Performance Improvement Revisited

Jennifer Rexford

Goals of this Lecture

- Improve program performance by exploiting knowledge of the underlying system
  - Compiler capabilities
  - Hardware architecture
  - Program execution
- And thereby:
  - Help you to write efficient programs
  - Review material from the second half of the course

When to Optimize Performance

Improving Program Performance

- Most programs are already "fast enough"
  - No need to optimize performance at all
  - Save your time, and keep the program simple/readable
- Most parts of a program are already "fast enough"
  - Usually only a small part makes the program run slowly
  - Optimize *only* this portion of the program, as needed
- Steps to improve execution (time) efficiency
  - Do timing studies (e.g., gprof)
  - Identify hot spots
  - **Optimize that part of the program**
  - Repeat as needed

Two Main Outputs of Gprof

- Call graph profile: detailed information per function
  - Which functions called it, and how much time was consumed?
  - Which functions it calls, how many times, and for how long?
  - We won't look at this output in any detail…
- Flat profile: one line per function
  - name: name of the function
  - %time: percentage of time spent executing this function
  - cumulative seconds: [skipping, as this isn't all that useful]
  - self seconds: time spent executing this function
  - calls: number of times function was called (excluding recursive)
  - self ms/call: average time per execution (excluding descendents)
  - total ms/call: average time per execution (including descendents)

Call Graph Output

```
                                  called/total       parents
index  %time    self descendents  called+self    name           index
                                  called/total       children

                                                     <spontaneous>
[1]     59.7   12.97        0.00                 internal_mcount [1]
                0.00        0.00       1/3           atexit [35]
-----------------------------------------------
                                                     <spontaneous>
[2]     40.3    0.00        8.75                 _start [2]
                0.00        8.75       1/1           main [3]
                0.00        0.00       2/3           atexit [35]
-----------------------------------------------
                0.00        8.75       1/1           _start [2]
[3]     40.3    0.00        8.75       1         main [3]
                0.00        8.32       1/1           getBestMove [4]
                0.00        0.43       1/1           clock [20]
                0.00        0.00       1/747130      GameState_expandMove [6]
                0.00        0.00       1/1           exit [33]
                0.00        0.00       1/1           Move_read [36]
                0.00        0.00       1/1           GameState_new [37]
                0.00        0.00       6/747135      GameState_getStatus [31]
                0.00        0.00       1/747130      GameState_applyDeltas [25]
                0.00        0.00       1/1           GameState_write [44]
                0.00        0.00       1/1           sscanf [54]
                0.00        0.00       1/1698871     GameState_getPlayer [30]
                0.00        0.00       1/1           _fprintf [58]
                0.00        0.00       1/1           Move_write [59]
                0.00        0.00       3/3           GameState_playerToStr [63]
                0.00        0.00       1/2           strcmp [66]
                0.00        0.00       1/1           GameState_playerFromStr [68]
                0.00        0.00       1/1           Move_isValid [69]
                0.00        0.00       1/1           GameState_getSearchDepth [67]
-----------------------------------------------
                0.00        8.32       1/1           main [3]
[4]     38.3    0.00        8.32       1         getBestMove [4]
                0.27        8.05       6/6           minimax [5]
                0.00        0.00       6/747130      GameState_expandMove [6]
                0.00        0.00      35/4755325     Delta_free [10]
                0.00        0.00       1/204617      GameState_genMoves [17]
                0.00        0.00       5/945027      Move_free [23]
                0.00        0.00       6/747130      GameState_applyDeltas [25]
                0.00        0.00       6/747129      GameState_unApplyDeltas [27]
                0.00        0.00       2/1698871     GameState_getPlayer [30]
-----------------------------------------------
                             747123                  minimax [5]
                0.27        8.05       6/6           getBestMove [4]
[5]     38.3    0.27        8.05       6+747123  minimax [5]
                0.63        3.56  747123/747130      GameState_expandMove [6]
                0.25        1.82 4755290/4755325     Delta_free [10]
                0.07        0.69  204616/204617      GameState_genMoves [17]
                0.02        0.36  945022/945027      Move_free [23]
                0.23        0.00  747123/747130      GameState_applyDeltas [25]
                0.22        0.00  747123/747129      GameState_unApplyDeltas [27]
                0.10        0.00 1698868/1698871     GameState_getPlayer [30]
                0.09        0.00  747129/747135      GameState_getStatus [31]
                0.01        0.00  542509/542509      GameState_getValue [32]
                             747123                  minimax [5]
-----------------------------------------------
                0.00        0.00       1/747130      main [3]
                0.00        0.00       6/747130      getBestMove [4]
                0.63        3.56  747123/747130      minimax [5]
[6]     19.3    0.63        3.56  747130         GameState_expandMove [6]
                0.47        2.99 4755331/5700361     calloc [7]
                0.11        0.00 2360787/2360787     .rem [28]
-----------------------------------------------
                0.00        0.00       1/5700361     Move_read [36]
                0.00        0.00       1/5700361     GameState_new [37]
                0.09        0.59  945028/5700361     GameState_genMoves [17]
                0.47        2.99 4755331/5700361     GameState_expandMove [6]
[7]     19.1    0.56        3.58 5700361         calloc [7]
                0.32        2.09 5700361/5700362     malloc [8]
                0.64        0.00 5700361/5700361     .umul [18]
                0.43        0.00 5700361/5700361     _memset [22]
                0.10        0.00 5700361/5700363     .udiv [29]
-----------------------------------------------
                0.00        0.00       1/5700362     _findbuf [41]
                0.32        2.09 5700361/5700362     calloc [7]
[8]     11.1    0.32        2.09 5700362         malloc [8]
                0.62        0.62 5700362/5700362     _malloc_unlocked <cycle 1> [13]
                0.23        0.20 5700362/11400732    _mutex_unlock [14]
                0.22        0.20 5700362/11400732    mutex_lock [15]
```

Complex format at the beginning… let's skip for now.

Flat Profile

```
  %  cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call name
 57.1     12.97    12.97                             internal_mcount [1]
  4.8     14.05     1.08  5700352     0.00     0.00  _free_unlocked [12]
  4.4     15.04     0.99                             _mcount (693)
  3.5     15.84     0.80 22801464     0.00     0.00  _return_zero [16]
  2.8     16.48     0.64  5700361     0.00     0.00  .umul [18]
  2.8     17.11     0.63   747130     0.00     0.01  GameState_expandMove [6]
  2.5     17.67     0.56  5700361     0.00     0.00  calloc [7]
  2.1     18.14     0.47 11400732     0.00     0.00  _mutex_unlock [14]
  1.9     18.58     0.44 11400732     0.00     0.00  mutex_lock [15]
  1.9     19.01     0.43  5700361     0.00     0.00  _memset [22]
  1.9     19.44     0.43        1   430.00   430.00  .div [21]
  1.8     19.85     0.41  5157853     0.00     0.00  cleanfree [19]
  1.4     20.17     0.32  5700366     0.00     0.00  _malloc_unlocked  [13]
  1.4     20.49     0.32  5700362     0.00     0.00  malloc [8]
  1.3     20.79     0.30  5157847     0.00     0.00  _smalloc          [24]
  1.2     21.06     0.27        6    45.00  1386.66  minimax [5]
  1.1     21.31     0.25  4755325     0.00     0.00  Delta_free [10]
  1.0     21.54     0.23  5700352     0.00     0.00  free [9]
  1.0     21.77     0.23   747130     0.00     0.00  GameState_applyDeltas [25]
  1.0     21.99     0.22  5157845     0.00     0.00  realfree [26]
  1.0     22.21     0.22   747129     0.00     0.00  GameState_unApplyDeltas [27]
  0.5     22.32     0.11  2360787     0.00     0.00  .rem [28]
  0.4     22.42     0.10  5700363     0.00     0.00  .udiv [29]
  0.4     22.52     0.10  1698871     0.00     0.00  GameState_getPlayer [30]
  0.4     22.61     0.09   747135     0.00     0.00  GameState_getStatus [31]
  0.3     22.68     0.07   204617     0.00     0.00  GameState_genMoves [17]
  0.1     22.70     0.02   945027     0.00     0.00  Move_free [23]
  0.0     22.71     0.01   542509     0.00     0.00  GameState_getValue [32]
  0.0     22.71     0.00      104     0.00     0.00  _ferror_unlocked [357]
  0.0     22.71     0.00       64     0.00     0.00  _realbufend [358]
  0.0     22.71     0.00       54     0.00     0.00  nvmatch [60]
  0.0     22.71     0.00       52     0.00     0.00  _doprnt [42]
  0.0     22.71     0.00       51     0.00     0.00  memchr [61]
  0.0     22.71     0.00       51     0.00     0.00  printf [43]
  0.0     22.71     0.00       13     0.00     0.00  _write [359]
  0.0     22.71     0.00       10     0.00     0.00  _xflsbuf [360]
  0.0     22.71     0.00        7     0.00     0.00  _memcpy [361]
  0.0     22.71     0.00        4     0.00     0.00  .mul [62]
  0.0     22.71     0.00        4     0.00     0.00  ___errno [362]
  0.0     22.71     0.00        4     0.00     0.00  _fflush_u [363]
  0.0     22.71     0.00        3     0.00     0.00  GameState_playerToStr [63]
  0.0     22.71     0.00        3     0.00     0.00  _findbuf [41]
```

Second part of profile looks like this; it's the simple (i.e.,useful) part; corresponds to the "prof" tool

Overhead of Profiling

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.1	**12.97**	**12.97**				**internal_mcount**
4.8	14.05	1.08	5700352	0.00	0.00	_free_unlocked
4.4	**15.04**	**0.99**				**_mcount (693)**
3.5	15.84	0.80	22801464	0.00	0.00	_return_zero
2.8	16.48	0.64	5700361	0.00	0.00	.umul [18]
2.8	17.11	0.63	747130	0.00	0.01	GameState_expa
2.5	17.67	0.56	5700361	0.00	0.00	calloc [7]
2.1	18.14	0.47	11400732	0.00	0.00	_mutex_unlock
1.9	18.58	0.44	11400732	0.00	0.00	mutex_lock
1.9	19.01	0.43	5700361	0.00	0.00	_memset [22]
1.9	19.44	0.43	1	430.00	430.00	.div [21]
1.8	19.85	0.41	5157853	0.00	0.00	cleanfree [19]
1.4	20.17	0.32	5700366	0.00	0.00	_malloc_unlo
1.4	20.49	0.32	5700362	0.00	0.00	malloc [8]
1.3	20.79	0.30	5157847	0.00	0.00	_smalloc
1.2	21.06	0.27	6	45.00	1386.66	minimax [5]
1.1	21.31	0.25	4755325	0.00	0.00	Delta_free [10]
1.0	21.54	0.23	5700352	0.00	0.00	free [9]
1.0	21.77	0.23	747130	0.00	0.00	GameState_appl
1.0	21.99	0.22	5157845	0.00	0.00	realfree [26]
1.0	22.21	0.22	747129	0.00	0.00	GameState_unAp
0.5	22.32	0.11	2360787	0.00	0.00	.rem [28]
0.4	22.42	0.10	5700363	0.00	0.00	.udiv [29]
0.4	22.52	0.10	1698871	0.00	0.00	GameState_getPl
0.4	22.61	0.09	747135	0.00	0.00	GameState_getSt

Malloc/calloc/free/...

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.1	12.97	12.97				internal_mcount [1]
4.8	**14.05**	**1.08**	**5700352**	**0.00**	**0.00**	**_free_unlocked [12]**
4.4	15.04	0.99				_mcount (693)
3.5	15.84	0.80	22801464	0.00	0.00	_return_zero [16]
2.8	16.48	0.64	5700361	0.00	0.00	.umul [18]
2.8	17.11	0.63	747130	0.00	0.01	GameState_expandMove
2.5	**17.67**	**0.56**	**5700361**	**0.00**	**0.00**	**calloc [7]**
2.1	**18.14**	**0.47**	**11400732**	**0.00**	**0.00**	**_mutex_unlock [14]**
1.9	**18.58**	**0.44**	**11400732**	**0.00**	**0.00**	**mutex_lock [15]**
1.9	**19.01**	**0.43**	**5700361**	**0.00**	**0.00**	**_memset [22]**
1.9	19.44	0.43	1	430.00	430.00	.div [21]
1.8	**19.85**	**0.41**	**5157853**	**0.00**	**0.00**	**cleanfree [19]**
1.4	**20.17**	**0.32**	**5700366**	**0.00**	**0.00**	**_malloc_unlocked [13]**
1.4	**20.49**	**0.32**	**5700362**	**0.00**	**0.00**	**malloc [8]**
1.3	**20.79**	**0.30**	**5157847**	**0.00**	**0.00**	**_smalloc [24]**
1.2	21.06	0.27	6	45.00	1386.66	minimax [5]
1.1	21.31	0.25	4755325	0.00	0.00	Delta_free [10]
1.0	**21.54**	**0.23**	**5700352**	**0.00**	**0.00**	**free [9]**
1.0	21.77	0.23	747130	0.00	0.00	GameState_applyDeltas
1.0	**21.99**	**0.22**	**5157845**	**0.00**	**0.00**	**realfree [26]**
1.0	22.21	0.22	747129	0.00	0.00	GameState_unApplyDeltas
0.5	22.32	0.11	2360787	0.00	0.00	.rem [28]
0.4	22.42	0.10	5700363	0.00	0.00	.udiv [29]
0.4	22.52	0.10	1698871	0.00	0.00	GameState_getPlayer
0.4	22.61	0.09	747135	0.00	0.00	GameState_getStatus
0.3	22.68	0.07	204617	0.00	0.00	GameState_genMoves [17]

expandMove

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.1	12.97	12.97				internal_mcount [1]
4.8	14.05	1.08	5700352	0.00	0.00	_free_unlocked [12]
4.4	15.04	0.99				_mcount (693)
3.5	15.84	0.80	22801464	0.00	0.00	_return_zero [16]
2.8	16.48	0.64	5700361	0.00	0.00	.umul [18]
2.8	**17.11**	**0.63**	**747130**	**0.00**	**0.01**	**GameState_expandMove**
2.5	17.67	0.56	5700361	0.00	0.00	calloc [7]
2.1	18.14	0.47	11400732	0.00	0.00	_mutex_unlock [14]
1.9	18.58	0.44	11400732	0.00	0.00	mutex_lock [15]
1.9	19.01	0.43	5700361	0.00	0.00	_memset [22]
1.9	19.44	0.43	1	430.00	430.00	.div [21]
1.8	19.85	0.41	5157853	0.00	0.00	cleanfree [19]
1.4	20.17	0.32	5700366	0.00	0.00	_malloc_unlocked [13]
1.4	20.49	0.32	5700362	0.00	0.00	malloc [8]
1.3	20.79	0.30	5157847	0.00	0.00	_smalloc [24]
1.2	21.06	0.27	6	45.00	1386.66	minimax [5]
1.1	21.31	0.25	4755325	0.00	0.00	Delta_free [10]
1.0	21.54	0.23	5700352	0.00	0.00	free [9]
1.0	21.77	0.23	747130	0.00	0.00	GameState_applyDeltas
1.0	21.99	0.22	5157845	0.00	0.00	realfree [26]

May be worthwhile to optimize this routine

Don't Even ***Think*** of Optimizing These

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.1	12.97	12.97				internal_mcount [1]
4.8	14.05	1.08	5700352	0.00	0.00	_free_unlocked [12]
4.4	15.04	0.99				_mcount (693)
3.5	15.84	0.80	22801464	0.00	0.00	__return_zero [16]
2.8	16.48	0.64	5700361	0.00	0.00	.umul [18]
2.8	17.11	0.63	747130	0.00	0.01	GameState_expandMove [6]
2.5	17.67	0.56	5700361	0.00	0.00	calloc [7]
2.1	18.14	0.47	11400732	0.00	0.00	_mutex_unlock [14]
1.9	18.58	0.44	11400732	0.00	0.00	mutex_lock [15]
1.9	19.01	0.43	5700361	0.00	0.00	_memset [22]
1.9	19.44	0.43	1	430.00	430.00	.div [21]
1.8	19.85	0.41	5157853	0.00	0.00	cleanfree [19]
1.4	20.17	0.32	5700366	0.00	0.00	_malloc_unlocked <cycle 1> [13]
1.4	20.49	0.32	5700362	0.00	0.00	malloc [8]
1.3	20.79	0.30	5157847	0.00	0.00	_smalloc <cycle 1> [24]
1.2	21.06	0.27	6	45.00	1386.66	minimax [5]
1.1	21.31	0.25	4755325	0.00	0.00	Delta_free [10]
1.0	21.54	0.23	5700352	0.00	0.00	free [9]
1.0	21.77	0.23	747130	0.00	0.00	GameState_applyDeltas [25]
1.0	21.99	0.22	5157845	0.00	0.00	realfree [26]
1.0	22.21	0.22	747129	0.00	0.00	GameState_unApplyDeltas [27]
0.5	22.32	0.11	2360787	0.00	0.00	.rem [28]
0.4	22.42	0.10	5700363	0.00	0.00	.udiv [29]
0.4	22.52	0.10	1698871	0.00	0.00	GameState_getPlayer [30]
0.4	22.61	0.09	747135	0.00	0.00	GameState_getStatus [31]
0.3	22.68	0.07	204617	0.00	0.00	GameState_genMoves [17]
0.1	22.70	0.02	945027	0.00	0.00	Move_free [23]
0.0	22.71	0.01	542509	0.00	0.00	GameState_getValue [32]
0.0	22.71	0.00	104	0.00	0.00	_ferror_unlocked [357]
0.0	22.71	0.00	4	0.00	0.00	_thr_main [367]
0.0	22.71	0.00	3	0.00	0.00	GameState_playerToStr [63]
0.0	22.71	0.00	2	0.00	0.00	strcmp [66]
0.0	22.71	0.00	1	0.00	0.00	GameState_getSearchDepth [67]
0.0	22.71	0.00	1	0.00	0.00	GameState_new [37]
0.0	22.71	0.00	1	0.00	0.00	GameState_playerFromStr [68]
0.0	22.71	0.00	1	0.00	0.00	GameState_write [44]
0.0	22.71	0.00	1	0.00	0.00	Move_isValid [69]
0.0	22.71	0.00	1	0.00	0.00	Move_read [36]
0.0	22.71	0.00	1	0.00	0.00	Move_write [59]
0.0	22.71	0.00	1	0.00	0.00	check_nlspath_env [46]
0.0	22.71	0.00	1	0.00	430.00	clock [20]
0.0	22.71	0.00	1	0.00	0.00	exit [33]
0.0	22.71	0.00	1	0.00	8319.99	getBestMove [4]
0.0	22.71	0.00	1	0.00	0.00	getenv [47]
0.0	22.71	0.00	1	0.00	8750.00	main [3]
0.0	22.71	0.00	1	0.00	0.00	mem_init [70]
0.0	22.71	0.00	1	0.00	0.00	number [71]
0.0	22.71	0.00	1	0.00	0.00	scanf [53]

Ways to Optimize Performance

- Better data structures and algorithms
  - Improves the *"asymptotic complexity"*
    - Better scaling of computation/storage as input grows
    - E.g., going from $O(n^2)$ sorting algorithm to $O(n \log n)$
  - Clearly important if large inputs are expected
  - Requires understanding data structures and algorithms
- Better source code the compiler can optimize
  - Improves the *"constant factors"*
    - Faster computation during each iteration of a loop
    - E.g., going from $1000n$ to $10n$ running time
  - Clearly important if a portion of code is running slowly
  - Requires understanding hardware, compiler, execution

Helping the Compiler Do Its Job

Optimizing Compilers

- Provide efficient mapping of program to machine
  - Register allocation
  - Code selection and ordering
  - Eliminating minor inefficiencies
- Don't (usually) improve asymptotic efficiency
  - Up to the programmer to select best overall algorithm
- Have difficulty overcoming "optimization blockers"
  - Potential function side-effects
  - Potential memory aliasing

Limitations of Optimizing Compilers

- Fundamental constraint
  - Compiler must not change program behavior
  - Ever, even under rare pathological inputs
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - Data ranges more limited than variable types suggest
  - Array elements remain unchanged by function calls
- Most analysis is performed only within functions
  - Whole-program analysis is too expensive in most cases
- Most analysis is based only on static information
  - Compiler has difficulty anticipating run-time inputs

Avoiding Repeated Computation

- A good compiler recognizes simple optimizations
  - Avoiding redundant computations in simple loops
  - Still, programmer may still want to make it explicit
- Example
  - Repetition of computation: n * i

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

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

Worrying About Side Effects

- Compiler cannot always avoid repeated computation
  - May not know if the code has a “side effect”
  - … that makes the transformation change the code’s behavior

- Is this transformation okay?

```
int func1(int x) {
  return f(x) + f(x) + f(x) + f(x);
}
```

```
int func1(int x) {
  return 4 * f(x);
}
```

- Not necessarily, if

```
int counter = 0;

int f(int x) {
  return counter++;
}
```

And this function may be defined in another file known only at link time!

Another Example on Side Effects

- Is this optimization okay?

```
for (i = 0; i < strlen(s); i++) {
   /* Do something with s[i] */
}
```

→

```
length = strlen(s);
for (i = 0; i < length; i++) {
   /* Do something with s[i] */
}
```

- Short answer: it depends
  - Compiler often cannot tell
  - Most compilers do not try to identify side effects
- Programmer knows best
  - And can decide whether the optimization is safe

Memory Aliasing

- Is this optimization okay?

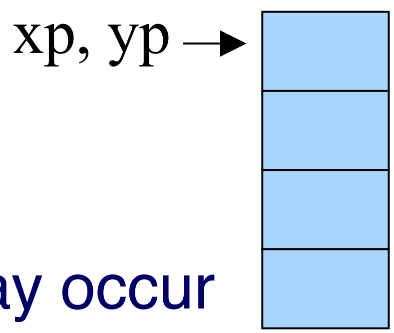
```
void twiddle(int *xp, int *yp) {
    *xp += *yp;
    *xp += *yp;
}
```

```
void twiddle(int *xp, int *yp) {
    *xp += 2 * *yp;
}
```

- Not necessarily, what if xp and yp are equal?
  - First version: result is 4 times *xp
  - Second version: result is 3 times *xp

Memory Aliasing

- Memory aliasing
  - Single data location accessed through multiple names
  - E.g., two pointers that point to the same memory location
- Modifying the data using one name
  - Implicitly modifies the values seen through other names



- Blocks optimization by the compiler
  - The compiler cannot tell when aliasing may occur
  - … and so must forgo optimizing the code
- Programmer often *does* know
  - And *can* optimize the code accordingly

Another Aliasing Example

- Is this optimization okay?

```
int *x, *y;
…
*x = 5;
*y = 10;
printf("x=%d\n", *x);
```

```
printf("x=5\n");
```

- Not necessarily
  - If y and x point to the same location in memory…
  - … the correct output is "x = 10\n"

Summary: Helping the Compiler

- Compiler can perform many optimizations
  - Register allocation
  - Code selection and ordering
  - Eliminating minor inefficiencies
- But often the compiler needs your help
  - Knowing if code is free of side effects
  - Knowing if memory aliasing will not happen
- Modifying the code can lead to better performance
  - Profile the code to identify the “hot spots”
  - Look at the assembly language the compiler produces
  - Rewrite the code to get the compiler to do the right thing

Exploiting the Hardware

Underlying Hardware

- Implements a collection of instructions
  - Instruction set varies from one architecture to another
  - Some instructions may be faster than others
- Registers and caches are faster than main memory
  - Number of registers and sizes of caches vary
  - Exploiting both spatial and temporal locality
- Exploits opportunities for parallelism
  - Pipelining: decoding one instruction while running another
    - Benefits from code that runs in a sequence
  - Superscalar: perform multiple operations per clock cycle
    - Benefits from operations that can run independently
  - Speculative execution: performing instructions before knowing they will be reached (e.g., without knowing outcome of a branch)

Addition Faster Than Multiplication

- Adding instead of multiplying
  - Addition is faster than multiplication
- Recognize sequences of products
  - Replace multiplication with repeated addition

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

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

Bit Operations Faster Than Arithmetic

- Shift operations to multiple/divide by powers of 2
  - “x >> 3” is faster than “x/8”
  - “x << 3” is faster than “x * 8”

53	0	0	1	1	0	1	0	1
53<<2	1	1	0	1	0	0	0	0

- Bit masking is faster than mod operation
  - “x & 15” is faster than “x % 16”

53	0	0	1	1	0	1	0	1
& 15	0	0	0	0	1	1	1	1
5	0	0	0	0	0	1	0	1

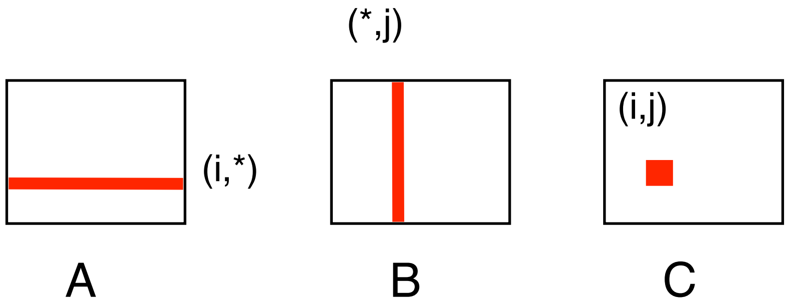
Caching: Matrix Multiplication

- Caches
  - Slower than registers, but faster than main memory
  - Both instruction caches and data caches
- Locality
  - Temporal locality: recently-referenced items are likely to be referenced in near future
  - Spatial locality: Items with nearby addresses tend to be referenced close together in time
- Matrix multiplication
  - Multiply n-by-n matrices A and B, and store in matrix C
  - Performance heavily depends on effective use of caches

Matrix Multiply: Cache Effects

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

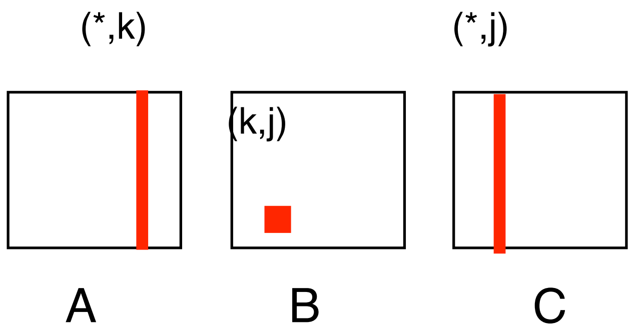
- Reasonable cache effects
  - Good spatial locality for A
  - Poor spatial locality for B
  - Good temporal locality for C



Matrix Multiply: Cache Effects

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

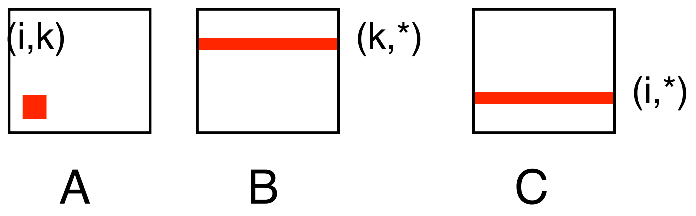
- Rather poor cache effects
  - Bad spatial locality for A
  - Good temporal locality for B
  - Bad spatial locality for C



Matrix Multiply: Cache Effects

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

- Good cache effects
  - Good temporal locality for A
  - Good spatial locality for B
  - Good spatial locality for C



Parallelism: Loop Unrolling

- What limits the performance?

```
for (i = 0; i < length; i++)
  sum += data[i];
```

- Limited apparent parallelism
  - One main operation per iteration (plus book-keeping)
  - Not enough work to keep multiple functional units busy
  - Disruption of instruction pipeline from frequent branches

- Solution: unroll the loop
  - Perform multiple operations on each iteration

Parallelism: After Loop Unrolling

- Original code

```
for (i = 0; i < length; i++)
  sum += data[i];
```

- After loop unrolling (by three)

```
/* Combine three elements at a time */
limit = length - 2;
for (i = 0; i < limit; i+=3)
  sum += data[i] + data[i+1] + data[i+2];

/* Finish any remaining elements */
for ( ; i < length; i++)
  sum += data[i];
```

Program Execution

Avoiding Function Calls

- Function calls are expensive
  - Caller saves registers and pushes arguments on stack
  - Callee saves registers and pushes local variables on stack
  - Call and return disrupt the sequence flow of the code
- Function inlining:

```
void g(void) {
   /* Some code */
}

void f(void) {
   …
   g();
   …
}
```

Some compilers support "inline" keyword directive.

```
void f(void) {
   …
   /* Some code */
   …
}
```

Writing Your Own Malloc and Free

- Dynamic memory management
  - **`malloc()`** to allocate blocks of memory
  - **`free()`** to free blocks of memory
- Existing **`malloc()`** and **`free()`** implementations
  - Designed to handle a wide range of request sizes
  - Good most of the time, but rarely the best for all workloads
- Designing your own dynamic memory management
  - Forego using traditional **`malloc()`** and **`free()`**, and write your own
  - E.g., if you know all blocks will be the same size
  - E.g., if you know blocks will usually be freed in the order allocated
  - E.g., <insert your known special property here>

Conclusion

- Work smarter, not harder
  - No need to optimize a program that is “fast enough”
  - Optimize only when, and where, necessary
- Speeding up a program
  - Better data structures and algorithms: better asymptotic behavior (the “COS 226 way”)
  - Optimized code: smaller constants (the “COS 217 way”)
- Techniques for speeding up a program
  - Coax the compiler
  - Exploit capabilities of the hardware
  - Capitalize on knowledge of program execution