OpenMP
SIMD Vectorization

Lars Koesterke

Charlie Dey

PCSE 5/1/18

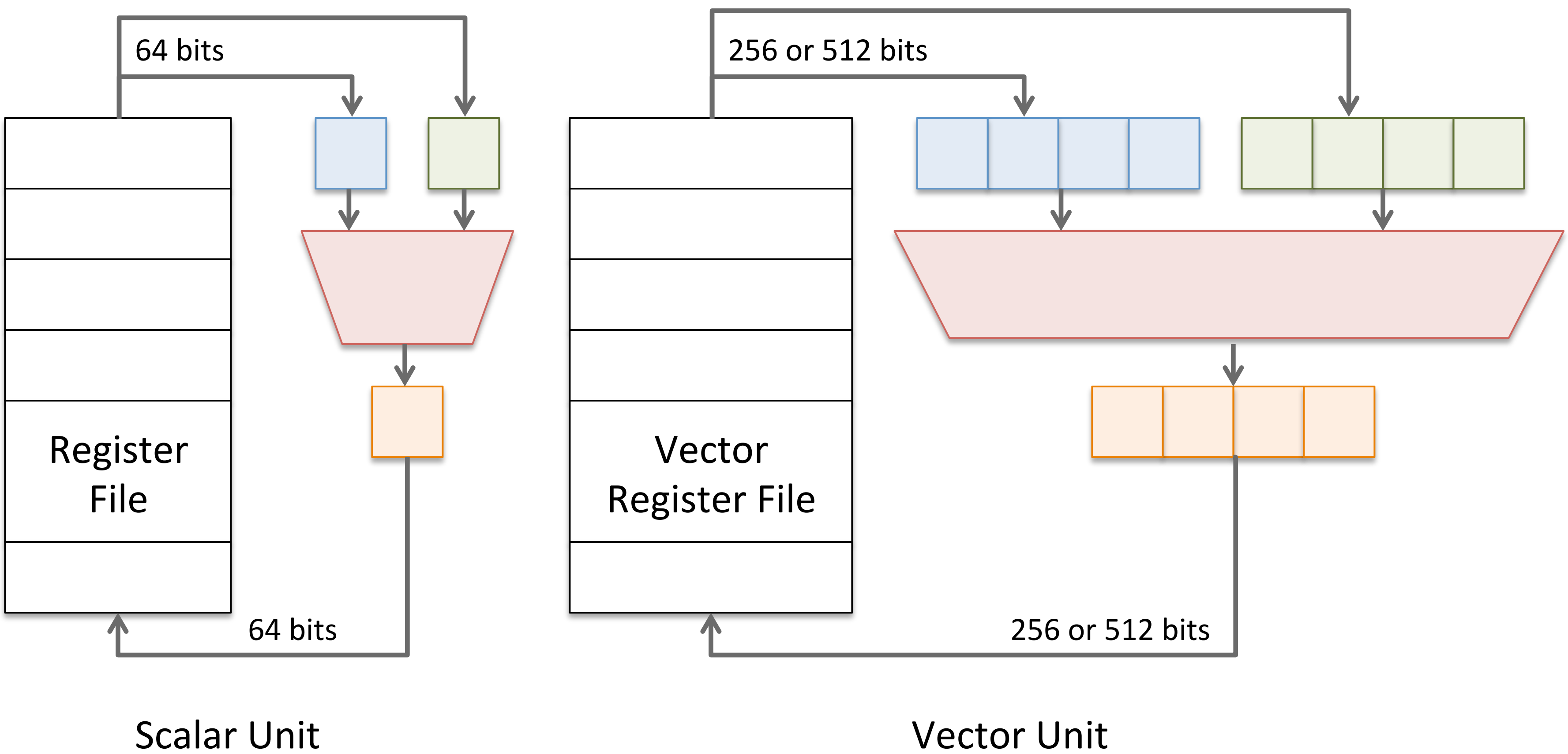
Learning objective

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

SIMD

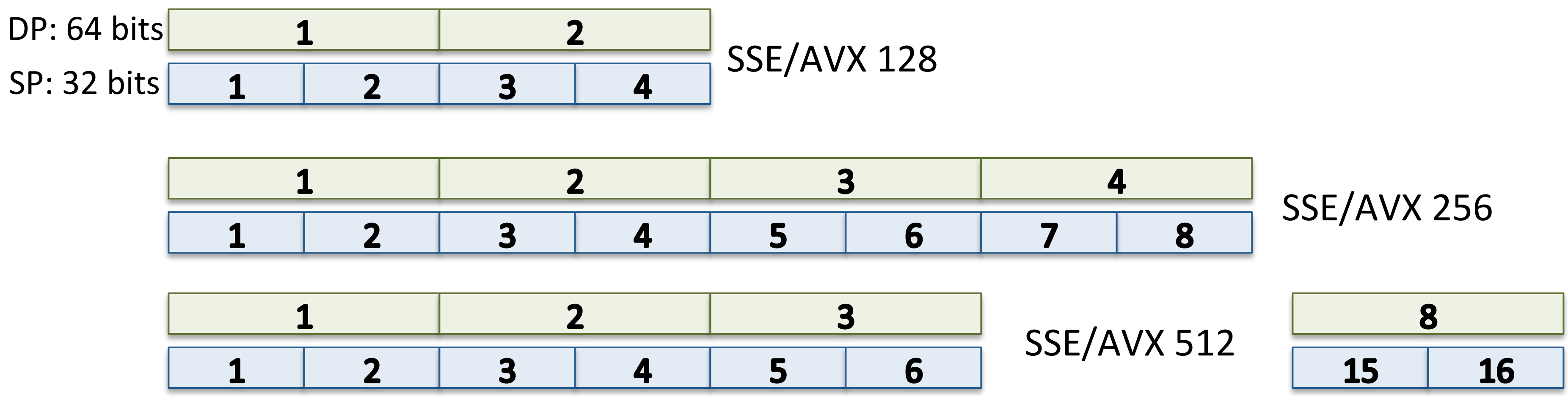
- Single Instruction, Multiple Data: an instruction applies the same operation to multiple data concurrently.
- Known as vectorization by scientific community.
- Speed Kills
  - Speed of microprocessors killed the Cray vector story in the 90's.
  - We are rediscovering how to use vectors
    - Intel & AMD: SSE2 since ~2001-2003
  - Microprocessor vectors were 2DP long (SSE) for many years.
- We live in a parallel universe: triple nested hierarchy
  - Parallelism: vector lanes (SIMD), cores (OpenMP or MPI), and nodes (MPI)
  - 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 Skylake server: 512 bits wide (Stampede2)
- Intel KNC - KNL: 512 bits wide (Stampede2)



SIMD

- Exploit parallelism by applying the same operation to multiple data in parallel
- Typically 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
```

Other Optimization Targets

- 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 hiccups
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, random is worst

How to vectorize the code?

- The compiler will attempt to vectorize (conservative approach: whenever safety can be guaranteed)
- The compiler can generate a vectorization report: use this report to guide code changes
- Use intrinsics/assembly code
- Use Cilk
- Use OpenMP 4.0 SIMD pragmas
  - Help or force the compiler to use SIMD instructions
  - Its portable!
- Use optimized libraries

Transforming the code
Example of 'Loop Unrolling'

```
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];
}
```

Scalar execution: 4 instructions
Vector execution: 1 SIMD instruction (if the compiler can proof correctness)

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 statements if it's safe
- Compiler may issue 2 SIMD instructions

Vectorization report

- Compiler tells whether a loop has or has not been vectorized
  - Compiler reveals some of the reasoning
  - Output created for every loop
  - User has to 'know' which loops are 'hot' (profiling essential)
- Recent versions of Intel Compiler give an estimate of the speedup if the loop is vectorized (Intel guesstimate)
- Compiler flags:
  - Intel: **`-qopt-report=2 -qopt-report-phase=vec`**)
  - gcc: **`-ftree-vectorize -ftree-vectorizer-verbose`**

Vectorization Report: Example

```
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
```


TACC
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

Some loops cannot be vectorized

- Data dependencies in loop bodies
  - Dependencies introduced by algorithm
  - Dependencies introduced by programmer
- Compiler may struggle to prove correctness of vectorization in complex loop bodies/complex code
  - Also when combined with other optimizations
- Additional complications
  - Loops with unknown number of iterations (while loop)
  - Loops with multiple exits
- Function calls add complexity (inlining)

Vectorization is not always efficient

- Vectorization with stride-1 memory access
  - 1 or 2 instructions to load/store data into registers
- High-stride or random memory access
  - 4 or 8 instructions to load/store data

Achieving automatic vectorization

- Read the vector report
  - Modify code (if possible) according to recommendation
- Follow guidelines as outline before
  - Stride-1 memory access
  - Avoid break/continue/cycle
  - Avoid pointer arithmetic
  - 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

Helping 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?
- Follow guidance from vector report

Helping the compiler (2)

- Add OpenMP SIMD directives
- Provide compiler with guidance
  - Assures independence of operations
  - "Do as I say, because I know what I'm doing"
- Directives:
  - Loops
  - No C arrays
  - Sections
  - Functions
  - C/C++ and Fortran

OpenMP SIMD evolution

- First appearance in OpenMP 4.0   (2013)
- Directives
  - SIMD for do/for loops
  - SIMD + worksharing for do/for loops
  - declare SIMD (for functions)

- SIMD refinements in OpenMP 4.5 (2015).

Why do we need SIMD pragmas

- Often independent-iteration loops don't vectorize.
  - Reason for vectorization failure: 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                    !optional:
```

- 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

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

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

OpenMP simd

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

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

SIMD enabled functions

- SIMDizable functions: can be invoked with either scalar or array elements
- Think of it as “inlining” with vector capability.

Consider:

```
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 an 'if' block
notinbranch	Function will never be called from inside an 'if' block

SIMD enabled functions

C/C++

```
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

C/C++

```
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

C/C++

```
#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

F90

```
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 can Workshare and SIMDize loop
  - Creates SIMD loop with chunk sizes 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

34

Alignment

C/C++

```
__attribute__((align(64)) float A[N];
__declspec(align(64))     float A[N];  // old syntax


_mm_malloc(size,64) / _mm_free()

__assume_aligned(A,64)
```

STACK

HEAP

Advise

Fortran

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


35

Alignment

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


36

Aliasing

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

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
```


38

Prefetching

- This is not part of OpenMP!!
- Used “extensively” 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