HOWTO WRITE FAST NUMERICAL CODE
EXERCISE 6

Pascal Spörri
pascal@spoerri.io

May 11, 2013

1 Warmup

The code of this exercise has been run on an Intel Core i5-3570K CPU and Ubuntu 12.04 with gcc 4.6.3.

```
1 void warmup(float *x, float *y, int size, float alpha)
2 {
3     #pragma ivdep
4     int i;
5
6     #pragma vector aligned
7     for (i=0; i<size; i++)
8     {
9         y[i] = x[2*i]+x[2*i]+x[2*i+1]/alpha;
10    }
11 }
```

Listing 1: Code of `base.c`

1.1 Baseline

The code of `base.c` has been compiled with the compiler flags: `-m64 -march=corei7-avx -fno -tree-vectorize -O3 -S`.

1.1.0.1 Assembly code:

```
1   .file "base.c"
2   .text
3   .p2align 4,,15
4   .globl  warmup
5   .type warmup, @function
6 warmup:
7 .LFB0:
8   .cfi_startproc
```

```
  testl %edx, %edx
  jle .L1
  xorl  %eax, %eax
  .p2align 4,,10
  .p2align 3
.L3:
  vmovss  4(%rdi,%rax,8), %xmm2
  vmovss  (%rdi,%rax,8), %xmm1
  vdivss  %xmm0, %xmm2, %xmm2
  vaddss  %xmm1, %xmm1, %xmm1
  vaddss  %xmm2, %xmm1, %xmm1
  vmovss  %xmm1, (%rsi,%rax,4)
  addq  $1, %rax
  cmpl  %eax, %edx
  jg  .L3
.L1:
  rep
  ret
  .cfi_endproc
.LFE0:
  .size warmup, .-warmup
  .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
  .section  .note.GNU-stack,"",@progbits
```

Listing 2: Compiler output of `base.c` using the flags `-m64 -march=corei7-avx -fno-tree-vectorize -O3 -S`

We observe that no vector instructions have been used.

1.2 Autovectorization

The code of `auto.c` has been compiled with the compiler flags: `-m64 -march=corei7-avx -O3 -S`.

```
  .file "auto.c"
  .text
  .p2align 4,,15
  .globl  warmup
  .type warmup, @function
warmup:
.LFB0:
  .cfi_startproc
  pushq %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq  %rsp, %rbp
  .cfi_def_cfa_register 6
  andq  $-32, %rsp
  addq  $16, %rsp
  testl %edx, %edx
```

```
17   jle .L1
18   movl  %edx, %r9d
19   movl  %edx, %ecx
20   shrl  $3, %r9d
21   leal  0(,%r9,8), %r8d
22   testl %r8d, %r8d
23   je  .L6
24   leaq  (%rdi,%rcx,8), %rax
25   cmpl  $7, %edx
26   leaq  (%rsi,%rcx,4), %rcx
27   seta  %r10b
28   cmpq  %rax, %rsi
29   seta  %al
30   cmpq  %rcx, %rdi
31   seta  %cl
32   orl %ecx, %eax
33   testb %al, %r10b
34   je  .L6
35   vmovaps .LC0(%rip), %ymm6
36   xorl  %eax, %eax
37   xorl  %ecx, %ecx
38   vshufps $0, %xmm0, %xmm0, %xmm7
39   vinsertf128 $1, %xmm7, %ymm7, %ymm7
40   .p2align 4,,10
41   .p2align 3
42 .L4:
43   vmovups (%rdi,%rax,2), %xmm4
44   addl  $1, %ecx
45   vmovups 32(%rdi,%rax,2), %xmm2
46   vinsertf128 $0x1, 16(%rdi,%rax,2), %ymm4, %ymm4
47   vinsertf128 $0x1, 48(%rdi,%rax,2), %ymm2, %ymm2
48   vshufps $136, %ymm2, %ymm4, %ymm3
49   vshufps $221, %ymm2, %ymm4, %ymm2
50   vperm2f128  $3, %ymm3, %ymm3, %ymm1
51   vshufps $68, %ymm1, %ymm3, %ymm5
52   vshufps $238, %ymm1, %ymm3, %ymm1
53   vperm2f128  $32, %ymm1, %ymm5, %ymm1
54   vmulps  %ymm6, %ymm1, %ymm3
55   vperm2f128  $3, %ymm2, %ymm2, %ymm1
56   vshufps $68, %ymm1, %ymm2, %ymm4
57   vshufps $238, %ymm1, %ymm2, %ymm1
58   vperm2f128  $32, %ymm1, %ymm4, %ymm1
59   vdivps  %ymm7, %ymm1, %ymm1
60   vaddps  %ymm1, %ymm3, %ymm1
61   vmovups %xmm1, (%rsi,%rax)
62   vextractf128  $0x1, %ymm1, 16(%rsi,%rax)
63   addq  $32, %rax
64   cmpl  %ecx, %r9d
65   ja  .L4
```

```
66    cmpl  %r8d, %edx
67    je  .L1
68  .L3:
69    leal  (%r8,%r8), %ecx
70    movslq  %r8d, %r9
71    xorl  %eax, %eax
72    movslq  %ecx, %rcx
73    leaq  (%rdi,%rcx,4), %rcx
74    leaq  (%rsi,%r9,4), %rdi
75    .p2align 4,,10
76    .p2align 3
77  .L5:
78    vmovss  4(%rcx,%rax,8), %xmm2
79    vmovss  (%rcx,%rax,8), %xmm1
80    vdivss  %xmm0, %xmm2, %xmm2
81    vaddss  %xmm1, %xmm1, %xmm1
82    vaddss  %xmm2, %xmm1, %xmm1
83    vmovss  %xmm1, (%rdi,%rax,4)
84    addq  $1, %rax
85    leal  (%r8,%rax), %esi
86    cmpl  %esi, %edx
87    jg  .L5
88  .L1:
89    leave
90    .cfi_remember_state
91    .cfi_def_cfa 7, 8
92    vzeroupper
93    ret
94  .L6:
95    .cfi_restore_state
96    xorl  %r8d, %r8d
97    jmp .L3
98    .cfi_endproc
99  .LFE0:
100   .size warmup, .-warmup
101   .section  .rodata.cst32,"aM",@progbits,32
102   .align 32
103 .LC0:
104   .long 1073741824
105   .long 1073741824
106   .long 1073741824
107   .long 1073741824
108   .long 1073741824
109   .long 1073741824
110   .long 1073741824
111   .long 1073741824
112   .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
113   .section  .note.GNU-stack,"",@progbits
```

Listing 3: Compiler output of `auto.c` using the flags `-m64 -march=corei7-avx -O3 -S`

We observe that the code contains multiple versions of the basecode. One vectorised version that uses SSE and one non vectorised version that uses non-SSE code. The code tries to select the optimal variant.

1.3 Manual Vectorization

We decided to implement the manual step using AVX registers:

```
#include <immintrin.h>
void warmup(float *x, float *y, int size, float alpha)
{
    #pragma ivdep
    int i;

    __m256 m = _mm256_set_ps(1.0/alpha, 2.0, 1.0/alpha, 2.0,
       1.0/alpha, 2.0, 1.0/alpha, 2.0);
    #pragma vector aligned
    for (i=0; i<size; i+=4)
    {
        __m256 t = _mm256_load_ps(x+2*i);
        __m256 l = _mm256_mul_ps(t, m); // premultiply
        __m256 r = _mm256_permute2f128_ps( l , l , 1); // swap
           lower and higher 128 bits
        __m256 res = _mm256_hadd_ps(l, r);
        __m128 s = _mm256_extractf128_ps (res, 0);
        _mm_store_ps(y+i,s); // store it
    }
}
```

1.4 Performance

Code	Flops/Cycle	Speedup	Compiler Flags
`base.c`	0.469	Baseline	`-m64 -march=corei7-avx -fno-tree-vectorize -O3`
`auto.c`	1.909	$4.07x$	`-m64 -march=corei7-avx -O3`
`manual.c`	3.18	$6.78x$	`-m64 -march=corei7-avx -O3`

Figure 1: Baseline comparison on an Intel Core i5-3570K CPU, Ubuntu 12.04 with gcc 4.6.3. Inputsize: 800

We observe a $6.78x$ speedup with respect to our the non-vectorized implementation and a $1.67x$ speedup with repsect to our auto vectorized implementation.

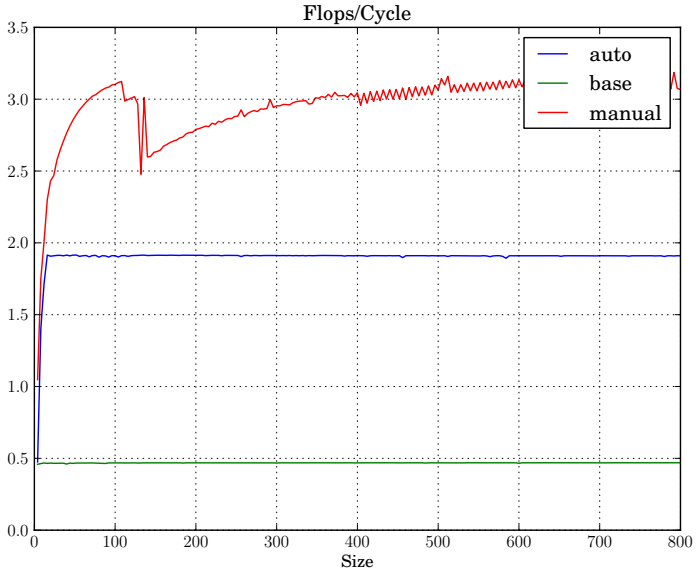

Figure 2: Comparsion plot of `base.c`, `auto.c` and `manual.c` using the compiler options described in figure 1 on an Intel Core i5-3570K CPU and Ubuntu 12.04 with gcc 4.6.3.

2 Vectorization

The code of this exercise has been run on an Intel Core i5-3570K CPU and Ubuntu 12.04 with gcc 4.6.3.

```
1  void FIR(float *y, float *x, float h0, float h1, float h2,
       float h3, int size)
2  {
3       #pragma ivdep
4       int i;
5
6       #pragma vector aligned
7       for (i=0; i<size-3; i++)
8       {
9            y[i] = h3*x[i] + h2*x[i+1] + h1*x[i+2] + h0*x[i+3];
10      }
11 }
```

Listing 4: Code of `base.c`

2.1 Baseline

The code of `base.c` has been compiled with the compiler flags: `-m64 -march=corei7-avx -fno -tree-vectorize -O3 -S`.

2.1.0.2 Assembly code:

```
  .file "base.c"
  .text
  .p2align 4,,15
  .globl  FIR
  .type FIR, @function
FIR:
.LFB0:
  .cfi_startproc
  subl  $3, %edx
  testl %edx, %edx
  jle .L1
  xorl  %eax, %eax
  .p2align 4,,10
  .p2align 3
.L3:
  vmulss  (%rsi,%rax,4), %xmm3, %xmm5
  vmulss  4(%rsi,%rax,4), %xmm2, %xmm4
  vaddss  %xmm4, %xmm5, %xmm4
  vmulss  8(%rsi,%rax,4), %xmm1, %xmm5
  vaddss  %xmm5, %xmm4, %xmm4
  vmulss  12(%rsi,%rax,4), %xmm0, %xmm5
  vaddss  %xmm5, %xmm4, %xmm4
  vmovss  %xmm4, (%rdi,%rax,4)
  addq  $1, %rax
  cmpl  %eax, %edx
  jg  .L3
.L1:
  rep
  ret
  .cfi_endproc
.LFE0:
  .size FIR, .-FIR
  .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
  .section  .note.GNU-stack,"",@progbits
```

Listing 5: Compiler output of `base.c` using the flags `-m64 -march=corei7-avx -fno-tree-vectorize -O3 -S`

We observe that no vector instructions have been used.

2.2 Autovectorization

The code of `auto.c` has been compiled with the compiler flags: `-m64 -march=corei7-avx -O3 -S`.

```
  .file "auto.c"
  .text
  .p2align 4,,15
```

```
  .globl  FIR
  .type FIR, @function
FIR:
.LFB0:
  .cfi_startproc
  pushq %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  subl  $3, %edx
  movq  %rsp, %rbp
  .cfi_def_cfa_register 6
  andq  $-32, %rsp
  addq  $16, %rsp
  testl %edx, %edx
  jle .L1
  leaq  32(%rdi), %r9
  movl  %edx, %r8d
  shrl  $3, %r8d
  leal  0(,%r8,8), %r11d
  testl %r11d, %r11d
  je  .L6
  leaq  32(%rsi), %rax
  cmpl  $7, %edx
  seta  %cl
  cmpq  %rax, %rdi
  seta  %r10b
  cmpq  %r9, %rsi
  seta  %al
  orl %eax, %r10d
  leaq  36(%rsi), %rax
  andl  %r10d, %ecx
  cmpq  %rax, %rdi
  leaq  4(%rsi), %rax
  seta  %r10b
  cmpq  %rax, %r9
  setb  %al
  orl %eax, %r10d
  leaq  40(%rsi), %rax
  andl  %r10d, %ecx
  cmpq  %rax, %rdi
  leaq  8(%rsi), %rax
  seta  %r10b
  cmpq  %rax, %r9
  setb  %al
  orl %eax, %r10d
  leaq  44(%rsi), %rax
  andl  %r10d, %ecx
  cmpq  %rax, %rdi
  leaq  12(%rsi), %rax
```

```
  seta  %r10b
  cmpq  %rax, %r9
  setb  %al
  orl %eax, %r10d
  testb %r10b, %cl
  je  .L6
  xorl  %eax, %eax
  xorl  %ecx, %ecx
  vshufps $0, %xmm3, %xmm3, %xmm9
  vshufps $0, %xmm2, %xmm2, %xmm8
  vshufps $0, %xmm1, %xmm1, %xmm7
  vshufps $0, %xmm0, %xmm0, %xmm6
  vinsertf128 $1, %xmm9, %ymm9, %ymm9
  vinsertf128 $1, %xmm8, %ymm8, %ymm8
  vinsertf128 $1, %xmm7, %ymm7, %ymm7
  vinsertf128 $1, %xmm6, %ymm6, %ymm6
  .p2align 4,,10
  .p2align 3
.L4:
  vmovups (%rsi,%rax), %xmm5
  addl  $1, %ecx
  vmovups 4(%rsi,%rax), %xmm4
  vinsertf128 $0x1, 16(%rsi,%rax), %ymm5, %ymm5
  vmulps  %ymm9, %ymm5, %ymm5
  vinsertf128 $0x1, 20(%rsi,%rax), %ymm4, %ymm4
  vmulps  %ymm8, %ymm4, %ymm4
  vaddps  %ymm4, %ymm5, %ymm4
  vmovups 8(%rsi,%rax), %xmm5
  vinsertf128 $0x1, 24(%rsi,%rax), %ymm5, %ymm5
  vmulps  %ymm7, %ymm5, %ymm5
  vaddps  %ymm5, %ymm4, %ymm4
  vmovups 12(%rsi,%rax), %xmm5
  vinsertf128 $0x1, 28(%rsi,%rax), %ymm5, %ymm5
  vmulps  %ymm6, %ymm5, %ymm5
  vaddps  %ymm5, %ymm4, %ymm4
  vmovups %xmm4, (%rdi,%rax)
  vextractf128  $0x1, %ymm4, 16(%rdi,%rax)
  addq  $32, %rax
  cmpl  %ecx, %r8d
  ja  .L4
  cmpl  %r11d, %edx
  movl  %r11d, %eax
  je  .L1
.L3:
  movslq  %eax, %rcx
  salq  $2, %rcx
  addq  %rcx, %rsi
  addq  %rcx, %rdi
  .p2align 4,,10
```

```
102   .p2align 3
103 .L5:
104   vmulss  (%rsi), %xmm3, %xmm5
105   addl  $1, %eax
106   vmulss  4(%rsi), %xmm2, %xmm4
107   vaddss  %xmm4, %xmm5, %xmm4
108   vmulss  8(%rsi), %xmm1, %xmm5
109   vaddss  %xmm5, %xmm4, %xmm4
110   vmulss  12(%rsi), %xmm0, %xmm5
111   addq  $4, %rsi
112   vaddss  %xmm5, %xmm4, %xmm4
113   vmovss  %xmm4, (%rdi)
114   addq  $4, %rdi
115   cmpl  %eax, %edx
116   jg  .L5
117 .L1:
118   leave
119   .cfi_remember_state
120   .cfi_def_cfa 7, 8
121   vzeroupper
122   ret
123 .L6:
124   .cfi_restore_state
125   xorl  %eax, %eax
126   jmp .L3
127   .cfi_endproc
128 .LFE0:
129   .size FIR, .-FIR
130   .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
131   .section  .note.GNU-stack,"",@progbits
```

Listing 6: Compiler output of `auto.c` using the flags `-m64 -march=corei7-avx -O3 -S`

We note that the compiler made heavy use of AVX instructions. The code also contains a loop version that works with arbitrary input sizes.

2.3 Manual Vectorization

Based on the output of the automatic vectorization we decided to implement a first version using AVX registers:

```
1 #include <immintrin.h>
2 #include <stdio.h>
3 void FIR(float *y, float *x, float h0, float h1, float h2,
      float h3, int size)
4 {
5     int i;
6     __m256 m0 = _mm256_set_ps(0.0,0.0,0.0,0.0, h0, h1, h2, h3);
7     __m256 m1 = _mm256_set_ps(0.0,0.0,0.0, h0, h1, h2, h3,0.0);
8     __m256 m2 = _mm256_set_ps(0.0,0.0, h0, h1, h2, h3,0.0,0.0);
```

```
9       __m256 m3 = _mm256_set_ps(0.0, h0, h1, h2, h3,0.0,0.0,0.0);
10
11      // use 32byte alignment
12      for (i=0; i<size-3; i+=4)
13      {
14          __m256 l = _mm256_loadu_ps(x+i);
15          __m256 a0 = _mm256_mul_ps(l,m0);
16          __m256 a1 = _mm256_mul_ps(l,m1);
17          __m256 a2 = _mm256_mul_ps(l,m2);
18          __m256 a3 = _mm256_mul_ps(l,m3);
19
20          // z0 is a0
21          __m256 z1 = _mm256_hadd_ps(a1, _mm256_permute2f128_ps(
                a1 , a1 , 1));
22          __m256 z2 = _mm256_hadd_ps(a2, _mm256_permute2f128_ps(
                a2 , a2 , 1));
23          __m256 z3 = _mm256_hadd_ps(a3, _mm256_permute2f128_ps(
                a3 , a3 , 1));
24
25
26          __m256 p0 = _mm256_hadd_ps(a0, z1);
27          __m256 p1 = _mm256_hadd_ps(z2, z3);
28
29          __m256 res = _mm256_hadd_ps(p0, p1);
30
31          __m128 s = _mm256_extractf128_ps (res, 0);
32          _mm_store_ps(y+i,s); // store it
33      }
34  }
```

Listing 7: First version of the manual vectorization (`manual.c`).

Since this version of code didn't provide any speedup compared to our non vectorized implementation (see figure 3) we decided to implement a version using SSE instructions:

```
1  #include <immintrin.h>
2  #include <stdio.h>
3  void FIR(float *y, float *x, float h0, float h1, float h2,
       float h3, int size)
4  {
5       int i;
6       __m128 m0 = _mm_set_ps(h3,h3,h3,h3);
7       __m128 m1 = _mm_set_ps(h2,h2,h2,h2);
8       __m128 m2 = _mm_set_ps(h1,h1,h1,h1);
9       __m128 m3 = _mm_set_ps(h0,h0,h0,h0);
10      __m128 l, r, a0, a1, a2, a3, b0, b1, b2, b3;
11      __m128i il, ir;
12
13      l = _mm_load_ps(x+i);
14      il = _mm_castps_si128(l);
```

```
15      for (i=0; i<size-3; i+=4)
16      {
17          r = _mm_load_ps(x+i+4);
18          ir = _mm_castps_si128(r);
19
20          a0 = l;
21          a1 = _mm_castsi128_ps(_mm_alignr_epi8(ir,il,4));
22          a2 = _mm_castsi128_ps(_mm_alignr_epi8(ir,il,8));
23          a3 = _mm_castsi128_ps(_mm_alignr_epi8(ir,il,12));
24          b0 = _mm_mul_ps(a0,m0);
25          b1 = _mm_mul_ps(a1,m1);
26          b2 = _mm_mul_ps(a2,m2);
27          b3 = _mm_mul_ps(a3,m3);
28
29          __m128 s1 = _mm_add_ps(b0,b1);
30          __m128 s2 = _mm_add_ps(b2,b3);
31          __m128 s = _mm_add_ps(s1,s2);
32          _mm_store_ps(y+i,s); // store it
33
34          l = r;
35          il = ir;
36      }
37  }
```

Listing 8: Version of the manual vectorization using SSE instructions (`manual3.c`).

Using this version we were better than our baseline but still slower than the automatically generated vectorization (see figure 3).

For our final version we decided to replace the `_mm_alignr_epi8` expressions with `_mm_loadu_ps` expressions:

```
1  #include <immintrin.h>
2  #include <stdio.h>
3  void FIR(float *y, float *x, float h0, float h1, float h2,
       float h3, int size)
4  {
5       int i;
6       __m128 m0 = _mm_set_ps(h3,h3,h3,h3);
7       __m128 m1 = _mm_set_ps(h2,h2,h2,h2);
8       __m128 m2 = _mm_set_ps(h1,h1,h1,h1);
9       __m128 m3 = _mm_set_ps(h0,h0,h0,h0);
10      __m128 l, r, a0, a1, a2, a3, b0, b1, b2, b3;
11      __m128i il, ir;
12
13      for (i=0; i<size-3; i+=4)
14      {
15          a0 = _mm_load_ps(x+i);
16          a1 = _mm_loadu_ps(x+i+1);
17          a2 = _mm_loadu_ps(x+i+2);
18          a3 = _mm_loadu_ps(x+i+3);
```

```
19          b0 = _mm_mul_ps(a0,m0);
20          b1 = _mm_mul_ps(a1,m1);
21          b2 = _mm_mul_ps(a2,m2);
22          b3 = _mm_mul_ps(a3,m3);
23
24          __m128 s1 = _mm_add_ps(b0,b1);
25          __m128 s2 = _mm_add_ps(b2,b3);
26          __m128 s = _mm_add_ps(s1,s2);
27          _mm_store_ps(y+i,s); // store it
28      }
29  }
```

Listing 9: Improved code based on `manual3.c` (`manual4.c`).

Using this code we were able to improve the our manually tuned code. We were able to achieve a $3.36x$ speedup compared to our baseline (see figure 3).

2.4 Performance

Code	Flops/Cycle	Speedup	Compiler Flags
`base.c`	1.92	Baseline	`-m64 -march=corei7-avx -fno-tree-vectorize -O3`
`auto.c`	6.65	$3.45x$	`-m64 -march=corei7-avx -O3`
`manual.c`	1.92	$1.00x$	`-m64 -march=corei7-avx -O3`
`manual3.c`	6.02	$3.13x$	`-m64 -march=corei7-avx -O3`
`manual4.c`	6.45	$3.36x$	`-m64 -march=corei7-avx -O3`

Figure 3: Baseline comparison on an Intel Core i5-3570K CPU, Ubuntu 12.04 with gcc 4.6.3.

We observe a $3.36x$ speedup with our `manual4.c` code (compiled with `-m64 -march=corei7-avx -O3` using GCC 4.6.3) compared to the non-vectorized baseline `base.c` (compiled with `-m64 -march=corei7-avx -fno-tree-vectorize -O3` using GCC 4.6.3). A performance plot has been provided here:

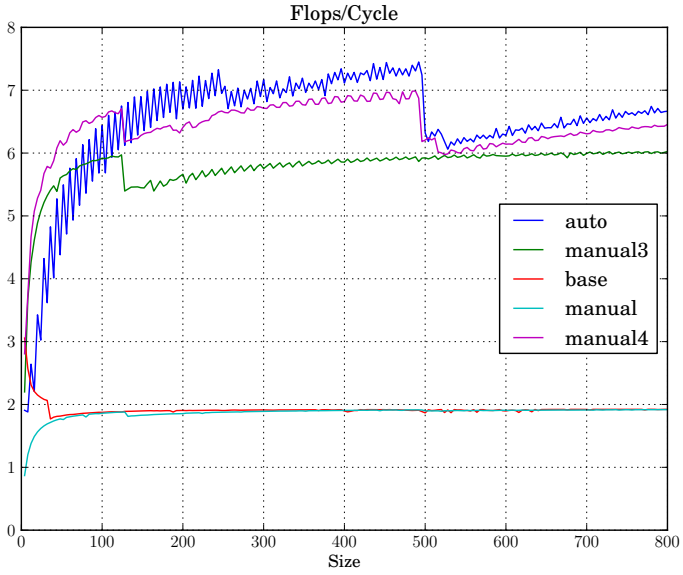

Figure 4: Comparsion plot of `base.c`, `auto.c` and `manual.c` using the compiler options described in figure 3 on an Intel Core i5-3570K CPU and Ubuntu 12.04 with gcc 4.6.3.