Vectorization with OpenMP

Kevin Olson   Ning Li

April 22, 2017

Experts in numerical algorithms
and HPC services

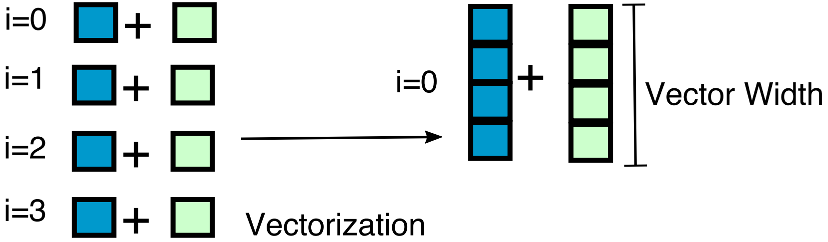
Introduction

- Here you will learn how to vectorize code using OpenMP
- Vectorization in some ways harder than parallelism
- Also less understood

What is Vectorization?

- Vectorization is a form of parallelism where the same operation is applied simultaneously to two, four, or more pieces of data.
- It utilizes special hardware rather than extra cores
- Mainly focuses on arithmetic in loops
- If the loop can operate in parallel, it can usually "vectorize"
- To get best performance, generally parallelize outer loops, vectorize inner loops (but both can be applied to a single loop).

What is Vectorization?



The addition operation is applied to all the data from i = 0 to i = 3 *at the same time*.

What Vectorization can do

- Improves performance to arithmetic heavy code
- Code must be specifically written with this in mind
- Sometimes rewriting is necessary

What Vectorization can do

- Vector registers range from 16 bytes to 64 bytes
- Sometimes more than one vector ALU
- e.g. The Xeon Phi processor has 2 vector ALUs with 64 byte wide registers
  - This is where the 2*16=32X speedup number comes from
  - (one single precision float is 4 bytes)

What Vectorization can do

Code such as

```
for(int i=0;i<32;i++){
  x[i]=y[i]+z[i];
}
```

potentially executed in a single clock cycle

What Vectorization can not do

- Vectorization does have limitations
- Mainly supports elementary operations and some special functions
- Not possible to vectorize arbitrary loops

Using OpenMP to vectorize code

- Starting in 4.0 with improvements in 4.5, OpenMP can vectorize loops
- Syntax deliberately similar to parallel
- But previous restrictions still apply

Using OpenMP to vectorize code

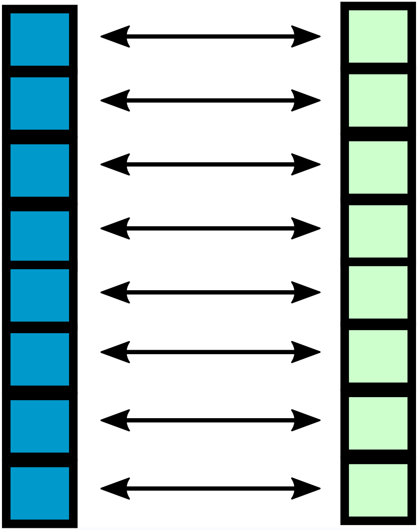
- Let's start with the simple case of vector add
- Start with a for loop
- Insert relevant pragma directing compiler to vectorize
- C: `#pragma omp simd`
- Fortran: `!$OMP SIMD` (NOTE: NO "END" necessary)

Using OpenMP to vectorize code

```c
#pragma omp simd
for(int i=0;i<32;i++){
  x[i]=y[i]+z[i];
}
```

```fortran
!$OMP SIMD
Do i = 1, 32
   x(i)=y(i)+z(i)
End Do
```

Using OpenMP to vectorize code



Unit Stride

- This is known as "unit stride access"
- Because each successive memory access is 1 unit (word) away
- Data items are "next" to each other in memory

Unit Stride: Examples

- Many applications can be rewritten into unit stride access
- Some immediately apply though:
  - level 2 BLAS (vector-vector operations)

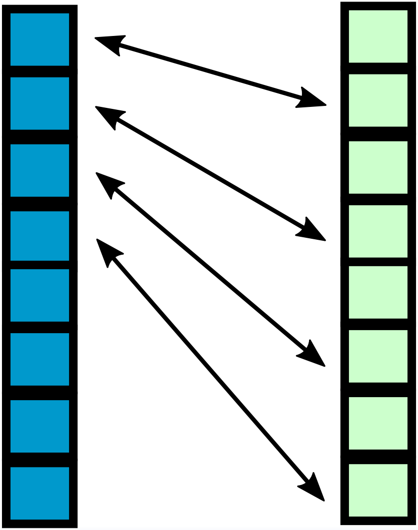
Unit Stride

- Unit stride is usually the most efficient
- Try to use this wherever possible
- But more general options available also
  - General strided access
  - Gather/scatter access

General Stride

```
#pragma omp simd
for(int i=0;i<32;i++){
  //p,q,r are integers
  x[p*i]=y[q*i]+z[r*i];
}
```

General Stride



General Stride: Examples

- General strides come up often with multidimensional arrays
- Tensor-tensor operations
- Finite difference stencils

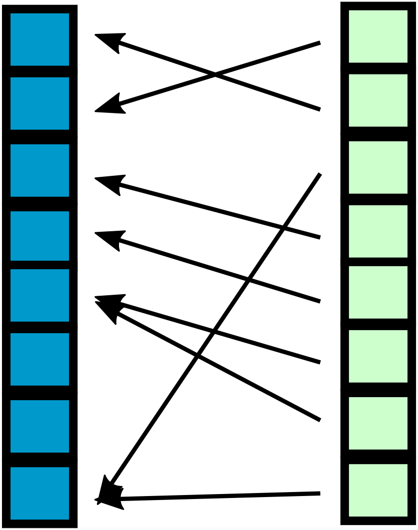
General Stride

- Strided access a little less efficient
- But still faster than sequential
- More general yet are gather/scatter

gather

```
#pragma omp simd
for(int i=0;i<32;i++){
  //ids is array of integers (indices)
  x[ids[i]]=y[i]+z[i];
}
```

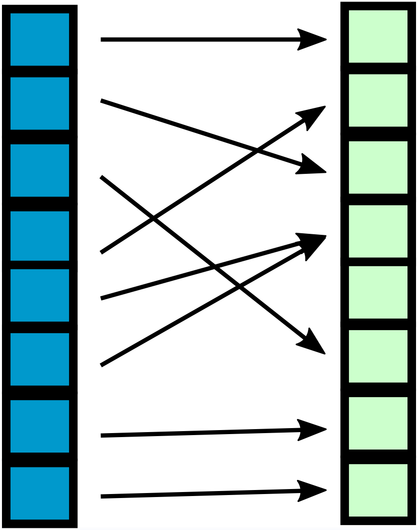
gather



scatter

```
#pragma omp simd
for(int i=0;i<32;i++){
  //ids1,ids2 are arrays of integers (indices)
  x[i]=y[ids1[i]]+z[ids2[i]];
}
```

scatter



Gather/scatter: Examples

- Gather/scatter useful on "random access" cases
- Sparse matrices
- Binning

gather/scatter

- Gather/scatter are the slowest
- Also not all platforms support them
- Xeon Phi processor does, latest Xeon does, older Xeon will emulate

Adding Complexity: Masked

- Also possible to vectorize limited conditional
- mainly simple if statement
- These are known as "masked" instructions

Adding Complexity: Masked

```
#pragma omp simd
for(int i=0;i<32;i++){
  if(y[i]>5.0)
    x[i]=y[i]+z[i];
  else
    x[i]=y[i]-z[i];
}
```

Adding Complexity: Reduction

- Finally, there are no "threads" in vectorization
- But limited "communication" possible between vector lanes
- Exactly same as in parallel case: reductions

Adding Complexity: Reduction

```
float red=0.0;
vector<float> x(32,1.0),y(32,1.0),z(32,1.0);
#pragma omp simd reduction(+:red)
for(int i=0;i<32;i++){
    x[i]=y[i]+z[i];
    red+=x[i];
}
cout<<"red="<<red<<endl;
```

```
red=64
```

Common Pitfalls

- Previous examples show where it works
- But also plenty of cases where it will not
- Unfortunately compiler can not error on compile
  - Necessary to test codes well to catch anything wrong

Loop Carried Dependency

- Loop carried dependencies cause problems for vectorization
- This is where one iterate depends on another
- For example memory-write at i=2 is read at i=4 (forward dependency)

Loop Carried Dependency

Compiling the following code

```
float red=0.0;
vector<float>x(32,1.0);
#pragma omp simd reduction(+:red)
for(int i=1;i<32;i++){
  x[i]=x[i-1]+5;
  red+=x[i];
}
cout<<"red="<<red<<endl;
```

Without vectorization:

```
red=2511
```

With vectorization:

```
red=276
```

Aliasing

- Aliased pointers in C can also cause problems
- Two pointers in C alias if the memory they point to overlaps
- This creates a loop carried dependency

Aliasing

Compiling the following code

```
float red=0.0;
vector<float>x(32,1.0);
float* y=&x[0]-1; // Pointing y at x's data
#pragma omp simd reduction(+:red)
for(int i=1;i<32;i++){
  x[i]=y[i]+5;
  red+=x[i]; }
cout<<"red="<<red<<endl;
```

Without vectorization:

```
red=2511
```

With vectorization:

```
red=276
```

Complex Code

- Another big cause of bad vectorization is too complex code
- For example: arbitrary function calls

Complex Code

With the OpenMP simd directive, the code

```
float custom_function(float in) \
      {return in*in + 0.2*in*in*in + in;}
#pragma omp simd
for(int i=0;i<32;i++){
  x[i]=custom_function(y[i]+z[i]);
}
```

can vectorize, but it will serialize the function call unless it is inlined.

Later we will see a good fix for this situation, so that loop body does not have to be completely inlined.

Advanced OpenMP Vectorization

- We close here with some advanced techniques
- For example: How to make a function "vectorizable"
- This solves the last example of bad vectorization

SIMD Functions

- A SIMD function is the OpenMP way to vectorize a function
- Let's use the failed example earlier and fix it

SIMD Functions

```
#pragma omp declare simd
__attribute__((nothrow))  \
    float custom_function(float in) \
    {return in*in + 0.2*in*in*in + in;}
#pragma omp simd
for(int i=0;i<32;i++){
  x[i]=custom_function(y[i]+z[i]); }
```

SIMD Functions: Fortran

- Fortran is slightly different
- Function or subroutine is declared SIMD inside the subroutine code
- For a more detailed discussion see: https://software.intel.com/en-us/articles/explicit-vector-programming-in-fortran

SIMD Functions

```
!$OMP SIMD
Do i = 1, 32
   Call custom_function(y(i)+z(i),x(i))
End Do

Subroutine custom_function(y,x)
!$OMP DECLARE SIMD(custom_function)
  real, intent(in)  :: y
  real, intent(out) :: x
! some code
End Subroutine custom_function
```

NOTE: Declaration should also be included in any interface blocks (not shown here)

SIMD Functions

- The simd function capability of OpenMP has a lot of features
- Instead of showing them all here, we leave to the practical
- It can give a lot of control over how the compiler vectorizes

Reading the Optimization Report

- You can diagnose many problems by reading the Intel optimization report
- It will explain when vectorization failed and why
- It will also show how code is vectorized, helping to diagnose performance issues

Reading the Optimization Report

The optimization report for vectorization can be generated with the compile flags:
-qopt-report-phase=vec -qopt-report=5

Summary

- Here you learned how to vectorize using OpenMP
- These are portable tools
- Used well, they can help code perform across architectures

Practical Exercise

VECTORIZING A CODE