Introduction to Code Optimization

(Speeding-up a single-CPU application written in C using code optimization techniques)

Oscar Peredo

ALGES
ADVANCED LABORATORY FOR
GEOSTATISTICAL SUPERCOMPUTING

Internal document

October 30th, 2013 - Santiago, Chile

Outline

Motivation

Optimizations

- Long latency operations
- Flow control
- Memory consciousness
- Vectorization (SIMDization)

Case studies

- Bioinformatics: protein docking
- Geostatistics: MPS

Comments

Outline

Motivation

Optimizations
- Long latency operations
- Flow control
- Memory consciousness
- Vectorization (SIMDization)

Case studies
- Bioinformatics: protein docking
- Geostatistics: MPS

Comments

Motivation

- How can we accelerate our C-written applications using only a general-purpose single core CPU?
- Can we obtain a *good* speedup just re-coding our application?
- *Code optimization*: set of techniques and code modifications (machine-independent and machine-dependent) that can help us to achieve this task.
- Bibliography:
  - Bryant, R. and O'Hallaron D., *Computer Systems: A Programmer's Perspective*, Pearson International 2nd Edition, 2011/2.
  - *Awareness of Architecture in Programming*, *Computer Architecture* (UPC-Barcelona master courses).
  - Bit twiddling hacks: `http://graphics.stanford.edu/~seander/bithacks.html`
  - Intel optimization manuals.

Outline

Optimization categories

- Long latency operations

  "Try to use bit hacks and avoid division operator $/$."

- Flow control

  "Reduce your branching overhead, avoiding too many if/else or loop iterations."

- Memory consciousness

  "Increase your data temporal/spatial locality and study your cache parameters."

- Vectorization/SIMDization

  "If you gather several arithmetic ops into one vectorial op, less cycles in the execution you will pay."

Outline

Motivation

Optimizations

- Long latency operations
- Flow control
- Memory consciousness
- Vectorization (SIMDization)

Case studies

- Bioinformatics: protein docking
- Geostatistics: MPS

Comments

Long latency operations

- Latency
- Bit hacks
- Arithmetic expressions
- Memoization
- Specialization of generic routines
- Reduction of procedure calls (user/library/OS)

Latency/Throughput

- Execution Latency: Cycles before a result is generated.
- Repetition Rate: Cycles before another instruction is run in the same functional unit.
- Throughput: Average number of instructions per cycle (not the same definition as Intel's manuals).

Latency

	Core2	Pentium4 (Willamette)	Pentium4 (Prescott)	AMD K8
add	1	0.5[1]	1	1
shl	1	4	1	1
imul	3	14	10	3
div[2]	23	70	80	39
load (L1 hit)	3	2	4	3
fp add	3	5	6	4
fp mul	5	7	8	4
fp div[3]	6,6,?	23,38,43	30,40,44	16,20,24

Throughput

	Pentium4 (Willamette)	Pentium4 (Prescott)	AMD K8
add	3	2.5	3
shl	1	1	3
imul	1/4	1	1
div	?	1/34	1/39

1. latency$<$1 due to P4-Willamette has a part with double frequency. 2. worst-case, is data-dependent. 3. Single, double and extended precision.

"Low latency, high throughput..."

Bit hacks

Transform some calculations into bit-based operations (low latency, high throughput ops). Some examples (32-bits):

- Computation of the absolute value of an integer

```
abs = (n ^ (n>>31)) - (n>>31);
```

- Compute the maximum between two integers (2 ways)

```
max = n - ((n-m) & ((n-m)>>31));
max = n - ((n-m) & -(n<m));
```

- Check that bits 4 or 12 are active (1)

```
((n & ((1<<12)|(1<<4))) != 0)
```

- Check that bits 4 and 12 are active (1)

```
((n & ((1<<12)|(1<<4))) == ((1<<12)|(1<<4)))
```

More examples: `http://graphics.stanford.edu/~seander/bithacks.html` and the book *Hacker's Delight* (strongly recommended, but hard to pass).

Arithmetic expressions

If bit hacks are not available, transform some arithmetic operations by less expensive ones (or remove them from scratch if they are not used):

- Integer Multiply: can be overwritten by a sequence of adds (a*3 = a + a + a) or shifts.
- Power with integers: can be substituted by shifts and multiplications.
- Integer division: can be substituted by the inverse multiplication, or shifts (if we divide positive values by a power of two).
- Module by a power of 2: can be substituted by an &.

With `gcc`, the flag `-funsafe-math-optimizations` allows the compiler to apply some arithmetic transformations, losing some numerical accuracy.

Memoization

Pre-compute expensive data and store it in a lookup table. In this way, we replace expensive computations by table accesses.
Example:

`// Main function,`
`// calls sin()/cos() several times`
`int main(int argc, char *argv[]){`
`   unsigned int i, r, j, n;`
`   double d, x, y;`
 	
`   if (argc == 1)`
`      n = N;`
`   else`
`      n = atoi(argv[1]);`
 	
`   srand(0);`
`   for (i=0; i<n; i++){`
`      r = rand();`
`      for (j=0, d=0; j<POINTS; j++){`
`         x = r*cos(d); y = r*sin(d);`
`         fwrite(&x, sizeof(x), 1, stdout);`
`         fwrite(&y, sizeof(y), 1, stdout);`
`         d += 2*M_PI/POINTS;`
`      }`
`   }`
`   return 0;`
`}`
	`// Lookup table`
	`Point lookupTable[POINTS];`
	`for (j=0, d=0; j<POINTS; j++){`
	`   lookupTable[j].cos=cos(d);`
	`   lookupTable[j].sin=sin(d);`
	`   d += 2*M_PI/POINTS;`
	`}`

Specialization

If some routine is always used with the same parameters, we can modify it to match those parameters. Example:

```
// Main function,
// multiplies a 2x2 matrix by a vector
   ...
   for (i=0; i<n; i++){
      for (j=0; j<n; j++){
         y[i] = y[i] + A[i][j]*x[j];
      }
   }
   ...
}
```

```
// Specialized code for n=2
   ...
   y[0] = y[0] + A[0][0]*x[0];
   y[0] = y[0] + A[0][1]*x[1];
   y[0] = y[0] + A[0][2]*x[2];
   y[0] = y[0] + A[0][3]*x[3];
   ...
   y[3] = y[3] + A[3][0]*x[0];
   y[3] = y[3] + A[3][1]*x[1];
   y[3] = y[3] + A[3][2]*x[2];
   y[3] = y[3] + A[3][3]*x[3];
   ...
```

The resulting code may increase in size considerably!

Reduction of procedure calls

Analyze how the library/system calls are used: `ltrace`/`strace`.
If there are too many, try to apply *buffering*.
Example:

```
// Main function,
// reads chars from stdin
// and writes them to stdout
   ...
    while ((n=read(0, &c, 1)) > 0)
    {
        if (write(1, &c, 1) < 0) error();
    }
   ...
}
```

```
// Buffered routine
   ...
   n=read(0, &bufferIN, BUFFSIZE);
   while (n > 0){
      minval= min(n,BUFFSIZE);
      m= write(1, &bufferIN, minval);
      if(m<0) error();
      n=read(0, &bufferIN, BUFFSIZE);
   }
   ...
```

Outline

Flow control

- Branches
- Inlining
- Loop unrolling

Branches

- `if/else/switch/for/while/do/...` and procedure calls generate *branches*.
- Branches are machine instructions (assembly) that perform the flow control of the execution of the program (they break the sequential order of the instructions execution).
- Example of branches (x86): conditionals (`jle`, `je`, ...), inconditionals (`call`, `ret`, `jmp`, ...).
- Branches induce an overhead in the program's execution, adding some delay cycles each time they are processed.
- For this reason, *we must avoid them*.

Branches

Example: lexicographical order or vector elements (`bne` is a branch instruction)

```
do I =1, N
    if (A(I) > A(I+1)) then
        temp = A(I)
        A(I) = A(I+1)
        A(I+1) = temp
        cambios = cambios +1
    endif
enddo
```

1$:	load **r8**, (r7)	load A(I)
	load **r9**, 4 (r7)	load A(I+1)
	cmple **r10**, r8, r9	A(I) ≤ A(I+1)
	bne r10, 2$	se intercambia si A(I) > A(I+1)
	store r9, (r7)	store A(I)
	store r8, 4 (r7)	store A(I+1)
	add **r5**, r5, #1	contador de cambios
2$:	add **r6**, r6, #1	contador de iteraciones
	add **r7**, r7, #4	índice del vector A
	cmple **r11**, r6, r4	r6 ≤ r4
	bne r11, 1$	iterar si aún no ha finalizado

	ciclos																															
instrucción	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
load **r8**, 0(r7)	CP	BUS	D/L	ALU	M	ES																										
load **r9**, 4 (r7)		CP	BUS	D/L	ALU	M	ES																									
cmple **r10**, r8, r9			CP	BUS	D/L	D/L	D/L	ALU	M	ES																						
bne r10, 2$				CP	BUS	BUS	BUS	D/L	D/L	D/L	ALU	M	ES																			
store r9, (r7)													CP	BUS	D/L	ALU	M	ES														
store r8, 4 (r7)														CP	BUS	D/L	ALU	M	ES													
add **r5**, r5, #1															CP	BUS	D/L	ALU	M	ES												
2$: add **r6**, r6, #1																CP	BUS	D/L	ALU	M	ES											
add **r7**, r7, #4																	CP	BUS	D/L	ALU	M	ES										
cmple **r11**, r6, r4																		CP	BUS	D/L	D/L	ALU	M	ES								
bne r11, 1$																			CP	BUS	BUS	D/L	D/L	D/L	ALU	M	ES					
1$: load **r8**, 0(r7)																											CP	BUS	D/L	ALU	M	ES
ciclos perdidos								2(D)			2(D)		4 (S)										1(D)		2 (D)			4 (S)				

Inlining

Use explicit definition of a procedure in place of a function call. The `gcc` compiler do it automatically with the flag `-O3`. Other related inlining flags: `-fno-inline`, `-finline-limit=n`, ...

We can also use manual inlining through macros definition in the preprocessing part of the program (`# define FOO(_x,_y) ...`).

Example:

```
// Main function,
// right shift of ints
...
int r_shift (int x, int y) {
   return x << y;
}
...
}
```

```
// Macro inlined version
...
// correct macro
#define r_shift_ok(_x,_y) ((_x) << (_y))
// incorrect macro, why?
#define r_shift_ko(_x,_y) (_x << _y)
...
```

Loop unrolling

Put more work in each loop iteration, expecting a decrease in the loop-end evaluation overhead.

Example:

`// Main function,`
`// add two vectors`
`...`
`for(i=0; i<N; i++)`
`a[i] = b[i] + c[i];`
`...`
`}`
	`// Loop unrolled, with degree 4`
	`...`
	`for(i=0; i<(N-3); i+=4) {`
	`a[i] = b[i] + c[i];`
	`a[i+1] = b[i+1] + c[i+1];`
	`a[i+2] = b[i+2] + c[i+2];`
	`a[i+3] = b[i+3] + c[i+3];`
	`}`
	`for(; i<N; i++) {`
	`a[i] = b[i] + c[i];`
	`}`
	`...`

Outline

Memory consciousness

- What is locality?
- Memory mountain
- Avoid aliasing
- Data alignment
- Memory bandwidth

Spatial Locality

Add all the elements of a 2D array:

```
int sumarrayrows(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;
}
```

```
int sumarraycols(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;
}
```

With `N=M=20000`, the elapsed time is 5s and 19s respectively. Why? Stride-k accesses, with $k > 1$, are *bottlenecks*. If `M=2,N=3`:

`sumarrayrows` **(stride-1 access pattern)**

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	2	3	4	5	6

`sumarraycols` **(stride-3 access pattern)**

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	3	5	2	4	6

"Higher stride access pattern $\Rightarrow$ Lower spatial locality $\Rightarrow$ Higher elapsed time"

Temporal Locality

Matrix multiplication:

```
void matxmat1(int **a, int **b, int **c){
 int i, j, k;
 for (k = 0; k < N; k++){
  for (i = 0; i < N; i++){
   for (j = 0; j < N; j++){
    c[i][j] = c[i][j] + a[i][k]*b[k][j];
   }
  }
 }
}
```

```
void matxmat2(int **a, int **b, int **c){
 int i, j, k, tmp;
 for (k = 0; k < N; k++){
  for (i = 0; i < N; i++){
   tmp = a[i][k];
   for (j = 0; j < N; j++){
    c[i][j] = c[i][j] + tmp*b[k][j];
   }
  }
 }
}
```

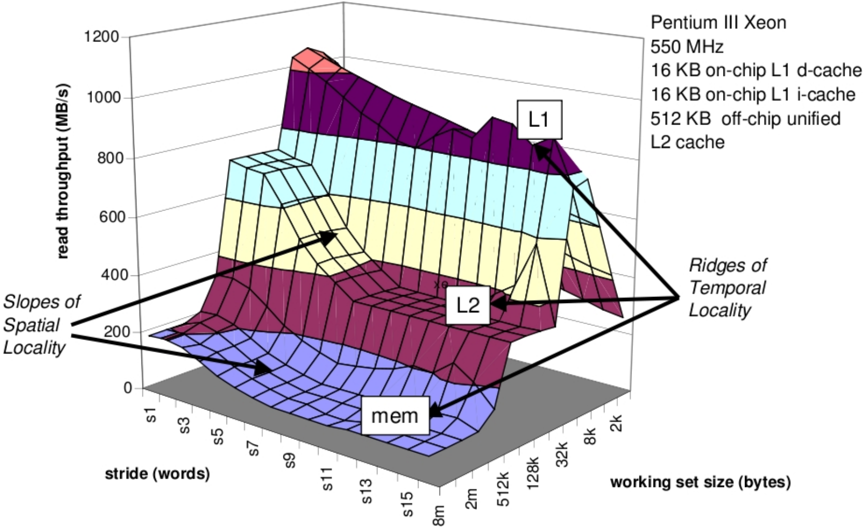
With `N=2048`, the elapsed time is 89s and 73s respectively (20% efficiency). Why?

	Memory accesses (read/writes)
`matxmat1`	$N^2 \times 4N$
`matxmat2`	$N^2 \times (3N+1)$

Speedup in memory accesses: $\frac{4N^3}{N^2 \times (3N+1)} = \frac{4}{3+\frac{1}{N}} \approx 1.333$

"Use register blocking and cache blocking (always)..."

Memory mountain



"Build the memory mountain in your machine and see where are your slopes and ridges."

Avoid aliasing

Compiler uses registers to avoid memory accesses, but sometimes it is not posible (even using optimization flags):

```
void multiply(int *x, int *y, int n){
   int i;
   for (i=0; i<n; i++)
      *x = *x + *y;
}
```

```
void multiply(int *x, int *y, int n){
   int i, tmp_x = *x, tmp_y = *y;
   for (i=0; i<n; i++)
      tmp_x = tmp_x + tmp_y;
   *x = tmp_x;
}
```

With $\texttt{N=2*10}^9$, the elapsed time is 8.71s and 5.55s respectively (57% efficiency). Why?

In presence of memory accesses, the compiler generates conservative code, because a memory location can be accessed using many variables. This is known as *aliasing*.
Example:

```
int y,*x; y=1; x=&y; printf("%d",*x); y=2; printf("%d",*x);
```

"Avoid aliasing and innecesary pointer references."

Data alignment

- A variable of 2^n bytes is already properly aligned if it is stored at a memory address that is divisible by 2^n:
  - An integer (4 bytes) stored in [$0xb0000004$, $0xb0000007$] is properly aligned:

$$0xb0000004_{hex} = 2952790020_{dec} = \underbrace{738197505}_{\in \mathbb{N}} \times 2^2$$

  - An integer (4 bytes) stored in [$0xb000000a$, $0xb000000d$] is not properly aligned:

$$0xb000000A_{hex} = 2952790026_{dec} = \underbrace{738197506.5}_{\notin \mathbb{N}} \times 2^2$$

- The access time to a data may depend on the data alignment. In IA-32, it is *faster* to access an integer (of 4 bytes) stored at $0xb0000004$ than to another stored at $0xb0000006$.
- Static data alignment: the compiler aligns the data your you. It uses *padding* (add more space at the end of some data structures to keep the good alignment). Use `gcc -Wpadded` to see where padding was introduced.
- Dynamic data alignment: the user allocates aligned memory. `malloc` returns memory aligned to 8 bytes, `posix_memalign` return memory aligned to the user needs.

"Check if your data is properly aligned to the container datatype"

Memory bandwidth

- Bandwidth: transfered bytes per second.
- Higher bandwidth implies faster memory accesses.
- Also, if we taking profit of the bandwidth, a reduction in the number of memory accesses can be obtained.

Examples:

```
// Re-organize structs
struct {    | struct {
   short s;|    char c;
   char c; |    char d;
   int i;  |    short s;
   char d; |    int i;
} ex3;     | } ex3_sorted;
// 12 bytes| 8 bytes
```

```
// Write 4 bytes each time
// to a char buffer
unsigned char buff[N];
unsigned int k, i;
...
*((unsigned int*) &buff[i]) = k;
i = i + 4;
// Transfering 4 chars
// packed into 1 int
```

”Exploit you machine’s bandwidth to decrease memory accesses time/number”

Outline

Vectorization

- Vector registers
- (Non) Vectorizable loops
- Autovectorization
- SIMD programming

Vector registers

New ISA vector extensions: MMX, SSE, SSE2, SSE3, SSE4, 3DNow!, FMA, CVT, XOP, AVX,...

```
#include <stdio.h>
#include <emmintrin.h>

// New variables:
// Vector registers (128 bits) SSE2
__m128i A1;
__m128i B1;
__m128i BITMASK;
__m128i A2;
__m128i B2;
...
```

Data type	Size (bytes/bits)	In 128 bits?
char	1/8	16
shot	2/16	8
int	4/32	4
float	4/32	4
double	8/64	2

Idea: use those 128 bits to apply vector operations to 16 chars, 8 shots, 4 ints, 4 floats or 2 doubles, *simultaneously*.

(Non)Vectorizable loops

In order to vectorize a loop, we need to study the *data dependencies* inside the loop.

```
// Vectorizable loops
...
char A[16], B[16], C[16];
for (i=0; i<16; i++)
   A[i] = B[i] + C[i];
...
...
for (i=0; i<(N-1); i++)
   A[i] = A[i+1] + B[i];
...
...
for (i=0; i<(N-3); i++) {
   A[i] = A[i+1] + B[i];
   B[i+1] = B[i+2] + B[i+3];
}
// why?
```

```
// Non vectorizable loops
...
char A[16], B[16], C[16];
for (i=1; i<16; i++)
   A[i] = A[i-1] + B[i];
...
...
while (*p != NULL) {
  *q++ = *p++;
}
// why?
```

It's not easy to identify non-vectorizable loops and modified them, the only way is with practice and experience. More samples of vectorizable loops:

http://gcc.gnu.org/projects/tree-ssa/vectorization.html#vectorizab

Autovectorization

If we are not experts in vectorization, we can still autovectorize our code, using the `gcc` compiler flags `-O3 -ftree-vectorize`.

```
// loop.c
#include <stdio.h>
#define N 128
int a[N], b[N];
void foo (void){
   int i;
   for (i = 0; i < N; i++)
      a[i] = i;
   for (i = 0; i < N; i+=5)
      b[i] = i;
}
int main(){
   foo();
}
```

```
$> gcc -c -O3 -ftree-vectorizer-verbose=2 loop.c
Analyzing loop at loop.c:12
12: not vectorized: complicated access pattern.
Analyzing loop at loop.c:9
Vectorizing loop at loop.c:9
9: LOOP VECTORIZED.
loop.c:5: note: vectorized 1 loops in function.
Analyzing loop at loop.c:12
12: not vectorized: complicated access pattern.
Analyzing loop at loop.c:9
Vectorizing loop at loop.c:9
9: LOOP VECTORIZED.
loop.c:16: note: vectorized 1 loops in function.
```

The generated code is different:

```
// no vectorized
foo:
   ...
   movl %eax, a(,%rax,4)
   addq $1, %rax
   cmpq $128, %rax
   jne .L2
   movl $b, %edx
   xorb %al, %al
   ...
```

```
// vectorized
foo:
   ...
   movdqa %xmm0, (%rax)   <- vector
   paddd %xmm1, %xmm0     <- instructions
   cmpq $a+512, %rax
   jne .L2
   movl $b, %edx
   xorl %eax, %eax
   ...
```

SIMD programming

If autovectorization doesn't help, you can still try to re-code your application using intrinsic instructions:

```
// non-SIMDized
void vector_add(
   int *A,
   int *B,
   int *C,
   int len,
   int N_iter)
{
   int i,j;
   for (j=0; j<N_iter; j++)
      for (i=0;i<len;i++)
         C[i] = A[i] + B[i];
}
```

```
// SIMDized
void vector_add(
   int *A,
   int *B,
   int *C,
   int len,
   int N_iter)
{
   int i,j, step;
   step = sizeof(__m128i)/sizeof(int);
   for (j=0; j<N_iter; j++)
      for (i=0; i<len; i=i+step) {
         __m128i *pa, *pb, *pc;
         pa = (__m128i*) &A[i];
         pb = (__m128i*) &B[i];
         pc = (__m128i*) &C[i];
         *pc = INTR(*pa, *pb);
      }
}
```

Elapsed time: 11.89s and 4.90s respectively (speedup 2.42x), compiling with `gcc -march=native -msse2`. More information: Intel optimization manuals.

Outline

Outline

Protein docking

Target: optimize electrostatic matrix computation code (80% of elapsed time).

Optimization	Time [seconds]	Speedup
Base	62.650	1x
Specialization of routines	56.690	1.105x
Branch reduction	38.422	1.631x
Vectorization	31.130	2.013x
Long latency ops reduction	30.764	2.036x
Loop unrolling	28.862	2.170x
Lookup table	24.850	2.521x
`pthreads` (2 cores)	18.756	3.340x

Outline

Multiple-point statistics

Target: optimize fitness/cost function of multiple-point simulation (96% of elapsed time, 1000x1000 images).

	100×100		1000×1000	
Optimization	Time (seconds)	Speed up	Time (seconds)	Speed up
Baseline	0.00319	1	0.402347	1
Increase data locality	0.00213	1.49x	0.26368	1.53x
Use Stack memory	0.00208	1.53x	0.257903	1.56x
Specialization of fitness	0.00181	1.76x	0.243201	1.65x
Branch reduction	0.00160	1.99x	0.222511	1.80x
Load reduction	0.00157	2.03x	0.214222	1.87x
`OpenMP` (2 cores)	0.002028	1.57x	0.109205	3.68x
`OpenMP` (4 cores)	0.012321	0.25x	0.056436	7.12x
`OpenMP` (8 cores)	0.027145	0.11x	0.036421	11.04x

Outline

Comments

- First of all, identify your bottlenecks or *kernels* (use `gprof` or any profiler).
- Try to optimize only those kernel codes (its difficult, but the reward worth it).
- After optimize them, try to parallelize them (its also difficult, but the speedup will be higher), using any technology available.
- If new technologies arrive, re-optimize your kernels to target them (in this way, your application can work in many architectures).
- Be methodical, organized and have good luck.

Thanks for your attention!

Contact (author): `operedo [at] alges.cl`

`http://www.alges.cl`