High-level code optimization

- Modern compilers are very good at low-level code optimization
  - fairly simple code transformations
  - limited by the compilers' ability to analyze the code
- The programmer can help the compiler by using a clear and simple programming style
- More advanced optimizations have to be done by the programmer
  - code transformation techniques applied at the source code level
  - need an understanding about the computations of program, how data is accessed and the dependencies between data

Operation counting

- Estimate how many load, store, floating-point and integer operations are executed in a loop
  - indicates how well the instruction mix fits the processor architecture
  - we know how many load/store, floating-point and integer operations can be executed per clock cycle
- *Example 1*: adding two arrays
  - one floating-point addition, three memory operations
  - load A[i], load B[i], add, store A[i]
  - ratio of memory to floating-point operations is 3:1
  - performance will be limited by the time to access memory

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

- Assume that address calculations, loop counter incrementing and branching are executed by separate functional units

Operation counting (cont.)

- *Example 2:* element-wise multiplication of arrays of complex numbers
  - real part in arrays *xr*, *yr*
  - imaginary part in arrays *xi*, *yi*

```
for (i=0; i<N; i++) {
  tmp = xr[i]*yr[i] - xi[i]*yi[i];
  xi[i] = xr[i]*yi[i] + xi[i]*yr[i];
  xr[i] = tmp;
}
```

- Six memory operations, six floating-point operations
  - load xr[i], load yr[i], multiply, load xi[i], load yi[i], multiply, subtact, store xr[i]
  - operands for the second statement are already loaded in registers
  - multiply, multiply, add, store xi[i]
- Better balance than in previous example
  - values loaded into registers are reused
  - but if we use a multiply-and-add instruction, the loop is still limited by memory references

Loop optimization

- Loops are important targets for high-level code optimization
  - heaviest computations in a program are normally in loop nests (loops within loops)
  - compilers may not be able to analyze complicated loop structures and do automatic code transformations
- Goals
  - improve memory access patterns
    - access data with unit stride
    - reuse values that are loaded into registers
  - increase instruction level parallelism
    - bigger basic blocks
- Loop unrolling is a very important code optimization method also on source code level
  - can also unroll outer loops in a nested loop structure

Outer loop unrolling

- If the inner loop can't be unrolled, outer loops may be unrolled
  - if the inner loop is very short
  - if data dependencies makes it impossible to unroll the inner loop
- Example:
  - unroll outer loop by 4
  - loads of Y[j] can be hoisted
- Loop unrolling increases register pressure

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

```
for (i=0; i<N; i+=4)
  for (j=0; j<N; j++) {
    A[i,j]   += X[i]*Y[j];
    A[i+1,j] += X[i+1]*Y[j];
    A[i+2,j] += X[i+2]*Y[j];
    A[i+3,j] += X[i+3]*Y[j];
  }
```

Loop fusion

- Combine loops that operate on the same data
  - improves cache usage, reuses values that have been loaded into registers
  - reduces loop overhead
  - increases instruction level parallelism

```
for (i=0; i<N; i++) {
  tmp[i] = X[i]*Y[i];
}
for (i=0; i<N; i++) {
  Z[i] = W[i]+tmp[i];
}
```

```
for (i=0; i<N; i++) {
  Z[i] = W[i]+X[i]*Y[i];
}
```

- Opposite technique is loop fission
  - split up big loops into smaller

Loop peeling

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

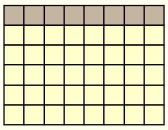
```
for (i=0; i<N; i++) {
  if (i=0)
     X[i] = 0;
  else if (i=N)
     X[i] = N;
  else
    X[i] = X[i]*c;
}
```

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

Loop interchange

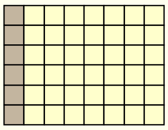
- Rearrange loops so that memory is accessed with unit stride
- In C and C++, matrixes are stored in row-major order
  - in Fortran, matrixes are stored in column-major order
- Accessing consecutive memory locations uses all data in cache lines
  - unit stride
  - automatic prefetching

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



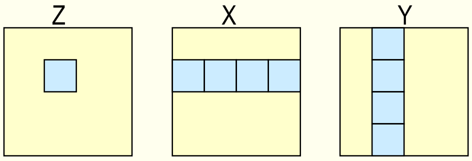
- Accessing non-consecutive memory locations generates large numbers of cache misses

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



Blocking

- Optimization for data that does not fit in the cache
- 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 *Z* = *X***Y*
  - *NxN* matrixes, *N* divisible by *blocksize*
  - do the multiplication one block at a a time



```
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 matrix is divided into blocks of size *blocksize x blocksize*

```
for (iblock=0; iblock<N; iblock+=blocksize) {
  ilimit = iblock + blocksize;

  for (jblock=0; jblock<N; jblock+=blocksize) {
    jlimit = jblock + blocksize;

    for (kblock=0; kblock<N; kblock+=blocksize) {
      klimit = kblock + blocksize;

      for (i=iblock; i<ilimit; i++) {
        for (j=jblock; j<jlimit; j++) {
          for (k=kblock; k<klimit; k++) {
            Z[i][j] += X[i][k] * Y[k][j];
          }
        }
      }
    }
  }
}
```

Pointers and aliasing

- Pointers in C may specify the same memory location
  - called aliasing
- When the compiler analyzes a program, it has to assume that data that is accessed through pointers may overlap
  - the compiler is not allowed to rearrange instructions using loop unrolling, instruction scheduling, hoisting or sinking
  - has to generate very conservative code for operations on data accessed through pointers
- Give the compiler as much information about data layout as possible
  - use static allocation instead of dynamic
- Compilers often have an option to assume no aliasing

```
#define N 1000
double A[N][N], B[N][N], d;
...
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    A[i][j] += B[i][j]*d;
```

Memory alignment

- Data alignment can have a significant impact on peformance
- The compiler by default aligns data on natural boundaries
  - 64-bit values are by default aligned on word boundaries
- Aligning 64-bit values on 8 byte boundaries can improve performance
  - increases memory usage
  - structures containing 64-bit data types will have a different memory layout than the default
- Data used in MMX and SSE operations should be aligned on 16 byte boundaries
- Aligning branch targets is more important for architectures with a traditional L1 data cache
  - not so important in architectures with a trace cache

gcc options for alignment

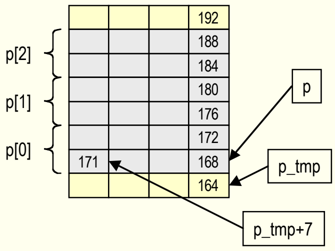
- gcc compiler switches for alignment
  - -malign-double
    - aligns double-precision variables on 8 byte boundaries (defalult is 4 byte boundaries)
  - -malign-jumps=*n*
    - align branch targets on 2^n byte boundaries (default is to align branches on 16 byte (=2^4) boundaries)
  - -malign-loops=*n*
    - align loops on 2^n byte boundaries (default is to align branches on 16 byte (=2^4) boundaries)
  - -malign-functions=*n*
    - align the start of functions to 2^n byte boundaries (default is 4 bytes for 386 and 16 bytes for 486 architecture)
  - -mpreferred-stack-boundary=*n*
    - attempt to keep stack aligned to 2^n byte boundaries (default is 16 bytes)

Explicite aligning

- Can also explicitely align pointers to dynamically allocated memory

```
/* Allocate an array of N 8-byte aligned double */
double *p_tmp, *p;
p_tmp = (double *)malloc(sizeof(double)*(N+1));
p = (p_tmp+7) & (-0x7);
```

- Allocate memory for one more element than needed
  - advance pointer to the memory block with 7 bytes (to end of the first doubleword)
  - mask out the 3 last bits to get an 8-byte aligned address



Aligning structures

- Members of structures should be naturally aligned
  - pad the structure to a multiple of the size of the largest member, if necessary
- Declare variables in a structure in order of size of members
  - largest members first, smallest last
- Arrays of structures will be naturally aligned
- Example:
  - two 8-byte double *x, y*
  - one 4-byte int *value*
  - one byte *flag*
  - three padding bytes

```
typedef struct {
    double x,y;
    int value;
    char flag;
    char pad[3];
} Point;
```

Arrays of Structures or Structures of Arrays

- AoS – Array of Structures
  - define a structure describing the data items we operate on
  - allocate an array of structures
  - structures are contiguous in memory (in a cache line)
- SoA – Structure of Arrays
  - structure containing a number of separate arrays for the items we operate on
  - allocate a number of arrays of same length
  - items in one array are contiguous in memory (in a cache line)
- SoA is better suited for SIMD operation
  - also better if some elements are accessed more seldom

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

Vertex V[N];
```

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

VerticeList V;
```

Avoiding cache trashing

- Avoid allocating contiguous arrays with a size (in bytes) that is a power of 2
  - arrays may map to the same cache line
  - L1 cache is 4-way (or 2-way) set associative
  - all accesses may map to the same location in cache

```
const int N=1024
...
double X[N], Y[N], Z[N];
int    a[N], b[N], c[N];
...
for (i=0; i<N; i++) {
  X[N] = Y[N] + Z[X];
  a[N] = b[N] + c[N];
}
```

- Pad arrays with a multiple of the cache line size
  - add 128 bytes to the size of arrays

```
const int N=1024;
const int N_p=N+16;
...
double X[N_p], Y[N_p], Z[N_p];
int    a[N_p], b[N_p], c[N_p];
...
for (i=0; i<N; i++) {
   ...
```

Branch prediction

- Eliminate branches
  - loop unrolling, unswitching, fusion, function inlining
- Avoid branches that can not be predicted
  - brances that depend on the dynamic execution
  - random behaviour can not be predicted
- Avoid deep nesting of subroutines
  - use iterative functions instead of recursive, if possible
- Order the cases in switch statements according to probability of occurence
  - most common case first

Floating-point computations

- Ensure that floating-point data is aligned
- Use multiplication instead of division
  - but beware of consequences for accuracy
- Avoid over- and underflow and denormal operands
  - keep floating-point values within range
    - overflow and underflow may cause very high overhead
    - small floating-point values can be represented with highest precision
  - use float or double as needed by the application
    - float operations are faster, especially division and square root
    - float also need less memory
- Minimise floating-point to integer conversions

Variables and declarations

- Provide the compiler with information about the computation
  - use prototypes for all functions
  - declare local functions as static
  - use the const type qualifier for constants
  - use local variables, minimise use of global variables
  - use arrays instead of pointers
- Use 32-bit data types for integer values
- Avoid the register modifier
  - the compiler can do better register allocation than the programmer
- Declare local variables in order of base type size
- Avoid unnecessary type casting
  - floating-point constants are by default double, unless explicitely declared as float: `x=y+3.1415f;`