PDC Summer School 2011
Brief introduction to
Serial performance measurement

KTH, August 22, 2011

Basic questions on HPC systems

We've seen that hardware is complicated (cache memories, TLBs, SSE-vectorization) and that the way we code impacts performance.

Most of this is hidden from the coder – we have to evaluate performance empirically, *a posteriori*, at various levels of interest:

- Which functions in my (large) code take the most time?
- Can the code be optimized? Where? Is it worth the effort?
- What did the compiler do with my code?
- Is arithmetic handled in SSE-registers or on the FPU stack?
- Are fused arithmetic operations (SSE) issued?
- Does my code generate a lot of TLB misses?
- Are branch mispredictions causing stalls?

We need tools

Basic timing

How long did it take to run the program?

Sampling profilers

Where is my program spending most time? What line? Which machine instruction?

Hardware event counting

How many hardware events (e.g. cache misses) were *actually* triggered?

Emulators

Try to estimate hardware events by emulation instead.

Will try to cover some of the most common tools on Linux.

An example program: Jacobi iteration

emacs@na55.nada.kth.se

File Edit Options Buffers Tools C Help

```c
#include <stdlib.h>
#include <sys/time.h>

/* Elementary jacobi iteration */
double jacobi_iteration(int N,
                        double* restrict A[N],
                        double* restrict B[N],
                        double* restrict F[N])
{
    double diff = 0, tmp;
    double h = 1.0/(N+1);
    int i,j;

    for(i=1; i<N-1; i++)
        for(j=1; j<N-1; j++)
        {
            /* Jacobi update */
            B[i][j] = 0.25*(A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])-
                h*h*F[i][j];

            /* Compute diff. Contribution from boundary is zero */
            tmp = A[i][j]-B[i][j];
            diff += tmp*tmp;
        }
    return diff;
}
```

--:--- jacobi_base.c 6% L7 (C/l Abbrev)----------

<nil> <drag-mouse-1> is undefined

Basic timing

UNIX time

```
> time ./jacobi
Jacobi iteration converged in 263 iterations.
0.420u 0.000s 0:00.41 102.4% 0+0k 0+0io 0pf+0w
```

System time in C

```
#include <sys/time.h>

double gettime(void)
{
    struct timeval tv;
    gettimeofday(&tv,NULL);
    return tv.tv_sec + 1e-6*tv.tv_usec;
}

int main(void){
    double t = gettime();
    jacobi_solver(100);
    t = gettime()-t;
}
```

Sampling profilers

Works by periodically stopping the program and investigating the stack.

- GNU 'gprof'
- Intel VTune
- Valgrind 'callgrind' + KDE-based GUI kcachegrind

On Mac OS X, 'Shark' part of XCode (very good)

Sampling profiler: GNU gprof

Basic and reliable. Flat profile and call graph.

- Compile code with flag '-pg'
- Run program
- Run profiler, 'gprof ./jacobi'

```
Terminal
File  Edit  View  Terminal  Help
na55:talk >
na55:talk >
na55:talk > gcc -pg -g -std=c99 jacobi_base.c -o jacobi
na55:talk > ./jacobi
Jacobi iteration converged in 369 iterations.
na55:talk > gprof --flat-profile -l --brief ./jacobi
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 48.51      2.35     2.35                             jacobi_iteration (jacobi_base.c:22 @ 4008a4)
 25.08      3.56     1.21                             jacobi_iteration (jacobi_base.c:26 @ 4009f0)
 24.25      4.74     1.17                             jacobi_iteration (jacobi_base.c:25 @ 4009a5)
  1.87      4.83     0.09                             jacobi_iteration (jacobi_base.c:19 @ 400a08)
  0.62      4.86     0.03                             jacobi_iteration (jacobi_base.c:19 @ 400898)
  0.00      4.86     0.00      738     0.00     0.00  jacobi_iteration (jacobi_base.c:13 @ 400844)
  0.00      4.86     0.00        3     0.00     0.00  init_matrix (jacobi_base.c:33 @ 400a3d)
  0.00      4.86     0.00        2     0.00     0.00  init_boundary (jacobi_base.c:42 @ 400aa7)
na55:talk >
```

Recompile with -pg

Run program

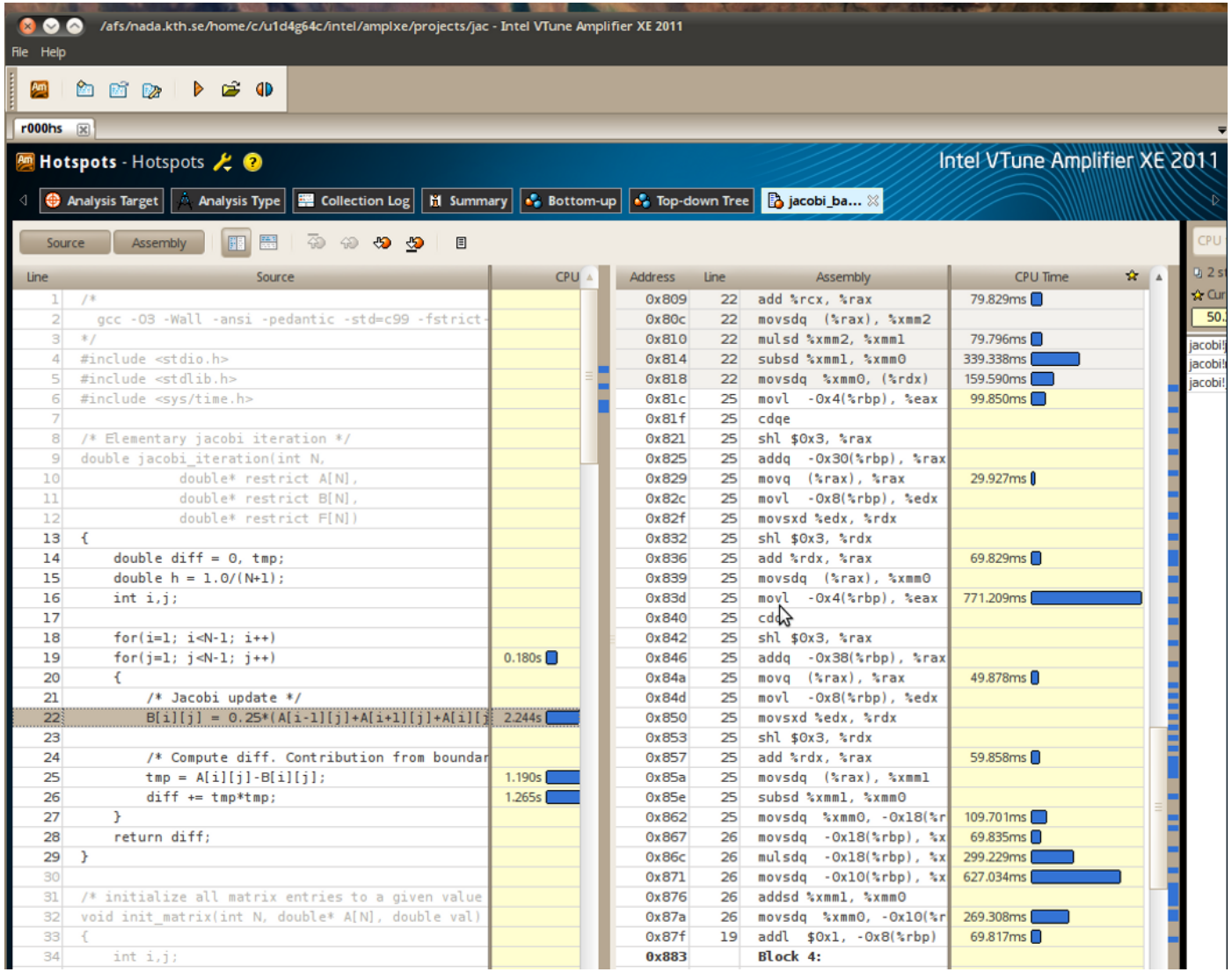
"Flat" profile

Function name

Line number

Sampling profiler: Intel VTune

Commercial, fancy GUI, sophisticated*, (marketing hype). Also some hardware events.
Free "non-commercial" download for Linux – so try it!



Hardware event counters

Registers used by the hardware manufacturers to debug and evaluate their designs are left and can be used for detailed profiling.

Papi

- Records actual number of hardware events that occurred!
- Free and open source: `http://icl.cs.utk.edu/~mucci/papiex/`
- By Phil Mucci (previously at PDC and gave this lecture)
- Not easy to install*: kernel modules, major dependency chain.
- What PAPI can record depends on hardware.

Papi front-ends

- PapiEx
- Cray PAT (though this is much more!)

* Mucci could do it :)

Hardware event counters

What can Papi determine?

```
> papi_avail
Available events and hardware information.

PAPI_L1_DCM   Level 1 data cache misses
PAPI_L2_DCM   Level 2 data cache misses
PAPI_TLB_DM   Data translation lookaside buffer misses
PAPI_HW_INT   Hardware interrupts
PAPI_BR_MSP   Conditional branch instructions mispredicted
PAPI_TOT_INS  Instructions completed
PAPI_FP_INS   Floating point instructions
PAPI_TOT_CYC  Total cycles

(...)
```

PapiEx basic usage

```
> papiex ./jacobi
(...)
Derived Metrics:
---------------
MFLOPS ............................          197.50
(...)
PAPI_TOT_CYC ......................      1.93957e+10
PAPI_FP_OPS .......................      1.44011e+09

PAPI_TOT_CYC                  : Total cycles
PAPI_FP_OPS                   : Floating point operations
```

Get specific counter:

```
> papiex -e PAPI_L1_DCM ./jacobi
(...)
L1 Data Cache Misses .........      1.17406e+09
```

Cray PAT

Very sophisticated framework for performance analysis (and MPI, OpenMP).
Commercial, available on high-end Cray systems.
∴ Not simple. Steps (see tutorial on lab):

- Select (PAPI) event group: `env PAT_RT_HWPC`
- Prepare the executable: `pat_build ./jacobi`
- Run: `aprun -n 1 ./jacobi+pat`
- View report: `pat_report jacobi+pat+<RUN ID>.xf` (next slide)

Hardware counter groups

```
> export PAT_RT_HWPC=

    1            Summary with TLB metrics
    2            L1 and L2 metrics
    3            Bandwidth information
    8            Instructions and branches
    12           Floating point operations (SSE)
```

Cray PAT Sample output

In counter group 1:

```
==========================================================================
   USER / jacobi_iteration / jacobi_base.c
--------------------------------------------------------------------------
  Samp%                                                  99.9%
  Samp                                                     772
  PAPI_L1_DCM                        2.621M/sec     32335186  misses
  PAPI_TLB_DM                        0.125M/sec      1541704  misses
  PAPI_L1_DCA                     1348.371M/sec  16633470668  refs
  PAPI_FP_OPS                     1726.230M/sec  21294737308  ops
  User time (approx)                12.336 secs  24671990403  cycles
  FLOPs                  1726.230M/sec  21294737308 ops   21.6%peak(DP)
  Computational intensity            0.86 ops/cycle    1.28  ops/ref
  MFLOPS (aggregate)               1726.23M/sec
  TLB utilization                 10789.02 refs/miss  21.072  avg uses
  D1 cache hit,miss ratios           99.8% hits         0.2%  misses
  D1 cache utilization (misses)    514.41 refs/miss  64.301  avg hits
```

Emulators

Try to extract same information as hardware counters record, but with modelling of the architecture.

- Valgrind 'cachegrind' + KDE-based GUI 'kcachegrind'
- Accumem ThreadSpotter (commercial)
  By Prof. Hagersten et. al. (now owned by Rogue Wave Software)

Results are only as accurate as the emulator. If the emulator has incorrect or incomplete parameters for the present architecture it will give a warning (and those warnings are important)

Emulator: Cachegrind

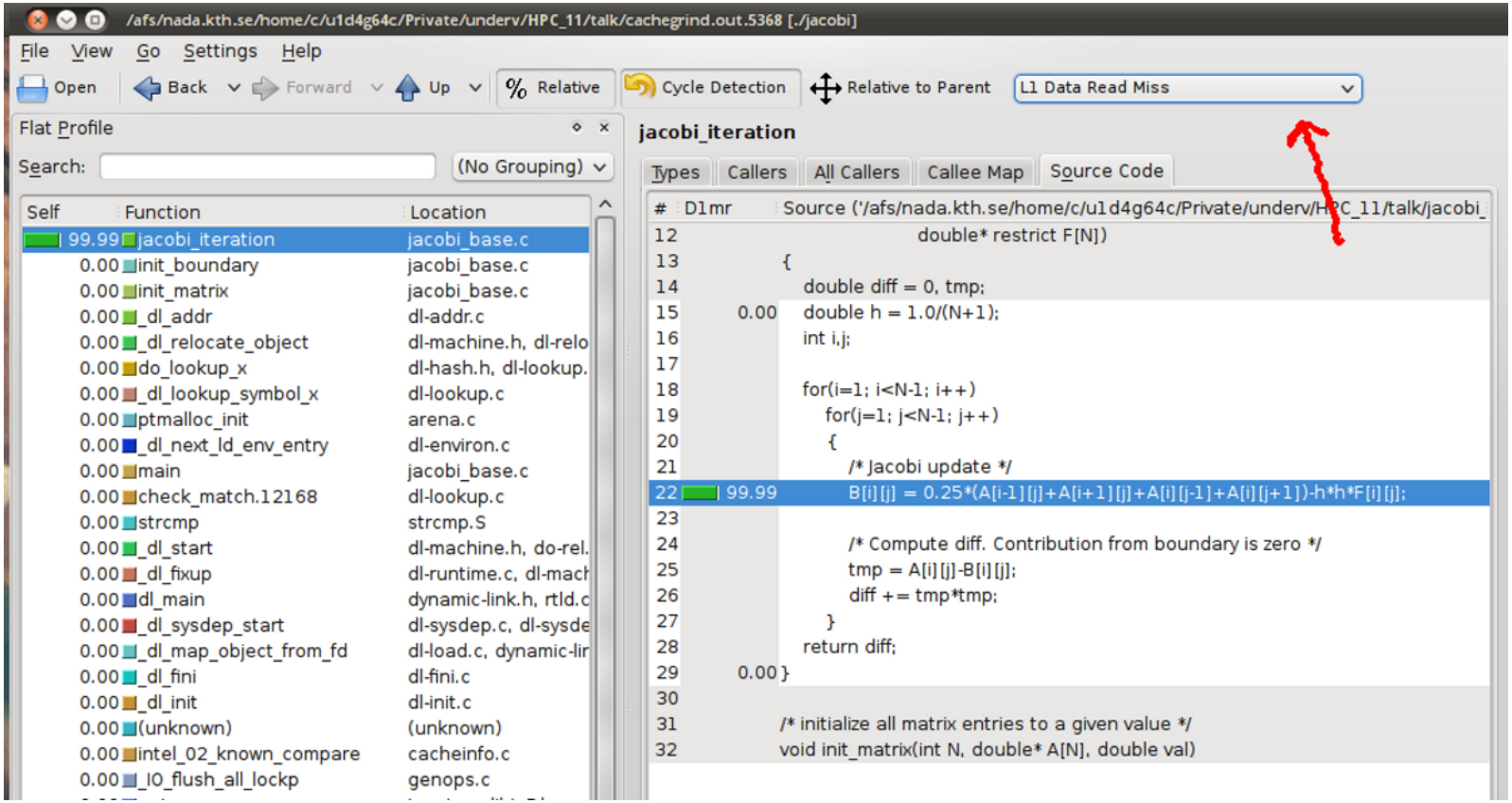
Part of the Valgrind tool-set that you of course already use for checking for memory leaks (right?).

```
Terminal
File  Edit  View  Terminal  Help
na55:talk > valgrind --tool=cachegrind ./jacobi
==5343== Cachegrind, a cache and branch-prediction profiler
==5343== Copyright (C) 2002-2009, and GNU GPL'd, by Nicholas Nethercote et al.
==5343== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info
==5343== Command: ./jacobi
==5343==


Jacobi iteration falied to converge.
==5343==
==5343== I   refs:      2,637,158,288
==5343== I1  misses:              806
==5343== L2i misses:              803
==5343== I1  miss rate:          0.00%
==5343== L2i miss rate:          0.00%
==5343==
==5343== D   refs:      1,282,217,568  (1,206,390,503 rd   + 75,827,065 wr)
==5343== D1  misses:        9,544,207  (    6,309,441 rd   +  3,234,766 wr)
==5343== L2d misses:        9,520,976  (    6,290,969 rd   +  3,230,007 wr)
==5343== D1  miss rate:           0.7% (          0.5%     +        4.2%  )
==5343== L2d miss rate:           0.7% (          0.5%     +        4.2%  )
==5343==
==5343== L2 refs:           9,545,013  (    6,310,247 rd   +  3,234,766 wr)
==5343== L2 misses:         9,521,779  (    6,291,772 rd   +  3,230,007 wr)
==5343== L2 miss rate:            0.2% (          0.1%     +        4.2%  )
na55:talk >
na55:talk >
na55:talk >
```

Note the argument: `valgrind --tool=cachegrind`

Emulator: Cachegrind GUI



KDE-based GUI `kcachegrind` available in most Linux distros. Similar tool in XCode on Mac OS X.

What did the compiler accomplish?

Example: did the compiler emit packed SSE memory transactions and arithmetic for the Jacobi inner loop?

OK, now it gets more technical!

Can generate disassembly from executable:

```
> objdump -d ./jacobi > jacobi_dump.asm
```

In this case, 134300 lines of assembly code! How do we determine where the inner loop is?

Use Valgrind (again):

```
> valgrind --tool=callgrind --dump-instr=yes --collect-jumps=yes
```

KCachegrind will now show us the loops:

Assembly-code annotation with Valgrind

./callgrind.out.5523 [./jacobi]

File View Go Settings Help

Open | Back | Forward | Up | % Relative | Cycle Detection | Relative to Parent | Instruction Fetch

main

Types | Callers | All Callers | Callee Map | Source Code

Event Type	Incl.	Self	Short	Formula
Instruction Fetch	100.00	98.47	Ir	

#	Ir	Hex		Assembly Instructions	Source Position
40 07B4	0.00	0f 57 db	xorps	%xmm3,%xmm3	jacobi_base.c:113
40 07B7	0.00	33 c0	xor	%eax,%eax	jacobi_base.c:113
40 07B9	0.00	4c 8b 5c c1 08	mov	0x8(%rcx,%rax,8),%r11	jacobi_base.c:88
40 07BE	0.00	45 33 f6	xor	%r14d,%r14d	jacobi_base.c:113
40 07C1	0.00	4c 8b 54 c6 08	mov	0x8(%rsi,%rax,8),%r10	jacobi_base.c:87
40 07C6	0.00	4c 8b 4c c6 10	mov	0x10(%rsi,%rax,8),%r9	jacobi_base.c:87
40 07CB	0.00	4c 8b 04 c6	mov	(%rsi,%rax,8),%r8	jacobi_base.c:87
40 07CF	0.00	48 8b 5c c7 08	mov	0x8(%rdi,%rax,8),%rbx	jacobi_base.c:86
40 07D4	0.00	0f 1f 44 00 00	nopl	0x0(%rax,%rax,1)	jacobi_base.c:86
40 07D9	0.00	0f 1f 80 00 00 00 00	nopl	0x0(%rax)	jacobi_base.c:86
40 07E0	3.34	f2 43 0f 10 6c f0 08	movsd	0x8(%r8,%r14,8),%xmm5	jacobi_base.c:87
40 07E7	3.34	f2 43 0f 58 6c f1 08	addsd	0x8(%r9,%r14,8),%xmm5	jacobi_base.c:113
40 07EE	3.34	f2 43 0f 58 2c f2	addsd	(%r10,%r14,8),%xmm5	jacobi_base.c:113
40 07F4	3.34	f2 43 0f 10 74 f2 08	movsd	0x8(%r10,%r14,8),%xmm6	jacobi_base.c:87
40 07FB	3.34	f2 43 0f 58 6c f2 10	addsd	0x10(%r10,%r14,8),%xmm5	jacobi_base.c:113
40 0802	3.34	f2 0f 59 e9	mulsd	%xmm1,%xmm5	jacobi_base.c:113
40 0806	3.34	f2 43 0f 10 64 f3 08	movsd	0x8(%r11,%r14,8),%xmm4	jacobi_base.c:88
40 080D	3.34	f2 0f 59 e0	mulsd	%xmm0,%xmm4	jacobi_base.c:113
40 0811	3.34	f2 0f 5c ec	subsd	%xmm4,%xmm5	jacobi_base.c:113
40 0815	3.34	f2 42 0f 11 6c f3 08	movsd	%xmm5,0x8(%rbx,%r14,8)	jacobi_base.c:86
40 081C	3.34	48 8b 5c c7 08	mov	0x8(%rdi,%rax,8),%rbx	jacobi_base.c:86
40 0821	3.34	f2 42 0f 5c 74 f3 08	subsd	0x8(%rbx,%r14,8),%xmm6	jacobi_base.c:113
40 0828	3.34	49 ff c6	inc	%r14	jacobi_base.c:113
40 082B	3.34	f2 0f 59 f6	mulsd	%xmm6,%xmm6	jacobi_base.c:113
40 082F	3.34	49 81 fe b8 0b 00 00	cmp	$0xbb8,%r14	jacobi_base.c:113
40 0836	3.34	f2 0f 58 de	addsd	%xmm6,%xmm3	jacobi_base.c:113
40 083A	3.34	72 a4	jb	4007e0 <main+0x3e0>	jacobi_base.c:113
				Jump 44 985 000 of 45 000 000 times to 0x4007E0	
40 083C	0.00	48 ff c0	inc	%rax	jacobi_base.c:113
40 083F	0.00	48 3d b8 0b 00 00	cmp	$0xbb8,%rax	jacobi_base.c:113
40 0845	0.00	0f 82 6e ff ff ff	jb	4007b9 <main+0x3b9>	jacobi_base.c:113
				Jump 14 995 of 15 000 times to 0x4007B9	
40 084B	0.00	44 8b 75 f0	mov	-0x10(%rbp),%r14d	jacobi_base.c:113
40 084F	0.00	48 8b 5d e8	mov	-0x18(%rbp),%rbx	jacobi_base.c:113

Assembly-code annotation with Valgrind

What do we learn from this?

```
f2 43 0f 10 6c f0 08    movsd   0x8(%r8,%r14,8),%xmm5
f2 43 0f 58 6c f1 08    addsd   0x8(%r9,%r14,8),%xmm5
f2 43 0f 58 2c f2       addsd   (%r10,%r14,8),%xmm5
f2 43 0f 10 74 f2 08    movsd   0x8(%r10,%r14,8),%xmm6
f2 43 0f 58 6c f2 10    addsd   0x10(%r10,%r14,8),%xmm5
f2 0f 59 e9             mulsd   %xmm1,%xmm5
f2 43 0f 10 64 f3 08    movsd   0x8(%r11,%r14,8),%xmm4
f2 0f 59 e0             mulsd   %xmm0,%xmm4
f2 0f 5c ec             subsd   %xmm4,%xmm5
f2 42 0f 11 6c f3 08    movsd   %xmm5,0x8(%rbx,%r14,8)
48 8b 5c c7 08          mov     0x8(%rdi,%rax,8),%rbx
f2 42 0f 5c 74 f3 08    subsd   0x8(%rbx,%r14,8),%xmm6
49 ff c6                inc     %r14
f2 0f 59 f6             mulsd   %xmm6,%xmm6
49 81 fe b8 0b 00 00    cmp     $0xbb8,%r14
f2 0f 58 de             addsd   %xmm6,%xmm3
72 a4                   jb      4007e0 <main+0x3e0>
```

Use "Intel Instruction Set Reference" (or Google)

- `'movsd'` is a single move of a double prec. number from memory to a SSE register (p. 3-718, vol. 2A)
- Corresponding vector instructions `movpd`,`mulpd`,`addpd` did not execute.

$\therefore$ The compiler failed to generate efficient SSE code. Do it yourself!?!