Lecture 25

Memory Hierarchy Optimizations & Locality Analysis
Caches: A Quick Review

• How do they work?
• Why do we care about them?
• What are typical configurations today?
• What are some important cache parameters that will affect performance?
Optimizing Cache Performance

- Things to enhance:
  - temporal locality
  - spatial locality

- Things to minimize:
  - conflicts (i.e. bad replacement decisions)

What can the compiler do to help?
Two Things We Can Manipulate

- **Time:**
  - When is an object accessed?

- **Space:**
  - Where does an object exist in the address space?

*How do we exploit these two levers?*
Time: Reordering Computation

• What makes it difficult to know *when* an object is accessed?

• How can we predict a *better time* to access it?
  • What information is needed?

• How do we know that this would be *safe*?
**Space: Changing Data Layout**

- What do we know about an object's location?
  - scalars, structures, pointer-based data structures, arrays, code, etc.

- How can we tell what a better layout would be?
  - how many can we create?

- To what extent can we safely alter the layout?
Types of Objects to Consider

- Scalars
- Structures & Pointers
- Arrays
Scalars

- Locals

- Globals

- Procedure arguments

- Is cache performance a concern here?
- If so, what can be done?

```c
int x;
double y;
foo(int a){
    int i;
    ...
    x = a*i;
    ...
}
```
Structures and Pointers

• What can we do here?
  • within a node
  • across nodes

```c
struct {
  int count;
  double velocity;
  double inertia;
  struct node *neighbors[N];
} node;
```

• What limits the compiler’s ability to optimize here?
Arrays

double A[N][N], B[N][N];
...
for i = 0 to N-1
  for j = 0 to N-1
    A[i][j] = B[j][i];

- usually accessed within loops nests
  - makes it easy to understand “time”
- what we know about array element addresses:
  - start of array?
  - relative position within array
Handy Representation: “Iteration Space”

for \( i = 0 \) to \( N-1 \)
  for \( j = 0 \) to \( N-1 \)
    \( A[i][j] = B[j][i] \);

• each position represents an iteration
Visitation Order in Iteration Space

for $i = 0$ to $N-1$
  for $j = 0$ to $N-1$
    $A[i][j] = B[j][i]$;

- Note: iteration space ≠ data space
When Do Cache Misses Occur?

for \( i = 0 \) to \( N-1 \)
for \( j = 0 \) to \( N-1 \)
\[ A[i][j] = B[j][i]; \]
When Do Cache Misses Occur?

for \( i = 0 \) to \( N-1 \)
\[
\text{for } j = 0 \text{ to } N-1
\]
\[
A[i+j][0] = i \times j;
\]
Optimizing the Cache Behavior of Array Accesses

- We need to answer the following questions:
  - when do cache misses occur?
    - use "locality analysis"
  - can we change the order of the iterations (or possibly data layout) to produce better behavior?
    - evaluate the cost of various alternatives
  - does the new ordering/layout still produce correct results?
    - use "dependence analysis"
Examples of Loop Transformations

- Loop Interchange
- Cache Blocking
- Skewing
- Loop Reversal
- ...

(we will briefly discuss the first two)
Loop Interchange

for $i = 0$ to $N-1$
for $j = 0$ to $N-1$
    $A[j][i] = i \times j$;

(assuming $N$ is large relative to cache size)
Cache Blocking (aka “Tiling“)

for $i = 0$ to $N-1$
for $j = 0$ to $N-1$
    $f(A[i], A[j])$

for $JJ = 0$ to $N-1$ by $B$
    for $i = 0$ to $N-1$
        for $j = JJ$ to $\max(N-1, JJ+B-1)$
            $f(A[i], A[j])$

now we can exploit temporal locality
Impact on Visitation Order in Iteration Space

for $i = 0$ to $N-1$
  for $j = 0$ to $N-1$
    $f(A[i], A[j]);$
  
for $JJ = 0$ to $N-1$ by $B$
  for $i = 0$ to $N-1$
    for $j = JJ$ to max($N-1, JJ+B-1$)
      $f(A[i], A[j]);$
Cache Blocking in Two Dimensions

for $JJ = 0$ to $N-1$ by $B$
for $KK = 0$ to $N-1$ by $B$
for $i = 0$ to $N-1$
for $j = JJ$ to max($N-1, JJ+B-1$)
for $k = KK$ to max($N-1, KK+B-1$)
c[$i, k$] += a[$i, j$]*b[$j, k$];

• brings square sub-blocks of matrix "b" into the cache
• completely uses them up before moving on
Predicting Cache Behavior through “Locality Analysis”

- Definitions:
  - **Reuse**: accessing a location that has been accessed in the past
  - **Locality**: accessing a location that is now found in the cache

- Key Insights
  - Locality only occurs when there is reuse!
  - BUT, reuse does not necessarily result in locality.
    - why not?
Steps in Locality Analysis

1. Find data reuse
   • if caches were infinitely large, we would be finished

2. Determine “localized iteration space”
   • set of inner loops where the data accessed by an iteration is expected to fit within the cache

3. Find data locality:
   • reuse \( \cap \) localized iteration space \( \Rightarrow \) locality
Types of Data Reuse/Locality

for $i = 0$ to $2$
for $j = 0$ to $100$

Spatial  Temporal  Group
Reuse Analysis: Representation

for i = 0 to 2
for j = 0 to 100
A[i][j] = B[j][0] + B[j+1][0];

- Map $n$ loop indices into $d$ array indices via array indexing function:

$$\vec{f}(\vec{\nu}) = H\vec{\nu} + \vec{c}$$

- $A[i][j] = A \left( \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$

- $B[j][0] = B \left( \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$

- $B[j+1][0] = B \left( \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right)$
Finding Temporal Reuse

- Temporal reuse occurs between iterations $\vec{v}_1$ and $\vec{v}_2$ whenever:
  $$H\vec{v}_1 + \vec{c} = H\vec{v}_2 + \vec{c}$$
  $$H(\vec{v}_1 - \vec{v}_2) = \vec{0}$$

- Rather than worrying about individual values of $\vec{v}_1$ and $\vec{v}_2$, we say that reuse occurs along direction vector $\vec{r}$ when:
  $$H(\vec{r}) = \vec{0}$$

- Solution: compute the nullspace of $H$
Temporal Reuse Example

for i = 0 to 2
   for j = 0 to 100
      A[i][j] = B[j][0] + B[j+1][0];

• Reuse between iterations \((i_1,j_1)\) and \((i_2,j_2)\) whenever:

\[
\begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
i_1 \\
j_1
\end{bmatrix}
+ \begin{bmatrix}
1 \\
0
\end{bmatrix}
= \begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
i_2 \\
j_2
\end{bmatrix}
+ \begin{bmatrix}
1 \\
0
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
i_1 - i_2 \\
j_1 - j_2
\end{bmatrix}
= \begin{bmatrix}
0 \\
0
\end{bmatrix}
\]

• True whenever \(j_1 = j_2\), and regardless of the difference between \(i_1\) and \(i_2\).
  • i.e. whenever the difference lies along the nullspace of \[
\begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix},
\]
  which is \(\text{span}\{(1,0)\}\) (i.e. the outer loop).
More Complicated Example

for $i = 0$ to $N-1$
  for $j = 0$ to $N-1$
    $A[i+j][0] = i \times j$;

$A[i+j][0] = A \left( \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$

• Nullspace of $\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} = \text{span}\{(1, -1)\}$. 
**Computing Spatial Reuse**

- Replace last row of $H$ with zeros, creating $H_s$
- Find the nullspace of $H_s$

**Result**: vector along which we access the same row
Computing Spatial Reuse: Example

for $i = 0$ to 2
  for $j = 0$ to 100

$A[i][j] = A \left( \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$

- $H_s = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$

- Nullspace of $H_s = \text{span}\{(0,1)\}$
  - i.e. access same row of $A[i][j]$ along inner loop
Computing Spatial Reuse: More Complicated Example

for $i = 0$ to $N-1$
    for $j = 0$ to $N-1$
        $A[i+j] = i*j$;

$A[i+j] = A \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix}$

- $H_s = \begin{bmatrix} 0 & 0 \end{bmatrix}$
- Nullspace of $H = \text{span}\{(1,-1)\}$
- Nullspace of $H_s = \text{span}\{(1,0),(0,1)\}$
Group Reuse

for $i = 0$ to 2
  for $j = 0$ to 100

- Only consider “uniformly generated sets”
  - index expressions differ only by constant terms
- Check whether they actually do access the same cache line
- Only the “leading reference” suffers the bulk of the cache misses
Localized Iteration Space

- Given finite cache, when does reuse result in locality?

for $i = 0$ to $2$
for $j = 0$ to $8$
  $A[i][j] = B[j][0] + B[j+1][0];$

Localized: both $i$ and $j$ loops (i.e. span$\{(1,0),(0,1)\}$)

for $i = 0$ to $2$
for $j = 0$ to $1000000$
  $A[i][j] = B[j][0] + B[j+1][0];$

Localized: $j$ loop only (i.e. span$\{(0,1)\}$)

- Localized if accesses less data than effective cache size
**Computing Locality**

- **Reuse Vector Space** ∩ **Localized Vector Space** ⇒ **Locality** Vector Space

- **Example:**
  ```
  for i = 0 to 2
      for j = 0 to 100
          A[i][j] = B[j][0] + B[j+1][0];
  ```

- **If both loops are localized:**
  - span{(1,0)} ∩ span{(1,0),(0,1)} ⇒ span{(1,0)}
  - i.e. temporal reuse does result in temporal locality

- **If only the innermost loop is localized:**
  - span{(1,0)} ∩ span{(0,1)} ⇒ span{}
  - i.e. no temporal locality