# Optimization Techniques, Loops

Andrey Kamaev
Itseez UNN 2016

# Loop Optimizations

Single loop optimizations
Nested loop optimizations

# Single Loop Optimizations

- Induction variable optimization
- Loop invariant code motion
- Scalarization
- Loop unswitching
- Loop peeling
- Loop fusion
- Loop fission
- Loop reversal
- Software pipelining
- Loop unrolling
- Loop vectorization

# Induction Variable Optimization

- Simplify expressions that change as a linear function of the loop index
  - the loop index is multiplied with a constant
  - replaces a multiplication with an addition
- Often used in address calculations when iterating over an array

```
for (int i = 0; i < N; i++) {
    int k = 4 * i + m;
    ...
}
```

```
int k = m;
for (int i = 0; i < N; i++) {
    ...
    k += 4;
}
```

# Loop Invariant Code Motion

- Eliminate invariant code from loops
  - in particular, try to move expensive calls (`malloc`, `strlen`, file open/close, ...) out of loops

```
for (int i = 0; i < strlen(s); i++)
{
    // do something with s[i];
}
```

```
int len = strlen(s);
for (int i = 0; i < len; i++)
{
    // do something with s[i];
}
```

# Scalarization

- Resolves aliasing conflicts
- Eliminates repeatitive memory reads

```
float coeffs[2];


for (int i = 0; i < N; i++) {
    a[i] = b[i]*coeffs[0]+coeffs[1];
}
```

```
float coeffs[2];
float c0 = coeffs[0], c1 = coeffs[1];

for (int i = 0; i < N; i++) {
    a[i] = b[i] * c0 + c1;
}
```

# Loop Unswitching

- Move loop invariant conditional constructs out of the loop
  - if or switchs tatements 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 **ILP**

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

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

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

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

# Loop Fusion

- Loop overhead reduced
- Better instruction overlap
- Temporary locality is improved = lower cache misses
- Be aware of associativity issues with array's mapping to the same cache line

```
for (int i = 0; i < N; i++)
    x = x * a[i] + b[i];
for (int j = 0; j < 2*N; j++)
    y = y + a[j] / b[j];
```

```
for (int i = 0; i < N; i++) {
    x = x * a[i] + b[i];
    y = y + a[i] / b[i];
}
for (int j = N; j < 2*N; j++)
    y = y + a[j] / b[j];
```

# Loop Fission

- Register pressure reduced
- Cache associativity conflicts are minimized
- Can be used to break loop-carried dependencies

```
for (int i = 0; i < N; i++) {
    a[i] = b[i] + 1;
    c[i] = 3 * a[i];
    f[i] = g[i] + h[i];
}
```

```
for (int i = 0; i < N; i++) {
    a[i] = b[i] + 1;
    c[i] = 3 * a[i];
}
for (int j = 0; j < N; j++)
    f[j] = g[j] + h[j];
```

# Loop Reversal

- Change the direction of loop iteration
  - Can improve cache performance
  - Enables other transformations

```
for (int i = 0; i < N; i++) {
    a[i] = b[i] + 1;
    c[i] = 3 * a[i];
}
for (int j = 0; j < N; j++)
    d[j] = c[j+1] + 1;
```

```
for (int i = N - 1; i >= 0; i--) {
    a[i] = b[i] + 1;
    c[i] = 3 * a[i];
}
for (int j = N - 1; j >= 0; j--)
    d[j] = c[j+1] + 1;
```

# Software Pipelining

- Applies instruction scheduling, allowing instructions within a loop to “wrap around” and execute in a different iteration of the loop
- Reduces the impact of long-latency operations, resulting in faster loop execution
- Emulates prefetching of data to reduce the impact of cache misses
- Often used with together with loop unrolling

```
for (int i = 0; i < N; i++) {
    a[i]++;
    b[i] = a[i] / 2;
}
```

```
for (int i = 0; i < N; i++) {
    a[i] = 3*b[i] + 1;
}
```

```
if (N > 0) {
    a[0]++;
    for (int i = 0; i < N-1; i++) {
        a[i+1]++;
        b[i] = a[i] / 2;
    }
    b[N-1] = a[N-1] / 2;
}
```

```
if (N > 0) {
    int t = b[0];
    for (int i = 0; i < N-1; i++) {
        int p = b[i+1];
        a[i] = 3 * t + 1;
        t = p;
    }
    a[N-1] = 3 * t + 1;
}
```

# Loop Unrolling

- Improves processor pipeline utilitization (higher **ILP**)
- Reduces loop overhead
- Can break loop-carried dependencies
- Needed for vectorization
- Increases register pressure
- Leads to cache associativity conflicts
- Requires additional code to process leftovers
- Can affect results of floating-point arithmetic

```
float sum = 0;
for (int i = 0; i < N; i++) {
    sum += a[i];
}
```

```
float sum = 0;
int i = 0;
for (; i < (N/4)*4; i++) {
    sum += a[i] + a[i+1] + a[i+2] + a[i+3];
}
for (; i < N; i++) {
    sum += a[i];
}
```

```
float sum = 0;
for (int i = 0; i < N; i++) {
    sum += a[i];
}
```

```
float sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0;
int i = 0;
for (; i < (N/4)*4; i++) {
    sum1 += a[i];
    sum2 += a[i+1];
    sum3 += a[i+2];
    sum4 += a[i+3];
}
float sum = sum1 + sum2 + (sum3 + sum4);
for (; i < N; i++) {
    sum += a[i];
}
```

# Nested Loop Optimizations

- Strip Mining
- Loop collapse
- Loop interchange
- Outer loop Unroll and Jam
- Loop blocking (tiling)

# Strip Mining

- Usually combined with other optimizations
  - Software prefetch is almost useless without this optimization
- Used to improve locality

```
for (int i = 0; i < N; i++) {
    a[i] = b[i];
    c[i] = c[i-1] + a[i];
}
```

```
for (int i = 0; i < N; i += strip_size) {
    for (int j = i; j < i + strip_size; j++) {
        a[j] = b[j];
        c[j] = c[j-1] + a[j];
    }
}
```

# Loop Collapse

- Reduces loop overhead
- Reduces number of used registers
- Simplifies vectorization

```
char a[N][N];
for (int j = 0; j < N; j++)
    for (int i = 0; i < N; i++)
        a[i][j]++;
```

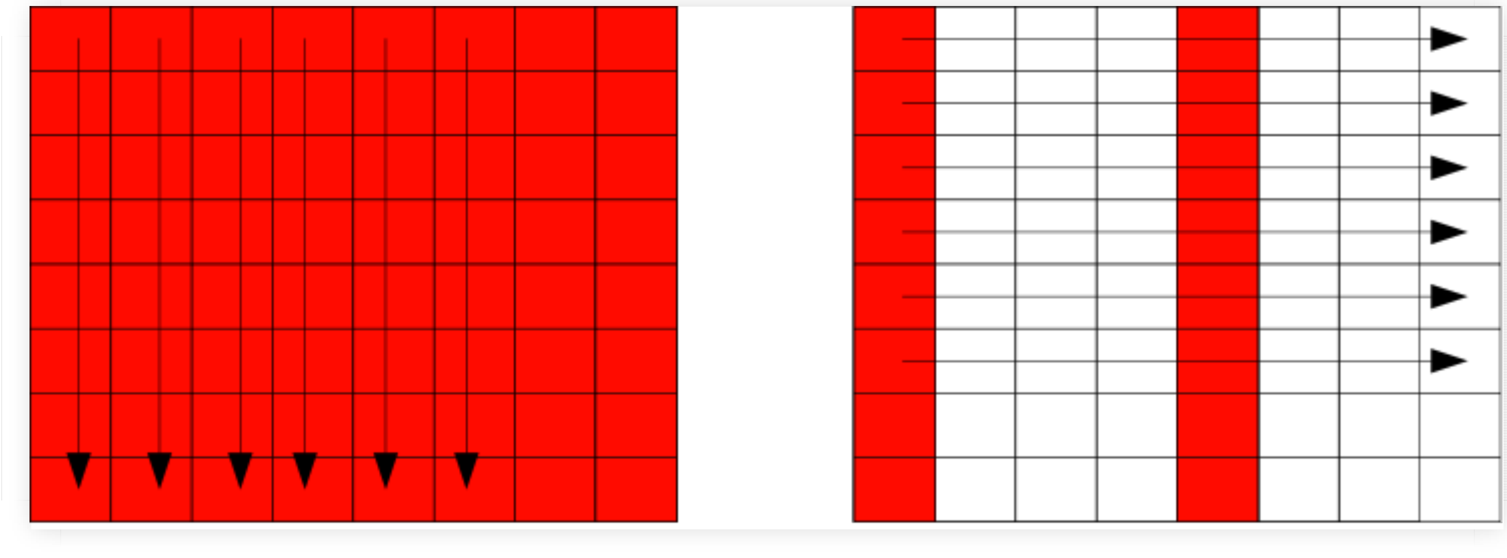
```
char a[N * N];
for (int j = 0; j < N * N; j++)
    a[j]++;
```

# Loop Interchange

- Used mostly to improve spatial locality

```
char a[N][N];
for (int j = 0; j < N; j++)
    for (int i = 0; i < N; i++)
        a[i][j]++;
```

```
char a[N][N];
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        a[i][j]++;
```



Red squares are misses

# Outer Loop Unroll and Jam

- More results can be kept in registers or in cache
- Used to reduce number of loads and stores on inner loops with invariants
- Be careful loop body does not become too large
  - Need registers for all intermediate values
  - Increases cache associativity requirements

```
float a[N][N], b[N][N], c[N];
for (int i = 0; i < N; i++)
    for (int j = 0; i < N; j++)
        a[i][j] = b[i][j] * c[j];
```

```
float a[N][N], b[N][N], c[N];
for (int i = 0; i < (N/4)*4; i+=4)
    for (int j = 0; i < N; j++) {
        a[i+0][j] = b[i+0][j] * c[j];
        a[i+1][j] = b[i+1][j] * c[j];
        a[i+2][j] = b[i+2][j] * c[j];
        a[i+3][j] = b[i+3][j] * c[j];
    }
```

# Loop Blocking

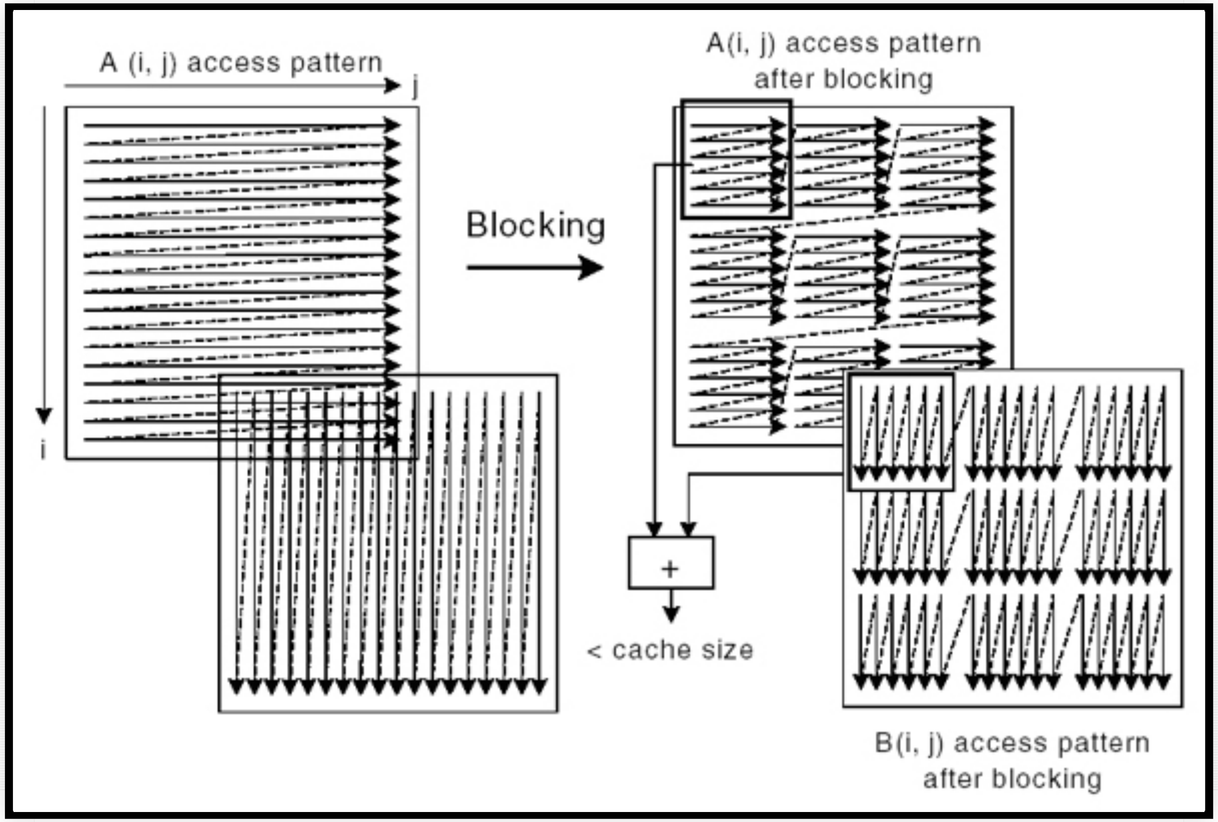
Loop Blocking = Loop Tiling ≈ Generalized Strip Mining

- Reduce memory pressure by making use of the caches
- Helps when potential for **re-use is high**
- Hard to choose optimal block size:
  - The bigger the blocks are, the greater the reduction in memory traffic
  - All accessed blocks must be able to fit into cache at the same time
  - If the blocks are too big, the **TLB** cache will thrash
- Square blocks are simpler, but rectangular blocks, in which the longest dimension corresponds to the inner loop, generally perform the best
- Can help for vectorization
- Can be used to improve precision of floating-point math

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

```
float A[N][N], B[N][N];
for (int i = 0; i < N; i += block_size) {
    for (int j = 0; j < N; j += block_size) {

        for (int ii = i; ii < i + block_size; ii++) {
            for (int jj = j; jj < j + block_size; jj++) {
                A[ii][jj] = A[ii][jj] + B[jj][ii];
            }
        }
    }
}
```



A (i, j) access pattern
j
i
Blocking
A(i, j) access pattern
after blocking
+
< cache size
B(i, j) access pattern
after blocking

# Advanced Examples

# Gather-Scatter Optimization

```
for (int i = 1; i < N; i++) {
    if (t[i] > 0) {
        a[i] = 2 * b[i-1];
    }
}
```

```
int n = 0;
for (int i = 1; i < N; i++) {
    if (t[i] > 0) {
        tmp[n] = i;
        n++;
    }
}

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

The computationally intensive loop runs only over the indices for which the condition was true and can be better optimized

# Ring buffer replacement

This is an example of L1-cache targeted optimization

step #1: scalarization

```
int buf[3];
for (int i = 0; i < N; i++) {
    buf[i%3] = X[i];
    int a = buf[(i-2)%3] * 68;
    a     += buf[(i-1)%3] * 99;
    a     += buf[i%3] * 68;
    X[i] = a;
}
```

```
int b0, b1, b2;
for (int i = 0; i < N; i++) {
    b2 = b1;
    b1 = b0;
    b0 = X[i];
    int a = b2 * 68;
    a     += b1 * 99;
    a     += b0 * 68;
    X[i] = a;
}
```

# step #2: unroll 3x

```
int b0, b1, b2;
for (int i = 0; i < N; i++) {
    b2 = b1;
    b1 = b0;
    b0 = X[i];
    int a = b2 * 68;
    a     += b1 * 99;
    a     += b0 * 68;
    X[i] = a;
}
```

```
int b0, b1, b2;
for (int i = 0; i < N; i+=3) {
    b1 = X[3*i];
    int a0 = b0 * 68;
    a0     += b2 * 99;
    a0     += b1 * 68;
    X[3*i] = a0;

    b2 = X[3*i+1];
    int a1 = b1 * 68;
    a1     += b0 * 99;
    a1     += b2 * 68;
    X[3*i+1] = a1;

    b0 = X[3*i+2];
    int a2 = b2 * 68;
    a2     += b1 * 99;
    a2     += b0 * 68;
    X[3*i+2] = a2;
}
```

# THE END

# Floptimization

Throw in some extra Flops to make the CPU perform more "work" at hardly any extra cost; often there is at least some headroom in the floating-point pipelines when running real applications.

Accumulating something in a register is a classic:

```
for (s = 0.0, i = 0; i < N; ++i) {
    p[i] = f(q[i]); // actual work
    s += p[i];      // floptimization
}
```