Vectorization

Eric J. Kostelich

ARIZONA STATE UNIVERSITY
SCHOOL OF **MATHEMATICAL & STATISTICAL SCIENCES**

Aug. 22, 2018

Parallelism is subject to Amdahl's Law

- Let p be the proportion of the program's cycles that are spent in parallelizable code
- The maximum possible speedup on n cpus is

$$\frac{1}{(1-p)+(p/n)}$$

- Example: if $p = 1/2$, then the maximum speedup is 2, no matter how many processors are used!
- We say that a code scales linearly if the speedup is a multiple of n with n processors

Overview of vectorization

- Vectorization is the simplest form of parallelism
- It is a form of SIMD (Single Instruction Multiple Data) programming
- Most scientific code is "loopy," such as SAXPY:

```
do j=1, n
    y(j) = y(j) + a*x(j)
enddo
```

- The result of each iteration is independent of the others

Innermost loops are the object of vectorization

- Vectorization performs loop iterations "at the same time"
- The compiler generates code to process several operands at a time, depending on the processor
- Examples: Intel SSE (Streaming SIMD Extension): 8 128-bit registers (4 single-precision or 2 double-precision operands per instruction)
- x86 AVX-512: 32 registers of 512 bits each (up to 8 double-precision operands)
- Armv8-A SVE: vector registers from 128 to 2048 bits
- NEC SX-Aurora: 64 vector registers, each with 256 operands

The vectorization paradigm

- SIMD is conceptually the simplest form of parallelism
- The elementwise addition C = A + B can be performed for all elements “at the same time”
- Vectorization is especially important for good performance in MATLAB
- Programming philosophy: Use language constructs to concentrate on what to calculate, instead of how to calculate it

Vectorization provides a limited paradigm

- Useful for "loopy" code that performs the same set of operations on large blocks of data
- Example: SAXPY operations: `y(i) = y(i) + a * x(i)`
- Not useful for:
  - Overlapping computation with i/o
  - Using multiple cores on compute-bound codes
  - Keeping a GUI responsive by spawning a separate task for a compute-intensive requests

General requirements for vectorization

- The loop count must be fixed at the start
- The order of operations must not matter
- x and y must not overlap in memory
- Only simple conditionals are allowed

```
do j=1, n
   y(j) = max(y(j), x(j))
enddo
```

Rules of thumb

- If you can write the expression in MATLAB or Fortran without explicit subscripts, then it's probably vectorizable
- SAXPY is one example:

```
real:: a, x(n), y(n)
...
y = y + a*x
```

- Elementwise min and max is another

```
y = max(y, x)
```

Independent order of execution

- To be data parallel, the results of a computation must not depend on the order in which the operands are processed
- If you would get different results by running the loop backwards (i.e., $j = n, n-1, n-2, \ldots, 1$) then it is not vectorizable
- Example: Vector assignment is data parallel:

```
y(:) = x(:)
```

- The result is the same whether we start from the beginning, the end, or somewhere in the middle of `x`

Independent order of execution, 2

- Recurrence operations are not data parallel:

```
do j=2, n–1
    y(j+1) = y(j) + y(j–1)
enddo
```

- You'll get a different result if you process the loop in reverse order
- This loop is not vectorizable

Constructs that prevent vectorization

- Loops with subroutine or external function calls:

```
do j=1, n
    z(j) = f(x(j))
enddo
```

- Actual or implied goto:

```
do j=1, n
    y(j) = y(j) + a∗x(j)
    if(y(j) ≥ 100) exit
enddo
```

Some reduction operations are vectorizable

- Suppose $n = 2^k$. The sum of the elements of the n-vector x can be computed with k vector operations
- First compute `y(1:n/2) = x(1:n/2) + x(n/2+1:n)`
- Then compute `z(1:n/4) = y(1:n/4) + y(n/4+1:n/2)`, etc.
- Pad with zeroes as needed for other values of n

Other vectorizable constructs

- Certain Fortran intrinsic functions are vectorizable:

```
real:: y(n), x(n), a
...
y = a*sin(x)
```

- C/C++ functions generally are not vectorizable unless the compiler provides special options and math libraries
- Fortran provides for user-defined elemental functions that can be applied to a vector of arguments

Memory hierarchies

- Registers: (\$\$\$\$) access time 1 cycle; 16–256 registers in most modern cpu designs
- Cache (in 1 to 3 levels): (\$\$\$) access times from 5 to 40 cycles; typically 32K–512K of level-1 cache to several MB of level-3 cache
- Main memory: (\$\$) access times $\sim$ 250 cycles; typically 16GB–1028GB per machine in modern designs
- Virtual memory: (\$) uses disk; virtually unlimited in size but access times are millions of cycles

Memory hierarchies for the Intel i7 Core processor

RAM (4,096+ MB)
latency: ~250 cycles

L3 cache (6 MB)
latency: 40 cycles

L2 cache
1/4 MB, 12 cycles

L1 cache
1/32 MB, 3 cycles

Vectorization and memory

- To a reasonable first approximation, the time required for a modern CPU to perform an arithmetic operation is zero
- The time required to run a program is dominated by memory accesses
- Slow memory is cheap—fast memory is expensive
- Cache memory offers the illusion of one big fast memory—if you manage it carefully!

Cache memory

- Suppose x is a vector and you access x(k)
- The CPU can stall for ~ 250 cycles while it waits for main memory to deliver the contents
- On a modern CPU, the time required to complete an algorithm depends mainly on the time required for memory to fetch and store each datum
- The hardware copies x(k) to each cache—so subsequent references are much faster

Cache lines

- Cache management is predicated on the assumption that once the program accesses x(k), the next memory access will be x(k+1)—mostly true in practice
- When x(k) is first accessed, the hardware actually fetches $\{$x(k), x(k+1), . . . , x(k+L–1)$\}$ and puts them into cache
- This set is called a cache line of size L
- Although accessing x(k) may take 250 cycles, accessing x(k+1) takes only 3 cycles

Cache lines and stride

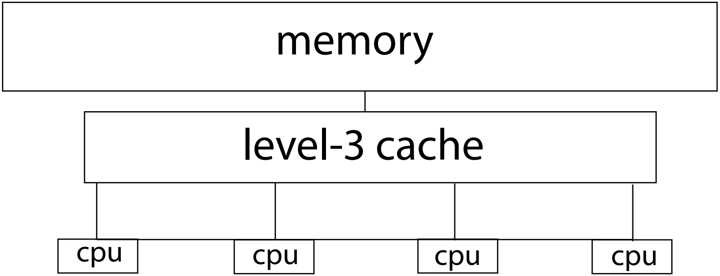
- If we first access `x(k)` and then access `x(k+s)` where $s > L$, then the result is a cache miss
- The stride is the distance between successive elements of an array that are accessed in a given region of code
- Stride $s = 1$ is the most efficient
- Important: Stride-1 accesses in MATLAB and Fortran require that you access an $m \times n$ array by columns
- In C/C++, access an $m \times n$ array by rows

Cache replacement

- The faster the cache, the smaller it is
- Caches are managed on a Last-In, First-Out (LIFO) basis
- The least recently accessed cache data is replaced once the cache fills up
- Data in L1 is retained in L2 until the L2 cache fills, and so on
- Each core in a multicore chip has its own L1 and L2 cache

Multicore processors share the same L3 cache

- The L3 cache usually is shared among all cores
- Threads are most efficient when they process small amounts of memory at a time on x86 machines



General programming strategies

- The working set is the data on which the innermost loops of the program are currently operating
- If the cache is small, then try to keep the working set small
- Complete as many operations as possible on the working set moving on to the next chunk of data
- Use stride-1 memory accesses whenever possible
- Break up long loops into multiple smaller loops that shrink the working set
- Minimize the number of temporary arrays

Tradeoffs in vectorization

- Memory access is the limiting factor in performance
- It's more important to manage cache efficiency than vectorization in compiled languages
- Vectors that are too large will spill from the cache
- Vectorization is essential to good performance in MATLAB
- General approach: Try to do as much as possible on suitably small chunks of data
- 32 KB is ~ 4,000 double-precision numbers

Example: Simple dense matrix multiplication

- Memory allocation strategy: Either rowwise or columnwise
- Rowwise: A(i,j) and A(i,j+1) are consecutive in memory
- Columnwise: A(i,j) and A(i+1,j) are consecutive
- One view of matrix-vector multipication:

$$(\mathbf{AB})_{ik} = \sum_{j=1}^{n} A_{ij} B_{jk}$$

i.e., the dot product of the ith row of $\mathbf{A}$ with the kth column of $\mathbf{B}$

Example: Simple dense matrix multiplication, 2

- The dot-product implementation is not cache friendly
- Either the row or the column traversal results in a cache miss on every subscript access for matrices of sufficient size
- The computation is a reduction operation, which does not vectorize as well as a linear combination of columns

Example: Simple dense matrix multiplication, 3

- Another view of matrix-matrix computation:

$$(\mathbf{AB})_k = \sum_{i=1}^{n} b_{ik}\mathbf{a}_i$$

  i.e., the kth column of the product is the linear combination of the columns of $\mathbf{A}$ with the elements of the kth column of $\mathbf{B}$
- MATLAB loop:

```
C(:,k) = 0;
for i=1:n
   C(:,k) = C(:,k) + B(i,k)*A(:,i);
end
```

The MATLAB code is cache friendly

- Vectors in MATLAB (and Fortran) are stored columnwise: `A(i,j)` and `A(i+1,j)` are consecutive in memory
- The elements of `C(:,k)` are consecutive, as are the elements of `B(i,k)` since the loop is over `i`

Example: Blocked matrix multiplication

- To compute $\mathbf{C} = \mathbf{AB}$, subdivide the matrices into blocks:

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)\left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right) = \left(\begin{array}{c|c} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ \hline A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{array}\right)$$

- The process can be continued recursively
- Each block can fit comfortably into L1 cache, and the algorithm still vectorizes—and can be multi-threaded