HELMHOLTZ
Helmholtz-Institut Mainz

Code Optimization

Dalibor Djukanovic

Generalities

This is an overview talk.
But technical.

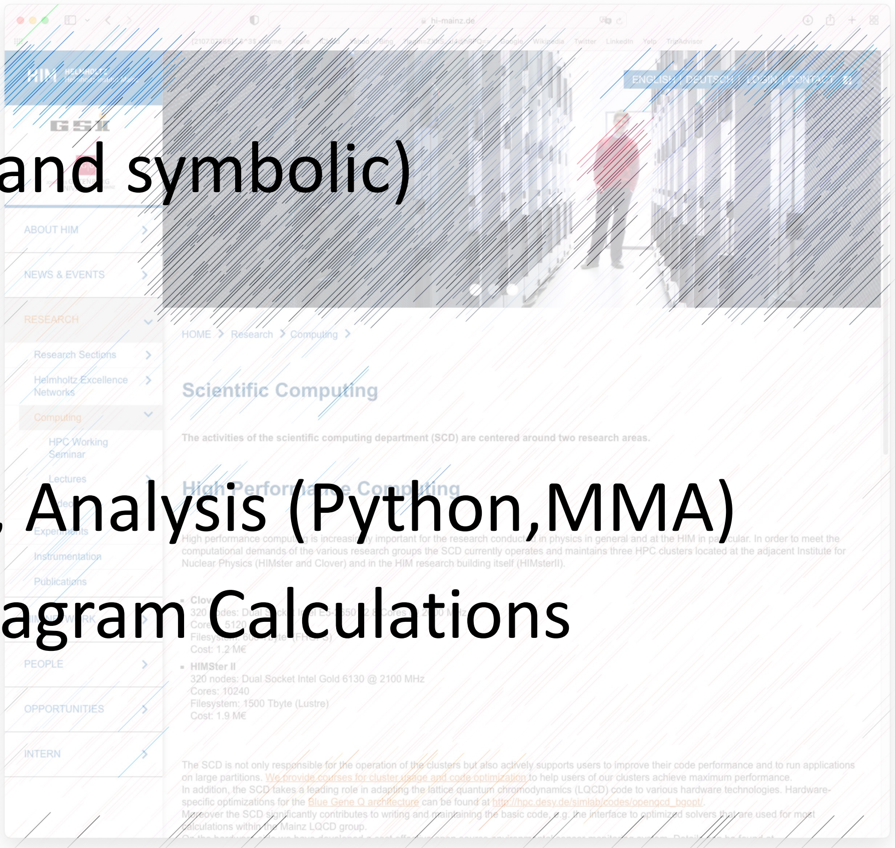
If you have specific questions/comments concerning code performance issues contact me

djukanov@uni-mainz.de

Reminder:
Fortnightly HPC seminar

My Background

- Member of the Nuclear Theory Group
- Doing Calculations in:
  - Lattice QCD numerical (and symbolic)
  - EFT (mostly symbolic)
- Optimization interests:
  - Numerical: MonteCarlo, Analysis (Python,MMA)
  - Symbolical: Feynman Diagram Calculations (FORM,MMA)
  - Get results fast
- Not a performance architect:
  - Some of the thing I say might not even be wrong!



Scientific Computing

The activities of the scientific computing department (SCD) are centered around two research areas.

High Performance Computing

High performance computing is increasingly important for the research conducted in physics in general and at the HIM in particular. In order to meet the computational demands of the various research groups the SCD currently operates and maintains three HPC clusters located at the adjacent Institute for Nuclear Physics (HIMster and Clover) and in the HIM research building itself (HIMsterII).

- Clover
  320 nodes: Dual Socket Intel E5-2650 2 8 Cores @ 2000 MHz
  Cores: 5120
  Filesystem: 600 Tbyte (FHGFS)
  Cost: 1.2 M€
- HIMSter II
  320 nodes: Dual Socket Intel Gold 6130 @ 2100 MHz
  Cores: 10240
  Filesystem: 1500 Tbyte (Lustre)
  Cost: 1.9 M€

The SCD is not only responsible for the operation of the clusters but also actively supports users to improve their code performance and to run applications on large partitions. We provide courses for cluster usage and code optimization to help users of our clusters achieve maximum performance.
In addition, the SCD takes a leading role in adapting the lattice quantum chromodynamics (LQCD) code to various hardware technologies. Hardware-specific optimizations for the Blue Gene Q architecture can be found at http://hpc.desy.de/simlab/codes/openqcd_bgopt/.
Moreover the SCD significantly contributes to writing and maintaining the basic code, e.g. the interface to optimized solvers that are used for most calculations within the Mainz LQCD group.

Contents - I

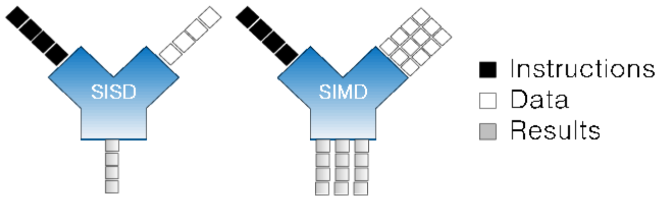
- Performance = Divide & Conquer
- Core Level Optimization:
  SIMD - Domain
  Optimize Code to use Oncore vector units
- Node Level Optimization:
  SIMT – Domain
  Optimize Code to use threads working in parallel on data
- Application Level Optimization:
  Algorithmic Improvements, Avoiding Data Transfer, Hiding Latencies

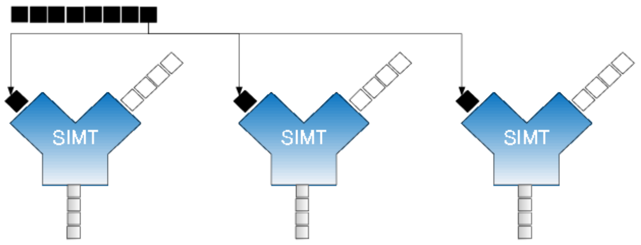
Contents - II

- I will try to cover the following aspects:
  - How can you get the most out of a modern CPU
  - How do you know where your code spends most of the time
- Takeaways:
  - Be modest. 20x improvement will (probably) not happen.
  - Try to understand your codes performance ceiling and the bottlenecks.
  - Get used to tools (profilers, debuggers, …).

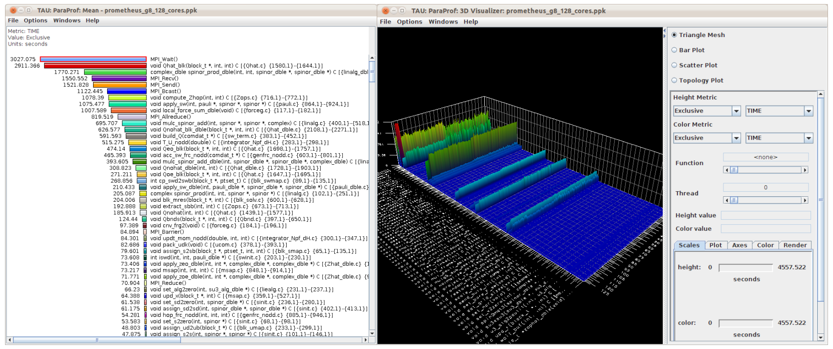
Outline

- Performance Paradigms

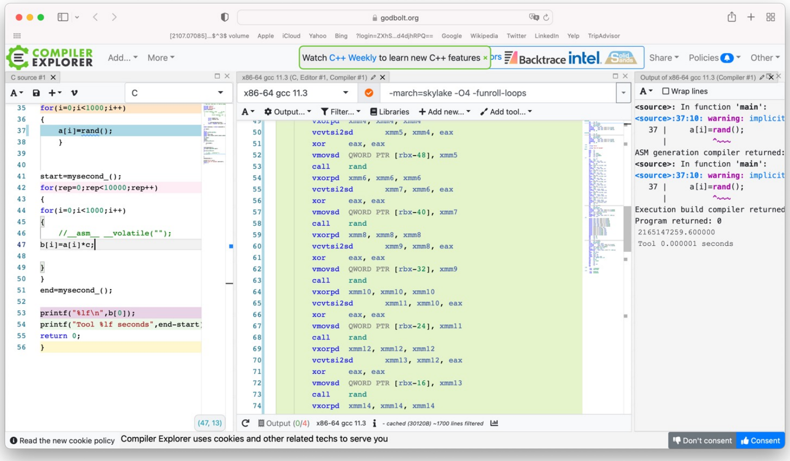




- Profiling



- Assembler Coding



```
for(i=0;i<1000;i++)
{
    a[i]=rand();
    }


start=mysecond_();
for(rep=0;rep<10000;rep++)
{
for(i=0;i<1000;i++)
{
    //__asm__ __volatile("");
b[i]=a[i]*c;

}
}
end=mysecond_();

printf("%lf\n",b[0]);
printf("Tool %lf seconds",end-start
return 0;
}
```

x86-64 gcc 11.3 -march=skylake -O4 -funroll-loops

```
        vcvtsi2sd       xmm5, xmm4, eax
        xor     eax, eax
        vmovsd  QWORD PTR [rbx-48], xmm5
        call    rand
        vxorpd  xmm6, xmm6, xmm6
        vcvtsi2sd       xmm7, xmm6, eax
        xor     eax, eax
        vmovsd  QWORD PTR [rbx-40], xmm7
        call    rand
        vxorpd  xmm8, xmm8, xmm8
        vcvtsi2sd       xmm9, xmm8, eax
        xor     eax, eax
        vmovsd  QWORD PTR [rbx-32], xmm9
        call    rand
        vxorpd  xmm10, xmm10, xmm10
        vcvtsi2sd       xmm11, xmm10, eax
        xor     eax, eax
        vmovsd  QWORD PTR [rbx-24], xmm11
        call    rand
        vxorpd  xmm12, xmm12, xmm12
        vcvtsi2sd       xmm13, xmm12, eax
        xor     eax, eax
        vmovsd  QWORD PTR [rbx-16], xmm13
        call    rand
        vxorpd  xmm14, xmm14, xmm14
```

```
<source>: In function 'main':
<source>:37:10: warning: implicit
   37 |     a[i]=rand();
      |          ^~~~
ASM generation compiler returned:
<source>: In function 'main':
<source>:37:10: warning: implicit
   37 |     a[i]=rand();
      |          ^~~~
Execution build compiler returned
Program returned: 0
 2165147259.600000
 Tool 0.000001 seconds
```

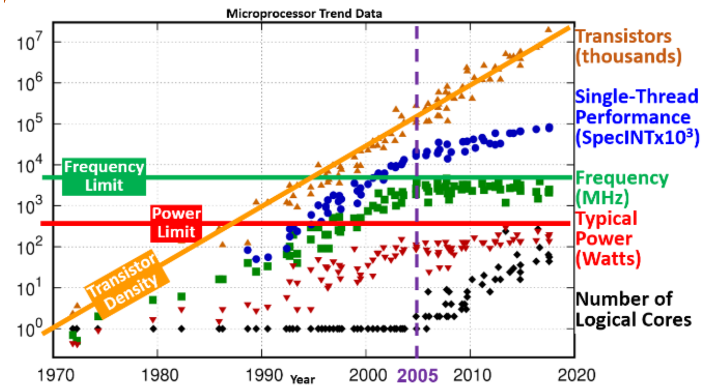
(47, 13) Output (0/4) x86-64 gcc 11.3 - cached (30120B) ~1700 lines filtered

Read the new cookie policy Compiler Explorer uses cookies and other related techs to serve you Don't consent Consent

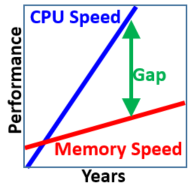
HIM HELMHOLTZ Helmholtz-Institut Mainz

History of the CPU

- Moore

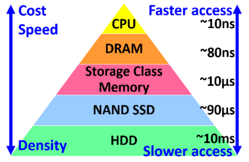


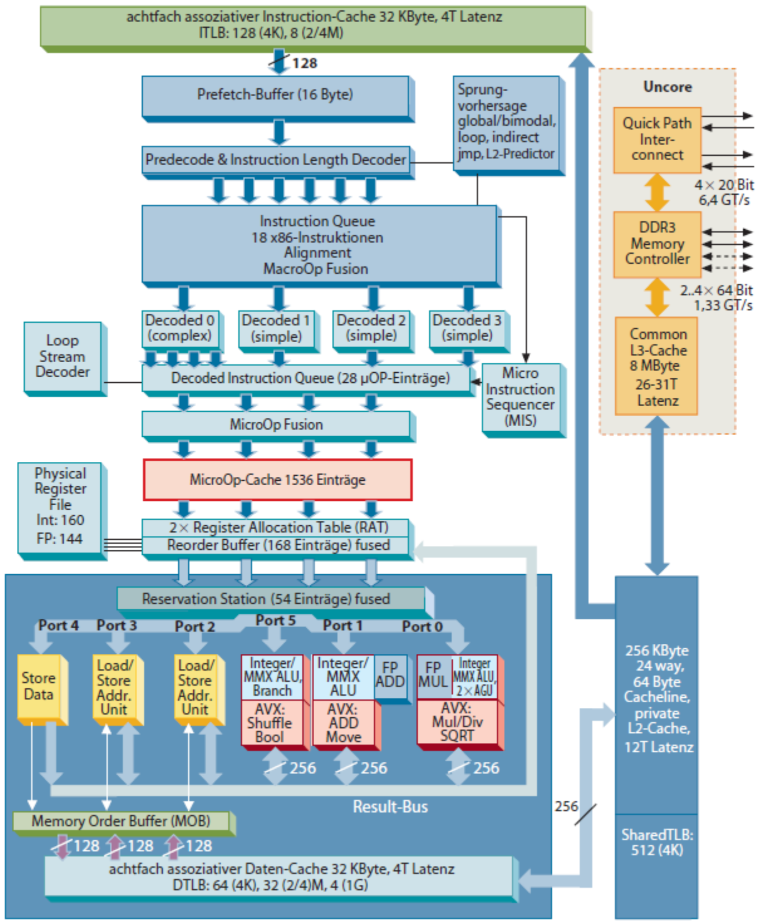
- CPUs went multicore/vector
- RAM does not keep up



(von Neumann-Bottleneck)

- Latency Hierarchies

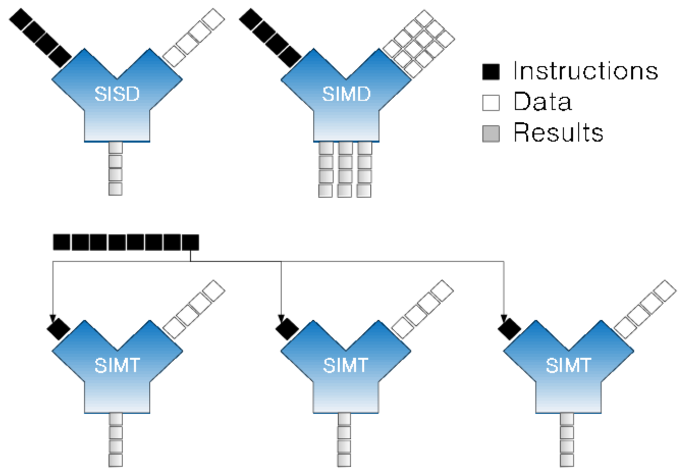




achtfach assoziativer Instruction-Cache 32 KByte, 4T Latenz
ITLB: 128 (4K), 8 (2/4M)
128
Prefetch-Buffer (16 Byte)
Sprung-vorhersage global/bimodal, loop, indirect jmp, L2-Predictor
Predecode & Instruction Length Decoder
Instruction Queue
18 x86-Instruktionen
Alignment
MacroOp Fusion
Loop Stream Decoder
Decoded 0 (complex)
Decoded 1 (simple)
Decoded 2 (simple)
Decoded 3 (simple)
Decoded Instruction Queue (28 µOP-Einträge)
Micro Instruction Sequencer (MIS)
MicroOp Fusion
MicroOp-Cache 1536 Einträge
Physical Register File
Int: 160
FP: 144
2× Register Allocation Table (RAT)
Reorder Buffer (168 Einträge) fused
Reservation Station (54 Einträge) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store Data
Load/ Store Addr. Unit
Load/ Store Addr. Unit
Integer/ MMX ALU, Branch
AVX: Shuffle Bool
Integer/ MMX ALU
FP ADD
AVX: ADD Move
FP MUL
Integer MMX ALU, 2× AGU
AVX: Mul/Div SQRT
256
256
256
Result-Bus
Memory Order Buffer (MOB)
128
128
128
achtfach assoziativer Daten-Cache 32 KByte, 4T Latenz
DTLB: 64 (4K), 32 (2/4M), 4 (1G)
256
Uncore
Quick Path Inter-connect
4× 20 Bit 6,4 GT/s
DDR3 Memory Controller
2..4× 64 Bit 1,33 GT/s
Common L3-Cache 8 MByte 26-31T Latenz
256 KByte 24 way, 64 Byte Cacheline, private L2-Cache, 12T Latenz
SharedTLB: 512 (4K)

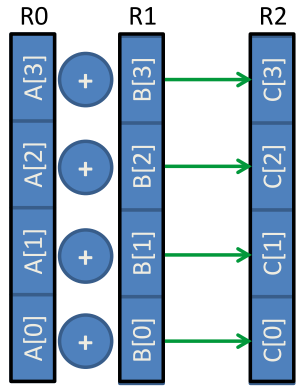
Many Core

- Performance gains through parallelism
  - Thread Level:
    - More cores running tasks in parallel
  - Core Level:
    - Execution units are vector machines
- Having one set of instructions and data we distinguish
  - SIMT: Single Instruction Multiple Threads
  - SIMD: Single Instruction Multiple Data

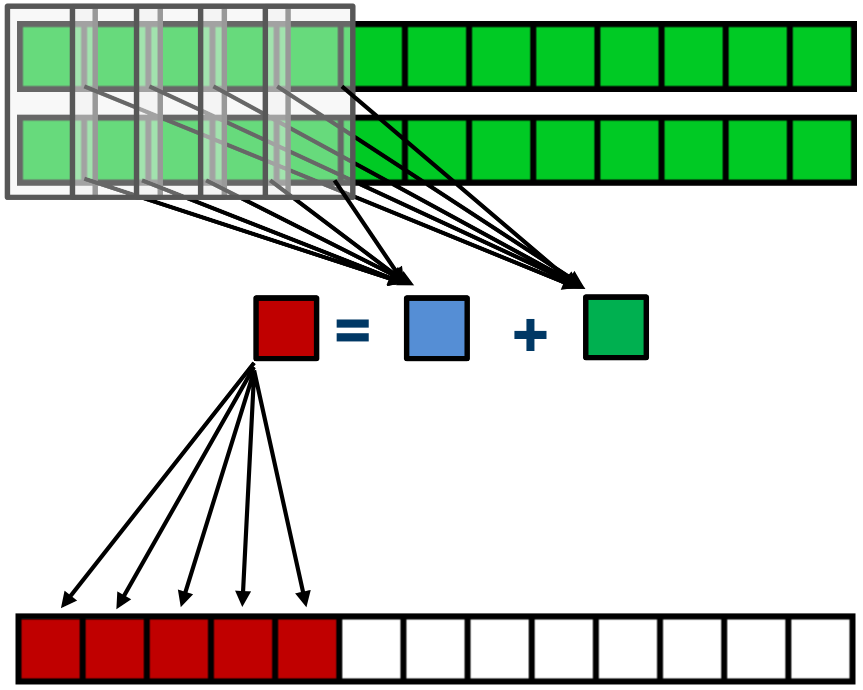


SIMD

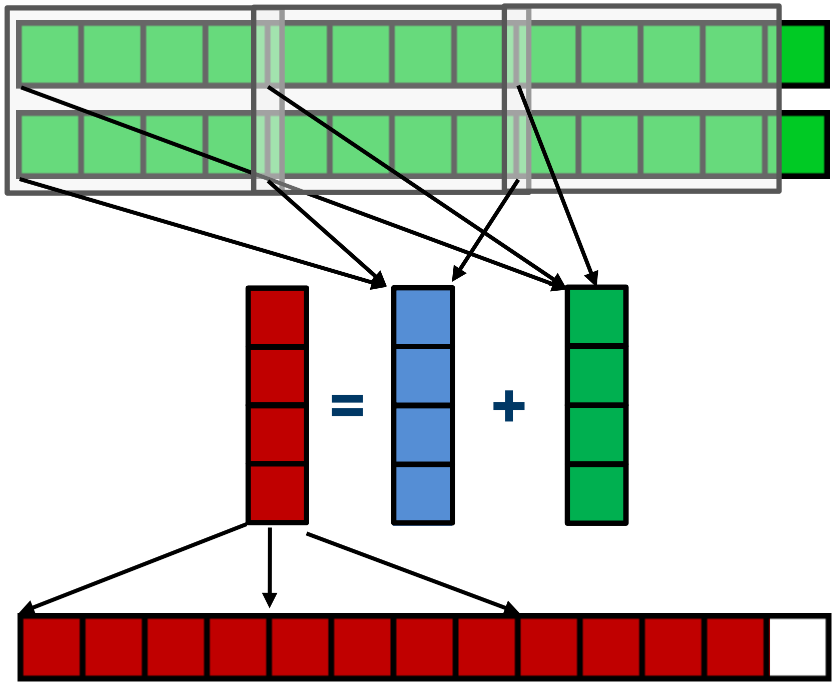
- Vector Execution of Scalar Code
- Works on registers



Scalar execution



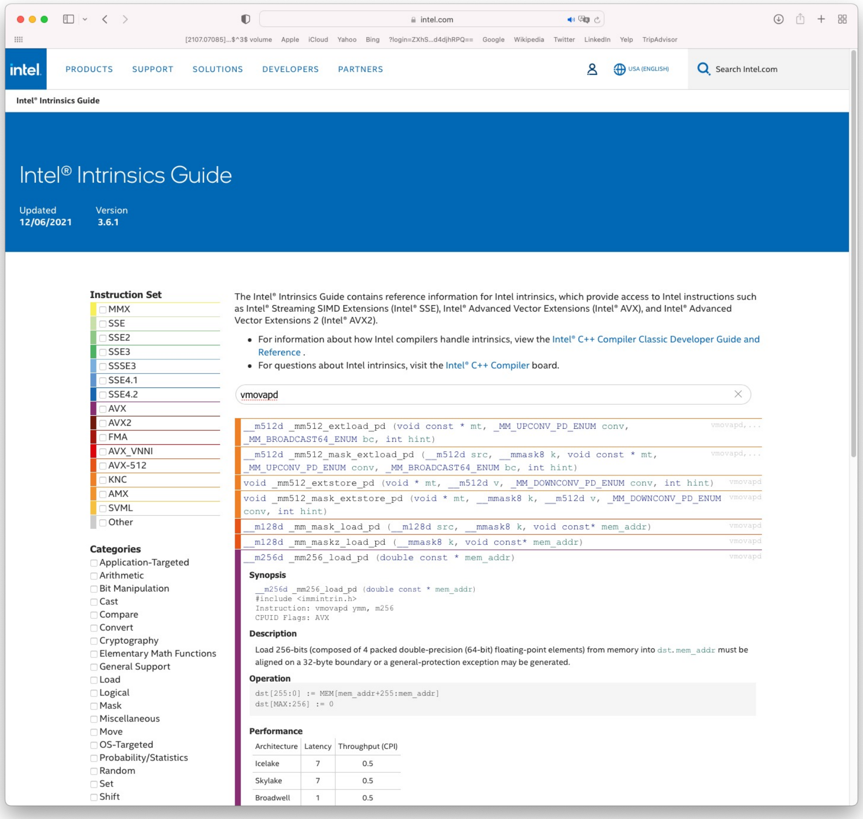
SIMD execution



SIMD

- Width of registers depend on CPU
  - 128 Bit SSE
  - 256 Bit AVX
  - 512 Bit AVX512
- Instruction set also depends on CPU
- Theoretical Peak (Himster2)

  Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz
- 16 (cores) x 8 (doubles in regs) x 2 (fma) x 2 (fma units) x 2.1 GHz = 1000 Gflops/s
- 2 Tflops per node


intel.com
[2107.07085]...^{3} volume Apple iCloud Yahoo Bing ?login=ZXhS...d4djhRPQ== Google Wikipedia Twitter LinkedIn Yelp TripAdvisor
intel.
PRODUCTS SUPPORT SOLUTIONS DEVELOPERS PARTNERS
USA (ENGLISH)
Search Intel.com

Intel® Intrinsics Guide

Intel® Intrinsics Guide

Updated 12/06/2021 Version 3.6.1

Instruction Set

- MMX
- SSE
- SSE2
- SSE3
- SSSE3
- SSE4.1
- SSE4.2
- AVX
- AVX2
- FMA
- AVX_VNNI
- AVX-512
- KNC
- AMX
- SVML
- Other

Categories

- Application-Targeted
- Arithmetic
- Bit Manipulation
- Cast
- Compare
- Convert
- Cryptography
- Elementary Math Functions
- General Support
- Load
- Logical
- Mask
- Miscellaneous
- Move
- OS-Targeted
- Probability/Statistics
- Random
- Set
- Shift

The Intel® Intrinsics Guide contains reference information for Intel intrinsics, which provide access to Intel instructions such as Intel® Streaming SIMD Extensions (Intel® SSE), Intel® Advanced Vector Extensions (Intel® AVX), and Intel® Advanced Vector Extensions 2 (Intel® AVX2).

- For information about how Intel compilers handle intrinsics, view the Intel® C++ Compiler Classic Developer Guide and Reference .
- For questions about Intel intrinsics, visit the Intel® C++ Compiler board.

vmovapd

```
__m512d _mm512_extload_pd (void const * mt, _MM_UPCONV_PD_ENUM conv,
_MM_BROADCAST64_ENUM bc, int hint)                                    vmovapd, ...
__m512d _mm512_mask_extload_pd (__m512d src, __mmask8 k, void const * mt,
_MM_UPCONV_PD_ENUM conv, _MM_BROADCAST64_ENUM bc, int hint)           vmovapd, ...
void _mm512_extstore_pd (void * mt, __m512d v, _MM_DOWNCONV_PD_ENUM conv, int hint)   vmovapd
void _mm512_mask_extstore_pd (void * mt, __mmask8 k, __m512d v, _MM_DOWNCONV_PD_ENUM
conv, int hint)                                                       vmovapd
__m128d _mm_mask_load_pd (__m128d src, __mmask8 k, void const* mem_addr)   vmovapd
__m128d _mm_maskz_load_pd (__mmask8 k, void const* mem_addr)          vmovapd
__m256d _mm256_load_pd (double const * mem_addr)                      vmovapd
```

Synopsis

```
__m256d _mm256_load_pd (double const * mem_addr)
#include <immintrin.h>
Instruction: vmovapd ymm, m256
CPUID Flags: AVX
```

Description

Load 256-bits (composed of 4 packed double-precision (64-bit) floating-point elements) from memory into dst. mem_addr must be aligned on a 32-byte boundary or a general-protection exception may be generated.

Operation

```
dst[255:0] := MEM[mem_addr+255:mem_addr]
dst[MAX:256] := 0
```

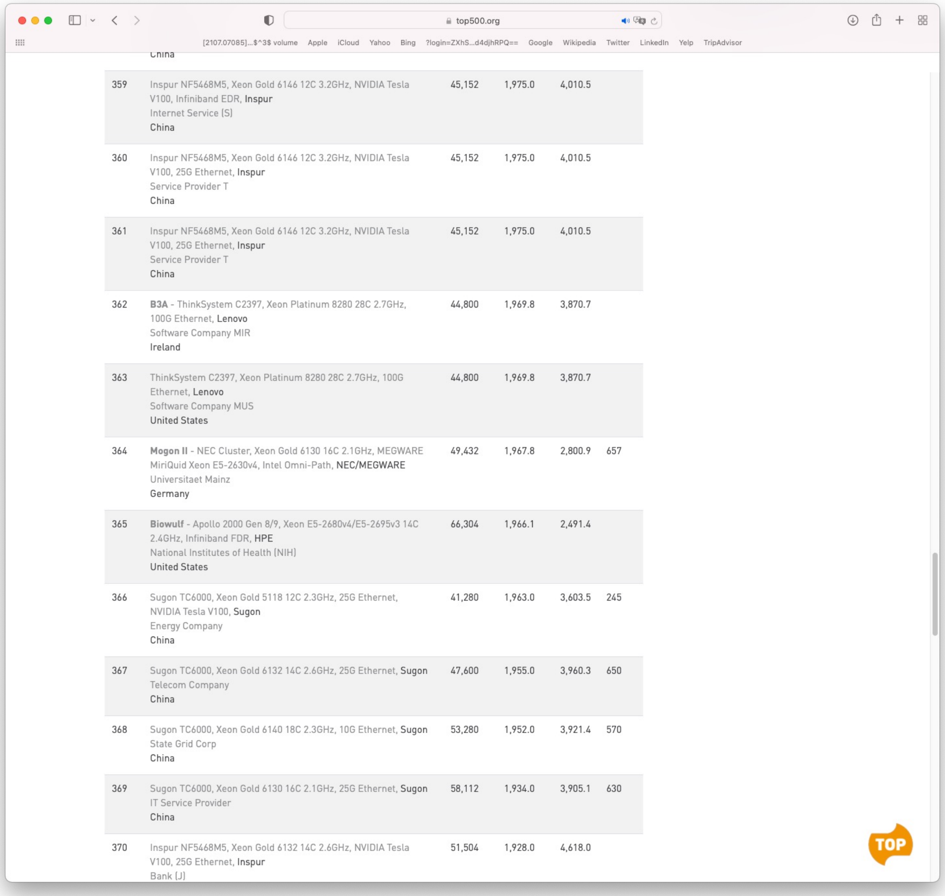
Performance

Architecture	Latency	Throughput (CPI)
Icelake	7	0.5
Skylake	7	0.5
Broadwell	1	0.5

HIM HELMHOLTZ Helmholtz-Institut Mainz

What to Expect?

- MogonII/HIMster2 still in Top500 List
- Pos 364
- R_{theo} = 2800 Tflops
- $R_{linpack}$ = 1967 Tflops
- @ 657 kWatt
- 70 % of peak looks promising
- Linpack not very common work load



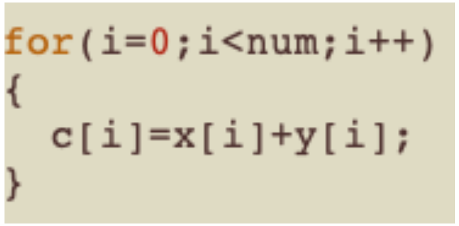
top500.org

	China				
359	Inspur NF5468M5, Xeon Gold 6146 12C 3.2GHz, NVIDIA Tesla V100, Infiniband EDR, Inspur Internet Service (S) China	45,152	1,975.0	4,010.5	
360	Inspur NF5468M5, Xeon Gold 6146 12C 3.2GHz, NVIDIA Tesla V100, 25G Ethernet, Inspur Service Provider T China	45,152	1,975.0	4,010.5	
361	Inspur NF5468M5, Xeon Gold 6146 12C 3.2GHz, NVIDIA Tesla V100, 25G Ethernet, Inspur Service Provider T China	45,152	1,975.0	4,010.5	
362	**B3A** - ThinkSystem C2397, Xeon Platinum 8280 28C 2.7GHz, 100G Ethernet, Lenovo Software Company MIR Ireland	44,800	1,969.8	3,870.7	
363	ThinkSystem C2397, Xeon Platinum 8280 28C 2.7GHz, 100G Ethernet, Lenovo Software Company MUS United States	44,800	1,969.8	3,870.7	
364	**Mogon II** - NEC Cluster, Xeon Gold 6130 16C 2.1GHz, MEGWARE MiriQuid Xeon E5-2630v4, Intel Omni-Path, NEC/MEGWARE Universitaet Mainz Germany	49,432	1,967.8	2,800.9	657
365	**Biowulf** - Apollo 2000 Gen 8/9, Xeon E5-2680v4/E5-2695v3 14C 2.4GHz, Infiniband FDR, HPE National Institutes of Health (NIH) United States	66,304	1,966.1	2,491.4	
366	Sugon TC6000, Xeon Gold 5118 12C 2.3GHz, 25G Ethernet, NVIDIA Tesla V100, Sugon Energy Company China	41,280	1,963.0	3,603.5	245
367	Sugon TC6000, Xeon Gold 6132 14C 2.6GHz, 25G Ethernet, Sugon Telecom Company China	47,600	1,955.0	3,960.3	650
368	Sugon TC6000, Xeon Gold 6140 18C 2.3GHz, 10G Ethernet, Sugon State Grid Corp China	53,280	1,952.0	3,921.4	570
369	Sugon TC6000, Xeon Gold 6130 16C 2.1GHz, 25G Ethernet, Sugon IT Service Provider China	58,112	1,934.0	3,905.1	630
370	Inspur NF5468M5, Xeon Gold 6132 14C 2.6GHz, NVIDIA Tesla V100, 25G Ethernet, Inspur Bank (J)	51,504	1,928.0	4,618.0	

HIM HELMHOLTZ Helmholtz-Institut Mainz

Performance Models

- Roofline Model
  - Assume perfect overlap of data transfer and ops
  - Latency is ignored
  - Count amount of „work“ against „data"



```
for(i=0;i<num;i++)
{
  c[i]=x[i]+y[i];
}
```

Data: 2 x load + 1 store = 24 byte (30)

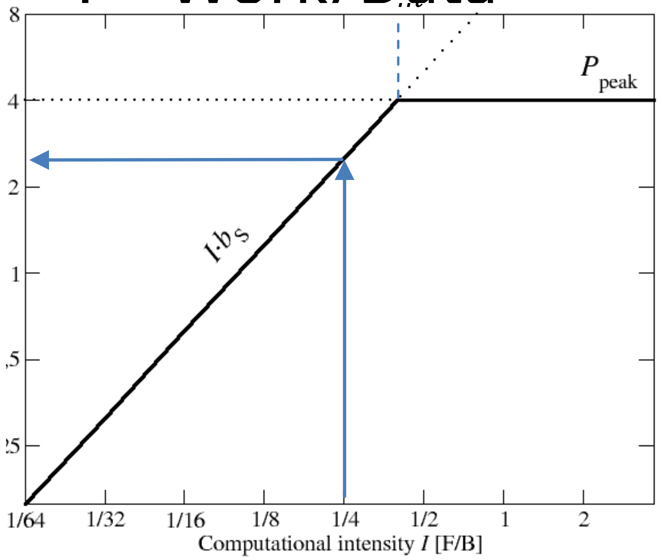
Work: 1 x add = 1 Flop

Intensity: Work/Data = 1 Flop/24 B

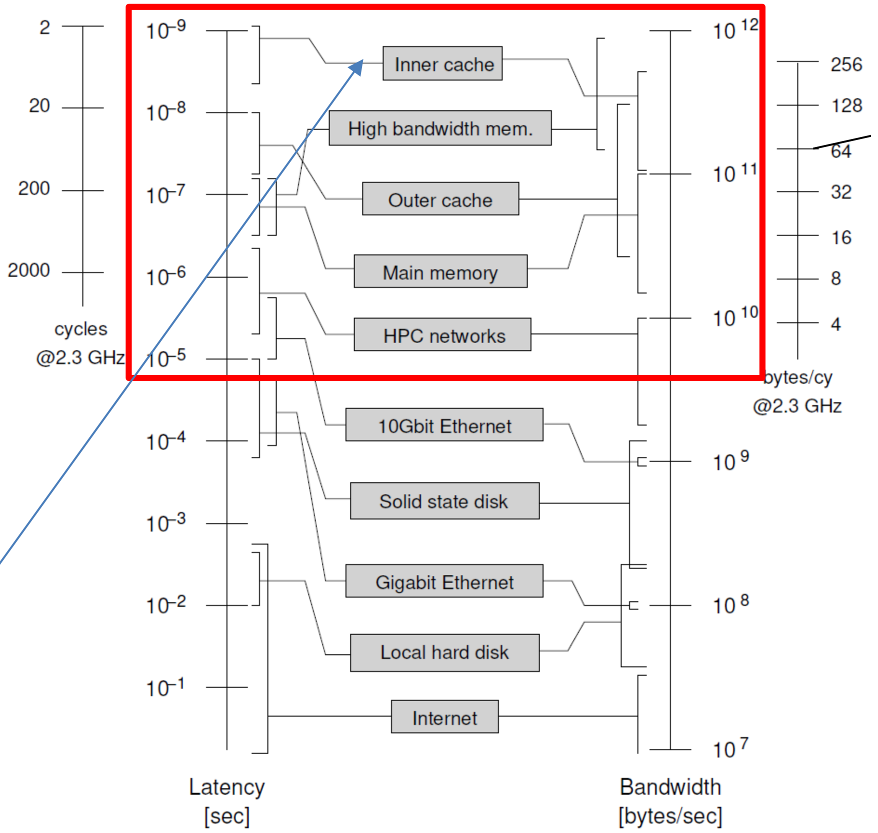
  - Roofline Model:
    - Take minimum of applicable Peak perf vs BW of slowest data path (suppose 190 Gbyte/s)
    - $P_{roofline}$= min(2 x 1075.2 Gflops, 1/24 x 190 Gflops)
      = 8 Gflops < 1 % peak

Roofline Model

- Ways to improve:
  - Increase Intensity
    Remember
    I = Work/Data



  - Avoid slow paths
    Keep data in caches



Roofline Model

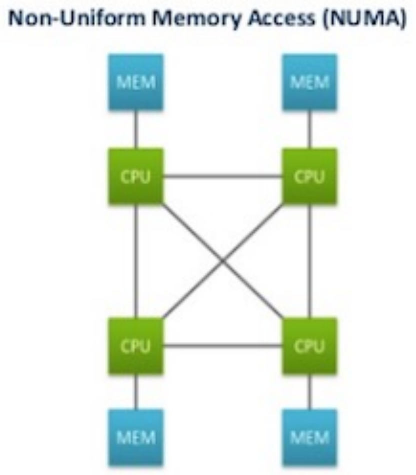
- Simple model tells you „speed of light“ for your kernel
- Refinements Possible:
  - Take into account cache hierarchy
  - Take into account latencies
  - Take into account pipeline design of CPUs

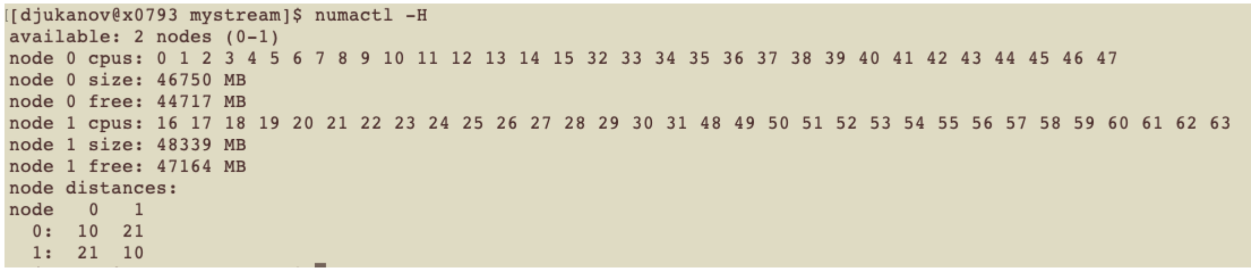
KERNCRAFT

- Usually I stop here at the simple model and try to get to „speed of light“ according to Roofline

Memory Alignment - Interlude

- **<u>Usually</u>** performance problem may be attributed to bad memory layout
- Ideal is stride-1 access pattern
- NUMA
  - – Avoid “foreign memory“
  - – Process pinning important
  - – Topic for HPC-seminar?

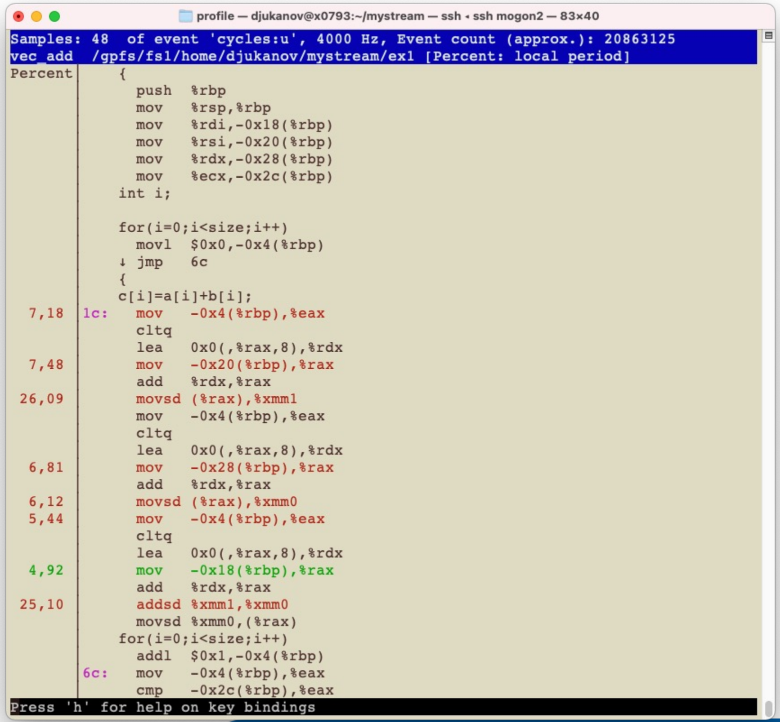




```
[djukanov@x0793 mystream]$ numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
node 0 size: 46750 MB
node 0 free: 44717 MB
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
node 1 size: 48339 MB
node 1 free: 47164 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

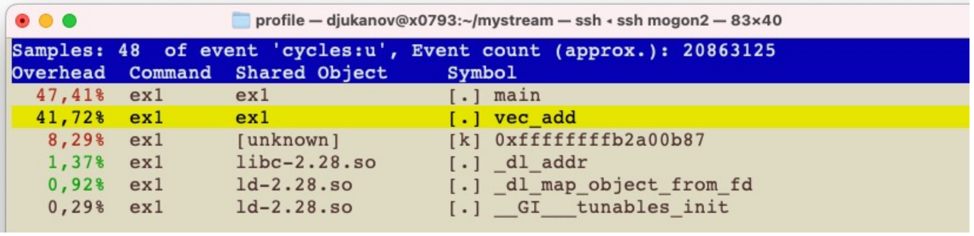
Profiling

- I want to SIMD my code, where do I start?
  - Profile code
    - Manually using timings
    - Automatically using some tool
      - Instrumentation
        - » Gprof, TAU …
      - Sampling
        - » Intel Vtune, perf …



profile — djukanov@x0793:~/mystream — ssh ◂ ssh mogon2 — 83×40

```
Samples: 48  of event 'cycles:u', 4000 Hz, Event count (approx.): 20863125
vec_add  /gpfs/fs1/home/djukanov/mystream/ex1 [Percent: local period]
Percent│      {
       │        push   %rbp
       │        mov    %rsp,%rbp
       │        mov    %rdi,-0x18(%rbp)
       │        mov    %rsi,-0x20(%rbp)
       │        mov    %rdx,-0x28(%rbp)
       │        mov    %ecx,-0x2c(%rbp)
       │      int i;
       │
       │      for(i=0;i<size;i++)
       │        movl   $0x0,-0x4(%rbp)
       │      ↓ jmp    6c
       │      {
       │      c[i]=a[i]+b[i];
  7,18 │1c:     mov    -0x4(%rbp),%eax
       │        cltq
       │        lea    0x0(,%rax,8),%rdx
  7,48 │        mov    -0x20(%rbp),%rax
       │        add    %rdx,%rax
 26,09 │        movsd  (%rax),%xmm1
       │        mov    -0x4(%rbp),%eax
       │        cltq
       │        lea    0x0(,%rax,8),%rdx
  6,81 │        mov    -0x28(%rbp),%rax
       │        add    %rdx,%rax
  6,12 │        movsd  (%rax),%xmm0
  5,44 │        mov    -0x4(%rbp),%eax
       │        cltq
       │        lea    0x0(,%rax,8),%rdx
  4,92 │        mov    -0x18(%rbp),%rax
       │        add    %rdx,%rax
 25,10 │        addsd  %xmm1,%xmm0
       │        movsd  %xmm0,(%rax)
       │      for(i=0;i<size;i++)
       │        addl   $0x1,-0x4(%rbp)
       │6c:     mov    -0x4(%rbp),%eax
       │        cmp    -0x2c(%rbp),%eax
Press 'h' for help on key bindings
```



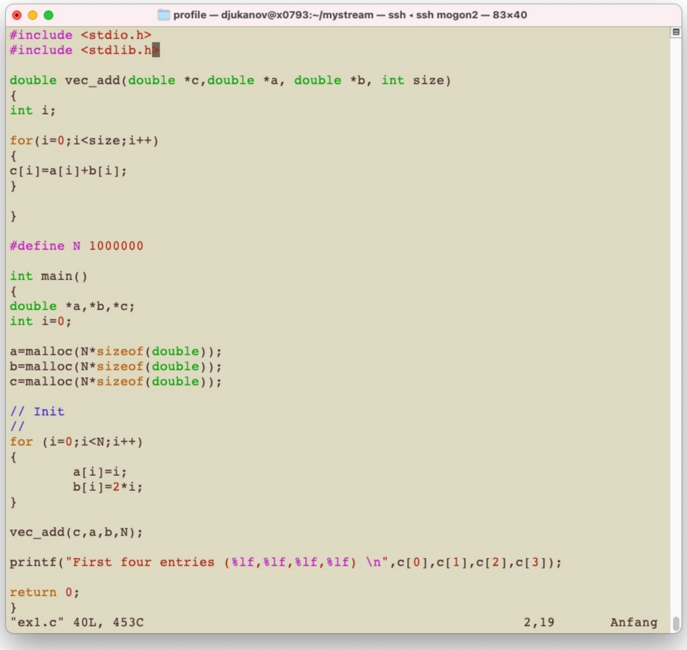
profile — djukanov@x0793:~/mystream — ssh ◂ ssh mogon2 — 83×40

```
Samples: 48  of event 'cycles:u', Event count (approx.): 20863125
Overhead  Command  Shared Object       Symbol
  47,41%  ex1      ex1                 [.] main
  41,72%  ex1      ex1                 [.] vec_add
   8,29%  ex1      [unknown]           [k] 0xffffffffb2a00b87
   1,37%  ex1      libc-2.28.so        [.] _dl_addr
   0,92%  ex1      ld-2.28.so          [.] _dl_map_object_from_fd
   0,29%  ex1      ld-2.28.so          [.] __GI___tunables_init
```

HELMHOLTZ Helmholtz-Institut Mainz

Perf

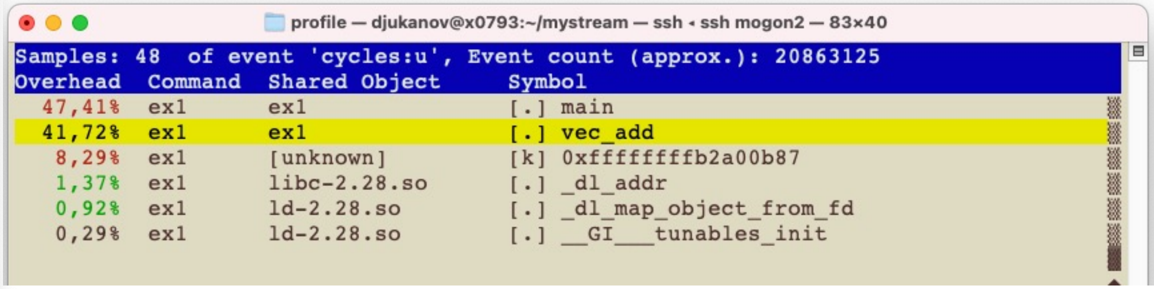
- Perf is a profiling tool already installed on most Linux systems
- No need to recompile binary
- May attach to running processes



profile — djukanov@x0793:~/mystream — ssh ‹ ssh mogon2 — 83×40

```c
#include <stdio.h>
#include <stdlib.h>

double vec_add(double *c,double *a, double *b, int size)
{
int i;

for(i=0;i<size;i++)
{
c[i]=a[i]+b[i];
}

}

#define N 1000000

int main()
{
double *a,*b,*c;
int i=0;

a=malloc(N*sizeof(double));
b=malloc(N*sizeof(double));
c=malloc(N*sizeof(double));

// Init
//
for (i=0;i<N;i++)
{
        a[i]=i;
        b[i]=2*i;
}

vec_add(c,a,b,N);

printf("First four entries (%lf,%lf,%lf,%lf) \n",c[0],c[1],c[2],c[3]);

return 0;
}
"ex1.c" 40L, 453C                                              2,19          Anfang
```

```
gcc -g <source> -o <bin>
perf record ./<bin>
perf report
```

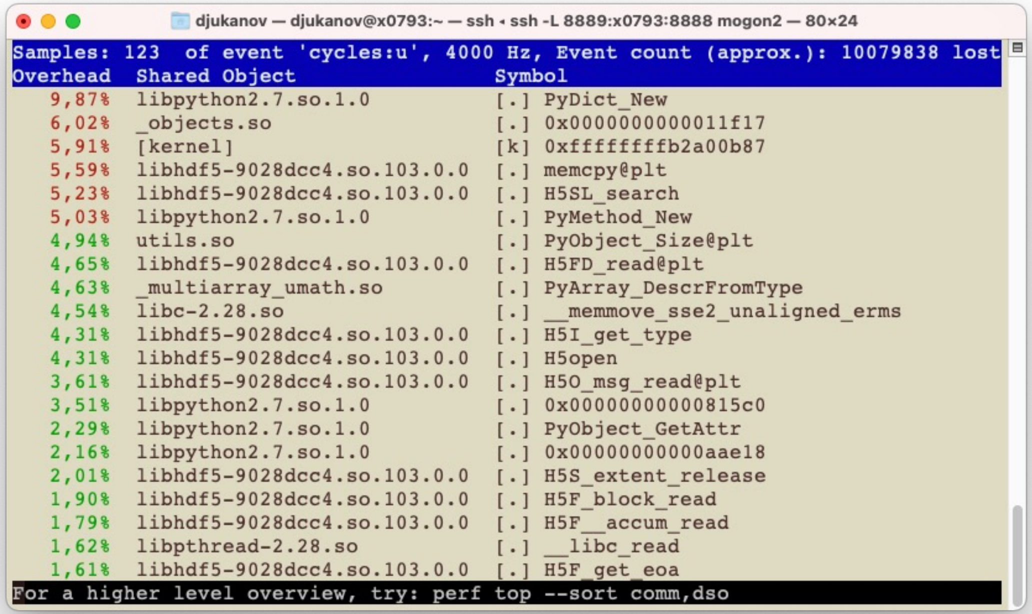


profile — djukanov@x0793:~/mystream — ssh ‹ ssh mogon2 — 83×40

```
Samples: 48  of event 'cycles:u', Event count (approx.): 20863125
Overhead  Command  Shared Object       Symbol
  47,41%  ex1      ex1                 [.] main
  41,72%  ex1      ex1                 [.] vec_add
   8,29%  ex1      [unknown]           [k] 0xffffffffb2a00b87
   1,37%  ex1      libc-2.28.so        [.] _dl_addr
   0,92%  ex1      ld-2.28.so          [.] _dl_map_object_from_fd
   0,29%  ex1      ld-2.28.so          [.] __GI___tunables_init
```

Perf

- Attach to running process
  perf top –p <pid>
- Right:
  Sample of Jupyter Notebook run
  – HDF5 reading
  – Creating new Dict
  – Moving data around
  – Some kernel thread messing around at 6 % (Omnipath)



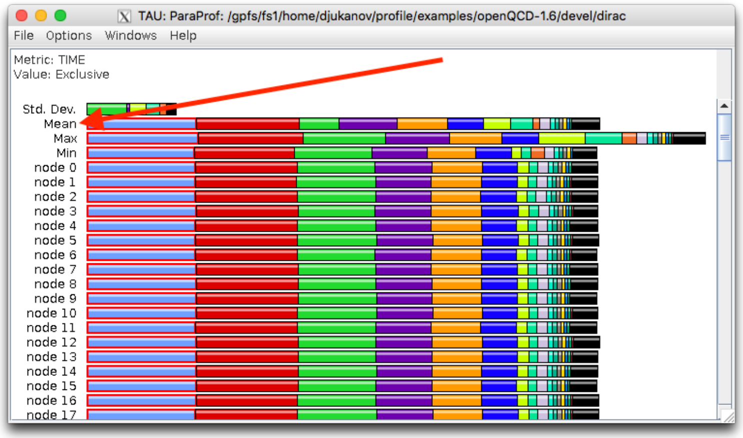
djukanov — djukanov@x0793:~ — ssh ◂ ssh -L 8889:x0793:8888 mogon2 — 80×24

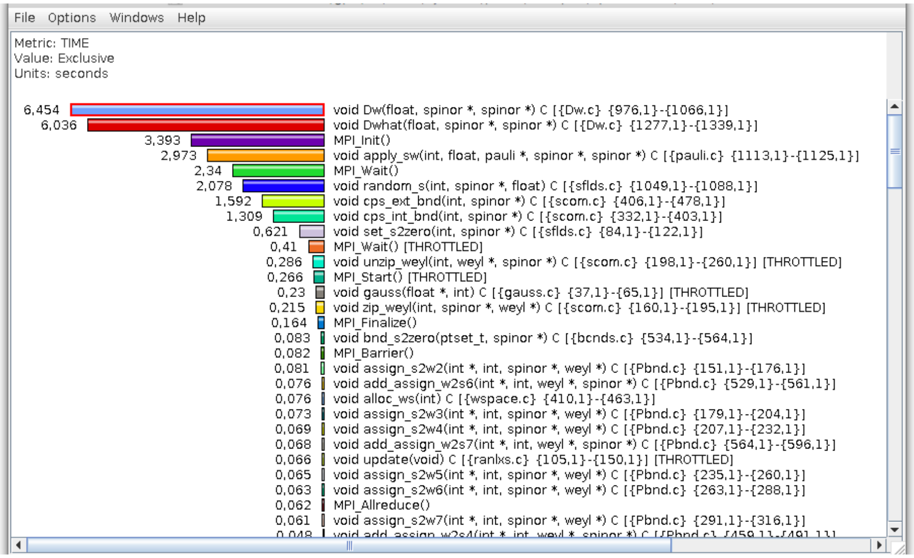
```
Samples: 123  of event 'cycles:u', 4000 Hz, Event count (approx.): 10079838 lost
Overhead  Shared Object                 Symbol
   9,87%  libpython2.7.so.1.0           [.] PyDict_New
   6,02%  _objects.so                   [.] 0x0000000000011f17
   5,91%  [kernel]                      [k] 0xffffffffb2a00b87
   5,59%  libhdf5-9028dcc4.so.103.0.0   [.] memcpy@plt
   5,23%  libhdf5-9028dcc4.so.103.0.0   [.] H5SL_search
   5,03%  libpython2.7.so.1.0           [.] PyMethod_New
   4,94%  utils.so                      [.] PyObject_Size@plt
   4,65%  libhdf5-9028dcc4.so.103.0.0   [.] H5FD_read@plt
   4,63%  _multiarray_umath.so          [.] PyArray_DescrFromType
   4,54%  libc-2.28.so                  [.] __memmove_sse2_unaligned_erms
   4,31%  libhdf5-9028dcc4.so.103.0.0   [.] H5I_get_type
   4,31%  libhdf5-9028dcc4.so.103.0.0   [.] H5open
   3,61%  libhdf5-9028dcc4.so.103.0.0   [.] H5O_msg_read@plt
   3,51%  libpython2.7.so.1.0           [.] 0x00000000000815c0
   2,29%  libpython2.7.so.1.0           [.] PyObject_GetAttr
   2,16%  libpython2.7.so.1.0           [.] 0x00000000000aae18
   2,01%  libhdf5-9028dcc4.so.103.0.0   [.] H5S_extent_release
   1,90%  libhdf5-9028dcc4.so.103.0.0   [.] H5F_block_read
   1,79%  libhdf5-9028dcc4.so.103.0.0   [.] H5F__accum_read
   1,62%  libpthread-2.28.so            [.] __libc_read
   1,61%  libhdf5-9028dcc4.so.103.0.0   [.] H5F_get_eoa
For a higher level overview, try: perf top --sort comm,dso
```

TAU

- Tuning and Analysis Tookit
- Application is instrumented in source code automatically by replacing CC with tau_cc.sh

  CC=tau_cc.sh -optTauSelectFile=./select.file
- Works with MPI

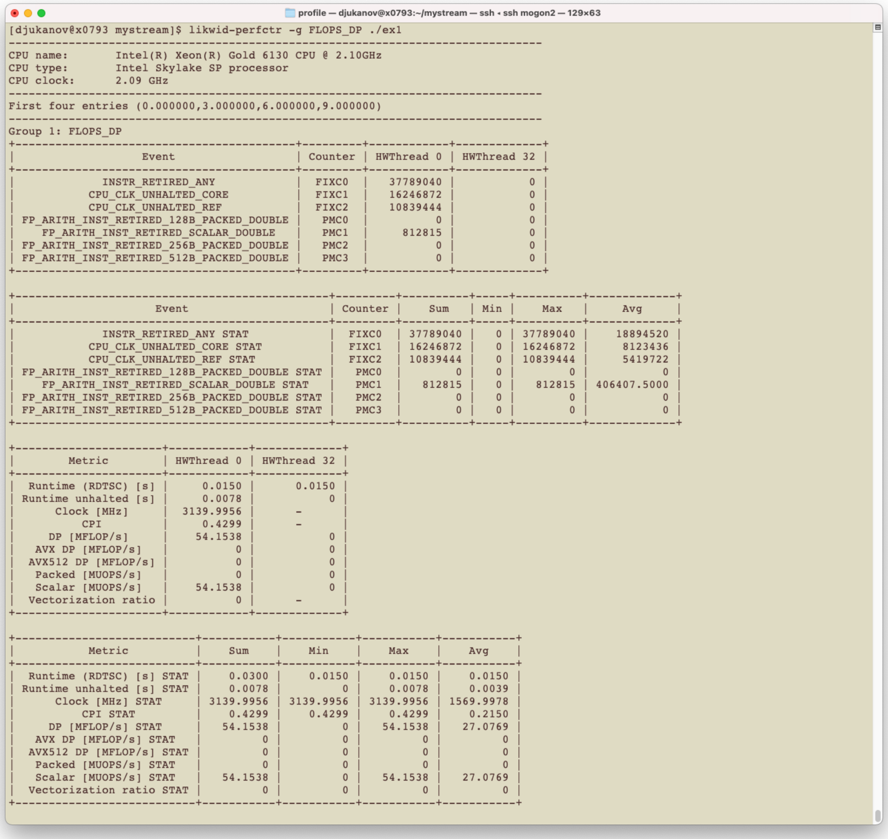




```
Metric: TIME
Value: Exclusive
Units: seconds

6,454  void Dw(float, spinor *, spinor *) C [{Dw.c} {976,1}-{1066,1}]
6,036  void Dwhat(float, spinor *, spinor *) C [{Dw.c} {1277,1}-{1339,1}]
3,393  MPI_Init()
2,973  void apply_sw(int, float, pauli *, spinor *, spinor *) C [{pauli.c} {1113,1}-{1125,1}]
2,34   MPI_Wait()
2,078  void random_s(int, spinor *, float) C [{sflds.c} {1049,1}-{1088,1}]
1,592  void cps_ext_bnd(int, spinor *) C [{scom.c} {406,1}-{478,1}]
1,309  void cps_int_bnd(int, spinor *) C [{scom.c} {332,1}-{403,1}]
0,621  void set_s2zero(int, spinor *) C [{sflds.c} {84,1}-{122,1}]
0,41   MPI_Wait() [THROTTLED]
0,286  void unzip_weyl(int, weyl *, spinor *) C [{scom.c} {198,1}-{260,1}] [THROTTLED]
0,266  MPI_Start() [THROTTLED]
0,23   void gauss(float *, int) C [{gauss.c} {37,1}-{65,1}] [THROTTLED]
0,215  void zip_weyl(int, spinor *, weyl *) C [{scom.c} {160,1}-{195,1}] [THROTTLED]
0,164  MPI_Finalize()
0,083  void bnd_s2zero(ptset_t, spinor *) C [{bcnds.c} {534,1}-{564,1}]
0,082  MPI_Barrier()
0,081  void assign_s2w2(int *, int, spinor *, weyl *) C [{Pbnd.c} {151,1}-{176,1}]
0,076  void add_assign_w2s6(int *, int, weyl *, spinor *) C [{Pbnd.c} {529,1}-{561,1}]
0,076  void alloc_ws(int) C [{wspace.c} {410,1}-{463,1}]
0,073  void assign_s2w3(int *, int, spinor *, weyl *) C [{Pbnd.c} {179,1}-{204,1}]
0,069  void assign_s2w4(int *, int, spinor *, weyl *) C [{Pbnd.c} {207,1}-{232,1}]
0,068  void add_assign_w2s7(int *, int, weyl *, spinor *) C [{Pbnd.c} {564,1}-{596,1}]
0,066  void update(void) C [{ranlxs.c} {105,1}-{150,1}] [THROTTLED]
0,065  void assign_s2w5(int *, int, spinor *, weyl *) C [{Pbnd.c} {235,1}-{260,1}]
0,063  void assign_s2w6(int *, int, spinor *, weyl *) C [{Pbnd.c} {263,1}-{288,1}]
0,062  MPI_Allreduce()
0,061  void assign_s2w7(int *, int, spinor *, weyl *) C [{Pbnd.c} {291,1}-{316,1}]
0,048  void add_assign_w2s4(int *, int, weyl *, spinor *) C [{Pbnd.c} {459,1}-{491,1}]
```

Profiling General

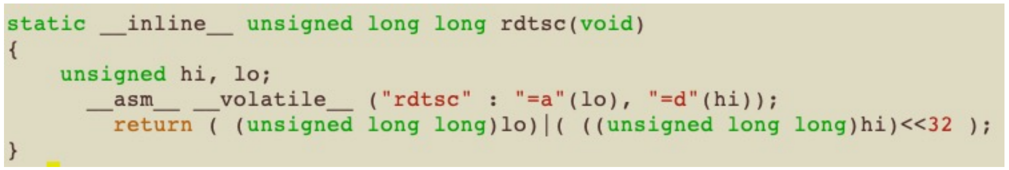
- Helps you identify HotSpots
- Helps you identify bottlenecks (performance counters)
- Be careful:
  - Overhead due to instrumentation or sampling
- Notable alternative


profile — djukanov@x0793:~/mystream — ssh ◂ ssh mogon2 — 129×63

```
[djukanov@x0793 mystream]$ likwid-perfctr -g FLOPS_DP ./ex1
--------------------------------------------------------------------------------
CPU name:       Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz
CPU type:       Intel Skylake SP processor
CPU clock:      2.09 GHz
--------------------------------------------------------------------------------
First four entries (0.000000,3.000000,6.000000,9.000000)
--------------------------------------------------------------------------------
Group 1: FLOPS_DP
+------------------------------------------+---------+------------+-------------+
|                   Event                  | Counter | HWThread 0 | HWThread 32 |
+------------------------------------------+---------+------------+-------------+
|             INSTR_RETIRED_ANY            |  FIXC0  |   37789040 |           0 |
|           CPU_CLK_UNHALTED_CORE          |  FIXC1  |   16246872 |           0 |
|           CPU_CLK_UNHALTED_REF           |  FIXC2  |   10839444 |           0 |
| FP_ARITH_INST_RETIRED_128B_PACKED_DOUBLE |   PMC0  |          0 |           0 |
|    FP_ARITH_INST_RETIRED_SCALAR_DOUBLE   |   PMC1  |     812815 |           0 |
| FP_ARITH_INST_RETIRED_256B_PACKED_DOUBLE |   PMC2  |          0 |           0 |
| FP_ARITH_INST_RETIRED_512B_PACKED_DOUBLE |   PMC3  |          0 |           0 |
+------------------------------------------+---------+------------+-------------+

+-----------------------------------------------+---------+----------+-----+----------+-------------+
|                     Event                     | Counter |    Sum   | Min |    Max   |     Avg     |
+-----------------------------------------------+---------+----------+-----+----------+-------------+
|             INSTR_RETIRED_ANY STAT            |  FIXC0  | 37789040 |   0 | 37789040 |    18894520 |
|           CPU_CLK_UNHALTED_CORE STAT          |  FIXC1  | 16246872 |   0 | 16246872 |     8123436 |
|           CPU_CLK_UNHALTED_REF STAT           |  FIXC2  | 10839444 |   0 | 10839444 |     5419722 |
| FP_ARITH_INST_RETIRED_128B_PACKED_DOUBLE STAT |   PMC0  |        0 |   0 |        0 |           0 |
|    FP_ARITH_INST_RETIRED_SCALAR_DOUBLE STAT   |   PMC1  |   812815 |   0 |   812815 | 406407.5000 |
| FP_ARITH_INST_RETIRED_256B_PACKED_DOUBLE STAT |   PMC2  |        0 |   0 |        0 |           0 |
| FP_ARITH_INST_RETIRED_512B_PACKED_DOUBLE STAT |   PMC3  |        0 |   0 |        0 |           0 |
+-----------------------------------------------+---------+----------+-----+----------+-------------+

+----------------------+------------+-------------+
|        Metric        | HWThread 0 | HWThread 32 |
+----------------------+------------+-------------+
|  Runtime (RDTSC) [s] |     0.0150 |      0.0150 |
| Runtime unhalted [s] |     0.0078 |           0 |
|      Clock [MHz]     |  3139.9956 |      -      |
|          CPI         |     0.4299 |      -      |
|     DP [MFLOP/s]     |    54.1538 |           0 |
|   AVX DP [MFLOP/s]   |          0 |           0 |
|  AVX512 DP [MFLOP/s] |          0 |           0 |
|   Packed [MUOPS/s]   |          0 |           0 |
|   Scalar [MUOPS/s]   |    54.1538 |           0 |
|  Vectorization ratio |          0 |      -      |
+----------------------+------------+-------------+

+---------------------------+-----------+-----------+-----------+-----------+
|           Metric          |    Sum    |    Min    |    Max    |    Avg    |
+---------------------------+-----------+-----------+-----------+-----------+
|  Runtime (RDTSC) [s] STAT |    0.0300 |    0.0150 |    0.0150 |    0.0150 |
| Runtime unhalted [s] STAT |    0.0078 |         0 |    0.0078 |    0.0039 |
|      Clock [MHz] STAT     | 3139.9956 | 3139.9956 | 3139.9956 | 1569.9978 |
|          CPI STAT         |    0.4299 |    0.4299 |    0.4299 |    0.2150 |
|     DP [MFLOP/s] STAT     |   54.1538 |         0 |   54.1538 |   27.0769 |
|   AVX DP [MFLOP/s] STAT   |         0 |         0 |         0 |         0 |
|  AVX512 DP [MFLOP/s] STAT |         0 |         0 |         0 |         0 |
|   Packed [MUOPS/s] STAT   |         0 |         0 |         0 |         0 |
|   Scalar [MUOPS/s] STAT   |   54.1538 |         0 |   54.1538 |   27.0769 |
|  Vectorization ratio STAT |         0 |         0 |         0 |         0 |
+---------------------------+-----------+-----------+-----------+-----------+
```

Profiling Caveats

- For the experts:
  - Timing code not easy on OutOfOrder execution archs and different Clocks
  - rdtsc: Instruction getting time stamp counters
  - Use serializing muops like cpuid

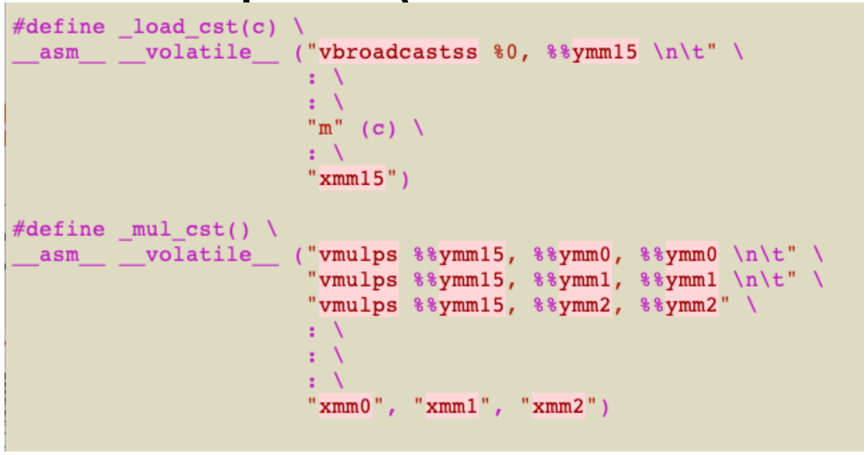


```
static __inline__ unsigned long long rdtsc(void)
{
    unsigned hi, lo;
      __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
        return ( (unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
}
```

- I haven‘t seen big differences!
  Maybe if you want cycle accurate measurements?

Assembler Coding

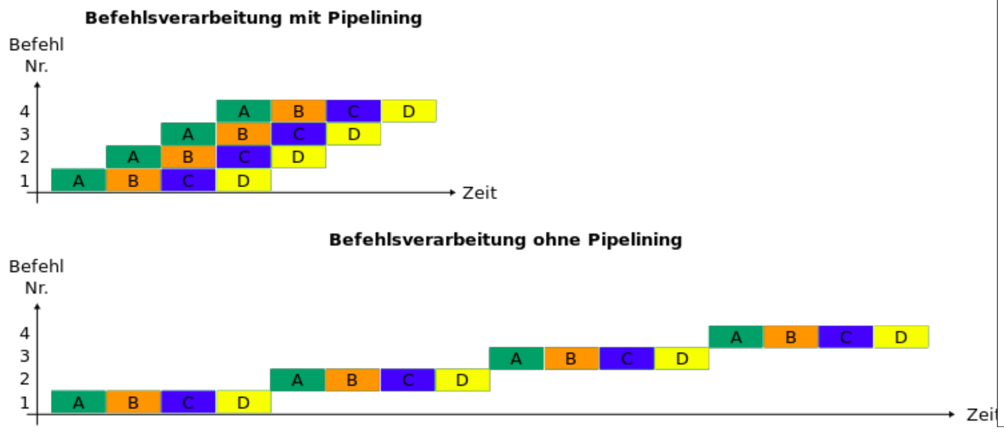
- Pros assembler:
  - Control over what is done
  - You can do better than the compiler (most of the time)
- Cons assembler:
  - Hard to maintain
  - Hard to read
  - Have to take care of everything
- Only use:
  - For best possible perf
  - For small kernels

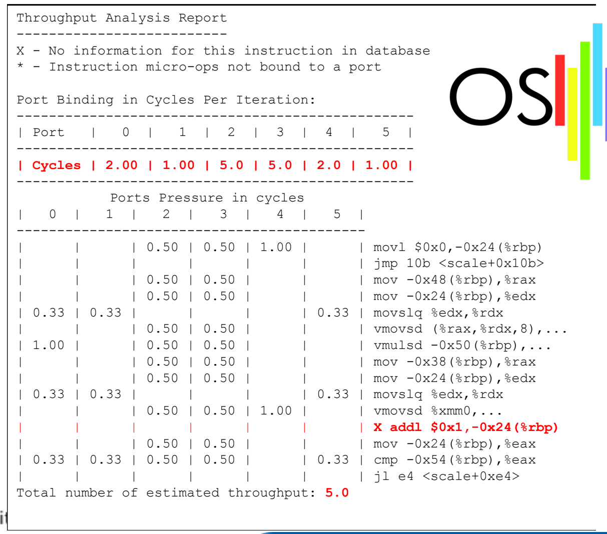


```
#define _load_cst(c) \
__asm__ __volatile__ ("vbroadcastss %0, %%ymm15 \n\t" \
                      : \
                      : \
                      "m" (c) \
                      : \
                      "xmm15")

#define _mul_cst() \
__asm__ __volatile__ ("vmulps %%ymm15, %%ymm0, %%ymm0 \n\t" \
                      "vmulps %%ymm15, %%ymm1, %%ymm1 \n\t" \
                      "vmulps %%ymm15, %%ymm2, %%ymm2" \
                      : \
                      : \
                      : \
                      "xmm0", "xmm1", "xmm2")
```

Assembler

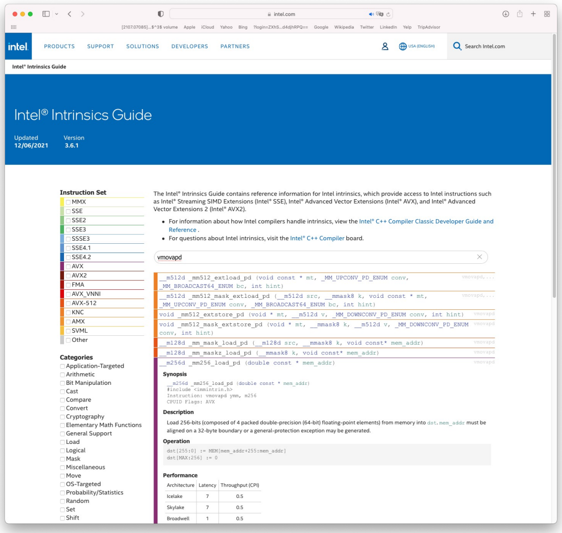
- Goals:
  - Learn how to code Assembler
  - Write a simple programm squaring a vector
- Notable Omissions:
  - Will not discuss in detail pipelining or latency hiding


Befehlsverarbeitung mit Pipelining
Befehl Nr.
4
3
2
1
A
B
C
D
Zeit
Befehlsverarbeitung ohne Pipelining
Befehl Nr.
4
3
2
1
A
B
C
D
Zeit


```
Throughput Analysis Report
--------------------------
X - No information for this instruction in database
* - Instruction micro-ops not bound to a port

Port Binding in Cycles Per Iteration:
-------------------------------------------------
| Port   |  0   |  1   | 2   | 3   | 4   |  5   |
-------------------------------------------------
| Cycles | 2.00 | 1.00 | 5.0 | 5.0 | 2.0 | 1.00 |
-------------------------------------------------

           Ports Pressure in cycles
|  0   |  1   |  2   |  3   |  4   |  5   |
-------------------------------------------
|      |      | 0.50 | 0.50 | 1.00 |      | movl $0x0,-0x24(%rbp)
|      |      |      |      |      |      | jmp 10b <scale+0x10b>
|      |      | 0.50 | 0.50 |      |      | mov -0x48(%rbp),%rax
|      |      | 0.50 | 0.50 |      |      | mov -0x24(%rbp),%edx
| 0.33 | 0.33 |      |      |      | 0.33 | movslq %edx,%rdx
|      |      | 0.50 | 0.50 |      |      | vmovsd (%rax,%rdx,8),...
| 1.00 |      | 0.50 | 0.50 |      |      | vmulsd -0x50(%rbp),...
|      |      | 0.50 | 0.50 |      |      | mov -0x38(%rbp),%rax
|      |      | 0.50 | 0.50 |      |      | mov -0x24(%rbp),%edx
| 0.33 | 0.33 |      |      |      | 0.33 | movslq %edx,%rdx
|      |      | 0.50 | 0.50 | 1.00 |      | vmovsd %xmm0,...
|      |      |      |      |      |      | X addl $0x1,-0x24(%rbp)
|      |      | 0.50 | 0.50 |      |      | mov -0x24(%rbp),%eax
| 0.33 | 0.33 | 0.50 | 0.50 |      | 0.33 | cmp -0x54(%rbp),%eax
|      |      |      |      |      |      | jl e4 <scale+0xe4>
Total number of estimated throughput: 5.0
```

Assembler

- Actually no need to write assembler to use vector units
  Intrinsics:
  – Look like ordinary function calls working on vector memory, e.g. __m256
  – We do not have to deal with registers
  – Could lead to spilling (or other things)
- We‘ll do the hard thing



intel.com

[2107.07085]...^{3} volume Apple iCloud Yahoo Bing ?login=ZXhS...d4djhRPQ== Google Wikipedia Twitter LinkedIn Yelp TripAdvisor

intel PRODUCTS SUPPORT SOLUTIONS DEVELOPERS PARTNERS USA (ENGLISH) Search Intel.com

Intel® Intrinsics Guide

Intel® Intrinsics Guide

Updated 12/06/2021 Version 3.6.1

Instruction Set

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☐ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX_VNNI
- ☐ AVX-512
- ☐ KNC
- ☐ AMX
- ☐ SVML
- ☐ Other

Categories

- ☐ Application-Targeted
- ☐ Arithmetic
- ☐ Bit Manipulation
- ☐ Cast
- ☐ Compare
- ☐ Convert
- ☐ Cryptography
- ☐ Elementary Math Functions
- ☐ General Support
- ☐ Load
- ☐ Logical
- ☐ Mask
- ☐ Miscellaneous
- ☐ Move
- ☐ OS-Targeted
- ☐ Probability/Statistics
- ☐ Random
- ☐ Set
- ☐ Shift

The Intel® Intrinsics Guide contains reference information for Intel intrinsics, which provide access to Intel instructions such as Intel® Streaming SIMD Extensions (Intel® SSE), Intel® Advanced Vector Extensions (Intel® AVX), and Intel® Advanced Vector Extensions 2 (Intel® AVX2).

- For information about how Intel compilers handle intrinsics, view the Intel® C++ Compiler Classic Developer Guide and Reference .
- For questions about Intel intrinsics, visit the Intel® C++ Compiler board.

vmovapd

```
__m512d _mm512_extload_pd (void const * mt, _MM_UPCONV_PD_ENUM conv, _MM_BROADCAST64_ENUM bc, int hint)    vmovapd, ...
__m512d _mm512_mask_extload_pd (__m512d src, __mmask8 k, void const * mt, _MM_UPCONV_PD_ENUM conv, _MM_BROADCAST64_ENUM bc, int hint)    vmovapd, ...
void _mm512_extstore_pd (void * mt, __m512d v, _MM_DOWNCONV_PD_ENUM conv, int hint)    vmovapd
void _mm512_mask_extstore_pd (void * mt, __mmask8 k, __m512d v, _MM_DOWNCONV_PD_ENUM conv, int hint)    vmovapd
__m128d _mm_mask_load_pd (__m128d src, __mmask8 k, void const* mem_addr)    vmovapd
__m128d _mm_maskz_load_pd (__mmask8 k, void const* mem_addr)    vmovapd
__m256d _mm256_load_pd (double const * mem_addr)    vmovapd
```

Synopsis

```
__m256d _mm256_load_pd (double const * mem_addr)
#include <immintrin.h>
Instruction: vmovapd ymm, m256
CPUID Flags: AVX
```

Description

Load 256-bits (composed of 4 packed double-precision (64-bit) floating-point elements) from memory into dst. mem_addr must be aligned on a 32-byte boundary or a general-protection exception may be generated.

Operation

```
dst[255:0] := MEM[mem_addr+255:mem_addr]
dst[MAX:256] := 0
```

Performance

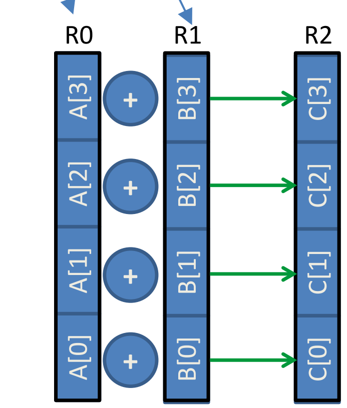
Architecture	Latency	Throughput (CPI)
Icelake	7	0.5
Skylake	7	0.5
Broadwell	1	0.5

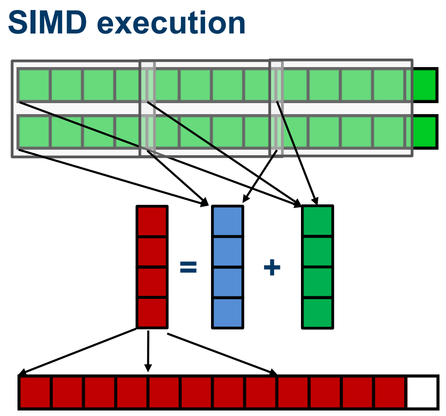
Assembler

- Example:
  We want to do
- Everything double
- Vectorize:
  We need to do
- In AVX can load 4 doubles

```
for(i=0;i<num;i++)
{
   c[i]=x[i]+y[i];
}
```

No Intraloop Dependency!





How?

Assembler

- Have different options (NASM, etc.)

```
for(i=0;i<num;i++)
{
  c[i]=x[i]+y[i];
}
```

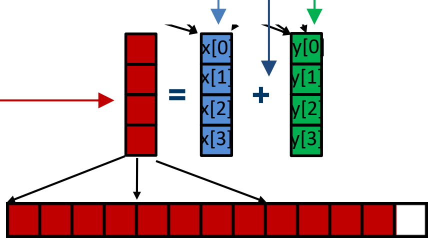
Do not delete

```
__asm__ __volatile__(\
                "vmovapd %1, %%ymm0\n"\
                "vmovapd %2, %%ymm1\n"\
                "vaddpd %%ymm0,%%ymm1,%%ymm0\n"\
                "vmovapd %%ymm0,%0\n"\
                :\
                "=m" (c[i*4])\
                :\
                "m" (x[i*4]),\
                "m" (y[i*4])\
                :\
                "ymm0", "ymm1");
```

Store operators

Load operators

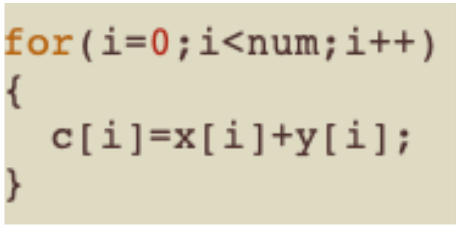
Clobber



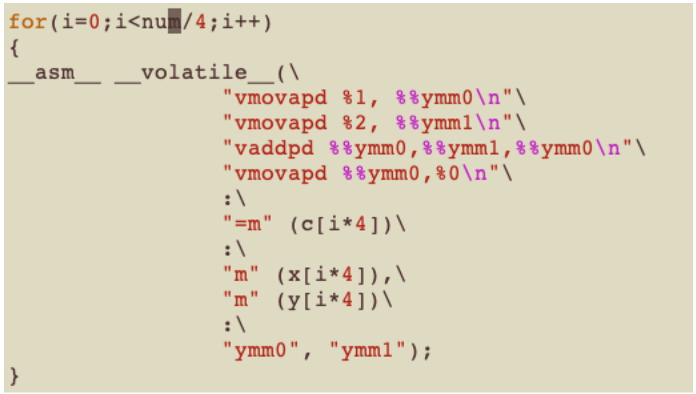
- Need 2 AVX Registers YMM0 & YMM1
- Need 2 Loads and 1 Store
- Need 1 Add on those registers
- Total 4 instructions to add 4+4 doubles
- Dependency:
  - Loads before add
  - Add before store
  - Store at the end

Assembler

- What we did is basically loop-unrolling



```
for(i=0;i<num;i++)
{
  c[i]=x[i]+y[i];
}
```

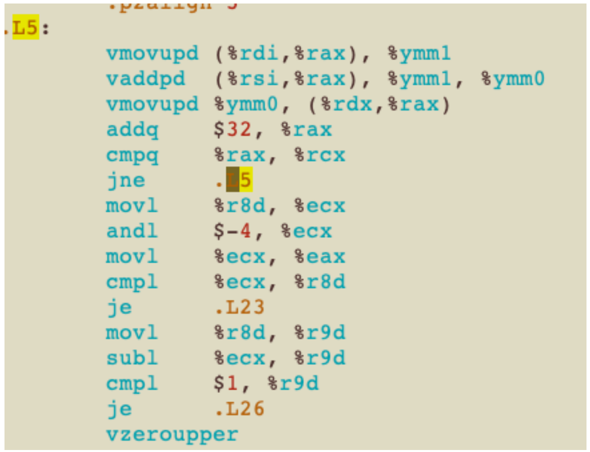


```
for(i=0;i<num/4;i++)
{
__asm__ __volatile__(\
                "vmovapd %1, %%ymm0\n"\
                "vmovapd %2, %%ymm1\n"\
                "vaddpd %%ymm0,%%ymm1,%%ymm0\n"\
                "vmovapd %%ymm0,%0\n"\
                :\
                "=m" (c[i*4])\
                :\
                "m" (x[i*4]),\
                "m" (y[i*4])\
                :\
                "ymm0", "ymm1");
}
```

- Compiler could do that in simple cases
- OpenMP Pragmas to tell compilers:
  - No Dependency:
    #pragma omp simd
- Need a lot of flags to get the above

Assembler

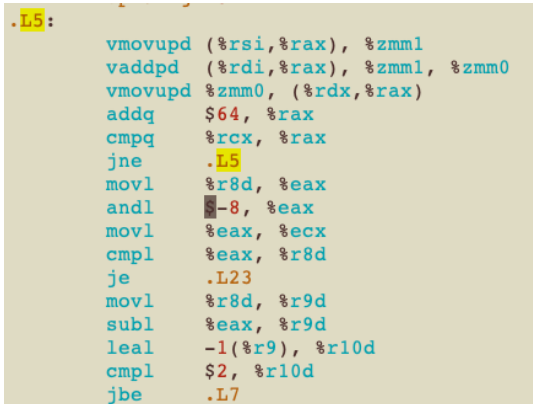
- gcc ex2.c -O3 -march=skylake-avx512 -S



```
.L5:
        vmovupd (%rdi,%rax), %ymm1
        vaddpd  (%rsi,%rax), %ymm1, %ymm0
        vmovupd %ymm0, (%rdx,%rax)
        addq    $32, %rax
        cmpq    %rax, %rcx
        jne     .L5
        movl    %r8d, %ecx
        andl    $-4, %ecx
        movl    %ecx, %eax
        cmpl    %ecx, %r8d
        je      .L23
        movl    %r8d, %r9d
        subl    %ecx, %r9d
        cmpl    $1, %r9d
        je      .L26
        vzeroupper
```

Not clear why. Frequency dropped @ AVX512
Note vmovupd vs vmovapd!

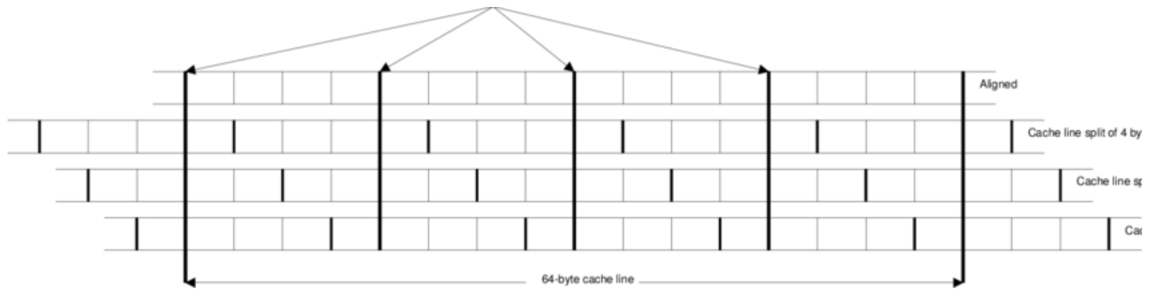
- Not using ZMM registers!
- gcc ex2.c -O3 -march=skylake-avx512 -mprefer-vector-width=512 -S



```
.L5:
        vmovupd (%rsi,%rax), %zmm1
        vaddpd  (%rdi,%rax), %zmm1, %zmm0
        vmovupd %zmm0, (%rdx,%rax)
        addq    $64, %rax
        cmpq    %rcx, %rax
        jne     .L5
        movl    %r8d, %eax
        andl    $-8, %eax
        movl    %eax, %ecx
        cmpl    %eax, %r8d
        je      .L23
        movl    %r8d, %r9d
        subl    %eax, %r9d
        leal    -1(%r9), %r10d
        cmpl    $2, %r10d
        jbe     .L7
```

Assembler

- Aligned Mem Access vmov**a**p(sd)
  Memory must be aligned at cache line boundary
- Avoids cache line splits



- Not a drastic penalty anymore

Assembler

- Example with Intraloop dependency

```
double square(int num,double *a, double c) {
    int i;
    for(i=0;i<num;i++)
    {
        c=c+a[i];
    }
    return 0;
}
```

- Every iteration needs result of the one before
- Add accumulators

```
double square(int num,double *a, double c) {
    int i;
    double c0,c1,c2,c3;
    for(i=0;i<num/4;i++)
    {
        c0=c0+a[i*4];
        c1=c1+a[i*4+1];
        c2=c2+a[i*4+2];
        c3=c3+a[i*4+3];
    }
    return c0+c1+c2+c3;
}
```

- 4-Way vectorization final horizontal add

Real World Example

- Lattice QCD Add 2 SU(3) Vectors

```
#define _vector_add(r,s1,s2) \
   (r).c1.re=(s1).c1.re+(s2).c1.re; \
   (r).c1.im=(s1).c1.im+(s2).c1.im; \
   (r).c2.re=(s1).c2.re+(s2).c2.re; \
   (r).c2.im=(s1).c2.im+(s2).c2.im; \
   (r).c3.re=(s1).c3.re+(s2).c3.re; \
   (r).c3.im=(s1).c3.im+(s2).c3.im
```

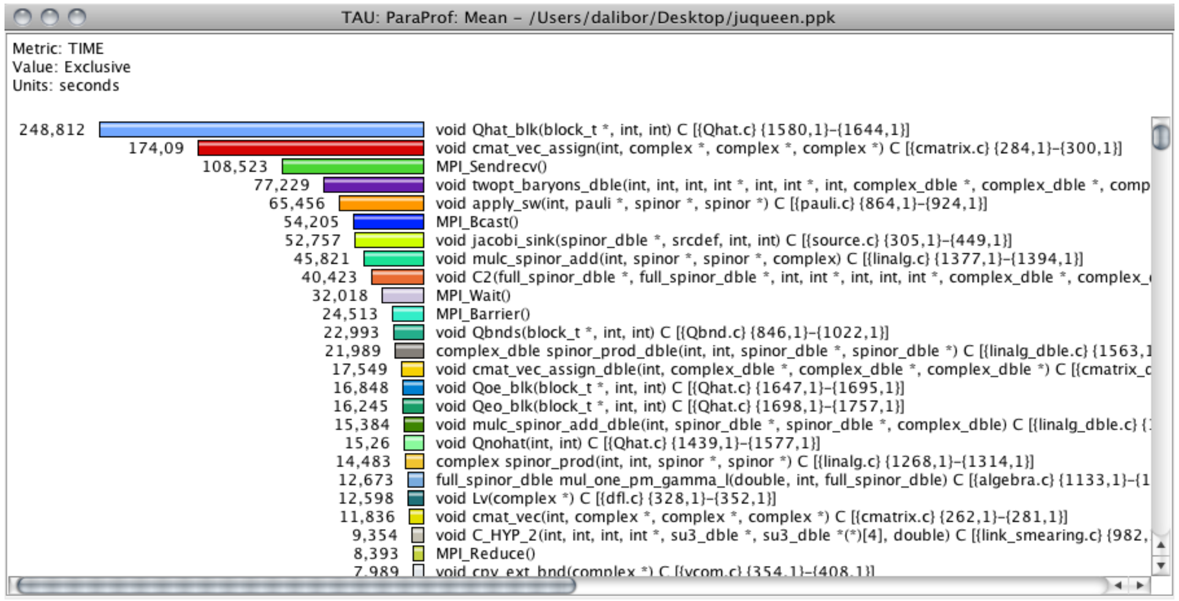
Real World Example

- Lattice QCD Add 2 SU(3) Vectors Intrinsics

```
vector4double v1,v2;
#define _vector_add_qpx(r,s1,s2) \
        v1=vec_lda(0,&((s1).c1.re)); \
        v2=vec_lda(0,&((s2).c1.re)); \
        vec_sta(vec_add(v1,v2),0,&((r).c1.re)); \
        v1=vec_ld2a(0,&((s1).c3.re)); \
        v2=vec_ld2a(0,&((s2).c3.re)); \
        vec_st2a(vec_add(v1,v2),0,&((r).c3.re));
```

Real World Example

- Profile to identify HotSpots



TAU: ParaProf: Mean – /Users/dalibor/Desktop/juqueen.ppk

```
Metric: TIME
Value: Exclusive
Units: seconds

248,812  void Qhat_blk(block_t *, int, int) C [{Qhat.c} {1580,1}-{1644,1}]
174,09   void cmat_vec_assign(int, complex *, complex *, complex *) C [{cmatrix.c} {284,1}-{300,1}]
108,523  MPI_Sendrecv()
77,229   void twopt_baryons_dble(int, int, int, int *, int, int *, int, complex_dble *, complex_dble *, comp
65,456   void apply_sw(int, pauli *, spinor *, spinor *) C [{pauli.c} {864,1}-{924,1}]
54,205   MPI_Bcast()
52,757   void jacobi_sink(spinor_dble *, srcdef, int, int) C [{source.c} {305,1}-{449,1}]
45,821   void mulc_spinor_add(int, spinor *, spinor *, complex) C [{linalg.c} {1377,1}-{1394,1}]
40,423   void C2(full_spinor_dble *, full_spinor_dble *, int, int *, int, int, int *, complex_dble *, complex_
32,018   MPI_Wait()
24,513   MPI_Barrier()
22,993   void Qbnds(block_t *, int, int) C [{Qbnd.c} {846,1}-{1022,1}]
21,989   complex_dble spinor_prod_dble(int, int, spinor_dble *, spinor_dble *) C [{linalg_dble.c} {1563,1
17,549   void cmat_vec_assign_dble(int, complex_dble *, complex_dble *, complex_dble *) C [{cmatrix_c
16,848   void Qoe_blk(block_t *, int, int) C [{Qhat.c} {1647,1}-{1695,1}]
16,245   void Qeo_blk(block_t *, int, int) C [{Qhat.c} {1698,1}-{1757,1}]
15,384   void mulc_spinor_add_dble(int, spinor_dble *, spinor_dble *, complex_dble) C [{linalg_dble.c} {
15,26    void Qnohat(int, int) C [{Qhat.c} {1439,1}-{1577,1}]
14,483   complex spinor_prod(int, int, spinor *, spinor *) C [{linalg.c} {1268,1}-{1314,1}]
12,673   full_spinor_dble mul_one_pm_gamma_l(double, int, full_spinor_dble) C [{algebra.c} {1133,1}-{1
12,598   void Lv(complex *) C [{dfl.c} {328,1}-{352,1}]
11,836   void cmat_vec(int, complex *, complex *, complex *) C [{cmatrix.c} {262,1}-{281,1}]
9,354    void C_HYP_2(int, int, int, int *, su3_dble *, su3_dble *(*)[4], double) C [{link_smearing.c} {982,
8,393    MPI_Reduce()
7,989    void cpy_ext_bnd(complex *) C [{vcom.c} {354,1}-{408,1}]
```

Real World Example

- Can get complicated

```
static vector4double perm1={2.000000,2.250000,3.000000,3.250000};
static vector4double perm2={2.500000,2.750000,3.500000,3.750000};
.
.
v1=vec_ld2a(0,&((*m).u[6]));
v2=vec_lda(0,&((*m).u[8]));
v3=vec_lda(0,&((*m).u[12]));
v4=vec_lda(0,&((*m).u[16]));
v5=vec_lda(0,&((*m).u[20]));
/* First 4 Components */

v100=vec_perm(v10,v10,perm0011);                    /*(u0,u0,u1,u1) */
s10=vec_sldw(s1,s2,2);                              /*s1.re, s1.im, s2.re, s2.im*/
s11=vec_sldw(s2,s1,2);                              /*s2.re, s2.im, s1.im, s1.im*/
v12=vec_perm(v2,v4,perm1);                          /*(u8,u9,u16,u17)*/
v13=vec_perm(v2,v4,perm2);                          /*(u10,u11,u18,u19)*/
v14=vec_perm(v3,v5,perm1);                          /*(u12,u13,u20,u21)*/
v15=vec_perm(v3,v5,perm2);                          /*(u14,u15,u22,u23)*/
r10=vec_xmul(v100,s10);
r11=vec_xxnpmadd(s11,vec_mul((vector4double)(1,1,1,-1),v1),vec_xmul(v1,s11));
r12=vec_xxnpmadd(v12,s3,vec_xmul(s3,v12));
r13=vec_xxnpmadd(v13,s4,vec_xmul(s4,v13));
r14=vec_xxnpmadd(v14,s5,vec_xmul(s5,v14));
r15=vec_xxnpmadd(v15,s6,vec_xmul(s6,v15));

r100=vec_add(r10,r11);
r101=vec_add(r12,r13);
r102=vec_add(r14,r15);
r110=vec_add(r100,r101);
r111=vec_add(r110,r102);
vec_sta(r111,0,&((*r).c1.c1.re));
```

Further Improvements:
Mix Loads and (F)MADs!
Order data dependency according to
Instruction latency. (See Bagel maybe?)
This can only be done in assembly, I guess
Not Sure how the compiler orders regs?

Bad Version not using Multiply-Add!
This was version 3. Changed in v4.

Improvements - Version 0.4

- After 4 weeks of tuning
- Using timing programs from devel/dirac subdirectory in DD-HMC
- Local Lattice 16x16x16x16
- Processorgrid 4x2x2x2
- Results in Mflops/process
- Speedup for QPXD in brackets

Routine	-O3 -qstrict	-O3 –qstrict -DQPXD	-O2
Qhat	688 (2.08x)	1428	640 (2.23x)
Qhat_blk	865 (2.15x)	1859	691 (2.69x)
Qnohat	562 (2.38x)	1335	499 (2.68x)
Qhat_dble	436 (1.78x)	774	438 (1.77x)
Qhat_blk_dble	576 (1.89x)	1089	566 (1.92x)
Qnohat_dble	403 (1.71x)	690	392 (1.76x)

- Using Prefetch by data cache block touch to preload __dcbt
- Set pipeline depth should improve DP code?

HELMHOLTZ
Helmholtz-Institut Mainz