Sequentian Code Optimization

CENG478 – Introduction to Parallel Computing
Spring 2019

Instructor:
Murat Manguoglu

Adapted from slides of CENG331 - Computer Organization, textbook: http://csapp.cs.cmu.edu/
Today

- Overview
- Generally Useful Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploiting Instruction-Level Parallelism
- Cache
- Dealing with Conditionals
Performance Realities

■ There’s more to performance than asymptotic complexity

■ Constant factors matter too!
  ▪ Easily see 10:1 performance range depending on how code is written
  ▪ Must optimize at multiple levels:
    ▪ algorithm, data representations, procedures, and loops

■ Must understand system to optimize performance
  ▪ How programs are compiled and executed
  ▪ How modern processors + memory systems operate
  ▪ How to measure program performance and identify bottlenecks
  ▪ How to improve performance without destroying code modularity and generality
Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies

- Don’t (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter

- Have difficulty overcoming “optimization blockers”
  - potential memory aliasing
  - potential procedure side-effects
Limitations of Optimizing Compilers

- **Operate under fundamental constraint**
  - Must not cause any change in program behavior
    - Except, possibly when program making use of nonstandard language features
    - Often prevents it from making optimizations that would only affect behavior under pathological conditions.

- **Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles**
  - e.g., Data ranges may be more limited than variable types suggest

- **Most analysis is performed only within procedures**
  - Whole-program analysis is too expensive in most cases
  - Newer versions of GCC do interprocedural analysis within individual files
    - But, not between code in different files

- **Most analysis is based only on static information**
  - Compiler has difficulty anticipating run-time inputs

- **When in doubt, the compiler must be conservative**
Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler
  
- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```c
void set_row(double *a, double *b, long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```c
long j;
int ni = n*i;
for (j = 0; j < n; j++)
a[ni+j] = b[j];
```
Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  \[ 16 \times x \quad \rightarrow \quad x \ll 4 \]
  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

```c
for (i = 0; i < n; i++) {
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
}
```

```c
int ni = 0;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
    ni += n;
}
```
Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with –O1

```c
/* Sum neighbors of i,j */
up = val[(i-1)*n + j ];
down = val[(i+1)*n + j ];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i-1)*n, (i+1)*n

```c
long inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplication: i*n
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
    {
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
    }
}
Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance
Calling Strlen

/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}

- **Strlen performance**
  - Only way to determine length of string is to scan its entire length, looking for null character.

- **Overall performance, string of length N**
  - N calls to strlen
  - Require times N, N-1, N-2, ..., 1
  - Overall $O(N^2)$ performance
Improving Performance

void lower(char *s)
{
    size_t i;
    size_t len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2
Optimization Blocker: Procedure Calls

- Why couldn’t compiler move `strlen` out of inner loop?
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
    - Procedure lower could interact with `strlen`

- Warning:
  - Compiler treats procedure call as a black box
  - Weak optimizations near them

- Remedies:
  - Use of inline functions
    - GCC does this with `-O1`
      - Within single file
  - Do your own code motion

```c
size_t lencnt = 0;
size_t strlen(const char *s) {
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```
Memory Matters

- Code updates \( b[i] \) on every iteration
- Why couldn’t compiler optimize this away?
Memory Aliasing

- Code updates $b[i]$ on every iteration
- Must consider possibility that these updates will affect program behavior

```c
/* Sum rows is of n X n matrix a 
   and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}

double A[9] =
{ 0, 1, 2,
  4, 8, 16},
32, 64, 128};
sum_rows1(A, B, 3);
```

Value of B:

- init: [4, 8, 16]
- i = 0: [3, 8, 16]
- i = 1: [3, 22, 16]
- i = 2: [3, 22, 224]
Removing Aliasing

- No need to store intermediate results
Optimization Blocker: Memory Aliasing

- **Aliasing**
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing
Exploiting Instruction-Level Parallelism

- Need general understanding of modern processor design
  - Hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can yield dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic
Benchmark Example: Data Type for Vectors

/* data structure for vectors */
typedef struct{
    size_t len;
    data_t *data;
} vec;

/* retrieve vector element and store at val */
int get_vec_element(*vec v, size_t idx, data_t *val)
{
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}

- Data Types
  - Use different declarations for data_t
    - int
    - long
    - float
    - double
Benchmark Computation

```c
void combine1(vec_ptr v, data_t *dest) {
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

**Data Types**
- Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`

**Operations**
- Use different definitions of `OP` and `IDENT`
  - `+ / 0`
  - `* / 1`
Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- In our case: CPE = cycles per OP
- T = CPE*n + Overhead
  - CPE is slope of line

![Graph with two lines representing psum1 and psum2 with slopes 9.0 and 6.0 respectively.](image)
Benchmark Performance

```c
void combine1(vec_ptr v, data_t *dest) {
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine1 unoptimized</td>
<td>22.68</td>
<td>20.02</td>
<td>19.98</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
<td>10.12</td>
<td>10.17</td>
</tr>
</tbody>
</table>

Compute sum or product of vector elements
Basic Optimizations

- Move `vec_length` out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

```c
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```
Effect of Basic Optimizations

Eliminates sources of overhead in loop

```c
void combine4(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
<td>10.12</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
</tbody>
</table>
Modern CPU Design

Instruction Control

Fetch Control

Instruction Cache

Instruction Decode

Retirement Unit

Register File

Prediction OK?

Register Updates

Data Cache

Retirement Unit

Fetch Control

Instruction Cache

Instruction Decode

Arith

Arith

Arith

Arith

Load

Store

Branch

Functional Units

Operation Results

Addr.

Data

Addr.

Data

Data Cache

Address

Instructions

Operations
**Superscalar Processor**

- **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

- **Benefit:** without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have.

- Most modern CPUs are superscalar.
- Intel: since Pentium (1993)
### Pipelined Functional Units

**Function Definition:**

```c
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage $i$ can start on new computation once values passed to $i+1$
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles
Haswell CPU

- 8 Total Functional Units

Multiple instructions can execute in parallel
2 load, with address computation
1 store, with address computation
4 integer
2 FP multiply
1 FP add
1 FP divide

Some instructions take > 1 cycle, but can be pipelined

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>3-30</td>
<td>3-30</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>3-15</td>
<td>3-15</td>
</tr>
</tbody>
</table>
x86-64 Compilation of Combine4

- Inner Loop (Case: Integer Multiply)

```assembly
.L519:
  imull (%rax,%rdx,4), %ecx  # t = t * d[i]
  addq $1, %rdx             # i++
  cmpq %rdx, %rbp           # Compare length:i
  jg .L519                 # If >, goto Loop
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
</tbody>
</table>
Combine4 = Serial Computation (OP = *)

- Computation (length=8)
  \[((((((((1 \times d[0]) \times d[1]) \times d[2]) \times d[3]) \times d[4]) \times d[5]) \times d[6]) \times d[7])\]

- Sequential dependence
  - Performance: determined by latency of OP
Loop Unrolling (2x1)

void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}

- Perform 2x more useful work per iteration
Effect of Loop Unrolling

- Helps integer add
  - Achieves latency bound

- Others don’t improve. **Why?**
  - Still sequential dependency

\[
x = (x \, \text{OP} \, d[i]) \, \text{OP} \, d[i+1];
\]
Loop Unrolling with Reassociation (2x1a)

```c
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Can this change the result of the computation?
- Yes, for FP. *Why?*
**Effect of Reassociation**

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1a</td>
<td>1.01</td>
<td>1.51</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential dependency
  - Why is that? (next slide)

2 func. units for FP *
2 func. units for load
4 func. units for int +
2 func. units for load
Reassociated Computation

What changed:
- Ops in the next iteration can be started early (no dependency)

Overall Performance
- N elements, D cycles latency/op
- \((N/2+1)\)*D cycles:
  \[ CPE = \frac{D}{2} \]
Loop Unrolling with Separate Accumulators (2x2)

```c
void unroll2a_combine(vec_ptr v, data_t *dest) {
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- Different form of reassociation
## Effect of Separate Accumulators

<table>
<thead>
<tr>
<th>Method</th>
<th>Operation</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1a</td>
<td>1.01</td>
<td>1.51</td>
<td>1.51</td>
</tr>
<tr>
<td>Unroll 2x2</td>
<td>0.81</td>
<td>1.51</td>
<td>1.51</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- **Int +** makes use of two load units
  
  \[
  x_0 = x_0 \ \text{OP} \ d[i]; \\
  x_1 = x_1 \ \text{OP} \ d[i+1]; \\
  \]

- 2x speedup (over unroll2) for Int *, FP +, FP *
Separate Accumulators

\[ x_0 = x_0 \text{ OP } d[i]; \]
\[ x_1 = x_1 \text{ OP } d[i+1]; \]

What changed:
- Two independent “streams” of operations

Overall Performance
- N elements, D cycles latency/op
- Should be \((N/2+1)D\) cycles:
  \[ \text{CPE} = \frac{D}{2} \]
- CPE matches prediction!

What Now?
Unrolling & Accumulating

- **Idea**
  - Can unroll to any degree L
  - Can accumulate K results in parallel
  - L must be multiple of K

- **Limitations**
  - Diminishing returns
    - Cannot go beyond throughput limitations of execution units
  - Large overhead for short lengths
    - Finish off iterations sequentially
## Unrolling & Accumulating: Double *

### Case
- Intel Haswell
- Double FP Multiplication
- Latency bound: 5.00. Throughput bound: 0.50

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K 1 2 3 4 6 8 10 12</td>
</tr>
<tr>
<td>1</td>
<td>5.01 5.01 5.01 5.01 5.01 5.01 5.01</td>
</tr>
<tr>
<td>2</td>
<td>2.51 2.51 2.51 2.51</td>
</tr>
<tr>
<td>3</td>
<td>1.67</td>
</tr>
<tr>
<td>4</td>
<td>1.25 1.26</td>
</tr>
<tr>
<td>6</td>
<td>0.84 0.88</td>
</tr>
<tr>
<td>8</td>
<td>0.63</td>
</tr>
<tr>
<td>10</td>
<td>0.51</td>
</tr>
<tr>
<td>12</td>
<td>0.52</td>
</tr>
</tbody>
</table>
## Unrolling & Accumulating: Int +

### Case
- Inte Haswell
- Integer addition
- Latency bound: 1.00. Throughput bound: 1.00

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1.27</td>
</tr>
<tr>
<td>2</td>
<td>0.81</td>
</tr>
<tr>
<td>3</td>
<td>0.74</td>
</tr>
<tr>
<td>4</td>
<td>0.69</td>
</tr>
<tr>
<td>6</td>
<td>0.56</td>
</tr>
<tr>
<td>8</td>
<td>0.54</td>
</tr>
<tr>
<td>10</td>
<td>0.54</td>
</tr>
<tr>
<td>12</td>
<td>0.56</td>
</tr>
</tbody>
</table>
## Achievable Performance

- Limited only by throughput of functional units
- Up to 42X improvement over original, unoptimized code

<table>
<thead>
<tr>
<th>Method</th>
<th>Operation</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Best</td>
<td></td>
<td>0.54</td>
<td>1.01</td>
<td>1.01</td>
<td>0.52</td>
</tr>
<tr>
<td>Latency Bound</td>
<td></td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td></td>
<td>0.50</td>
<td>1.00</td>
<td>1.00</td>
<td>0.50</td>
</tr>
</tbody>
</table>
Programming with AVX2

YMM Registers

- 16 total, each 32 bytes
- 32 single-byte integers
- 16 16-bit integers
- 8 32-bit integers
- 8 single-precision floats
- 4 double-precision floats
- 1 single-precision float
- 1 double-precision float
SIMD Operations

- **SIMD Operations: Single Precision**
  
  \[ \text{vaddsd} \%ymm0, \%ymm1, \%ymm1 \]

- **SIMD Operations: Double Precision**
  
  \[ \text{vaddpd} \%ymm0, \%ymm1, \%ymm1 \]
# Using Vector Instructions

## Make use of AVX Instructions
- Parallel operations on multiple data elements
- See Web Aside OPT:SIMD on CS:APP web page

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar Best</td>
<td>0.54</td>
<td>1.01</td>
</tr>
<tr>
<td>Vector Best</td>
<td>0.06</td>
<td>0.24</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>0.50</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>Vec Throughput Bound</td>
<td>0.06</td>
<td>0.12</td>
</tr>
</tbody>
</table>
Getting High Performance

- Good compiler and flags
- Don’t make any obvious mistakes
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers: procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)
- Tune code for machine
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly (Covered later in course)
Some fundamental and enduring properties of hardware and software:

- Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
- The gap between CPU and main memory speed is widening.
- Well-written programs tend to exhibit good locality.

These fundamental properties complement each other beautifully.

They suggest an approach for organizing memory and storage systems known as a memory hierarchy.
Example Memory Hierarchy

L0: Regs
- CPU registers hold words retrieved from the L1 cache.

L1: L1 cache (SRAM)
- L1 cache holds cache lines retrieved from the L1 cache.

L2: L2 cache (SRAM)
- L2 cache holds cache lines retrieved from the L2 cache.

L3: L3 cache (SRAM)
- L3 cache holds cache lines retrieved from L3 cache.

L4: Main memory (DRAM)
- Main memory holds disk blocks retrieved from local disks.

L5: Local secondary storage (local disks)
- Local disks hold files retrieved from disks on remote servers.

L6: Remote secondary storage (e.g., Web servers)
- Larger, slower, and cheaper (per byte) storage devices
# Examples of Caching in the Mem. Hierarchy

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>What is Cached?</th>
<th>Where is it Cached?</th>
<th>Latency (cycles)</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4-8 bytes words</td>
<td>CPU core</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-Chip TLB</td>
<td>0</td>
<td>Hardware MMU</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-byte blocks</td>
<td>On-Chip L1</td>
<td>4</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-byte blocks</td>
<td>On-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4-KB pages</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware + OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Disk cache</td>
<td>Disk sectors</td>
<td>Disk controller</td>
<td>100,000</td>
<td>Disk firmware</td>
</tr>
<tr>
<td>Network buffer cache</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>NFS client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web proxy server</td>
</tr>
</tbody>
</table>
Management of Memory Hierarchy

- Small/fast storage, e.g., registers
  - Address usually specified in instruction
  - Generally implemented directly as a register file
    - but hardware might do things behind software’s back, e.g., stack management, register renaming

- Larger/slower storage, e.g., main memory
  - Address usually computed from values in register
  - Generally implemented as a hardware-managed cache hierarchy
    - hardware decides what is kept in fast memory
    - but software may provide “hints”, e.g., don’t cache or prefetch
Cache Memories

- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- **CPU looks first for data in cache, then in main memory**
- **Typical system structure:**

![Diagram of a computer system showing the CPU chip, register file, ALU, bus interface, I/O bridge, memory bus, and main memory.](image-url)
Itanium-2 On-Chip Caches
(Intel/HP, 2002)

Level 1: 16KB, 4-way s.a., 64B line, quad-port (2 load+2 store), single cycle latency

Level 2: 256KB, 4-way s.a., 128B line, quad-port (4 load or 4 store), five cycle latency

Level 3: 3MB, 12-way s.a., 128B line, single 32B port, twelve cycle latency
Power 7 On-Chip Caches [IBM 2009]

- 32KB L1 I$/core
- 32KB L1 D$/core
- 3-cycle latency

- 256KB Unified L2$/core
- 8-cycle latency

- 32MB Unified Shared L3$
- Embedded DRAM (eDRAM)
- 25-cycle latency to local slice
IBM z196 Mainframe Caches 2010

- 96 cores (4 cores/chip, 24 chips/system)
  - Out-of-order, 3-way superscalar @ 5.2GHz
- L1: 64KB I-$/core + 128KB D-$/core
- L2: 1.5MB private/core (144MB total)
- L3: 24MB shared/chip (eDRAM) (576MB total)
- L4: 768MB shared/system (eDRAM)
Intel Core i7 Cache Hierarchy

Processor package

Core 0

- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

Core 3

- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

... 

L3 unified cache (shared by all cores)

- L1 i-cache and d-cache: 32 KB, 8-way, Access: 4 cycles
- L2 unified cache: 256 KB, 8-way, Access: 10 cycles
- L3 unified cache: 8 MB, 16-way, Access: 40-75 cycles

Block size: 64 bytes for all caches.

Main memory
Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions

- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories
The Memory Mountain

- **Read throughput** *(read bandwidth)*
  - Number of bytes read from memory per second (MB/s)

- **Memory mountain**: Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.
long data[MAXELEMS]; /* Global array to traverse */

/* test - Iterate over first "elems" elements of 
   * array “data” with stride of "stride", using 
   * using 4x4 loop unrolling. 
   */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}

Call test() with many combinations of elems and stride.
For each elems and stride:
1. Call test() once to warm up the caches.
2. Call test() again and measure the read throughput (MB/s)
The Memory Mountain

Slopes of spatial locality

Ridges of temporal locality

Aggressive prefetching

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size
Matrix Multiplication

**ijk (& jik):**
- 2 loads, 0 stores
- misses/iter = 1.25

**kij (& ikj):**
- 2 loads, 1 store
- misses/iter = 0.5

**jki (& kji):**
- 2 loads, 1 store
- misses/iter = 2.0

```c
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}

for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}

for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```
Core i7 Matrix Multiply Performance

Cycles per inner loop iteration vs. Array size (n)

- jki / kji
- ijk / jik
- kij / ikj
Example: Matrix Multiplication

c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
Cache Miss Analysis

Assume:
- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than $n$)

First iteration:
- $n/8 + n = 9n/8$ misses
- Afterwards in cache: (schematic)
Blocked Matrix Multiplication

```c
double *c = calloc(sizeof(double), n*n);

void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

`matmult/bmm.c`
Cache Summary

- Cache memories can have significant performance impact

- You can write your programs to exploit this!
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it’s read from memory.