Hewlett Packard Enterprise

Performance Optimization: Improving Single-core Efficiency

LUMI Advanced Workshop

March 5–7, 2025

Context of This Talk

- Scope of the presentation
  - Poor performance of an application can have many causes: placement, OS, compilers, libraries, hardware, etc.
  - We will consider possible techniques to optimize execution on a single core
  - In practice this means we should consider if we are using the processor hardware and memory hierarchy efficiently and if the compiler is compiling our application for optimum performance

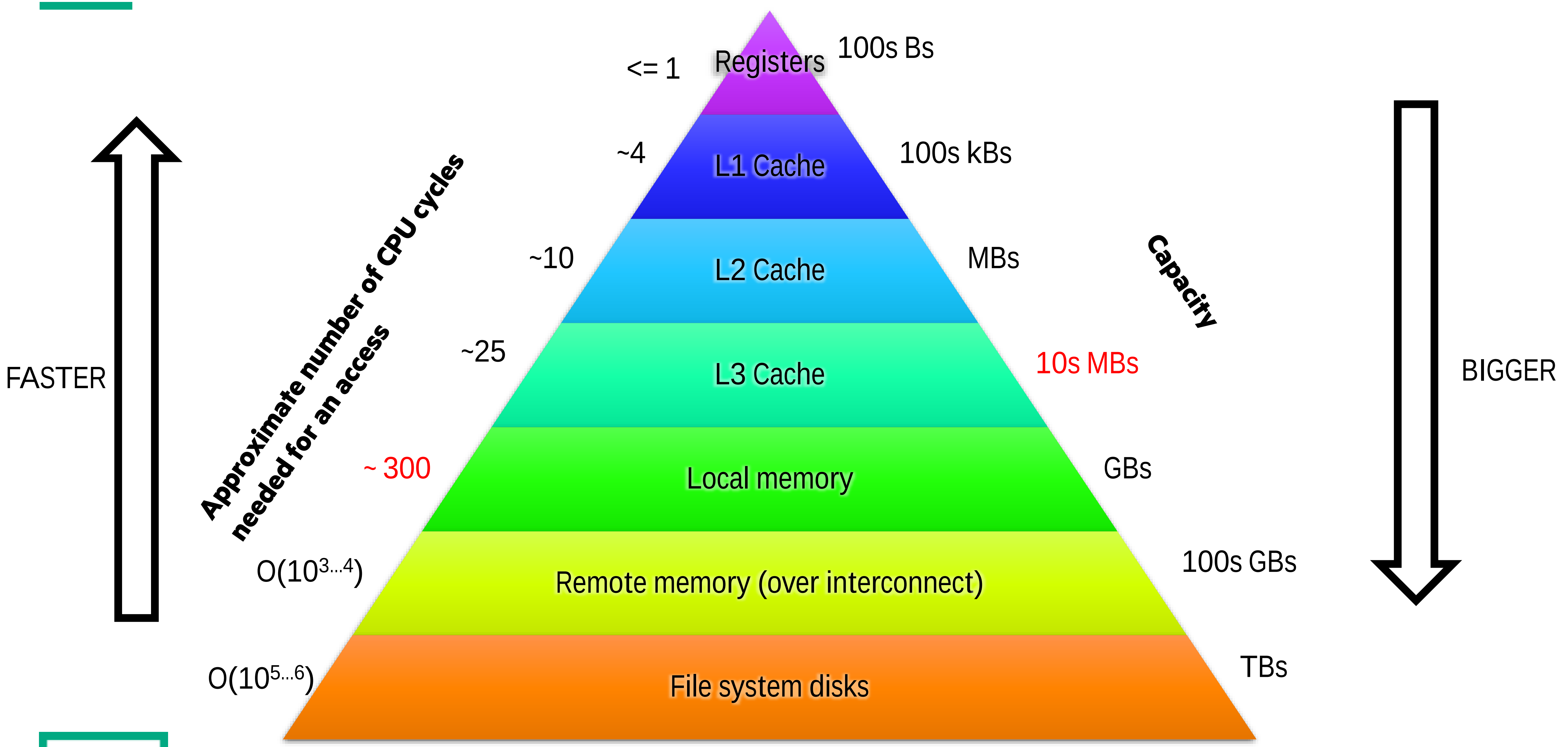
- Processors have seen numerous evolutions in the last few years
  - (slight) increase in frequency, wider vector registers, execution pipelines
  - Imbalance between computational and the memory performance (memory wall)

- Important for programmers
  - Optimize performance-critical parts and be sure to use representative test-cases and problem sizes
  - Optimize data movement
  - Help the compiler generate efficient instruction sequences

Doesn’t the Compiler do Everything?

- Not yet...
  - Standard answer, unchanged for last 50 or so years
- You can make a big difference to code performance
  - Helping the compiler spot optimization opportunities
  - Using the insight of your application
  - Removing obscure (and obsolescent) “optimizations” in older code
    - Simple code is the best, until proven otherwise
- No fixed rules: optimize on case-by-case basis
  - But first, check what the compiler is already doing
- What we cover in this talk:
  - Loop optimization: Loop Unrolling, Strip Mining
  - Cache optimization: Cache Blocking, Data Alignment
  - Vectorization: compiler hints

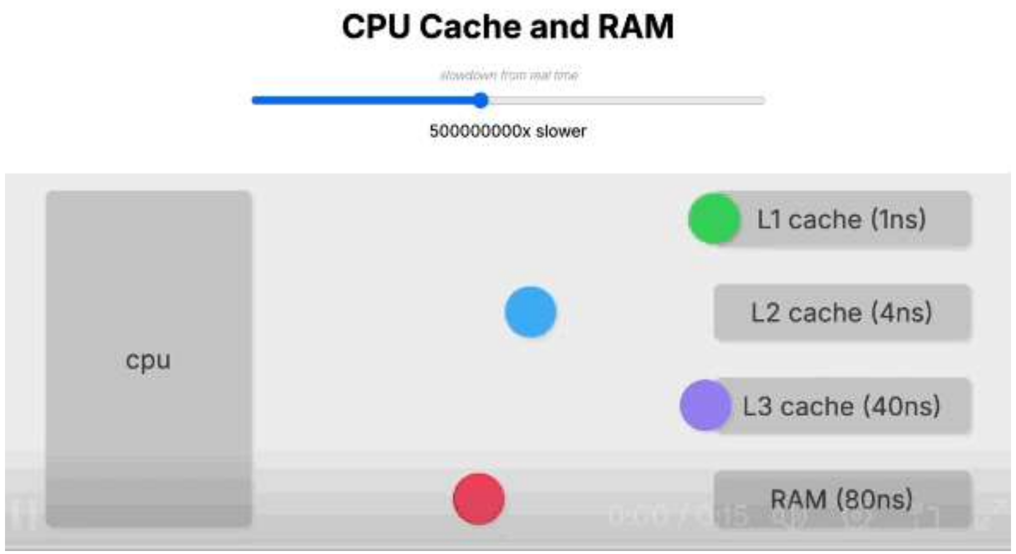
Keep Your Friends Close and Data Even Closer



System Latencies

Table 2.2 Example Time Scale of System Latencies

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 μs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia



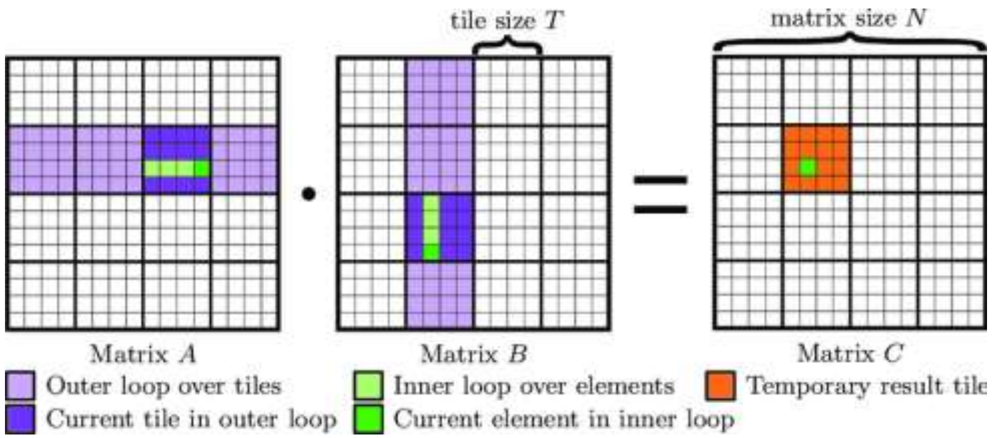
"Systems Performance: Enterprise and the Cloud" by Brendan Gregg

https://benjdd.com/

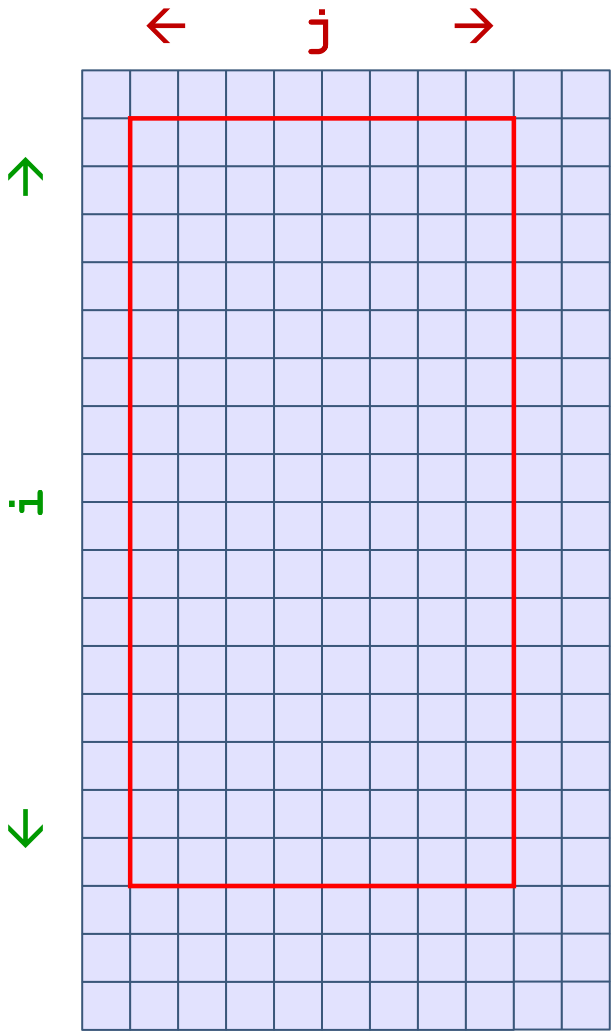
DATA LOCALITY

Cache Blocking

- Cache blocking is a combination of strip mining and loop interchange, designed to increase data reuse.
  - Takes advantage of temporal locality: re-reference array elements already referenced
  - Good blocking will take advantage of spatial locality: work with the cache lines!
- Many ways to block any given loop nest
  - Which loops get blocked?
  - What block size(s) to use?
- Analysis can reveal which ways are beneficial
  - How big is your cache?
    – L1 is 32 KB
  - How many cache lines can it hold?
    – each line typically 64B, so
  - How many cache lines are needed per loop iteration?
  - ...
- But trial-and-error is probably faster
  - or autotuning of the code



Cache Blocking



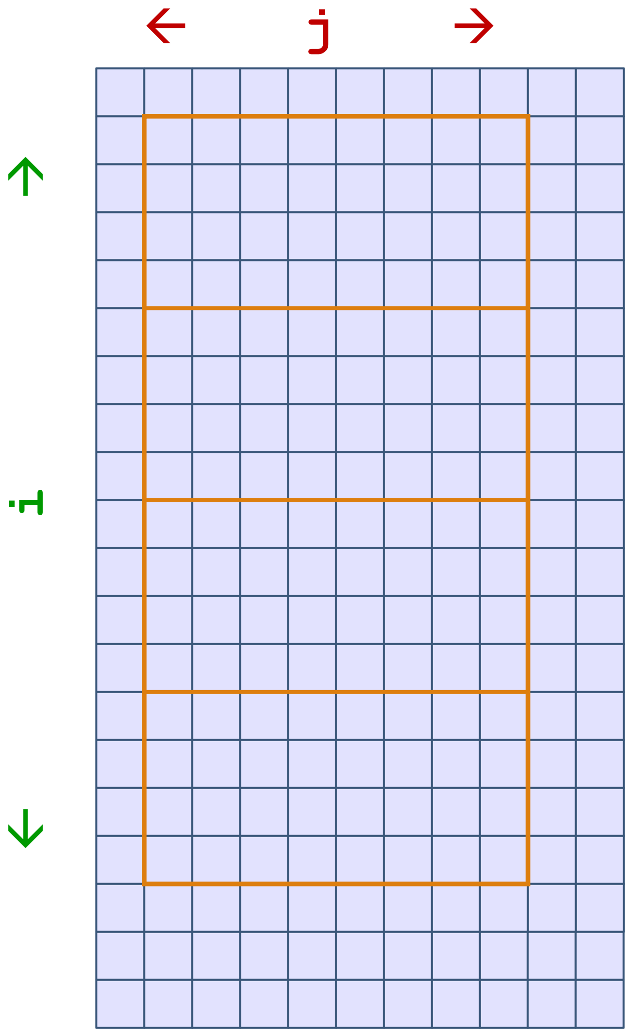
Cache line counter: 120

```
DO j = 1, 8
   DO i = 1, 16
      a = u(i-1,j) + u(i+1,j) &
        + u(i,j-1) + u(i,j+1) &
        - 4*u(i,j)
```

(column-major)

- Imagine a CPU architecture where:
  - each cache line holds 4 array elements
  - cache can hold 12 lines of data (48 cells)
  - Each execution of i-loop needs:
    - 3*CEILING[(16+2)/4]=15 cache lines
- No cache reuse b/w j-loop iterations
  - Because 15 is greater than 12
    - iteration j loaded `u(i:i+3,j+1)` (4 elements)
    - iteration j+1 could reuse this (for central term)
    - but it's already been evicted from the cache
- Cache misses per loopnest iteration
  - 8*15 = 120

Blocking to Increase Reuse



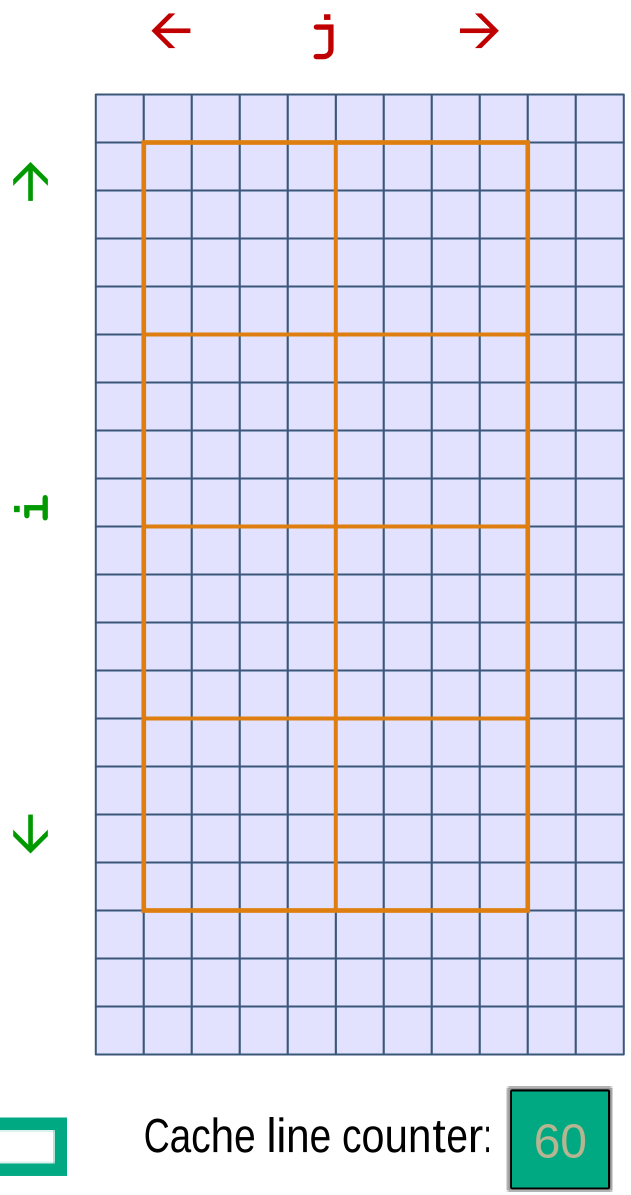
- Block the inner loop

```
DO ib = 1, 16, 4
   DO j = 1, 8
      DO i = ib, ib + 4-1
         a = u(i-1,j) + u(i+1,j) &
           + u(i,j-1) + u(i,j+1) &
           - 4*u(i,j)
```

- Cache lines per j-loop iteration:
  - 3*CEILING[(4+2)/4]=6 cache lines
  - So can hold data for 4 j-values in cache
  - because (4+2)*CEILING[(4+2)/4]=12
- Cache liner per ib-loop iteration
  - (8+2)*CEILING[(4+2)/4]=20
- Cache misses per loopnest iteration
  - (16/4)*20=80
  - reduced from 120
- Better temporal locality

Cache line counter: 80

Cache Blocking



- Iterate over 4×4 blocks (or "tiles")

```
DO jb = 1, 8, 4
   DO ib = 1, 16, 4
      DO j = jb, jb + 4-1
         DO i = ib, ib + 4-1
            a = u(i-1,j) + u(i+1,j) &
              + u(i,j-1) + u(i,j+1) &
              - 4*u(i,j)
```

- Cache lines per tile:
  - (4+2)*CEILING[(4+2)/4]=12
  - Can reuse for (some of) next tile
- Cache lines for each jb-iteration
  - (4+2)*CEILING[(16+2)/4]=30
- Cache misses per loopnest iteration
  - (8/4)*30=60
  - reduced from 80
    - which was reduced from 120
- Better spatial locality

Cache Blocking with Cray Directives

- **Don’t modify your code to do explicit blocking**
- CCE blocks well
- Get the loopmark listing
  - Identifies which loops were blocked
  - Gives the block size the compiler chose
- **...but sometimes CCE blocks better with help**

<table>
<tr><th>Original loopnest</th><th>Loopnest with help</th><th>Equivalent explicit code</th></tr>
<tr>
<td><pre>do k = 6, nz-5
 do j = 6, ny-5
  do i = 6, nx-5
   ! stencil
  enddo
 enddo
enddo</pre></td>
<td><pre>!dir$ blockable(j,k)
!dir$ blockingsize(16)
do k = 6, nz-5
 do j = 6, ny-5
  do i = 6, nx-5
   ! stencil
  enddo
 enddo
Enddo</pre></td>
<td><pre>do kb = 6,nz-5,16
 do jb = 6,ny-5,16
  do k = kb,MIN(kb+16-1,nz-5)
   do j = jb,MIN(jb+16-1,ny-5)
    do i = 6, nx-5
     ! stencil
    enddo
   enddo
  enddo
 enddo
enddo</pre></td>
</tr>
</table>

Further Cache Optimizations: loop fusion

- If multiple loopnests process a large array
  - First element of array will be out of cache when start second loopnest
- Improving cache use
  - Consider fusing the loopnests
    - Completely: just have one loopnest
    - Partial: have one outer loop, containing multiple inner loops

<table>
<tr><th>Original code</th><th>Complete fusion</th><th>Partial fusing</th></tr>
<tr><td><pre>
do j = 1, Nj
 do i = 1, Ni
  a(i,j)=b(i,j)*2
 enddo
enddo

do j = 1, Nj
 do i = 1, Ni
  a(i,j)=a(i,j)+1
 enddo
enddo
</pre></td><td><pre>
do j = 1, Nj
 do i = 1, Ni
  a(i,j)=b(i,j)*2
  a(i,j)=a(i,j)+1
 enddo
enddo
</pre></td><td><pre>
do j = 1, Nj
 do i = 1, Ni
  a(i,j)=b(i,j)*2
 enddo
 do i = 1, Ni
  a(i,j)=a(i,j)+1
 enddo
enddo
</pre></td></tr>
</table>

Further Cache Optimizations: function fusion

Original code	After changes

`CALL sub1(a,b)`
`CALL sub2(a)`
 	
`SUBROUTINE sub1(a,b)`
`  do j=1,Nj`
`    do i=1,Ni`
`      a(i,j)=b(i,j)*2`
`    enddo`
`  enddo`
`END SUBROUTINE sub1`
	`do j = 1, Nj`
	`  CALL sub1(a,b,j)`
	`  CALL sub2(a,j)`
	`enddo`
	
	`SUBROUTINE sub1(a,b,j)`
	`  do i=1,Ni`
	`    a(i,j)=b(i,j)*2`
	`  enddo`
	`END SUBROUTINE sub1`

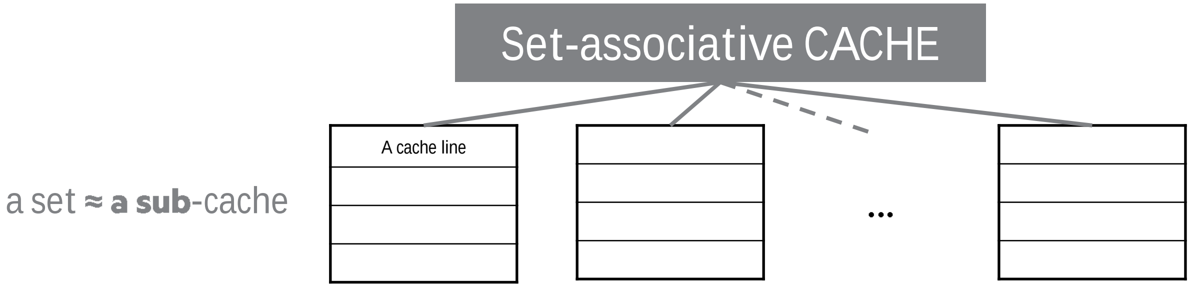
- Perhaps cache block before fusing
  - Fuse one or more of the outer blocking loops
- If multiple subprograms process the array
  - Remove one or more outer loops (or all loops) from subprograms
  - Haul loop into parent routine, pass in index values instead
  - Might want to ensure that compiler is inlining this routine
  - This technique is useful if you want to use OpenMP/OpenACC
- Fortran note
  - With array syntax

```
a(:,:)=b(:,:)*2
a(:,:)=a(:,:)+1
```

compiler unlikely
to fuse any loops

More on Increasing Reuse of Data in Cache

Reducing data cache conflict between arrays accessed in the same loop



- Adding pad arrays to align data in caches (changing memory layout)

Original code	With padding arrays

`REAL(8) A(64,64), B(64,64)`
`REAL(8) C(64,64)`
 	
`DO i=1,64`
`  C(i,1)= A(i,1)+b(i,1)`
`END DO`
	`REAL(8) A(64,64), pad1(16), B(64,64)`
	`REAL(8) pad2(16), C(64,64)`
	
	`DO i=1,64`
	`  C(i,1)= A(i,1)+b(i,1)`
	`END DO`

- Usually compiler does automatic padding, but you may be able to get better results adding your own pad arrays.

Cache Stats

CrayPAT profiling with `export PAT_RT_HWPC=2` (L1 and L2 metrics)

```
Time%                                          0.2%
Time                                       0.000003
Calls                                             1
PAPI_L1_DCA              455.433M/sec          1367 ops
DC_L2_REFILL_MOESI        49.641M/sec           149 ops
DC_SYS_REFILL_MOESI        0.666M/sec             2 ops
BU_L2_REQ_DC              74.628M/sec           224 req
User time                  0.000 secs          7804 cycles
Utilization rate                              97.9%
L1 Data cache misses      50.308M/sec           151 misses
LD & ST per D1 miss                            9.05 ops/miss
D1 cache hit ratio                            89.0%
LD & ST per D2 miss                          683.50 ops/miss
D2 cache hit ratio                            99.1%
L2 cache hit ratio                            98.7%
Memory to D1 refill        0.666M/sec             2 lines
Memory to D1 bandwidth    40.669MB/sec          128 bytes
L2 to Dcache bandwidth  3029.859MB/sec         9536 bytes
```


Better Cache Locality

Time%		0.1%	
Time		0.000002	
Calls		1	
PAPI_L1_DCA	689.986M/sec	1333	ops
DC_L2_REFILL_MOESI	33.645M/sec	65	ops
DC_SYS_REFILL_MOESI		0	ops
BU_L2_REQ_DC	34.163M/sec	66	req
User time	0.000 secs	5023	cycles
Utilization rate		95.1%	
L1 Data cache misses	33.645M/sec	65	misses
LD & ST per D1 miss		20.51	ops/miss
D1 cache hit ratio		95.1%	
LD & ST per D2 miss		1333.00	ops/miss
D2 cache hit ratio		100.0%	
L2 cache hit ratio		100.0%	
Memory to D1 refill		0	lines
Memory to D1 bandwidth		0	bytes
L2 to Dcache bandwidth	2053.542MB/sec	4160	bytes

VECTORIZATION (DATA LEVEL PARALLELISM)

Single Instruction, Multiple Data (SIMD):
CPU instructions (SSE, AVX)... but also GPUs.

When does the Compiler Vectorize

- What can be vectorized
  - Mostly loops (+ SLP vectorization a.k.a. superword-level parallelism)
- Usually only one loop is vectorizable in loopnest
  - And most compilers only consider inner loop
- Optimizing code to allow compilers to use vector instructions
  - Relies on code being vectorizable
  - Or in a form that the compiler can convert to be vectorizable
    - Some compilers are better at this than others
- Check the compiler output listing and/or assembler listing
  - Look for packed SSE/AVX instructions

LOOP UNROLLING

Loop Unrolling (Theory)

Original code	After partial unrolling

`do i=1,N`
` a(i)=a(i) + b(i)`
`enddo`
	`do i=1,N,4`
	` a(i)  =a(i)   + b(i)`
	` a(i+1)=a(i+1) + b(i+1)`
	` a(i+2)=a(i+2) + b(i+2)`
	` a(i+3)=a(i+3) + b(i+3)`
	`enddo`
	`<cleanup if N%4!=0>`

- Increases the work per loop iteration
  - higher computational intensity
    - more floating-point operations per memory operation (load or store)
  - more computation per iteration
    - can pipeline better in CPU
    - more opportunities for vectorization
- Combination of loop blocking and unwinding
- may completely unwind the loop

Loop Unrolling (Reality)

- Most compilers will unroll loops automatically
  - But probably will concentrate on inner loops
- When might we help?
  - When the compiler didn't unroll and should have done so
  - When the compiler doesn't know about tripcounts (this loop is small)
  - When we have a small outer loop
    - Maybe move it to be the innermost loop and completely unroll
- <u>Avoid</u> unrolling loops manually
  - Reduces portability
  - Compiler will likely reroll the loop and unroll it again
- Instead help compiler to do it using <u>directives</u>, e.g.:
  - CCE specify loop count = n if it can be estimated

    ```
    !DIR$ loop_info n
    ```
  - CCE: Force unrolling loop, optional `i` times.

    ```
    !DIR$ unroll (i)
    #pragma unroll i
    ```

COMPILER AND VECTORIZATION

Helping Vectorization

- Is there a good reason for this?
  - There is an overhead in setting up vectorization; maybe it's not worth it
    - Could you unroll inner (or outer) loop to provide more work?
- Does the loop have dependencies?
  - information carried between iterations
    - e.g., counter: **`total = total + a(i)`**
  - No:
    - Tell the compiler that it is safe to vectorize
      - `!dir$ IVDEP` directive above loop (CCE, but works with most compilers)
      - C99: restrict keyword
  - Yes:
    - Rewrite code to use algorithm without dependencies, e.g.
      - promote loop scalars to vectors (single dimension array)
      - use calculated values (based on loop index) rather than iterated counters, e.g.
        - Replace: **`count = count + 2; a(count) = ...`**
        - By: **`a(2*i) = ...`**
      - move **`if`** statements outside the inner loop
        - may need temporary vectors to do this (otherwise use masking operations)
    - If you need to do too much extra work to vectorize, may not be worth it.

Code Restructuring Example

```
65.      1                PF = 0.0
66.    + 1 2-----<        DO 44030 I = 2, N
67.      1 2               AV    = B(I) * RV
68.      1 2               PB    = PF
69.      1 2               PF    = C(I)
70.      1 2               IF ((D(I) + D(I+1)) .LT. 0.) PF = -C(I+1)
71.      1 2               AA    = E(I) - E(I-1) + F(I) - F(I-1)
72.      1 2              1      + G(I) + G(I-1) - H(I) - H(I-1)
73.      1 2               BB    = R(I) + S(I-1) + T(I) + T(I-1)
74.      1 2              1      - U(I) - U(I-1) + V(I) + V(I-1)
75.      1 2              2      - W(I) + W(I-1) - X(I) + X(I-1)
76.      1 2               A(I) = AV * (AA + BB + PF - PB + Y(I) - Z(I)) + A(I)
77.      1 2-----> 44030 CONTINUE
```

Loop-carried dependency

```
ftn-6254 ftn: VECTOR LP44030, File = lp44030.f, Line = 66
  A loop starting at line 66 was not vectorized because a recurrence was found on
"pf" at line 69.
```

Code Restructuring Example

```
101.     1              VPF(1) = 0.0
102.     1 Vr2---<      DO 44031 I = 2, N
103.     1 Vr2           AV     = B(I) * RV
104.     1 Vr2           VPF(I) = C(I)
105.     1 Vr2           IF ((D(I) + D(I+1)) .LT. 0.) VPF(I) = -C(I+1)
106.     1 Vr2           AA    = E(I) - E(I-1) + F(I) - F(I-1)
107.     1 Vr2         1       + G(I) + G(I-1) - H(I) - H(I-1)
108.     1 Vr2           BB    = R(I) + S(I-1) + T(I) + T(I-1)
109.     1 Vr2         1       - U(I) - U(I-1) + V(I) + V(I-1)
110.     1 Vr2         2       - W(I) + W(I-1) - X(I) + X(I-1)
111.     1 Vr2           A(I) = AV * (AA + BB + VPF(I) - VPF(I-1) + Y(I) - Z(I)) + A(I)
112.     1 Vr2---> 44031 CONTINUE
```

Promote to vector

```
ftn-6005 ftn: SCALAR LP44030, File = lp44030.f, Line = 102
  A loop starting at line 102 was unrolled 2 times.

ftn-6204 ftn: VECTOR LP44030, File = lp44030.f, Line = 102
  A loop starting at line 102 was vectorized.
```

Quick check:
2.2x – 3x faster

When does the Cray Compiler Vectorize?

- The Cray compiler vectorizes loops
  - Constant strides are best, indirect addressing is bad
    - Scatter/gather operations (not efficiently implemented in AVX2)
  - Can vectorize across inlined functions
  - Needs to know loop tripcount (but only at runtime)
    - i.e., Do while -style loops will not vectorize
  - No recursion allowed
    - if you have this, consider rewriting the loop
  - If you can't vectorize the entire loop, consider splitting it
  - Think about cache line : you may need to use new structures (structure of arrays, avoid array of structures)
- Check the compiler output to see what it did
  - CCE Fortran: `-hlist=a` `-fsave-loopmark`
  - CCE C/C++: `-fsave-loopmark`
- Clues from CrayPAT's HWPC measurements
  - `export PAT_RT_HWPC=13` or 14 # Floating point operations SP,DP
  - Complicated, but look for ratio of operations/instructions > 1
    - expect 8 for pure AVX2 with double precision floats

CONCLUSION

CCE Fortran Directives

The Cray compiler supports a full and growing set of directives and pragmas

Fortran Cheat Sheet:

- !dir$ concurrent
- !dir$ ivdep
- !dir$ interchange
- !dir$ unroll
- !dir$ loop_info [max_trips] [cache_na] ... Many more
- !dir$ blockable

More info:

- `man directives` (only Fortran part)
- `man loop_info`

OpenMP is adding similar controls:

For example, OpenMP 5.1
Loop Transformations (tile/unroll)

Concluding Remarks

- Understand what the compiler does
  - Use hardware counters and profiling
  - Ask for the compiler feedback

- And remember what we said at the start
  - You should only attempt to optimize performance-critical parts and be sure to use representative test-cases and problem sizes

- Help the compiler to understand your code
  - Simple constructs, most basic optimizations usually better to leave for the compiler
  - Avoid branches in loops
  - Avoid indirect addressing
  - Avoid non-constant striding in loops

Questions?