Advanced OpenMP Vectorization

TACC OpenMP Team

milfeld/lars/agomez@tacc.utexas.edu

These slides & Labs: tinyurl.com/tacc-openmp

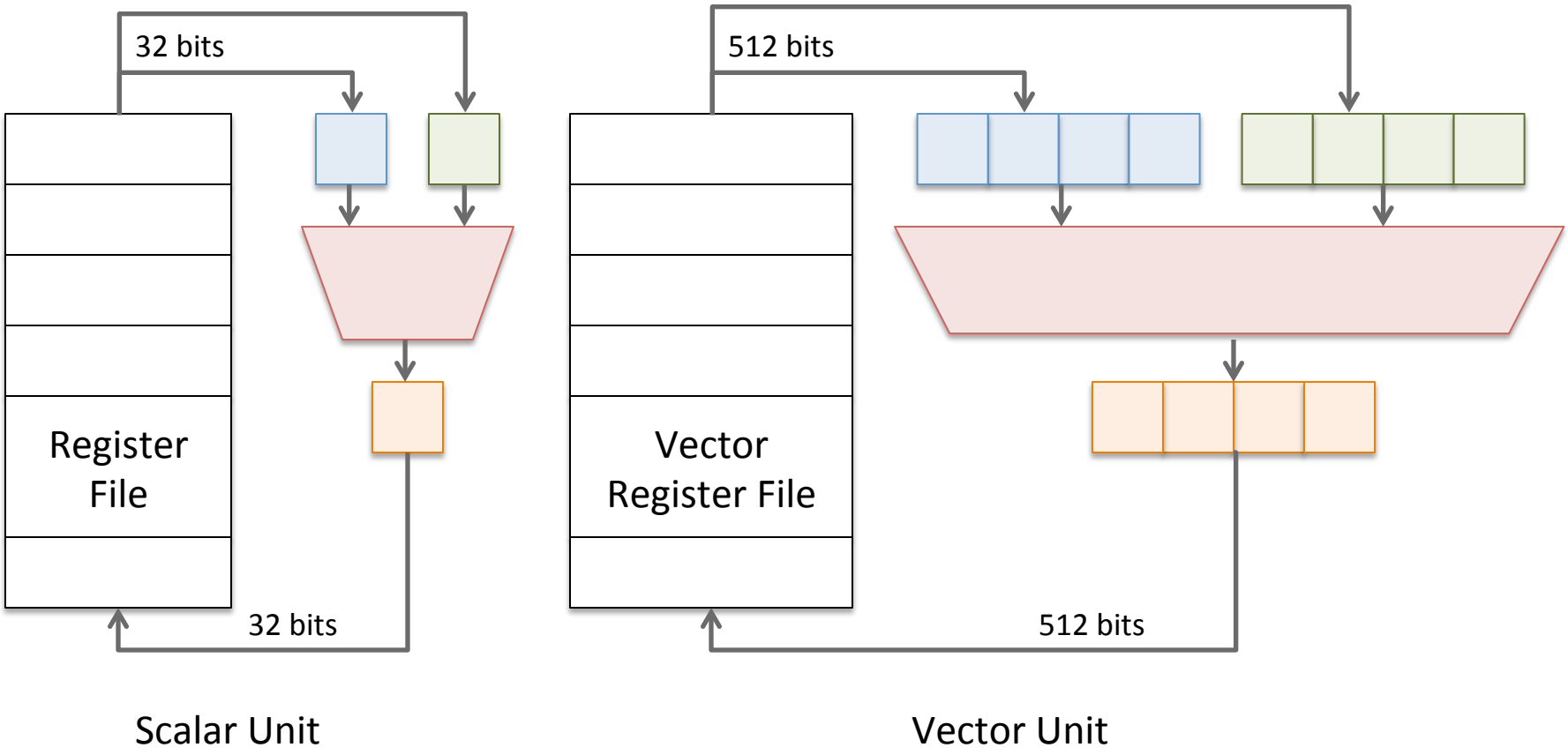
Learning objective

- Vectorization: what is that?
  - Past, present and future
  - Data dependencies
- Code transformation
- SIMD Directives
  - SIMD loop directives
  - SIMD Enabled Functions
- SIMD and Threads
  - OpenMP
- Beyond Present Directives

SIMD

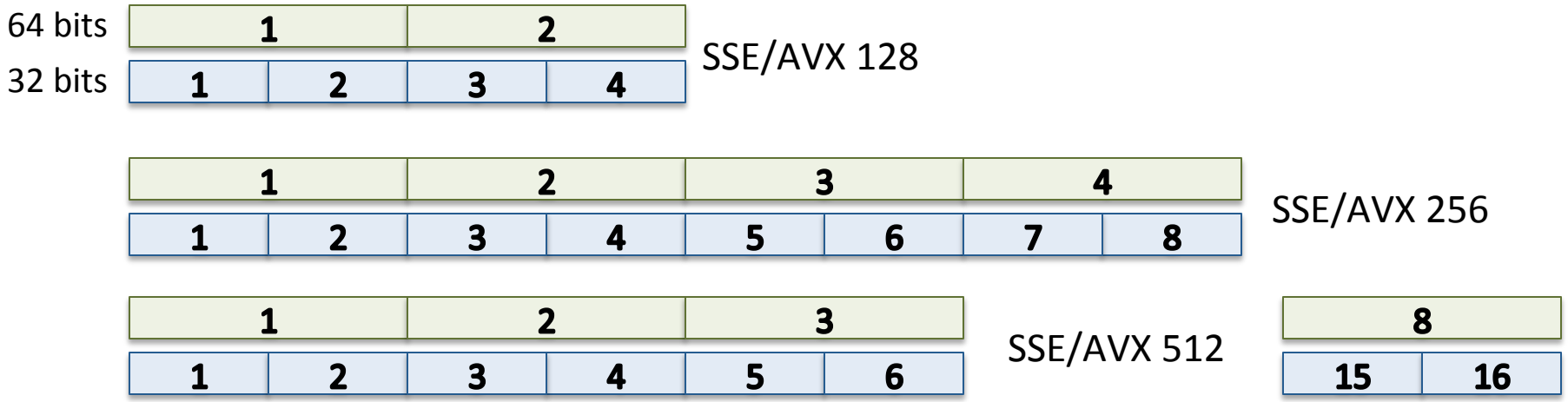
- Single Instruction, Multiple Data: an instruction applies the same operation to multiple data concurrently.
- Known as vectorization by scientific community.
- Speed Kills
  - Speed of microprocessors killed the Cray vector story in the 90's.
  - We are rediscovering how to use vectors.
  - Microprocessor vectors were 2DP long for many years.
- We live in a parallel universe
  - It's not just about parallelism with SIMD → Thread and MPI tasking.
  - Vectorization does not always imply great performance.
- SIMD directives = great start (much more can be done with the instr. set)

Vector hardware



Vector hardware

- Intel Core: 128 bits wide
- Intel Sandy Bridge - Haswell: 256 bits wide
- Intel KNC - KNL: 512 bits wide



SIMD

- Exploit parallelism by applying the same operation to multiple data in parallel
- Normally applies to array operations in loops

```
for (int i=0 ; i<N; i++) {
    a[i] = b[i] + c[i];
}
```

```
do i=1, n
    a(i) = b(i) + c(i)
end do
```

Why?

- SIMD registers are getting wider now, but there are other factors to consider, besides width.

Caches:	Maybe non-coherent, possible 9 layers of memory later
Alignment:	Avoid cache-to-register hickups
Prefetching:	Sometimes users can improve the compiler's result
Data arrangement:	AoS (Array of Structures) vs SoA (Structure of Arrays), gather, scatter, permutes
Masking:	Allows conditional execution, but you get lower performance
Striding:	1 is best

How to vectorize the code?

- The compiler will try to vectorize (conservative approach)
- The compiler can generate a vectorization report: use this report to change your code
- Use intrinsics/assembly code
- Use Cilk
- Use OpenMP 4.0 SIMD pragmas
- Use optimized libraries

Transforming the code

```
for (int i=0 ; i<N; i++) {
    a[i] = b[i] + c[i];
}
```

Compiler

```
for (int i=0 ; i<N; i+=4) {
    a[i]   = b[i]   + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
    a[i+3] = b[i+3] + c[i+3];
}
```

Transforming the code

```
for (int i=0 ; i<N; i++) {
    a[i] = b[i] + c[i];
    d[i] = e[i] + f[i];
}
```

Compiler

```
for (int i=0 ; i<N; i+=4) {
    a[i]   = b[i]   + c[i];
    d[i]   = e[i]   + f[i];
    a[i+1] = b[i+1] + c[i+1];
    d[i+1] = e[i+1] + f[i+1];
    a[i+2] = b[i+2] + c[i+2];
    d[i+2] = e[i+2] + f[i+2];
    a[i+3] = b[i+3] + c[i+3];
    d[i+3] = e[i+3] + f[i+3];
}
```

```
for (int i=0 ; i<N; i+=4) {
    a[i]   = b[i]   + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
    a[i+3] = b[i+3] + c[i+3];
    d[i]   = e[i]   + f[i];
    d[i+1] = e[i+1] + f[i+1];
    d[i+2] = e[i+2] + f[i+2];
    d[i+3] = e[i+3] + f[i+3];
}
```

The compiler can change the order of the statements if it's safe

Vectorization report

- Compiler tells whether a loop has or has not been vectorized
- Recent versions of Intel Compiler give an estimate of the speedup if the loop is vectorized (using Intel secret function)
- Flags:
  - Intel: **`-vec-report=#level`** (6 is good, deprecated)

    (changing to **`-qopt-report -qopt-reporth-phase=vec`**)
  - gcc: **`-ftree-vectorize -ftree-vectorizer-verbose`**

Vectorization Report

```
 LOOP BEGIN at kernel.cpp(27,20) inlined into kernel.cpp(74,5)
      remark #15300: LOOP WAS VECTORIZED
      remark #15448: unmasked aligned unit stride loads: 2
      remark #15449: unmasked aligned unit stride stores: 1
      remark #15450: unmasked unaligned unit stride loads: 1
      remark #15475: --- begin vector loop cost summary ---
      remark #15476: scalar loop cost: 14
      remark #15477: vector loop cost: 0.560
      remark #15478: estimated potential speedup: 24.790
      remark #15488: --- end vector loop cost summary ---
 LOOP END

LOOP BEGIN at kernel2.cc(14,3)
      remark #15344: loop was not vectorized: vector dependence prevents
                     vectorization
      remark #15346: vector dependence: assumed FLOW dependence between
                     line 15 and nuclide_grids line 15
LOOP END
```

Why is this loop not vectorized

- Some algorithms can make vectorization difficult
- Programmers can make things difficult for the compilers:
  - The number of iterations of the loop should be known at entry time
  - Single exit (and entry)
- Function calls need extra work
- Data dependencies

How to achieve automatic vectorization

- Avoid break/continue/cycle
- Avoid pointers
- Simple loops
- Countable loops
- Single data type
- Make sure that the operators work directly with the data type
- Use optimized libraries (i.e. MKL)

Data dependencies

- FLOW dependency - Read After Write (RAW): an instruction depends on the result of a previous instructions

```
for (int i=1 ; i<N; i++) {
    b[i] = b[i-1] + c[i];
}
```

- ANTI-dependency - Write After Read (WAR): an instruction needs a result that is later updated

```
for (int i=0 ; i<N-1; i++) {
    a[i] = b[i] + c[i];
    d[i] = a[i+1];
}
```

- OUTPUT dependency - Write After Write (WAW): the order of instructions will affect the result

```
for (int i=0 ; i<N-1; i++) {
    a[i] = b[i] + c[i];
    d[i] = a[i] + e[i];
    a[i] = f[i] - g [i];
}
```

- C pointers can hide data dependencies: memory locations might overlap

What to do to help the compiler (1)

- Remove data dependencies:

Reorder statements	Compiler is not always able to reorder the code
Loop fusion	Merge adjacent loops
Loop fission	Split a single loop into more than one
Loop unrolling	Expand the loop (remove branches)
Scalar expansion	Use extra storage to remove dependencies
Loop splitting/peeling	Remove some iterations

- Can you change the algorithm?

What to do to help the compiler (2)

- SIMD directives
- Give compilers the help they want for vectorization
  - Assures independence of operations
  - "Do as I say, because I know what I'm doing"
- Directives:
  - Loops
  - No C arrays; has omp sections
  - Functions
  - C/C++ and Fortran

OpenMP SIMD evolution

- First appeared in OpenMP 4.0 2013
- Directives
  - SIMD
  - SIMD + worksharing for do/for loops
  - declare SIMD

- SIMD refinements in OpenMP 4.5 (2015).

Why we need SIMD pragmas

- Often independent-iteration loops don't vectorize.
  - Reason for vectorization failure: too many pointers, complicated indexing …
  - SIMD pragmas instruct the compiler to create SIMD operations for iterations of the loops.

Without pragma, vec-report=2 was helpful:

remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

```
void foo(double a[n][n], double b[n][n], int end){
#pragma omp simd
for (int i=0 ; i<end ; i++) {
     a[i][0] = (b[i][0] - b[i+1][0]);
     a[i][1] = (b[i][1] - b[i+1][1]);
   }
}
```

OpenMP simd

```
#pragma omp simd [clause][[,]clause]                    (C/C++)
    for (…;…;…)
```

```
!$omp simd [clause][[,]clause]                          (F90)
    do-loop
!$omp end simd
```

- Clause:

safelen(length)	maximum distance between concurrent instructions
simdlen(length)	preferred number of iterations to be executed concurrently
linear(list[:linear-step])	linear relationship respect the iteration space
aligned(list[:alignment])	
private(list)	
lastprivate(list)	
reduction(op: list)	
collapse(n)	

OpenMP simd

```
#pragma omp simd
for (int i=0; i<n; i++) {
    a[i] = b[i] + c[i] * alpha;
}
```

```
!$omp simd
do i=1, n
    a(i) = b(i) + c(i) * alpha
end do
!$omp end simd
```

OpenMP simd

```
#pragma omp simd private(tmp) reduction(+:sum)
for (int i=0; i<n; i++) {
     tmp = b[i] + c[i] * alpha;
     sum += temp;
}
```

```
!$omp simd private(tmp) reduction(+:sum)
do i=1, n
     tmp = b(i) + c(i) * alpha
     sum = sum + sum
end do
!$omp end simd
```

OpenMP simd

```
#pragma omp simd safelen(8)
for (int i=0; i<n; i++) {
    a[i] = b[i] + c[i] * alpha;
}
```

```
!$omp simd safelen(8)
do i=1, n
    a[i] = b(i) + c(i) * alpha
end do
!$omp end simd
```

SIMD enabled functions

- SIMDizable functions: can be invoked with either scalar or array elements

```
double foo(double r, double s, double t);

void driver (double R[N], double S[N], double T[N]){
   for (int i=0; i<N; i++){
     A[i] = foo(R[i],S[i],T[i]);
   }
}
```

OpenMP declare simd

- Applied to a function to create one or more versions of the function that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop

```
#pragma omp declare simd [clause][[,]clause]   (C/C++)
```

```
!$omp declare simd [clause][[,]clause]         (F90)
```

- Clause:

simdlen(length)	Preferred number of iterations to be executed concurrently
linear(list[:linear-step])	Objects in *list* have a linear relationship with respect to the iteration
aligned(list[:alignment])	Objects in *list* are aligned to the number of bytes indicated
uniform(list)	Objects in *list* have invariant value for all concurrent invocations
inbranch	Function will be always called from inside a conditional statement
notinbranch	Function will ever be called from a conditional statement

SIMD enabled functions

```
double foo(double r, double s, double t);

void driver (double R[N], double S[N], double T[N]){
    for (int i=0; i<N; i++){
      A[i] = foo(R[i],S[i],T[i]);
    }
}
```

```
#pragma omp declare simd simdlen(4)
#pragma omp declare simd notinbranch
double foo(double r, double s, double t);

void driver (double R[N], double S[N], double T[N]){
    #pragma omp simd
    for (int i=0; i<N; i++){
      A[i] = foo(R[i],S[i],T[i]);
    }
}
```

SIMD enabled functions

```
double foo(double* r, double* s, int i, double c) {
   return r[i] * s[i] + c;
}

double bar(double* r, double* s, double c) {
   return *r * *s + c;
}

void driver (double* r, double* s, double* res, double c){
   for (int i=0; i<N; i++){
     res[i] = foo(r, s, i, c);
   }
   for (int i=0; i<N; i++){
     res[i] = bar(&r[i], &s[i], c);
   }
}
```

SIMD enabled functions

```
#pragma omp declare simd uniform(r,s,c) linear(i:1)
double foo(double* r, double* s, int i, double c) {
   return r[i] * s[i] + c;
}

#pragma omp declare simd uniform(c) linear(r,s:1)
double bar(double* r, double* s, double c) {
   return *r * *s + c;
}

void driver (double* r, double* s, double* res, double c){
   #pragma omp simd
   for (int i=0; i<N; i++){
     res[i] = foo(r, s, i, c);
   }
   #pragma omp simd
   for (int i=0; i<N; i++){
     res[i] = bar(&r[i], &s[i], c);
   }
}
```

SIMD enabled functions

```
function foo(r, s, i, c) result (a) {
!$omp declare simd(foo) uniform(r,s,c) linear(i:1)
   implicit none
   integer           :: i
   double precision :: r(*), s(*), c, a
   a = r(i) + b(i) + c
end function


subroutine driver (r, s, N, c)
   implicit none
   double precision           :: r(N), s(N), res(N), c
   integer                    :: N,I
   double precision, external :: foo
   !$omp simd
   do i=1, N
       res(i) = foo(r, s, i, c)
   end do
end subroutine
```

SIMD and threads – OpenMP worksharing

- OMP Directives Workshares and SIMDizes loop
  - Creates SIMD loop uses chunks in increments of the vector size.
  - Remaining iterations are distributed “consistently”.

Syntax: combined directives

```
#pragma omp parallel for simd [clause][[,]clause]   (C/C++)
```

```
!$omp parallel do simd [clause][[,]clause]      (F90)
```

clauses:

any do/for clause — data sharing attributes, nowait, etc.

any SIMD clause …

SIMD and threads – OpenMP worksharing

```
#pragma omp declare simd
double foo(double r, double s, double t);


#pragma omp parallel for simd
for (i=start; i<end; i++){
    foo(a[i], b[i], i);
}
```

Extra stuff!!

- The following slides are not about OpenMP
- You might need some of them to get good performance

Getting good performance

- Data must be in cache -> prefetching, cache reuse,…
- Alignment
- Padding
- Use efficient vector operation

Alignment

C

```
float A[N] __attribute__((align(64));
__declspec(align(64)) float A[N];
_mm_malloc() / _mm_free()
__assume_aligned(A,64)
```

Fortran

```
!dir$ attributes align: 64::A
!dir$ assume_aligned A:64
```

Alignment

```
void foo() {
    float a[N] __attribute__((aligned(32)));
    float *B;
    B = (float *) _mm_malloc(N*sizeof(float),32);
    …
    _mm_free(B);
}
```

Aliasing

- Sometimes functions won't be automatically vectorized when pointers are involved
  - Two different points might point to the same location (aliasing)
- If you know that this is never the case: restrict keyword
  - You also want to pass the -restrict flag to the compiler.
  - Maybe you even want the -ansi-alias flag

Aliasing

```
void foo(float* a, float* b, float* c, float d) {
    for (int i=0; i<N; ++i) {
        c[i] = a[i] + b[i]*d;
    }
}
```

```
void foo(float * restrict a, float * restrict b, float
* restrict c, float d) {
    for (int i=0; i<N; ++i) {
        c[i] = a[i] + b[i]*d;
    }
}
```

```
$ icc -restrict source.c -o source.out
```

Prefetching

- This is not part of OpenMP!!
- Used by Intel for the MIC architecture.
- It might be very important to get good performance

```
#pragma prefetch [var[:hint[:distance]]
```

```
void foo(float* a, float* b, float* c, float d) {
    #pragma prefetch a:1:64
    #pragma prefetch a:0:6
    for (int i=0; i<N; ++i) {
        c[i] = a[i] + b[i]*d;
    }
}
```

Comparing performance of vectorization

1. Compile your code without vectorization (-no-vec)
2. Time the execution
3. Compile code without optimizations
4. Time the execution
5. Compile optimized code (removed dependencies, fixed alignment,…)
6. Time the execution