CMPT 295
Unit – Memory Hierarchy
Lecture 34 – Locality, Memory Hierarchy and Caching
Last Lecture

1. Superscalar processor
   - Can execute multiple instructions in one clock cycle since processor has multiple functional units
   - Rearrange execution order of instructions to avoid hazards that negatively impact throughput -> called out of order execution

2. Optimization based on exploiting instruction-level parallelism
   - Software developer can restructure code to take advantage of instruction-level parallelism
     - Loop unrolling + reassociating + accumulating
Goal for this unit

- Learn to optimize our code based on our understanding of memory hierarchy and caching
  - Taking advantage of *locality*
Today’s Menu

- Locality – Cache-friendly code optimization
- Memory hierarchy
- Caching
- Cache Memories
Memory model so far

- Linear (contiguous) array of bytes
- If the memory address resolution is a byte, then …
  - Each byte is addressable
  - Each byte is accessible in constant time
- 2D array can be viewed as:
- And in memory, it is stored as:

<table>
<thead>
<tr>
<th>A_{0,0}</th>
<th>A_{0,1}</th>
<th>A_{0,2}</th>
<th>A_{0,3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_{1,0}</td>
<td>A_{1,1}</td>
<td>A_{1,2}</td>
<td>A_{1,3}</td>
</tr>
<tr>
<td>A_{2,0}</td>
<td>A_{2,1}</td>
<td>A_{2,2}</td>
<td>A_{2,3}</td>
</tr>
</tbody>
</table>
Do these two functions differ in their performance?

// M = 3, N = 4
int sumArray1(int A[M][N]) {
    int i, j, sum;
    sum = 0;
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++)
            sum += A[i][j];
    }
    return sum;
}

// M = 3, N = 4
int sumArray2(int A[M][N]) {
    int i, j, sum;
    sum = 0;
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++)
            sum += A[j][i];
    }
    return sum;
}
Locality

- Principle of Locality
  - Programs tend to use data and instructions with addresses near or equal to those they have used recently

- Temporal locality
  - Recently referenced data or instructions are likely to be referenced again in the near future

- Spatial locality
  - Data or instructions with nearby addresses tend to be referenced close together in time

- Goal
  - Being able to look at code and get a qualitative sense of its locality is a key skill for a software developer
Locality and stride

// M = 3, N = 4
int sumArray1(int A[M][N]) {
    int i, j, sum;
    sum = 0;
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++)
            sum += A[i][j];
    }
    return sum;
}

// M = 3, N = 4
int sumArray2(int A[M][N]) {
    int i, j, sum;
    sum = 0;
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++)
            sum += A[j][i];
    }
    return sum;
}

Memory address: 0xFF5EFD80 ...D84 ...D88 ...D8C ...D90 ...

Stride-1: each iteration of inner loop accesses an element 1 away from previous element

Stride-N: each iteration of inner loop accesses an element N away from previous element
Conclusion: good or bad locality?

Function **sumArray1** has __good__ spatial locality with respect to array A.

Function **sumArray2** has __bad__ spatial locality with respect to array A.

Function **sumArray1** has __good__ temporal locality with respect to sum.

Function **sumArray2** has __good__ temporal locality with respect to sum.

```c
// M = 3, N = 4
int sumArray1(int A[M][N]) {
    int i, j, sum;
    sum = 0;
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++)
            sum += A[i][j];
    }
    return sum;
}

// M = 3, N = 4
int sumArray2(int A[M][N]) {
    int i, j, sum;
    sum = 0;
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++)
            sum += A[j][i];
    }
    return sum;
}
```
The statements below describe the code above:

- **Data references**
  - Array elements are referenced in succession (stride-1 reference pattern)
  - Variable total is referenced at each iteration

- **Instruction references**
  - Instructions are referenced in sequence
  - Loop is cycled through repeatedly

- Which ones are examples of **spatial locality** and which ones are examples of **temporal locality**?

```c
total = 0;
for (i = 0; i < n; i++)
    total += B[i];
return total;
```
Let’s try again! – Homework!

- Can we rearrange the loops so that the function scans the 3D array $A$ with a stride-1 reference pattern (and thus has good spatial locality)?

```c
int sumArray(int A[N][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += A[k][i][j];

    return sum;
}
```
Another example of **temporal** locality 
(within the execution of 1 iteration)

```c
// Around 2970 clock cycles
int str_alnum3(char *s) {
    int i;
    int len = strlen(s);
    for (i = 0; i < len; i++) {
        if (!(((s[i] >= 'a') && (s[i] <= 'z')) ||
             ((s[i] >= 'A') && (s[i] <= 'Z')) ||
             ((s[i] >= '0') && (s[i] <= '9')))) {
            return 0;
        }
    }
    return 1;
}
```

Also, an example of **spatial** locality 
(visiting adjacent letter in the string at each iteration of the loop)

```c
// Around 2917 clock cycles
int str_alnum4(char *s) {
    int i;
    int len = strlen(s);
    char aChar;
    for (i = 0; i < len; i++) {
        aChar = s[i];
        if (!(((aChar >= 'a') && (aChar <= 'z')) ||
             ((aChar >= 'A') && (aChar <= 'Z')) ||
             ((aChar >= '0') && (aChar <= '9')))) {
            return 0;
        }
    }
    return 1;
}
```
Memory structured to take advantage of locality

- When data or instructions are accessed often, they are copied (hence accessible) from faster memory components.
- What do you mean faster memory components?
  - In reality, memory is structured in hierarchy of memory components (storage devices).
  - Each having different capacity, cost and access time.
- So far, we know CPU registers and main memory (RAM):
  - Size-> CPU registers: ~KB   main memory: ~GB
  - Access time -> CPU registers: 0.33ns main memory: 20ns
Cache

- DRAM is big, but slow
  - waiting 60 cycles per access is impractical
  - both fetch & memory stages would have to be expanded

- **Strategy**: Cache
  - Add a layer to memory hierarchy -> using smaller, faster memory
    - Store frequently used memory references
  - Takes advantage of locality principle:
    - Temporal locality: recent memory references will be referenced again soon
    - Spatial locality: memory nearby recent memory references will be referenced soon

“Cache” means “a hidden storage place”
Example: food cache
Adding caching to memory hierarchy

CPU Registers

~ 4 cycles per access

Cache (SRAM)

~ 60 cycles per access

Main Memory (DRAM)

Cache holds several lines, each 64 bytes
- each line can hold a copy of 64 bytes from main memory (or be empty)

CPU interacts with cache instead of main memory
- if line in cache, data is served (cache hit)
- if line is not in cache, cache must load it from main memory (cache miss)

holds a small number of values for calculations and most frequently used data

holds a subset of main memory

holds program text & data
Example Memory Hierarchy

- **L0 - CPU Registers**
  - Backs up subset of
  - Latency: 1 clock cycle per access
  - Size: 1 KB

- **L1 Cache**
  - i-cache (instruction)
  - d-cache (data)
  - Backs up subset of
  - Latency: 4 clock cycles
  - Size: 32 KB

- **L2 Cache**
  - Unified (instruction + data)
  - Backs up subset of
  - Latency: 10 clock cycles
  - Size: 256 KB

- **L3 Cache**
  - Unified (instruction + data)
  - Backs up subset of
  - Latency: 40-75 clock cycles
  - Size: 8 MB

- **L4 - Main Memory**
  - Backs up subset of
  - Latency: 200 clock cycles
  - Size: 16 GB

- **L5 - Local secondary storage (local disks)**
  - Backs up subset of
  - Latency: M’s clock cycles
  - Size: TB’s

Storage devices:
- Faster, more expensive (per byte), smaller
- Slower, cheaper (per byte), larger
Summary

- Speed separation between registers (1 cycle / access) and main memory (60 cycles / access) is huge
- To narrow this gap, add cache
  - Use faster memory components (SRAM: 4 cycles / access) to hold copy of portion of main memory likely to be used in near future
  - Takes advantage of locality
    - Temporal locality: access the same location again soon
    - Spatial locality: access a nearby location soon
- Understanding memory hierarchy and caching allows us to write cache-friendly code, i.e., programs with good locality which tend to run faster because such code maximizes data access from higher levels of memory hierarchy (i.e., from faster memory components)
Next Lecture

- Locality – Cache-friendly code optimization
- Memory hierarchy
- Caching
- Cache Memories