Lecture 16:
Memory Hierarchy Optimizations

I. Caches: A Quick Review
II. Iteration Space & Loop Transformations
III. Locality Analysis

[ALSU 7.4.2-7.4.3, 11.2-11.5]
I. 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

Example: Can rearrange field order to improve cache performance

• What limits the compiler's ability to optimize here?

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

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
II. Iteration Space and Loop Transformations

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

• each position represents an iteration (not an array element)
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 $N-1$

$A[i][j] = B[j][i]$;

Assume row major order, $N$ large, 2 elements per cache line

When Do Cache Misses Occur?

```c
double A[2N-1][N];

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

Assume row major order, 2 elements per cache line

**Row major layout of A:**


If N large then all misses. What if N is small? see above
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: iterate thru iteration space in the loops at an angle
- Loop Reversal: execute iterations in a loop in reverse order
- ...

*(we will briefly discuss the first two; see ALSU 11.7.8 for others)*
Loop Interchange

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

Assume row major order, $N$ large, 4 elements per cache line
Cache Blocking (aka “Tiling”)

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

for $i = 0$ to $N-1$
for $j = JJ$ to $\min(N-1, JJ+L-1)$
$f(A[i], A[j])$;

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 $L$
      for $i = 0$ to $N - 1$
         for $j = JJ$ to $\min(N - 1, JJ + L - 1)$
            $f(A[i], A[j])$
Cache Blocking in Two Dimensions

- brings square sub-blocks of matrix “b” into the cache
- completely uses them up before moving on
- reduces the number of misses from $\frac{N^3}{L}$ or $N^3$ to only $\frac{2N^3}{LC}$ (C=cache size, L=line size)
III. 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

double A[3][N], B[N][3];

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

(assume row-major, 2 elements per cache line, N small)
Reuse Analysis: Representation

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

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

\[
\vec{f}(\vec{i}) = H\vec{i} + \vec{c}
\]

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

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

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

\[
\text{for } i = 0 \text{ to } 2 \\
\quad \text{for } j = 0 \text{ to } 100 \\
\quad A[i][j] = B[j][0] + B[j+1][0];
\]

- For \(B[j+1][0]\) 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

\[
\text{for } i = 0 \text{ to } N-1 \\
\quad \text{for } j = 0 \text{ to } N-1 \\
\quad A[i+j][0] = i \times j;
\]

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

- Nullspace of \[
\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}
\] = span\{(1, -1)\}, i.e. when \(\Delta i = -\Delta j\).
Computing Spatial Reuse

- Assume two array elements share the same cache line iff they differ only in the last dimension
  - E.g., share the same row in a 2-dimensional array
  - Why is this a reasonable approximation? row major order
  - What are its limitations? A row is made up of many cache lines
    Large row could be larger than the cache

- 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

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

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

- Nullspace of $H_s = \text{span}\{(0,1)\}$, i.e., the inner loop
  - access same row of $A[i][j]$ along inner loop
**Group Reuse (reuse from different static accesses)**

\[
\text{for } i = 0 \text{ to } 2 \\
\text{for } j = 0 \text{ to } 100 \\
A[i][j] = B[j][0] + B[j+1][0];
\]

\[
H = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}
\]

- Limit the analysis to consider only accesses with same \(H\)
  - i.e., index expressions that differ only in their constant terms
- Determine when access same location (temporal) or same row (spatial)
- Only the “leading reference” suffers the bulk of the cache misses

![Diagram of arrays A[i][j] and B[j][0] and B[j+1][0]]
Localized Iteration Space

- Given finite cache, when does reuse result in locality?
- Localized if accesses less data than effective cache size

\[
\text{for } i = 0 \text{ to } 2 \\
\text{for } j = 0 \text{ to } 7 \\
A[i][j] = B[j][0] + B[j+1][0];
\]

\[
\text{for } i = 0 \text{ to } 2 \\
\text{for } j = 0 \text{ to } 1000000 \\
A[i][j] = B[j][0] + B[j+1][0];
\]

\text{Localized:} \text{ both } i \text{ and } j \text{ loops}

\text{Localized:} \text{ } j \text{ loop only}
Computing Locality

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

- **Example:**
  
  ```plaintext
  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:**
  - \( \text{span}\{(1,0)\} \cap \text{span}\{(1,0),(0,1)\} \Rightarrow \text{span}\{(1,0)\} \)
  - i.e. temporal reuse does result in **temporal locality**

- **If only the innermost loop is localized:**
  - \( \text{span}\{(1,0)\} \cap \text{span}\{(0,1)\} \Rightarrow \text{span}\{} \)
  - i.e. no **temporal locality**
Today’s Class: Memory Hierarchy Optimizations

I. Caches: A Quick Review
II. Iteration Space & Loop Transformations
III. Locality Analysis

Wednesday’s Class

• Array Dependence Analysis & Parallelization
  – ALSU 11.6