



EPSRC

ARCHER Single Node Optimisation

Optimising for the Memory Hierarchy

Slides contributed by Cray and EPCC



Overview

- Motivation
- Types of memory structures
- Reducing memory accesses
- Utilizing Caches
- Prefetching
- Pointer aliasing



Motivation

- Why is memory structure important?
 - With current hardware memory access has become the most significant resource impacting program performance.
 - Changing memory structures can have a big impact on code performance.
 - Memory structures are frequently global to the program
 - Different code sections communicate via memory structures.
 - The programming cost of changing a memory structure can be very high.



Programmer's perspective:

- Memory structures are the programmers responsibility
 - At best the compiler can add small amounts of padding in limited circumstances.
 - Compilers can (and hopefully will) try to make best use of the memory structures that you specify (e.g. uni-modular transformations)
- Changing the memory structures you specify may allow the compiler to generate better code.



Types of data structure

- Arrays
- Pointer arrays
- records/structures
- Trees and lists
- Objects



Arrays

- Arrays are large blocks of memory indexed by integer index
- Probably the most common data structure used in HPC codes
- Good for representing regularly discretised versions of dense continuous data

$$f(x,y,z) \rightarrow F[i][j][k]$$



Arrays

- Multi dimensional arrays use multiple indexes (shorthand)

```
REAL A(100,100,100)
```

```
A (i,j,k) = 7.0
```

```
REAL A(1000000)
```

```
A(i+100*j+10000*k) = 7.0
```

```
float A[100][100][100];
```

```
A [i][j][k] = 7.0
```

```
float A[1000000];
```

```
A(k+100*j+10000*i) = 7.0
```

- Address calculation requires computation but still relatively cheap.
- Compilers have better chance to optimise where dimension sizes are known at compile time.



Arrays

- Many codes loop over array elements
 - Data access pattern is regular and easy to predict
- Good spatial locality achieved by accessing neighbouring elements on consecutive iterations of the innermost loop.
- Unless loop nest order and array index order match the access pattern may not be optimal for cache re-use.
 - Compiler can potentially address these problems by transforming the loops.
 - But often can do a better job when provided with a more cache-friendly index order.



Bad spatial locality

```
do i=1,n
  do j=1,m
    a(i,j)=a(i,j)+b(i,j)
  end do
end do
```

```
for (j=0;j<M;j++) {
  for (i=0;i<N;i++) {
    a[i][j]+=b[i][j];
  }
}
```

Good spatial locality

```
do j=1,m
  do i=1,m
    a(i,j)=a(i,j)+b(i,j)
  end do
end do
```

```
for (i=0;i<N;i++) {
  for (j=0;j<M;j++) {
    a[i][j]+=b[i][j];
  }
}
```



Dynamic sized arrays (Fortran)

- Not always possible/desirable to fix array sizes at compile time
 - Fortran allows arrays to be dynamically sized based on subroutine arguments.
- Address calculation can still be optimised using CSE.
- Size of slowest moving index is not needed in address computation.
 - Fortran actually allows this dimension to be unspecified in subroutine arguments (assumed size arrays)



Dynamic sized arrays (C)

- C requires array dimensions to be known at compile time.
- However can make slowest dimension variable with pointers and **typedef**

```
typedef float Mat[2][2];
Mat *data =(Mat *) malloc(n*sizeof(Mat));
for(i=0;i<n;i++){
    for(j=0;j<2;j++){
        for(k=0;k<2;k++){
            data[i][j][k] = 12.0;
        }
    }
}
```



Pointer arrays

- Alternative to multi-dimensional arrays
 - Pointer to: array of pointers to: array of pointers to: Data

```
float ***data;
data = (float ***) malloc(2*sizeof(float **));
for(i=0;i<2;i++){
    data[i]=(float **) malloc(2*sizeof(float *));
    for(j=0;j<2;j++){
        data[i][j] = (float *) malloc(n*sizeof(float));
        for(k=0;k<n;k++){
            data[i][j][k] = 12.0;
        }
    }
}
```

- Note reverse index order to previous example!



Pointer arrays II

- In C the use-syntax is the same as for arrays
 - $a[l][j][k] = 7.0;$
 - But actually equivalent to
 - $p1 = a[l]$
 - $p2 = p1[j]$
 - $p2[k] = 7.0$
- Advantage
 - The “columns” are allocated separately and need not be the same length
- Disadvantages
 - Need multiple memory accesses per element access.
 - Need more memory to store all the pointers
 - Less regular access pattern
 - Messy to create/destroy



Records/structures

- Collection of values (of varying types)
 - C structs
 - F90 user defined types
- Good for representing multi-valued data or sparse/scattered data.
- Related variables are stored close together may help cache use.
 - If a code section only uses a subset of the values cache use may suffer.
- Easy to add/re-order members without breaking code as members are referenced by name not position.
 - much harder to remove them.



Structures and the compiler

- Programmer only specifies what a structure contains.
- Compiler chooses layout within the structure.
- In C the compiler usually preserves the order of members but inserts padding between members if needed to meet alignment constraints
 - i.e. Doubles must be aligned on double-word boundaries.
 - Padding reduces cache-line utilisation so order members to reduce padding.
- Similarly in Fortran but can use SEQUENCE keyword to force deterministic layout.



Arrays of structs or structs of arrays?

```
struct Part{  
    double x;  
    double y;  
    double z;  
    int index;  
    double mass;  
}  
Part data[numParts];
```

Array of structs

or

```
struct AllParts{  
    double x[numParts];  
    double y[numParts];  
    double z[numParts];  
    int index[numParts];  
    double mass[numParts];  
}  
AllParts data;
```

Struct of arrays



Array of structs:

- May have good temporal locality if there is lots of computation on each struct
- May have poor spatial locality if computations don't
- Unfavourable for vector loads/stores
- Natural for OO design

Struct of arrays

- May have better spatial locality (use all data on cache line), but worse temporal locality
- More favourable for vector loads/stores
- Less natural for OO design



Arrays of structs of (short) arrays

```
struct FourVecParts{
    double x[4];
    double y[4];
    double z[4];
    int index[4];
    double mass[4];
}
FourVecParts data[ (numParts+3) / 4 ] ;
```

- Vector friendly without compromising temporal locality too much?
- Not at all natural from a design perspective!



Objects

- Usually implemented much the same as structures
- But objects are opaque
 - Language restricts access to the internal data.
 - Usually need to use special access functions.
- Much easier to change underlying data structure as this is only visible to small fraction of the program
- Access functions introduce additional overhead
 - Function calls
 - Memory copies
- Really only a problem for small low-level objects



Trees/lists

- Structures/Objects can contain pointers to other structures.
 - Can construct trees and lists etc.
- Very flexible and can grow dynamically
 - Same problems as pointer arrays.
 - Additional memory accesses to navigate data
 - Additional storage to store pointers
 - Access pattern is very hard to predict.
- Limited navigation
 - Can only follow access pattern supported by pointer structure
 - e.g. cannot jump to middle of a list without traversing half the nodes.



High level data structures

- Many modern languages have built in-support for high level data structures such as
 - Lists
 - Trees
 - Sets
 - Maps
 - Etc.
- May be available either as built-in data-types or as standard libraries.
 - Have the same intrinsic advantages/disadvantages as home made equivalents but typically better tested and optimised.



What can go wrong

- Poor cache/page use
 - Lack of spatial locality
 - Lack of temporal locality
- Unnecessary memory accesses
 - pointer chasing
 - array temporaries
- Aliasing problems
 - Use of pointers can inhibit code optimisation



Reducing memory accesses

- Memory accesses are often the most important limiting factor for code performance.
 - Many older codes were written when memory access was relatively cheap.
- Things to look for:
 - Unnecessary pointer chasing
 - pointer arrays that could be simple arrays
 - linked lists that could be arrays.
 - Unnecessary temporary arrays.
 - Tables of values that would be cheap to re-calculate.



Utilizing caches

- Want to avoid cache conflicts
 - This happens when too much related data maps to the same cache set.
 - Arrays or array dimensions proportional to (cache-size/set-size) can cause this.
 - Rarely a problem with 8- and 16-way associative caches on XC30
 - Lots of accesses in a loop to arrays with power-of-2 dimensions might still be bad
 - Can pad arrays to avoid this.



Utilizing caches II

- Want to use all of the data in a cache line
 - loading unwanted values is a waste of memory bandwidth.
 - structures are good for this
 - Or loop fastest over the corresponding index of an array.
- Place variables that are used together close together
 - Also have to worry about alignment with cache block boundaries.
- Avoid “gaps” in structures
 - In C structures may contain gaps to ensure the address of each variable is aligned with its size.



Bad Cache Alignment

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

cf: 8



Good Cache Alignment

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



Cache blocking

- A combination of:
 - strip mining (also called loop blocking, loop tiling...)
 - loop interchange
- Designed to increase data reuse:
 - temporal reuse: reuse array elements already referenced
 - spatial reuse: good use of cache lines
- Many ways to block any given loop nest
 - Which loops should be blocked?
 - What block size(s) will work best?



- Analysis can reveal which ways are beneficial
 - How big is your cache?
 - L1 is 32kB on Ivybridge.
 - 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

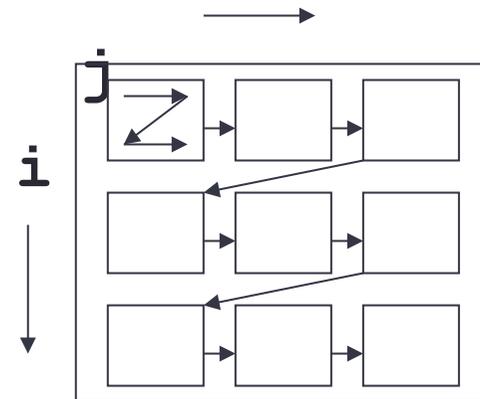
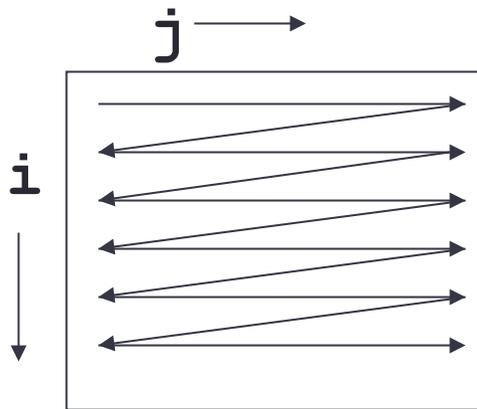


Loop tiling

```
for (i=0;i<n;i++){  
  for (j=0;j<n;j++){  
    a[i][j]=b[j][i];  
  }  
}
```



```
for (ii=0;ii<n;ii+=B){  
  for (jj=0;jj<n;jj+=B){  
    for (i=ii;i<ii+B;i++){  
      for (j=jj;j<jj+B;j++){  
        a[i][j]=b[j][i];  
      }  
    }  
  }  
}
```



Loop tiling for vectorisation

```
for (i=0;i<n;i++){
  for (j=1;j<n-1;j++){
    a[i][j]=(a[i][j-1] + a[i][j+1])/2.0 ;
  }
}
```

j loop won't vectorise due to dependencies

```
for (ii=0;ii<n;ii+=B){
  for (j=1;j<n-1;j++){
    for (i=ii;i<ii+B;i++){
      a[j][i]=(a[j-1][i] + a[j+1][i])/2.0 ;
    }
  }
}
```

i loop will vectorise
but note change of data layout



Further cache optimisations

- If multiple loop nests process a large array
 - First element of array will be out of cache when second loop nest starts
- Improving cache use
 - Consider fusing the loop nests
 - Completely: just have one loop nest
 - Partial: have one outer loop, containing multiple inner loops
 - Beware that too much fusion can result in lots of temporaries and cause the compiler to run out of registers....



Original code	Complete fusion	Partial fusing
<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>	<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>	<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>



Further cache optimisations

- 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 very useful if you want to use OpenMP/OpenACC
- Beware of Fortran
 - array syntax often bad
 - $a(:, :) = b(:, :) * 2$
 - $a(:, :) = a(:, :) + 1$
 - compiler unlikely to fuse any loops



Original code

```
CALL sub1(a,b)
CALL sub2(a)

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

After hauling

```
do j = 1, Nj
  CALL sub1(a,b,j)
  CALL sub2(a,j)
enddo

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



Optimising for TLB

- Aim to reuse data on a page
 - i.e. treat similarly to a cache
- Standard-sized pages are 4kB
 - But you can use larger "huge" pages
 - 128kB, 512kB, 2MB,... 64MB
 - Almost always benefit HPC applications
 - regular data accesses
 - huge pages give fewer TLB misses
 - Huge pages can also help communication performance



- To use huge pages (see `man intro_hugepages`)
 - Load chosen `craype-hugepages*` module
 - See `module avail craype-hugepages` for list of available options
 - 2M or 8M are usually most successful on Cray XC30
 - Compile as before
 - Make sure this module is also loaded in PBS jobscript
 - quick cheat: can load a different-sized hugepages module at runtime
 - compile-time module enables hugepages, runtime one determines actual size



Prefetch

- Some processors (including Ivy Bridge) prefetch automatically
- Regular access patterns are recognized and cache lines fetched in advance.
 - Usually only works for contiguous sequence of cache misses.
- Processor has a set of stream buffers
 - Each holds address of an active stream
 - Loads to the current block causes the next block to be prefetched and the stream address to be updated.
 - Streams are established by series of cache misses to consecutive locations



Using streams

- To utilize stream hardware use linear access patterns where possible
 - Only the order of cache block accesses needs to be linear, not each word access.
- Most loops will require multiple streams
 - If the loop requires more streams than are supported in hardware no prefetching will take place for some of the loads.
 - Consider splitting the loop.
- Prefetching typically cannot cross OS page boundaries
 - huge pages may help



Pointer aliasing

- Pointers are variables containing memory addresses.
 - Pointers are useful but can seriously inhibit code performance.
- Compilers try very hard to reduce memory accesses.
 - Only loading data from memory once.
 - Keep variables in registers and only update memory copy when necessary.
- Pointers could point anywhere, so to be safe compiler will:
 - Reload all values after write through pointer
 - Synchronize all variables with memory before read through pointer



Pointers and Fortran

- F77 had no pointers
- Arguments passed by reference (address)
 - Subroutine arguments are effectively pointers
 - But it is illegal Fortran if two arguments overlap
- F90/F95 has restricted pointers
 - Pointers can only point at variables declared as a “target” or at the target of another pointer
 - Compiler therefore knows more about possible aliasing problems
- Try to avoid F90 pointers for performance critical data structures.



Pointers and C

- In C pointers are unrestricted
 - Can therefore seriously inhibit performance
- Almost impossible to do without pointers
 - malloc requires the use of pointers.
 - Pointers used for call by reference. Alternative is call by value where all data is copied!
- Use the C99 `restrict` keyword where possible
- ...or else use compiler flags
 - CCE: `-h restrict`
 - Intel: `-fnoalias`
 - GNU: ??
- Explicit use of scalar temporaries may also reduce the problem

