INTRODUCTION TO PERFORMANCE OPTIMIZATION AND TUNING

Bei Wang, Ph.D.
HPC Software Engineer
Research Computing, Princeton University

PICSciE/Research Computing Fall Training
Nov 27, 2018

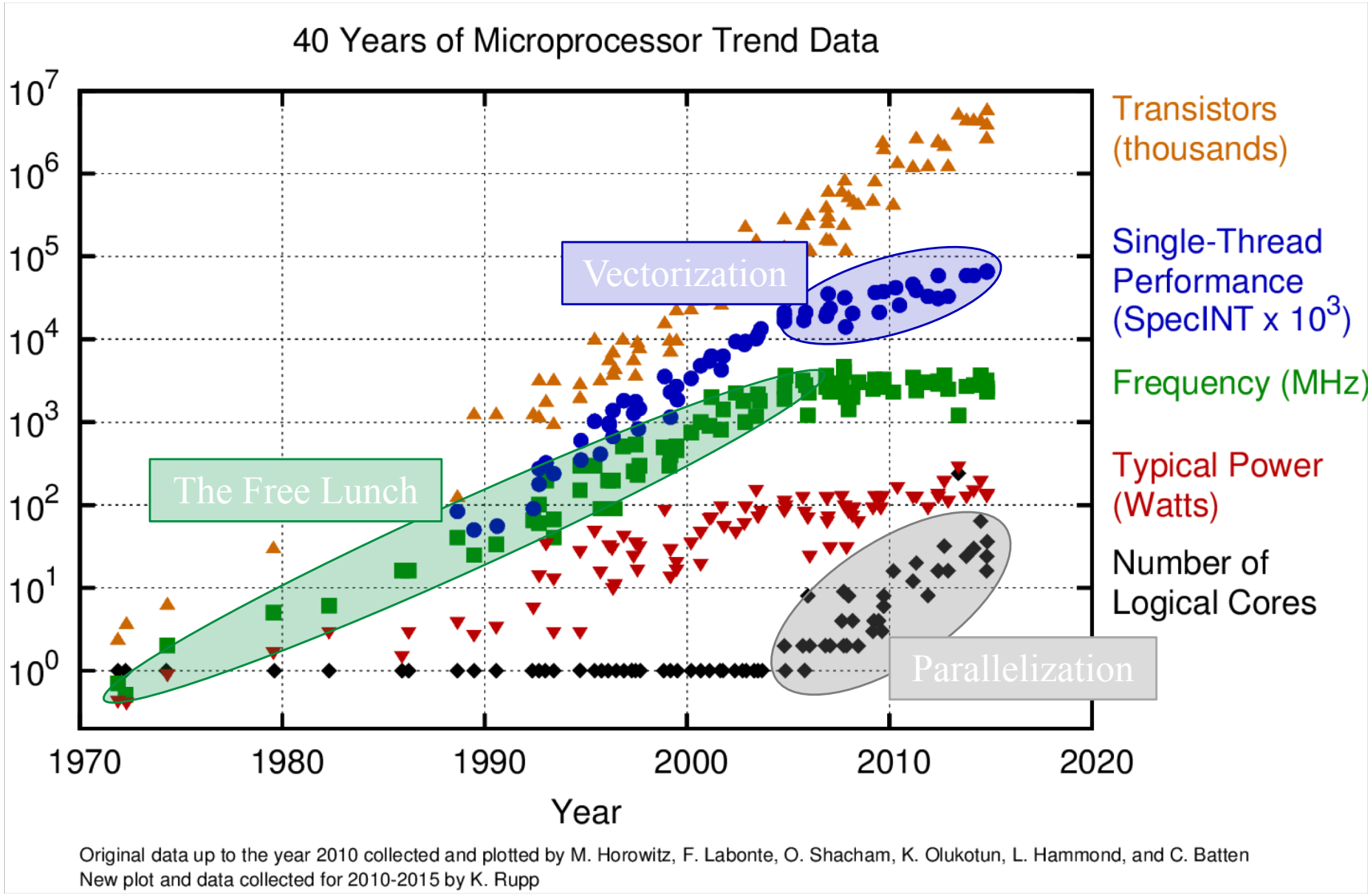
About Me

- Bei Wang, Ph.D., **HPC Software Engineer** at Research Computing
- 7 years at Princeton University Working on Research and Development in Parallel Computing Applications (plasma physics and fluid dynamics domains)
- Co-PI of **Intel Parallel Computing Center (IPCC)** at Princeton
- Office: 334 Lewis Library (TW), 111 Peyton Hall (MTF)
- Email: beiwang@princeton.edu

Outline

- A **practical guidance** to performance optimization and tuning using **vectorization**
  - Hardware evolution
  - Auto-vectorization
  - Explicit vectorization with OpenMP SIMD
  - Vectorization efficiency
  - Intel Advisor
- Focused primarily on the **HPC recourses at Princeton**
  - Hardware: Intel Broadwell and Skylake processor
  - Compiler: Intel compiler
  - Software: Written in C/C++ and Fortran language
  - Most principles apply universally

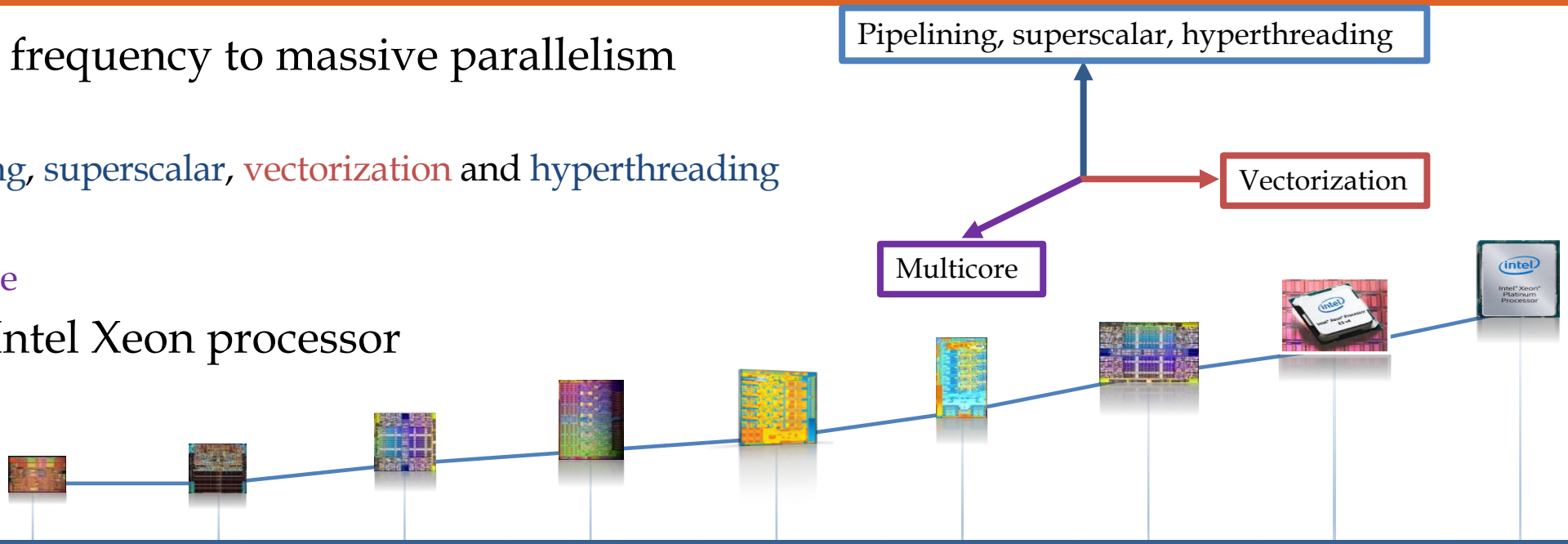
“Free Lunch is Over”



https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

Trend of Processor Evolution

- From high frequency to massive parallelism
  - Core
    - Pipelining, superscalar, vectorization and hyperthreading
  - Processor
    - Multicore
- Example: Intel Xeon processor



Intel® Xeon® Processor	64-bit	5100 series	5500 series	5600 series	E5-2600	E5-2600 V2	E5-2600 V3	E5-2600 V4	Platinum 8180
Cores	1	2	4	6	8	12	18	22	28
Threads	2	2	8	12	16	24	36	44	56
SIMD Width	128	128	128	128	256	256	256	256	512
	2005	2007	2009	2010	2012	2013	2014	2016	2017

- Application developers need to write software to take advantage of the computing power

Peak FLOPS

- Terminology

  FLOP = Floating-point Operation

  FLOPs = Floating-pint Operations

  FLOPS = Floating-point Operations Per Second

- Peak FLOPS = sockets x $\frac{cores}{socket}$ x $\frac{cycles}{second}$ x $\frac{FLOPs}{cycle}$

- $\frac{FLOPs}{cycle}$ depends on the chip architecture:
  - Haswell/Broadwell(AVX2, 256-bit)
    16 double precision (DP) FLOPs/cycle: two, 4-wide fused multiply-add (FMA) instructions
    32 single precision (SP) FLOPs/cycle: two, 8-wide FMA instructions
  - Skylake (AVX512, 512-bit)
    32 DP FLOPs/cycle: two 8-wide FMA instructions
    64 SP FLOPs/cycle: two 16-wide FMA instructions

Exercise 1: Find the Peak FLOPS of Intel Skylake Processor on Adroit

- Log in Adroit: ssh -l usersname adroit.princeton.edu
- Find the CPU information: lscpu
  - How many sockets?
  - How many cores per socket?
  - What's the processor frequency?
- The Skylake processor has 512-bits vector register and can do two FMA instructions per clock cycle
- What is the peak FLOPS?
  - DP FLOPS = 2x16x2.6x10^9x2x8x2 = 2.662 TFlop (10^{12} FLOPS)
  - SP FLOPS = 2x16x2.6x10^9x2x16x2 = 5.324 TFlop (10^{12} FLOPS)

Vectorization is essential to achieve peak FLOPS

Vectorization 101

- Simple example:

```
for (int i=0; i<N; i++)
    z[i] = x[i] + y[i];
```

Scalar code

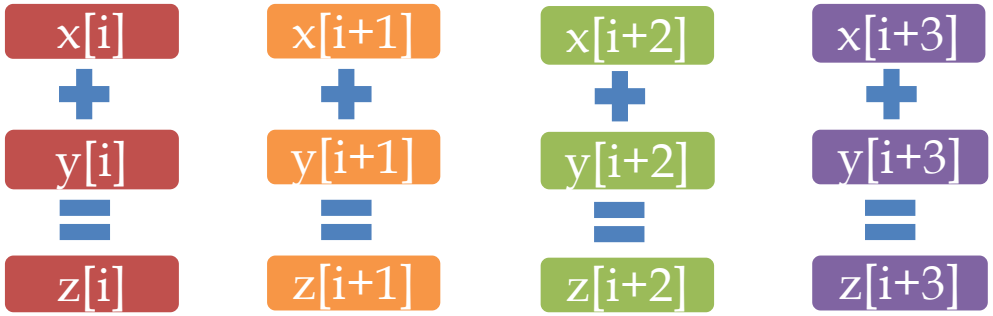
Loop N Times

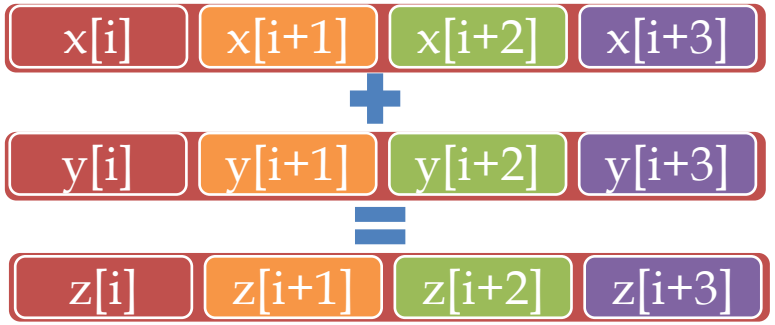
```
ld r1, addr1
ld r2, addr2
add r3, r1, r2
st addr3, r3
```

Vector code

Loop N/4 Times

```
vld vr1, addr1
vld vr2, addr2
vadd vr3, vr1, vr2
vst addr3, vr3
```





Aspects of Vectorization

1. Hardware
   - Registers and functional units
   - Instructions sets (E.g., SSE, AVX/AVX2, AVX512)
2. Software (how to vectorize you code?)
   a) Compiler-based vectorization (portable)
      - **Auto-Vectorization** (no change of code)
      - **Semi-Auto-Vectorization**: auto-vectorization enhanced by compiler directives (e.g., #pragma vector…)
      - **Explicit Vector Programming**: auto-vectorization with explicitly control using OpenMP (e.g., #pragma omp simd)
   b) Manually by explicit syntax (not portable)
      - Vector intrinsics
      - Assembly language
   c) Vendor-supplied optimized libraries

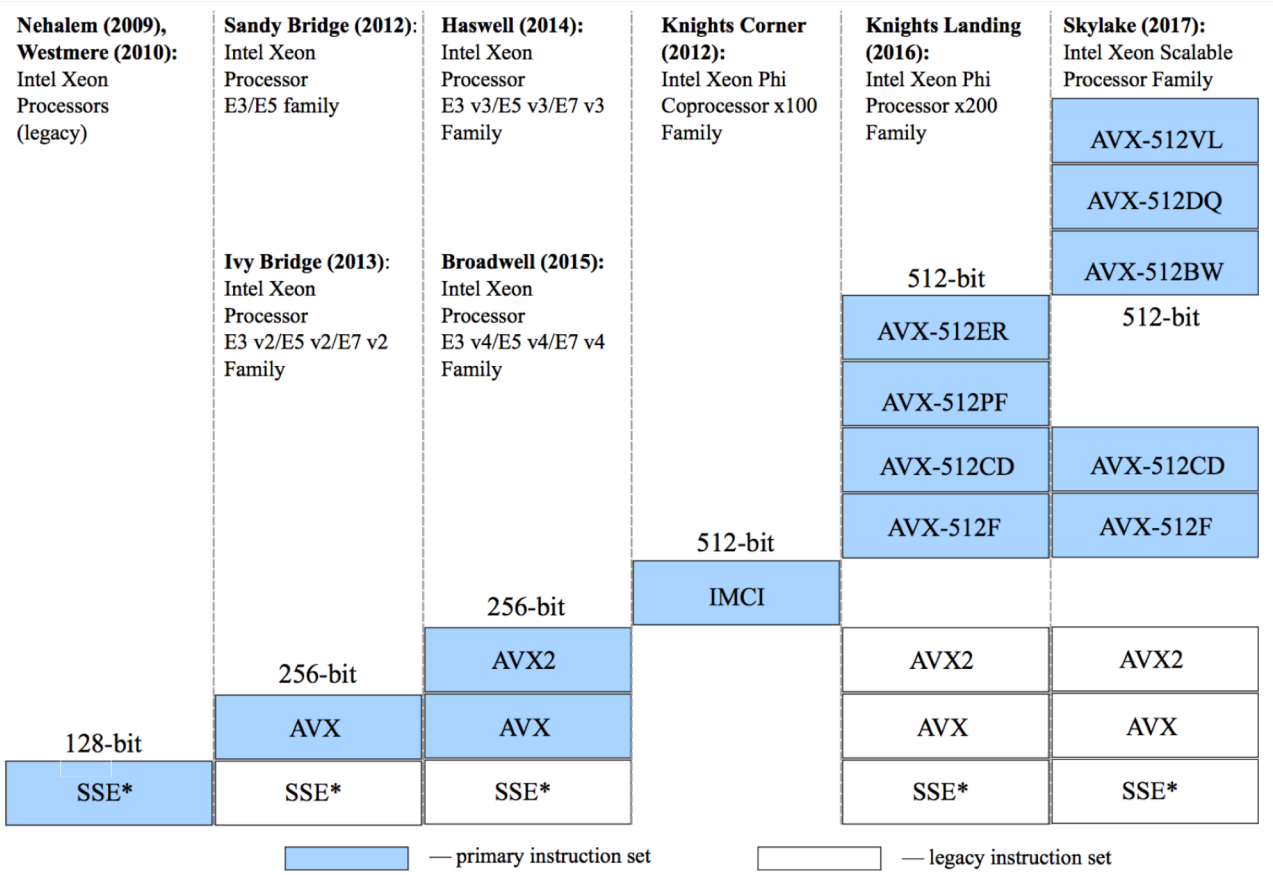


Figure 1: Instruction sets supported by different architectures.

"*Capabilities of Intel AVX-512 in Intel Xeon Scalable Processors (Skylake)*", Alaa Eltablawy and Andrey Vladimirov, Colfax International, 2017

Compiler Vectorization Switch I

- Intel compilers start vectorizing at optimization level **-O2**
  - Default is SSE instructions, 128-bit vector width
  - To tune vectors to the host machine: **-xHost**
  - To optimize across objects (e.g., to inline functions): **-ipo**
  - To disable vectorization: **-no-vec**
- GCC compilers start vectorizing at optimization level **-O3**
  - Default for x86_64 is SSE (see output from gcc -v, no other flags)
  - To tune vectors to the host machine: **-march=native**
  - To optimize across objects (e.g., to inline): **-fwhole-program**
  - To disable vectorization: **-fno-tree-vectorize** (after **-O3**)
- Why disable or downsize vectors? To gauge their benefit!

Compiler Vectorization Switch II

- Intel compilers for specific processor
  - Use **-xCORE-AVX2** to compile for AVX2, 256-bit vector width
  - Use **-xCOMMON-AVX512** to compile with AVX-512F + AVX-512CD
  - For SKL-SP: **-xCORE-AVX512 -qopt-zmm-usage=high**
  - For KNL (MIC architecture): **-xMIC-AVX512**
- GCC compilers for specific processor
  - Use **-mavx2** to compile for AVX2
  - For SKL-SP/KNL common subset: **-mavx512f -mavx512cd**
  - GCC 4.9+ has separate options for each AVX-512 extension
  - GCC 5.3+ has **-march=skylake-avx512**
  - GCC 6.1+ has **-march=knl**

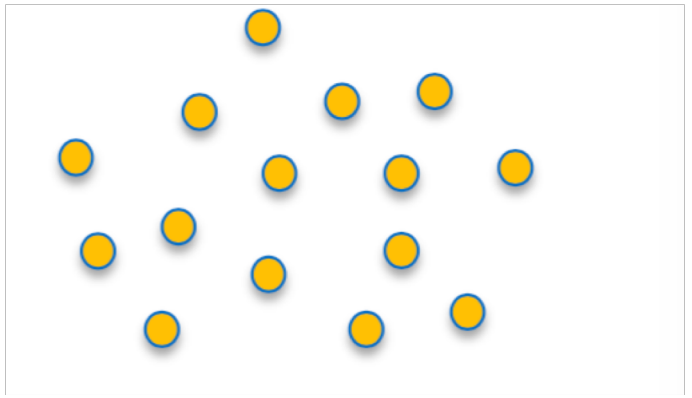
Vectorization Report

- Intel compiler options:
  - -qopt-report[=n]: tells the compiler to generate an optimization report in an *.optrpt file (https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-qopt-report-qopt-report#62679876-5691-40B3-BC90-E2D32643960D)
    - The –n controls the amount of detail of the *.optrpt report
  - -qopt-report-phase[=list]: specify one and more optimizer phases for which optimization reports are generated (https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-qopt-report-phase-qopt-report-phase#2058006C-1E74-4ECB-AA1A-E4F0E018E839)
    - vec: the phase for vectorization
    - all: all optimizer phases (default)
- GCC compiler option:
  - -ftree-vectorizer-verbose[=n] (deprecated in version 6.3.0)
  - -fopt-info-vec
  - -fopt-info-vec-missed

Note: for the rest of the lecture, we will focus on Intel compilers

Nbody Problem

```
// Loop over particles that experience force
for (int i = 0; i < nParticles; i++) {

  // Components of the gravity force on particle i
  float Fx = 0, Fy = 0, Fz = 0;

  // Loop over particles that exert force: vectorization expected here
  for (int j = 0; j < nParticles; j++) {

    // Avoid singularity and interaction with self
    const float softening = 1e-20;

    // Newton's law of universal gravity
    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;
    const float drSquared  = dx*dx + dy*dy + dz*dz + softening;
    const float drPower32  = pow(drSquared, 3.0/2.0);

    // Calculate the net force
    Fx += dx / drPower32;
    Fy += dy / drPower32;
    Fz += dz / drPower32;

  }

  // Accelerate particles in response to the gravitational force
  particle[i].vx += dt*Fx;
  particle[i].vy += dt*Fy;
  particle[i].vz += dt*Fz;
}
```



$$\vec{F}_{ij} = \frac{G m_i m_j}{\left|\vec{r}_j - \vec{r}_i\right|^3}\left(\vec{r}_j - \vec{r}_i\right)$$

$$\vec{F} = m\vec{a} = m\frac{d\vec{v}}{dt} = m\frac{d^2\vec{x}}{dt^2}$$

The example code assumes G=1 and m=1 for all particles

Exercise 2: Check Performance with Different Compiler Vectorization Switches

- Log into Adroit:
  - ssh –l usersname adroit.princeton.edu
- Clone the repo:
  - git clone https://github.com/beiwang2003/Intro_to_PerfTuning_Winter2018.git
- Load the intel compiler:
  - module load intel
- Compile and run the code with different compiling flags
  1. Edit "ICXXFLAGS" in Makefile -O0, e.g., ICXXFLAGS=-O0
  2. Do "make app-ICC"
  3. Check the vectorization report (nbody.oicc.optrpt file)
  4. Run the code with "./app-ICC"
  5. Record the performance number
  6. Repeat the above steps for
     - ICXXFLAGS=-O2
     - ICXXFLAGS=-O2 –xCORE-AVX2
     - ICXXFLAGS=-O2 –xCORE-AVX512 –qopt-zmm-usage=high

  Note: do "make app-ICC clean" before each make

Follow-up Questions

- Which case results in the best result?

Compiler flags	Performance GFLOPS
-O0	0.7
-O2	9.2
-O2 –xCORE-AVX2	15.0
-O2 –xCORE-AVX512 –qopt-zmm-usage=high	17.1

  - "ICXXFLAGS=-O2 –xCORE-AVX512 –qopt-zmm-usage=high" uses the AVX512 instructions on 512-bit register
- Do you see the expected speed up? Why?
  - No machine specific flag: SSE instructions on 128-bit register by default
  - -xCORE-AVX2: AVX2 instructions on 256-bit register
  - -xCORE-AVX512 and -qopt-zmm-usage=high: AVX512 instruction on 512-bit register

Auto-Vectorization by Compiler

To be vectorizable by the compiler (auto-vectorization), loops must meet the following criteria:

1. Countable: the number of iterations should be known at entry of the loop at runtime. Exit of the loop must not be data dependent.
   - No conditional termination (e.g., break, while loops, …)
2. Straight-line code: no jump or branch such as "switch" statements, but masked assignments are allowed
   - compiler can convert "if-then-else" statements to masked assignments
3. Must be the innermost loop if nested, unless the compiler is able to
   - Fully unroll the inner loop
   - Interchange the inner and outer loops
4. No function or subroutine calls, unless these are
   - Math library functions such as sin, cos, pow, sqrt, etc
   - Inlined
5. No loop-carried dependency

https://software.intel.com/en-us/articles/requirements-for-vectorizable-loops

Help the Compiler to Vectorize (Intel)

COMPILER HINT	DESCRIPTION
#pragma ivdep	ignore potential (unproven) data dependences
#pragma vector always	override efficiency heuristics
#pragma vector [non]temporal	hint to use or not use streaming stores
#pragma vector [un]aligned	assert [un]aligned property
#pragma novector	disable vectorization for the loop
#pragma distribute point	hint to split loop at this point
#pragma loop count <int>	hint for likely trip count
restrict	Keyword to assert exclusive access through pointer; requires command line option "-restrict"
__assume_aligned(<var>, <int>)	Assert alignment property

https://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers

Note: compiler directives are only hint to the compiler

Example: Pointer Aliasing

```
                    1. restrict      restrict
void scale(double *a, double *b, int size)  {
                                        2. #pragma ivdep
                                        3. #pragma omp simd
                                        for (int i=0; i<size; i++)
                                             b[i] = 2.5*a[i]
}
```

- The compiler may not vectorize a loop if there is a potential dependency.
- For example, how about "if b[i] is pointed to a[i-1]"?
- If you know that this is not the case, you can help the compiler by
  1. -restrict –std=c90 compiler flags and 'restrict' keyword
  2. #pragma ivdep
  3. #pragma omp simd (more later)

Example: Indirect Addressing

```
void fun(double * restrict a, double * restrict b, double * restrict x, int * restrict index, int size){
    1. #pragma vector always
    2. #pragma omp simd
     for (int i=0; i<size; i++)
        b[i] += a[i] * x[index[i]];
}
```

- The compiler rarely vectorizes loops with non-unit stride or with indirect addressing unless the amount of computational work is large compared to the overhead from non-contiguous memory access
- You can help the compiler to vectorize the loop by
  1. #pragma vector always
  2. #pragma omp simd (more later)

Guided Auto-Vectorization by OpenMP 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, alignment, etc)
- OpenMP SIMD directives are commands to the compiler, not hints
  - E.g., #pragma omp simd
  - Compiler does no dependency and cost-benefit analysis
  - Programmer is responsible for correctness
- Available for OpenMP 4.0 and beyond (portable)
  - -qopenmp or –qopenmp-simd to enable

OpenMP SIMD Directives (5.0)

- Apply **#pragma omp simd** to a loop to indicate the loop can be transformed to a SIMD loop
  - **Available clauses:**
    - SAFELEN (max iteration which can be executed concurrently)
    - SIMDLEN (vector length)
    - LINEAR
    - ALGINED (tell compiler about the alignment information)
    - NONTEMPORAL (use streaming store)
    - PRIVATE
    - LASTPRIVATE
    - REDUCTION
    - COLLAPSE

    } Similar to OpenMP for threading

- Apply **#pragma omp declare simd** to a function to enable the creation of vectorized versions of the function
- Apply **#pragma omp ordered simd** to a statement to enable serial operation of the statement

https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

Example: Histogram

```
#pragma omp simd
for (int i=0; i<N; i++) {
      double y = fun(x[i]);
      int ih = floor( (y-y0)*invbinw );
      #pragma omp ordered simd
      bin[ih] = bin[ih]+1;
}
```

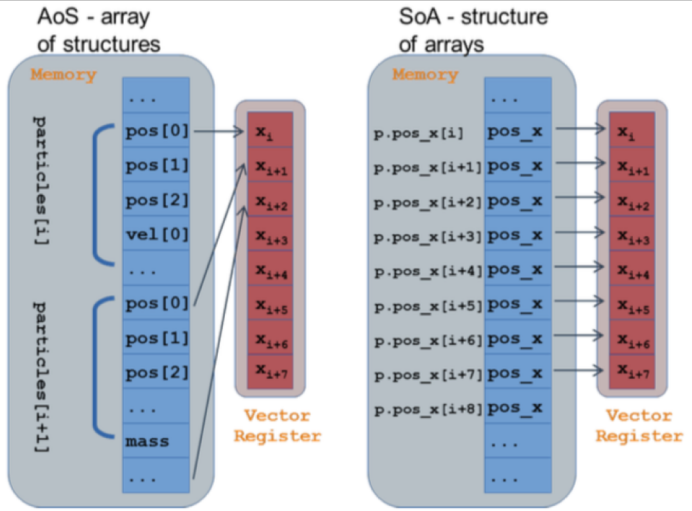
```
#pragma omp declare simd
double fun(double x) {
      double twopi = 2.0*acos(1.0);
      double y = sin(x*twopi);

      return y;
}
```

- Compiler will refuse to vectorize the above loop due to
  - User defined function call
  - Data dependency
- Can be vectorized with OpenMP SIMD by:
  - Making fun() a SIMD function
  - Using OMP ORDERED SIMD pragma to address data dependency

Data Access and Alignment

- Unit-stride access
  - Using Structure of Array (SoA) vs. Array of Structure (AoS) data layout

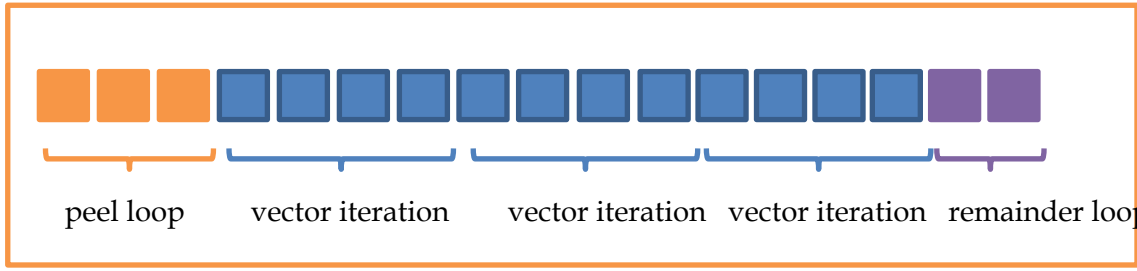
```
struct Particle
{
  public:
    ...
    real_type pos[3];
    real_type vel[3];
    real_type acc[3];
    real_type mass;
};

struct ParticleSoA
{
  public:
    ...
    real_type *pos_x,*pos_y,*pos_z;
    real_type *vel_x,*vel_y,*vel_z;
    real_type *acc_x,*acc_y;*acc_z
    real_type *mass;
};
```



The compiler might generate gather instruction for non-unit-stride access

- Looping through the array so its "fast" dimension is innermost

- Data alignment with certain byte boundary
  - Align the base pointer
  - Make sure the starting indices are aligned for each vectorized loop (for each thread)
  - Use pragmas/directives and clauses to tell the compiler that memory accesses are aligned



Peel and remainder loop are scalar or masked vector iterations

Exercise 3: Data Access and Alignment

- Compile and run the code nbody.cc example with SoA data layout for particles
  1. Edit "ICXXFLAGS" in Makefile to turn on the use of SoA data layout
     E.g., ICCXXFLAGS =-O2 –xCORE-AVX512 –qopt-zmm-usage=high **–DSoA**
  2. Do "make app-ICC" (Do "make app-ICC clean" first)
  3. Check the vectorization report
  4. Run with "./app-ICC"
  5. Record the performance number
- Repeat the above steps with data alignment,
  - e.g., add **–DAligned** compiling flag
  - Compare the vectorization report with the earlier case (no peel loop anymore!)
- Can you find any remining issue in the code?
  - Hint: check the vectorization report

```
LOOP BEGIN at nbody.cc(68,5)
   remark #15388: vectorization support: reference xp[j] has aligned access   [ nbody.cc(75,24) ]
   remark #15388: vectorization support: reference yp[j] has aligned access   [ nbody.cc(76,24) ]
   remark #15388: vectorization support: reference zp[j] has aligned access   [ nbody.cc(77,24) ]
   remark #15305: vectorization support: vector length 16
   remark #15309: vectorization support: normalized vectorization overhead 0.507
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 3
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 167
   remark #15477: vector cost: 13.560
   remark #15478: estimated potential speedup: 11.250
   remark #15486: divides: 3
   remark #15487: type converts: 3
   remark #15488: --- end vector cost summary ---
LOOP END
```

Exercise 4: Remove Type Conversion

- Compile and run the code nbody.cc example W/O type conversion

1. Edit "ICXXFLAGS" in Makefile with **–DNo_FP_Conv** option
   ICCXXFLAGS =-O2 –xCORE-AVX512 –qopt-zmm-usage=high –DSoA –DAligned **–DNo_FP_Conv**
2. Do "make app-ICC" (Do "make app-ICC clean" first)
3. Check the vectorization report
4. Run with "./app-ICC"
5. Record the performance number

Strip Mining

Original loop:

```
for (int i=0; i<N; i++) {

// do work

}
```

Loop with strip mining:

```
for (int i=0; i<N; i+=STRIP_SIZE) {
    for (int j=0; j<STRIP_SIZE; j++) {
        // do work
    }
}
```

- Transform a single loop into two nested loops
- The outer loop strides through "strips" of the iteration space
- The inner loop operates in iterations inside the strip ("mining" it)
- The strip size usually be chosen as a multiply of the vector length
- If the iteration count is not a multiple of the strip size, we need to implement a remainder loop

Case Study with Athena++
Particle subroutines

The particle subroutines of Athena++ are developed by Lev Arzamasskiy (leva@astro.princeton.edu) and Matthew Kunz at Astrophysical science department of Princeton University.

move.cpp

Interpolate from grid to particle (gather) and then update the particle properties

```
Real *x1, *x2, *x3, *v1, *v2, *v3;
__mm_malloc(&x1,…);
…
__mm_malloc(&v3,…);

….

for (long p=0; p<nparticle; p++)
  {
    a = (x1[p] - x1s) * dx1 + isg;
    ig = (int)(a);
    is = ig - 1;
    d = a - ig;
    wei1[0] = 0.5 * (1.0 - d) * (1.0 - d);
    wei1[1] = 0.75 - (d - 0.5) * (d - 0.5);
    wei1[2] = 0.5 * d * d;

    a = (x2[p] - x2s) * dx2 + jsg;
    ig = (int)(a);
    js = ig - 1;
    d = a - ig;
    wei2[0] = 0.5 * (1.0 - d) * (1.0 - d);
    wei2[1] = 0.75 - (d - 0.5) * (d - 0.5);
    wei2[2] = 0.5 * d * d;


    a = (x3[p] - x3s) * dx3 + ksg;
    ig = (int)(a);
    ks = ig - 1;
    d = a - ig;
    wei3[0] = 0.5 * (1.0 - d) * (1.0 - d);
    wei3[1] = 0.75 - (d - 0.5) * (d - 0.5);
    wei3[2] = 0.5 * d * d;


    bfld1 = 0.0; bfld2 = 0.0; bfld3 = 0.0;
    efld1 = 0.0; efld2 = 0.0; efld3 = 0.0;
    totwei=0; eb=0.0;

    // interpolate from grid to particle (gather)
    for (k0=0; k0<=2; k0++){
      for (j0=0; j0<=2; j0++){
        for (i0=0; i0<=2; i0++){
          w = wei3[k0] * wei2[j0] * wei1[i0];
          totwei += w;
          bfld1 += w * fcoup(0,ks+k0,js+j0,is+i0);
          bfld2 += w * fcoup(1,ks+k0,js+j0,is+i0);
          bfld3 += w * fcoup(2,ks+k0,js+j0,is+i0);
          efld1 += w * fcoup(3,ks+k0,js+j0,is+i0);
          efld2 += w * fcoup(4,ks+k0,js+j0,is+i0);
          efld3 += w * fcoup(5,ks+k0,js+j0,is+i0);
          eb    += w * fcoup(6,ks+k0,js+j0,is+i0);
        }
      }
    }

    bsq   = std::max(bfld1 * bfld1 + bfld2 * bfld2 +     bfld3 *
bfld3, TINY_NUMBER);
    edotb = efld1 * bfld1 + efld2 * bfld2 + efld3 * bfld3;
    diff = (eb*totwei - edotb)/bsq;
    efld1 += diff*bfld1;
    efld2 += diff*bfld2;
    efld3 += diff*bfld3;
    …

    // update particle position and velocity
    v1[p] = v1n + efld1 + vp2 * s3 - vp3 * s2;
    v2[p] = v2n + efld2 + vp3 * s1 - vp1 * s3;
    v3[p] = v3n + efld3 + vp1 * s2 - vp2 * s1;

    x1[p] = x1n + v1[p] * 0.5 * dt;
    x2[p] = x2n + v2[p] * 0.5 * dt;
    x3[p] = x3n + v3[p] * 0.5 * dt;
}
```

deposit.cpp

Interpolate from particle to grid (scatter)

```
for (long p=0; p<nparticle; p++)
  {
    a = (x1[p] - x1s) * dx1 + isg;
    ig = (int)(a);
    is = ig - 1;
    d = a - ig;
    wei1[0] = 0.5 * (1.0 - d) * (1.0 - d);
    wei1[1] = 0.75 - (d - 0.5) * (d - 0.5);
    wei1[2] = 0.5 * d * d;

    a = (x2[p] - x2s) * dx2 + jsg;
    ig = (int)(a);
    js = ig - 1;
    d = a - ig;
    wei2[0] = 0.5 * (1.0 - d) * (1.0 - d);
    wei2[1] = 0.75 - (d - 0.5) * (d - 0.5);
    wei2[2] = 0.5 * d * d;

    a = (x3[p] - x3s) * dx3 + ksg;
    ig = (int)(a);
    ks = ig - 1;
    d = a - ig;
    wei3[0] = 0.5 * (1.0 - d) * (1.0 - d);
    wei3[1] = 0.75 - (d - 0.5) * (d - 0.5);
    wei3[2] = 0.5 * d * d;

    // interpolate from particle to grid (scatter)

    for (k0=0; k0<=2; k0++){
      for (j0=0; j0<=2; j0++){
        for (i0=0; i0<=2; i0++){
          w = wei3[k0] * wei2[j0] * wei1[i0];
          mcoup(0,ks+k0,js+j0,is+i0) += w;
          mcoup(1,ks+k0,js+j0,is+i0) += w * v1[p];
          mcoup(2,ks+k0,js+j0,is+i0) += w * v2[p];
          mcoup(3,ks+k0,js+j0,is+i0) += w * v3[p];
        }
      }
    }
  }
```

Are *Move* and *Deposit* Being Vectorized?

(move.optrpt)

```
LOOP BEGIN at src/particle/move.cpp(64,3)
   remark #25096: Loop Interchange not done due to: Imperfect Loop Nest (Either at Source or due to other Compiler Transformations)
   remark #25451: Advice: Loop Interchange, if possible, might help loopnest. Suggested Permutation : ( 1 2 3 4 ) --> ( 2 3 1 4 )
   remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

   LOOP BEGIN at src/particle/move.cpp(99,5)
      remark #25101: Loop Interchange not done due to: Original Order seems proper
      remark #25452: Original Order found to be proper, but by a close margin
      remark #15541: outer loop was not auto-vectorized: consider using SIMD directive
      remark #25436: completely unrolled by 3

      LOOP BEGIN at src/particle/move.cpp(100,7)
         remark #15541: outer loop was not auto-vectorized: consider using SIMD directive
         remark #25436: completely unrolled by 3

         LOOP BEGIN at src/particle/move.cpp(101,9)
            remark #15388: vectorization support: reference wei1[i0] has aligned access   [ src/particle/move.cpp(102,37) ]
            remark #15389: vectorization support: reference fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0))] has unaligned access   [ src/particle/move.cpp(104,29) ]
            remark #15389: vectorization support: reference fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcoup.nx3_))] has unaligned access   [ src/particle/move.cpp(105,29) ]
            remark #15389: vectorization support: reference fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcoup.nx3_*2))] has unaligned access   [ src/particle/move.cpp(106,29) ]
            remark #15389: vectorization support: reference fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcoup.nx3_*3))] has unaligned access   [ src/particle/move.cpp(107,29) ]
            remark #15389: vectorization support: reference fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcoup.nx3_*4))] has unaligned access   [ src/particle/move.cpp(108,29) ]
            remark #15389: vectorization support: reference fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcoup.nx3_*5))] has unaligned access   [ src/particle/move.cpp(109,29) ]
            remark #15389: vectorization support: reference fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcoup.nx3_*6))] has unaligned access   [ src/particle/move.cpp(110,29) ]
            remark #15381: vectorization support: unaligned access used inside loop body
            remark #15335: loop was not vectorized: vectorization possible but seems inefficient. Use vector always directive or -vec-threshold0 to override
            remark #15305: vectorization support: vector length 2
            remark #15309: vectorization support: normalized vectorization overhead 0.373
            remark #15448: unmasked aligned unit stride loads: 1
            remark #15450: unmasked unaligned unit stride loads: 7
            remark #15475: --- begin vector cost summary ---
            remark #15476: scalar cost: 74
            remark #15477: vector cost: 41.500
            remark #15478: estimated potential speedup: 1.180
            remark #15488: --- end vector cost summary ---
            remark #25436: completely unrolled by 3
         LOOP END

         LOOP BEGIN at src/particle/move.cpp(101,9)
         LOOP END

         LOOP BEGIN at src/particle/move.cpp(101,9)
         LOOP END
      LOOP END

      LOOP BEGIN at src/particle/move.cpp(100,7)

         LOOP BEGIN at src/particle/move.cpp(101,9)
         LOOP END

         LOOP BEGIN at src/particle/move.cpp(101,9)
         LOOP END

         LOOP BEGIN at src/particle/move.cpp(101,9)
         LOOP END
      LOOP END

      LOOP BEGIN at src/particle/move.cpp(100,7)

         LOOP BEGIN at src/particle/move.cpp(101,9)
         LOOP END

         LOOP BEGIN at src/particle/move.cpp(101,9)
         LOOP END

         LOOP BEGIN at src/particle/move.cpp(101,9)
         LOOP END
      LOOP END
   LOOP END
LOOP END
```

The compiler failed to vectorize the inner most loop

deposit.optrpt

```
LOOP BEGIN at src/particle/deposit.cpp(57,3)
   remark #25096: Loop Interchange not done due to: Imperfect Loop Nest (Either at Source or due to other Compiler Transformations)
   remark #25452: Original Order found to be proper, but by a close margin
   remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

   LOOP BEGIN at src/particle/deposit.cpp(83,5)
      remark #25101: Loop Interchange not done due to: Original Order seems proper
      remark #25452: Original Order found to be proper, but by a close margin
      remark #15541: outer loop was not auto-vectorized: consider using SIMD directive
      remark #25436: completely unrolled by 3

      LOOP BEGIN at src/particle/deposit.cpp(84,7)
         remark #15541: outer loop was not auto-vectorized: consider using SIMD directive
         remark #25436: completely unrolled by 3

         LOOP BEGIN at src/particle/deposit.cpp(85,9)
            remark #15344: loop was not vectorized: vector dependence prevents vectorization
            remark #15346: vector dependence: assumed OUTPUT dependence between mcoup.pdata_[?-1+i0+mcoup.nx1_*(?-1+j0+mcoup.nx2_*(ig-1+k0))] (87:16) and mcoup.pdata_[?-1+i0+mcoup.nx1_*(?-1+j0+mcoup.nx2
            remark #15346: vector dependence: assumed OUTPUT dependence between mcoup.pdata_[?-1+i0+mcoup.nx1_*(?-1+j0+mcoup.nx2_*(ig-1+k0+mcoup (90:16) and mcoup.pdata_[?-1+i0+mcoup.nx1_*(?-1+j0+mcoup.
            remark #25436: completely unrolled by 3
         LOOP END

         LOOP BEGIN at src/particle/deposit.cpp(85,9)
         LOOP END

         LOOP BEGIN at src/particle/deposit.cpp(85,9)
         LOOP END
      LOOP END

      LOOP BEGIN at src/particle/deposit.cpp(84,7)

         LOOP BEGIN at src/particle/deposit.cpp(85,9)
         LOOP END

         LOOP BEGIN at src/particle/deposit.cpp(85,9)
         LOOP END

         LOOP BEGIN at src/particle/deposit.cpp(85,9)
         LOOP END
      LOOP END

      LOOP BEGIN at src/particle/deposit.cpp(84,7)

         LOOP BEGIN at src/particle/deposit.cpp(85,9)
         LOOP END

         LOOP BEGIN at src/particle/deposit.cpp(85,9)
         LOOP END

         LOOP BEGIN at src/particle/deposit.cpp(85,9)
         LOOP END
      LOOP END
   LOOP END
LOOP END
```

The compiler failed to vectorize the inner most loop because of vector dependence in scatter operation

First Attempt:

- Using **#pragma omp simd** in the outer most loop

move.cpp

Use simd directives

```
#pragma omp simd private(…)
for (long p=0; p<nparticle; p++)
  {
    a = (x1[p] - x1s) * dx1 + isg;
    ig = (int)(a);
    is = ig - 1;
    d = a - ig;
    wei1[0] = 0.5 * (1.0 - d) * (1.0 - d);
    wei1[1] = 0.75 - (d - 0.5) * (d - 0.5);
    wei1[2] = 0.5 * d * d;

    a = (x2[p] - x2s) * dx2 + jsg;
    ig = (int)(a);
    js = ig - 1;
    d = a - ig;
    wei2[0] = 0.5 * (1.0 - d) * (1.0 - d);
    wei2[1] = 0.75 - (d - 0.5) * (d - 0.5);
    wei2[2] = 0.5 * d * d;


    a = (x3[p] - x3s) * dx3 + ksg;
    ig = (int)(a);
    ks = ig - 1;
    d = a - ig;
    wei3[0] = 0.5 * (1.0 - d) * (1.0 - d);
    wei3[1] = 0.75 - (d - 0.5) * (d - 0.5);
    wei3[2] = 0.5 * d * d;


    bfld1 = 0.0; bfld2 = 0.0; bfld3 = 0.0;
    efld1 = 0.0; efld2 = 0.0; efld3 = 0.0;
    totwei=0; eb=0.0;

    // interpolate from grid to particle (gather)
    for (k0=0; k0<=2; k0++){
      for (j0=0; j0<=2; j0++){
        for (i0=0; i0<=2; i0++){
          w = wei3[k0] * wei2[j0] * wei1[i0];
          totwei += w;
          bfld1 += w * fcoup(0,ks+k0,js+j0,is+i0);
          bfld2 += w * fcoup(1,ks+k0,js+j0,is+i0);
          bfld3 += w * fcoup(2,ks+k0,js+j0,is+i0);
          efld1 += w * fcoup(3,ks+k0,js+j0,is+i0);
          efld2 += w * fcoup(4,ks+k0,js+j0,is+i0);
          efld3 += w * fcoup(5,ks+k0,js+j0,is+i0);
          eb    += w * fcoup(6,ks+k0,js+j0,is+i0);
        }
      }
    }

    bsq   = std::max(bfld1 * bfld1 + bfld2 * bfld2 +     bfld3 *
bfld3, TINY_NUMBER);
    edotb = efld1 * bfld1 + efld2 * bfld2 + efld3 * bfld3;
    diff = (eb*totwei - edotb)/bsq;
    efld1 += diff*bfld1;
    efld2 += diff*bfld2;
    efld3 += diff*bfld3;
    …

    // update particle position and velocity
    v1[p] = v1n + efld1 + vp2 * s3 - vp3 * s2;
    v2[p] = v2n + efld2 + vp3 * s1 - vp1 * s3;
    v3[p] = v3n + efld3 + vp1 * s2 - vp2 * s1;

    x1[p] = x1n + v1[p] * 0.5 * dt;
    x2[p] = x2n + v2[p] * 0.5 * dt;
    x3[p] = x3n + v3[p] * 0.5 * dt;
}
```

move.optrpt

```
LOOP BEGIN at src/particle/move.cpp(69,3)
<Peeled loop for vectorization>
   remark #15389: vectorization support: reference this->x1[p] has unaligned access   [ src/particle/move.cpp(73,13) ]
   remark #15389: vectorization support: reference this->x2[p] has unaligned access   [ src/particle/move.cpp(74,13) ]
   remark #15389: vectorization support: reference this->x3[p] has unaligned access   [ src/particle/move.cpp(75,13) ]
   remark #15389: vectorization support: reference this->v1[p] has unaligned access   [ src/particle/move.cpp(76,13) ]
   remark #15389: vectorization support: reference this->v2[p] has unaligned access   [ src/particle/move.cpp(77,13) ]
   remark #15389: vectorization support: reference this->v3[p] has unaligned access   [ src/particle/move.cpp(78,13) ]
   remark #15389: vectorization support: reference this->x1[p] has unaligned access   [ src/particle/move.cpp(155,5) ]
   remark #15389: vectorization support: reference this->x2[p] has unaligned access   [ src/particle/move.cpp(156,5) ]
   remark #15389: vectorization support: reference this->x3[p] has unaligned access   [ src/particle/move.cpp(157,5) ]
   remark #15389: vectorization support: reference this->v1[p] has unaligned access   [ src/particle/move.cpp(158,5) ]
   remark #15389: vectorization support: reference this->v2[p] has unaligned access   [ src/particle/move.cpp(159,5) ]
   remark #15389: vectorization support: reference this->v3[p] has unaligned access   [ src/particle/move.cpp(160,5) ]
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15335: peel loop was not vectorized: vectorization possible but seems inefficient. Use vector always directive or -vec-threshold0 to override
   remark #15305: vectorization support: vector length 2
   remark #15309: vectorization support: normalized vectorization overhead 0.135
   remark #25015: Estimate of max trip count of loop=1
LOOP END
```

1. Peel loop exists

2. Unaligned access

```
   remark #15328: vectorization support: irregularly indexed load was emulated for the variable <fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcoup>,
om memory   [ src/particle/move.cpp(120,29) ]
   remark #15328: vectorization support: irregularly indexed load was emulated for the variable <fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcoup>,
om memory   [ src/particle/move.cpp(121,29) ]
   remark #15328: vectorization support: irregularly indexed load was emulated for the variable <fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcoup>,
om memory   [ src/particle/move.cpp(122,29) ]
   remark #15305: vectorization support: vector length 2
   remark #15309: vectorization support: normalized vectorization overhead 0.299
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15321: Compiler has chosen to target XMM/YMM vector. Try using -qopt-zmm-usage=high to override
   remark #15450: unmasked unaligned unit stride loads: 6
   remark #15451: unmasked unaligned unit stride stores: 6
   remark #15462: unmasked indexed (or gather) loads: 189
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 2089
   remark #15477: vector cost: 3212.000
   remark #15478: estimated potential speedup: 0.630
   remark #15486: divides: 3
   remark #15487: type converts: 6
   remark #15488: --- end vector cost summary ---
```

3. No potential speed up

Second Attempt:

- Using **posix_memalign** to allocate particle arrays with alignment
- Telling the compiler that memory accesses are aligned

move.cpp

Ensure Alignment

```
Real *x1, *x2, *x3, *v1, *v2, *v3;
posix_memalign(&x1,…,64);
…
posix_memalign(&v3,…,64);

#pragma omp simd aligned(x1,x2,x3,v1,v2,v3 :64) private(…)
for (long p=0; p<nparticle; p++)
  {
    a = (x1[p] - x1s) * dx1 + isg;
    ig = (int)(a);
    is = ig - 1;
    d = a - ig;
    wei1[0] = 0.5 * (1.0 - d) * (1.0 - d);
    wei1[1] = 0.75 - (d - 0.5) * (d - 0.5);
    wei1[2] = 0.5 * d * d;

    a = (x2[p] - x2s) * dx2 + jsg;
    ig = (int)(a);
    js = ig - 1;
    d = a - ig;
    wei2[0] = 0.5 * (1.0 - d) * (1.0 - d);
    wei2[1] = 0.75 - (d - 0.5) * (d - 0.5);
    wei2[2] = 0.5 * d * d;


    a = (x3[p] - x3s) * dx3 + ksg;
    ig = (int)(a);
    ks = ig - 1;
    d = a - ig;
    wei3[0] = 0.5 * (1.0 - d) * (1.0 - d);
    wei3[1] = 0.75 - (d - 0.5) * (d - 0.5);
    wei3[2] = 0.5 * d * d;


    bfld1 = 0.0; bfld2 = 0.0; bfld3 = 0.0;
    efld1 = 0.0; efld2 = 0.0; efld3 = 0.0;
    totwei=0; eb=0.0;

    // interpolate from grid to particle (gather)
    for (k0=0; k0<=2; k0++){
      for (j0=0; j0<=2; j0++){
        for (i0=0; i0<=2; i0++){
          w = wei3[k0] * wei2[j0] * wei1[i0];
          totwei += w;
          bfld1 += w * fcoup(0,ks+k0,js+j0,is+i0);
          bfld2 += w * fcoup(1,ks+k0,js+j0,is+i0);
          bfld3 += w * fcoup(2,ks+k0,js+j0,is+i0);
          efld1 += w * fcoup(3,ks+k0,js+j0,is+i0);
          efld2 += w * fcoup(4,ks+k0,js+j0,is+i0);
          efld3 += w * fcoup(5,ks+k0,js+j0,is+i0);
          eb    += w * fcoup(6,ks+k0,js+j0,is+i0);
        }
      }
    }

    bsq   = std::max(bfld1 * bfld1 + bfld2 * bfld2 +     bfld3 *
bfld3, TINY_NUMBER);
    edotb = efld1 * bfld1 + efld2 * bfld2 + efld3 * bfld3;
    diff = (eb*totwei - edotb)/bsq;
    efld1 += diff*bfld1;
    efld2 += diff*bfld2;
    efld3 += diff*bfld3;
    …

    // update particle position and velocity
    v1[p] = v1n + efld1 + vp2 * s3 - vp3 * s2;
    v2[p] = v2n + efld2 + vp3 * s1 - vp1 * s3;
    v3[p] = v3n + efld3 + vp1 * s2 - vp2 * s1;

    x1[p] = x1n + v1[p] * 0.5 * dt;
    x2[p] = x2n + v2[p] * 0.5 * dt;
    x3[p] = x3n + v3[p] * 0.5 * dt;


}
```

move.optrpt

```
LOOP BEGIN at src/particle/move.cpp(75,3)
   remark #25096: Loop Interchange not done due to: Imperfect Loop Nest (Either at Source or due to other Compiler Transformations)
   remark #25451: Advice: Loop Interchange, if possible, might help loopnest. Suggested Permutation : ( 1 2 3 4 ) --> ( 2 3 1 4 )
   remark #25456: Number of Array Refs Scalar Replaced In Loop: 81
   remark #15388: vectorization support: reference this->x1[p] has aligned access   [ src/particle/move.cpp(79,13) ]
   remark #15388: vectorization support: reference this->x2[p] has aligned access   [ src/particle/move.cpp(80,13) ]
   remark #15388: vectorization support: reference this->x3[p] has aligned access   [ src/particle/move.cpp(81,13) ]
   remark #15388: vectorization support: reference this->v1[p] has aligned access   [ src/particle/move.cpp(82,13) ]
   remark #15388: vectorization support: reference this->v2[p] has aligned access   [ src/particle/move.cpp(83,13) ]
   remark #15388: vectorization support: reference this->v3[p] has aligned access   [ src/particle/move.cpp(84,13) ]
   remark #15388: vectorization support: reference this->x1[p] has aligned access   [ src/particle/move.cpp(161,5) ]
   remark #15388: vectorization support: reference this->x2[p] has aligned access   [ src/particle/move.cpp(162,5) ]
   remark #15388: vectorization support: reference this->x3[p] has aligned access   [ src/particle/move.cpp(163,5) ]
   remark #15388: vectorization support: reference this->v1[p] has aligned access   [ src/particle/move.cpp(164,5) ]
   remark #15388: vectorization support: reference this->v2[p] has aligned access   [ src/particle/move.cpp(165,5) ]
   remark #15388: vectorization support: reference this->v3[p] has aligned access   [ src/particle/move.cpp(166,5) ]
```

1. No peel loop anymore

```
   remark #15328: vectorization support: irregularly indexed load was emulated for the variable <fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx
om memory    [ src/particle/move.cpp(126,29) ]
   remark #15328: vectorization support: irregularly indexed load was emulated for the variable <fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx
om memory    [ src/particle/move.cpp(127,29) ]
   remark #15328: vectorization support: irregularly indexed load was emulated for the variable <fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx
om memory    [ src/particle/move.cpp(128,29) ]
   remark #15305: vectorization support: vector length 2
   remark #15309: vectorization support: normalized vectorization overhead 0.294
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15321: Compiler has chosen to target XMM/YMM vector. Try using -qopt-zmm-usage=high to override
   remark #15448: unmasked aligned unit stride loads: 6
   remark #15449: unmasked aligned unit stride stores: 6
   remark #15462: unmasked indexed (or gather) loads: 189
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 2089
   remark #15477: vector cost: 3199.500
   remark #15478: estimated potential speedup: 0.640
   remark #15486: divides: 3
   remark #15487: type converts: 6
   remark #15488: --- end vector cost summary ---
```

2. Aligned access

3. Still no potential speed up, unfortunately

Third Attempt

- Use **strip mining** to help the compiler do vectorization
- Use **loop fission** to expose more parallelism

move.cpp

Two nested loops

```
Real bfld1v[SIMD_WIDTH] __attribute__((aligned(64)));
Real bfld2v[SIMD_WIDTH] __attribute__((aligned(64)));
Real bfld3v[SIMD_WIDTH] __attribute__((aligned(64)));
Real efld1v[SIMD_WIDTH] __attribute__((aligned(64)));
Real efld2v[SIMD_WIDTH] __attribute__((aligned(64)));
Real efld3v[SIMD_WIDTH] __attribute__((aligned(64)));
Real ebv[SIMD_WIDTH]    __attribute__((aligned(64)));
Real totweiv[SIMD_WIDTH]    __attribute__((aligned(64)));
```

```
for (int p=0; p<nparticle; p+=SIMD_WIDTH){

    Real *__restrict__ x1p = x1 + p;
    Real *__restrict__ x2p = x2 + p;
    Real *__restrict__ x3p = x3 + p;
    Real *__restrict__ v1p = v1 + p;
    Real *__restrict__ v2p = v2 + p;
    Real *__restrict__ v3p = v3 + p;

    #pragma omp simd aligned(x1p,x2p,x3p,v1p,v2p,v3p :64) simdlen(SIMD_WIDTH) private(…)
    for (int pp=0; pp<SIMD; pp++) {

        x1tmp = x1p[pp];
        x2tmp = x2p[pp];
        x3tmp = x3p[pp];
        v1tmp = v1p[pp];
        v2tmp = v2p[pp];
        v3tmp = v3p[pp];

        a = (x1tmp - x1s) * dx1 + isg;
        ig = (int)(a);
        is = ig - 1;
        d = a - ig;
        wei1[0] = 0.5 * (1.0 - d) * (1.0 - d);
        wei1[1] = 0.75 - (d - 0.5) * (d - 0.5);
        wei1[2] = 0.5 * d * d;

        a = (x2tmp - x2s) * dx2 + jsg;
        ig = (int)(a);
        js = ig - 1;
        d = a - ig;
        wei2[0] = 0.5 * (1.0 - d) * (1.0 - d);
        wei2[1] = 0.75 - (d - 0.5) * (d - 0.5);
        wei2[2] = 0.5 * d * d;

        a = (x3tmp - x3s) * dx3 + ksg;
        ig = (int)(a);
        ks = ig - 1;
        d = a - ig;
        wei3[0] = 0.5 * (1.0 - d) * (1.0 - d);
        wei3[1] = 0.75 - (d - 0.5) * (d - 0.5);
        wei3[2] = 0.5 * d * d;
```

```
bfld1v[pp] = 0.0; bfld2v[pp] = 0.0; bfld3v[pp] = 0.0;
efld1v[pp] = 0.0; efld2v[pp] = 0.0; efld3v[pp] = 0.0;
ebv[pp] = 0.0; totweiv[pp]=0.0;

for (k0=0; k0<=2; k0++){
  for (j0=0; j0<=2; j0++){
    for (i0=0; i0<=2; i0++){
       w = wei3[k0] * wei2[j0] * wei1[i0];
       totweiv[pp] += w;
       bfld1v[pp] += w * fcoup(0,ks+k0,js+j0,is+i0);
       bfld2v[pp] += w * fcoup(1,ks+k0,js+j0,is+i0);
       bfld3v[pp] += w * fcoup(2,ks+k0,js+j0,is+i0);
       efld1v[pp] += w * fcoup(3,ks+k0,js+j0,is+i0);
       efld2v[pp] += w * fcoup(4,ks+k0,js+j0,is+i0);
       efld3v[pp] += w * fcoup(5,ks+k0,js+j0,is+i0);
       ebv[pp]    += w * fcoup(6,ks+k0,js+j0,is+i0);
    }
  }
}
```

```
   #pragma omp simd aligned(x1p,x2p,x3p,v1p,v2p,v3p :64)   simdlen(SIMD_WIDTH)
private(…)
   for (int pp=0; pp<SIMD; pp++) {
      x1tmp = x1p[pp];
      x2tmp = x2p[pp];
      x3tmp = x3p[pp];
      v1tmp = v1p[pp];
      v2tmp = v2p[pp];
      v3tmp = v3p[pp];

      bsq   = std::max(bfld1v[pp] * bfld1v[pp] + bfld2v[pp] * bfld2v[pp] + bfld3v[pp]*
bfld3v[pp], TINY_NUMBER);
      edotb = efld1v[pp] * bfld1v[pp] + efld2v[pp] * bfld2v[pp] + efld3v[pp] *
bfld3v[pp];
      diff = (ebv[pp]*totweiv[pp] - edotb)/bsq;
      efld1v[pp] += diff*bfld1v[pp];
      efld2v[pp] += diff*bfld2v[pp];
      efld3v[pp] += diff*bfld3v[pp];

      …

      x1p[pp] = x1n + v1tmp * halfdt;
      x2p[pp] = x2n + v2tmp * halfdt;
      x3p[pp] = x3n + v3tmp * halfdt;
      v1p[pp] = v1tmp;
      v2p[pp] = v2tmp;
      v3p[pp] = v3tmp;
   }
 }
```

Loop fission

move.optrpt (1/2)

```
LOOP BEGIN at src/particle/move.cpp(79,3)
   remark #25084: Preprocess Loopnests: Moving Out Store    [ src/particle/move.cpp(86,30) ]
   remark #25084: Preprocess Loopnests: Moving Out Store    [ src/particle/move.cpp(87,30) ]
   remark #25084: Preprocess Loopnests: Moving Out Store    [ src/particle/move.cpp(88,30) ]
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at src/particle/move.cpp(93,59)
      remark #25096: Loop Interchange not done due to: Imperfect Loop Nest (Either at Source or due to other Compiler Transformations)
      remark #25451: Advice: Loop Interchange, if possible, might help loopnest. Suggested Permutation : ( 1 2 3 4 ) --> ( 2 3 1 4 )
      remark #15388: vectorization support: reference x1p[pp] has aligned access   [ src/particle/move.cpp(95,13) ]
      remark #15388: vectorization support: reference x2p[pp] has aligned access   [ src/particle/move.cpp(96,13) ]
      remark #15388: vectorization support: reference x3p[pp] has aligned access   [ src/particle/move.cpp(97,13) ]
```

Aligned access

```
      remark #15415: vectorization support: irregularly indexed load was generated for the variable <fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcoup>, part of index
e.cpp(144,27) ]
      remark #15415: vectorization support: irregularly indexed load was generated for the variable <fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcoup>, part of index
e.cpp(145,27) ]
      remark #15305: vectorization support: vector length 16
      remark #15309: vectorization support: normalized vectorization overhead 0.034
      remark #26012: vectorization support: data layout of a private variable wei1 was optimized, converted to SoA
      remark #26012: vectorization support: data layout of a private variable wei2 was optimized, converted to SoA
      remark #26012: vectorization support: data layout of a private variable wei3 was optimized, converted to SoA
      remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
      remark #15448: unmasked aligned unit stride loads: 193
      remark #15449: unmasked aligned unit stride stores: 224
      remark #15462: unmasked indexed (or gather) loads: 189
      remark #15475: --- begin vector cost summary ---
      remark #15476: scalar cost: 2182
      remark #15477: vector cost: 2413.930
      remark #15478: estimated potential speedup: 0.870
      remark #15487: type converts: 6
      remark #15488: --- end vector cost summary ---
      remark #25015: Estimate of max trip count of loop=1
```

No potential speed up (but it is ok)

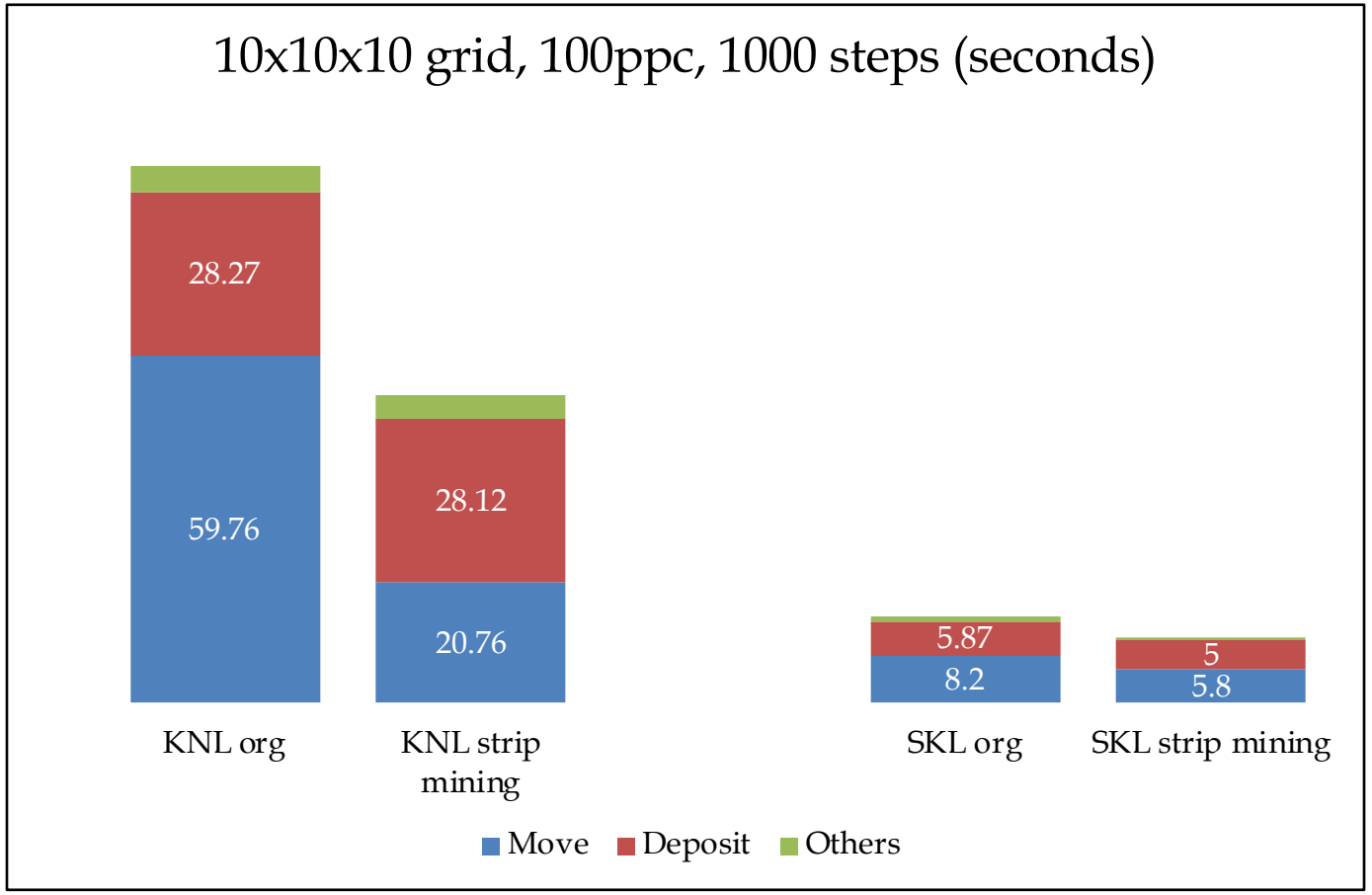
move.optrpt (2/2)

```
LOOP BEGIN at src/particle/move.cpp(153,59)
   remark #15388: vectorization support: reference x1p[pp] has aligned access    [ src/particle/move.cpp(154,15) ]
   remark #15388: vectorization support: reference x2p[pp] has aligned access    [ src/particle/move.cpp(155,15) ]
   remark #15388: vectorization support: reference x3p[pp] has aligned access    [ src/particle/move.cpp(156,15) ]
   remark #15388: vectorization support: reference v1p[pp] has aligned access    [ src/particle/move.cpp(157,15) ]
   remark #15388: vectorization support: reference v2p[pp] has aligned access    [ src/particle/move.cpp(158,15) ]
   remark #15388: vectorization support: reference v3p[pp] has aligned access    [ src/particle/move.cpp(159,15) ]
   remark #15388: vectorization support: reference bfld1v[pp] has aligned access   [ src/particle/move.cpp(165,24) ]
   remark #15388: vectorization support: reference bfld1v[pp] has aligned access   [ src/particle/move.cpp(165,37) ]
   remark #15388: vectorization support: reference bfld2v[pp] has aligned access   [ src/particle/move.cpp(165,50) ]
   remark #15388: vectorization support: reference bfld2v[pp] has aligned access   [ src/particle/move.cpp(165,63) ]
   remark #15388: vectorization support: reference bfld3v[pp] has aligned access   [ src/particle/move.cpp(165,76) ]
   remark #15388: vectorization support: reference bfld3v[pp] has aligned access   [ src/particle/move.cpp(165,88) ]
   remark #15388: vectorization support: reference x1p[pp] has aligned access    [ src/particle/move.cpp(193,7) ]
   remark #15388: vectorization support: reference x2p[pp] has aligned access    [ src/particle/move.cpp(194,7) ]
   remark #15388: vectorization support: reference x3p[pp] has aligned access    [ src/particle/move.cpp(195,7) ]
   remark #15388: vectorization support: reference v1p[pp] has aligned access    [ src/particle/move.cpp(196,7) ]
   remark #15388: vectorization support: reference v2p[pp] has aligned access    [ src/particle/move.cpp(197,7) ]
   remark #15388: vectorization support: reference v3p[pp] has aligned access    [ src/particle/move.cpp(198,7) ]
   remark #15305: vectorization support: vector length 16
   remark #15309: vectorization support: normalized vectorization overhead 0.057
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 49
   remark #15449: unmasked aligned unit stride stores: 16
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 367
   remark #15477: vector cost: 69.620
   remark #15478: estimated potential speedup: 4.980
   remark #15486: divides: 3
   remark #15488: --- end vector cost summary ---
   remark #25015: Estimate of max trip count of loop=1
LOOP END
```

Aligned access

~5x potential speed up

Performance Comparison



	Move	Deposit
KNL	2.87x	1.00x
SKL	1.41x	1.17x

Strip-Mining Lessons Learned

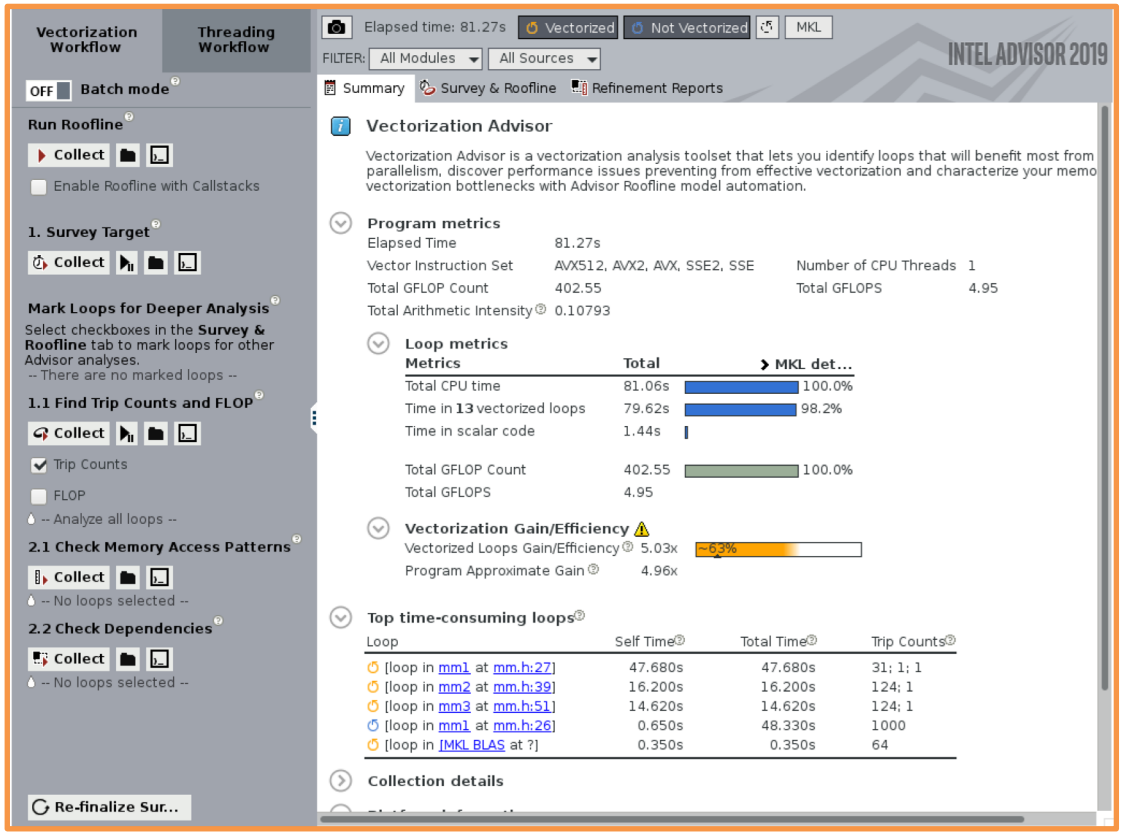
Strip-mining is useful because it:

1. Helps the compiler detects data alignment
   - Data alignment typically means two things:
   - Aligning the base-pointer where the space is allocated for the array (no peel loop)
   - Making sure the starting indices have alignment properties for each vectorized loop
2. Allows loop splitting to expose vectorization (without additional memory footprint)
   - The compiler refuses to vectorize a loop because a non-vectorizable operation is present in it (*Move* and *Deposit*)
   - Split the loop to two loops, one contains non-vectorizable operations and one contains vectorizable ones
3. Allows vectorization to co-exist with multi-threading and tiling

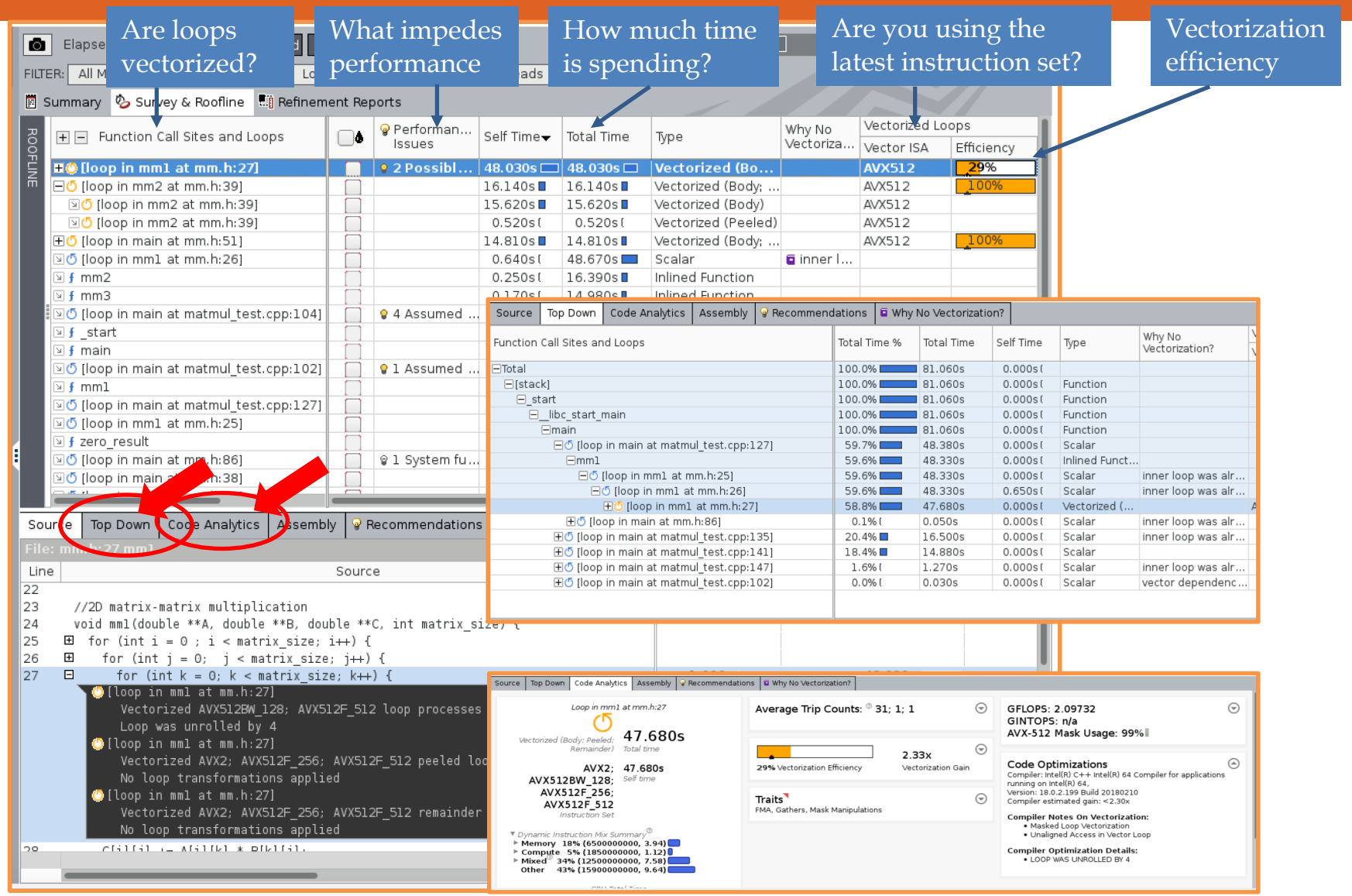
Intel Advisor

- Vectorization Advisor
  - **Survey**: find the vectorization information for loops and provide suggestions for improvement
  - **Trip Counts**: generate a Roofline Chart
  - Dependencies: determine if it is safe to force vectorization
  - Memory Access Patterns (MAP): see how you access the data

- Threading Advisor
  - Suitability: predict how well your proposed threading model will scale



Survey

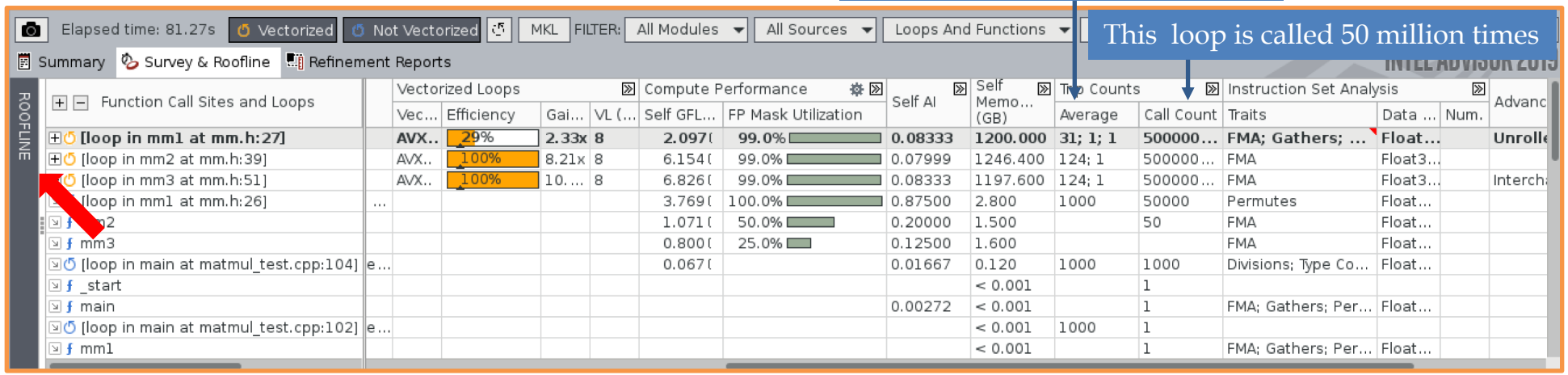


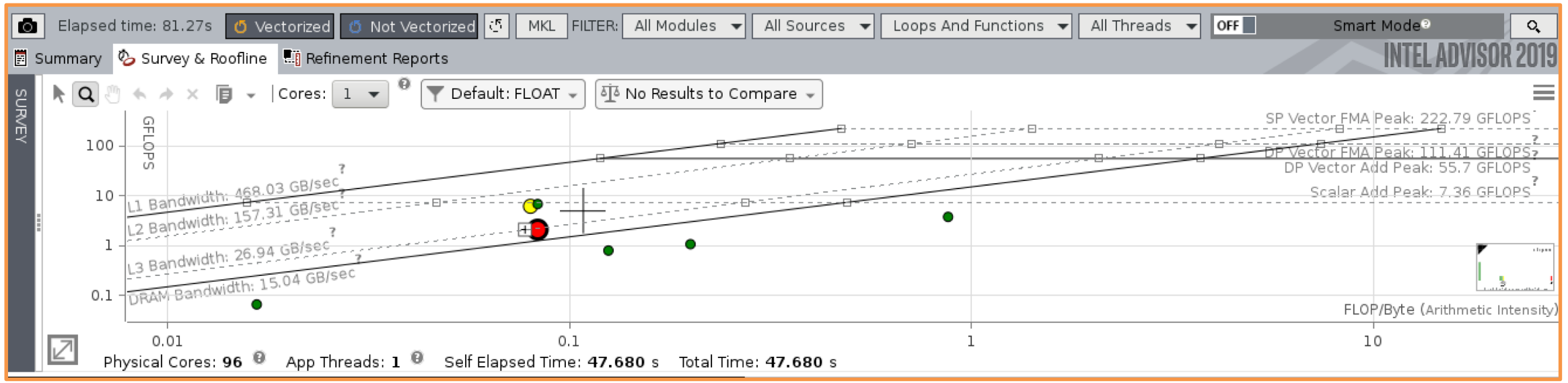
Run this command to collect the data (remotely): advixe-cl –c survey -project-dir $<PROJ_DIR> -no-auto-finalize -- $<EXE> $<ARGS>

Trip Counts

This loop's scalar count is ~248

This loop is called 50 million times





Run the following command to collect the data (remotely):

advixe-cl –c tripcounts -flop -project-dir $<PROJ_DIR> -no-auto-finalize -- $<EXE> $<ARGS>

Note: it is important to use the same project directory as the survey analysis

Exercise 5: Using Advisor to Examine Nbody Problem

- Enable X11 forwarding
  - "ssh –Y -C <user>@adroit.princeton.edu
  - Will need local xserver (XQuartz for OSX, Xming for Windows)
- Load environment modules
  - module load intel intel-advisor
- Run the provided script to submit a Advisor wrapped job to the scheduler
  - ./submit_to_scheduler
- Open the resulting directory with Intel Advisor
  - advixe-gui nbody-advisor
- Explore "Survey" report
- Run the provided script to submit a Advisor wrapped job with tripcounts analysis
  - In ./submit.slurm, change the job execution command to advisor command line with tripcounts analysis
  - ./submit_to_scheduler
- Open the resulting directory with Intel Advisor
  - advixe-gui nbody-advisor
- Explore "Tripcounts & Roofline" report

Good References

- Demystifying Vectorization, Colfax Interational, Webinar, June 2017
- Exploiting Parallelism for Intel Xeon Processors, J.D.Patel, Intel Skylake Training at Princeton, 2018
- Vector Parallelism on Multi- and Many-Core Processors, Steve Lantz and Matevz Tadel, CoDas-HEP Summer School, July 2018
- Compiling and Tuning for Performance using Intel Advanced Vector Extensions 512, Carlos Rosales-Fernandez, SC18, Intel Speakership Tutorial