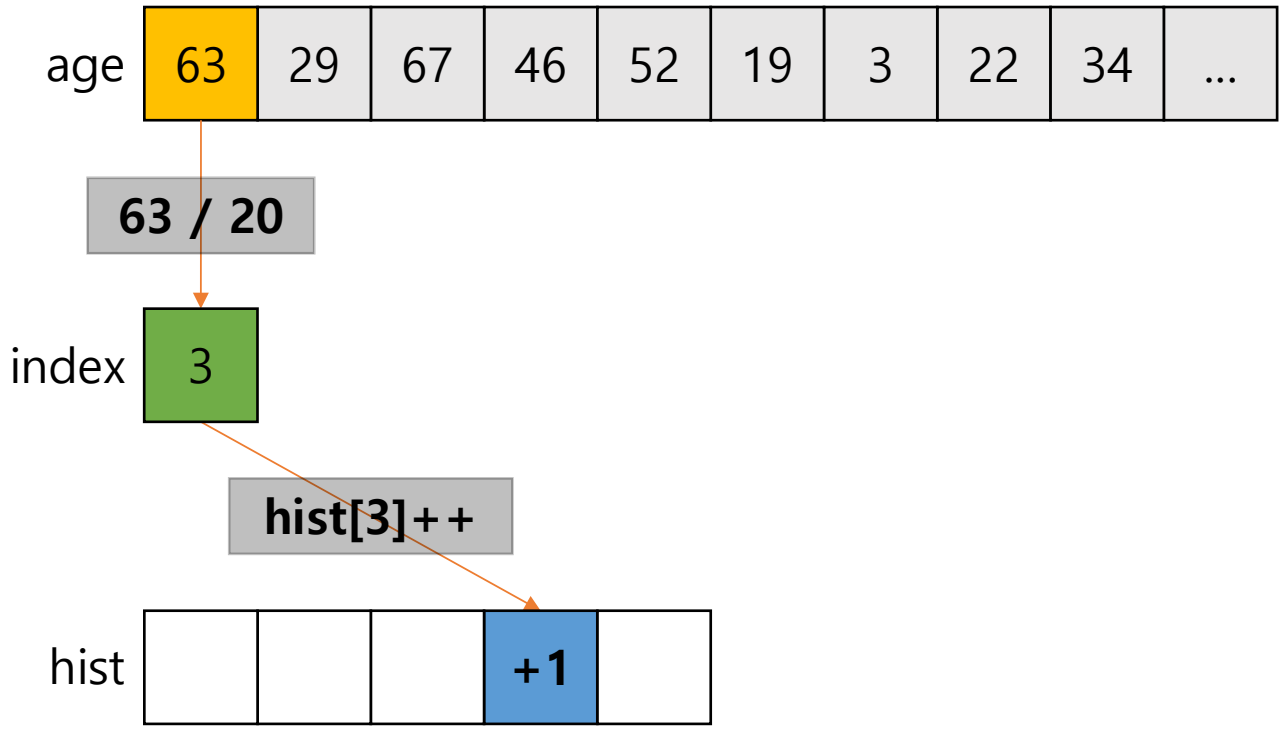
```
VVdot_O0.ex is running
        omp_num_thread: 256
        1 VVdot time: 40.624151 (secs)
                sum: 218448213359909056.000000

        2 VVdot time: 20.869335 (secs)
                sum: 218448213359909056.000000
VVdot_O1.ex is running
        omp_num_thread: 256
        1 VVdot time: 13.103378 (secs)
                sum: 218448213359909056.000000

        2 VVdot time: 2.876818 (secs)
                sum: 218448213359909056.000000
VVdot_O2.ex is running
        omp_num_thread: 256
        1 VVdot time: 11.007064 (secs)
                sum: 218448213360000000.000000

        2 VVdot time: 1.867134 (secs)
                sum: 218448213360000000.000000
VVdot_O2_AVX512.ex is running
        omp_num_thread: 256
        1 VVdot time: 9.952939 (secs)
                sum: 218448213360000000.000000

        2 VVdot time: 1.393572 (secs)
                sum: 218448213360000000.000000
VVdot_O3_AVX512.ex is running
        omp_num_thread: 256
        1 VVdot time: 9.946410 (secs)
                sum: 218448213360000000.000000

        2 VVdot time: 1.378017 (secs)
                sum: 218448213360000000.000000
```

Example 3: Histogram

Human friendly code

```
for(int i=0; i<N; i++)
{
    int index = (int)(age[i] / 20);
    hist[index]++;
}
```

```
int index[VL];
for(int i=0; i<N; i+=VL)
{
    for(int j=i; j<i+VL; j++)
        index[j-i] = (int)(age[j] / 20);
    for(int j=0; j<VL; j++)
        hist[index[j]]++;
}
```

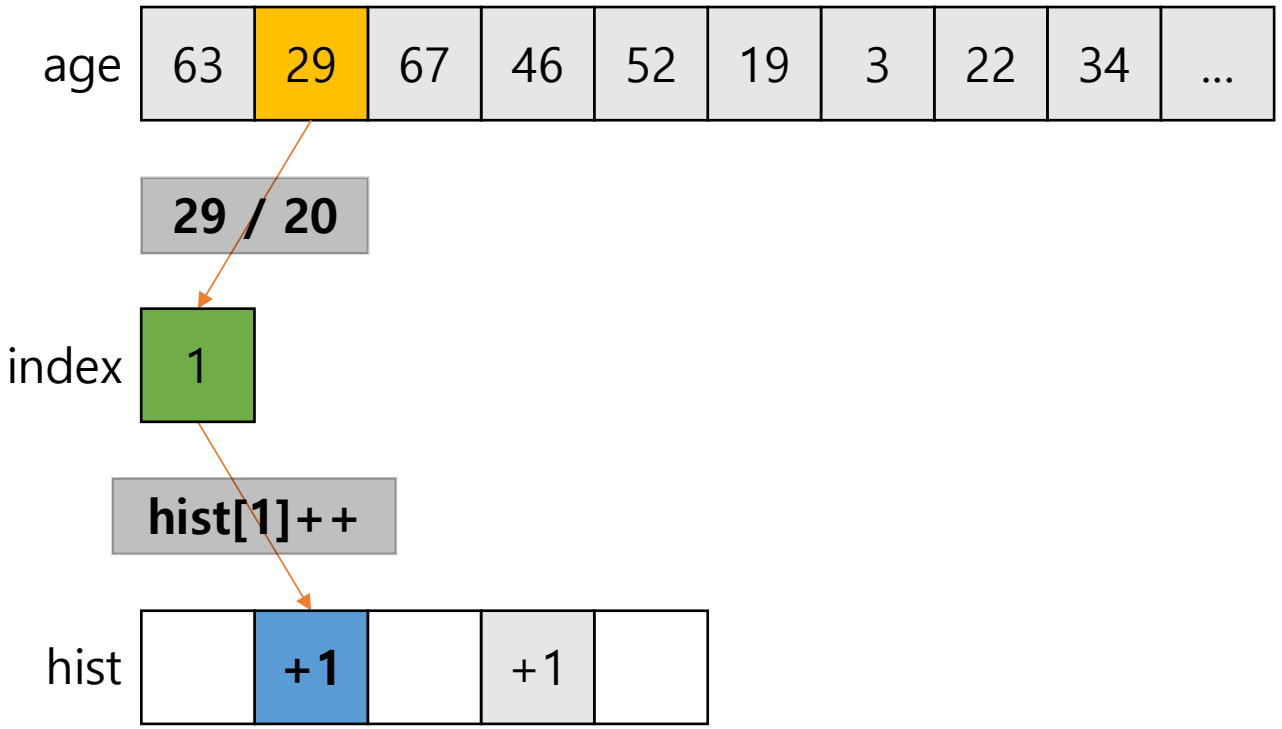


Example 3: Histogram

Human friendly code

```
for(int i=0; i<N; i++)
{
    int index = (int)(age[i] / 20);
    hist[index]++;
}
```

```
int index[VL];
for(int i=0; i<N; i+=VL)
{
    for(int j=i; j<i+VL; j++)
        index[j-i] = (int)(age[j] / 20);
    for(int j=0; j<VL; j++)
        hist[index[j]]++;
}
```

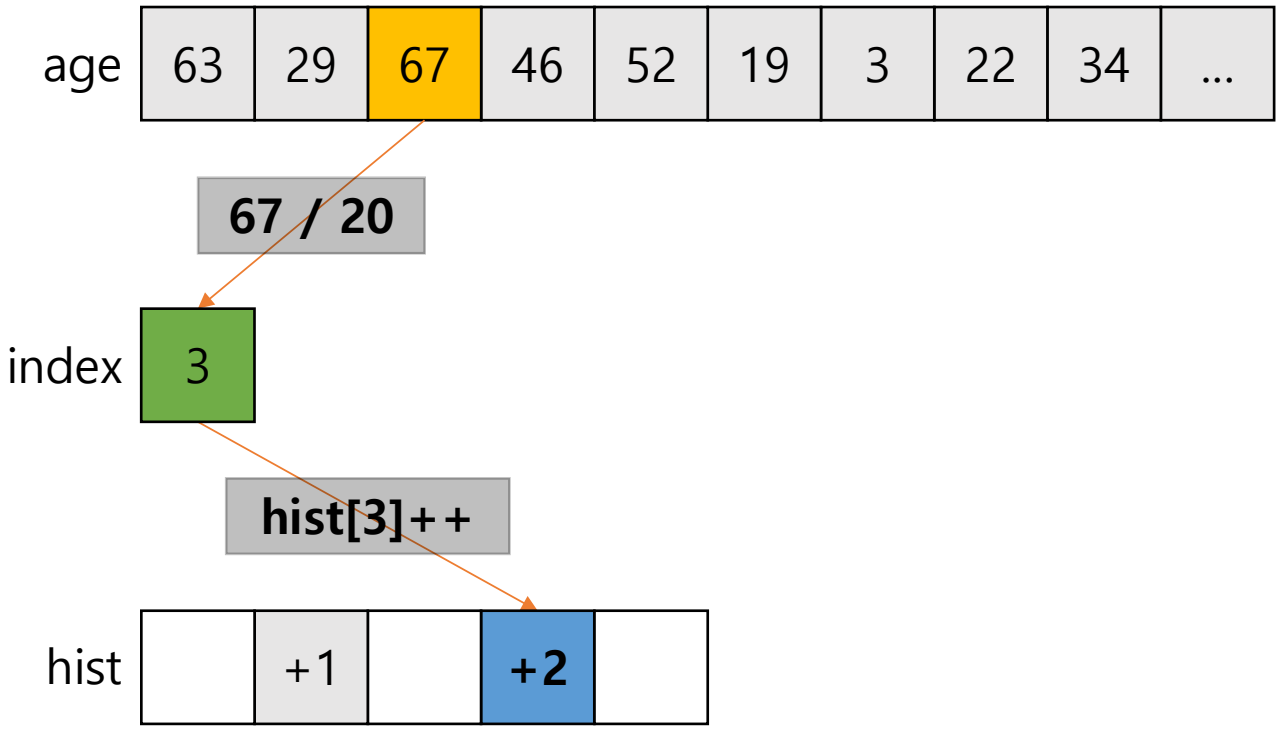


Example 3: Histogram

Human friendly code

```
for(int i=0; i<N; i++)
{
    int index = (int)(age[i] / 20);
    hist[index]++;
}
```

```
int index[VL];
for(int i=0; i<N; i+=VL)
{
    for(int j=i; j<i+VL; j++)
        index[j-i] = (int)(age[j] / 20);
    for(int j=0; j<VL; j++)
        hist[index[j]]++;
}
```

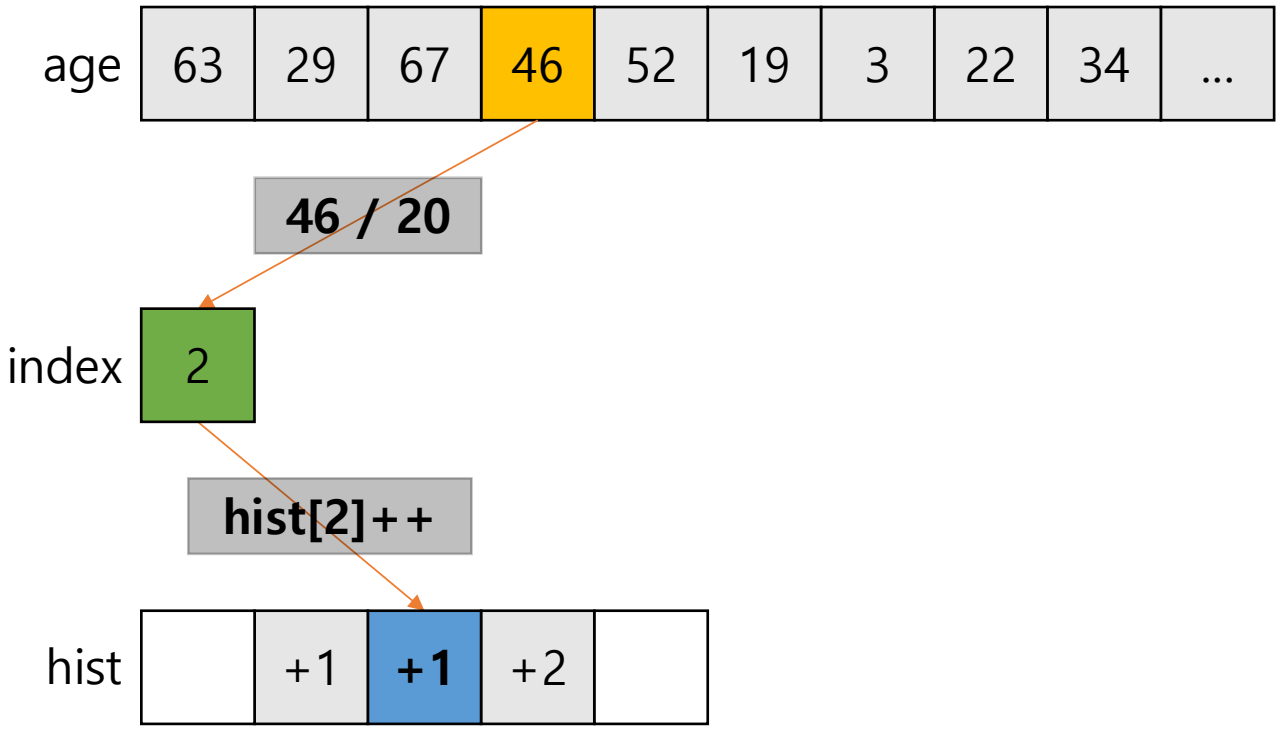


Example 3: Histogram

Human friendly code

```
for(int i=0; i<N; i++)
{
    int index = (int)(age[i] / 20);
    hist[index]++;
}
```

```
int index[VL];
for(int i=0; i<N; i+=VL)
{
    for(int j=i; j<i+VL; j++)
        index[j-i] = (int)(age[j] / 20);
    for(int j=0; j<VL; j++)
        hist[index[j]]++;
}
```

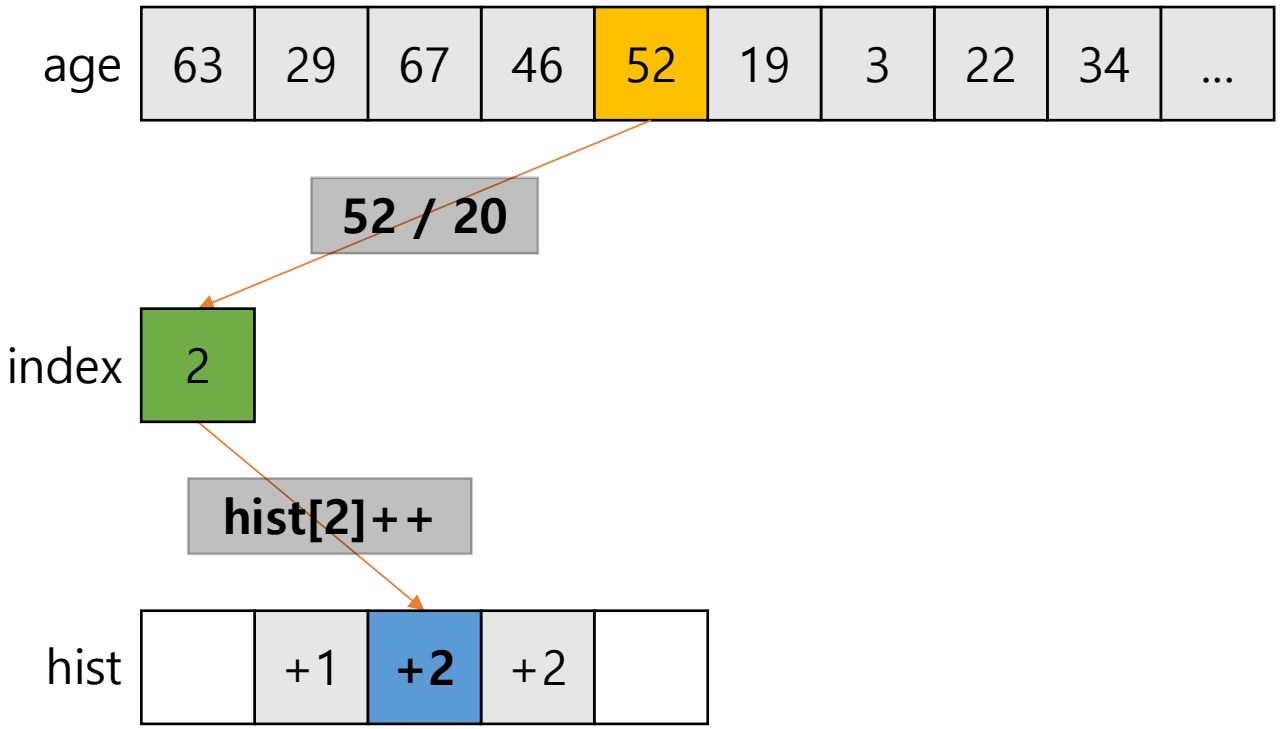


Example 3: Histogram

Human friendly code

```
for(int i=0; i<N; i++)
{
    int index = (int)(age[i] / 20);
    hist[index]++;
}
```

```
int index[VL];
for(int i=0; i<N; i+=VL)
{
    for(int j=i; j<i+VL; j++)
        index[j-i] = (int)(age[j] / 20);
    for(int j=0; j<VL; j++)
        hist[index[j]]++;
}
```

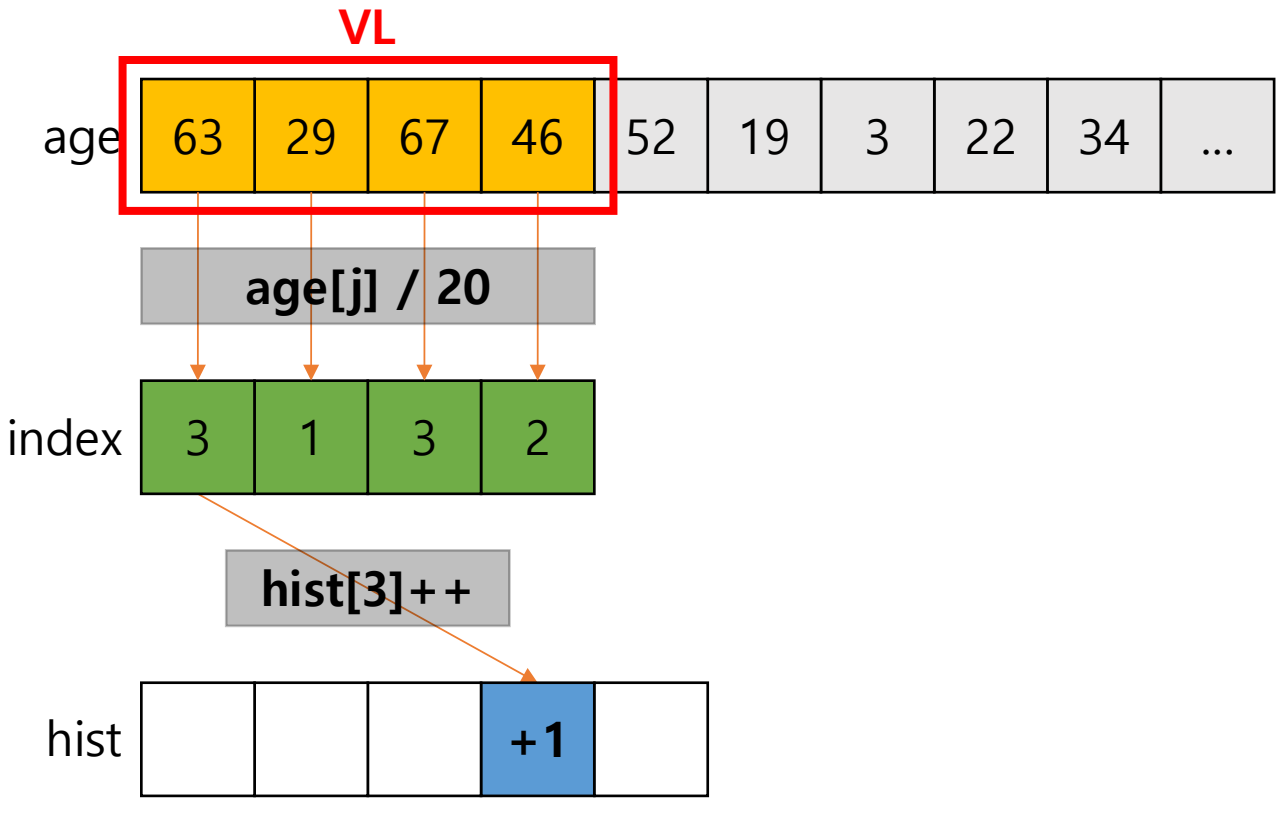


Example 3: Histogram

Cache & Vectorization friendly code

```
for(int i=0; i<N; i++)
{
    int index = (int)(age[i] / 20);
    hist[index]++;
}
```

```
int index[VL];
for(int i=0; i<N; i+=VL)
{
    for(int j=i; j<i+VL; j++)
        index[j-i] = (int)(age[j] / 20);
    for(int j=0; j<VL; j++)
        hist[index[j]]++;
}
```

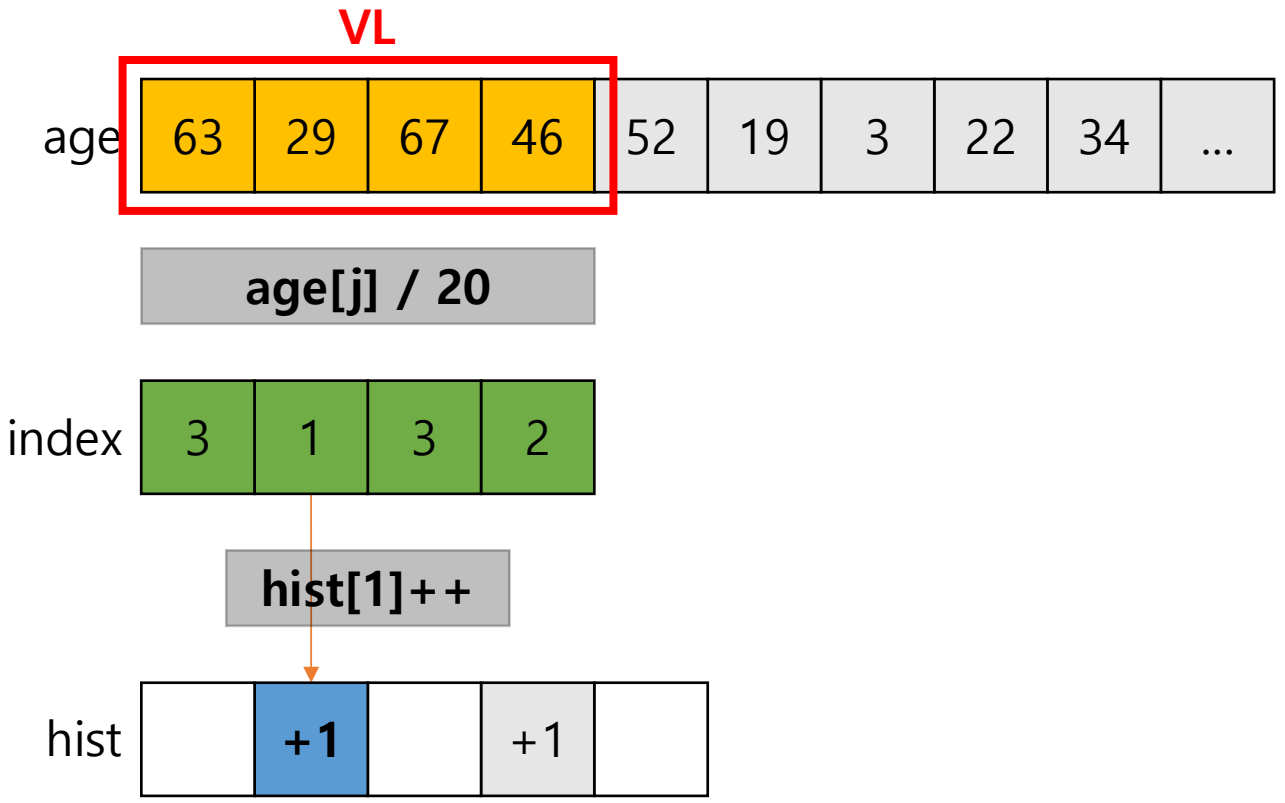


Example 3: Histogram

Cache & Vectorization friendly code

```
for(int i=0; i<N; i++)
{
    int index = (int)(age[i] / 20);
    hist[index]++;
}
```

```
int index[VL];
for(int i=0; i<N; i+=VL)
{
    for(int j=i; j<i+VL; j++)
        index[j-i] = (int)(age[j] / 20);
    for(int j=0; j<VL; j++)
        hist[index[j]]++;
}
```

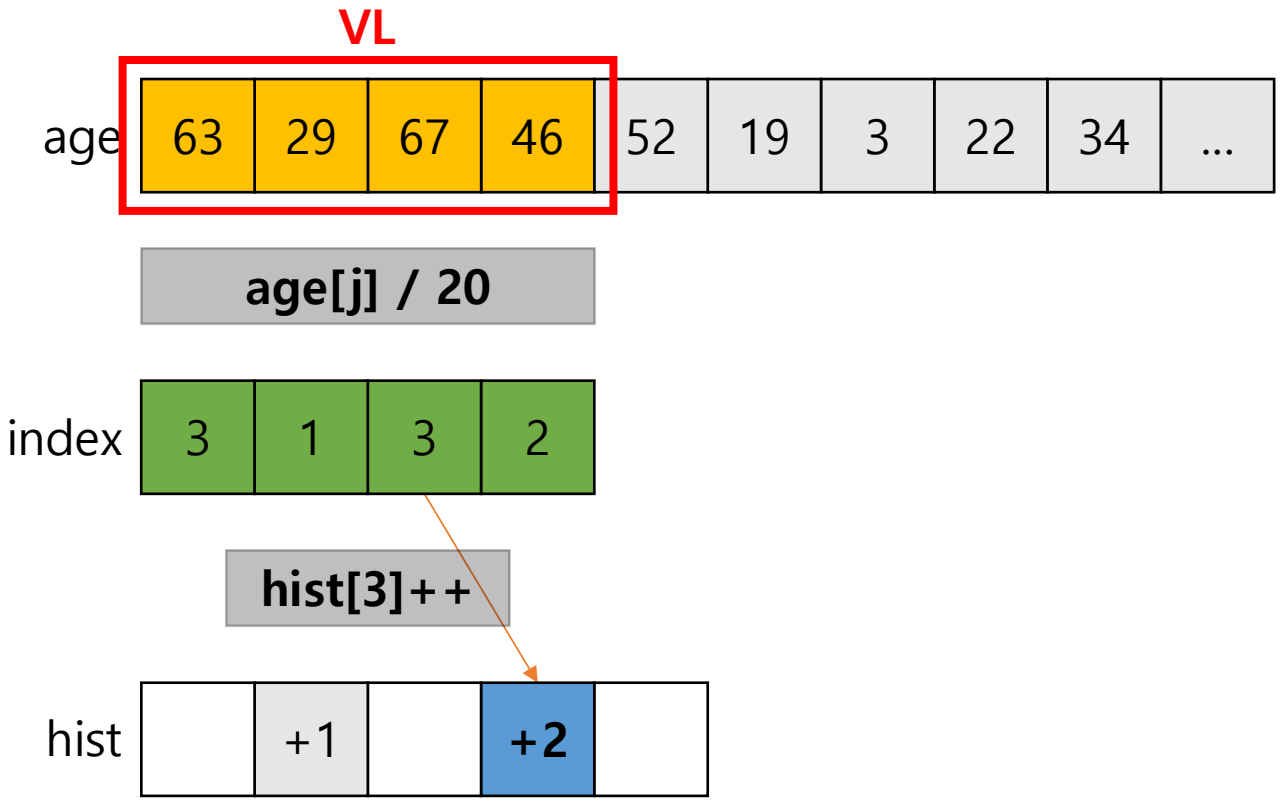


Example 3: Histogram

Cache & Vectorization friendly code

```
for(int i=0; i<N; i++)
{
    int index = (int)(age[i] / 20);
    hist[index]++;
}
```

```
int index[VL];
for(int i=0; i<N; i+=VL)
{
    for(int j=i; j<i+VL; j++)
        index[j-i] = (int)(age[j] / 20);
    for(int j=0; j<VL; j++)
        hist[index[j]]++;
}
```

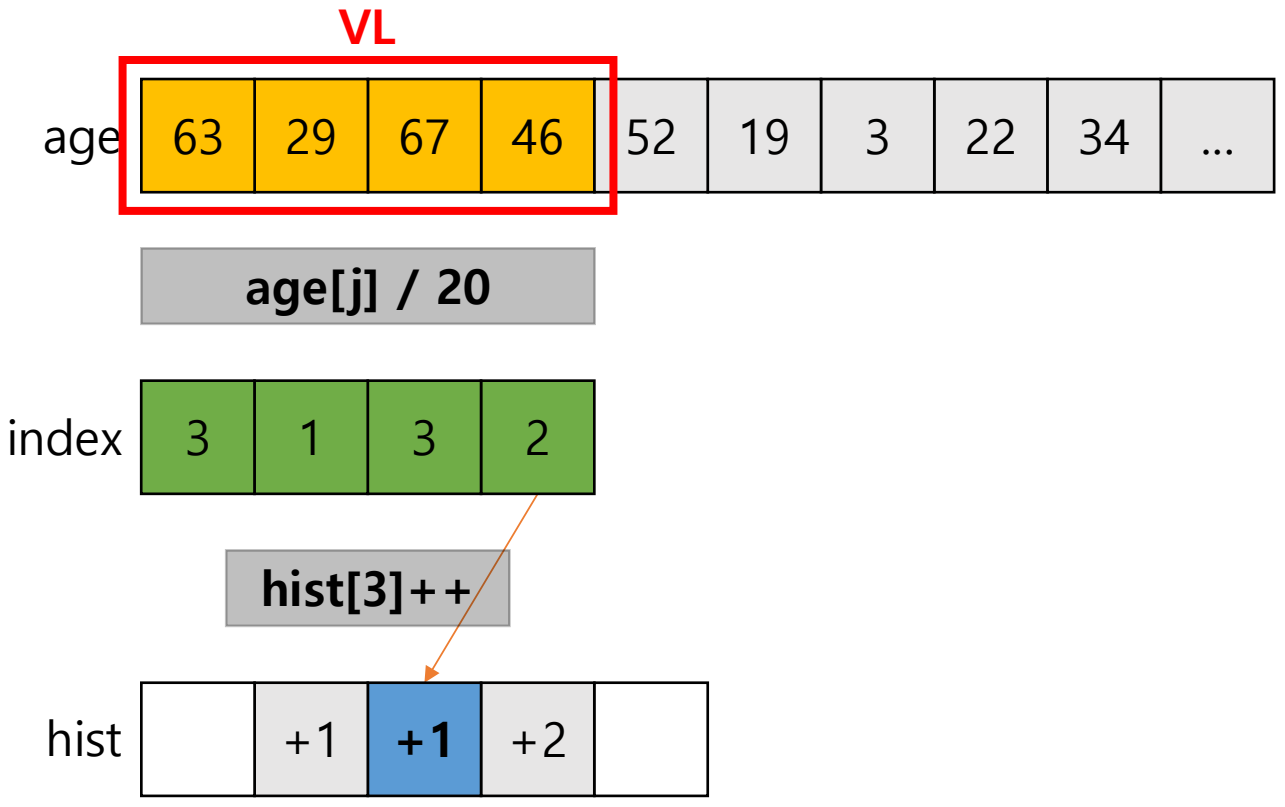


Example 3: Histogram

Cache & Vectorization friendly code

```
for(int i=0; i<N; i++)
{
    int index = (int)(age[i] / 20);
    hist[index]++;
}
```

```
int index[VL];
for(int i=0; i<N; i+=VL)
{
    for(int j=i; j<i+VL; j++)
        index[j-i] = (int)(age[j] / 20);
    for(int j=0; j<VL; j++)
        hist[index[j]]++;
}
```

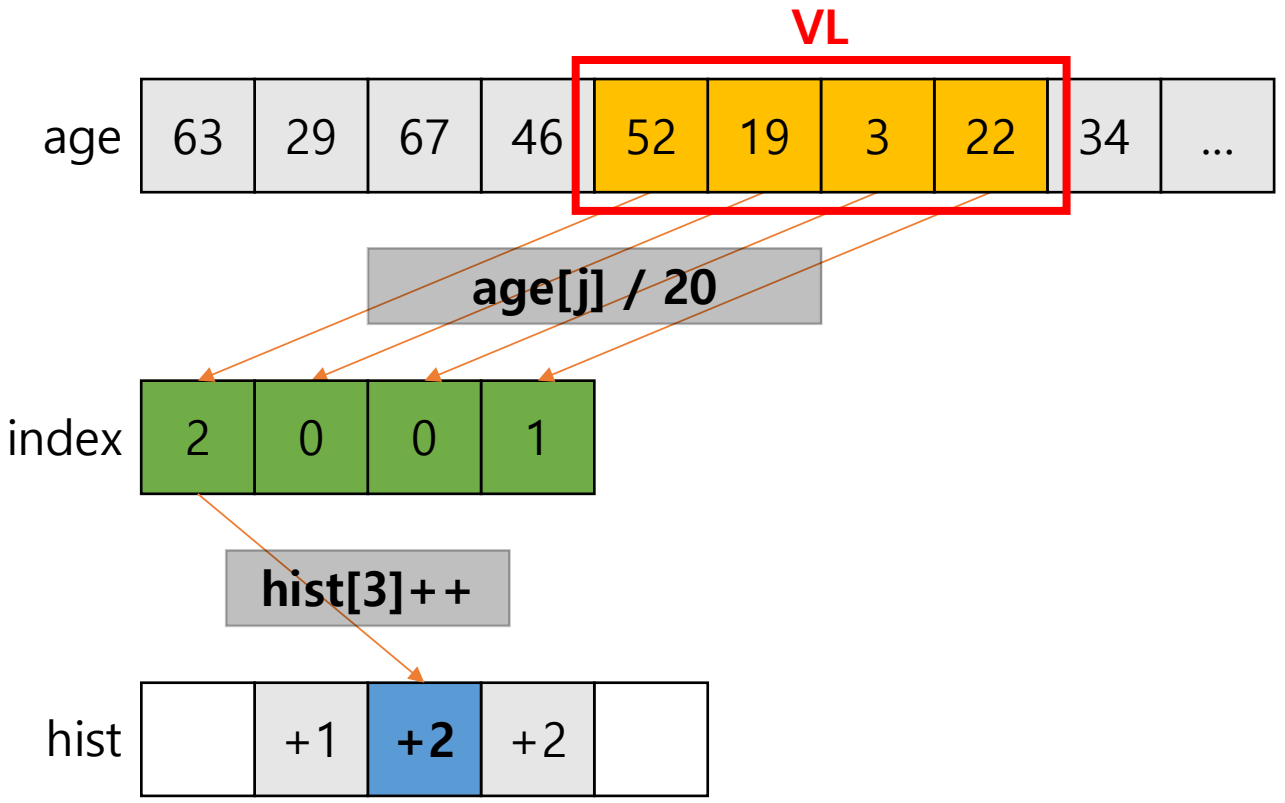


Example 3: Histogram

Cache & Vectorization friendly code

```
for(int i=0; i<N; i++)
{
    int index = (int)(age[i] / 20);
    hist[index]++;
}
```

```
int index[VL];
for(int i=0; i<N; i+=VL)
{
    for(int j=i; j<i+VL; j++)
        index[j-i] = (int)(age[j] / 20);
    for(int j=0; j<VL; j++)
        hist[index[j]]++;
}
```



Example 3: Histogram

Source code – Histogram.c

```
01  #include <stdio.h>
02  #include <time.h>
03  #include <omp.h>
04  #define N 960000000
05  #define VL 512
06
07  int main(int argc, char *argv[]) {
08      srand(time(NULL));
09      double time;
10      int *age = (int*)_mm_malloc(sizeof(int)*N, 64);
11      int hist[5];
12
13      int randomNum = 0;
14      #pragma omp parallel
15      {
16          randomNum = rand() % 100;
17          #pragma omp for simd
18          #pragma vector aligned
19          for(int i=0; i<N; i++)
20              age[i] = randomNum;
21      }
22  ////////////////////////////////////////////////
23      for(int i=0; i<5; i++) hist[i] = 0;
24      time = -omp_get_wtime();
25      for(int i=0; i<N; i++) {
26          int index = (int)(age[i] / 20);
27          hist[index]++;
28      }
29      time += omp_get_wtime();
30      printf("\t1 Histogram time: %lf (secs)\n", time);
31      for(int i=0; i<5; i++) printf("\t\t%d\n", hist[i]);
32      printf("\n");
```

```
33  ////////////////////////////////////////////////
34      for(int i=0; i < 5; i++) hist[i] = 0;
35      time = -omp_get_wtime();
36      #pragma omp parallel
37      {
38          int *index = (int*)_mm_malloc(sizeof(int)*VL, 64);
39          int hist_private[5];
40          for(int i=0; i<5; i++) hist_private[i] = 0;
41          #pragma omp for
42          for(int i=0; i<N; i+=VL) {
43              #pragma omp simd
44              #pragma vector aligned
45              for(int j=i; j<i+VL; j++)
46                  index[j-i] = (int)(age[j] / 20);
47
48              for(int j=0; j<VL; j++)
49                  hist_private[index[j]]++;
50          }
51          #pragma omp critical
52          {
53              for(int i=0; i<5; i++)
54                  hist[i] += hist_private[i];
55          }
56          _mm_free(index);
57      }
58      time += omp_get_wtime();
59      printf("\t2 Histogram time: %lf (secs)\n", time);
60      for(int i=0; i<5; i++) printf("\t\t%d\n", hist[i]);
61      printf("\n");
```

```
62  ////////////////////////////////////
63      _mm_free(age);
64
65      return 0;
66  }
```

Example 3: Histogram

Results

```
[root@node01 03_Histogram]# ./Histogram_O2.ex
        1 Histogram time: 11.912648 (secs)
                191250000
                251250000
                157500000
                142500000
                217500000

        2 Histogram time: 0.159659 (secs)
                191250000
                251250000
                157500000
                142500000
                217500000
```

```
[root@node01 03_Histogram]# ./Histogram_O2_AVX512.ex
        1 Histogram time: 11.880893 (secs)
                112500000
                285000000
                202500000
                153750000
                206250000

        2 Histogram time: 0.057357 (secs)
                112500000
                285000000
                202500000
                153750000
                206250000
```

```
[root@node01 03_Histogram]# ./Histogram_O3_AVX512.ex
        1 Histogram time: 16.631359 (secs)
                172500000
                247500000
                202500000
                187500000
                150000000

        2 Histogram time: 0.085372 (secs)
                172500000
                247500000
                202500000
                187500000
                150000000
```

???

???

Example 3: Histogram

Optimization Report

```
 71 LOOP BEGIN at Histogram.c(25,2)
 72    remark #15540: loop was not vectorized: auto-vectorization is disabled with -no-
 73 LOOP END
```

Let's Check this code line

```
 94 LOOP BEGIN at Histogram.c(25,2)
 95    remark #15388: vectorization support: reference age[i] has aligned access   [ Hi
 96    remark #15416: vectorization support: irregularly indexed store was generated fo
 97    remark #15415: vectorization support: irregularly indexed load was generated for
 98    remark #15305: vectorization support: vector length 16
 99    remark #15309: vectorization support: normalized vectorization overhead 0.039
100    remark #15300: LOOP WAS VECTORIZED
101    remark #15442: entire loop may be executed in remainder
102    remark #15448: unmasked aligned unit stride loads: 1
103    remark #15462: unmasked indexed (or gather) loads: 1
104    remark #15463: unmasked indexed (or scatter) stores: 1
105    remark #15475: --- begin vector cost summary ---
106    remark #15476: scalar cost: 25
107    remark #15477: vector cost: 15.930
108    remark #15478: estimated potential speedup: 1.560
109    remark #15482: vectorized math library calls: 1
110    remark #15488: --- end vector cost summary ---
111    remark #15499: histogram: 2
112    remark #25015: Estimate of max trip count of loop=60000000
113
114    LOOP BEGIN at <compiler generated>
115       remark #25460: No loop optimizations reported
116    LOOP END
117 LOOP END
118
119 LOOP BEGIN at Histogram.c(25,2)
120 <Remainder loop for vectorization>
121    remark #15389: vectorization support: reference age[i] has unaligned access   [
122    remark #15381: vectorization support: unaligned access used inside loop body
123    remark #15305: vectorization support: vector length 16
124    remark #15309: vectorization support: normalized vectorization overhead 0.055
125    remark #15301: REMAINDER LOOP WAS VECTORIZED
126
127    LOOP BEGIN at <compiler generated>
128       remark #25460: No loop optimizations reported
129    LOOP END
130 LOOP END
```

Example 3: Histogram

Optimization Report

```
156     LOOP BEGIN at Histogram.c(48,4)
157         remark #15540: loop was not vectorized: auto-vectorization is disabled with -
158         remark #25438: unrolled without remainder by 2
159     LOOP END
160 LOOP END
```

Let's Check this code line

```
248     LOOP BEGIN at Histogram.c(48,4)
249         remark #15388: vectorization support: reference index[j] has aligned access
250         remark #15388: vectorization support: reference index[j] has aligned access
251         remark #15416: vectorization support: irregularly indexed store was generated
252         remark #15415: vectorization support: irregularly indexed load was generated
253         remark #15305: vectorization support: vector length 16
254         remark #15300: LOOP WAS VECTORIZED
255         remark #15448: unmasked aligned unit stride loads: 1
256         remark #15462: unmasked indexed (or gather) loads: 1
257         remark #15463: unmasked indexed (or scatter) stores: 1
258         remark #15475: --- begin vector cost summary ---
259         remark #15476: scalar cost: 10
260         remark #15477: vector cost: 9.370
261         remark #15478: estimated potential speedup: 1.060
262         remark #15488: --- end vector cost summary ---
263         remark #15499: histogram: 2
264         remark #25015: Estimate of max trip count of loop=32
265
266         LOOP BEGIN at <compiler generated>
267            remark #25460: No loop optimizations reported
268         LOOP END
269     LOOP END
270 LOOP END
```

Example 4: Loop Dependency

Loop Dependency and Vectorization

```
#define N 100000000

int *a = malloc(sizeof *a * (N + 1));

for (int i = 0; i < N; i++)
  a[i] = a[i + 1];

for (int i = 1; i <= N; i++)
  a[i] = a[i - 1];
```

Example 4: Loop Dependency

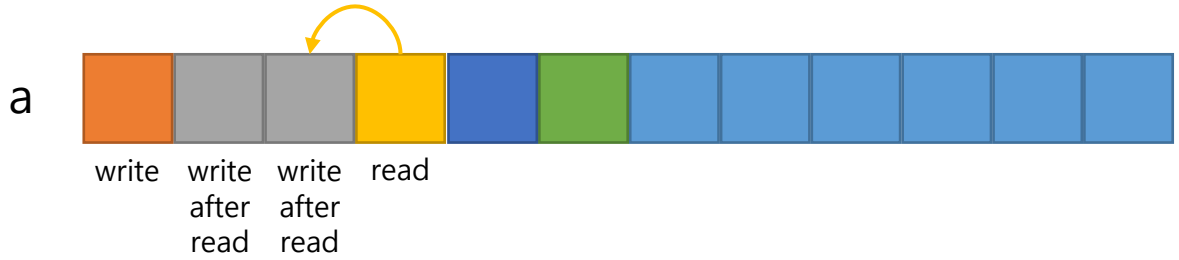
Loop Dependency and Vectorization

```
#define N 100000000

int *a = malloc(sizeof *a * (N + 1));

for (int i = 0; i < N; i++)
  a[i] = a[i + 1];

for (int i = 1; i <= N; i++)
  a[i] = a[i - 1];
```

Example 4: Loop Dependency

Loop Dependency and Vectorization

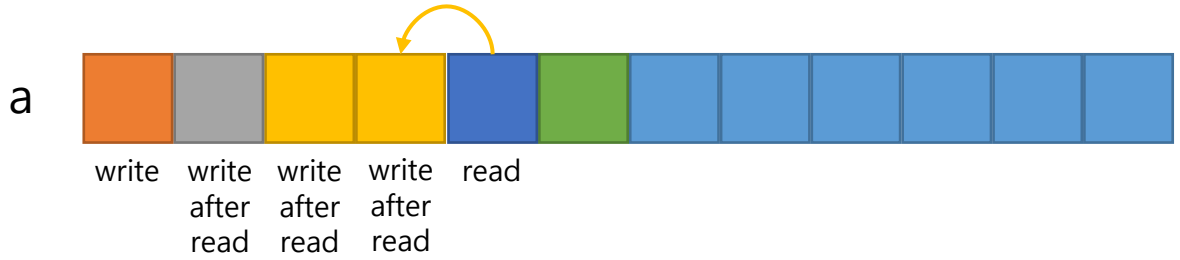
```
#define N 100000000

int *a = malloc(sizeof *a * (N + 1));

for (int i = 0; i < N; i++)
  a[i] = a[i + 1];

for (int i = 1; i <= N; i++)
  a[i] = a[i - 1];
```



Example 4: Loop Dependency

Loop Dependency and Vectorization

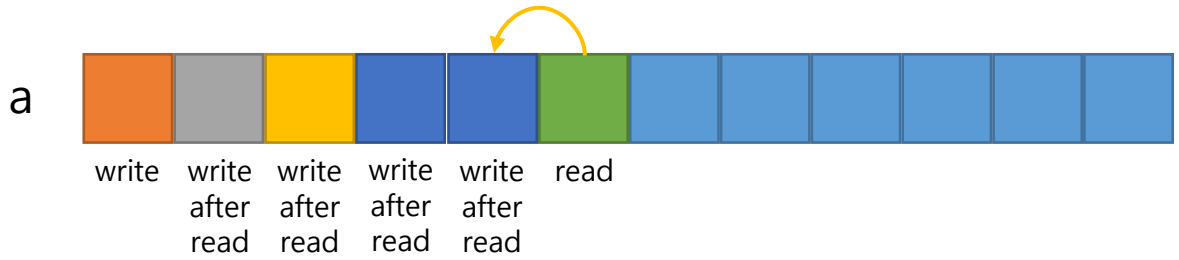
```
#define N 100000000

int *a = malloc(sizeof *a * (N + 1));

for (int i = 0; i < N; i++)
  a[i] = a[i + 1];

for (int i = 1; i <= N; i++)
  a[i] = a[i - 1];
```



Example 4: Loop Dependency

Loop Dependency and Vectorization

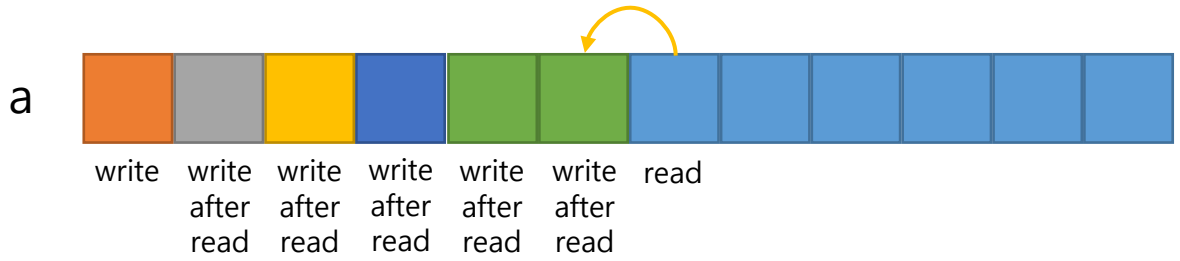
```
#define N 100000000

int *a = malloc(sizeof *a * (N + 1));

for (int i = 0; i < N; i++)
  a[i] = a[i + 1];

for (int i = 1; i <= N; i++)
  a[i] = a[i - 1];
```



Example 4: Loop Dependency

Loop Dependency and Vectorization

```
#define N 100000000

int *a = malloc(sizeof *a * (N + 1));

for (int i = 0; i < N; i++)
  a[i] = a[i + 1];

for (int i = 1; i <= N; i++)
  a[i] = a[i - 1];
```



Example 4: Loop Dependency

Loop Dependency and Vectorization

```
#define N 100000000

int *a = malloc(sizeof *a * (N + 1));

for (int i = 0; i < N; i++)
  a[i] = a[i + 1];

for (int i = 1; i <= N; i++)
  a[i] = a[i - 1];
```

Example 4: Loop Dependency

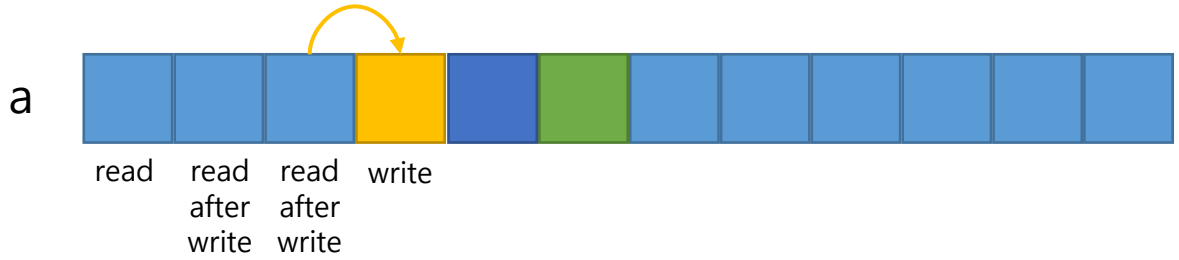
Loop Dependency and Vectorization

```
#define N 100000000

int *a = malloc(sizeof *a * (N + 1));

for (int i = 0; i < N; i++)
  a[i] = a[i + 1];

for (int i = 1; i <= N; i++)
  a[i] = a[i - 1];
```

Example 4: Loop Dependency

Loop Dependency and Vectorization

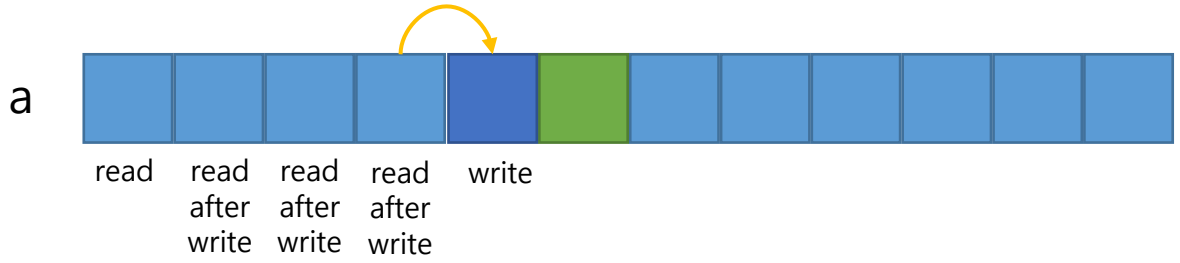
```
#define N 100000000

int *a = malloc(sizeof *a * (N + 1));

for (int i = 0; i < N; i++)
  a[i] = a[i + 1];

for (int i = 1; i <= N; i++)
  a[i] = a[i - 1];
```



Example 4: Loop Dependency

Loop Dependency and Vectorization

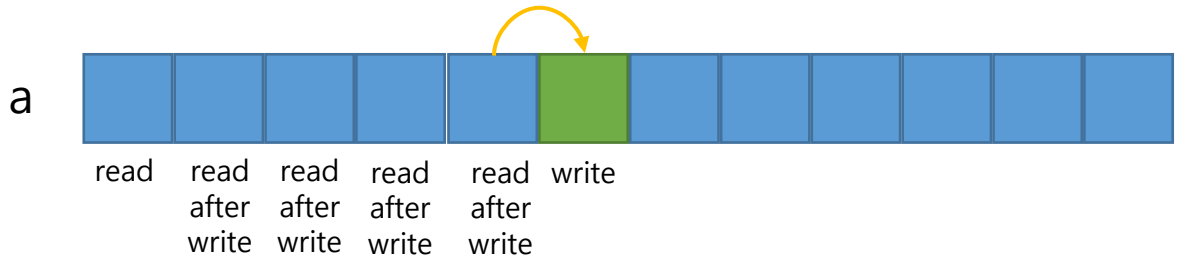
```
#define N 100000000

int *a = malloc(sizeof *a * (N + 1));

for (int i = 0; i < N; i++)
  a[i] = a[i + 1];

for (int i = 1; i <= N; i++)
  a[i] = a[i - 1];
```



Example 4: Loop Dependency

Loop Dependency and Vectorization

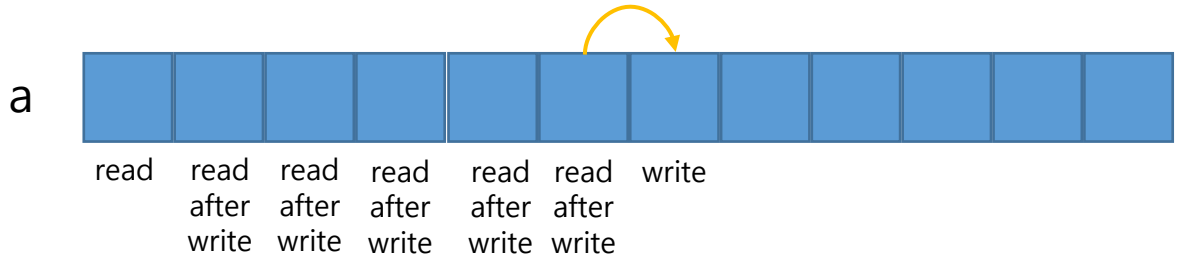
```
#define N 100000000

int *a = malloc(sizeof *a * (N + 1));

for (int i = 0; i < N; i++)
  a[i] = a[i + 1];

for (int i = 1; i <= N; i++)
  a[i] = a[i - 1];
```



Example 4: Loop Dependency

Loop Dependency and Vectorization

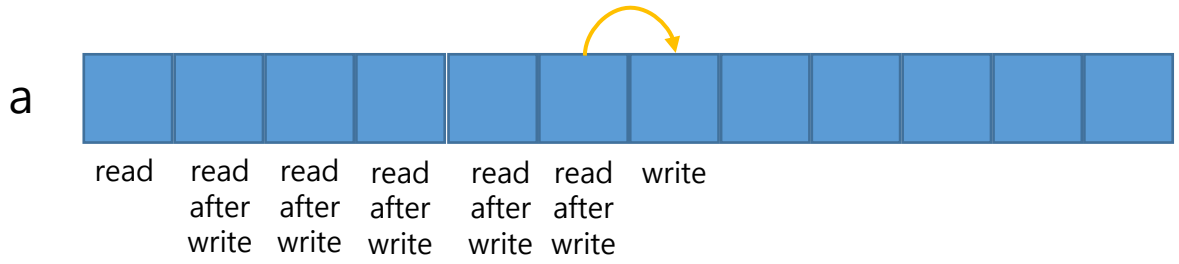
```
#define N 100000000

int *a = malloc(sizeof *a * (N + 1));

for (int i = 0; i < N; i++)
  a[i] = a[i + 1];

for (int i = 1; i <= N; i++)
  a[i] = a[i - 1];
```



Example 4: Loop Dependency

Loop Dependency and Vectorization

```
#define N 100000000

int *a = malloc(sizeof *a * (N + 1));

for (int i = 0; i < N; i++)
  a[i] = a[i + 1];

for (int i = 1; i <= N; i++)
  a[i] = a[i - 1];
```



WAR : write after read
RAW : read after write
Which one is vectorizable?

Example 4: Loop Dependency

Source Code

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3
 4 #define N 100000000
 5
 6 void init(double *a, int n)
 7 {
 8     for (size_t i = 0; i < n; i++)
 9         a[i] = (double)i;
10 }
11
12 double result(double *a, int n)
13 {
14     double ret = 0.;
15     for (size_t i = 0; i < n; i++)
16         ret += a[i];
17     return ret;
18 }
19
20 int main()
21 {
22     double *a = malloc(sizeof *a * N);
23
```

```
24     init(a, N);
25     for (size_t i = 0; i < N - 1; i++)
26         a[i] = a[i + 1];
27     printf("write after read : %e\n", result(a, N));
28
29     init(a, N);
30     for (size_t i = 1; i < N; i++)
31         a[i] = a[i - 1];
32     printf("read after write : %e\n", result(a, N));
33
34     init(a, N);
35     #pragma simd
36     for (size_t i = 0; i < N - 1; i++)
37         a[i] = a[i + 1];
38     printf("write after read (simd) : %e\n", result(a, N));
39
40     init(a, N);
41     #pragma simd
42     for (size_t i = 1; i < N; i++)
43         a[i] = a[i - 1];
44     printf("read after write (simd) : %e\n", result(a, N));
45
46     free(a);
47     return 0;
48 }
```

Example 4: Loop Dependency

Results of Code Run

- `icc -std=c99 -qopt-report=5 -xMIC-AVX512 -o loop loop.c`
  - $ ./loop
  - write after read : 5.000000e+15
  - read after write : 0.000000e+00
  - write after read (simd) : 5.000000e+15
  - read after write (simd) : 5.000000e+15

Incorrect!

Example 4: Loop Dependency

Optimization Report

- LOOP BEGIN at loop.c(30,5)
- remark #25401: memcopy(with guard) generated
- remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

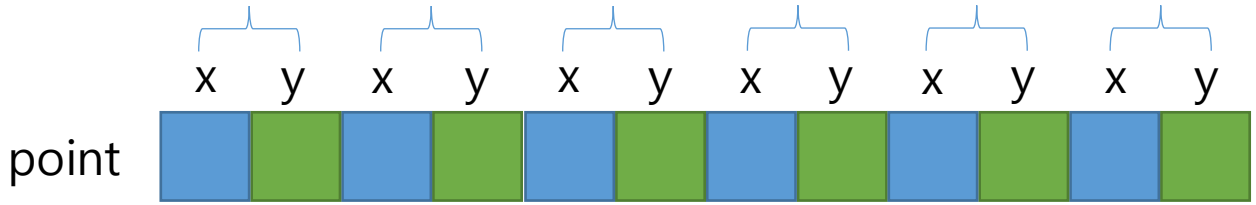
- LOOP BEGIN at loop.c(30,5)
- <Multiversioned v2>
- remark #15304: loop was not vectorized: non-vectorizable loop instance from multiversioning
- remark #25439: unrolled with remainder by 2
- remark #25456: Number of Array Refs Scalar Replaced In Loop: 2
- LOOP END

- LOOP BEGIN at loop.c(30,5)
- <Remainder, Multiversioned v2>
- LOOP END
- LOOP END

Example 5: Structure of Array vs Array of Structure

SOA and Vectorization

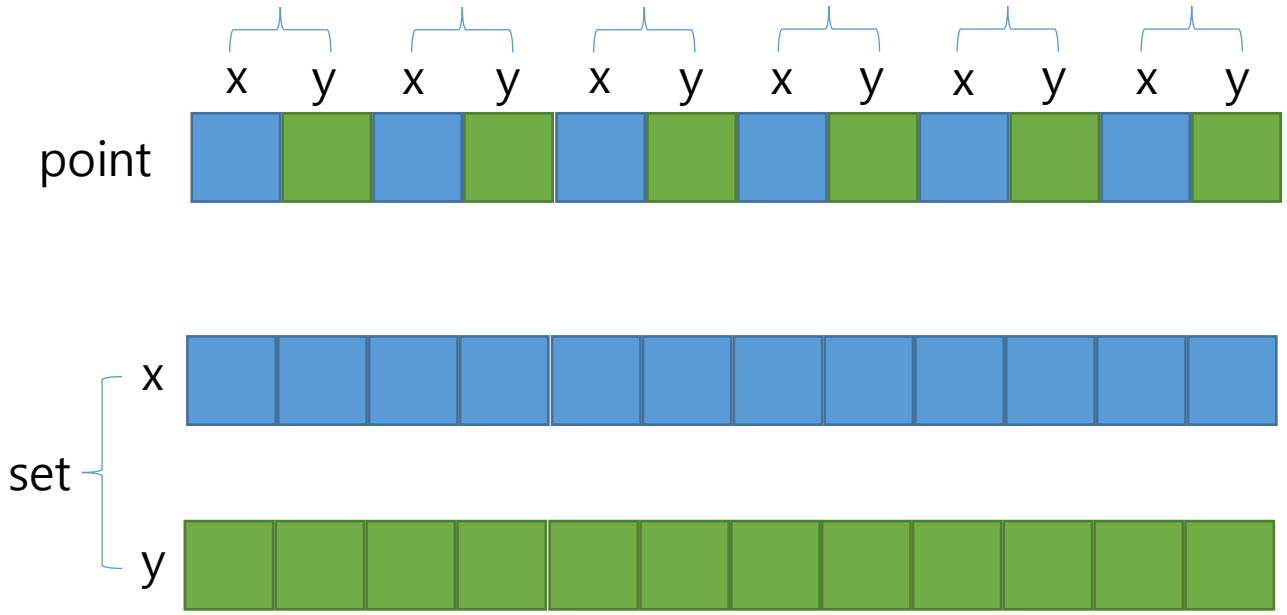
```
#define N 100000000

struct {
  double x;
  double y;
} *point = malloc(sizeof *point * N);

struct {
  double *x;
  double *y;
} set;
set.x = malloc(sizeof *(set.x) * N);
set.y = malloc(sizeof *(set.y) * N);
```



Example 5: Structure of Array vs Array of Structure

SOA and Vectorization

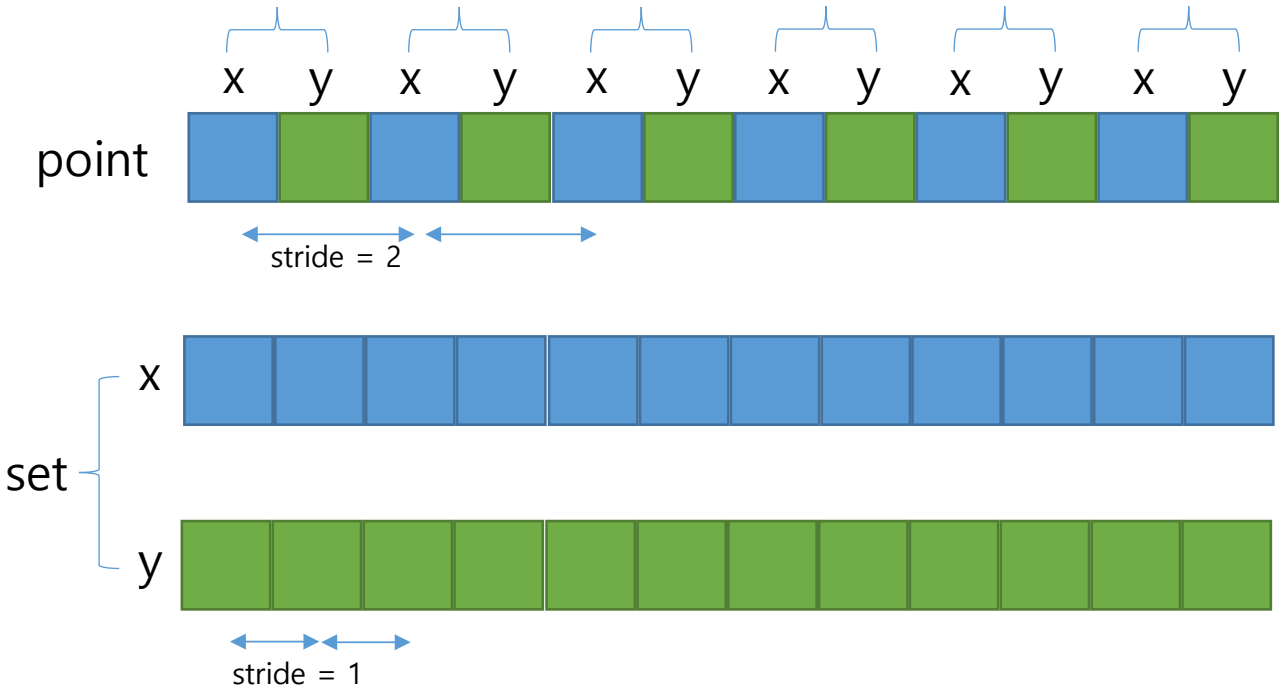
```
#define N 100000000

struct {
  double x;
  double y;
} *point = malloc(sizeof *point * N);

struct {
  double *x;
  double *y;
} set;
set.x = malloc(sizeof *(set.x) * N);
set.y = malloc(sizeof *(set.y) * N);
```



Example 5: Structure of Array vs Array of Structure

SOA and Vectorization

```
#define N 100000000

struct {
  double x;
  double y;
} *point = malloc(sizeof *point * N);

struct {
  double *x;
  double *y;
} set;
set.x = malloc(sizeof *(set.x) * N);
set.y = malloc(sizeof *(set.y) * N);
```



Example 5: Structure of Array vs Array of Structure

Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 100000000

int main()
{

    struct { double x; double y; } *point = malloc(sizeof *point * N);
    struct { double *x; double *y; } set;
    set.x = malloc(sizeof *(set.x) * N);
    set.y = malloc(sizeof *(set.y) * N);

    for (int i = 0; i < N; i++) {
        point[i].x = (double)(i);
        point[i].y = 0.;
        set.x[i] = (double)(i);
        set.y[i] = 0.;
    }

    double time = -omp_get_wtime();
    for (int i = 0; i < N; i++)
        point[i].y = 2. * point[i].x;
    time += omp_get_wtime();
    printf("Array of Structure: %f (secs)\n", time);

    time = -omp_get_wtime();
    for (int i = 0; i < N; i++)
        set.y[i] = 2. * set.x[i];
    time += omp_get_wtime();
    printf("Structure of Array: %f (secs)\n", time);

    free(point);
    free(set.x);
    free(set.y);

    return 0;
}
```

Example 5: Structure of Array vs Array of Structure

Performance on Xeon Phi Knights Landing 7210 (64 cores, 4 HyperT/core) without MCDRAM

- `icc -std=c99 -qopt-report=5 -xMIC-AVX512 -o soa soa.c -lgomp`
  - $ ./soa
  - Array of Structure: 0.262025 (secs)
  - Structure of Array: 0.123625 (secs)

Example 5: Structure of Array vs Array of Structure

Optimization Report

- LOOP BEGIN at soa.c(23,5)
- remark #15416: vectorization support: non-unit strided store was generated for the variable <point->y[i]>, stride is 2 [soa.c(24,9)]
- remark #15415: vectorization support: non-unit strided load was generated for the variable <point->x[i]>, stride is 2 [soa.c(24,27)]
- remark #15305: vectorization support: vector length 16
- remark #15399: vectorization support: unroll factor set to 2
- remark #15300: LOOP WAS VECTORIZED
- remark #15452: unmasked strided loads: 1
- remark #15453: unmasked strided stores: 1
- remark #15475: --- begin vector cost summary ---
- remark #15476: scalar cost: 7
- remark #15477: vector cost: 4.180
- remark #15478: estimated potential speedup: 1.670
- remark #15488: --- end vector cost summary ---
- remark #25015: Estimate of max trip count of loop=3125000
- LOOP END

- LOOP BEGIN at soa.c(29,5)
- remark #15388: vectorization support: reference set.y[i] has aligned access [soa.c(30,9)]
- remark #15389: vectorization support: reference set.x[i] has unaligned access [soa.c(30,25)]
- remark #15381: vectorization support: unaligned access used inside loop body
- remark #15412: vectorization support: streaming store was generated for set.y[i] [soa.c(30,9)]
- remark #15412: vectorization support: streaming store was generated for set.y[i] [soa.c(30,9)]
- remark #15305: vectorization support: vector length 16
- remark #15309: vectorization support: normalized vectorization overhead 1.182
- remark #15300: LOOP WAS VECTORIZED
- remark #15442: entire loop may be executed in remainder
- remark #15449: unmasked aligned unit stride stores: 1
- remark #15450: unmasked unaligned unit stride loads: 1
- remark #15467: unmasked aligned streaming stores: 2
- remark #15475: --- begin vector cost summary ---
- remark #15476: scalar cost: 7
- remark #15477: vector cost: 0.680
- remark #15478: estimated potential speedup: 10.180
- remark #15488: --- end vector cost summary ---
- remark #25015: Estimate of max trip count of loop=6250000
- LOOP END

Example 6: Pi Calculation

Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

#define NTRIAL 100000000

int main()
{
    double *x = malloc(sizeof *x * NTRIAL);
    double *y = malloc(sizeof *y * NTRIAL);
    int *z = malloc(sizeof *z * NTRIAL);
    double time;
    unsigned short seed[3] = {155, 0, 155};
    seed48(&seed[0]);

    int count = 0;
    time = -omp_get_wtime();
    for (size_t i = 0; i < NTRIAL; i++) {
        double xx = 2. * drand48() - 1.;
        double yy = 2. * drand48() - 1.;
        if (xx*xx + yy*yy <= 1.) count++;
    }
    time += omp_get_wtime();
    printf("value: %e\ncore-time: %f(secs)\n\n", 4.*count/NTRIAL, time);
```

```c
    time = -omp_get_wtime();
    count = 0;
    for (size_t i = 0; i < NTRIAL; i++)
        x[i] = 2. * drand48() - 1.;
    for (size_t i = 0; i < NTRIAL; i++)
        y[i] = 2. * drand48() - 1.;
    for (size_t i = 0; i < NTRIAL; i++)
        if (x[i]*x[i] + y[i]*y[i] <= 1.) count++;
    time += omp_get_wtime();
    printf("value: %e\ncore-time: %f(secs)\n\n", 4.*count/NTRIAL, time);

    time = -omp_get_wtime();
    count = 0;
    for (size_t i = 0; i < NTRIAL; i++)
        x[i] = 2. * drand48() - 1.;
    for (size_t i = 0; i < NTRIAL; i++)
        y[i] = 2. * drand48() - 1.;
    for (size_t i = 0; i < NTRIAL; i++)
        z[i] = 1 - floor(x[i]*x[i] + y[i]*y[i]);
    for (size_t i = 0; i < NTRIAL; i++)
        count += z[i];
    time += omp_get_wtime();
    printf("value: %e\ncore-time: %f(secs)\n\n", 4.*count/NTRIAL, time);

    free(x);
    free(y);
    free(z);
    return 0;
}
```

Example 6: Pi Calculation

Performance on Xeon Phi Knights Landing 7210 (64 cores, 4 HyperT/core) without MCDRAM

- `icc -std=c99 -qopt-report=5 -xMIC-AVX512 -o pi pi.c -lgomp -D_GNU_SOURCE`
  - $ ./pi
  - value: 3.141697e+00
  - core-time: 2.681885(secs)

  - value: 3.141340e+00
  - core-time: 1.370413(secs)

  - value: 3.141749e+00
  - core-time: 1.226880(secs)

Example 6: Pi Calculation

Optimization Report

- LOOP BEGIN at pi.c(19,5)
- remark #15382: vectorization support: call to function drand48(void) cannot be vectorized [pi.c(20,26)]
- remark #15382: vectorization support: call to function drand48(void) cannot be vectorized [pi.c(21,26)]
- remark #15344: loop was not vectorized: vector dependence prevents vectorization
- remark #15346: vector dependence: assumed OUTPUT dependence between call:drand48(void) (20:26) and call:drand48(void) (21:26)
- remark #15346: vector dependence: assumed OUTPUT dependence between call:drand48(void) (21:26) and call:drand48(void) (20:26)
- LOOP END

- LOOP BEGIN at pi.c(33,5)
- remark #15305: vectorization support: vector length 16
- remark #15309: vectorization support: normalized vectorization overhead 1.467
- remark #15300: LOOP WAS VECTORIZED
- remark #15442: entire loop may be executed in remainder
- remark #15448: unmasked aligned unit stride loads: 1
- remark #15450: unmasked unaligned unit stride loads: 1
- remark #15475: --- begin vector cost summary ---
- remark #15476: scalar cost: 21
- remark #15477: vector cost: 1.870
- remark #15478: estimated potential speedup: 11.190
- remark #15488: --- end vector cost summary ---
- remark #25015: Estimate of max trip count of loop=6250000
- LOOP END

- LOOP BEGIN at pi.c(44,5)
- remark #15389: vectorization support: reference x[i] has unaligned access [pi.c(45,26)]
- remark #15389: vectorization support: reference x[i] has unaligned access [pi.c(45,31)]
- remark #15389: vectorization support: reference y[i] has unaligned access [pi.c(45,38)]
- remark #15389: vectorization support: reference y[i] has unaligned access [pi.c(45,43)]
- remark #15388: vectorization support: reference z[i] has aligned access [pi.c(45,9)]
- remark #15381: vectorization support: unaligned access used inside loop body
- remark #15412: vectorization support: streaming store was generated for z[i] [pi.c(45,9)]
- remark #15412: vectorization support: streaming store was generated for z[i] [pi.c(45,9)]
- remark #15305: vectorization support: vector length 32
- remark #15309: vectorization support: normalized vectorization overhead 0.050
- remark #15300: LOOP WAS VECTORIZED
- remark #15442: entire loop may be executed in remainder
- remark #15449: unmasked aligned unit stride stores: 1
- remark #15450: unmasked unaligned unit stride loads: 2
- remark #15467: unmasked aligned streaming stores: 2
- remark #15475: --- begin vector cost summary ---
- remark #15476: scalar cost: 120
- remark #15477: vector cost: 10.680
- remark #15478: estimated potential speedup: 11.220
- remark #15482: vectorized math library calls: 1
- remark #15487: type converts: 1
- remark #15488: --- end vector cost summary ---
- remark #25015: Estimate of max trip count of loop=3125000
- LOOP END