PERFORMANCE OPTIMISATION

Adrian Jackson
adrianj@epcc.ed.ac.uk

Hardware design

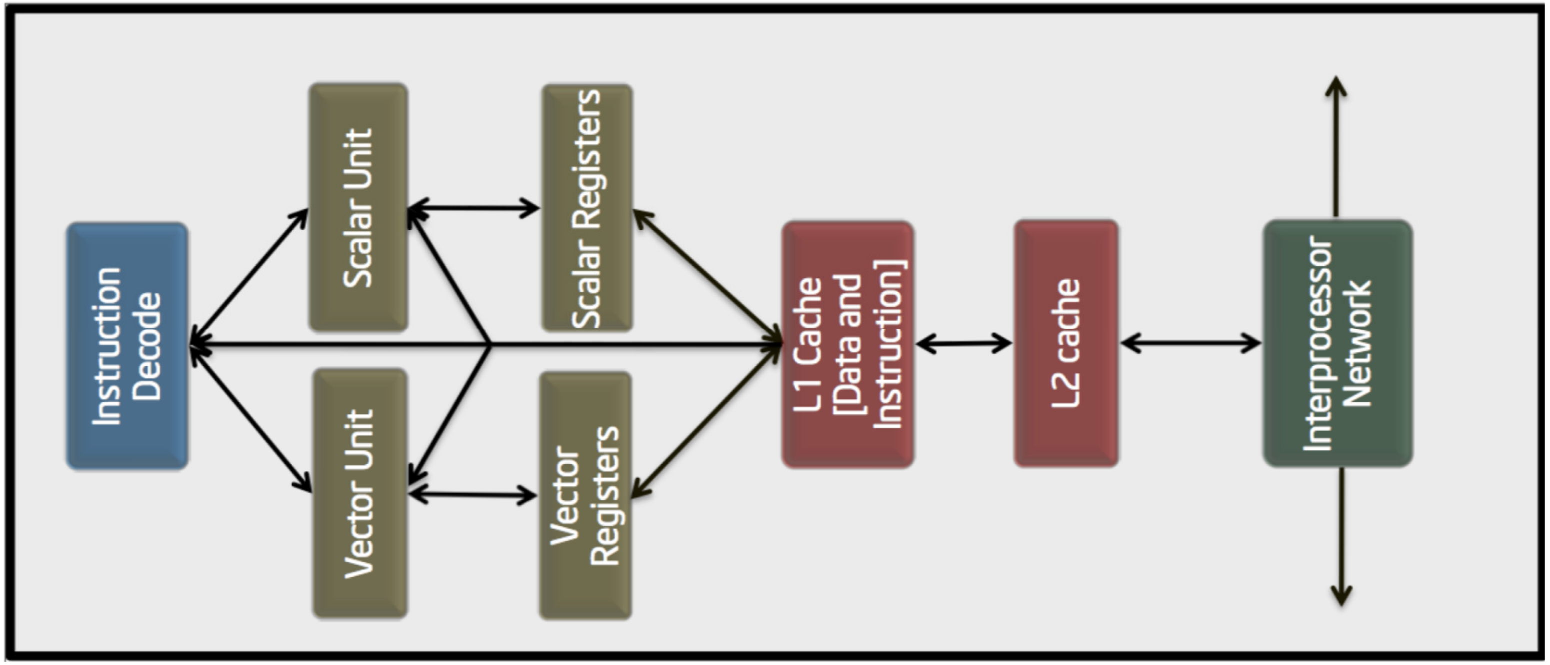


Image from Colfax training material

Pipeline

- Simple five stage pipeline:

1. Instruction fetch
   - get instruction from instruction cache
2. Instruction decode and register fetch
   - can be done in parallel
3. Execution
   - e.g. in ALU or FPU
4. Memory access
5. Write back to register

Hardware issues

Three major problems to overcome:

- Structural hazards
  - two instructions both require the same hardware resource at the same time
- Data hazards
  - one instruction depends on the result of another instruction further down the pipeline
- Control hazards
  - result of instruction changes which instruction to execute next (e.g. branches)

Any of these can result in stopping and restarting the pipeline, and wasting cycles as a result.

Hazards

- Data hazard: result of one instruction (say addition) is required as input to next instruction (say multiplication).
  - This is a read-after-write hazard (RAW) (most common type)
  - can also have WAR (concurrent) and WAW (overwrite problem)
- When a branch is executed, we need to know the result in order to know which instruction to fetch next.
- Branches will stall the pipeline for several cycles
  - almost whole length of time branch takes to execute.
  - Branches account for ~10% of instructions in numeric codes
  - vast majority are conditional
  - ~20% for non-numeric

Locality

- Almost every program exhibits some degree of locality.
  - Tend to reuse recently accessed data and instructions.
- Two types of data locality:

1. Temporal locality

   A recently accessed item is likely to be reused in the near future.

   e.g. if **`x`** is read now, it is likely to be read again, or written, soon.

2. Spatial locality

   Items with nearby addresses tend to be accessed close together in time.

   e.g. if **`y[i]`** is read now, **`y[i+1]`** is likely to be read soon.

Cache

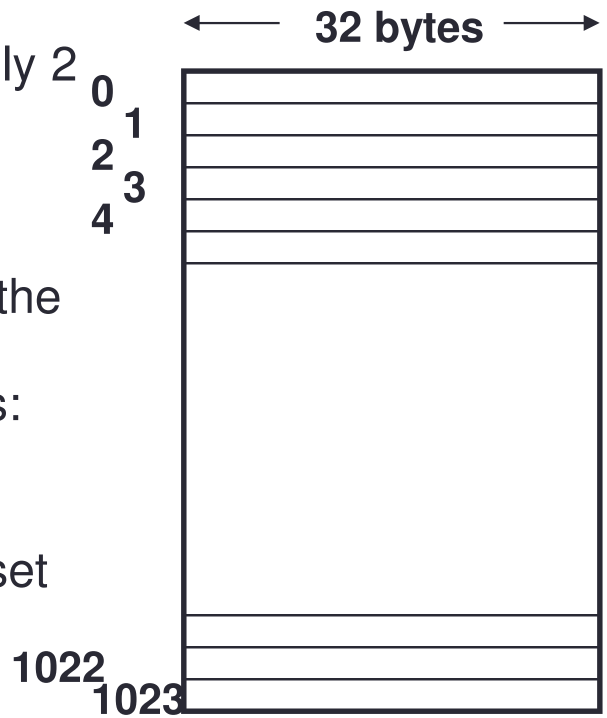
- Cache can hold copies of data from main memory locations.
- Can also hold copies of instructions.
- Cache can hold recently accessed data items for fast re-access.
- Fetching an item from cache is much quicker than fetching from main memory.
  - 3 nanoseconds instead of 100.
- For cost and speed reasons, cache is much smaller than main memory.
- A cache block is the minimum unit of data which can be determined to be present in or absent from the cache.
- Normally a few words long: typically 32 to 128 bytes.
- N.B. a block is sometimes also called a line.

Cache design

- When should a copy of an item be made in the cache?
- Where is a block placed in the cache?
- How is a block found in the cache?
- Which block is replaced after a miss?
- What happens on writes?
- Methods must be simple (hence cheap and fast to implement in hardware).
  - Always cache on reads
  - If a memory location is read and there isn't a copy in the cache (read miss), then cache the data.
  - What happens on writes depends on the write strategy

Cache design cont.

- Cache is organised in blocks.
  - Each block has a number
- Simplest scheme is a direct mapped cache
- Set associativity
  - Cache is divided into sets (group of blocks typically 2 or 4)
  - Data can go into any block in its set.
- Block replacement
  - Direct mapped cache there is no choice: replace the selected block.
  - In set associative caches, two common strategies:
  - Random: Replace a block in the selected set at random
  - Least recently used (LRU): Replace the block in set which was unused for longest time.
  - LRU is better, but harder to implement.



Cache performance

- Average memory access cost =

  hit time + miss ratio x miss time

  - hit time: time to load data from cache to CPU
  - miss ratio: proportion of accesses which cause a miss
  - miss time: time to load data from main memory to cache

- Cache misses can be divided into 3 categories:

Compulsory or cold start

- first ever access to a block causes a miss

Capacity

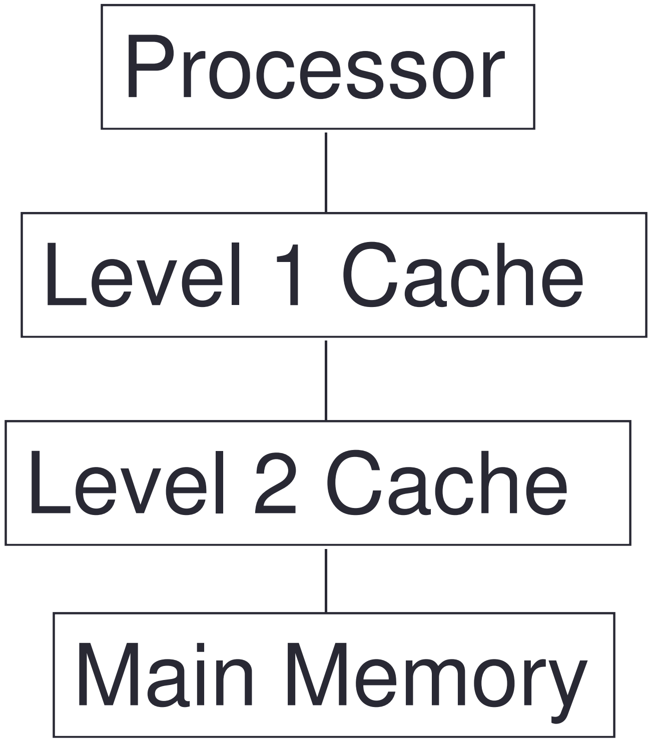
- misses caused because the cache is not large enough to hold all data

Conflict

- misses caused by too many blocks mapping to same set.

Cache levels

- One way to reduce the miss time is to have more than one level of cache.



Cache conflicts

- 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.
- Assume a 1024 word direct mapped cache

```
REAL A(1024), B(1024), C(1024), X
COMMON /DAT/ A,B,C     ! Contiguous
            DO I=1,1024
                  A(I) = B(I) + X*C(I)
            END DO
```

- Corresponding elements map to the same block so each access causes a cache miss.
  - Insert padding in common block to fix this

Conflicts cont.

- Conflicts can also occur within a single array (internal)

```
REAL A(1024,4), B(1024)
DO I=1,1024
       DO J=1,4
              B(I) = B(I) + A(I,J)
       END DO
END DO
```

- Fix by extending array declaration
- Set associated caches reduce the impact of cache conflicts.
- If you have a cache conflict problem you can:
  - Insert padding to remove the conflict
  - change the loop order
  - unwind the loop by cache block size and introduce scalar temporaries to access each block once only
  - permute index order in array (Global edit but can often be automated).

Cache utilisation

- 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 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.

Memory structures

- Why is memory structure important?
  - Memory structures are typically completely defined by the programmer.
    - At best compilers can add small amounts of padding.
    - Any performance impact from memory structures *has* to be addressed by the programmer or the hardware designer.
  - 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 typically global to the program
    - Different code sections communicate via memory structures.
    - The programming cost of changing a memory structure can be very high.

AoS vs SoA

- Array of structures (AoS)
  - Standard programming practise often group together data items in object like way:

```
struct {
   int a; int b; int c;
} struct coord;
coord particles[100];
```

  - Iterating over individual elements of structures will not be cache friendly
- Structure of Arrays (SoA)
  - Alternative is to group together the elements in arrays:

```
struct {
   int a[100]; int b[100]; int c[100];
} struct coords;
coords particles;
```

- Which gives best performance depends on how you use your data
- FORTRAN complex numbers is example of this
  - If you work on real and imaginary parts of complex numbers separately then AoS format is not efficient

Memory problems

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

Arrays

- Arrays are large blocks of memory indexed by integer index
  - Multi dimensional arrays use multiple indexes (shorthand)

```
REAL  A(100,100,100)           REAL A(1000000)
A (i,j,k) =  7.0               A(i+100*j+10000*k) = 7.0
float  A[100][100][100];       float A[1000000];
A [i][j][k] =  7.0             A(k+100*j+10000*i) = 7.0
```

- Address calculation requires computation but still relatively cheap.
- Compilers have better chance to optimise where array bounds are known at compile time.
- Many codes loop over array elements
  - Data access pattern is regular and easy to predict
- Unless loop nest order and array index order match the access pattern may not be optimal for cache re-use.

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.

Vector temporaries

- Old vector code often had many simple loops with intermediate results in temporary arrays

```
REAL V(1024,3), S(1024), U(3)
DO I=1,1024
        S(I) = U(1)*V(I,1)
END DO
DO I=1,1024
        S(I) = S(I) + U(2)*V(I,2)
END DO
DO I=1,1024
        S(I) = S(I) + U(3)*V(I,3)
END DO
DO J=1,3
        DO I=1,1024
                V(I,J) = S(I) * U(J)
        END DO
END DO
```

- Can merge loops and use a scalar

```
REAL V(1024,3), S, U(3)
DO I=1,1024
       S = U(1)*V(I,1) + U(2)*V(I,2) + U(3)*V(I,3)
       DO J=1,3
              V(I,J) = S * U(J)
       END DO
END DO
```

- Vector compilers are good at turning scalars into vector temporaries but the reverse operation is hard.

Problems with writes

- Array initialization
  - Large array initializations may be particularly slow when using write allocate caches.
    - We only want to perform lots of writes to overwrite junk data.
    - The cache will carefully load all the junk data before overwriting it.
    - Especially nasty if the array is sized generously but everything is initialized
  - Work arounds
    - Use special HW features to zero the array (compiler directives).
    - Combine initialization with the first access loop
      - This increases the chance of a programming error so have a debugging options to perform original initialization as well

Prefetching

- Many processors have special prefetch instructions to request data to be loaded into cache.
- Compilers will try to insert these automatically
- For best results will probably need compiler directives to be inserted.
  - Read the compiler manual.
- Write-allocate caches may have instructions to zero cache lines
  - Useful for array initialization
  - Probably need directives again.

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, to be safe:
  - 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!
- Compilers may have #pragma extensions or compiler flags to assert pointers do not overlap
  - Usually not portable between platforms
- Explicit use of scalar temporaries may reduce the problem

Compiler optimisations

- We will consider a set of optimisations which a typical optimising compiler might perform.
- We will illustrate many transformations at the source level.
  - important to remember that compiler is making transformations at IR or assembly level

Programmer's perspective:

These are (largely) optimisations which you would expect a compiler to do, and should very rarely be hand-coded.

Compiler optimisations

- Constant folding
  - Propagate constants through code and insert pre-calculated values if they don't change
- Algebraic simplification
  - Eliminating unnecessary operations
- Copy and constant propagation
  - Replace variables if they are the same
- Redundancy elimination
  - Common subexpression elimination, loop invariant code motion, dead code removal
- Simple loop optimisation
  - Strength reduction (replace computation based on loop variable with increments), induction variable removal (replace with loop variable variant),

Inlining

- Inlining replaces a procedure call with the a copy of the procedure body.
- Can enable other optimisations
    - especially if call is inside a loop
- Benefits must be weighed against:
    - increase in code size (risk of more instruction cache misses)
    - increased register pressure
- Handling complex control flow or static/SAVE variables is a bit tricky.

Loop unrolling

- Loops with small bodies generate small basic blocks of assembly code
  - lot of dependencies between instructions
  - high branch frequency
  - little scope for good instruction scheduling
- Loop unrolling is a technique for increasing the size of the loop body
  - gives more scope for better schedules
  - reduces branch frequency
  - make more independent instructions available for multiple issue.
- Replace loop body by multiple copies of the body
- Modify loop control
  - take care of arbitrary loop bounds
- Number of copies is called unroll factor

Loop unrolling

```
do i=1,n
   a(i)=b(i)+d*c(i)
end do
```

→

```
do i=1,n-3,4
  a(i)=b(i)+d*c(i)
  a(i+1)=b(i+1)+d*c(i+1)
  a(i+2)=b(i+2)+d*c(i+2)
  a(i+3)=b(i+3)+d*c(i+3)
end do
do j = i,n
  a(j)=b(j)+d*c(j)
end do
```

- Choice of unroll factor is important (usually 2,4,8)
  - if factor is too large, can run out of registers
- Cannot unroll loops with complex flow control
  - hard to generate code to jump out of the unrolled version at the right place
- Function calls
  - except in presence of good interprocedural analysis and inlining
- Conditionals
  - especially control transfer out of the loop
- Pointer/array aliasing

epcc THE UNIVERSITY OF EDINBURGH

Outerloop unrolling

- If we have a loop nest, then it is possible to unroll one of the outer loops instead of the innermost one.
- Can improve locality.

```
do i=1,n
  do j=1,m
    a(i,j)=c*d(j)
  end do
end do
```

2 loads for 1 flop

→

```
do i=1,n,4
  do j=1,m
    a(i,j)=c*d(j)
    a(i+1,j)=c*d(j)
    a(i+2,j)=c*d(j)
    a(i+3,j)=c*d(j)
  end do
end do
```

5 loads for 4 flops

Variable expansion

- Variable expansion can help break dependencies in unrolled loops
  - improves scheduling opportunities
- Close connection to reduction variables in parallel loops

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

unroll

```
for (i=0,i<n,i+=2){
   b+=a[i];
   b+=a[i+1];
}
```

expand **b**

```
for (i=0,i<n,i+=2){
   b1+=a[i];
   b2+=a[i+1];
}
b=b1+b2;
```

Divisions

- Division operation is costly (10s of instructions)
- Can often be replaced by a multiplication:

```
do i=1,n
  do j=1,m
    a(i,j)=d(j)/2
  end do
end do
```

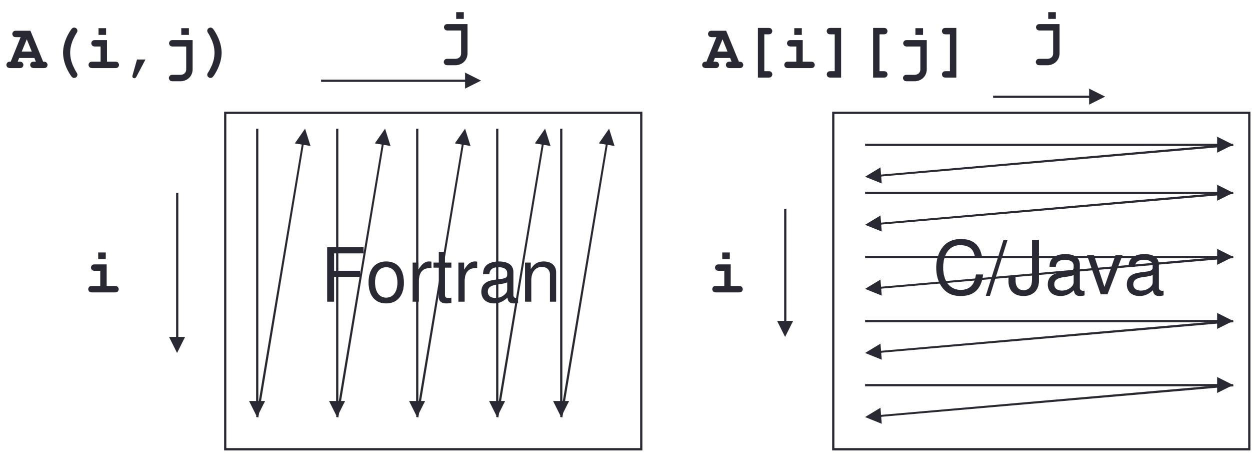
→

```
tempdiv = 1/2
do i=1,n
  do j=1,m
    a(i,j)=d(j)*tempdiv
  end do
end do
```

- Hard for compiler to do this if using floating point numbers (will alter results)

Further optimisations

- These optimisations are not done by all compilers.
- Whereas it is (relatively) easy for a compiler to work out whether a given transformation reduces the number of instructions required, it is much harder for it to predict cache misses.
- You may need to consider implementing this type of optimisation by hand. In a nest of more than one loop, loop order is important for exploiting spatial locality in caches.
- Recall that in Fortran, arrays are laid out by columns, whereas in C (and Java) they are laid out by rows.



Loop interchange

- Loop interchange swaps the loops in a double loop nest
- Can be generalised to reordering loop nests of depth 3 or more
  - loop permutation

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

- Does not traverse memory locations in order
- Poor spatial locality

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

- Traverses memory locations in order
- Good spatial locality

Loop fusion

- If two adjacent loops have the same iteration space, their bodies can be merged (provided dependencies are respected).
- Can improve temporal locality
  - or may reduce the number of memory references required.

```
for (j=0;j<n;j++){
  a[j]+=1;
}
for (i=0;i<n;i++){
  b[i]=a[i]*2;
}
```

→

```
for (j=0;j<n;j++){
  a[j]+=1;
  b[j]=a[j]*2;
}
```

Loop distribution

- Loop distribution is in the inverse of loop fusion
- Can reduce conflict/capacity misses
  - can also reduce register pressure in large loop bodies
- Choosing whether to fuse/distribute can be tricky!

```
for (j=0;j<n;j++){
  a[j]+=1;
  b[j]*=2;
}
```

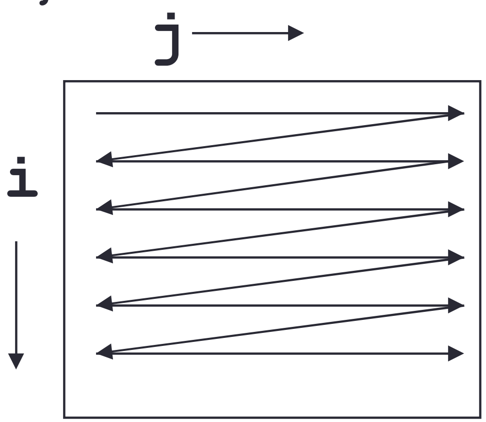
→

```
for (j=0;j<n;j++){
  a[j]+=1;
}
for (j=0;j<n;j++){
  b[j]*=2;
}
```

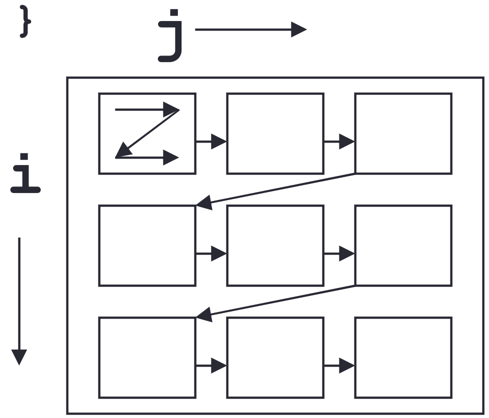
Loop tiling

- Loop tiling increases the depth of a loop nest
- Improves temporal locality by reordering traversal of iteration space into compact blocks.
- Also known as loop blocking, strip mining + interchange, unrolling and jamming.

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




```
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[i][j];
      }
    }
  }
}
```



Array padding

- It is easier to transform loops than arrays
- Loop transforms are purely local in the program
- Array transforms may have effects elsewhere
- Array padding consists of adding additional, unused space between array, or between dimensions of arrays.
- Can reduce conflict misses.

```
float a[2][4096];
for (j=0;j<n;j++){
  a[1][j]+=1;
  a[2][j]*=2;
}
```

→

```
float a[2][4096+64];
for (j=0;j<n;j++){
  a[1][j]+=1;
  a[2][j]*=2;
}
```

Loop tiling and array padding

- Loop tiling is most effective when there is some reuse of data within a tile.
- Need to choose the tile size such that all the data accessed by the tile fits into cache.
  - need to err on the small side, because of potential conflict misses, especially in direct mapped caches.
  - may utilise multiple levels of tiling for multiple levels of cache
- It is easier to transform loops than arrays
- Loop transforms are purely local in the program
- Array transforms may have effects elsewhere
- Array padding consists of adding additional, unused space between array, or between dimensions of arrays.
- Can reduce conflict misses

Local vs global variables

- Compiler analysis is more effective with local variables
- Has to make worst case assumptions about global variables
- Globals could be modified by any called procedure (or by another thread).
- Use local variables where possible
- Automatic variables are stack allocated: allocation is essentially free.
- In C, use file scope globals in preference to externals

Conditionals

- Even with sophisticated branch prediction hardware, branches are bad for performance.
- Try to avoid branches in innermost loops.
  - if you can't eliminate them, at least try to get them out of the critical loops.
- Simple example:

```
do i=1,k
  if (n .eq. 0) then
     a(i) = b(i) + c
  else
    a(i) = 0.
  endif
end do
```

→

```
if (n .eq. 0) then
  do i=1,k
    a(i) = b(i) + c
  end do
else
  do i=1,k
    a(i) = 0.
  end do
endif
```

Conclusions

- Lots of different approaches
- Simple steps can give big benefits
  - Compiler flags
  - Awareness of memory layout for coding
- Need to really understand performance before starting work
  - Profiling, hardware counters, etc…
- Consider portability