Lecture 13: Vectors

William Gropp
www.cs.illinois.edu/~wgropp

Overview

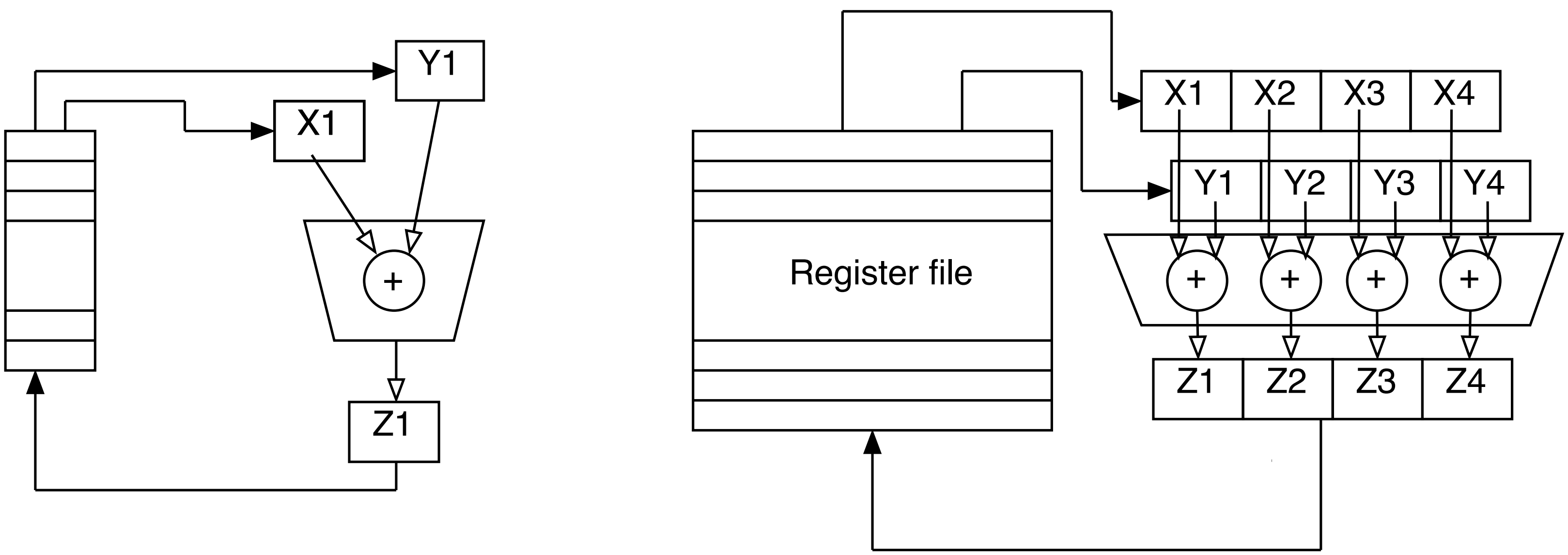
- Parallelism with the processor
  - Add vectors to the architecture
- Simple performance models
  - Add vector operations
- Challenges with vectorization
  - Dependence and alignment
- This is a very basic introduction
  - An entire course can (and has!) been taught on program optimization through vectorization

Parallelism Within the Processor

- The clock speed of an individual processing core has not increased much since 2006
- How to get more performance from the same chip to keep up with expectations (Moore's "Law")?
- Note that in modern processors, the floating point units are physically small relative to cache, instruction processing
- To get more performance, need more operations at once, but without using more instructions
  - ♦ Solution "vectors"

Scalar and Vector Architecture

- Vectors operate on 128 bit (16 byte) operands
  - 4 floats or ints
  - 2 doubles
- Data paths 128 bits vide for vector unit



PARALLEL@ILLINOIS

Example Code

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

(ignoring address and loop calculations:)

- Scalar Code:
- Repeat n times:
  - ld r1, addr1
    ld r2, addr2
    fadd r3, r1, r2
    st r3, addr2

- Vector Code:
- Repeat n/4 times:
  - vld vr1, addr1
    vld vr2, addr2
    vfadd vr3, vr1, vr2
    st vr3, addr2

Vector Performance

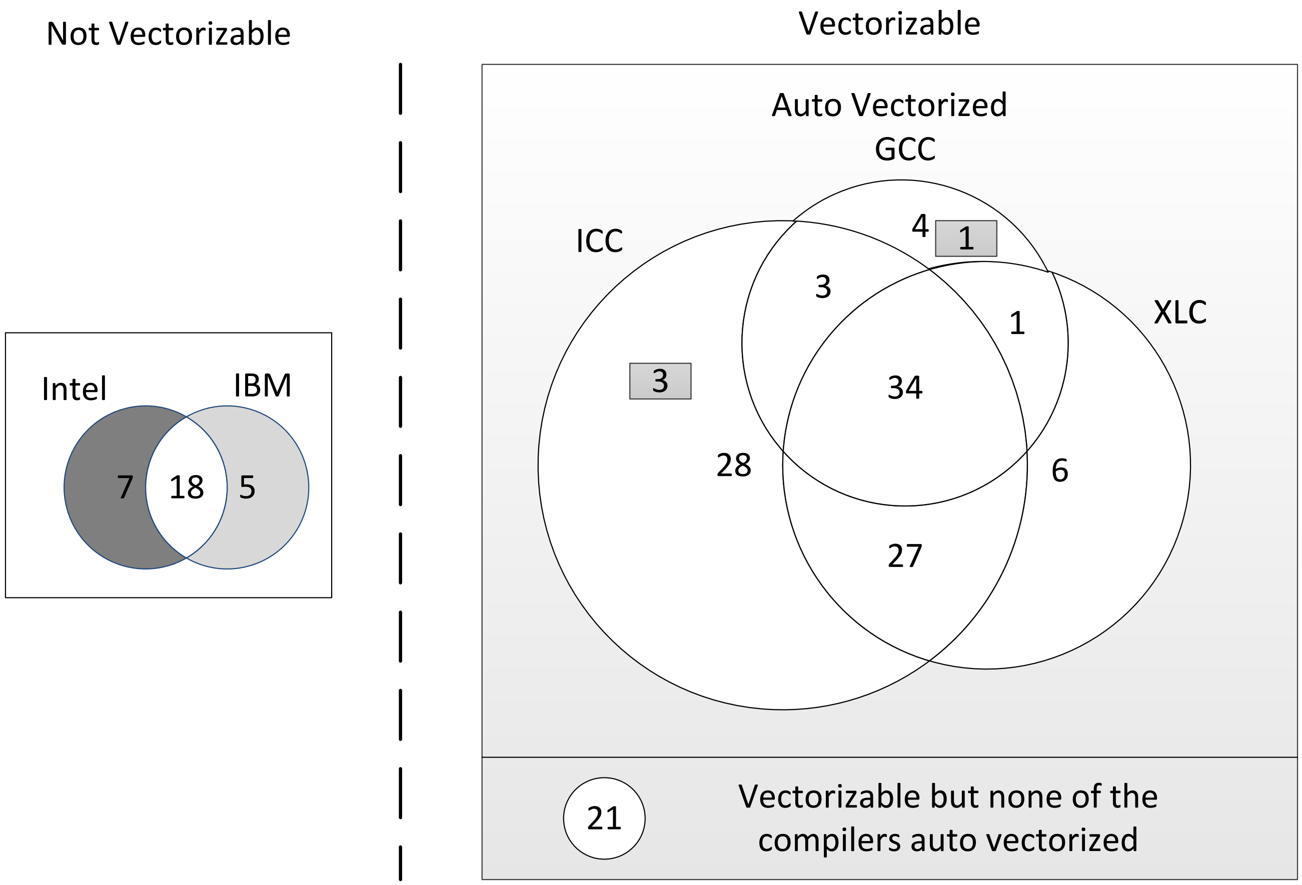
System	Scalar (sec)	Vector (sec)	Ratio
Macbook/gcc	2.86	1.57	1.82
Blue Waters/ craycc	6.73	3.09	2.18

- a, b, c floats (4 bytes)
- n = 32000
- Question: how much cache memory does that require? What level of cache might hold that data?
- Data from tsc.c program (more later)

Using Vector Operations

- Vectorizing compiler
  - C or Fortran code
  - May improve with manual code structuring and special directives
- Intrinsics
  - Some systems provide ways to program vector operations directly, within the C or (sometimes) Fortran program
- Assembly language
  - Fallback of last resort, but can provide extra performance

How Good are Compilers at Vectorizing Codes?



S. Maleki, Y. Gao, T. Wong, M. Garzarán, and D. Padua. *An Evaluation of Vectorizing Compilers*. PACT 2011.

Media Bench II Applications

Appl	XLC	ICC	GCC	XLC	ICC	GCC
	Automatic			Manual		
JPEG Enc	-	1.33	-	1.39	2.13	1.57
JEPG Dec	-	-	-	-	1.14	1.13
H263 Enc	-	-	-	1.25	2.28	2.06
H263 Dec	-	-	-	1.31	1.45	-
MPEG2 Enc	-	-	-	1.06	1.96	2.43
MPEG2 Dec	-	-	1.15	1.37	1.45	1.55
MPEG4 Enc	-	-	-	1.44	1.81	1.74
MPEG4 Dec	-	-	-	1.12	-	1.18

Table shows **whole program speedups** measured against unvectorized application

S. Maleki, Y. Gao, T. Wong, M. Garzarán, and D. Padua. *An Evaluation of Vectorizing Compilers*. PACT 2011.

Vectorizing compiler: How do you know if you have succeeded?

- Compiler reports
- Performance compared to non-vector (scalar) code
  - ♦ Easiest: include # of vector operations in model, with their own rates (e.g., c_v).
  - ♦ Simplest assumption: 1 vector op per cycle (e.g., 4 float ops/cycle)
  - ♦ Note that vector ops also pipelined
    - Often not visible because only used in loops

Sample Compiler Report

- Craycc "loopmark" output

```
375.  + 1----------<   for (int nl = 0; nl < 2*ntimes; nl++) {
376.    1 Vr4------<     for (int i = 0; i < LEN; i++) {
377.    1 Vr4              c[i] = a[i] * b[i];
378.    1 Vr4------>     }
379.  + 1                dummy(a, b, c, d, e, aa, bb, cc, 0.);
380.    1---------->   }
```

- Annotations mean:
  - ♦ V = vectorized
  - ♦ R4 = unrolled by 4 (e.g., 4 floats at a time in the vector instruction)

1867

PARALLEL@ILLINOIS

Memory is Still an Issue

- Note that we didn't have enough bandwidth for DAXPY using one floating point unit
- Vector operations more valuable when data is in cache

A Very Basic Performance Model

- for (i=0; i<n; i++)
  c[i] = a[i]+b[i];
- Very simple (and not very good) assumptions:
  - $(1/4)c=c_v$; $r=w=r_v=w_v$

Model	Scalar	Vector	Ratio
Floating point only	$T_s=nc$	$T_v=nc_v=(n/4)c$	$T_s/T_v = 4$
With memory	$T_s=nc+3r$	$T_v=(n/4)c+3r$	$T_s/T_v=16/13 = 1.2$

Aliasing and Vectorization

- Each vector load makes a *copy* of 16 bytes of memory.
- The compiler must know whether the operations in one iteration of the loop will change data that it has already copied (thus invalidating the copy)
  - ♦ Just as for pipelining computations
- Follows is one example, from matrix-matrix multiplication

Version 1

```
• int s111(float** M1, float** M2, float** M3)
• ...
• for (int nl = 0; nl < ntimes/(10*LEN2); nl++) {
•     for (int i = 0; i < LEN2; i++) {
•       for (int j = 0; j < LEN2; j++) {
•         M3[i][j] = (float)0.;
•         for (int k = 0; k < LEN2; k++) {
•           M3[i][j] += M1[i][k]*M2[k][j];
•         }
•       }
•     }
•     dummy(a, b, c, d, e, aa, bb, cc, 0.);
•   }
```

1867

PARALLEL@ILLINOIS

Version 2

```
int s111_1(float** __restrict__ M1, float** __restrict__ M2, float** __restrict__ M3)
…
 for (int nl = 0; nl < ntimes/(10*LEN2); nl++) {
    for (int i = 0; i < LEN2; i++) {
      for (int j = 0; j < LEN2; j++) {
        M3[i][j] = (float)0.;
        for (int k = 0; k < LEN2; k++) {
          M3[i][j] += M1[i][k]*M2[k][j];
        }
      }
    }
    dummy(a, b, c, d, e, aa, bb, cc, 0.);
  }
```

Version 2: What’s Different

- int s111_1(float** **__restrict__** M1, float** **__restrict__** M2, float** **__restrict__** M3)
- …
- for (int nl = 0; nl < ntimes/(10*LEN2); nl++) {
-     for (int i = 0; i < LEN2; i++) {
-       for (int j = 0; j < LEN2; j++) {
-         M3[i][j] = (float)0.;
-         for (int k = 0; k < LEN2; k++) {
-           M3[i][j] += M1[i][k]*M2[k][j];
-         }
-       }
-     }
-     dummy(a, b, c, d, e, aa, bb, cc, 0.);
-   }

Some Results

Version	MacBook/gcc	Blue Waters/ Craycc
Without __restrict__	0.87	9.09
With __restrict__	0.85	0.3

- __restrict__ a common but non-standard attribute
- Standard C has restrict, but until recently many compilers provided only __restrict__
- A restricted pointer references data that is not referenced through any other pointer
  - Essentially gives C routines the same semantics that Fortran guarantees
- Whether the compiler *needs* or *exploits* restrict depends on many factors

Understanding These Results

- Consider again
  for (i=0; i<n; i++) c[i]=a[i]+b[i];
- Vector operations will load blocks of 4 elements, e.g., (a[0],a[1],a[2],a[3]) into a vector register
- Vector operation then does
- c[0:3] = a[0:3] + b[0:3];
  c[4:7] = a[4:7] + b[4:7];
  …
- What can go wrong?

Understanding These Results

- What if c and a point to overlapping memory?
  - ♦ For example c = a+1 (&c[0] = &a[1])?
- Then the loop really is:
  - ♦ c[0] = a[0] + b[0]; // c[0] is a[1],so
    c[1] = c[0] + b[1]; // c[1] is a[2],so
    c[2] = c[1] + b[2];
    …
  - ♦ A very different result than when a, b, and c do not overlap

Understanding These Results

- Compiler must either be told (e.g., with restrict), check at runtime ($n^2/2$ checks for n pointers), or guess (dangerous, potentially incorrect optimizations) whether pointers are aliased
- Good practice in C/C++: use restrict everywhere you might want vectorization and aliasing is not possible/permitted
  - Craycc on Blue Waters gave 4x improvement in one scalar case simply by adding restrict

Aside: Handling Compiler Features

- You can use open source build tools to check for different capabilities of a compiler
- *Never* assume a particular system/compiler has a feature
  - Systems and compilers change with time
- One common tool is autoconf
  - A "configure" script runs commands to test for features
- AC_C_RESTRICT
  - Tests for "restrict"
  - If *not* present, adds
    - #define restrict
  - to a header file (or compiler options)

Data Dependencies and Vectorization

- Data dependencies can arise in many ways
  - We've see one potential dependence – aliasing in the variables
  - Code may also have explicit data dependencies
  - Some prohibit or reduce vectorization, others are benign
- See the tutorial by Garzaran et al, "Program Optimization Through Loop Vectorization"; material on dependence here drawn from that tutorial

Definition of Dependence

- A statement S is data dependent on T if
  - T executes before S
  - S and T access the same data item
  - A least one access is a write

Some Types of Dependencies

- Flow dependence (also *true* dependence)
  - X = A + B
C = X + A
- Anti dependence
  - A = X + B
X = C + D
- Output dependence
  - X = A+B
X = C+D

Importance of Data Dependence

- Data dependencies can require a order for the execution of statements
- Statements that are not dependent can be reordered, executed in parallel, or combined into a vector operation
- Statements that are dependent must be handled carefully by the compiler (and the user!)

Example of True Dependence

- To clearly see dependencies, often easiest to unroll the loop.
- for (i=0; i<n; i++) {
  a[i] = b[i]+1; //S1
  c[i] = a[i]+2;} //S2
- a[0]=b[0]+1;
  c[0]=a[0]+2;
  a[1]=b[1]+1;
  c[1]=a[1]+2;
  …

Example of True Dependence

- To clearly see dependencies, often easiest to unroll the loop.
- for (i=1; i<n; i++) {
  a[i] = b[i]+1;   //S1
  c[i] = a[i-1]+2;} //S2
- a[1]=b[1]+1;
  c[1]=a[0]+2;
  a[2]=b[2]+1;
  c[2]=a[1]+2;
  …

This dependence of of distance 1

Dependencies and Vectorization

- A statement in a loop that is not in a cycle of the dependence graph can be vectorized
- This statement is *sufficient* but not *necessary*
  - It may be possible to transform the statement in a way that can be vectorized
- Recurrences are hard for the compiler to vectorize
  - May need to rewrite or use a different algorithm

Alignment: The Issue

- When data is moved between memory and the vector registers, most hardware is most efficient when the data is *aligned* on a 16 byte boundary.
  - This is common for other data types – doubles are on 8 byte boundaries and ints are on 4 (assuming sizeof(int) is 4)
- Unfortunately, neither C nor Fortran has a basic datatype corresponding to this type of vector
- Consider
  int myroutine (float *b, *c) {
  for (i=1; i<n; i++) b[i] = c[i+3];…
- Can the compiler use vector loads and stores?
- Maybe – depends on the hardware.
- Even if so, unaligned accesses may be slower – sometimes much slower (another reason to have a performance *expectation)*

Alignment: In Your Program

- There are non-standard ways to ask for and sometimes claim alignment:
  - ♦ float a[100] __attribute__((aligned(16)); //gcc-style
  - ♦ __alignx(16,a); // IBM altivec
- For dynamically allocated memory, either memalign or posix_memalign can be used
- Troublesome to handle in a perfectly portable way
  - ♦ Source-to-source transformations may be the best choice
  - ♦ Tools exist, such as Orio, to apply such transformations (and Orio applied to this very situation for IBM Blue Gene C compiler)

Questions for Discussion

- Find out how to request your compiler apply vectorization. For some systems, this is the default.
- Find out how (or if) you can get a report from the compiler about its success at vectorization.
- Read your compiler's documentation to find out what special directives or command line options can affect vectorization

Questions for Discussion

- Write a small C program to see if adding restrict helps or hurts performance. Use two files: one for the main program that calls the tests, and a separate for routines that perform the operation you are testing.
- Does it matter how many arrays are used? For example, is the behavior of
  - g[i]=a[i]+b[i]+c[i]+d[i]+e[i]+f[i]

  different from
  - c[i] = a[i] + b[i]

PARALLEL@ILLINOIS