Explicit Vector Programming

November 2021

Agenda

- Limitations of Auto-Vectorization
- OpenMP* SIMD directive
  - Syntax, examples
- OpenMP SIMD functions
  - Syntax, examples
- Special Idioms
  - Compress, histogram patterns

Obstacles to Auto-Vectorization

- Multiple loop exits
  - Or trip count unknown at loop entry
- Dependencies between loop iterations
  - Mostly, avoid read-after-write “flow” dependencies
- Function or subroutine calls
  - Except where inlined
- Nested (Outer) loops
  - Unless inner loop fully unrolled
- Complexity
  - Too many branches
  - Too hard or time-consuming for compiler to analyze

software.intel.com/articles/requirements-for-vectorizable-loops

OpenMP* SIMD Programming

Vectorization is so important
➔ consider explicit vector programming

Modeled on OpenMP* for threading (explicit parallel programming)

Enables reliable vectorization of complex loops the compiler can't auto-vectorize
E.g. outer loops

Directives are commands to the compiler, not hints
#pragma omp simd

Compiler does no dependency and cost-benefit analysis !!
Programmer is responsible for correctness (like OpenMP threading)
E.g. PRIVATE, REDUCTION or ORDERED clauses

Incorporated in OpenMP since version 4.0 ⇒ portable
-qopenmp or -qopenmp-simd to enable

OpenMP* SIMD pragma

- Use #pragma omp simd (-qopenmp-simd enabled by default)

```
void addit(double* a, double* b, int m,
           int n, int x)
{
  for (int i = m; i < m+n; i++) {
        a[i] = b[i] + a[i-x];
  }
}
```

```
void addit(double* a, double* b, int m,
           int n, int x)
{
#pragma omp simd  // I know x<0
  for (int i = m; i < m+n; i++) {
        a[i] = b[i] + a[i-x];
  }
}
```

- Loop was multiversioned
  - With more pointers the compiler will not do it
- Use SIMD pragma when you **KNOW** that a given loop is safe to vectorize

  The Intel® Compiler will vectorize if at all possible
  - (ignoring dependency or efficiency concerns)
  - Minimizes source code changes needed to enforce vectorization

godbolt.org/z/c71ddYaMj

Clauses for OMP SIMD directives

The programmer (i.e. you!) is responsible for correctness

- Just like for race conditions in loops with OpenMP* threading

Available clauses:

- PRIVATE
- LASTPRIVATE
- REDUCTION
- COLLAPSE

} like OpenMP for threading (for nested loops)

- LINEAR (additional induction variables)
- SIMDLEN (preferred number of iterations to execute concurrently)
- SAFELEN (max iterations that can be executed concurrently)
- ALIGNED (tells compiler about data alignment)

Example: Outer Loop Vectorization

```
#ifdef  KNOWN_TRIP_COUNT
#define MYDIM 3
#else                              // pt     input  vector of points
#define MYDIM nd                   // ptref  input  reference point
#endif                             // dis    output vector of distances
#include <math.h>

void dist( int n, int nd, float pt[][MYDIM], float dis[], float ptref[]) {
/* calculate distance from data points to reference point */

#pragma omp simd
           for (int ipt=0; ipt<n; ipt++) {
                      float d = 0.;

                      for (int j=0; j<MYDIM; j++) {
                                 float t = pt[ipt][j] - ptref[j];
                                 d+= t*t;
                      }

                      dis[ipt] = sqrtf(d);
           }
}
```

Outer loop with high trip count

Inner loop with low trip count

godbolt.org/z/9odrETGhn

Outer Loop Vectorization

-std=c99 -xcore-avx512 -qopt-report-phase=loop,vec -qopt-report-file=stderr

```
...
LOOP BEGIN at <source>(11,2)
  remark #15542: loop was not vectorized: inner loop was already vectorized
...
 LOOP BEGIN at <source>(14,3)
    remark #15300: LOOP WAS VECTORIZED
```

- We can vectorize the outer loop by adding the pragma

```
#pragma omp simd
```

- Would need private clause for d and t if declared outside SIMD scope

```
...
LOOP BEGIN at <source>(11,2)
  remark #15301: SIMD LOOP WAS VECTORIZED
  remark #26013: Compiler has chosen to target XMM/YMM vector. Try using -qopt-zmm-usage=high to override

  LOOP BEGIN at <source>(14,3)
    remark #15548: loop was vectorized along with the outer loop
  LOOP END
LOOP END
...
```

Unrolling the Inner Loop

- If the trip count is fixed and the compiler knows it, the inner loop can be fully unrolled. Outer loop vectorization is more efficient also because stride is now known

- -std=c99 -xcore-avx512 -qopt-report-phase=loop,vec -qopt-report-file=stderr -DKNOWN_TRIP_COUNT

```
LOOP BEGIN at <source>(11,2)
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

  LOOP BEGIN at <source>(14,3)
    remark #25436: completely unrolled by 3   (pre-vector)
  LOOP END
LOOP END
```

Outer Loop Vectorization - performance

- Speed-up you may get by using explicit vectorization

Optimization Options	Speed-up	What's going on
-O1	1.0x	No vectorization
-O2 -xavx	1.5x	Inner loop vectorization
-O2 -xavx	3.5x	Outer loop vectorization unknown stride
-O2 -xavx -DKNOWN_TRIP_COUNT	6.5x	Inner loop fully unrolled known outer loop stride
-O2 -xcore-avx2 -DKNOWN_TRIP_COUNT	7.4x	+ Intel® AVX2 including FMA instructions

OpenMP* SIMD functions

A way to vectorize loops containing calls to functions that can’t be inlined

intel®

Loops Containing Function Calls

Function calls can have side effects that introduce a loop-carried dependency, preventing vectorization

Possible remedies:

- Inlining
  - best for small functions
  - Must be in same source file, or else use -ipo
- OMP SIMD pragma or directive to vectorize rest of loop, while preserving scalar calls to function (last resort)
- SIMD-enabled functions
  - Good for large, complex functions and in contexts where inlining is difficult
  - Call from regular “for” loop

SIMD-enabled Function

Compiler generates SIMD-enabled (vector) version of a scalar function that can be called from a vectorized loop:

- #pragma omp simd may not be needed in simpler cases

```
#pragma omp declare simd uniform(y,z,xp,yp,zp)
float func(float x, float y, float z, float xp, float yp, float zp)
{
float denom = (x-xp)*(x-xp) + (y-yp)*(y-yp) + (z-zp)*(z-zp);
  denom = 1./sqrtf(denom);
  return denom;
}
…
#pragma omp simd  private(x)  reduction(+:sumx)
for (i=1; i<nx; i++) {
    x = x0 + (float) i * h;
    sumx = sumx + func(x, y, z, xp, yp, zp);
  }
```

y, z, xp, yp and zp are constant, x can be a vector

These clauses are required for correctness, just like for OpenMP*

godbolt.org/z/GxrjYa617

Clauses for SIMD-enabled Functions

#pragma omp declare simd

- UNIFORM — argument is never vector
- LINEAR (REF|VAL|UVAL) — additional induction variables use REF(X) when vector argument is passed by reference (Fortran default)
- INBRANCH / NOTINBRANCH — specify whether function will be called conditionally
- SIMDLEN — vector length
- ALIGNED — asserts that listed variables are aligned
- PROCESSOR(cpu) — Intel extension, tells compiler which processor to target, e.g. core_2nd_gen_avx, haswell, knl, skylake_avx512
  NOT controlled by -x... switch, may default to SSE
  Simpler is to target processor specified by -x switch using **-vecabi=cmdtarget**

Processor targeting for SIMD functions

Default ABI requires passing arguments in 128 bit xmm registers

- even with -xCORE-AVX512
- may result in inefficient 128 bit code instead of 256 or 512 bit

#pragma omp declare simd

remark #15347: FUNCTION WAS VECTORIZED with **xmm**, simdlen=**4**, **unmasked**
remark #15347: FUNCTION WAS VECTORIZED with **xmm**, simdlen=**4**, **masked**

Ways to target newer processors with longer vectors:

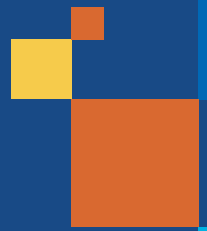
- Compile with **-vecabi=cmdtarget** -xcore-avx512 (simplest!)
  - takes SIMD length from -x switch
- Or use PROCESSOR clause, e.g.

#pragma omp declare simd processor(skylake_avx512) notinbranch

remark #15347: FUNCTION WAS VECTORIZED with **zmm**, simdlen=**16**, **unmasked**

Special Idioms

Compiler must recognize to handle apparent dependencies



intel®

Compress Loop Pattern

Auto-vectorization

godbolt.org/z/MabhfvsMv

```
int compress(double *a, double * __restrict b, int na)
{
    int nb = 0;
    for (int ia=0; ia <na; ia++)
    {
        if (a[ia] > 0.)
            b[nb++] = a[ia];
    }

    return nb;
}
```

VCOMPRESSPD|PS|D|Q	Store sparse packed floating-point values into dense memory
VEXPANDPD|PS|D|Q	Load sparse packed floating-point values from dense memory

double/single-precision/doubleword/quadword

```
vcompresspd YMMWORD PTR [rsi+rax*8]{k1}, ymm1
```

Compress Loop Pattern

godbolt.org/z/MabhfvsMv

Auto-vectorization

Targeting Intel® AVX2

-xcore-avx2 -qopt-report-file=stderr -qopt-report-phase=vec

LOOP BEGIN

remark #15344: loop was not vectorized: vector dependence prevents vectorization.

remark #15346: vector dependence: assumed FLOW dependence between b[nb] (7:4) and a[ia] (7:4)

LOOP END

Targeting Intel® AVX-512

-xcore-avx512 -qopt-report-file=stderr -qopt-report-phase=vec

LOOP BEGIN

remark #15300: LOOP WAS VECTORIZED

LOOP END

```
movsxd      rax, eax
xor         r11d, r11d
kmovw       r8d, k1
popcnt      r11d, r8d
vcompresspd YMMWORD PTR [rsi+rax*8]{k1}, ymm1
add         eax, r11d
```

Key Take Aways

Compress/Expand loop pattern doesn't vectorize on architectures like Intel® AVX2 and the previous ones and does with Intel® AVX-512

Compress Loop Pattern

Complex example

godbolt.org/z/qPTPx4aK7

```
int compress(int n1, int n2, float a[][n2], float b[__restrict])
{
  int nb = 0;
  for (int i1 = 0; i1 < n1; i1++)
  {
    float sc = 0.f;
    for (int i2 = 0; i2 < n2; i2++)
      sc += a[i1][i2];
    if (sc > 0.f)
      b[nb++] = sc;
  }
return nb;
}
```

- Loop was vectorized
- Part with dependencies is scalar (ordered)

Compress Loop Pattern

Complex example

godbolt.org/z/qPTPx4aK7

```
#pragma omp simd
  for (int i1 = 0; i1 < n1; i1++)
  {
    float sc = 0.f;
    for (int i2 = 0; i2 < n2; i2++)
      sc += a[i1][i2];
#pragma omp ordered simd monotonic(nb:1)
    if (sc > 0.f)
      b[nb++] = sc;
  }
```

Key Take Aways

1. Outer loop vectorization can be achieved using OpenMP SIMD pragma.
2. The Compress/Expand loop pattern can hinted to compiler using monotonic clause.
3. The ordered clause takes into account the nb dependency (if omitted, wrong results)

Compress Loop Performance

Optimization Options	Speed-up	What's going on
Simple Loops (–O2 –xCORE-AVX2)	1.0	No vectorization
(-O2 –xCORE-AVX512)	12.8x	Auto-vectorized
Nested Loops (-O2 –xCORE-AVX512)	1.0x	Loop was auto-vectorized as ordered
Monotonic (-O2 –xCORE-AVX512)	4.1x	vcompress instruction used

Key Take Aways

1. Ordered clause will serialize the execution of the compress logic
2. Monotonic clause hints the compiler on the specific loop pattern and helps code generation in picking vcompress/vexpand vector instruction which leads to better performance.

Histogram Loop pattern

Auto-vectorization

godbolt.org/z/vr6qoMvTE

```
for (i=0; i<n; i++)
{
  y = sinf(x[i]*twopi);
  ih = floor((y-bot)*invbinw);
  ih = ih > 0 ? ih : 0;
  ih = ih < nbin ? ih : nbin;
  h[ih] = h[ih] + 1;
}
```

VPCONFLICT instruction detects elements with previous conflicts in a vector of indexes

- Allows to generate a mask with a subset of elements that are gurarnteed to be conflict free

```
vpconflictd zmm1{k1}, zmm2
```

- Store to h is a scatter
- ih can have the same value for different values of i
- Vectorization with a SIMD directive would cause incorrect results

Histogram Loop pattern

godbolt.org/z/vr6qoMvTE

Auto-vectorization

Targeting Intel® AVX2

-xcore-avx2 -qopt-report-file=stderr -qopt-report-phase=vec -qopt-report=3

LOOP BEGIN
remark #15344: loop was not vectorized: vector dependence prevents vectorization.
remark #15346: vector dependence: assumed FLOW dependence between h[ih] (13:5) and h[ih] (13:5)
LOOP END

Targeting Intel® AVX-512

-xcore-avx512 -qopt-report-file=stderr -qopt-report-phase=vec -qopt-report=3

LOOP BEGIN
remark #15300: LOOP WAS VECTORIZED
LOOP END

```
vpminsd   ymm25, ymm4, ymm11                          #12.5
vpconflictd ymm5, ymm25                               #13.5
vpaddd    ymm26, ymm25, DWORD BCST .L_2il0floatpacket.7[rip]
vpgatherdd ymm23{k1}, DWORD PTR [r15+ymm25*4]         #13.1
vpsrld    ymm6, ymm5, 4                               #13.5
```

Key Take Aways

Histogram loop pattern doesn't vectorize on architectures like Intel® AVX2 and the previous ones and does with Intel® AVX512

Histogram Loop pattern

Complex example

godbolt.org/z/hj38Mq7Tq

```
for (int i=0; i<n; i++)
{
  float y = myfun(x[i]);
  int ih = floor( (y-bot)*invbinw );
  ih = ih >= 0 ? ih : 0;
  ih = ih <= nbin-1 ? ih : nbin-1;
  ++contents[ih];
}
```

remark #15543: loop was not vectorized: loop with function call not considered an optimization candidate.

Histogram Loop pattern

Complex example

godbolt.org/z/xee558oac

- Can be vectorized with OpenMP* by:
- Making myfun() a SIMD function
- Using the OMP ORDERED SIMD pragma/directive
- Add the OVERLAP hint to help compiler vectorize more efficiently

```
#pragma omp declare simd
float myfun(float x);

#pragma omp simd
for (int i=0; i<n; i++)
{
  float y = myfun(x[i]);
  int ih = floor( (y-bot)*invbinw );
  ih = ih >= 0 ? ih : 0;
  ih = ih <= nbin-1 ? ih : nbin-1;
#pragma omp ordered simd overlap(ih)
  ++contents[ih];
}
```

Key Take Aways

1. Outer loop vectorization can be achieved using OpenMP SIMD pragma.
2. The Histogram loop pattern can hinted to compiler using overlap clause.
3. The ordered clause takes into account the nb dependency (if omitted, wrong results)

Histogram Loop pattern

vecabi compiler option

godbolt.org/z/cPzr94GKx

```
#pragma omp declare simd
float myfun(float x) {
  float twopi=2.f*acosf(-1.f);
  float y = sinf(x*twopi);
  return y;
}
```

- Compiler creates both vector and scalar versions
- Use -vecabi=cmdtarget to target instruction set specified by –x switch
- Else ABI requires arguments to be passed using xmm registers (Intel® SSE)

Key Take Aways

1. -vecabi option help to pass arguments via ymm/zmm registers

Histogram Performance

Optimization Options	Speed-up	What's going on
-O2 -xCORE-AVX2	1.0x	Non-vectorized
-O2 -xCORE-AVX512	9.0x	Vectorized

Key Take Aways

Speedup really depends on the data set being histogramed.

Lesser the conflict within the SIMD register, more is the speedup.

Lab exercises

- 2 options:
  - Check generated ASM and opt report using Godbolt links

    Complete “let’s try” tasks
  - Try full version of examples from github on Intel DevCloud

    git clone https://github.com/fbaru-dev/hpc-workshop.git

Exercise 1 – skx_512

NB: Set *ulimit –s unlimited* before run

git clone https://github.com/fbaru-dev/hpc-workshop.git

- You should observe the difference between the two cases and look at the assembly code
- Go to the folder skylake-avx512/compress/01
- Type *make* to compile the default case with AVX2. The compiler report is generated and you can read/interpret it. It does not vectorize the compress loop.
- Type *make run* to run the test and measure the timing.
- Type *make AVX512=yes* to compile for the AVX512, observe the change in the compiler report, run the test with *make run* and measure the timing.
- Generate the assembly code with *make AVX512 asm.*

Exercise 2 – skx_512

- You should observe the difference between the two cases
- Go to the folder skylake-avx512/compress/02
- Type *make AVX512=yes* to compile the default case with AVX512. The compiler report is generated and you can read/interpret it. It vectorizes the inner loop.
- Type *make run* to run the test and measure the timing.
- Type *make AVX512=yes SIMD=yes* to compile with openmp simd enabled, observe the change in the compiler report, run the test with *make run* and measure the timing.

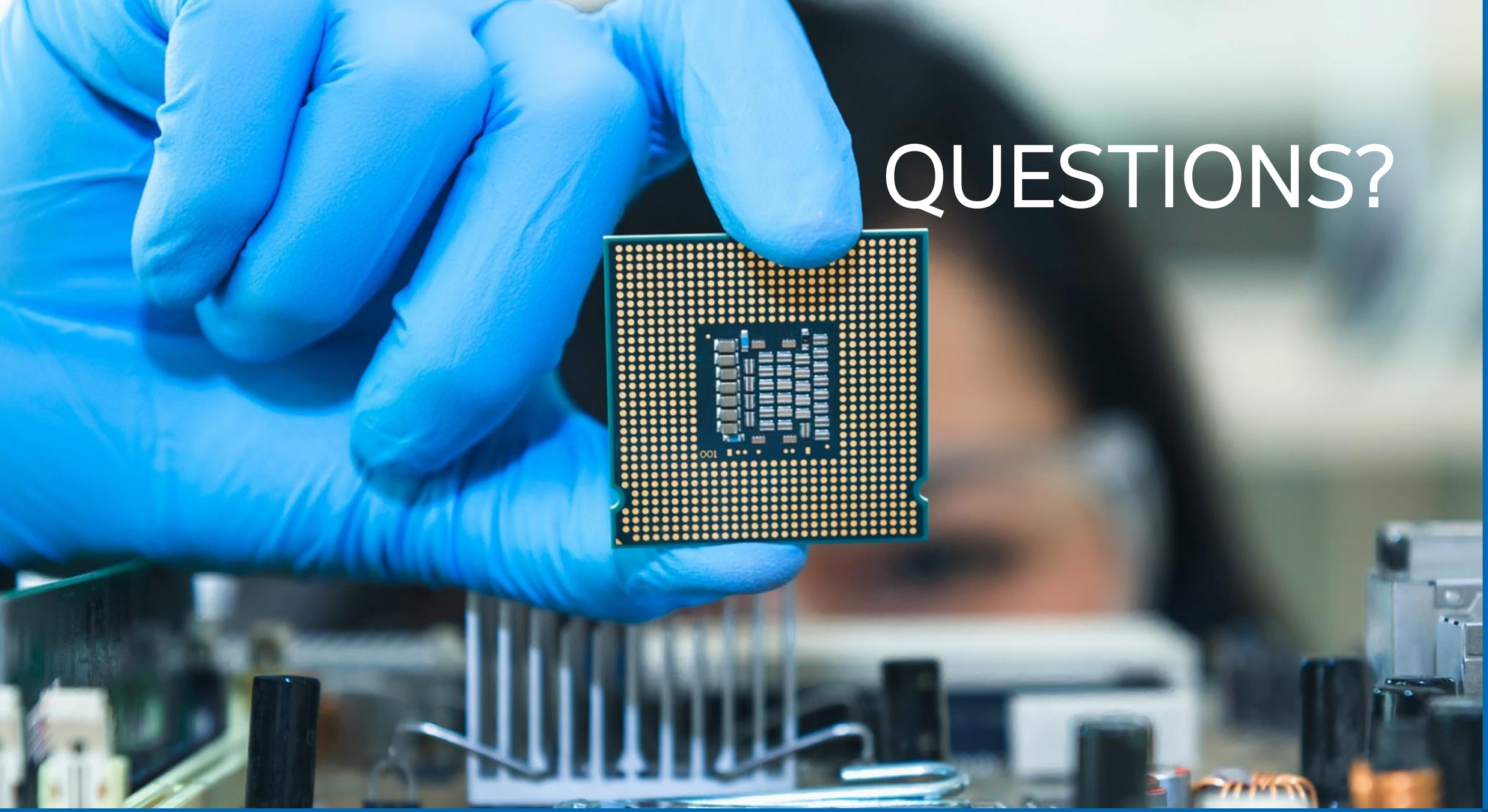
NB: Set *ulimit –s unlimited* before run

Exercise 3 – skx_512

- You should observe the difference between the two cases and look at the assembly code
- Go to the folder skylake-avx512/histo/01
- Type *make* to compile the default case with AVX2. The compiler report is generated and you can read/interpret it. It does not vectorize the histogram patetrn loop.
- Type *make run* to run the test and measure the timing.
- Type *make AVX512=yes* to compile for the AVX512, observe the change in the compiler report, run the test with *make run* and measure the timing.
- Generate the assembly code with *make AVX512 asm.*

Exercise 4 – skx_512

- You should observe the difference between the two cases and look at the simd vectorized code
- Go to the folder skylake-avx512/histo/02
- Type *make* to compile the default case with AVX512. The compiler report is generated and you can read/interpret it. It does not vectorize the histogram pattern loop.
- Type *make run* to run the test and measure the timing.
- Type *make SIMD=yes* to compile the SIMD version, observe the change in the compiler report, run the test with *make run* and measure the timing.


QUESTIONS?

Notices & Disclaimers

- This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice.
- Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.
- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.
- INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
-

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804


intel