How to Write Fast Numerical Code
Spring 2015

Lecture: Memory hierarchy, locality, caches

Instructor: Markus Püschel
TA: Gagndeepr Singh, Daniele Spampinato, Alen Stojanov

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

DFT $2^n$ (single precision) on Pentium 4, 2.53 GHz

[Graph showing performance comparison between DFT algorithms]

- Left alignment
- Attractive font (sans serif, avoid Arial)
  Calibri, Helvetica, Gill Sans MT, ...
- Horizontal y-label
- No y-axis (superfluous)
- Main line possibly emphasized (red, thicker)
- Background/grid inverted for better layering
- No legend; makes decoding easier
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 evolved much slower

Core 2 Duo:
Peak performance: 2 SSE two operand ops/cycles consumes up to 64 Bytes/cycle

Core 2 Duo:
Bandwidth 2 Bytes/cycle

Solution: Caches/Memory hierarchy
Typical Memory Hierarchy

L0: registers
L1: on-chip L1 cache (SRAM)
L2: on-chip L2 cache (SRAM)
L3: main memory (DRAM)
L4: local secondary storage (local disks)
L5: remote secondary storage (tapes, distributed file systems, Web servers)

Small, faster, costlier per byte

Larger, slower, cheaper per byte

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

Abstracted Microarchitecture: Example Core 2 (2008) and Core i7 Sandybridge (2011)

Throughput (tp) is measured in doubles/cycle. For example: 2 double FP = 8 bytes
Latency (lat) is measured in cycles
1 double floating point (FP) = 8 bytes
Rectangles not to scale

Core #1
Core #2
Core 2 Duo:
Core #1
Core #2

ISA
L1 DRAM
L1 cache
L2 cache
L3 cache
L4 cache

ISA
L1 DRAM
L1 cache
L2 cache
L3 cache
L4 cache

L1: 32 KB 8-way 64B CB
L2: 4 MB 16-way 64B CB
L3: 256 KB 128-way 64B CB

L1: 16 FP register
L2: 128 FP register

CISC ops
RISC ops

1 Core

Multiple microarchitecture:

Core i7 Sandybridge: 2×128 KB L2 cache 2×64 KB L1 cache: int 32/32, fp 128/128 RAM: 4 GB
vector (AVX) per 4

Core #1
Core #2
Core #3
Core #4

L1: 32 KB 8-way 64B CB
L2: 4 MB 16-way 64B CB
L3: 256 KB 128-way 64B CB

Lat: millions
tp: ~1/100

Hard disk
depends on platform

0.5 TB

Source: Intel manual (chapter 2)
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?

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

- **Data**:
  - Temporal: `sum` referenced in each iteration
  - Spatial: array `a[]` accessed in stride-1 pattern

- **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

```c
int sum_array_rows(int 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(int 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(int 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?

Operational Intensity Again

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

  ![Operational intensity](image)

  \[ I(n) = \frac{W(n)}{Q(n)} \]

  - #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:** $I(n) \leq O(\log(n))$

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

**MMM:** $I(n) \leq O(n)$

- 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

- Naturally supports *temporal locality*
- *Spatial locality* is supported by transferring data in blocks
  - Core 2: one block = 64 B = 8 doubles

---

**General Cache Mechanics**

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

Request: 12

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)
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 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
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
- $E =$ associativity, $E=1$: direct mapped

$S = 2^s$ sets

$B = 2^b$ bytes per cache block (the data)

Cache size: $S \times E \times B$ data bytes

<table>
<thead>
<tr>
<th>Location</th>
<th>Tag</th>
<th>Set Index</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>t bits</td>
<td>s bits</td>
<td>b bits</td>
</tr>
</tbody>
</table>

Address of word:

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

Cache Read

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

$S = 2^s$ sets

$E = 2^e$ lines per set

$E =$ associativity, $E=1$: direct mapped

$B = 2^b$ bytes per cache block (the data)
Small Example, Part 1

\[ x[0] \]

\[ \downarrow \]

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

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

\% Matlab style code

```
for j = 0:1
  for i = 0:7
    access(x[i])
  end
end
```

**Access pattern:** 0123456701234567

**Hit/Miss:** MHMHMHMHMHMHMHMH

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

Small Example, Part 2

\[ x[0] \]

\[ \downarrow \]

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

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

\% Matlab style code

```
for j = 0:1
  for i = 0:2:7
    access(x[i])
  end
  for i = 1:2:7
    access(x[i])
  end
end
```

**Access pattern:** 0246135702461357

**Hit/Miss:** MMMMMMMMMMMMMMM

**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: MHHHHHHHHHHHHHHHH

**Result:** 4 misses, 12 hits (is optimal, why?)

**Spatial locality:** yes

**Temporal locality:** yes

### 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

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

### Write-back/write-allocate (Core) vs. Write-through/no-write-allocate

<table>
<thead>
<tr>
<th>Write-back/write-allocate (Core)</th>
<th>Write-through/no-write-allocate</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Write-back/write-allocate Diagram" /></td>
<td><img src="image2.png" alt="Write-through/no-write-allocate Diagram" /></td>
</tr>
</tbody>
</table>

**Write-hit**

1. load
2. update

**Write-miss**

1. update
2. update

**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

```plaintext
% t = 1,2,4,8,... 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 ≥ 4S:</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Cache Access Diagram" /></td>
<td><img src="image" alt="Cache Access Diagram" /></td>
<td><img src="image" alt="Cache Access Diagram" /></td>
<td><img src="image" alt="Cache Access Diagram" /></td>
<td><img src="image" alt="Cache Access Diagram" /></td>
</tr>
<tr>
<td>Spatial locality: if ( n/t \leq C )</td>
<td>Some spatial locality: if ( n/t \leq C/2 )</td>
<td>No spatial locality: if ( n/t \leq C/4 )</td>
<td>No spatial locality: if ( n/t \leq C/8 )</td>
<td>No spatial locality: if ( n/t \leq 2 )</td>
</tr>
</tbody>
</table>
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
- You should be able to roughly simulate a computation on paper
- Blocking to improve locality
- Two-power strides are problematic (conflict misses)