CRAY

POWERED BY EXPERIENCE

CRAY USER GROUP INCORPORATED

Cray X1 Basic Optimization Techniques

James L. Schwarzmeier
Cray Inc.
jads@cray.com
715-726-4756

Outline

- Steps for profiling – finding out what performance you are getting – Loopmark, irtc(), CPAT, compiler options, directives
- Level 1 optimizations
  - write code with maximum stream & vector parallelism exposed to compiler
  - achieve vectorization and streaming
- Level 2 optimizations
  - may start slower than micros, but *can* achieve ~ 15-40% of peak!
  - *Improving* vectorization
  - *Improving* streaming (also, why stream?)
  - *improving* cache hit rates → links vectorization and streaming (case studies of optimizations and payoffs)
  - *Improving* communication performance

MSP vs SSP vs OpenMP

- **Advantages of MSP:**
  - more powerful processor – good if app has limited processor-level parallelism
  - attacks new level of parallelism, sometimes
  - improves “surface to volume” ratio for many DM jobs
  - reduces number of PEs during barriers and global communications
- **Advantages of SSP (-Ossp or –hssp) :**
  - best if application does *not have* much stream parallelism and scalability of application is good
- **Advantages of OpenMP:**
  - DYNAMIC and GUIDED options help with load balancing (NOTE: MSP always uses *static* work distribution among SSPs)
  - even smaller ‘surface-to-volume’ ratio than MSP
  - fewer MPI processes for communication

Level 1: Vectorization
where are you coming from?

- if coming from microprocessor code…
  - if necessary, restructure code to place nested loops of parallelism in routines (next page)
  - strive for reasonable loop lengths, N > 50
  - cache blocking may be good for X1, but not if VL < ~50
- if coming from SX6 code…
  - might be ok. But, may want to re-order loops to reduce #Vloads per flop by vectorizing *outer* loops, etc.
  - extremely long inner vector loops sometimes cause poor cache performance ➔ stripmine vector loop

Peformance in hand of code developers, cont.

- Bad for X1

```
 do ie = 1,nelem
   call small_work(x(1,1,ie),…)
enddo

subroutine small_work(a, b,..)

  do j = 1,n      ← n ~ 4-8
   do l = 1,m     ← m ~ 4-8
     …few flops…
  enddo;enddo
end
```

- Good for X1

```
 call big_work(x, m, n, nelem)

subroutine big_work(x, m, n, .)

 do ie = 1, nelem
   do j = 1,n
     do l = 1,m
        …many flops…
  enddo;enddo;enddo
end
```

Level 1 Optimizations-- vectorization

- reasons why compiler cannot *vectorize* loops
  - recurrences:
    - x(I) = x(I-1)+…
  - subscript ambiguities:
    - x(I+K) = x(I)+… ← sign of K unknown
    - x(ind(I)) = x(ind(I)) + b(I) ← repeated indices for ind(I)?
  - subroutine calls, system calls (I/O)
  - spaghetti code -- complicated branching
- try to eliminate these problems
  - isolate recurrences from other code, different algorithm?
  - !dir$ concurrent, if no real dependencies [#pragma concurrent]
  - inline subroutines, eliminate or move system calls to separate loop
  - is spaghetti code necessary?

Level 1 Optimizations -- streaming

- reasons why compiler cannot *stream* loops
  - problem: data dependencies between SSPs
    - do j = 1,n-1
    - x(1:m,j) = x(1:m,j+1) + b(1:m,j) ← stream j+1 not independent
    - of stream j
    - compiler streams and vectorizes 1:m – OK *if* m large
  - solution: *vectorize j, stream 1:m*
    - !dir$ prefervector
    - do j = 1,n-1
    - x(1:m,j) = x(1:m,j+1) + b(1:m,j)

CUG 2004

Level 1 Optimizations -- streaming

– problem: local work array not SSP private
  - dimension a(100) ← compiler *will* privatize automatically
  - dimension a(N) ← compiler will *not* privatize automatically
  - do j = 1,n ← want to stream over j but can't because a(1:m) independent of j
  - a(1:m) = c(1:m,j)
  - . . . use a(1:m) . . .

– solution: *manually privatize a(1:m)*
  - dimension a(N,4) ← work array replicated explicitly or via CSD's
  - do issp = 1,4
  - do j = issp,n,4
  - a(1:m, issp) = c(1:m,j)
  - . . . Use a(1:m,issp) . . .

CUG 2004

Level 1 Optimizations -- streaming

- problem: subroutine calls
  - do j = 1,n
  - call work(j) ← compiler unsure if work(j) can be executed in
  - MSP mode
- solution: *use CSD & compile ftn … -Ogen_private_callee work.f*
  - !CSD$ PARALLEL DO PRIVATE (..)
  - do j = 1,n
  - call work(j)
  - enddo
  - !CSD$ END PARALLEL DO
  - *cc … -hgen_private_callee work.c*
  - #pragma csd parallel for private (…) schedule(static,1) { }

CUG 2004

Streaming Optimization: False Dependence on Temp Array

```
common /something/ atemp(n)
do j = 1,m
   do i = 1, n
      atemp( i ) = sqrt( b(i,j) )
      c(i,j) = c(i,j) + atemp(i)
enddo; enddo
```

- Inner loop vectorizes
- Outer loop does not stream due to false dependence on atemp

NO False Dependence on Temp Scalar Variable

```
real stemp
do j = 1,m
   do i = 1, n
      stemp = sqrt( b(i,j) )
      c(i,j) = c(i,j) + stemp
enddo; enddo
```

- Inner loop vectorizes
- Outer loop streams; More efficient
- May manually fuse loops to remove temporary arrays

Streaming/Vectorizing Gather/Scatter Kernel:
What developer should write

- if *no* repeated indices of indx(i) for i = 1,n, insert directive

```
!dir$ concurrent
do i = 1, n   ! Loop will Vectorize, Stream, and Unroll
      a( indx(i) ) = a( indx(i) ) + b(i)
enddo
```

- if there *are* repeated indices, can indx(I) values be sorted into chunks that have no repeats?

Level 2 – how to *improve* vectorization

- seek to:
  - increase granularity of work in leaf routines befitting a 12.8 GFLOPS processor
  - ***decrease # Vreferences* per flop**: (saves memory bandwidth)
    - **write tightly nested loops -- allows compiler greater loop interchange ability, or, manually interchange loops to vectorize *outer* loops**
    - **compiler *fuses* loops to save memory references, or, manually restructure loops to eliminate temporary arrays in favor of register-carried temporaries**
    - **compiler does loop unrolling, but, sometimes manually unroll *outer* loops (see MxM example) can further reduce #Vrefs**
  - vectorize loops with longer VL (compiler can't always tell)
    - **!dir$ prefervector [#pragma prefervector]**
  - eliminate bad vector strides (large power of 2)
  - experiment with making some arrays be *non*-allocating
  - experiment with improving cache hit rates by stripmining, …

Level 2 -- *Improving* vectorization

- X1 compiler has risen to occasion: does great job *interchanging* loops, vectorizing *outer* loops, and *unrolling* loops to minimize #Vloads

```
 6.  C----------<      do l = 1,l2          ← dimension(64,j2,k2,l2) :: a, d
 7.  C Mr-------<       do k = 1,k2
 8.  C Mr i-----<         do j = 1,j2
 9.  C Mr i Vs--<           do i=1,64
10.  C Mr i Vs                a(i,j,k,l) = b(i) + c(i,j) + d(i,j,k,l)
11.  C Mr i Vs->>   enddo; enddo; enddo; enddo
```

A loop starting at line 6 was collapsed into the loop starting at line 7.

A loop starting at line 7 was not vectorized because a better candidate was found at line 9

A loop starting at line 7 was unrolled 2 times.

A loop starting at line 7 was multi-streamed.

A loop starting at line 8 was interchanged with the loop starting at line 9.

CUG 2004

Level 2: *Improving* vectorization

- *unkown* loop bounds means user can aid in optimizing:
  - when vectorizing outer loops, indices of *smallest rank* arrays go as *outer* loops (i.e., b(i))
  - for similar rank arrays, loops should be ordered as *decreasing* length towards *outer* loops

Form 1	**Form 2**	**Form 3**	**Form 4**
do l = 1,l2	do l = 1,l2	do i = l,64 ← **V**	do i=1,64 ← **V**
do k = 1,k2	do k = 1,k2	do j = 1, j2	do j = 1,j2
do j = 1,j2	do i = 1,64 ← **V**	do k = 1,k2	do l = 1,l2
do i=1,64 ←**V**	do j = 1,j2	do l = 1,l2	do k = 1,k2

←---------------------------- a(i,j,k,l) = b(i) + c(i,j,l) + d(i,j,k) -----------------------------------→

#Vloads =	#Vloads =	#Vloads =	#Vloads =
3*j2*k2*l2	k2*l2+2*j2*k2*l2	1+j2*k2+j2*k2*l2	1+j2*l2+j2*k2*l2

  - if users knows l2<<k2, Form 4 has j2*(k2-l2) *fewer* Vrefs than Form 3
- being able to vectorize outer loops crucial to reducing #Vrefs

Level 2: *Improving* vectorization

- vectorize longer loops if compiler doesn't know better

```
!dir$ prefervector
do j = 1, j2                    ← do if i2 << 64 and j2 > 64, even though non-unit stride
!dir$ nextscalar
   do i = 1,i2
      a(i,j) = b(i,j) + c(i,j)
```

- eliminate large PO2 vector strides

```
dimension x(16,100)
do j = 1,100
  y(j) = F(x(1,j), x(2,j), …, x(16,j))
enddo
```

  - bad, since consecutive j-values use only 1 (out of 4) ports to E
  - re-dimension as x(20, 100) ← now consecutive j-values use consecutive ports to E (but, takes more memory)

- if spatial locality *unlikely*, try making non-unit stride references be *non*-allocating: !dir$ no_cache_alloc a, b at top of subroutine. Also for very large stride-1 arrays when other arrays could get reuse in cache.

Level 2: *Improving* Streaming

- Extend streamed region to outermost loops*, or above,* in order to minimize MSP startups

```
!CSD$ PARALLEL PRIVATE(k1,k2,…)
      k1 = 1; k2 = 3                          }
      if(ir .eq.    1) k1 = 2                  }  redundantly executed
      if(ir .eq. nr+1) k2 = 2                  }  by each SSP
      do i = 1,nii,maxVL*nssp                  }
!CSD$ DO SCHEDULE(STATIC, 1)
       do issp = 0,nssp-1                   ← streamed loop
```

- Use 'cyclic' work distribution to improve load balancing

```
!CSD$ PARALLEL DO SCHEDULE(STATIC,1)
     do k = 1,n          ←SSP0 takes k={1,5,9,..} rather than {1:n/4}, etc.
!dir$ prefervector
       do i = k,n        ← 'triangular' load imbalance --small k-values have
         x(i,k) = …         the most vector work!
```

Case Studies of Optimizations and Payoffs

- best way to illustrate optimizations combines vectorization and streaming
- three case studies
  - Case 1: customer loops #1
    - from **1610 MFLOPS → 6900 MFLOPS**
  - Case 2: 64b matrix multiply (Fortran)
    - from **3650 MFLOPS → 10730 MFLOPS**
  - Case 3: customer loops # 3
    - from **3820 MFLOPS → 6900 MFLOPS**

Case I

- DO 10 fills temp work array 'W(MS,3,3)' in terms of array XV
- DO 20 uses W to calculate S(MS,3)

```
parameter (MS = 2**18)      ! NOTE: 2**18 = size(E cache)
dimension XV(MS,3), W(MS,3,3), S(MS,3)
K1 = 1; K2 = 64; K3 = 4096
M0   = 0; M1 = K1 + M0; M2 = K2 + M0; M3 = K3 + M0
DO 10 I=ISTRT(IR),MS
 W(I,1,1) = XV(I+M0,1) + XV(I+M1,1)      ← define work array W in terms of data array XV
 W(I,2,1) = XV(I+M0,1) + XV(I+M2,1)
 W(I,3,1) = XV(I+M0,1) + XV(I+M3,1)
 W(I,1,2) = XV(I+M0,2) + XV(I+M1,2)
 W(I,2,2) = XV(I+M0,2) + XV(I+M2,2)
 W(I,3,2) = XV(I+M0,2) + XV(I+M3,2)
 W(I,1,3) = XV(I+M0,3) + XV(I+M1,3)
 W(I,2,3) = XV(I+M0,3) + XV(I+M2,3)
 W(I,3,3) = XV(I+M0,3) + XV(I+M3,3)
10 CONTINUE
```

Case I (cont)

```
 DO 20 I=ISTRT(IR),MS
 S(I,1) =  W(I,3,1)*(W(I+K3,2,2)*W(I,2,3) - W(I+K3,2,3)*W(I,2,2))
&       + (W(I,3,2)*(W(I+K3,2,3)*W(I,2,1) - W(I+K3,2,1)*W(I,2,3))
&       +  W(I,3,3)*(W(I+K3,2,1)*W(I,2,2) - W(I+K3,2,2)*W(I,2,1)))
 S(I,2) =  W(I,1,1)*(W(I+K1,3,2)*W(I,3,3) - W(I+K1,3,3)*W(I,3,2))
&       + (W(I,1,2)*(W(I+K1,3,3)*W(I,3,1) - W(I+K1,3,1)*W(I,3,3))
&       +  W(I,1,3)*(W(I+K1,3,1)*W(I,3,2) - W(I+K1,3,2)*W(I,3,1)))
 S(I,3) =  W(I,2,1)*(W(I+K2,1,2)*W(I,1,3) - W(I+K2,1,3)*W(I,1,2))
&       + (W(I,2,2)*(W(I+K2,1,3)*W(I,1,1) - W(I+K2,1,1)*W(I,1,3))
&       +  W(I,2,3)*(W(I+K2,1,1)*W(I,1,2) - W(I+K2,1,2)*W(I,1,1)))
20 CONTINUE
```

Case I (cont)

- Chronology of optimizations

Version	Optimization	MFLOPS/MSP	Comments
Source 0	as is	**616** -do 10	#Vloads=12 #Vstores=9 #F=9
		2462 -do 20	#Vloads=18 #Vstores=3 #F=42
		1610 -total	#Vloads=30 #Vstores=12 #F=51
			CI = 51/42 **= 1.21**
Source 1	changed lda	**716** -do 10	2**18 = 2MB, Size(E)
	arrays MS=2**18 →	**3092** -do 20	For given i each SSP
	2**18+512 to improve	**1950** -total	want loads of XV(i:i+127,1:3),
	hit rates for XV(i,j),		XV(i+4096+127,1:3),...W
	W(i,j,k), as j,k = 1,2,3		to map to *different* cache
			sets
Source 2	!dir$ no_cache_alloc XV	**781** -do 10	
	so XV does not pollute E	**3069** -do 20	
	for W in DO 10 loop. DO	**2023** -total	
	20 loop uses only W		

Case I (cont)

Source 3	Eliminate DO 10 loop by	**5231**	a) eliminates stores and
	replacing array W with		later loads of W in favor
	Fortran statement functions		of loads of XV, b) has
	in DO 20		smaller footprint in cache,
			XV has smaller size than W
			c) allows compiler max reuse
			of XV in vector registers
			#Vloads=26, #Vstores=3, #F=51
			CI = 51/29 **= 1.76**

Source 4	eliminate !dir$ no_cache_	**6902**	this is purely improvement due
	alloc XV, since we want		to using cache rather than
	combined DO 1020 loop to		memory
	temporal locality for XV		

Case I (cont)

```
    W11(I) = XV(I+M0,1) + XV(I+M1,1)    !  Fortran function statements
    W21(I) = XV(I+M0,1) + XV(I+M2,1)
    W31(I) = XV(I+M0,1) + XV(I+M3,1)
    W12(I) = XV(I+M0,2) + XV(I+M1,2)
    W22(I) = XV(I+M0,2) + XV(I+M2,2)
    W32(I) = XV(I+M0,2) + XV(I+M3,2)
    W13(I) = XV(I+M0,3) + XV(I+M1,3)
    W23(I) = XV(I+M0,3) + XV(I+M2,3)
    W33(I) = XV(I+M0,3) + XV(I+M3,3)
  DO 20 I=ISTRT(IR),MS
    S(I,1) =  W31(I)*(W22(I+K3)*W23(I) - W23(I+K3)*W22(I))  ← array W completely gone
   &       + (W32(I)*(W23(I+K3)*W21(I) - W21(I+K3)*W23(I))
   &       +  W33(I)*(W21(I+K3)*W22(I) - W22(I+K3)*W21(I)))
    S(I,2) =  W11(I)*(W32(I+K1)*W33(I) - W33(I+K1)*W32(I))
   &       + (W12(I)*(W33(I+K1)*W31(I) - W31(I+K1)*W33(I))
   &       +  W13(I)*(W31(I+K1)*W32(I) - W32(I+K1)*W31(I)))
    S(I,3) =  W21(I)*(W12(I+K2)*W13(I) - W13(I+K2)*W12(I))
   &       + (W22(I)*(W13(I+K2)*W11(I) - W11(I+K2)*W13(I))
   &       +  W23(I)*(W11(I+K2)*W12(I) - W12(I+K2)*W11(I)))
   20 CONTINUE
```

Case II, MxM

- Matrix multiply (all Fortran): from **3650 →10730** MFLOPS
  - insert directives to force vectorization of inner loop ala DAXPY
  - baseline -- this ran in 3650 MFLOPS

```
                     parameter (M=400,N=M,L=M,ldm=400,ldn=400)
                     real*8 x(ldm,n), y(ldm,l), z(ldl,n), sumj, sumjp1
                    !dir$ preferstream
MC--------<             do j = 1,N
MC V M---<>                x(:,j) = 0.
MC               !dir$ nounroll
MC 2------<                do k = 1,L
MC 2              !dir$ nointerchange
MC 2              !dir$ prefervector
MC 2 V----<                   do i = 1,M
MC 2 V                           x(i,j)   = x(i,j) + y(i,k)*z(k,j)     ← DAXPY
MC 2 V---->                   enddo   ! do i = 1,M
MC 2------>                enddo             ! do k=1,L
MC-------->             enddo              ! do j = 1,N,2
```

Case II (cont)

- Chronology of optimizations

Version	Optimization	MFLOPS/ MSP	Comments sets
Source 0	as is	**3650**	#Vloads=2N**3 #Vstores=N**3 #F=2N**3 **CI = 2/3**
Source 1	removed all directives	**7690**	#Vloads=(N**2+N**3), #Vstores=2N**2 **CI** ~ 2N**3/N**3 = **2.0** unrolling k hides latency of Sloads of z
Source 3	let M → 1000 *miss E*	**5369**	with *large M*, x(1000,1000),y(1000,1000)

Source 1 listing:

```
33.  MC---------<        do j = 1,N
34.  MC V M----<>          x(:,j) = 0.
35.  MC ir------<          do k = 1,L
36.  MC ir V----<            do i = 1,M
37.  MC ir V                 x(i,j)   = x(i,j) + y(i,k)*z(k,j)
38.  MC ir V---->            enddo  ! do i = 1,M
39.  MC ir------>          enddo          ! do k=1,L
40.  MC--------->        enddo            ! do j = 1,N
```

CUG 2004

Case II (cont)

- Chronology of optimizations

```
--------------------------------------------------------------------------------
Source 4 | unroll do j = 1,N,2        | 10222  | #Vloads=(N**2+1/2N**3), #Vstores=N**2
         |                            |        | CI ~ 2N**3/(1/2)N**3 = 4
    !CSD$ PARALLEL DO SCHEDULE(STATIC,1)
        do j = 1,N,2    !   <-- MSP here with 'cyclic' work distr.
    !dir$ prefervector
         do i = 1,M
          x(i,j)   = 0.      !   <-- zero Vreg, not memory
          x(i,j+1)  = 0.     !   <-- zero Vreg, not memory
    !dir$ unroll 8
          do k = 1,L
           x(i,j)   = x(i,j)   + y(i,k)*z(k,j)
           x(i,j+1) = x(i,j+1) + y(i,k)*z(k,j+1)
          enddo      ! do k=1,L
         enddo       ! do i = 1,M
        enddo        ! do j = 1,N
    !CSD$ END PARALLEL DO
```

CUG 2004

Case II (cont)

- Chronology of optimizations

```
--------------------------------------------------------------------------------
Source 5 | unroll do j = 1,N,4          | 10730  | #Vloads=(N**2+1/4N**3), #Vstores=N**2
         |                              |        | CI ~ 2N**3/(1/4)N**3 = 8
     !CSD$ PARALLEL DO SCHEDULE(STATIC,1)
          do j = 1,N,4    !    <-- MSP here with 'cyclic' work distr.
     !dir$ prefervector
           do i = 1,M
            x(i,j) = 0.;x(i,j+1) = 0.;x(i,j+2) = 0.;x(i,j+3) = 0.
     !dir$ unroll 8
            do k = 1,L
             x(i,  j)   = x(i,j)   +  y(i,k)*z(k,j)
             x(i,j+1) = x(i,j+1) + y(i,k)*z(k,j+1)
             x(i,j+2) = x(i,j+2) + y(i,k)*z(k,j+2)
             x(i,j+3) = x(i,j+3) + y(i,k)*z(k,j+3)
            enddo       ! do k=1,L
           enddo        ! do i = 1,M
          enddo         ! do j = 1,N
     !CSD$ END PARALLEL DO
```

Conclusions

- good code on X1 has nested loop bodies, many flops per memory reference, vector & stream parallelism evident to compiler
- Cray X1 vector/stream optimizations *can* deliver high performance – *X1 responds well*
  - vectorize all important loops and strive for large granularity vector/stream loops
  - reducing # vector references key
  - enabling cache hits can be important
- communication optimization → John L.