Vectorization

V. Ruggiero (v.ruggiero@cineca.it)
Roma, 19 July 2017
SuperComputing Applications and Innovation Department

Outline

Topics

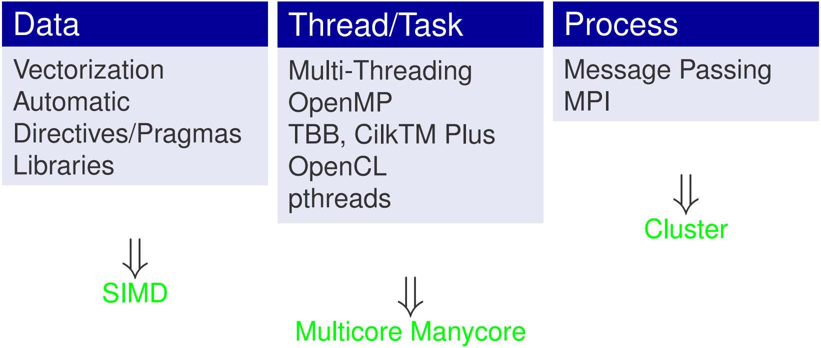
Introduction

Data Dependencies

Overcoming limitations to SIMD-Vectorization

Vectorization

Parallelism



Data

Vectorization
Automatic
Directives/Pragmas
Libraries

⇓ SIMD

Thread/Task

Multi-Threading
OpenMP
TBB, CilkTM Plus
OpenCL
pthreads

⇓ Multicore Manycore

Process

Message Passing
MPI

⇓ Cluster

Topics covered

- What are the microprocessor vector extensions or SIMD (Single Instruction Multiple Data Units)
- How to use them
  - Through the compiler via automatic vectorization
    - Manual transformations that enable vectorization
    - Directives to guide the compiler
  - Through intrinsics
- Main focus on vectorizing through the compiler
  - Code more readable
  - Code portable

Outline

Topics

Introduction

Data Dependencies

Overcoming limitations to SIMD-Vectorization

Vectorization

What is Vectorization?

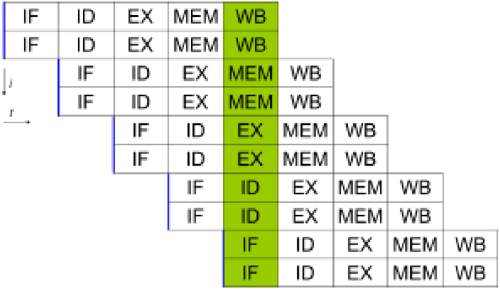
- Hardware Perspective: Specialized instructions, registers, or functional units to allow in-core parallelism for operations on arrays (vectors) of data.
- Compiler Perspective: Determine how and when it is possible to express computations in terms of vector instructions
- User Perspective: Determine how to write code in a manner that allows the compiler to deduce that vectorization is possible.

What Happened To Clock Speed?

- Everyone loves to misquote Moore's Law:
  - "CPU speed doubles every 18 months."
- Correct formulation:
  - "Available on-die transistor density doubles every 18 months."
- For a while, this meant easy increases in clock speed
- Greater transistor density means more logic space on a chip

Clock Speed Wasn’t Everything

- Chip designers increased performance by adding sophisticated features to improve code efficiency.
- Branch-prediction hardware.
- Out-of-order and speculative execution.
- Superscalar chips.
- Superscalar chips look like conventional single-core chips to the OS.
- Behind the scenes, they use parallel instruction pipelines to (potentially) issue multiple instructions simultaneously.



IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

SIMD Parallelism

- CPU designers had, in fact, been exposing explicit parallelism for a while.
- MMX is an early example of a SIMD (Single Instruction Multiple Data) instruction set.
  - Also called a vector instruction set.
- Normal, scalar instructions operate on single items in memory.
  - Can be different size in terms of bytes, of course.
  - Standard x86 arithmetic instructions are scalar. (ADD, SUB, etc.)
- Vector instructions operate on packed vectors in memory.
- A packed vector is conceptually just a small array of values in memory.
  - A 128-bit vector can be two doubles, four floats, four int32s, etc.
  - The elements of a 128-bit single vector can be thought of as v[0], v[1], v[2], and v[3].

SIMD Parallelism

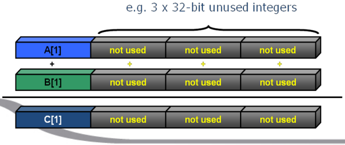
- Vector instructions are handled by an additional unit in the CPU core, called something like a vector arithmetic unit.
- If used to their potential, they can allow you to perform the same operation on multiple pieces of data in a single instruction.
    - Single-Instruction, Multiple Data parallelism.
    - Your algorithm may not be amenable to this...
    - ... But lots are. (Spatially-local inner loops over arrays are a classic.)
- It has traditionally been hard for the compiler to vectorise code efficiently, except in trivial cases.
    - It would suck to have to write in assembly to use vector instructions...

Vector units

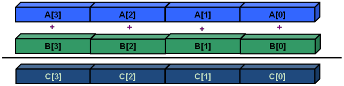
- Auto-vectorization is transforming sequential code to exploit the SIMD (Single Instruction Multiple Data) instructions within the processor to speed up execution times
- Vector Units performs parallel floating/integer point operations on dedicate SIMD units
  - Intel: MMX, SSE, SSE2, SSE3, SSE4, AVX
- Think vectorization in terms of loop unrolling
- Example: summing 2 arrays of 4 elements in one single instruction

```
C(0) = A(0) + B(0)
C(1) = A(1) + B(1)
C(2) = A(2) + B(2)
C(3) = A(3) + B(3)
```

no vectorization



vectorization



SIMD - evolution

- SSE: 128 bit register (Intel Core - AMD Opteron)
  - 4 floating/integer operations in single precision
  - 2 floating/integer operations in double precision
- AVX: 256 bit register (Intel Sandy Bridge - AMD Bulldozer)
  - 8 floating/integer operations in single precision
  - 4 floating/integer operations in double precision
- MIC: 512 bit register (Intel Knights Corner - 2013)
  - 16 floating/integer operations in single precision
  - 8 floating/integer operations in double precision

Executing Our Simple Example

- Intel Haswell 2.40 GHz per node

  intel 16.0.3 gnu 4.9.2 pgi 16.3

- KNC 1.1 GHz

  MIC 16.0.3

- KNL 1.40 GHz

  KNL 17.0.4

S000

```
for (i=0; i<LEN; i++)
c[i] = a[i] + b[i];
```

scalar	3.45	3.43	3.41	74.18	29.30
vectorized	2.18	2.14	2.27	8.94	4.15
speedup	1.58	1.60	1.50	8.30	7.07

How do we access the SIMD units?

- C or fortran code and vectorizing compiler

```
for (i=0; i<LEN; i++)
c[i] = a[i] + b[i];
```

- Macros or Vector Intrinsics

```
void example(){
__m128 rA, rB, rC;
for (int i = 0; i <LEN; i+=4){
rA = _mm_load_ps(&a[i]);
rB = _mm_load_ps(&b[i]);
rC = _mm_add_ps(rA,rB);
_mm_store_ps(&C[i], rC);
}}
```

- Assembly Language

```
..B8.5
movaps a(,%rdx,4), %xmm0
addps b(,%rdx,4), %xmm0
movaps  %xmm0, c(,%rdx,4)
addq $4, %rdx
cmpq $rdi, %rdx
ji ..B8.5
```

Vector-aware coding

- Know what makes vectorizable at all
  - "for" loops (in C) or "do" loops (in fortran) that meet certain constraints
- Know where vectorization will help
- Evaluate compiler output
  - Is it really vectorizing where you think it should?
- Evaluate execution performance
  - Compare to theoretical speedup
- Know data access patterns to maximize efficiency
- Implement fixes: directives, compilation flags, and code changes
  - Remove constructs that make vectorization impossible/impractical
  - Encourage and (or) force vectorization when compiler doesn’t, but should
  - Better memory access patterns

Writing Vector Loops

- Basic requirements of vectorizable loops:
  - Countable at runtime
    - Number of loop iterations is known before loop executes
    - No conditional termination (break statements)
  - Have single control flow
    - No Switch statements
    - ’if’ statements are allowable when they can be implemented as masked assignments
  - Must be the innermost loop if nested
    - Compiler may reverse loop order as an optimization!
  - No function calls
    - Basic math is allowed: pow(), sqrt(), sin(), etc
    - Some inline functions allowed

When vectorization fails

- Not Inner Loop: only the inner loop of a nested loop may be vectorized, unless some previous optimization has produced a reduced nest level. On some occasions the compiler can vectorize an outer loop, but obviously this message will not then be generated.
- Low trip count:The loop does not have sufficient iterations for vectorization to be worthwhile.
- Vectorization possible but seems inefficient:the compiler has concluded that vectorizing the loop would not improve performance. You can override this by placing **`#pragma vector always`** (C C++) or **`!dir$ vector always`** (Fortran) before the loop in question
- Contains unvectorizable statement: certain statements, such as those involving switch and printf , cannot be vectorized

When vectorization fails

- Subscript too complex: an array subscript may be too complicated for the compiler to handle. You should always try to use simplified subscript expressions
- Condition may protect exception: when the compiler tries to vectorize a loop containing an if statement, it typically evaluates the RHS expressions for all values of the loop index, but only makes the final assignment in those cases where the conditional evaluates to TRUE. In some cases, the compiler may not vectorize because the condition may be protecting against accessing an illegal memory address. You can use the **`#pragma ivdep`** to reassure the compiler that the conditional is not protecting against a memory exception in such cases.
- Unsupported loop Structure: loops that do not fulfill the requirements of countability, single entry and exit, and so on, may generate these messages

https://software.intel.com/en-us/articles/vectorization-diagnostics-for-intelr-c-compiler-150-and-above

When vectorization fails

- Operator unsuited for vectorization: Certain operators, such as the % (modulus) operator, cannot be vectorized
- Non-unit stride used: non-contiguous memory access.
- Existence of vector dependence: vectorization entails changes in the order of operations within a loop, since each SIMD instruction operates on several data elements at once. Vectorization is only possible if this change of order does not change the results of the calculation

Vectorized loops? (intel compiler)

- Vectorization is enabled by the flag -vec and by default at -O2.

```
-vec-report[N] (deprecated)
-qopt-report[=N] -qopt-report-phase=vec
```

N	Diagnostic Messages
0	No diagnostic messages; same as not using switch and thus default
1	Tells the vectorizer to report on vectorized loops.
2	Tells the vectorizer to report on vectorized and non-vectorized loops.
3	Tells the vectorizer to report on vectorized and non-vectorized loops and any proven or assumed data dependencies.
4	Tells the vectorizer to report on non-vectorized loops.
5	Tells the vectorizer to report on non-vectorized loops and the reason why they were not vectorized.
6	Tells the vectorizer to use greater detail when reporting on vectorized and non-vectorized loops and any proven or assumed data dependencies.
7	Tells the vectorizer to emit vector code quality message ids and corresponding data values for vectorized loops. It provides information such as the expected speedup, memory access patterns, and the number of vector idioms for vectorized loops.

Vectorized loops?

gnu compiler

- Vectorization is enabled by the flag -ftree-vectorize and by default at -O3.

```
-ftree-vectorizer-verbose=[N] (deprecated)
-fopt-info-vec
```

pgi compiler

- Vectorization is enabled by the flag -Mvec and by default at -fast or -fastsse .

```
-Minfo-vec
```

Vectorization Report (intel compiler):example

```
ifort -O3 -qopt-report=5
```

```
 LOOP BEGIN at matmat.F90(51,1)
      remark #25427: Loop Statements Reordered
      remark #15389: vectorization support: reference C has unaligned access
      remark #15389: vectorization support: reference B has unaligned access
[ matmat.F90(50,1) ]
      remark #15389: vectorization support: reference A has unaligned access
[ matmat.F90(49,1) ]
      remark #15381: vectorization support: unaligned access used inside loop body
[ matmat.F90(49,1) ]
      remark #15301: PERMUTED LOOP WAS VECTORIZED
      remark #15451: unmasked unaligned unit stride stores: 3
      remark #15475: --- begin vector loop cost summary ---
      remark #15476: scalar loop cost: 229
      remark #15477: vector loop cost: 43.750
      remark #15478: estimated potential speedup: 5.210
      remark #15479: lightweight vector operations: 24
      remark #15480: medium-overhead vector operations: 2
      remark #15481: heavy-overhead vector operations: 1
      remark #15482: vectorized math library calls: 2
      remark #15487: type converts: 2
      remark #15488: --- end vector loop cost summary ---
      remark #25015: Estimate of max trip count of loop=28
   LOOP END
```

When vectorization fails

- Programmers need to provide the necessary information
- Programmers need to transform the code

- Add compiler directives
- Transform the code
- Program using vector intrinsics

Example code

```
time1 = time();

for (i=0; i<32000; i++)
c[i] = a[i] + b[i];


time2 = time();
```

Example code

- Added an outer loop that runs (serially)
  - to increase the running time of the loop

```
time1 = time();
for (j=0; j<200000; j++){
for (i=0; i<32000; i++)
c[i] = a[i] + b[i];

}
time2 = time();
```

Example code

- Added an outer loop that runs (serially)
  - to increase the running time of the loop
- Call a dummy () function that is compiled separately
  - to avoid loop interchange or dead code elimination

```
time1 = time();
for (j=0; j<200000; j++){
for (i=0; i<32000; i++)
c[i] = a[i] + b[i];
dummy()
}
time2 = time();
```

Example code

- Added an outer loop that runs (serially)
  - to increase the running time of the loop
- Call a dummy () function that is compiled separately
  - to avoid loop interchange or dead code elimination
- Access the elements of one output array and print the result
  - to avoid dead code elimination

```
time1 = time();
for (j=0; j<200000; j++){
for (i=0; i<32000; i++)
c[i] = a[i] + b[i];
dummy()
}
time2 = time();
for (j=0; j<32000; j++)
ret+= a[i];
printf (" Time %f , result %f ", (time2-time1), ret) ;
```

Compiler directives

```
void test(float*                  A,
          float*                  B,
          float*                  C,
          float*                  D,
          float*                  E)
{
  for  (int i = 0; i <LEN; i++){
  A[i]=B[i]+C[i]+D[i]+E[i];
  }
}
```

Compiler directives

```
void test(float* __restrict__     A,
          float* __restrict__     B,
          float* __restrict__     C,
          float* __restrict__     D,
          float* __restrict__     E)
{
  for (int i = 0; i <LEN; i++){
  A[i]=B[i]+C[i]+D[i]+E[i];
  }
}
```

Compiler directives

S1111

```
void test(float* __restrict__ A,
          float* __restrict__ B,
          float* __restrict__ C,
          float* __restrict__ D,
          float* __restrict__ E)
{
  for (int i = 0; i <LEN; i++){
  A[i]=B[i]+C[i]+D[i]+E[i];
  }
}
```

scalar	2.41	2.41	2.41	47.97	17.06
vectorized	1.36	1.41	1.33	30.51	3.46
speedup	1.77	1.71	1.81	1.57	4.93

Loop Transformations

S136

```
for (int i = 0; i < LEN2; i++){
  float sum = (float)0.0;
  for (int j = 0; j < LEN2; j++){
    sum += aa[j][i];
  }
  e[i] = sum;
}
```

scalar	2.50	2.61	2.94	43.62	17.35
vectorized	2.74	0.66	2.15	129.34	6.92
speedup	0.91	3.95	1.37	0.33	2.51

Loop Transformations

S136_1

```
for (int i = 0; i < LEN2; i++){
  sum[i] = (float)0.0;
  for (int j = 0; j < LEN2; j++){
    sum[i] += aa[j][i];
  }
  e[i] = sum[i];
}
```

scalar	2.65	2.61	3.07	43.72	17.24
vectorized	2.76	0.65	0.27	129.88	6.98
speedup	0.96	4.01	11.37	0.33	2.47

Loop Transformations

S136_2

```
for (int i = 0; i < LEN2; i++)
  e[i] = (float)0.0;
for (int j = 0; j < LEN2; j++){
  for (int i = 0; i < LEN2; i++){
    e[i] += aa[j][i];
  }
}
```

scalar	1.01	1.00	0.98	21.93	9.94
vectorized	0.29	0.37	0.27	2.66	0.96
speedup	3.48	2.70	3.63	8.24	10.35

Intrinsics (SSE)

```
#define n 1024
__attribute__ ((aligned(16))) float a[n], b[n], c[n];
int main() {
for (i = 0; i < n; i++) {
c[i]=a[i]*b[i];
}
}
```

⇓

```
#include <xmmintrin.h>
#define n 1024
__attribute__((aligned(16))) float a[n], b[n], c[n];
int main() {
__m128 rA, rB, rC;
for (i = 0; i < n; i+=4) {
rA = _mm_load_ps(&a[i]);
rB = _mm_load_ps(&b[i]);
rC= _mm_mul_ps(rA,rB);
_mm_store_ps(&c[i], rC);
}}
```

Outline

Topics

Introduction

Data Dependencies

Overcoming limitations to SIMD-Vectorization

Vectorization

Data Dependencies

- The notion of dependence is the foundation of the process of vectorization.
- It is used to build a calculus of program transformations that can be applied manually by the programmer or automatically by a compiler

Definition of Dependencies

- Statement S is said to be data dependent on statement T if
    - T executes before S in the original sequential/scalar program
    - S and T access the same data item
    - At least one of the accesses is a write

Data Dependencies

- Read after write: When a variable is written in one iteration and read in a subsequent iteration, also known as a flow dependency:

```
A[0]=0;
for (j=1; j<MAX; j++)
A[j]=A[j-1]+1;
// this is equivalent to:
A[1]=A[0]+1; A[2]=A[1]+1; A[3]=A[2]+1; A[4]=A[3]+1;
```

- The above loop cannot be vectorized safely because if the first two iterations are executed simultaneously by a SIMD instruction, the value of A[1] may be used by the second iteration before it has been calculated by the first iteration which could lead to incorrect results.

Data Dependencies

- Write-after-read: When a variable is read in one iteration and written in a subsequent iteration, sometimes also known as an anti-dependency

```
for (j=1; j<MAX; j++)
A[j-1]=A[j]+1;
// this is equivalent to:
A[0]=A[1]+1; A[1]=A[2]+1; A[2]=A[3]+1; A[3]=A[4]+1;
```

- This is not safe for general parallel execution, since the iteration with the write may execute before the iteration with the read. However, for vectorization, no iteration with a higher value of j can complete before an iteration with a lower value of j, and so vectorization is safe (i.e., gives the same result as non- vectorized code) in this case.

Data Dependencies

- Read-after-read: These situations aren’t really dependencies, and do not prevent vectorization or parallel execution. If a variable is not written, it does not matter how often it is read.
- Write-after-write: Otherwise known as ’output’ dependencies, where the same variable is written to in more than one iteration, are in general unsafe for parallel execution, including vectorization.

Data Dependencies

- Dependencies indicate an execution order that must be honored.
- Executing statements in the order of the dependencies guarantee correct results.
- Statements not dependent on each other can be reordered, executed in parallel, or coalesced into a vector operation.

Data Dependencies and vectorization

- A statement inside a loop which is not in a cycle of the dependence graph can be vectorized
- When cycles are present, vectorization can be achieved by:
  - Separating (distributing) the statements not in a cycle
  - Removing dependencies
  - Freezing loops
  - Changing the algorithm

Distributing

```
for (i=1; i<n; i++){
b[i] = b[i] + c[i];
a[i] = a[i-1]*a[i-2]+b[i];
c[i] = a[i] + 1;
}
```

```
b[1:n-1] = b[1:n-1] + c[1:n-1];
for (i=1; i<n; i++){
a[i] = a[i-1]*a[i-2]+b[i];
}
c[1:n-1] = a[1:n-1] + 1;
```

Removing dependencies

```
for (i=0; i<n; i++){
a = b[i] + 1;
c[i] = a + 2;
```

```
for (i=0; i<n; i++){
a’[i] = b[i] + 1;
c[i] = a’[i] + 2;
}
a=a’[n-1]
```

```
a’[0:n-1] = b[0:n-1] + 1;
c[0:n-1] = a’[0:n-1] + 2;
a=a’[n-1]
```

Freezing Loops

```
for (i=1; i<n; i++) {
for (j=1; j<n; j++) {
a[i][j]=a[i][j]+a[i-1][j];
}
}
```

```
for (i=1; i<n; i++) {
a[i][1:n-1]=a[i][1:n-1]+a[i-1][1:n-1];
}
```

Changing the algorithm

- When there is a recurrence, it is necessary to change the algorithm in order to vectorize.
- Compiler use pattern matching to identify the recurrence and then replace it with a parallel version.
- Examples or recurrences include:
  - Reductions (S+=A[i])
  - Linear recurrences (A[i]=B[i]*A[i-1]+C[i])
  - Boolean recurrences (if (A[i]>max) max = A[i])

Changing the algorithm

```
a[0]=b[0];
for (i=1; i<n; i++)
a[i]=a[i-1]+b[i];
```

```
a[0:n-1]=b[0:n-1];
for (i=0;i<k;i++) /* n = 2 k */
a[2**i:n-1]=a[2**i:n-1]+b[0:n-2**i];
```

Changing the algorithm

- Different algorithm for the same problem could be vectorazable or not
  - Gauss-Seidel: data dependencies, can not be vectorized

```
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
    a[i][j] = w0 * a[i][j] +
      w1*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
```

  - Jacobi: no data dependence, can be vectorized

```
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
     b[i][j] = w0*a[i][j] +
       w1*(a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]);
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
     a[i][j] = b[i][j];
```

Stripmining

- Stripmining is a simple transformation

```
for (i=1; i<n; i++){
...
}
```

```
/* n is a multiple of q */
for (k=1; k<n; k+=q){
for (i=k; i<k+q-1; i++){
...
}
}
```

- It is typically used to improve locality.

Stripmining

```
for (i=1; i<n; i++){
a[i] = b[i] + 1;
c[i] = a[i] + 2;
}
```

Striminine

```
for (k=1; k<n; k+=q){
/* q could be size of vector register */
for (i=k; i < k+q; i++){
a[i] = b[i] + 1;
c[i] = a[i-1] + 2;
}
}
```

Vectorize

```
for (i=1; i<n; i+=q){
a[i:i+q-1] = b[i:i+q-1] + 1;
c[i:i+q-1] = a[i:i+q] + 2;
}
```

Outline

Topics

Introduction

Data Dependencies

Overcoming limitations to SIMD-Vectorization

- Data Dependencies
- Data Alignment
- Aliasing
- Non-unit strides
- Conditional Statements

Vectorization

Loop Vectorization

- Loop Vectorization is not always a legal and profitable transformation.
- Compiler needs:
  - The compiler figures out dependencies by
    - Compute the dependencies
    - Solving a system of (integer) equations (with constraints)
    - Demonstrating that there is no solution to the system of equations
  - Remove cycles in the dependence graph
  - Determine data alignment
  - Vectorization is profitable

Outline

Topics

Introduction

Data Dependencies

Overcoming limitations to SIMD-Vectorization

- Data Dependencies
- Data Alignment
- Aliasing
- Non-unit strides
- Conditional Statements

Vectorization

Dependence Graphs

- Acyclic Dependence Graphs (ADG):
  - All dependencies are forward:
    - Vectorized by the compiler
  - Some backward dependencies:
    - Sometimes vectorized by the compiler
- Cycles in the Dependence Graph (CDG)
  - Self-antidependence:
    - Vectorized by the compiler
  - Recurrence:
    - Usually vectorized by the compiler
  - Other examples

ADG: Forward Dependencies

S113

```
for (i=0; i<LEN; i++) {
a[i]= b[i] + c[i]
d[i] = a[i] + (float) 1.0;
}
```

scalar	6.61	6.59	6.61	106.33	44.92
vectorized	3.52	4.90	4.56	14.77	10.34
speedup	1.88	1.34	1.45	7.20	4.34

ADG: Backward Dependencies reordering

S114

```
for (i=0; i<LEN; i++) {
a[i]= b[i] + c[i];
d[i] = a[i+1]+(float)1.0;
}
```

S114_1

```
for (i=0; i<LEN; i++) {
d[i] = a[i+1]+(float)1.0;
a[i]= b[i] + c[i];
}
```

S114

scalar	6.55	6.66	6.64	111.3	46.44
vectorized	4.01	...	...	14.88	10.04
speedup	1.63	...	...	7.48	4.62

S114_1

scalar	6.55	6.63	6.63	111.50	46.67
vectorized	4.01	4.21	4.28	14.88	10.04
speedup	1.63	1.57	1.55	7.49	4.65

ADG: Backward Dependencies reordering II

S214

```
for (int i=1;i<LEN;i++) {
a[i]=d[i-1]+(float)sqrt(c[i]);
d[i]=b[i]+(float)sqrt(e[i]);
}
```

S214_1

```
for (int i=1;i<LEN;i++) {
d[i]=b[i]+(float)sqrt(e[i]);
a[i]=d[i-1]+(float)sqrt(c[i]);
}
```

S214

scalar	1.42	2.61	2.83	16.80	12.13
vectorized	0.51	...	...	1.40	1.03
speedup	2.78	...	...	12.0	11.78

S214_1

scalar	1.43	2.61	2.83	16.82	11.93
vectorized	0.51	...	...	1.40	1.05
speedup	2.80	...	...	12.0	11.32

ADG: I

S115

```
for (int i=0;i<LEN-1;i++){
b[i] = a[i] + (float) 1.0;
a[i+1] = b[i] + (float) 2.0;
}
```

scalar	12.04	12.04	12.79	68.13	116.56
vectorized	...	...	...	...	...
speedup	...	...	...	...	...

ADG: II

S116

```
for (int i=1;i<LEN;i++){
a[i] = b[i] + c[i];
d[i] = a[i] + e[i−1];
e[i] = d[i] + c[i];
}
```

scalar vectorized speedup	12.05	12.05	13.57	197.88	122.09 ... ...

ADG: III 1

S117

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

scalar	3.05	2.87	2.92	62.72	29.67
vectorized	1.26	1.43	1.29	5.72	3.87
speedup	2.42	2.01	2.26	10.98	7.67

ADG: III 2

S118

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

scalar vectorized speedup	6.02	6.03	6.77	57.60	59.06
	...	...	...	...	...
	...	...	...	...	...

ADG: IV

S119

```
for (int i=4;i<LEN;i++){
a[i]=a[i-4]+b[i];
}
```

scalar	3.21	4.54	2.74	68.04	31.70
vectorized	2.25	1.54	...	28.34	20.98
speedup	1.41	2.95	...	2.91	1.51

ADG: V

S121

```
for (int i = 0; i < LEN-1; i++) {
for (int j = 0; j < LEN; j++)
a[i+1][j] = a[i][j] + 1;
}
```

scalar	5.09	7.82	4.66	81.62	103.33
vectorized	2.13	2.24	2.24	18.65	14.28
speedup	2.39	3.49	2.08	4.38	7.23

ADG: VI 1

S122

```
for (int i=0;i<LEN;i++){
a[r[i]]=a[r[i]]*(float)2.0;
}
```

scalar	2.77	3.10	2.65	56.38	28.11
vectorized	...	...	...	...	15.79
speedup	...	...	...	...	1.78

ADG: VI 2

S123

```
for (int i=0;i<LEN;i++){
r[i] = i;
a[r[i]]=a[r[i]]*(float)2.0;
}
```

scalar	3.36	3.49	3.28	58.77	26.10
vectorized	...	1.16	...	...	15.01
speedup	...	3.01	...	...	1.73

Loop Transformations

- Compiler Directives
- Loop Distribution or loop fission
- Node Splitting
- Scalar expansion
- Loop Peeling
- Loop Fusion
- Loop Unrolling
- Loop Interchanging

Compiler Directives I

- When the compiler does not vectorize automatically due to dependencies the programmer can inform the compiler that it is safe to vectorize
- **`#pragma ivdep`**:this tells the compiler to ignore vector dependencies in the loop that immediately follows the directive/pragma. However, this is just a recommendataion, and the compiler will not vectorize the loop if there is a clear dependency.
- Use **`#pragma ivdep`** only when you know that the assumed loop dependencies are safe to ignore.

Compiler Directives I

S124_1

S124

```
for (int i=0;i<LEN-k;i++)
a[i]=a[i+k]+b[i];
```

```
if (k>=0)
for (int i=0;i<LEN-k;i++)
a[i]=a[i+k]+b[i];
if (k<0)
for (int i=0);i<LEN-k;i++)
a[i]=a[i+k]+b[i];
```

S124

scalar	3.04	3.75	2.75	85.73	30.16
vectorized	...	...	...	...	..
speedup	...	...	...	...	...

S124_1

scalar	3.03	4.06	2.74	85.73	29.71
vectorized	...	3.74	...	...	...
speedup	...	1.08	...	...	...

Compiler Directives I

S124_2

```
if (k>=0)
#pragma ivdep
for (int i=0;i<LEN-k;i++)
a[i]=a[i+k]+b[i];
if (k<0)
for (int i=0);i<LEN-k;i++)
a[i]=a[i+k]+b[i];
```

scalar	2.80	3.74	2.75	84.81	26.82
vectorized	1.41	...	1.29	7.39	4.12
speedup	1.98	...	2.13	11.48	6.51

Compiler Directives II

- **`#pragma vector`**: This overrides default heuristics for vectorization of the loop. You can provide a clause for a specific task. For example, it will try to vectorize the immediately-following loop that the compiler normally would not vectorize because of a performance efficiency reason. As another example.
- **`#pragma novector`**: This tells the compiler to disable vectorizaton for the loop that follows
- You can use **`#pragma vector always`** to override any efficiency heuristics during the decision to vectorize or not, and to vectorize non-unit strides or unaligned memory accesses. The loop will be vectorized only if it is safe to do so. The outer loop of a nest of loops will not be vectorized, even if **`#pragma vector always`** is placed before it

Compiler Directives III

- **`#pragma simd`**: This is used to enforce vectorization for a loop that the compiler doesn’t auto-vectorize even with the use of vectorization hints such as **`#pragma vector always`** or **`#pragma ivdep`**. Because of this nature of enforcement, it is called user-mandated vectorization. A clause can be accompanied to give a more specific direction (see documentation).

#pragma ivdep versus #pragma simd

- #pragma ivdep
  - Implicit vectorization
  - Notifies the compiler about the absence of pointer aliasing
  - Based on practicability and costs, the compiler decides about vectorization
- #pragma simd
  - Explicit
  - Enforces vectorization rergardless of the costs
  - If no parameter is provided, the vector length of the SIMD unit is assumed

Loop Distribution

S216

```
for (int i = 0; i < LEN; i++) {
      a[i] = (float)sqrt(b[i]) + (float)sqrt(c[i]);
      s216_dummy(a,b,c);
    }
}
```

scalar	1.41	1.70	2.82	18.70	12.10
vectorized	...	...	...	...	...
speedup	...	...	...	...	...

Loop Distribution

S216_1

```
for (int i = 0; i < LEN; i++) {
      a[i] = (float)sqrt(b[i]) + (float)sqrt(c[i]);
    }
    for (int i = 0; i < LEN; i++) {
      s216_dummy(a,b,c);
    }
```

scalar	1.93	2.40	3.34	17.35	14.48
vectorized	0.74	...	...	3.76	4.58
speedup	2.61	...	...	4.61	3.16

Node Splitting

S126

```
for (int i=0;i<LEN-1;i++){
a[i]=b[i]+c[i];
d[i]=(a[i]+a[i+1])*(float)0.5;
}
```

S126_1

```
for (int i=0;i<LEN-1;i++){
e[i]=a[i+1];
a[i]=b[i]+c[i];
d[i]=(a[i]+e[i])*(float)0.5;
}
```

S126

scalar	10.46	6.81	6.66	199.22	106.34
vectorized	4.67	...	...	45.80	13.93
speedup	2.24	...	...	4.35	7.63

S126_1

scalar	10.10	10.08	8.90	214.53	79.86
vectorized	5.45	6.32	6.10	23.23	16.10
speedup	1.85	1.59	1.46	9.23	4.96

Scalar Expansion

S139

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

S139_1

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

S139

scalar	0.44	0.44	0.46	6.55	4.31
vectorized	0.18	0.19	0.19	0.57	0.49
speedup	2.44	2.31	2.42	11.49	8.96

S139_1

scalar	0.44	0.44	0.66	11.66	4.61
vectorized	0.18	0.19	0.39	1.24	0.49
speedup	2.44	2.31	1.69	9.40	9.41

Loop Peeling

- Remove the first/s or the last/s iteration of the loop into separate code outside the loop
- It is always legal, provided that no additional iterations are introduced.
- This transformation is useful to enforce a particular initial memory alignment on array references prior to loop vectorization

Loop Peeling

S127

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

S127_1

```
a[0]= a[0] + a[0];
for (int i=1;i<LEN;i++){
a[i] = a[i] + a[0]
}
```

S127

scalar	3.01	2.58	2.29	62.11	29.59
vectorized	...	...	...	...	...
speedup	...	...	....	...	...

S127_1

scalar	2.53	3.19	2.31	47.08	25.78
vectorized	1.00	...	...	4.33	2.72
speedup	2.53	...	...	10.87	9.48

Loop Interchanging

S228

```
for (j=1; j<LEN; j++){
 for (i=j; i<LEN; i++){
  A[i][j]=A[i-1][j]+(float)1.0;
}}
```

S228_1

```
for (i=j; i<LEN; i++){
 for (j=1; j<LEN; j++){
  A[i][j]=A[i-1][j]+(float)1.0;
}}
```

S228

scalar	2.03	2.05	2.17	14.74	8.85
vectorized	...	...	...	31.05	...
speedup	...	...	...	0.47	...

S228_1

scalar	0.23	0.48	0.25	2.84	1.87
vectorized	0.16	0.14	0.13	1.09	0.37
speedup	1.77	3.43	1.92	2.60	5.05

Reductions

S131

```
sum =0;
for (int i=0;i<LEN;++i){
sum+= a[i];
}
```

scalar	3.01	6.01	6.01	31.39	29.50
vectorized	0.55	1.50	0.76	2.20	0.99
speedup	5.47	4.00	7.91	14.27	29.3

Reductions

S132

```
x = a[0];
index = 0;
for (int i=0;i<LEN;++i){
if (a[i] > x)  {
x = a[i];
index = i;
}}
```

scalar	6.02	4.02	4.02	60.14	93.60
vectorized	2.01	...	...	4.98	2.17
speedup	2.99	...	...	12.08	44.57

Induction variables

- Induction variable is a variable that can be expressed as a function of the loop iteration variable

S133

```
float s = (float)0.0;
for (int i=0;i<LEN;i++){
s += (float)2.;
a[i] = s * b[i];
}
```

scalar	4.05	6.23	6.21	57.63	36.72
vectorized	1.56	1.64	...	5.11	3.55
speedup	2.60	3.80	...	11.28	10.34

Induction variables

S133_1

```
for (int i=0;i<LEN;i++){
a[i] = (float)2.*(i+1)*b[i];
}
```

scalar	4.73	5.35	4.09	94.21	49.21
vectorized	1.24	1.49	1.23	6.57	3.38
speedup	3.81	3.59	3.32	14.34	14.56

Induction variables

S134

```
for (int i=0;i<LEN;i++) {
*a = *b + *c;
a++; b++; c++;
}
```

S134_1

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

scalar	3.27	3.23	3.12	66.23	30.82
vectorized	2.11	2.13	2.15	19.00	4.15
speedup	1.55	1.52	1.45	3.48	7.43

Outline

Topics

Introduction

Data Dependencies

Overcoming limitations to SIMD-Vectorization

- Data Dependencies
- Data Alignment
- Aliasing
- Non-unit strides
- Conditional Statements

Vectorization

Data Alignment

- Vector loads/stores load/store 128 consecutive bits to a vector register.
- Data addresses need to be 16-byte (128 bits) aligned to be loaded/stored
- To know if a pointer is 16-byte aligned, the last digit of the pointer address in hex must be 0.
- Note that if &*b*[0] is 16-byte aligned, and is a single precision array, then &*b*[4] is also 16-byte aligned

```
__attribute__ ((aligned(16))) float B[1024];
int main(){
printf("%p, %p\n", &B[0], &B[4]);
}
```

```
Output:
0x7fff1e9d8580, 0x7fff1e9d8590
```

Data Alignment

- In many cases, the compiler cannot statically know the alignment of the address in a pointer
- The compiler assumes that the base address of the pointer is 16-byte aligned and adds a run-time checks for it
  - if the runtime check is false, then it uses another code (which may be scalar)

Data Alignment

- Manual 16-byte alignment can be achieved by forcing the base address to be a multiple of 16.

```
__attribute__ ((aligned(16))) float b[N];
float* a = (float*) memalign(16,N*sizeof(float));
```

- When the pointer is passed to a function, the compiler should be aware of where the 16-byte aligned address of the array starts.

```
void func1(float *a, float *b,
float *c) {
__assume_aligned(a, 16);
__assume_aligned(b, 16);
__assume_aligned(c, 16);
for int (i=0; i<LEN; i++) {
a[i] = b[i] + c[i];
}
```

Alignment in a struct

```
#include <stdio.h>
struct st{
char A;
int B[64]                               ;
float C;
int D[64]                               ;
};
int main(){
struct st s1;
printf("%p, %p, %p, %p\n", &s1.A, s1.B, &s1.C, s1.D);
}
```

Output:

```
0x7fff4bbeeb80, 0x7fff4bbeeb84, 0x7fff4bbeec84, 0x7fff4bbeec88
```

Alignment in a struct

```
#include <stdio.h>
struct st{
char A;
int B[64] __attribute__ ((aligned(16)));
float C;
int D[64] __attribute__ ((aligned(16)));
};
int main(){
struct st s1;
printf("%p, %p, %p, %p\n", &s1.A, s1.B, &s1.C, s1.D);
}
```

Output:

```
0x7fffa3644fb0, 0x7fffa3644fc0, 0x7fffa36450c0, 0x7fffa36450d0
```

Consistency of SIMD results

The alignment can effect reproducibility: because the order of the calculations can change

- Try to align to the SIMD register size
  - MMX: 8 Bytes;
  - SSE2: 16 bytes,
  - AVX: 32 bytes
  - MIC: 64 bytes
  - KNL: 64 bytes
- Try to align blocks of data to cacheline size - ie 64 bytes

Outline

Topics

Introduction

Data Dependencies

Overcoming limitations to SIMD-Vectorization

- Data Dependencies
- Data Alignment
- Aliasing
- Non-unit strides
- Conditional Statements

Vectorization

Aliasing

- Writing "clean" code is a good starting point to have the code vectorized
  - Prefer array indexing instead of explicit pointer arithmetic
  - Use restrict keyword to tell the compiler that there is no array aliasing
- The use of the restrict keyword in pointer declarations informs the compiler that it can assume that during the lifetime of the pointer only this single pointer has access to the data addressed by it that is, no other pointers or arrays will use the same data space. Normally, it is adequate to just restrict pointers associated with the left-hand side of any assignment statement. Without the restrict keyword, the code will not vectorize.

```
void f(int n, float *x, float *y, float *restrict z, float *d1, float *d2)
{
for (int i = 0; i < n; i++)
z[i] = x[i] + y[i]-(d1[i]*d2[i]);
}
```

Outline

Topics

Introduction

Data Dependencies

Overcoming limitations to SIMD-Vectorization

- Data Dependencies
- Data Alignment
- Aliasing
- Non-unit strides
- Conditional Statements

Vectorization

Non-unit Stride I

S135

```
typedef struct{int x, y, z}
point;
point pt[LEN];
for (int i=0; i<LEN; i++) {
pty[i] *= scale;
}
```

scalar	3.84	3.89	3.19	38.69	21.98
vectorized	3.66	...	...	41.06	23.26
speedup	1.04	...	...	0.94	0.94

Non-unit Stride I

S135_1

```
int ptx[LEN], int pty[LEN],
int ptz[LEN];
for (int i=0; i<LEN; i++) {
pty[i] *= scale;
}
```

scalar	2.41	2.51	2.51	36.62	19.75
vectorized	0.82	0.82	2.24	2.98	2.51
speedup	2.94	3.06	1.12	12.29	7.87

Non-unit Stride II

S136

```
for (int i = 0; i < LEN2; i++){
  float sum = (float)0.0;
  for (int j = 0; j < LEN2; j++){
    sum += aa[j][i];
  }
  e[i] = sum;
}
```

scalar	2.50	2.61	2.94	42.62	17.35
vectorized	2.74	0.66	2.15	129.34	6.92
speedup	0.91	3.95	1.37	0.33	2.51

Non-unit Stride II

S136_1

```
    for (int i = 0; i < LEN2; i++){
      sum[i] = (float)0.0;
      for (int j = 0; j < LEN2; j++){
        sum[i] += aa[j][i];
      }
      e[i] = sum[i];
    }
```

scalar	2.05	2.61	3.07	43.72	17.24
vectorized	2.76	0.65	0.27	129.88	6.98
speedup	0.96	4.01	11.37	0.34	2.47

Non-unit Stride II

S136_2

```
  for (int i = 0; i < LEN2; i++)
  e[i] = (float)0.0;
for (int j = 0; j < LEN2; j++){
  for (int i = 0; i < LEN2; i++){
    e[i] += aa[j][i];
  }
}
```

scalar	1.01	1.00	0.98	21.93	9.94
vectorized	0.29	0.37	0.27	2.66	0.90
speedup	3.48	2.70	3.63	8.24	11.04

Outline

Topics

Introduction

Data Dependencies

Overcoming limitations to SIMD-Vectorization

- Data Dependencies
- Data Alignment
- Aliasing
- Non-unit strides
- Conditional Statements

Vectorization

Conditional Statements

S137

```
    for (int i = 0; i < LEN; i++){
      if (C[i] > (float) -1.0)
        A[i] = A[i] * B[i] + D[i];
    }
```

S137_1

```
#pragma vector always
for (int i = 0; i < LEN; i++){
      if (C[i] > (float) -1.0)
        A[i] = A[i] * B[i] + D[i];
    }
```

S137

scalar	5.30	5.43	5.39	163.20	55.71
vectorized	...	...	2.84	22.97	9.41
speedup	...	...	1.89	7.10	5.92

S137_1

scalar	5.30	5.41	5.38	163.36	56.17
vectorized	2.99	...	2.84	14.87	9.16
speedup	1.77	...	1.89	10.98	6.13

Intrinsic

- Intrinsics are vendor/architecture specific
- Intrinsics are useful when
  - the compiler fails to vectorize
  - when the programmer thinks it is possible to generate better code than the one produced by the compiler

Splitting with intrinsic

S126_2

```
#include <xmmintrin.h>
#define n 1000
int main() {
__m128 rA1, rA2, rB, rC, rD;
__m128 r5=_mm_set1_ps((float)0.5)
for (i = 0; i < LEN-4; i+=4) {
rA2= _mm_loadu_ps(&a[i+1]);
rB= _mm_load_ps(&b[i]);
rC= _mm_load_ps(&c[i]);
rA1= _mm_add_ps(rB, rC);
rD= _mm_mul_ps(_mm_add_ps(rA1,rA2),r5);
_mm_store_ps(&a[i], rA1);
_mm_store_ps(&d[i], rD); }}
```

intrinsic	4.33	4.23	3.67	22.64	66.34
speedup	1.08	1.60	1.66	0.61	0.69

Vectorization: array notation

- Using array notation is a good way to guarantee the compiler that the iterations are independent
  - In Fortran this is consistent with the language array syntax
    a(1:N) = b(1:N) + c(1:N)
  - In C the array notation is provided by Intel Cilk Plus
    a[1:N] = b[1:N] + c[1:N]
- Beware:
  - The first value represents the lower bound for both languages
  - But the second value is the upper bound in Fortran whereas it is the length in C
  - An optional third value is the stride both in Fortran and in C
  - Multidimensional arrays supported, too

Intel Advisor

- Compile
  - icc -g -std=c99 -O3 -qopt-report=5 -xmic-avx512 dummy.o tsc.c -o runvec
- Run
  - advixe-cl -collect survey [–project-dir nomedir] ./runvec
- Visualize
  - advixe-gui e000/e000.advixeexp

Outline

Topics

Introduction

Data Dependencies

Overcoming limitations to SIMD-Vectorization

Vectorization

How to Succeed in Vectorization? I

- Most frequent reason of failing vectorization is Dependence:
  - Minimize dependencies among iterations by design!
- Alignment: Align your arrays/data structures
- Function calls in loop body: Use aggressive in-lining (IPO)
- Complex control flow/conditional branches:
  - Avoid them in loops by creating multiple versions of loops
- Unsupported loop structure: Use loop invariant expressions
- Not inner loop:
  - Manual loop interchange possible? for example Intel Compilers 12.1 and higher can do
  - outer loop vectorization now as well!
- Mixed data types:
  - Avoid type conversions in rare cases Intel Compiler cannot do automatically

How to Succeed in Vectorization? II

- Non-unit stride between elements:
  - Possible to change algorithm to allow linear/consecutive access?
- Loop body too complex reports: Try splitting up the loops!
- Vectorization seems inefficient reports:
  - Enforce vectorization, benchmark and verify results!

Vectorization:conclusions

- Microprocessor vector extensions can contribute to improve program performance and the amount of this contribution is likely to increase in the future as vector lengths grow.
- Compilers are only partially successful at vectorizing
- When the compiler fails, programmers can
  - add compiler directives
  - apply loop transformations
- If after transforming the code, the compiler still fails to vectorize (or the performance of the generated code is poor), the only option is to program the vector extensions directly using intrinsics or assembly language.