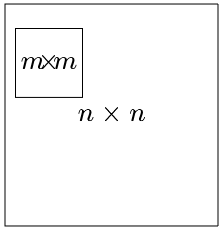
Loop Optimizations for Aggregate Array Computations

Annie Liu and Scott Stoller

[Liu Stoller 1998]: Yanhong A. Liu and Scott D. Stoller "Loop Optimizations for Aggregate Array Computations" In Proceedings of the 1998 IEEE International Conference on Computer Languages.

Example: Local Summation Problem

image: blur



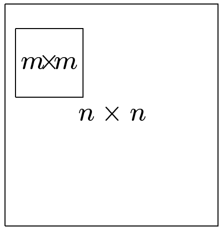
straightforward program:

$$
\begin{array}{l}
\mathbf{for}\ i := 0\ \mathbf{to}\ n-m\ \mathbf{do} \\
\quad \mathbf{for}\ j := 0\ \mathbf{to}\ n-m\ \mathbf{do} \\
\quad\quad sum[i,j] := 0; \\
\quad\quad \mathbf{for}\ k := 0\ \mathbf{to}\ m-1\ \mathbf{do} \\
\quad\quad\quad \mathbf{for}\ l := 0\ \mathbf{to}\ m-1\ \mathbf{do} \\
\quad\quad\quad\quad sum[i,j] := sum[i,j] + a[i+k, j+l]
\end{array}
$$

overlapping aggregate array redundancy $\longrightarrow O(n^2 m^2)$ time.

Example: Local Summation Problem

image: blur



optimized program:

$$
\begin{array}{l}
\ldots \\
\mathbf{for}\ i := 1\ \mathbf{to}\ n-m\ \mathbf{do} \\
\quad \mathbf{for}\ j := 1\ \mathbf{to}\ n-m\ \mathbf{do} \\
\quad\quad b[i-1+m, j] := b[i-1+m, j-1] \\
\quad\quad\quad\quad -a[i-1+m, j-1] \\
\quad\quad\quad\quad +a[i-1+m, j-1+m]; \\
\quad\quad sum[i,j] := sum[i-1,j] - b[i-1,j] \\
\quad\quad\quad\quad +b[i-1+m, j] \\
\ldots
\end{array}
$$

eliminate redundancy $\longrightarrow O(n^2)$ time: 4 $\pm$ operations per pixel.

messy boundary computations $\cdots$

Incrementalizing Aggregate Array Computations

1. identify an AAC and updates to its parameters
2. incrementalize an AAC with respect to an update,
   exploit array data dependence analysis and algebraic properties of subcomputations
3. form a new loop using incrementalized AACs,
   with appropriate initializations

- maintain additional values of AACs,
  use cache-and-prune

message: • the optimization can be done automatically.
• the automation is easy to implement.

Applications

- many can be written easily and clearly using arrays with overlapping redundancy:

  e.g., image processing, computational geometry, computer graphics, multimedia, matrix computation, list processing, graph algorithms, distributed property detection, serializing parallel programs

- eliminating overlapping redundancy is crucial for performance

  current practice: compilers can not do it;
  smart programmers manually code optimized programs without automated support.

- goal of programming-languages and compilers research:

  allow programs to be written easily and clearly and executed efficiently.

Related Work

- optimizations on loops and arrays
  - APL compiler optimizations [Guibas:Wyatt:78,Franksen:85]
    loop fusion, series exps [Goldberg:Paige:84,Warren:84,Waters:91]
  - specialization [Knoblock:Ruf:96,Auslander:et:al:96,Engler:96]
  - pipelining, loop reordering [Banerjee:90,Maydan:et:al:93,Kelly:Pugh:94,Allan:et:al:9
  - directionals [Fisher:Highnam:88,Fisher:Leon:Highnam:90,Highnam:91]

  none achieves incrementalization

- optimizations similar to incrementalization
  - strength reduction of arithmetic ops [Cocke:Kennedy:77]
  - finite differencing of set and bag ops [Paige:Koenig:82,Yelling:Strom:91]
  - promot/accumul, inc. of rec. functions [Bird:84,Smith:90,Liu:Tei:95]

  none handles aggregate computations on array

Outline

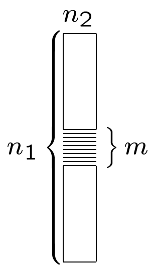
1. identify an AAC and updates to its parameters
2. incrementalize an AAC with respect to an update
3. form a new loop using incrementalized AACs

- maintain additional values of AACs

- examples and performance results

Example: Row Summation Problem

$$
\begin{array}{l}
\textbf{for } i := 0 \textbf{ to } n_1 - m \textbf{ do} \\
\quad s[i] := 0; \\
\quad \textbf{for } k := 0 \textbf{ to } m-1 \textbf{ do} \\
\quad\quad \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do} \\
\quad\quad\quad s[i] := s[i] + a[i+k, l]
\end{array}
$$



time: $O(n_1 n_2 m)$

our goal: reduce time, reduce space.

1. Aggregate Array Computation

a loop that computes accumulated quantities over array elements

$$\textbf{for } i := e_1 \textbf{ to } e_2 \textbf{ do } \; v := f(v,\; g(a[.i.], \ldots))$$

– accumulating variable: v
– contributing array reference: $a[.i.]$
– contributing function: g, uses $a[.i.]$
– accumulating function: f, updates v with results of g

<u>contributing set</u>:

$$S(A) = \{\langle a[.i.]\rangle \mid e_1 \le i \le e_2\}$$

multiple array refs $a_1 \ldots a_k$, conjunction R of all range constraints:

$$S(A) = \{\langle a_1, \ldots, a_k\rangle \mid R\}$$

1. Aggregate Array Computation — Example

$\textbf{for } i := 0 \textbf{ to } n_1 - m \textbf{ do}$
$\quad s[i] := 0;$
$\quad \textbf{for } k := 0 \textbf{ to } m - 1 \textbf{ do}$
$\quad\quad \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do}$
$\quad\quad\quad s[i] := s[i] + a[i+k, l]$

AACs:	A_k, A_l
accum var:	$s[i]$
contr arr ref:	$a[i+k, l]$
contr func:	$g(u) = u$
accum func:	$f(v, u) = v + u$

contributing sets:

$$\begin{aligned} S(A_k) &= \{\langle a[i+k, l]\rangle \mid 0 \le k < m \;\wedge\; 0 \le l < n_2\} \\ S(A_l) &= \{\langle a[i+k, l]\rangle \mid 0 \le l < n_2\} \end{aligned}$$

1. Update Operation to AAC

an update to a parameter of AAC: $\oplus_w$

– parameters are subscripts of contributing array references
– updates are loop variables of the enclosing loops
– default is "increment by 1"

example:

$\textbf{for } i := 0 \textbf{ to } n_1 - m \textbf{ do}$
$\quad s[i] := 0;$
$\quad \textbf{for } k := 0 \textbf{ to } m - 1 \textbf{ do}$
$\quad\quad \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do}$
$\quad\quad\quad s[i] := s[i] + a[i+k, l]$

AAC	param	update op
A_k	i, m, n_2	$\oplus_i$
A_l	i, k, n_2	$\oplus_i,\ \oplus_k$

goal: incrementalize AAC under update operations
— compute $A_k[i := i+1]$, abbr. as $A_k^{\oplus_i}$, using result of A_k

2. Incrementalize AAC w.r.t. Update

step 1: compute differences between contributing sets:

$$decS(A, \oplus) = S(A) - S(A^{\oplus}) \qquad incS(A, \oplus) = S(A^{\oplus}) - S(A)$$

step 2: remove contribution from $decS$, add contrib. from $incS$

$v^{\oplus} := v;$
$\textbf{for } i := last(decS(A, \oplus)) \textbf{ downto } first(decS(A, \oplus)) \textbf{ do}$
$\quad v^{\oplus} := f^{-1}(v^{\oplus}, g(a[.i.], \ldots));$
$\textbf{for } i := first(incS(A, \oplus)) \textbf{ to } last(incS(A, \oplus)) \textbf{ do}$
$\quad v^{\oplus} := f(v^{\oplus}, g(a[.i.], \ldots))$

where – inverse of f: f^{-1}.
– ordering on elements of a set of array refs: first, last, at the end

cond: if $decS$ is not empty, f must have an inverse.
if $decS$ not at end of A, or $incS$ not at end of $A^{\oplus}$, f must be assoc.& comm.

2. Incrementalize AAC w.r.t. Update — Example

step 1:

$$S(A_k) = \{\langle a[i+k,l]\rangle \mid 0 \le k < m \ \wedge \ 0 \le l < n_2\}$$
$$S(A_k^{\oplus_i}) = \{\langle a[i+1+k,l]\rangle \mid 0 \le k < m \ \wedge \ 0 \le l < n_2\}$$

$$decS(A_k, \oplus_i) = S(A_k) - S(A_k^{\oplus_i}) = \{\langle a[i,l]\rangle \mid 0 \le l < n_2\}$$
$$incS(A_k, \oplus_i) = S(A_k^{\oplus_i}) - S(A_k) = \{\langle a[i+m,l]\rangle \mid 0 \le l < n_2\}$$

step 2:

$s[i+1] := 0;$
for $k := 0$ **to** $m-1$ **do**
  for $l := 0$ **to** $n_2 - 1$ **do**
    $s[i{+}1] := s[i{+}1] + a[i{+}1{+}k, l]$

$s[i+1] := s[i];$
for $l := n_2 - 1$ **downto** 0 **do**
  $s[i+1] := s[i+1] - a[i,l];$
for $l := 0$ **to** $n_2 - 1$ **do**
  $s[i{+}1] := s[i{+}1] + a[i{+}m, l]$

$O(n_2 m) \longrightarrow O(n_2)$ **win:** $|decS(A,\oplus)| + |incS(A,\oplus)| < |S(A^{\oplus})|$

both steps can use Omega.

3. Form Incrementalized Loop

– unroll the first iteration to initialize;
– use incrementalized versions for the remaining iterations.

A_k :

$s[i] := 0;$
for $k := 0$ **to** $m-1$ **do**
  for $l := 0$ **to** $n_2 - 1$ **do**
    $s[i] := s[i] + a[i{+}k, l]$

inc-ed $A_k^{\oplus_i}$:

$s[i+1] := s[i];$
for $l := n_2 - 1$ **downto** 0 **do**
  $s[i+1] := s[i+1] - a[i,l];$
for $l := 0$ **to** $n_2 - 1$ **do**
  $s[i{+}1] := s[i{+}1] + a[i{+}m, l]$

optimized program:

init with $i = 0$:
$s[0] := 0;$
for $k := 0$ **to** $m-1$ **do**
  for $l := 0$ **to** $n_2 - 1$ **do**
    $s[0] := s[0] + a[k, l]$

for $i := 1$ **to** $n_1 - m$ **do**

inc with i dec 1:
  $s[i] := s[i{-}1];$
  for $l := n_2 - 1$ **downto** 0 **do**
    $s[i] := s[i] - a[i{-}1, l];$
  for $l := 0$ **to** $n_2 - 1$ **do**
    $s[i] := s[i] + a[i{-}1{+}m, l]$

Maintain Additional Values of AACs

cache-and-prune:

- transform AACs to store all intermediate results and related auxiliary information
- incrementalize all AACs from one iteration to the next
- prune out stored values that were not used

when forming incrementalized loop, use AACs that store additional values for the initial iteration.

Maintain Additional Values of AACs (1)

cache:
⋆ store in fresh array, with enclosing loop variables as subscripts
∘ for associative f, save results of inner AACs independently
▷ remove subscript redundancy

⋆ **for** $i := 0$ **to** $n_1 - m$ **do**
  $s[i] := 0;$
  for $k := 0$ **to** $m-1$ **do**
    for $l := 0$ **to** $n_2 - 1$ **do**
      $s[i] := s[i] + a[i{+}k, l];$
    $s_1[i,k] := s[i]$

∘ $s[i] := 0;$
  for $k := 0$ **to** $m-1$ **do**
    $s_1[i,k] := 0;$
    for $l := 0$ **to** $n_2 - 1$ **do**
      $s_1[i,k] := s_1[i,k] + a[i{+}k, l];$
    $s[i] := s[i] + s_1[i,k]$

▷ change $s_1[i,k]$ to $s_1[i+k]$

Maintain Additional Values of AACs (2)

incrementalize:

– only one s_1 remains to be computed iteratively

– computing $s[i+1]$ has a single-element in $decS$, same for $incS$

$$\begin{array}{l}
s_1[i+m] := 0; \\
\mathbf{for}\ l := 0\ \mathbf{to}\ n_2 - 1\ \mathbf{do} \\
\quad s_1[i+m] := s_1[i+m] + a[i+m, l]; \\
s[i+1] := s[i] - s_1[i] + s_1[i+m]
\end{array}$$

Maintain Additional Values of AACs (3)

prune leaves the code unchanged.

form incrementalized loop:

$$\begin{array}{r|l}
 & s[0] := 0; \\
\text{init} & \mathbf{for}\ k := 0\ \mathbf{to}\ m-1\ \mathbf{do} \\
\text{using (I)} & \quad s_1[k] := 0; \\
\text{with } i = 0 & \quad \mathbf{for}\ l := 0\ \mathbf{to}\ n_2 - 1\ \mathbf{do} \\
 & \quad\quad s_1[k] := s_1[k] + a[k, l]; \\
 & \quad s[0] := s[0] + s_1[k]; \\
 & \mathbf{for}\ i := 1\ \mathbf{to}\ n_1 - m\ \mathbf{do} \\
\text{inc} & \quad s_1[i-1+m] := 0; \\
\text{using (II)} & \quad \mathbf{for}\ l := 0\ \mathbf{to}\ n_2 - 1\ \mathbf{do} \\
\text{with } i \text{ dec } 1 & \quad\quad s_1[i-1+m] := s_1[i-1+m] + a[i-1+m, l]; \\
 & \quad s[i] := s[i-1] - s_1[i-1] + s_1[i-1+m]
\end{array}$$

improved more: $2n_1n_2 \longrightarrow n_1n_2$

Summary and Discussion

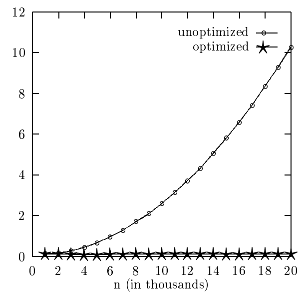
- – identify AACs and update operations
  – store values of all AACs not stored
  – incrementalize all AACs under update operations
  analyze loop vars and array subscripts, generate code for update
  – form new loops
- multiple loops: transform inside out
- – can all be automated
  – can be easily implemented
- the principle of the method is general

Partial Sums Example

$$\begin{array}{l}
\mathbf{for}\ i := 1\ \mathbf{to}\ n\ \mathbf{do} \\
\quad s[i] := 0; \\
\quad \mathbf{for}\ j := 1\ \mathbf{to}\ i\ \mathbf{do} \\
\quad\quad s[i] := s[i] + a[j]
\end{array}$$

$$\begin{array}{l}
s[1] := a[1]; \\
\mathbf{for}\ i := 2\ \mathbf{to}\ n\ \mathbf{do} \\
\quad s[i] := s[i-1] + a[i]
\end{array}$$

running time (in seconds):

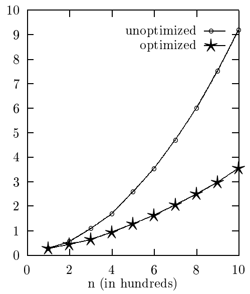


Local Summation Example

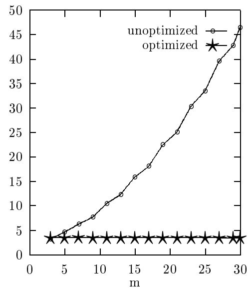
```
sum[0,0] := 0;
for k := 0 to m-1 do
    b[k,0] := 0;
    for l := 0 to m-1 do
        b[k,0] := b[k,0] + a[k,l];
    sum[0,0] := sum[0,0] + b[k,0];
for j := 1 to n-m do
    sum[0,j] := 0;
    for k := 0 to m-1 do
        b[k,j] := b[k,j-1] - a[k,j-1] + a[k,j-1+m];
        sum[0,j] := sum[0,j] + b[k,j];
for i := 1 to n-m do                                                     ⋆
    b[i-1+m,0] := 0;
    for l := 0 to m-1 do
        b[i-1+m,0] := b[i-1+m,0] + a[i-1+m,l];
    sum[i,0] := sum[i-1,0] - b[i-1,0] + b[i-1+m,0];
    for j := 1 to n-m do                                                 ⋆
        b[i-1+m,j] := b[i-1+m,j-1] - a[i-1+m,j-1] + a[i-1+m,j-1+m];      ⋆
        sum[i,j] := sum[i-1,j] - b[i-1,j] + b[i-1+m,j]                   ⋆
```


Local Summation Example

running time (in seconds):



$m = 10$



$n = 1000$

Conclusion

- the optimization is new, powerful, automatable
- it combines existing methods and techniques
- the underlying principle is general
- open:
  - floating-point precision
  - space consumption
  - more general AACs