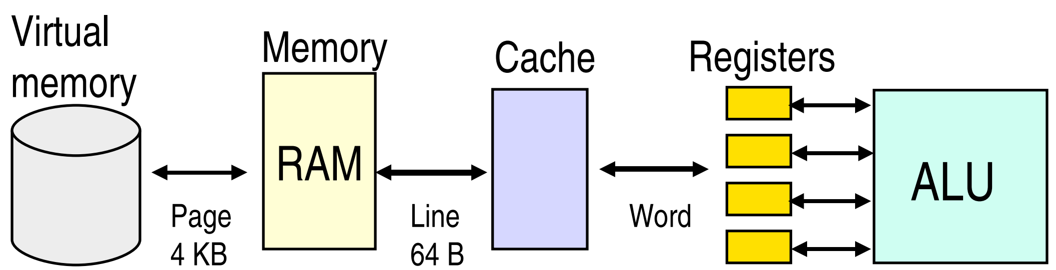
Cache memory

- Modern processors have a hierarchical memory organisation
  - registers
  - cache memory
  - main memory
  - disk memory



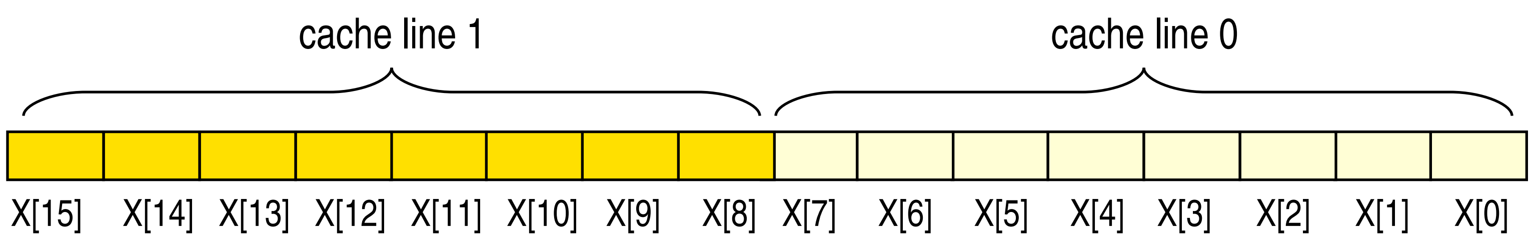
- Typical access times (on Intel Core2)

register	immediately (0 clock cycles)
L1 (on-chip)	3 clock cycles
L2 (on-chip)	13 clock cycles
memory	100 clock cycles
disk	100 000 – 1 000 000 clock cycles

Cache lines

- Data is transferred between main memory and cache one *cache line* at a time
  - in the AMD Opteron the cache line size is 64 bytes
- When a memory location is accessed for the first time, the whole cache line containing the address is copied from memory to the cache
  - a cache replacement policy defines how old data in the cache is replaced with new data
  - tries to keep frequently used data in the cache – Least Recently Used algorithm
  - for each memory access, the computer first checks if the cache line containing the memory location already is in the cache (cache hit) or not (cache miss)

Example: an array of floating-point values
double X[N];



Principle of locality

- A hierarchical memory organization works well because of the principle of locality
  - *temporal locality*: a memory location that is accessed in a program is likely to be accessed again in the near future
  - *spatial locality*: memory locations that are near to some memory location that has been accessed will probably also be accessed in the near future
- Programs with good locality can use the hierarchical memory organization efficiently
  - use *all* of the data that is brought in to the lower memory levels
  - reuse data that have already been brought in to the lower memory levels

Declaring variables and constants

- Variables should be declared in order of type size
  - declare largest types first, smallest last
  - otherwise the compiler may insert padding bytes for alignment
- Use local variables for computations in procedures
  - local variables are stored on the stack, together with the arguments of the procedure
  - use global or static variables only if it is absolutely necessary
- Declare all constant values as constants
  - use the `const` type qualifier for constant values

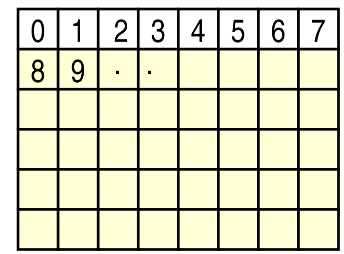
Declaring data structures

- Declare data structures in order of type size
  - otherwise the compiler inserts padding bytes to align the members of the structure
  - this leads to inefficient cache usage when accessing arrays of structures
- The size of a data structure might not be the expected
  - it is often larger than the sum of the members, because of padding
  - use the `sizeof` function to get the actual size of a data structure
- Data structures of size ≥ 16 bytes can be aligned to cache line boundaries
  - -cache_align compiler flag in PGI compilers

Accessing data with unit stride

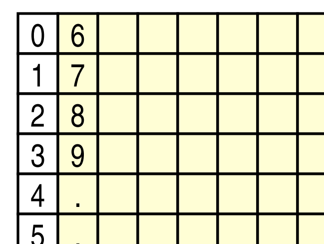
- Arrange loops so that memory is accessed with unit stride
- In C and C++, matrices are stored in row-major order
  - in Fortran, matrices are stored in column-major order
- Accessing consecutive memory locations uses all the data in a cache line
  - improves locality
  - uses automatic prefetching

```
for (i=0; i<rows; i++)
  for (j=0; j<cols; j++)
    X[i][j] = 0;
```



- Accessing non-consecutive memory locations may generate large numbers of cache misses

```
for (j=0; j<cols; j++)
  for (i=0; i<rows; i++)
    X[i][j] = 0;
```



Arrays of Structures or Structures of Arrays

- Array of Structures (AoS)
  - a structure describing some related data items
  - allocated as an array of structures
  - the members of the structure are contiguous in memory
- Structure of Arrays (SoA)
  - a structure containing separate arrays for the data items
  - allocated as a number of arrays of the same length
  - items in one array are contiguous in memory
- Which one is more efficient depends on how the elements are accessed in the computation
  - if we access all members, AoS is better
  - if we only access one (or a few) of the members, SoA is better

```
typedef struct {
    double x,y,z;
    int a, b;
} Vertex;

Vertex V[N];
```

```
typedef struct{
    double x[N];
    double y[N];
    double z[N];
    int a[N];
    int b[N];
} VerticeList;

VerticeList V;
```

Blocking

- Divide the data into smaller blocks which fit in the cache
  - do the computation on one block of data at a time
- Choose the blocksize so that all the data needed to compute one block fits into cache
- Example: matrix multiplication
  - normal $O(N^3)$ algorithm

```
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      for (k=0; k<N; k++)
         Z[i][j]+=X[i][k]*Y[k][j];
```

Matrix multiplication with cache blocking

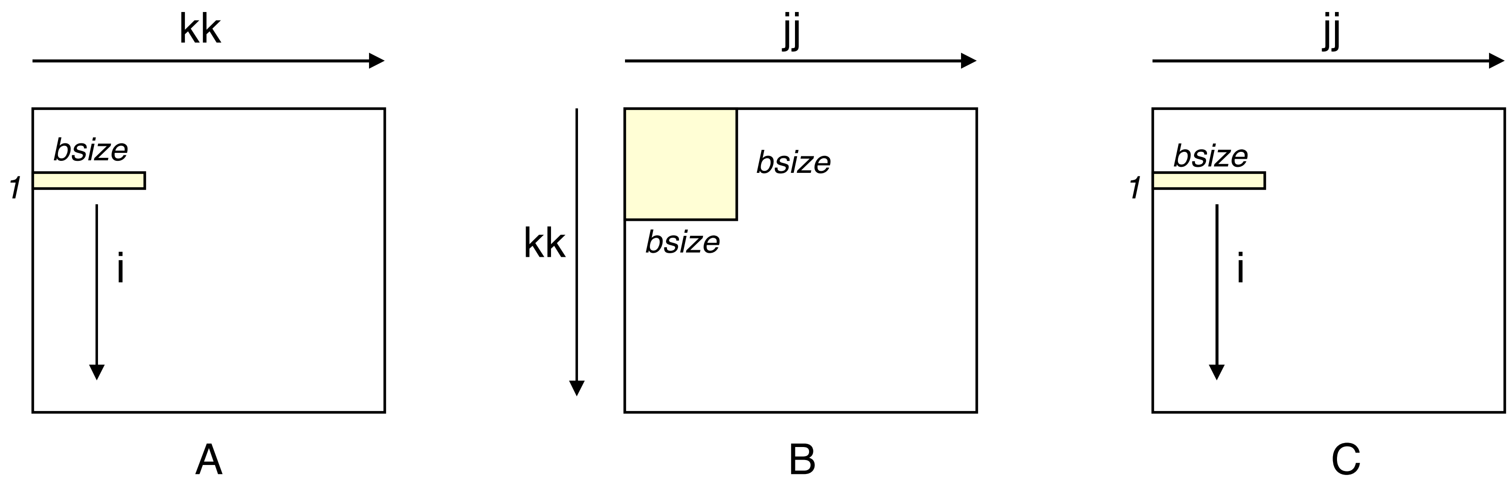
- The algorithm is from the book by Bryant & O'Hallaron

```
void matrixmult(float *A, float *B, float *C, int N, int blocksize){
  float sum;
  int i, j, k;
  int iblock, jblock, kblock;

  /* Blocked matrix multiplication */
  for (kblock=0; kblock<N; kblock+=blocksize) {
    for (jblock=0; jblock<N; jblock+=blocksize) {
      for (i=0; i<N; i++) {
        for (j=jblock; j<jblock+blocksize; j++) {
          sum = C[i*N+j];
          for (k=kblock; k<kblock+blocksize; k++) {
            sum += A[i*N+k]*B[k*N+j];
          }
          C[i*N+j] = sum;
        }
      }
    }
  }
}
```

Illustration of blocked matrix multiply

- The innermost *j-* and *k*-loops multiplies a 1**bsize* slice of A by a *bsize*bsize* block of B and accumulates into a a 1*bsize slice of C



Use *1 x bsize* row sliver *bsize* times

Use *bsize x bsize* block *n* times in succession

Update successive elements of *1 x bsize* row sliver

Branches

- Branch instruction (jumps, calls, returns) are very common in many types of applications
  - in high level languages branches correspond to if- and switch-statements, loops and procedure calls and returns
- Conditional branches can cause problems in a deeply pipelined architecture
- Modern processors use advanced branch prediction mechanisms
  - predict the outcome of branch instructions
  - fetch the next instruction from the predicted execution path

Programming for efficient branch behavior

- Eliminate branches if possible
  - loop unrolling, unswitching, fusion, function inlining
  - order compound boolean expressions for short-circuit evaluation
  - enable the compiler to generate *conditional move* instructions or SSE *min* and *max* operations
- Avoid branches that can not be predicted
  - branches that depend on the dynamic program execution
  - random branches
  - indirect calls and jumps (function pointers, jump tables)
- Avoid too deep nesting of subroutines
  - use iterative functions instead of recursive, if possible

Loop unrolling

- Repeat the body of the loop *k* times and reduced the iteration count by a factor *k*
  - *k* is called the unrolling factor
  - can not assume that *N* is divisible by *k*

```
for (i=0; i<N; i++)
  sum += X[i];
```

- Can also reduce dependences by unrolling loops

```
const int k=5; /* Unrolling factor */
int limit = length-(k-1);
sum=0.0;
for (i=0; i<limit; i+=k) {
  sum += X[i];
  sum += X[i+1}; /* Unroll by k */
  sum += X[i+2];
  sum += X[i+3];
  sum += X[i+4];
}
/* Finish reminding elements */
for (; i<length; i++)
  sum += X[i];
```

```
const int k=5;
int limit = length-(k-1);
sum0=sum1=sum2=sum3=sum4=0.0;
for (i=0; i<limit; i+=k) {
  sum0 += X[i];
  sum1 += X[i+1];
  sum2 += X[i+2];
  sum3 += X[i+3];
  sum4 += X[i+4];
}
for (; i<length; i++)
  sum0 += X[i];
sum0 += sum1+sum2+sum3+sum4;
```

Unrolling small loops

- Small loops with a fixed loop count and a small loop body can be completely unrolled
  - the compiler can automatically unroll simple lops

```
/* 3D transform
   Multiply the vector v with a 4x4 transformation matrix m */
for (i=0; i<4; i++) {
    r[i] = 0;
    for (j=0; j<4; j++) {
        r[i] = m[i][j] * v[j];
    }
}
```

```
r[0] = m[0][0]*v[0] + m[1][0]*v[1] + m[2][0]*v[2] + m[3][0]*v[3];
r[1] = m[0][1]*v[0] + m[1][1]*v[1] + m[2][1]*v[2] + m[3][1]*v[3];
r[2] = m[0][2]*v[0] + m[1][2]*v[1] + m[2][2]*v[2] + m[3][2]*v[3];
r[3] = m[0][3]*v[0] + m[1][3]*v[1] + m[2][3]*v[2] + m[3][3]*v[3];
```

Evaluating boolean expressions

- C and C++ uses short-circuit evaluation for compound boolean expressions
  - if *a* evaluates to TRUE in an expression *if (a || b)*, then *b* is not evaluated
  - if *a* evaluates to FALSE in an expression *if (a && b)*, then *b* is not evaluated
- If one of the expressions is known to be true more often than the other, arrange the expressions so that the evaluation is shortened
  - if the boolean expressions have side effects or are dependent, they can not be rearranged
- If one expression is more predictable, place that first
- If one expression is much faster to calculate, place that first

Order of branches

- Order branches in *if*- and *switch*-statements so that the most likely case is first
  - if the case expressions are contiguous, the compiler may translate a *switch*-statement into a jump table
  - if they are noncontiguous, use a series of *if-else* statements instead

```
switch (value) {/* Most likely case first */
  case 0: handle_0(); break;
  case 1: handle_1(); break;
  case 2: handle_2(); break;
  case 3: handle_3(); break;
}
```

```
if (a==0) {
  /* Handle case for a==0 */
}
else if (a==8) {
  /* Handle case for a==8 */
}
else {
  /* Handle default case */
}
```

Loop unswitching

- Move loop-invariant conditional constructs out of the loop
  - if- or switch-statements which are independent of the loop index can be moved outside of the loop
  - the loop is instead repeated in the different branches of the if- or case- statement
  - removes branch instructions from within the loop, increases instruction level parallelism

```
for (i=0; i<N; i++) {
   if (a>0)
      X[i] = a;
   else
      X[i] = 0;
}
```

```
if (a>0) {
  for (i=0; i<N; i++)
       X[i] = a;
}
else {
  for (i=0; i<N; i++)
       X[i] = 0;
}
```

Loop peeling

- A small number of iterations from the beginning and/or end of a loop are removed and executed separately
  - for example the handling of boundary conditions
- Removes branches from the loop
  - results in larger basic blocks
  - more instruction level parallelism

```
for (i=0; i<N; i++) {
  if (i==0)
     X[i] = 0;
  else if (i==N)
     X[i] = N;
  else
    X[i] = X[i]*c;
}
```

```
X[i] = 0;
for (i=1; i<N-1; i++) {
  X[i] = X[i]*c;
}
X[N] = N;
```

Loop fusion and loop fission

- Loop fusion combines multiple loops operating over the same index range into one single loop
  - may lead to better locality of reference
  - may increase register pressure
- Loop fission breaks up a complicated loop into multiple smaller loops, iterating over the same index range
  - may achieve better locality of reference and reduce register pressure

```
for (i=0; i<N; i++) {
   A[i] = 0.0;
   B[i] = 1.0;
}
```

```
for (i=0; i<N; i++) {
   A[i] = 0.0;
}
for (i=0; i<N; i++) {
   B[i] = 1.0;
}
```