Multi-dimensional Arrays
Allocation and Layout of Arrays

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Importance of Array Layout

• We now know that trying to enhance spatial locality in our memory accesses is important for performance

• The reasons are somewhat circular
  • Architects observed that programmers tend to access nearby locations: e.g., linear sweep through a 1-dimensional (1-D) array
  • Provided features in hardware that improve performance for such accesses
    • Cache lines contain 64 contiguous bytes
    • Hardware prefetcher

• This is the reason why it is important to know how arrays are laid out in memory
  • Of course, a 1-D array is laid out as expected
Statically Allocated Multi-dimensional Arrays

• These are either
  • Global variables or
  • Declared inside a function (and so allocated on stack)

• int A[10][50];

• float B[e1][e2][e3];
  • Where e1, e2, e3 are expressions made of constants
  • C (C99 onwards) allows these expressions to contain variables, such as those passed as parameters of functions

• Layout for statically allocated 2-D arrays in C and C++ is “row major”
  • A[i][j+1] is adjacent in memory to A[i][j]
Layout and Cache Lines

• With row-major layout, locations within a row, in consecutive columns are next to each other in memory.
Dynamically Allocated Multi-dimensional Arrays

• A common method for allocating these is to create arrays of array-pointers

• But this is bad for locality

• Consecutive rows may be arbitrarily separated in memory

• Padding in allocation wastes memory

• Reduced predictability for loop accesses means prefetchers do not perform well

A = ... malloc(sizeof(float*)*M);
for (i = 0; i < M; i++)
    A[i] = ... malloc(sizeof(float) * N);
A better, and more general, method is to allocate all the space with one allocation call

And then to index calculation explicitly

Indexing is somewhat awkward, but you get used to it

- You can use macros: indexOf(i,j,N)
- The apparently complicated index calculations are not expensive, because the compiler can easily optimize them

Or use an array of pointers into a contiguously allocated space

```c
A = malloc(sizeof(float*)*M*N);
...
Instead of A[i][j], use: A[N*i + j]
```
Dynamically Allocated Multi-dimensional Arrays III

• You can use similar index expressions for higher dimensional arrays
• By convention, and to retain the pattern of static allocation, the index expressions are written so as to make the last dimension contiguous

\[
A = (float *)\text{malloc}(\text{sizeof(float)} \times L \times M \times N); \quad \text{// in C}
\]

\[
A = \text{new float}[L \times M \times N]; \quad \text{// in C++}
\]

... Instead of \(A[i][j][k]\), use: \(A[M \times N \times i + N \times j + k]\)
Cache Optimizations
Estimating Performance with Cache Misses
To be able to effectively program a modern multiprocessor, we have to understand what it is made up of and how it came to be the way it is today
What Determines Sequential Performance

• After the code has been compiled (so compilers are out of the picture)

• The floating point units can process arithmetic at a certain rate

• The memory system can bring data into registers at a certain rate
  • By “rate,” we mean bandwidth (i.e., bytes/second)

• Which rate decides performance?
  • The slowest one

• This is quantified in the idea of floating point (or arithmetic) intensity
  • I.e., how many double precision arithmetic operations does a given code do per word (or byte) transferred between memory and registers via “load” or “store” operations
Example Code for Estimating Performance

A loop with some data accesses:

```
for (i=0; i<N; i++)
    x += A[i];
```

Assumptions:
- Clock rate 2 GHz (0.5 ns period per clock cycle)
- 1 FP per cycle (note: if we had FMAD operation, it could be 2)
- N is 1,000,000
- Cache line size is 64 bytes
- A is an array of doubles
  - 8 bytes each
- Cache miss penalty: 50 ns

If there were no cache misses:
N * 0.5 ns = 0.5 ms

With cache misses:
N * 0.5 ns + (N/8) * 50 ns = 0.5 ms + 6.25 ms = 6.75 ms

More than 10 times slower
Arithmetic Intensity: Example

• What is arithmetic intensity for the following loops?

- Loop 1: for(i=0;i<N;i++)
  x+=A[i];
- Loop 2: for(i=0;i<N;i++)
  x+=A[i]*A[i];
- Loop 3: for(i=0;i<N;i++)
  x+=A[i]*A[i]*A[i];

• In each iteration, there is only 1 word loaded: A[i]
  • Why are we not counting x?
    • Because x will be in a register. Loaded once at the beginning of the loop

• How many floating point operations per iteration?
  • Let us count “+” and “*”s separately
    • 1, 2, and 3 respectively (We don’t count integer arithmetic in i++. Why?)

• So, arithmetic intensity of Loop1: 1 FP/word (or 1FP/8bytes: 0.125)
• Loop2: double, Loop 3: triple (i.e 3/8)
Improving Arithmetic Intensity: Example 2

- What is arithmetic intensity for the following loops?

```c
for(i=0;i<N;i++)
    x += A[i];
for(i=0;i<N;i++)
    s += A[i]*A[i];
```

Code 1

```c
for(i=0;i<N;i++)
    x += A[i];
for(i=0;i<N;i++)
    s += A[i]*A[i];
```

Code 2

- Code 1 does 1FP op per load
- **Code 2 does 2 FP ops per load, and accomplishes the same result**
  - Loop2 will be faster
Improving Arithmetic Intensity: Example 3

• What is arithmetic intensity for the following loops?

```c
for(i=0;i<N;i++)
    x += A[i];
for(i=0;i<N;i++)
```

```c
for(i=0;i<N;i++)
    x += A[i];
```

Code 1 Code 2

• No floating point ops in the second loop, but still code2 is better, because it incurs fewer cache misses
Cache Based Optimizations

• For a given code, with a fixed arithmetic intensity, how to improve performance?

• The basic idea is to decrease the number of cache misses
Doubly Nested Loop

```
for(i=0; i<N; i++)
    for(j=0; j<M; j++)
        x += A[j][i];
```

• What is the problem? Count the number of misses
• Assume the cache size is less than $N \times w$,
  • Where $w$ is the number of words per cache line
• Every access will lead to a cache miss
  • $N^2$ cache misses
Fixing the Doubly Nested Loop: Reordering

for (j = 0; j < M; j++)
    for (i = 0; i < N; i++)
        x += A[j][i]
Cache Optimizations: Improving Reuse
Matrix Vector Multiplication
Matrix Vector Multiply

• Assume cache is smaller than N words
• A and C incur only compulsory misses \((N^2/w, N/w\) respectively)
• B is loaded multiple times, with \(N^2/w\) misses
  • For each row of A, B is traversed once, but by the time we go to the next row, the older portions of B are out of the cache

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
```
Matrix Vector Multiply: improve reuse of B?

- Idea: let us reuse a value from B (say B[j]) multiple times
- Lets say we load B[0]..
  - Which calculations need it?
- A loop interchange will reuse B[0], but A accesses will suffer
  - Column order traversal
- But if we do loop interchange only for $X$ rows, the lines (orange) will still be in cache

```c
for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
        C[i] += A[i][j]*B[j]
```

\[
x = \begin{bmatrix}
    A \\
    B \\
    C
\end{bmatrix}
\]
Matrix Vector Multiply: improved

for (i = 0; i < N; i+= X)
    for (j = 0; j < N; j++)
        for( k = 0; k < X; k++)
            C[i+k] += A[i+k][j]*B[j]

• Assume cache is smaller than N words
• A and C incur only compulsory misses (N^2/w, N respectively)
• B is reused X times with total N^2/x*w misses
  • For each X rows of A, B is traversed once
Cache Optimizations: Tiling
Matrix Transpose

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Classic Example of Optimizing for Cache Size

- Matrix transpose
- A matrix accesses: $N^2$ misses!
- B accesses are fine:
  - Only compulsory misses: $N^2/w$ misses

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        B[i][j] = A[j][i]
```
Solution: Tiling

Let us assume $N$ is a multiple of $X$, the tile-size.

$$\text{for } (i = 0; i < N; i += X)$$

$$\text{for } (j = 0; j < N; j += X)$$

$$\text{for } (p = 0; p < X; p++)$$

$$\text{for } (q = 0; q < Y; q++)$$

$$B[i+p][j+q] = A[j+q][i+p]$$
Cache-Oblivious Algorithms

• All of the above ideas were for taking into account the specific finite cache size
• Also, we focused on only one cache level, but in reality there are L1, L2, and L3
• Another idea is to write your algorithms in a way that ignores the specific cache size, but still improves cache performance
  • Cache-oblivious algorithms, which are typically recursive
Cache-Oblivious Algorithms
Cache-Oblivious Algorithms

B

L.V.Kale

A
Cache-Oblivious Matrix Transpose

recTranspose(A, x, y, B, t, N) { // t is tilesize
    if (t < X)
        transpose(A, x, y, B, t, N);
    else {
        recTranspose(A, x, y, B, t/2, N);
        recTranspose(A, x, y+t/2, B, t/2, N);
        recTranspose(A, x+t/2, y, B, t/2, N);
        recTranspose(A, x+t/2, y+t/2, B, t/2, N);
    }
}