How to Write Fast Numerical Code
Spring 2017

Lecture: Memory hierarchy, locality, caches

Instructor: Markus Püschel
TA: Alen Stojanov, Georg Ofenbeck, Gagandeep Singh

Organization

- Temporal and spatial locality
- Memory hierarchy
- Caches


Part of these slides are adapted from the course associated with this book
Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus bandwidth doubled every 36 months

Core i7 Haswell:
Peak performance:
2 AVX three operand (FMA) ops/cycles consumes up to 192 Bytes/cycle

Core i7 Haswell:
Bandwidth
16 Bytes/cycle

Solution: Caches/Memory hierarchy

Typical Memory Hierarchy

CPU registers hold words retrieved from L1 cache
L1 cache holds cache lines retrieved from L2 cache
L2 cache holds cache lines retrieved from main memory
Main memory holds disk blocks retrieved from local disks
Local disks hold files retrieved from disks on remote network servers
remote secondary storage (tapes, distributed file systems, Web servers)
local secondary storage (local disks)
main memory (DRAM)
on-chip L2 cache (SRAM)
on-chip L1 cache (SRAM)
registers

L0:
L1:
L2:
L3:
L4:
L5:

Smaller, faster, costlier per byte
Larger, slower, cheaper per byte

CPU
Main Memory
Why Caches Work: Locality

- **Locality**: Programs tend to use data and instructions with addresses near or equal to those they have used recently. *History of locality*

- **Temporal locality**: Recently referenced items are likely to be referenced again in the near future

- **Spatial locality**: Items with nearby addresses tend to be referenced close together in time
Example: Locality?

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Data:**
  - Temporal: `sum` referenced in each iteration
  - Spatial: array `a[]` accessed consecutively

- **Instructions:**
  - Temporal: loops cycle through the same instructions
  - Spatial: instructions referenced in sequence

**Being able to assess the locality of code is a crucial skill for a performance programmer**

Locality Example #1

```
int sum_array_rows(double a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```
Locality Example #2

```c
int sum_array_cols(double a[M][N]) {
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Locality Example #3

```c
int sum_array_3d(double a[M][N][K]) {
    int i, j, k, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < K; k++)
                sum += a[k][i][j];
    return sum;
}
```

How to improve locality?

![Graph showing performance in flops/cycle vs k-i-j-i-j-k for M = N = K]

CPU: Intel(R) Core(TM) i7-4980HQ CPU @ 2.80GHz
gcc: Apple LLVM version 8.0.0 (clang-800.0.42.1)
flags: -O3 -fno-vectorize
Operational Intensity Again

- Definition: Given a program P, assume cold (empty) cache

  Operational intensity: \( I(n) = \frac{W(n)}{Q(n)} \)

  * #flops (input size n)
  * #bytes transferred cache ↔ memory (for input size n)

- Examples: Determine asymptotic bounds on \( I(n) \)
  - Vector sum: \( y = x + y \) \( O(1) \)
  - Matrix-vector product: \( y = Ax \) \( O(1) \)
  - Fast Fourier transform \( O(\log(n)) \)
  - Matrix-matrix product: \( C = AB + C \) \( O(n) \)

Compute/Memory Bound

- A function/piece of code is:
  - Compute bound if it has high operational intensity
  - Memory bound if it has low operational intensity

- Relationship between operational intensity and locality?
  - They are closely related
  - Operational intensity only describes the boundary last level cache/memory
Effects

**FFT:** \( I(n) = O(\log(n)) \)

**MMM:** \( I(n) = O(n) \)

**Graphs:**
- **Discrete Fourier Transform (DFT) on 2 x Core 2 Duo 3 GHz (single)**
- **Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz (double)**

- **Up to 40-50% peak**
- **Performance drop outside last level cache (LLC)**
- **Most time spent transferring data**

- **Up to 80-90% peak**
- **Performance can be maintained outside LLC**
- **Cache miss time compensated/hidden by computation**

Cache

- **Definition:** Computer memory with short access time used for the storage of frequently or recently used instructions or data

- **Spatial locality** is supported by transferring data in blocks
  - Core family: one block = 64 B = 8 doubles

- **Naturally supports temporal locality**

Diagram:
- CPU → Cache → Main Memory
## General Cache Mechanics

![Diagram](https://via.placeholder.com/150)

**Cache**

<table>
<thead>
<tr>
<th>4</th>
<th>9</th>
<th>10</th>
<th>3</th>
</tr>
</thead>
</table>

**Memory**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Smaller, faster, more expensive memory caches a subset of the blocks.

Data is copied in block-sized transfer units.

Larger, slower, cheaper memory viewed as partitioned into “blocks”.

## General Cache Concepts: Hit

### Request: 14

**Data in block b is needed**

**Block b is in cache:**

Hit!
General Cache Concepts: Miss

Data in block b is needed

Block b is not in cache: Miss!

Block b is fetched from memory

Block b is stored in cache

• Placement policy: determines where b goes
• Replacement policy: determines which block gets evicted (victim)

Cache

Memory

Request: 12

Types of Cache Misses (The 3 C’s)

- **Compulsory (cold) miss**
  Occurs on first access to a block

- **Capacity miss**
  Occurs when working set is larger than the cache

- **Conflict miss**
  Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot

- Not a clean classification but still useful
Cache Structure

- Draw a direct mapped cache ($E = 1$, $B = 4$ doubles, $S = 8$)
- Show how blocks are mapped into cache

Example (S=8, E=1)

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}

int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache, a[0][0] goes here

B = 32 byte = 4 doubles

blackboard
Cache Structure

- Add associativity (E = 2, B = 4 doubles, S = 8)
- Show how elements are mapped into cache

Example (S=4, E=2)

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}

int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j
assume: cold (empty) cache, a[0][0] goes here
B = 32 byte = 4 doubles
General Cache Organization (S, E, B)

- \( E = 2^e \) lines per set
- \( S = 2^s \) sets
- \( B = 2^b \) bytes per cache block (the data)

Cache size:
\[ S \times E \times B \text{ data bytes} \]

Cache Read

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

Address of word:
- Tag
- Set index
- Block offset
- Data begins at this offset
Terminology

- **Direct mapped cache:**
  - Cache with $E = 1$
  - Means every block from memory has a unique location in cache

- **Fully associative cache**
  - Cache with $S = 1$ (i.e., maximal $E$)
  - Means every block from memory can be mapped to any location in cache
  - In practice too expensive to build
  - One can view the register file as a fully associative cache

- **LRU (least recently used) replacement**
  - when selecting which block should be replaced (happens only for $E > 1$), the least recently used one is chosen

Small Example, Part 1

Cache: $E = 1$ (direct mapped), $S = 2$, $B = 16$ (2 doubles)
Array (accessed twice in example): $x = x[0], \ldots, x[7]$

Access pattern: 0123456701234567
Hit/Miss: MHMHMHMHMHMHMHMH

Result: 8 misses, 8 hits
Spatial locality: yes
Temporal locality: no

% Matlab style code
for $j = 0:1$
  for $i = 0:7$
    access($x[i]$)
Small Example, Part 2

Array (accessed twice in example)
\[ x = x[0], \ldots, x[7] \]

Cache:
- \( E = 1 \) (direct mapped)
- \( S = 2 \)
- \( B = 16 \) (2 doubles)

Access pattern:
0246135702461357

Hit/miss:
MMMMMMMMMMMMMMMM

Result: 16 misses
Spatial locality: no
Temporal locality: no

Small Example, Part 3

Array (accessed twice in example)
\[ x = x[0], \ldots, x[7] \]

Cache:
- \( E = 1 \) (direct mapped)
- \( S = 2 \)
- \( B = 16 \) (2 doubles)

Access pattern:
0123012345674567

Hit/miss:
MHMHMHMHMHMHMHMHMH

Result: 4 misses, 12 hits (is optimal, why?)
Spatial locality: yes
Temporal locality: yes
### Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache: misses / accesses = 1 – hit rate

- **Hit Time**
  - Time to deliver a block in the cache to the processor
  - Core 2:
    - 3 clock cycles for L1
    - 14 clock cycles for L2

- **Miss Penalty**
  - Additional time required because of a miss
  - Core 2: about 100 cycles for L2 miss

### What about writes?

- **What to do on a write-hit?**
  - **Write-through**: write immediately to memory
  - **Write-back**: defer write to memory until replacement of line

- **What to do on a write-miss?**
  - **Write-allocate**: load into cache, update line in cache
  - **No-write-allocate**: writes immediately to memory

<table>
<thead>
<tr>
<th>Write-back/write-allocate (Core)</th>
<th>Write-through/no-write-allocate</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image.png" alt="Diagram" /></td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write-hit</th>
<th>Write-miss</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Write-hit</th>
<th>Write-miss</th>
</tr>
</thead>
</table>

- What to do on a write-hit?
  - **Write-through**: write immediately to memory
  - **Write-back**: defer write to memory until replacement of line

- What to do on a write-miss?
  - **Write-allocate**: load into cache, update line in cache
  - **No-write-allocate**: writes immediately to memory
Example: (Blackboard)

- $z = x + y$, $x$, $y$, $z$ vector of length $n$
- assume they fit jointly in cache + cold cache
- memory traffic $Q(n)$?
- operational intensity $I(n)$?

Locality Optimization: Blocking

- Example: MMM (blackboard)
## The Killer: Two-Power Strided Working Sets

% \( t = 1, 2, 4, 8, \ldots \) a 2-power
% size of working set: \( n/t \)
for (i = 0; i < n; i += t)
  access(x[i])
for (i = 0; i < n; i += t)
  access(x[i])

**Cache:** \( E = 2, B = 4 \) doubles

<table>
<thead>
<tr>
<th>( t = 1 )</th>
<th>( t = 2 )</th>
<th>( t = 4 )</th>
<th>( t = 8 )</th>
<th>( t \geq 4S )</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Cache Diagram" /></td>
<td><img src="image" alt="Cache Diagram" /></td>
<td><img src="image" alt="Cache Diagram" /></td>
<td><img src="image" alt="Cache Diagram" /></td>
<td><img src="image" alt="Cache Diagram" /></td>
</tr>
</tbody>
</table>

Spatial locality: No spatial locality
Temporal locality: Temporal locality:
If \( n/t \leq C/8 \)
If \( n/t \leq 2 \)

---

## The Killer: Where Can It Occur?

- **Accessing two-power size 2D arrays (e.g., images) columnwise**
  - 2d Transforms
  - Stencil computations
  - Correlations
- **Various transform algorithms**
  - Fast Fourier transform
  - Wavelet transforms
  - Filter banks
Summary

- It is important to assess temporal and spatial locality in the code
- Cache structure is determined by three parameters
  - block size
  - number of sets
  - associativity
- You should be able to roughly simulate a computation on paper
- Blocking to improve locality
- Two-power strides are problematic (conflict misses)