Improving The Memory Performance of Loops And Arrays

Optimizing And Tuning Scientific Codes
Scientific Computing

- Use high-end computers to solve scientific problems
  - Computation and/or data intensive simulations
- Use loops to operate on multi-dimensional arrays
  - Structured data (regular computation):
    - Locations/offsets of data known before runtime
    - E.g., subscripting arrays via linear combinations of loop index variables
    - Represented by dense matrix and stencil computations
  - Unstructured data (irregular computation):
    - Structure of data unknown until runtime
    - E.g., subscripting arrays via indirect arrays
    - Represented by sparse matrices, trees, and graphs
- Here, we focus on regular scientific computations
Optimizing For Multicore

- Efficient utilization of
  - Single CPU
    - Cache hierarchy
      - Data affinity and locality
    - SIMD Vectorization
  - Concurrent CPU
    - Multi-threading in OpenMP, Pthread, ...
    - Shared memory
    - Synchronization
Source-level Optimizations

- Automatically applied by compilers to program source
  - Goal: efficient utilization of machine resources
- Reordering of computation and data structures
  - Loop optimizations: reordering of loop evaluation
    - Loop interchange, fusion, blocking, parallelization, unrolling, and unroll&jam
  - Data optimizations: re-organization of data layout
    - Array blocking, scalar replacement
- Reduction of computation cost
  - Prefetching, strength reduction
Reordering Optimizations

- A reordering transformation
  - Changes the ordering of code or data, without adding or deleting any operations.
  - Goal: better utilization of machine resources

- Computation reordering
  - Does not eliminate dependences. Unsafe optimizations may change the relative source and sink of a dependence

- Data reordering
  - It does not change the data, but can change where the data is located

- A computation reordering optimization is safe if it preserves the relative direction of each dependence
- A data reordering optimization is safe if all accesses to each original data item are correctly redirected to the new location
Optimizing Memory Accesses

- Optimizations
  - Reuse data already in cache (locality)
    - Reduce memory bandwidth requirement
  - Prefetch data ahead of time
    - Reduce memory latency requirement

- Two types of cache reuse
  - Temporal reuse
    - After bringing a value into cache, use the same value multiple times
  - Spatial reuse
    - After bringing a value into cache, use its neighboring values in the same cache line

- Cache reuse is limited by
  - cache size, cache line size, cache associativity, replacement policy

DO I = 1, M
DO J = 1, N
  A(I) = A(I) + B(J)
ENDDO
ENDDO

DO I = 1, M
DO J = 1, N
  A(I, J)=A(I,J)+B(I,J)
ENDDO
ENDDO
Optimizing Memory Performance

- Computation optimizations: reordering of instructions
  - Improve temporal and spatial cache reuse
    - Loop interchange/permutation
    - Loop blocking (strip-mining + interchange)
  - Improve register reuse
    - Loop unrolling and unroll&jam

- Data optimizations: rearrange layout of data
  - Static layout transformation
    - A single layout for each variable
  - Dynamic layout transformation
    - Different layout based on variable use
    - Array blocking and scalar replacement
Loop Interchange

- A reordering transformation that
  - Changes the nesting order of loops

- Example
  
  ```plaintext
  DO I = 1, N
    DO J = 1, M
      S A(I,J+1) = A(I,J) + B
    ENDDO
  ENDD
  ```

- After loop interchange
  
  ```plaintext
  DO J = 1, M
    DO I = 1, N
      S A(I,J+1) = A(I,J) + B
    ENDDO
  ENDD
  ```

- Leads to
  
  ```plaintext
  DO J = 1, M
    S A(1:N,J+1) = A(1:N,J) + B
  ENDDO
  ```
Safety of Loop Interchange

Not all loop interchanges are safe

\[
\begin{align*}
\text{DO } J &= 1, M \\
\text{DO } I &= 1, N \\
A(I,J+1) &= A(I+1,J) + B \\
\text{ENDDO}
\end{align*}
\]

Direction vector: \((<, >)\)

\[
\begin{align*}
J &= 4 & S(1,4) & S(2,4) & S(3,4) & S(4,4) \\
J &= 3 & S(1,3) & S(2,3) & S(3,3) & S(4,3) \\
J &= 2 & S(1,2) & S(2,2) & S(3,2) & S(4,2) \\
J &= 1 & S(1,1) & S(2,1) & S(3,1) & S(4,1) \\
I &= 1 & & & & \\
I &= 2 & & & & \\
I &= 3 & & & & \\
I &= 4 & & & &
\end{align*}
\]
Dependence Direction Matrix

- **Direction matrix** of a loop nest contains
  - A row for each dependence direction vector between statements contained in the nest.

```plaintext
  DO I = 1, N
    DO J = 1, M
      DO K = 1, L
        A(I+1,J+1,K) = A(I,J,K) + A(I,J+1,K+1)
      ENDDO
    ENDDO
  ENDDO
```

- The direction matrix for the loop nest is: \[
  \begin{pmatrix}
    < & < & = \\
    < & = & >
  \end{pmatrix}
\]

- A loop permutation is legal if and only if
  - the direction matrix, after the same permutation is applied to its columns, has no dependence with ">” as the leftmost non-"=" direction
Optimization Profitability

- Which loop should be innermost or outermost?
  - Reduce the number of interfering data accesses between reuse of the same (or neighboring) data

- Approach: attach a cost function when each loop is placed innermost
  - Assuming cache line size is L

- Innermost K loop = N*N*N*(1+1/L)+N*N
- Innermost J loop = 2*N*N*N+N*N
- Innermost I loop = 2*N*N*N/L+N*N

- Reorder loop from innermost in the order of increasing cost
  - Limited by safety of loop interchange

DO I = 1, N
DO J = 1, N
  DO K = 1, N
    C(I, J) = C(I, J) + A(I, K) * B(K, J)
  ENDDO
ENDDO
ENDDO
Loop Blocking

- Goal: separate computation into blocks, where cache can hold the entire data used by each block

- Example

```
DO J = 1, M
  DO I = 1, N
    D(I) = D(I) + B(I,J)
  ENDDO
ENDDO
```

- Assuming $N$ is large, $2*N*M/C$ cache misses (memory accesses)

- After blocking (strip-mine-and-interchange)

```
DO jj = 1, M, T
  DO I = 1, N
    DO J = jj, MIN(jj+T-1, M)
      D(I) = D(I) + B(I, J)
    ENDDO
  ENDDO
ENDDO
```

- Assuming $T$ is small, $(M/T)*(N/C) + M*N/C$ misses
Alternative Ways of Blocking

\[
\text{DO } jj = 1, M, T \\
\text{DO } I = 1, N \\
\text{DO } J = jj, \text{MIN}(jj+T-1, M) \\
\text{D}(I) = D(I) + B(I, J) \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO}
\]

\[
\text{DO } ii = 1, N, T \\
\text{DO } J = 1, M \\
\text{DO } I = ii, \text{MIN}(ii+T-1, N) \\
\text{D}(I) = D(I) + B(I, J) \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO}
\]

\[
\text{DO } jj = 1, M, T_j \\
\text{DO } ii = 1, N, T_i \\
\text{DO } J = jj, \text{MIN}(jj+T_j-1,M) \\
\text{DO } I = ii, \text{MIN}(ii+T_i-1, N) \\
\text{D}(I) = D(I) + B(I, J) \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO}
\]
The Blocking Transformation

- The transformation takes a group of loops $L_0, \ldots, L_k$
  - Strip-mine each loop $L_i$ into two loops $L_i'$ and $L_i''$
  - Move all strip counting loops $L_0', L_1', \ldots, L_k'$ to the outside
  - Leave all strip traversing loops $L_0'', L_1'', \ldots, L_r''$ inside

- Safety of blocking
  - Strip-mining is always legal
  - Loop interchange is not always legal
  - All participating loops must be safe to be moved outside
    - Each loop has only “=“ or “<“ in all dependence vectors

- Profitability of Blocking: can enable cache reuse by an outer loop that
  - Carries small-threshold dependences (including input dep)
  - With loop index (in small stride) enumerating the contiguous dimension of an array and in no other dimension
Loop Interchange For Locality

\begin{align*}
\text{do } & J = 1, m \\
\text{do } & I = 1, n \\
& b(I) = b(I) + a(J, I) \\
\text{enddo} \\
\text{enddo}
\end{align*}

\begin{align*}
\text{do } & I = 1, n \\
\text{do } & J = 1, m \\
& b(I) = b(I) + a(J, I) \\
\text{enddo} \\
\text{enddo}
\end{align*}
Loop Blocking For Locality

```
do l = 1, n
  do J = 1, m
    b(l) = b(l) + a(J)
  enddo
enddo
```

```
do l = 1, n, b
  do J = 1, m
    do l = l1, min(l1 + b)
      b(l) = b(l) + a(J)
    enddo
  enddo
enddo
```
Optimizing for Register Usage

- Registers are part of the memory hierarchy
  - Compare to cache, compilers have complete control over what data to put in register
  - Can use registers to hold scalar variables
- Goal: use scalars to replace multiple array references
  - Enable reuse of array references in a single loop body
  - Through loop unrolling and unroll&jam
Loop Unrolling

- An enabling transformation to expose opportunities for other optimizations
  - Reduce the number of branches by a factor 4
  - Provide a bigger basic block (loop body) for local optimization
    - Better instruction scheduling and register allocation
  - Always safe if applied correctly. But must be careful when handling loop bounds do not divide unrolling factors

Original loop

```plaintext
do i = 1 to n by 1
  a(i) = a(i) + b(i)
end
```

Unrolled by 4, n = 100

```plaintext
do i = 1 to 100 by 4
  a(i) = a(i) + b(i)
  a(i+1) = a(i+1) + b(i+1)
  a(i+2) = a(i+2) + b(i+2)
  a(i+3) = a(i+3) + b(i+3)
end
```
Loop Unroll-and-Jam

DO I = 1, N*2
  DO J = 1, M
    A(I) = A(I) + B(J)
  ENDDO
ENDDO

Can we put B(J) into a register and reuse the reference?

DO I = 1, N*2, 2
  DO J = 1, M
    A(I) = A(I) + B(J)
    A(I+1) = A(I+1) + B(J)
  ENDDO
ENDDO

Unroll outer loop twice and then fuse the copies of the inner loop
Now can reuse register for B(J)
But require one more register for A

Goal: explore register reuse by outer loops
  - Compare to loop blocking
    - Different iterations of outer loop unrolled
  - Often called register blocking
    - May increase register pressure at the innermost loop

Transformation: two alternative ways to get the combined result
  - Unroll an outer loop, apply multi-level loop fusion to the unrolled loops
  - Strip-mine outer loop, interchange strip loop inside, then unroll strip loop
Safety of Unroll-and-Jam

DO I = 1, N*2
    DO J = 1, M
        A(I+1,J-1)=A(I,J)+B(I,J)
    ENDDO
ENDDO

Apply unroll-and-jam

This is wrong!

Direction vector: (<,>)
- This makes loop interchange illegal

Unroll-and-Jam is similar to blocking
- It must be safe to interchange the strip traversing outer loop with the inner loop
Array Copying vs. Scalar Replacement

- **Array copying**: dynamic layout transformation for arrays
  - Copy arrays into local buffers before computation
  - Copy modified local buffers back to array

- **Previous work**
  - Lam, Rothberg and Wolf, Temam, Granston and Jalby
    - Copy arrays after loop blocking
  - Optimizing irregular applications
    - Data access patterns not known until runtime
    - Dynamic layout transformation --- through libraries
  - **Scalar Replacement**
    - Equivalent to copying single array element into scalars
    - Carr and Kennedy: applied to inner loops

- **Unify scalar replacement and array copying (Yi LCPC’05)**
  - Improve cache and register locality
  - Automatically insert copy operations to ensure safety
  - Heuristics to reduce buffer size and copy cost
Scalar Replacement

- Convert array references to scalar variables to improve performance of register allocation

```plaintext
DO I = 1, N
  DO J = 1, M
    A(I) = A(I) + B(J)
  ENDDO
ENDDO

A(I) can be left in a register throughout the inner loop
```

```plaintext
DO I = 1, N
  T = A(I)
  DO J = 1, M
    T = T + B(J)
  ENDDO
  A(I) = T
ENDDO
```

- All loads and stores to A in the inner loop have been eliminated
- High chance of T being allocated to a register by register allocation
Unroll-and-Jam + Scalar Repl

DO I = 1, N*2, 2
  DO J = 1, M
    A(I) = A(I) + B(J)
    A(I+1) = A(I+1) + B(J)
  ENDDO
ENDDO

DO I = 1, N*2, 2
  s0 = A(I)
  s1 = A(I+1)
  DO J = 1, M
    t = B(J)
    s0 = s0 + t
    s1 = s1 + t
  ENDDO
  A(I) = s0
  A(I+1) = s1
ENDDO

- Reduce the number of memory loads by half
Array Copy: Matrix Multiplication

```c
A_buf[0:m*l] = A[0:m,0:l];
for (j=0; j<n; ++j) {
    C_buf[0:m] = C[0:m, j*m];
    for (k=0; k<l; ++k) {
        B_buf = B[k+j*l];
        for (i=0; i<m; ++i)
            C_buf[i] = C_buf[i] + alpha * A_buf[i+k*m]*B_buf;
    }
    C[0:m,j*m]=C_buf[0:m];
}
```

- Layout of data depends on ordering of their accesses
  - Often applied in sync with blocking
Array Copy: Imprecise Dependences

```
for (j=0; j<n; ++j)
    for (k=0; k<l; ++k)
        for (i=0; i<m; ++i) {
            C[f(i,j,m)] = C[i+j*m] + alpha * A[i+k*m]*B[k+j*l];
```

- Array references connected by imprecise deps
  - Cannot precisely determine a mapping between subscripts
  - Sometimes may refer to the same location, sometimes not
  - Not safe to copy into a single buffer
    - Never attempt to copy them
Profitabilities

- **When should we apply scalar replacement?**
  - Profitable unless register pressure too high
  - No overhead

- **When should we apply array copy?**
  - When regular cache conflict misses occur
  - When prefetching of array elements is needed
    - 10-40% improvement observed
  - Overhead is 0.5-8% when not beneficial

- **How should the different optimizations be combined (composed)?**
  - How should each optimization be customized to maximize their collective effectiveness?
Summary

- Common loop optimizations for improving memory performance
  - Transformation, safety and profitability
    - Loop interchange
    - Loop blocking
    - Loop unrolling
    - Loop unroll&jam
    - Scalar replacement
    - Array copying
  - Programmable composition and customization of the optimizations