TEXAS ADVANCED COMPUTING CENTER

WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

Optimizing Vectorization & Multitasking on Manycore

Todd Evans

January 27th, 2017

3 General Rules for Optimization

Applies to Serial, Vectorized, and Multithreaded Code

1. Make compute intensive parts of code clean
   1.1 Compiler likes linear, predictable data structures
   1.2 Save OO for staging and managing workflow
   1.3 Compiler will do most of the work for you
2. Use optimization options (flags)
   2.1 tell compiler how hard to work
   2.2 tell compiler what instructions are allowed
3. Use math libraries (lapack/blas, MKL)
   3.1 These may thread automagically for you
   3.2 Teams of engineers have developed these over decades
   3.3 Specialized for particular architectures (e.g. KNL)

How-to Vectorize Your Program

Approaches

- Assembly (Bad)
  - Error prone, probably won't be that fast, not portable
- Instrinsics (Better)
  - Less error prone, probably not the fastest possible, still not portable
- Compiler (Good)
  - This is often the best idea!
  - Portable
  - Automatically vectorizes your code
- Link to optimized, vectorized libraries if available (Very good!)
  - Teams of scientists and engineers have put years into optimizing these
  - Portable
  - MKL

Vector Aware Programming

- Most vectorization opportunities are in loops
- Keep loops simple to help compiler
- Evaluate compiler output (tools exist for this)
- Evaluate Performance Improvement
- Consider Memory Access patterns (unit stride is good)
- Vector Math Libraries should be used for special functions
- MKL or equivalent should be used for linear algebra

How-to Achieve Automatic Loop Vectorization

Compiler flags - Loops will vectorize automatically

- Intel
  - `-O2` (SSE2/3/4)
  - `-xhost` (chooses most advanced)
  - `-xavx/-xavx2/-xmic-avx512`
- GCC
  - `-O3` (SSE2/3/4)
  - `-march=native` (chooses most advanced)
  - `-mavx/-mavx2/-mavx512 -mavx512cd -mavx512er -mavx512pf`

How-to Achieve Loop Vectorization

Characteristics required for loop vectorization

- Countable
  - Number of iterations must be know before loop executes
  - No conditional termination
- Single flow control
  - No switch statements
  - if statements are possible if masked
- Only innermost loop is vectorized
- No function calls allowed (except inlined or VML)
- No loop *dependencies*

Loop Dependencies

- Vectorization changes order of computation from sequential
- Compiler wants to vectorize but is careful to ensure correct results (conservative)
- Compiler performs dependency analysis
  - Dependent on vector width
- Compiler performs cost analysis
  - Not always faster to vectorize
  - Overhead associated with checking dependencies at runtime
  - short loops not always worth it

Dependency Analysis

Read After Write (RAW)

Variable is written to then read
Not Vectorizable
Intel calls this FLOW

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

Write After Read (WAR)

Variable is read then written to
Vectorizable
Intel calls this ANTI

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

Dependency Analysis

Read After READ (RAR)

Variable is read then read

Vectorizable

Not really a dependency

```
for (i=1; i<N; i++)
  a[i]=b[i%M]+c[i];
```

Write After Write (WAW)

Variable is written to then written to

NOT Vectorizable

```
for (i=1; i<N-1; i++)
  a[i%M]=b[i]+c[i];
```

Read After Write

Dependency Analysis: Do loads, operations, and stores across iterations in parallel produce the same results as executing each iteration sequentially?

```
for (i=1; i<N; i++)
    a[i]=a[i-1]*b[i];
```

iteration 1	a(2)	=	a(1)	*	b(2)
iteration 2	a(3)	=	a(2)	*	b(3)
iteration 3	a(4)	=	a(3)	*	b(4)
iteration 4	a(5)	=	a(4)	*	b(5)
	Store a(2-5)		Load a(1-4)	operate	Load b(2-5)

Read After Write

Dependency Analysis: Do loads, operations, and stores across iterations in parallel produce the same results as executing each iteration sequentially? NO!

Sequential

iteration 1	a(2)=2	=	a(1)=1	*	b(2)=2
iteration 2	a(3)=4	=	a(2)=2	*	b(3)=2
iteration 3	a(4)=8	=	a(3)=4	*	b(4)=2
iteration 4	a(5)=16	=	a(4)=8	*	b(5)=2

Vectorized

iteration 1	a(2)=2	=	a(1)=1	*	b(2)=2
iteration 2	a(3)=2	=	a(2)=1	*	b(3)=2
iteration 3	a(4)=2	=	a(3)=1	*	b(4)=2
iteration 4	a(5)=2	=	a(4)=1	*	b(5)=2

Write After Read

Dependency Analysis: Do loads, operations, and stores across iterations in parallel produce the same results as executing each iteration sequentially?

```
for (i=1; i<N-1; i++)
    a[i]=a[i+1]*b[i];
```

iteration 1	a(1)	=	a(2)	*	b(1)
iteration 2	a(2)	=	a(3)	*	b(2)
iteration 3	a(3)	=	a(4)	*	b(3)
iteration 4	a(4)	=	a(5)	*	b(4)
	Store a(1-4)		Load a(2-5)	operate	Load b(1-4)

Read After Write

Dependency Analysis: Do loads, operations, and stores across iterations in parallel produce the same results as executing each iteration sequentially? Yes!

Sequential

iteration 1	a(1)=2	=	a(2)=1	*	b(1)=2
iteration 2	a(2)=2	=	a(3)=1	*	b(2)=2
iteration 3	a(3)=2	=	a(4)=1	*	b(3)=2
iteration 4	a(4)=2	=	a(5)=1	*	b(4)=2

Vectorized

iteration 1	a(1)=2	=	a(2)=1	*	b(1)=2
iteration 2	a(2)=2	=	a(3)=1	*	b(2)=2
iteration 3	a(3)=2	=	a(4)=1	*	b(3)=2
iteration 4	a(4)=2	=	a(5)=1	*	b(4)=2

Striding: Cache Reuse

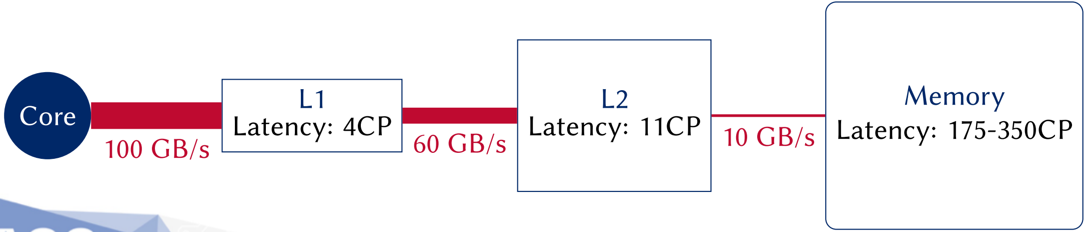
Distance between elements accessed in memory

- unit stride: `for(i=0;i<N;i++) a[i]=b[i];`
  stride of 2: `for(i=0;i<N;i+=2) a[i]=b[i];`
  stride of 4: `for(i=0;i<N;i+=4) a[i]=b[i];`
- whole cache lines (8 DP elements) are always moved from memory to cache
  - slow to move data from memory to cache
  - unit stride uses each element (enhances cache reuse)
  - unit stride allows 8 DP to be loaded from cache into vector register w/ 1 load!
  - non-unit stride requires multiple loads :(
- effect is greatest when reading from memory, least when reading from L1

Why use caches?

Memory Hierarchy

- Caches are smaller but faster than memory
- May have multiple levels of cache: L1, L2 . . .
- Helps keep cores fed with data
- Cache reuse is often critical to performance



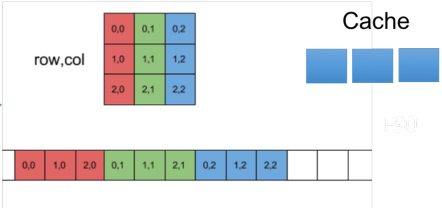
Striding

1-D Array

Stride 1 access is best! Uses all the elements in the cache line!

Multi-D Array

Fortran: Stride 1 access on "inner" dimension is best.
C/C++: Stride 1 access on "outer" dimension is best.




```
do j=1,3
  do i=1,3
   a(i,j)=b(i,j)*s
  enddo
endo
```

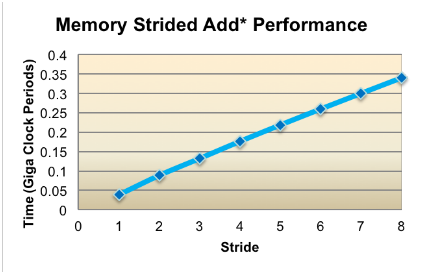
```
for(j=0;j<3;j++)
  for(i=0;i<3;i++)
   a[j][i]=b[j][i]*s;
```

Strided Access

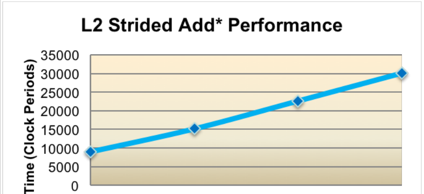
Memory BW is scarce in Manycore

- Striding is even more important
- Striding through memory is bad
  - stride 2 = 1/2 effective bandwidth
  - stride 4 = 1/4 effective bandwidth
  - stride 8 = 1/8 effective bandwidth
- Striding through L2 and L1 not *as* bad
  - still requiring more loads

```
*do i = 1,4000000*istride, istride
   a(i) = b(i) + c(i) * sfactor
enddo
```



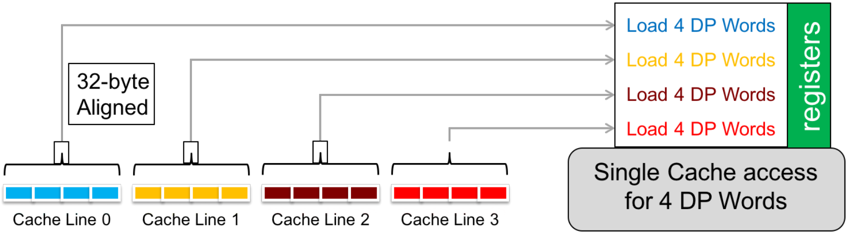

```
*do i = 1, 2048*istride, istride
   a(i) = b(i) + c(i) * sfactor
enddo
```

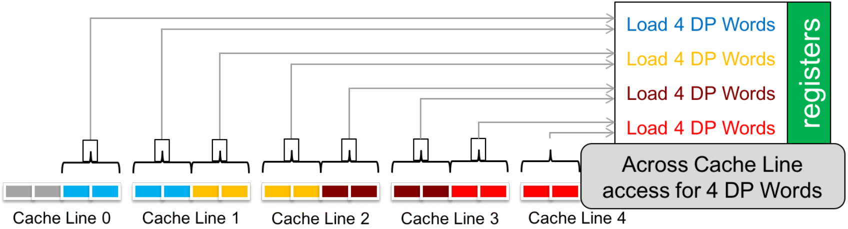


Alignment

- Data objects can be created in memory on specific byte boundaries
- Can increase efficiency of loads and stores
  - A vector load can only access one cache line at a time
- For AVX/AVX2 (SNB/HSW), alignment to 32B boundaries
  - allows a single load to move 4 DP data elements
- For AVX-512/MIC-AVX-512 (KNL), alignment to 64B boundaries
  - allows a single load to move 8 DP data elements
- Compiler can detect lack of alignment and peel off iterations to achieve optimal alignment for bulk of loop
  - Some overhead
  - `void *memalign(size_t alignment, size_t size);`

Alignment: Loads

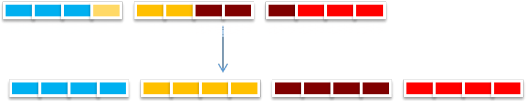




Alignment: Padding

In multi-dimensional array, resizing (padding) lower dimensions for alignment can improve performance

```
a(4,3)→a(4,4)
```



```
for(i=0;i<nx;i++)
    for(j=0;j<ny;j++)
        b(i,j) = a(i,j)
```

Inlining

Functions within loops prevent vectorization

- Inlining can overcome this
- If function call and def are in same file, `-ip` or `-O3` inlines
- If function call and def are in separate file, only `-ipo` inlines
- After inlining vectorization occurs

```
for(i=0;i<nx;i++) {
    x=x0+i*h;
    sum=sum+
        do_r2(x,y,xp,yp);
    }
```

```
double do_r2(double x,double y,
        double xp, double yp) {
        double r2;
        r2=(x-xp)*(x-xp)+
            (y-yp)*(y-yp);
        return r2;
        }
```

Inlining effects

Before Inlining

```
double r2(x,x0,y,y0)
    return (x-x0)*(x-x0)+(y-y0)*(y-y0);

int main(){
...
for (i=0;i<Nx;i++)
    for (j=0;j<Ny;j++)
        d[j]+=r2(i,a0,j,b0);
...
return 0;
}
```

After Inlining

```
int main(){
...
for (i=0;i<Nx;i++)
    for (j=0;j<Ny;j++)
        d[j]+=(i-x0)*(i-x0)+(j-y0)*(j-y0)
...
return 0;
}
```

Inlining	Vectorization	Time (ms)
no	no	1.55
yes	no	0.44
yes	yes	0.056

Aliasing

- Two different pointers can have the same target (e.g. point to the same array or array element)
- Loops in routines that use pointer references might not vectorize
- Compiler has to be certain vectorization does not introduce dependencies
- In functions that pass pointers, ensure the compiler can detect whether references overlap

Example: my_cp(N,x,x-1)

```
void my_cp(int N,double *a,double *b) {
    for (int i=0;i<N;i++)
        a[i]=b[i];
    }
```

Aliasing

Compiler can test for overlap in simple circumstances

```
void my_cp(int N,double *a,double *b) {
    if (a+N>b || b+N>a)
        for (int i=0;i<N;i++)
            a[i]=b[i];
    else
        for (i=0;i<N;i+=4){
            a[i+0]=b[i+0];
            a[i+1]=b[i+1];
            a[i+2]=b[i+2];
            a[i+3]=b[i+3];
            }
    }
```

Aliasing

Overlap test can be too expensive in some cases

```
void my_cp(int nx,double *a,double *b,int *ioff) {
    for (int i=0;i<nx;i++)
        a[i]=b[i+*ioff];
```

- Compiler decides runtime overlap test is not cost effective
- b may be aliased to a and/or *ioff might point to a[i]
- multiple possible dependencies
- `-ansi-alias` flag prevents compiler from aliasing different types (enables vectorization in this example)

Pragmas

Pragmas can also tell compiler no aliasing: Developer's responsibility

- #pragma ivdep
  - no assumed dependencies
  - vectorize if dependency not proven
- #pragma vector always
  - vectorize if no assumed or proven dependencies
  - ignore cost
- #pragma novector
  - never vectorize
- #pragma simd
  - always vectorize
  - ignore cost and safety

Vectorization Summary

- KNL with MIC-AVX-512 is here: 16 simultaneous FLOPs
- Wider vector register does not guarantee improved performance
- Ensure no explicit dependencies
- Ensure no functions (or inline)
- Ensure no aliasing

Multitasking

Basic Concepts

- Multiprocessing: tasks are processes
  - each task has own address space
- Multithreading: tasks are threads within a process
  - tasks share address space
- *speedup*: runtime of parallel versus serial $\frac{T_p}{T_s}$
- *efficiency*: speedup versus number of tasks $\frac{T_p/\#tasks}{T_s}$
  - Typically less than 1
- Process/memory affinity: 1 task per core w/ nearby memory
- Contention for shared resources (cache/memory/interconnect)
- Dependencies (same as vectorization issues)
- *Granularity*

Granularity: Load Balancing vs Overhead

- Granularity is the amount of work per task
- It is important to balance granularity for parallel efficiency
- Too much granularity can lead to load imbalance
  - Some tasks could be idle for long periods, waiting for others to complete
- Too little granularity can lead to large overhead
  - Parallelizing is not free
    - Partitioning problem
    - Assigning work to tasks
    - Communicating data between tasks
    - Coordinating tasks

Load Imbalance Example

A loop with 10 iterations, divided among 9 threads

```
for(j=0;j<10;j++){...}
```

- Each iteration takes 1 thread 1 day
- 9 iterations are done the 1st day by 9 threads
- 1 iteration is done the 2nd day by 1 thread (9 threads are idle for a day :( )
- Negligible communication overhead but poor load balance

Parallel Overhead Example

A loop with 10 iterations, divided among 10 threads

```
for(j=0;j<10;j++){...}
```

- Setup of parallelization requires 1ms
- Each iteration takes 1 thread 0.1ms
- 10 iterations are done by 10 threads in 1 ms
- Total parallel time is 2ms $\rightarrow$ serial would have run in 1ms!

Loop Modifications for Parallel Performance

- Loops are often where parallelization can be realized
  - Divide iterations among tasks
  - Iterations must be independent
- Loop modifications can make parallelization possible
  - Remove dependencies between iterations
- Loop modifications can improve granularity
  - Can improve load balancing
  - Can reduce overhead

General tip for good loop granularity

Parallelize over loops with high iteration counts relative to the number of threads.

- Overhead is less significant: Time to setup versus time spent on work is small
- Load imbalance is less significant: Time for “extra” iterations versus time all threads are working is less

Loop fission

Can enable parallelization by removing dependencies

Later iteration needs result from earlier iteration (dependency):

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

Loop fission can remove this:

```
for (j=1; j<N; j++){
    b[j]+=1;
}

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

Loop Fusion

Can increase granularity → reduces parallelization and loop overhead

```
for (j=0;j<N;j++)
        a[j]=b[j]*c[j];
 for (j=0;j<N;j++)
        d[j]=e[j]*f[j];
```

Loop fusion

```
for (j=0;j<N;j++){
        a[j]=b[j]*c[j];
        d[j]=e[j]*f[j];
}
```

Merge nested loops

Increases iteration count → better load balancing

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        d[i][j]=e[i]*f[j];
```

Merge

```
for (k=0; k<N*N; k++){
    i=k/N;
    j=k%N;
    d[i][j]=e[i]*f[j];
}
```

Load balancing with OpenMP

OpenMP has *scheduling*

```
#pragma omp parallel for schedule(scheduletype, chunksize)
  for(k=0;k<N*N;k++)
          work(k);
```

- static
  - default scheduler
  - same iterations for every thread
  - very little overhead
  - no load balancing
- dynamic
  - assign work in chunksize (default=1)
  - when thread completes, new work is assigned
  - overhead from synchronization
- guided
  - similar to dynamic
  - initially gives large chunks of iterations to tasks
  - gradually reduces chunks of iterations to min of chunksize
  - sometimes overhead than dynamic

Aside: Super-linear Speedup/Slowdown

Speed-up exceeds number of cores

- Usually happens when problem is decomposed small enough to fit in each core's cache
- Serial problem size did not fit in cache - streamed from memory and back, possibly many times
- Basically improved cache line reuse
- Be careful analyzing speed-up with datasets that are too small!

Slowdown for larger problem sizes even with more threads

- Little cache reuse
- Memory accesses are dominating
- Hide latency

Compiler Generated Optimization Reports

The Intel compiler can generate optimization reports

- vectorization reports
  - what loops were or were not vectorized and why
  - vectorization count
  - peeled remainder loops
- OpenMP parallelization
- inlining
- loop transformations
  - vectorization
  - cache reuse

Compiler Generated Optimization Reports

Pretty easy to use

- `$ icc -xhost -qopt-report=5 example.c -o example`
- Two main sections:
  - `Report from: Interprocedural optimizations [ipo]`

```
-> EXTERN: (23,23) _vla_alloc(long)
-> EXTERN: (34,7) clock(void)
-> INLINE: (35,12) rdtsc() (isz = 4) (sz = 9)
-> INLINE: (43,12) rdtsc() (isz = 4) (sz = 9)
```

  - `Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]`

```
LOOP BEGIN at vector.c(39,5)
remark #25444: Loopnest Interchanged: ( 1 2 3 ) --> ( 2 1 3 )
remark #15542: loop was not vectorized: inner loop was already vectorized   [ vector.c(39,\
5) ]
```

Compiler Generated Optimization Reports

```
LOOP BEGIN at omp_vector.c(46,7)
   <Peeled loop for vectorization>
      remark #15389: vectorization support: reference B[i][j] has unaligned access   [ omp_vector.c(47,10) ]
      remark #15381: vectorization support: unaligned access used inside loop body
      remark #15335: peel loop was not vectorized: vectorization possible but seems inefficient. Use vector always
      remark #15305: vectorization support: vector length 8
      remark #15309: vectorization support: normalized vectorization overhead 2.158
      remark #25015: Estimate of max trip count of loop=7
   LOOP END

   LOOP BEGIN at omp_vector.c(46,7)
      remark #25085: Preprocess Loopnests: Moving Out Load and Store   [ omp_vector.c(47,2) ]
      remark #15388: vectorization support: reference B[i][j] has aligned access   [ omp_vector.c(47,10) ]
      remark #15305: vectorization support: vector length 8
      remark #15399: vectorization support: unroll factor set to 8
      remark #15309: vectorization support: normalized vectorization overhead 0.643
      remark #15300: LOOP WAS VECTORIZED
      remark #15448: unmasked aligned unit stride loads: 1
      remark #15475: --- begin vector cost summary ---
      remark #15476: scalar cost: 7
      remark #15477: vector cost: 0.870
      remark #15478: estimated potential speedup: 6.100
      remark #15488: --- end vector cost summary ---
   LOOP END

   LOOP BEGIN at omp_vector.c(46,7)
   <Remainder loop for vectorization>
      remark #15388: vectorization support: reference B[i][j] has aligned access   [ omp_vector.c(47,10) ]
      remark #15305: vectorization support: vector length 8
      remark #15309: vectorization support: normalized vectorization overhead 2.111
      remark #15301: REMAINDER LOOP WAS VECTORIZED
   LOOP END
```

Questions?