High Performance Computing – Tutorial

Dr.-Ing. Martin Bernreuther

Höchstleistungsrechenzentrum der
Universität Stuttgart

SS 2020

Part I

Tutorial: Streaming benchmark

Stream benchmark

- Create a C-program to calculate vector operations like
  1. $a[i] = b[i] \qquad \forall i = 0, \ldots, N-1$
  2. $a[i] = b[i] + c[i] \qquad \forall i = 0, \ldots, N-1$
  3. $a[i] = s \cdot b[i] + c[i] \qquad \forall i = 0, \ldots, N-1$
  4. $a[i] = b[i] \cdot c[i] + d[i] \qquad \forall i = 0, \ldots, N-1$

  also see http://www.cs.virginia.edu/stream/ or likwid-bench

  $\rightarrow$ we'll focus on the last variant, the Schönauer vector triade
- Vectors are one-dimensional dynamical arrays of length N, which initially are allocated and initialized with values.
  (thus there's already a "first touch" and cache read).
- the (BLAS1) vector operations will be executed *nrepeat* times, the walltime is measured with e.g. gettimeofday to calculate an average.
- determine the average FlOp/s and memory bandwidth for varying N (and *STRIDE*, double vs. single precision. . .) and visualize it with e.g. gnuplot
- these stream benchmarks are memory-bound!

valgrind

- valgrind takes control of a program and runs it on a synthetic CPU (The program binary needs not to be modified in any way.)
- valgrind supports several platforms, like currently (v3.15) e.g.

  x86/Linux: up to and including SSSE3, but not higher – no SSE4, AVX, AVX2. This target is in maintenance mode now..

  AMD64/Linux: up to and including AVX2. This is the primary development target and tends to be well supported.

  ARM/Linux: supported since ARMv7.

  ARM64/Linux: supported for ARMv8.

  ...

- before execution of a code part it is handed to a tool (formerly referred to as skin), which adds instrumentation code

another similar utility: intel pin

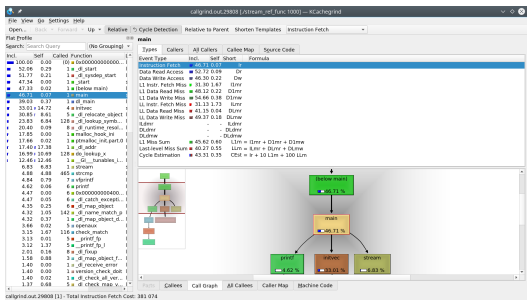
valgrind tools

- the cache behaviour of the benchmarks can be simulated using the cachegrind tool: valgrind --tool=cachegrind <binary>
- especially for applications with multiple functions, the "callgrind" tool can determine these performance values and more on a functional level and visualize it with "kcachegrind"
- tools for multithreading applications:
  (e.g. pthreads; there might be problems with (Linux) futex/OpenMP)
  - "helgrind" – a thread error detector
  - "drd" (**D**ata **R**ace **D**etection) – a thread error detector
- tools for memory usage:
  - "memcheck" – a memory error detector (default)
  - "massif" – a heap profiler
  - "DHAT" (**D**ynamic **H**eap **A**nalysis **T**ool)

Using cachegrind & callgrind

compile with debugging information ("-g") for source file related output

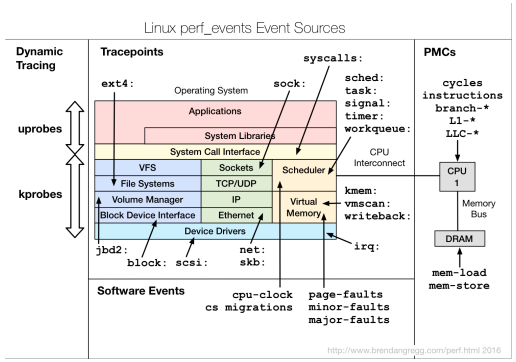
- Cachegrind
  - valgrind --tool=cachegrind --branch-sim=yes ./stream_ref_func 10000
    produces a text file (cachegrind.out.*) with cache usage information
  - cg_annotate --auto=yes --show-percs=yes cachegrind.out.*
    to show the cache ref./misses for each line of the source code
- Callgrind (extension of cachegrind)
  - valgrind --tool=callgrind --cache-sim=yes ./stream_ref_func 1000
    produces a text file callgrind.out.*
    you might also use more options, like e.g.
    --simulate-wb=yes --simulate-hwpref=yes --cacheuse=yes
  - kcachegrind callgrind .out.*
    to visualize the data



remember: cache accesses refer to cachelines

Linux perf

- **P**erformance **C**ounters for **L**inux/perf is a userspace performance analyzing tool of the Linux kernel
  (unlike e.g. for OProfile no daemon is required; reduce /proc/sys/kernel/perf_event_paranoid)
- list events with e.g. perf list ––long–desc ––details
  (perf list hw pmu to list hardware events only)



source:www.brendangregg.com

- more perf commands

perf top [–e <event_type>]	dynamic list of running functions (like top)
perf stat –ad –– <cmd_with_options>	gather performance counter statistics
perf record <cmd_with_options>|–p <PID>	record profile
perf report	display profile
perf annotate –d <cmd>	source code annotation

Linux perf stat example

- perf stat example output:

```
$ export LC_NUMERIC=C
$ perf stat -d taskset 1 ./stream_ref.gcc 100000 1
 Performance counter stats for 'taskset 1 ./stream_ref.gcc 100000 1':
      2.190340      task-clock (msec)         #    0.864 CPUs utilized
             0      context-switches          #    0.000 K/sec
             0      cpu-migrations            #    0.000 K/sec
           920      page-faults               #    0.420 M/sec
       7504129      cycles                    #    3.426 GHz
       7072664      instructions              #    0.94  insn per cycle
       1287923      branches                  #  588.001 M/sec
         21833      branch-misses             #    1.70% of all branches
       1249007      L1-dcache-loads           #  570.234 M/sec
        150852      L1-dcache-load-misses     #   12.08% of all L1-dcache hits
<not counted>      LLC-loads                                     (0.00%)
<not counted>      LLC-load-misses                               (0.00%)
  0.002535691 seconds time elapsed
$ perf stat -e cycles,instructions,\
  fp_arith_inst_retired.scalar_double,fp_arith_inst_retired.256b_packed_double\
  ./stream_avx.gcc 100
 Performance counter stats for './stream_ref-avx2fma3.gcc 100':
          1628452      cycles
           897762      instructions              #    0.55  insn per cycle
                6      fp_arith_inst_retired.scalar_double
               50      fp_arith_inst_retired.256b_packed_double
     0.000947457 seconds time elapsed
```

(scalar FP instructions used for time calculation; FMA instructions count twice)

Performance Application Programming Interface

- The **P**erformance **A**pplication **P**rogramming **I**nterface (PAPI) is a machine independent API to provide access to performance counters
- check PAPI events availability and get information with papi_avail
- include papi.h and compile/link with −lpapi
- might use e.g. the high level function

  int PAPI_flops(**float** *rtime, **float** *ptime, **long long** *flpops, **float** *mflops);

  before and after the loop(s) to determine the FLOP/s...

  (How are e.g. AVX instructions counted?)

Loop unrolling for the vector triad

```
for(i=0;i<N;i+=SIMD_WIDTH)
{/* for SIMD_WIDTH==4 */
 a[i]=b[i]*c[i]+d[i];
 a[i+1]=b[i+1]*c[i+1]+d[i+1];
 a[i+2]=b[i+2]*c[i+2]+d[i+2];
 a[i+3]=b[i+3]*c[i+3]+d[i+3];
 /* might be replaced by a single SIMD instr.*/
}
```

- with explicit successive calculations of consecutive loop iterations, it might be easier for the compiler to optimize and generate AVX-instructions.

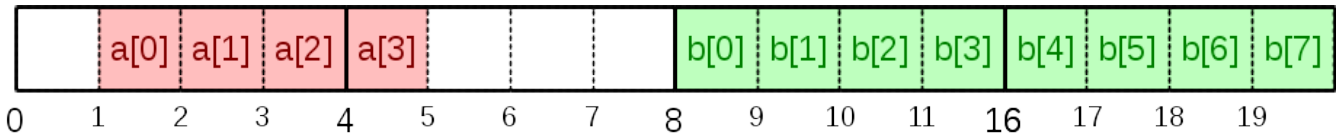
  It's possible to check the compiler dependent optimization looking at an assembler listing. ▸ objdump
- shouldn't be necessary for this simple example, but sometimes still does not lead to the hoped for result...

AVX intrinsics variant

- explicit utilization of AVX intrinsics to increase performance
- 4 double precision values (with 8 Byte = 64 Bit each) are processed simultaneously in 32 Byte (= 256 Bit) registers.
  Assumption: N is divisible by 4 ($N\%4 == 0$)

  (otherwise "padding" with dummy-elements or special treatment with scalar operations)
- Data-alignment
  - "packed double" operations access data with starting addresses being $4 \cdot 8 = 32$ byte aligned.
  - to ensure alignment the memory allocation should use suitable functions to replace malloc.

Memory allocation with data alignment



- "MultiMedia" Intrinsics aligned memory allocation:

```
#include <mm_malloc.h>
void* _mm_malloc (int size, int align);
void _mm_free (void *p);
```

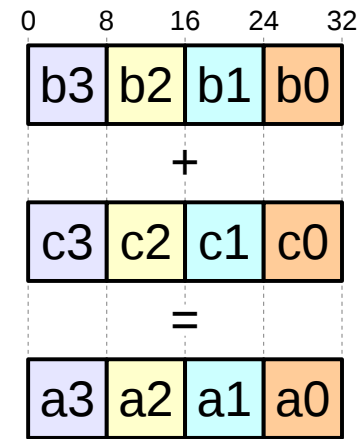
- POSIX posix_memalign, C11 aligned_alloc :

```
#include <stdlib.h>
int posix_memalign(void **memptr, size_t alignment, size_t size);
void *aligned_alloc(size_t alignment, size_t size);
```

- gcc offers e.g. keyword __attribute__ (aligned(X)) for static arrays and structs/classes
- C++11: alignas(X); also check std::align and std::aligned_storage
- Library functions: Boost.Align

AVX intrinsics version details

- AVX-Commands
  - to use AVX commands it's necessary to
    #include <immintrin.h>
  - usage of AVX double precision registers: __m256d va;
  - set values: va=_mm256_set_pd(a3,a2,a1,a0); or va=_mm256_set1_pd(a);
  - load values (aligned) from an array: (**p**acked **d**ouble)
    va=_mm256_load_pd(&a[i]);
  - write values (aligned) to an array: _mm256_store_pd(&a[i],va);
    alternative: _mm256_stream_pd(&a[i],va);
  - multiplication, addition:
    va=_mm256_mul_pd(vb,vc);
    va=_mm256_add_pd(va,vd);
    //va=_mm256_fmadd_pd(vb,vc,vd);
  - AMD Interlagos, Intel Haswell:
    also HW-supported **F**used **M**ultiply **A**dd 4
- compiler optimization also applies SIMD-instructions
  (possibly specify architecture, e.g. gcc `-mavx`)



AVX intrinsics variant: vector triad excerpt

```
for(i=0;i<N;i+=SIMD_WIDTH) {
  vb=_mm256_load_pd(&b[i]);
  vc=_mm256_load_pd(&c[i]);
  vd=_mm256_load_pd(&d[i]);
#ifdef USE_AVX_FMA4
  va=_mm256_fmadd_pd(vb,vc,vd);
#else
  va=_mm256_add_pd(_mm256_mul_pd(vb,vc),vd);
#endif
#ifdef USE_AVX_STREAM
  _mm256_stream_pd(&a[i],va);
#else
  _mm256_store_pd(&a[i],va);
#endif
}
```

more AVX intrinsics variations

- replace _mm256_load_pd(&a[i]) with _mm256_loadu_pd(&a[i]) and _mm256_store_pd(&a[i]) with _mm256_storeu_pd(&a[i]) to allow unaligned memory access and compare the performance.
- replace _mm256_load_pd(&a[i]) with va=_mm256_set_pd(a[i+3],a[i+2],a[i+1],a[i]); to simulate gathering the data from different variables instead of loading aligned array data.
- setting a mask with _mm256_set_epi64x and masking loads with _mm256_maskload_pd and stores with _mm256_maskstore_pd will add flexibility for e.g. strided access or conditional statements. Using

```
__m256i vmask=_mm256_set_epi64x(-1,-1,-1,-1);
```

will retain the functionality of the unmasked version, but probably still influence the performance.

SSE2 intrinsics variant

- as a comparison and also for CPUs, which do not support AVX extensions (cmp. "flags" in "/proc/cpuinfo"),
  analogous also a SSE2 variant can be created
- SSE: 2 double precision values are processed simultaneously.
- alignment: $2 \cdot 8 = 16$ byte
- **#include** <emmintrin.h> (for SSE2)
  MMX:mmintrin.h, SSE:xmmintrin.h, SSE3:pmmintrin.h, SSSE3:tmmintrin.h, SSE4.1:smmintrin.h, SSE4.2:nmmintrin.h, SSE4A:ammintrin.h, ...
- SSE2 double precision registers: __m128d va;
- prefix for the commands: "_mm_"
- set values: va=_mm_set_pd(a1,a0);
- no "fma" (fused multiply-add) or "stream" instruction
- _mm_prefetch(ADDRESS,LOCALITYHINT) might give hints for prefetching

Intel intrinsics: performance

Performance (Latency; Throughput [CPI]):

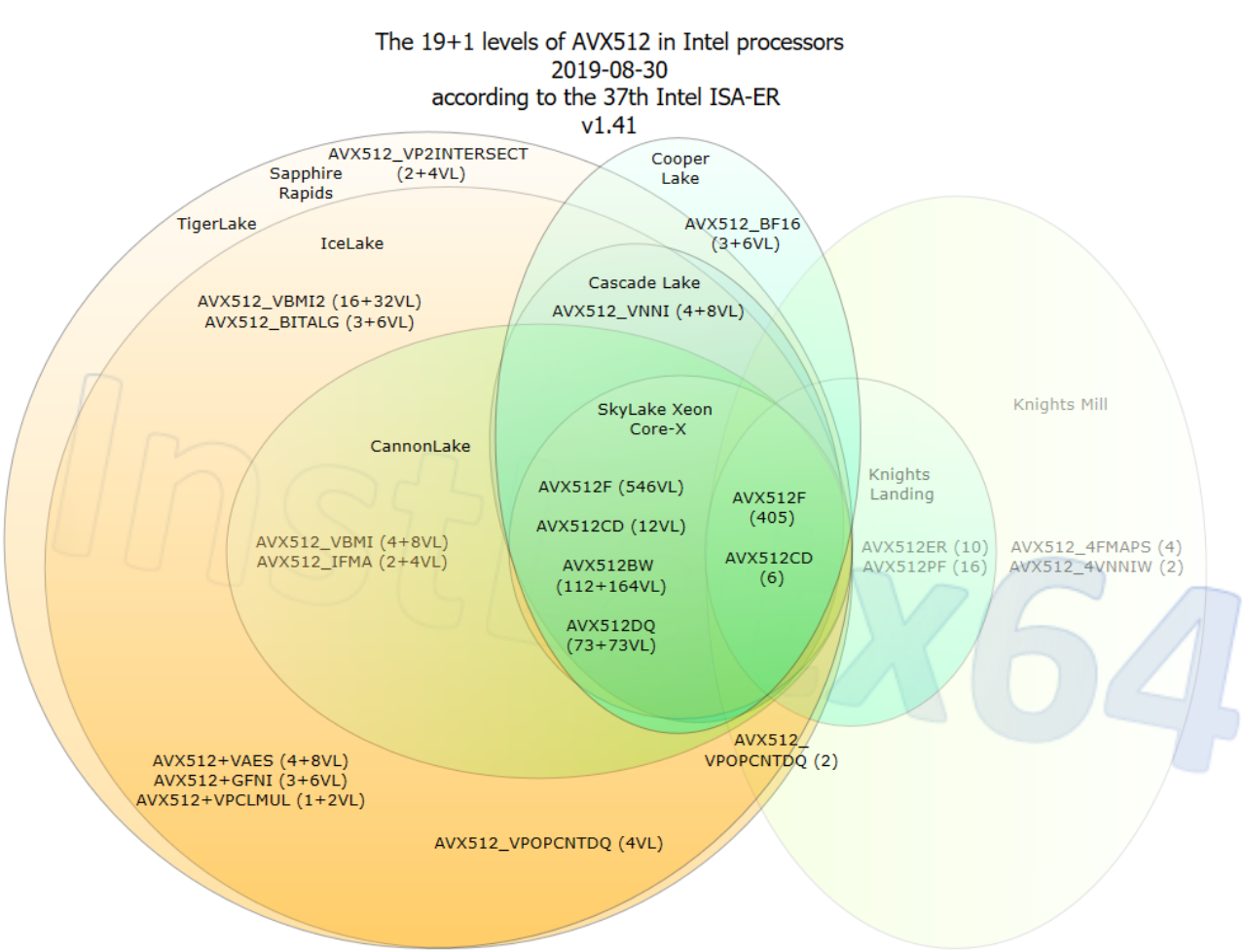
Instruction	Architecture			
	Ivy Bridge	Haswell	Broadwell	Skylake
_mm_load_pd	1; 1	1; 0.5	1; 0.5	1; 0.33
_mm_store_pd	1; 1	1; 0.5	1; 0.5	1; 0.33
_mm_add_pd	3; 1	3; 1	3; 1	4; 0.5
_mm_mul_pd	5; 1	5; 0.5	5; 0.5	3; 0.5
_mm256_load_pd	1; 1	1; 0.5	1; 0.5	1; 0.25
_mm256_store_pd	1; 1	1; 0.5	1; 0.5	1; 0.25
_mm256_stream_pd	-; 1	-; 1	-; 1	-; 1
_mm256_add_pd	3; 1	3; 1	3; 1	4; 0.5
_mm256_mul_pd	5; 1	5; 0.5	3; 0.5	4; 0.5
_mm256_fmadd_pd	-	5; 0.5	5; 0.5	4; 0.5
_mm256_loadu_pd	1; 1	1; 0.5	1; 0.5	1; 0.25
_mm256_storeu_pd	1; 1	1; 0.5	1; 0.5	1; 0.25

(source: Intel Intrinsics Guide)

AVX512 intrinsics variant

- for actual (Intel) CPUs analogous also a AVX512 variant can be created (check "flags" in "/proc/cpuinfo")
- AVX512: 8 double precision values are processed simultaneously.
- alignment: $8 \cdot 8 = 64$ byte
- **#include** <immintrin.h> (like AVX and AVX2)
- AVX512 double precision registers: __m512d va;
- prefix for the commands: "_mm512_"
- set values: va=_mm512_set_pd(a7,a6,a5,a4,a3,a2,a1,a0);

Intel CPUs: AVX512 levels & support



source:instlatx64.atw.hu

Excurs: Approximation of exp() function

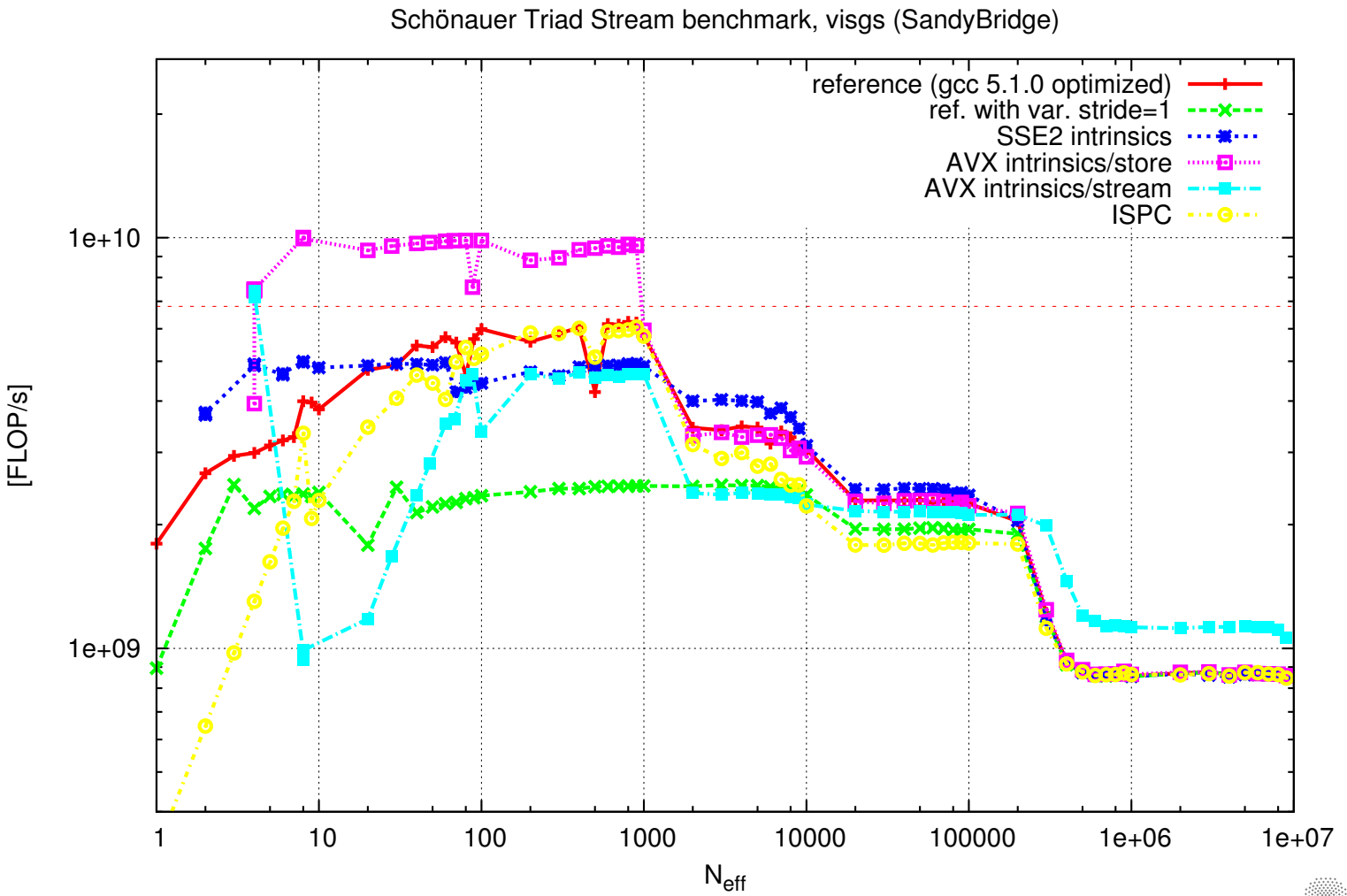
- a Taylor series can be used to approximate and replace a function $f(x)$: $f(x) = \lim_{o\to\infty} \sum_{i=0}^{o} \frac{f^{(i)}(x_0)}{n!}(x - x_0)^i$
- to approximate $\exp(x) = e^x$ with this polynomial, we get $exp(x) \approx \sum_{i=0}^{o} \frac{\exp(x_0)}{i!}(x - x_0)^i$ and for the Maclaurin series ($x_0 = 0$) $exp(x) \approx \sum_{i=0}^{o} \frac{1}{i!}x^i$
- o can be reduced using multiple x_0 (e.g. based on the exponent of x)
- the sum can be calculated using FMA within a loop $sum+ = a_i \cdot \tilde{x}^i$ (Horner's method: $sum = sum \cdot \tilde{x} + a_{o-i}$), where the factors a_i can also be pre-computed.
- SIMD units of width w can be used with loop unrolling and calculating partial sums first.
  (A $O(\log(w))$ algorithm to reduce the final sum of w partial sums might be inefficient for small w.)
- use of SIMD calculating $exp(x_i)$ for each element of a vector?

ISPC variant

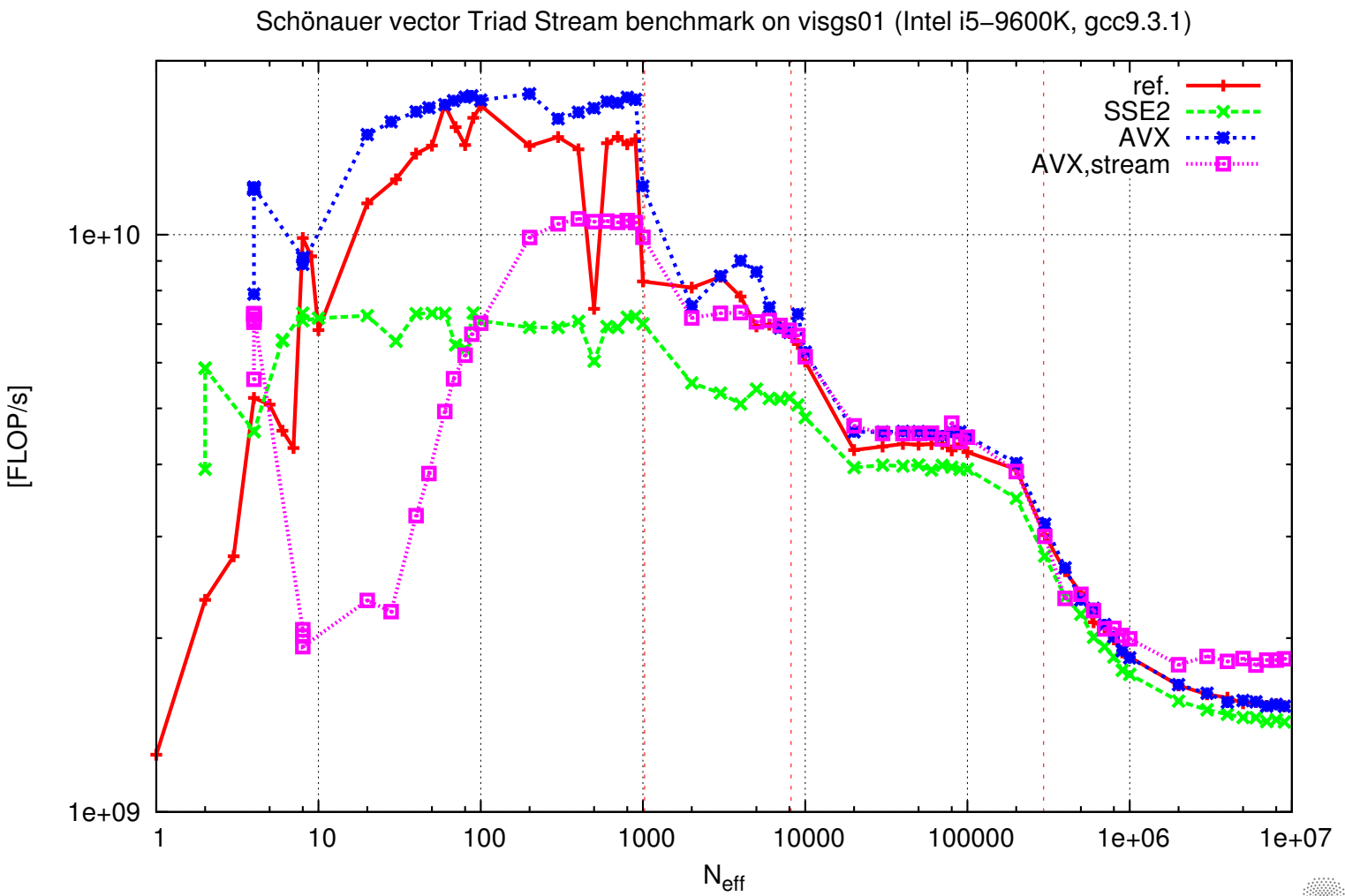
- as a starting point, a stream version is used, where the kernel is moved to a (external) function.
- to create a Intel SPMD Program Compiler version, this function is replaced with a ISPC version compiled with the ispc compiler:

```
export void stream(uniform const Tindex N
                  ,uniform Tfloat a[]
                  ,uniform const Tfloat b[]
                  ,uniform const Tfloat c[]
                  ,uniform const Tfloat d[]
                  ) {
    foreach (i = 0 ... N) {
        a[i]=b[i]*c[i]+d[i];
    }
}
```

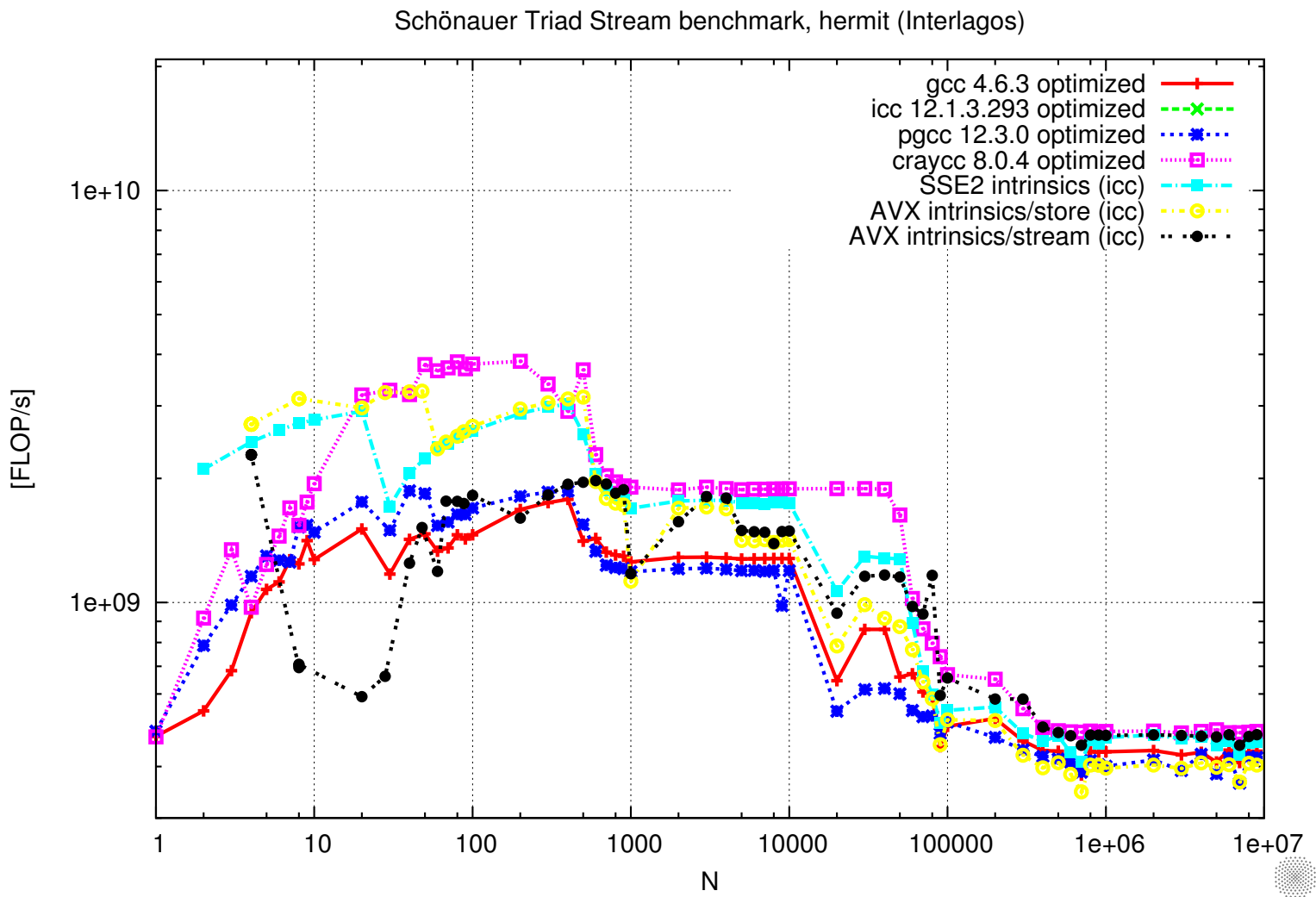
vTriad Stream Benchmark on a VIS-SgS-Computer



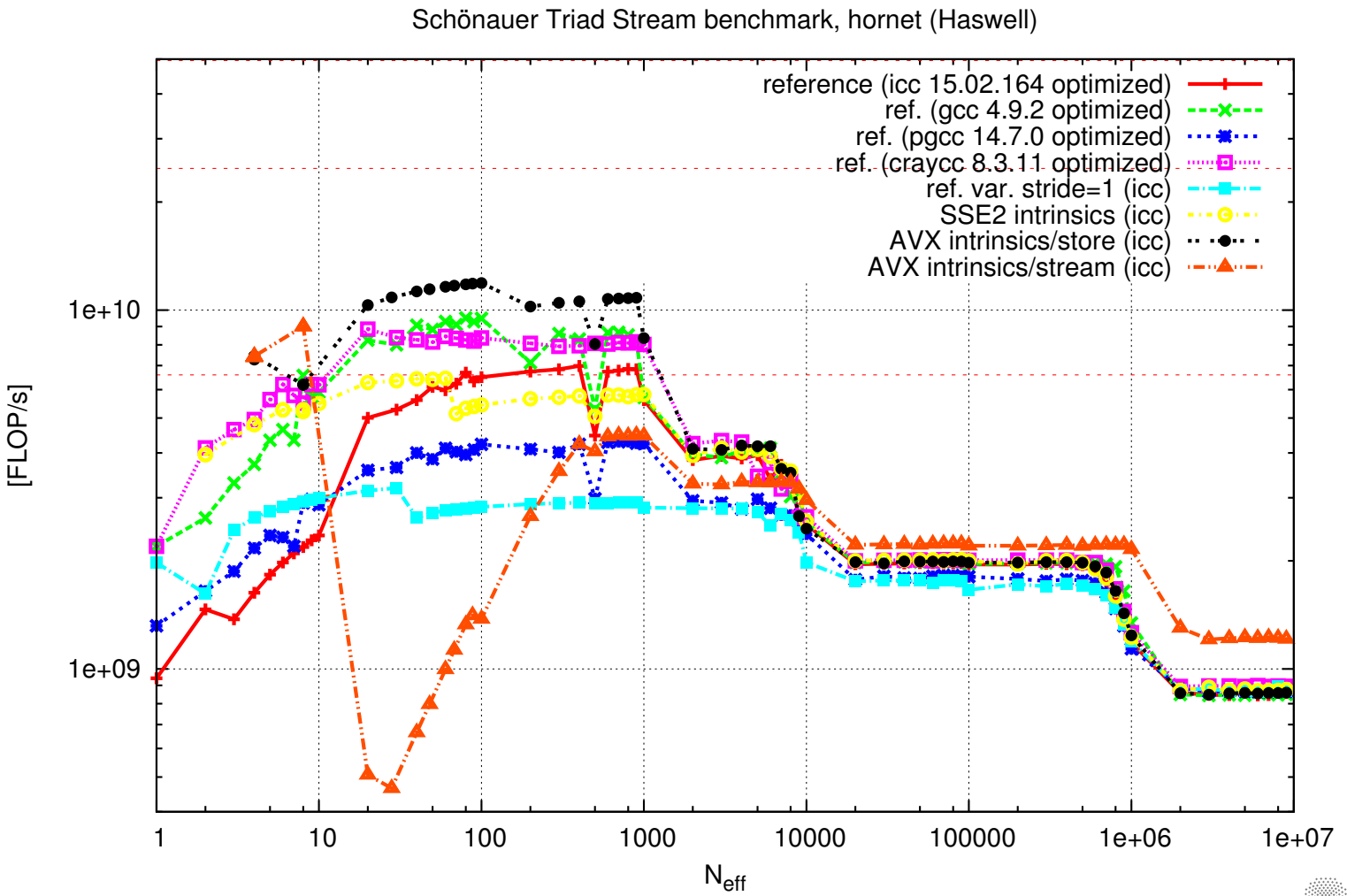
vTriad Stream Benchmark on a VIS-SgS-Computer



vTriad Stream Benchmark on Hermit



vTriad Stream Benchmark on Hornet/Hazelhen



objdump

- with the help of e.g. `objdump -d <objfile>` it's possible to take a look at the assembler code of the variants.
- setting the `-M intel-mnemonic` option, objdump will produce Intel instead of AT&T mnemonic code syntax.
- alternatively the gcc, clang, icc and pgcc can generate an assembler listing using the -S option
- the main program can be found in the "<main>" section, the computation part between the "gettimeofday" calls

x86_64 assembler

- a command consists of the
  1. instruction (Opcode)
  2. up to 2 operands
- Register
  - **G**eneral **P**urpose **R**egisters (64-Bit):
    RAX (Accumulator), RBX (Base register), RCX (Counter), RDX (Data register), RBP (Base-Pointer), RSI (Source-Index), RDI (Destination-Index), RSP (Stack-Pointer), R8,...,R15,...
  - additional 64-bit registers:
    RIP (Instruction-Pointer), RFLAGS (FLAGS, status),
    XMM0,...,XMM15 (SSE), YMM0,...,YMM15 (AVX),
    ZMM0,...,ZMM31 (AVX512)
  - e.g. for 64-bit registers rax,r8:
    eax,r8d address lowest 32 bits; ax,r8w lowest 16 bits;
    al,r8b lowest 8 bits (ah higher 8 bits of ax)

x86_64 assembler

- a command consists of the
  1. instruction (Opcode)
  2. up to 2 operands
- Registernames
  RAX,RBX,RCX,RDX,RBP,RSI,RDI,RSP,R8,...,R15,...,XMM0,...,XMM15,YMM0,...,YMM15,...
- assembly source file sections:
  Data, Block Started by Symbol (uninitialized data), Text (code)
- memory adressing (for a[i] or *(a+i)):
  $Base + Displacement + i * Size$
- **p**acked **d**ouble suffix: "vaddpd" and "vmulpd" e.g. denote addition- and multiplication SSE2/AVX instructions.
- can these AVX-instructions already be found in the standard reference variant? (using compiler optimization-options)

objdump: stream_ref vTriad computation part

```
4007f0:       e8 0b fe ff ff          call   400600 <gettimeofday@plt>
4007f5:       31 c0                   xor    eax,eax
4007f7:       31 d2                   xor    edx,edx
4007f9:       48 8b 4d 90             mov    rcx,QWORD PTR [rbp-0x70]


4007fd:       f2 41 0f 10 04 d7       movsd  xmm0,QWORD PTR [r15+rdx*8]
400803:       f2 0f 59 04 d1          mulsd  xmm0,QWORD PTR [rcx+rdx*8]
400808:       48 8b 4d 88             mov    rcx,QWORD PTR [rbp-0x78]
40080c:       f2 0f 58 04 d1          addsd  xmm0,QWORD PTR [rcx+rdx*8]
400811:       f2 41 0f 11 04 d6       movsd  QWORD PTR [r14+rdx*8],xmm0
400817:       48 ff c2                inc    rdx
40081a:       4c 39 ea                cmp    rdx,r13
40081d:       75 da                   jne    4007f9 <main+0x1a9>
40081f:       48 ff c0                inc    rax
400822:       49 39 c4                cmp    r12,rax
400825:       75 d0                   jne    4007f7 <main+0x1a7>
400827:       48 8d 7d c0             lea    rdi,[rbp-0x40]
40082b:       31 f6                   xor    esi,esi
40082d:       48 89 55 80             mov    QWORD PTR [rbp-0x80],rdx

400831:       e8 ca fd ff ff          call   400600 <gettimeofday@plt>
```

objdump: stream_sse2 vTriad computation part

```
40073e:       e8 cd fe ff ff          call   400610 <gettimeofday@plt>
400743:       31 c9                   xor    ecx,ecx
400745:       4c 89 e2                mov    rdx,r12
400748:       0f 1f 84 00 00 00 00    nop    DWORD PTR [rax+rax*1+0x0]
40074f:       00
400750:       48 8b 75 98             mov    rsi,QWORD PTR [rbp-0x68]
400754:       31 c0                   xor    eax,eax
400756:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
40075d:       00 00 00
400760:       c4 c1 79 28 04 c7       vmovapd xmm0,XMMWORD PTR [r15+rax*8]
400766:       c5 f9 59 04 c6          vmulpd xmm0,xmm0,XMMWORD PTR [rsi+rax*8]
40076b:       c4 c1 79 58 04 c6       vaddpd xmm0,xmm0,XMMWORD PTR [r14+rax*8]
400771:       c5 f8 29 04 c3          vmovaps XMMWORD PTR [rbx+rax*8],xmm0
400776:       48 83 c0 02             add    rax,0x2
40077a:       48 39 d0                cmp    rax,rdx
40077d:       75 e1                   jne    400760 <main+0x100>
40077f:       48 83 c1 01             add    rcx,0x1
400783:       49 39 cd                cmp    r13,rcx
400786:       75 c8                   jne    400750 <main+0xf0>
400788:       48 8d 7d c0             lea    rdi,[rbp-0x40]
40078c:       31 f6                   xor    esi,esi


40078e:       e8 7d fe ff ff          call   400610 <gettimeofday@plt>
```

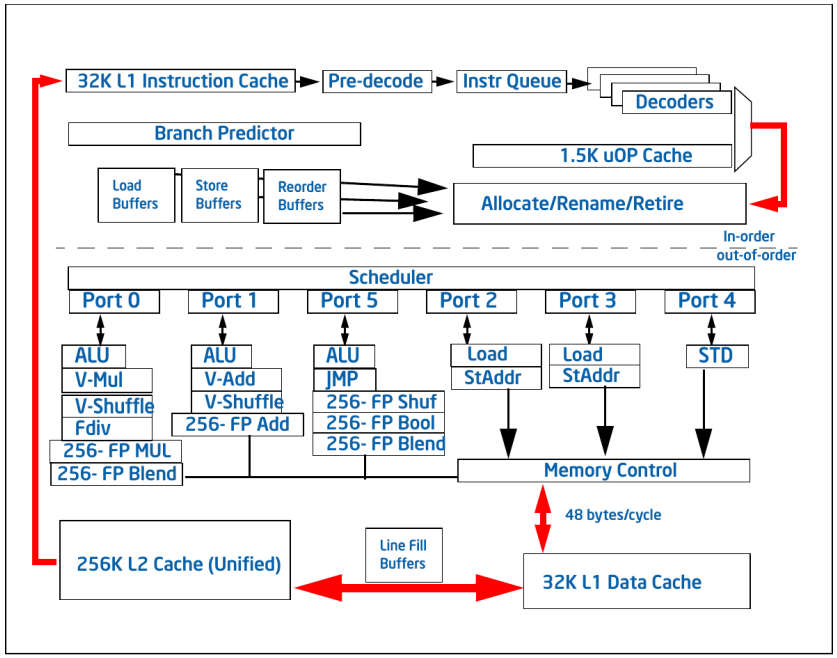
objdump: stream_avx vTriad computation part

```
40074d:       e8 be fe ff ff          call   400610 <gettimeofday@plt>
400752:       4c 8b 5d 90             mov    r11,QWORD PTR [rbp-0x70]
400756:       31 c9                   xor    ecx,ecx
400758:       4c 89 ea                mov    rdx,r13
40075b:       0f 1f 44 00 00          nop    DWORD PTR [rax+rax*1+0x0]

400760:       31 c0                   xor    eax,eax
400762:       66 0f 1f 44 00 00       nop    WORD PTR [rax+rax*1+0x0]

400768:       c4 c1 7d 28 04 c7       vmovapd ymm0,YMMWORD PTR [r15+rax*8]
40076e:       c4 c1 7d 59 04 c4       vmulpd ymm0,ymm0,YMMWORD PTR [r12+rax*8]
400774:       c4 c1 7d 58 04 c3       vaddpd ymm0,ymm0,YMMWORD PTR [r11+rax*8]
40077a:       c5 fd 29 04 c3          vmovapd YMMWORD PTR [rbx+rax*8],ymm0
40077f:       48 83 c0 04             add    rax,0x4
400783:       48 39 d0                cmp    rax,rdx
400786:       75 e0                   jne    400768 <main+0x108>
400788:       48 83 c1 01             add    rcx,0x1
40078c:       49 39 ce                cmp    r14,rcx
40078f:       75 cf                   jne    400760 <main+0x100>
400791:       48 8d 7d c0             lea    rdi,[rbp-0x40]
400795:       31 f6                   xor    esi,esi
400797:       4c 89 5d 88             mov    QWORD PTR [rbp-0x78],r11
40079b:       c5 f8 77                vzeroupper
40079e:       e8 6d fe ff ff          call   400610 <gettimeofday@plt>
```

Intel Architecture Code Analyzer

- the Intel Architecture Code Analyzer (IACA) perform a static analysis of the data dependencies, the throughput and the latency of binary code snippets.
- in the source code parts can be labeled through markers

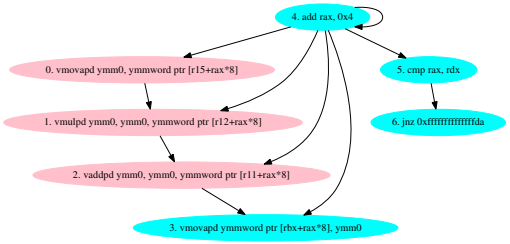
  (instrumentation of the source code)
- IACA shows the assignment of kernel instructions to processor ports and determine the critical path



Intel® 64 and IA-32 Architectures Optimization Reference Manual, S. 2-4 (42), Figure 2-1

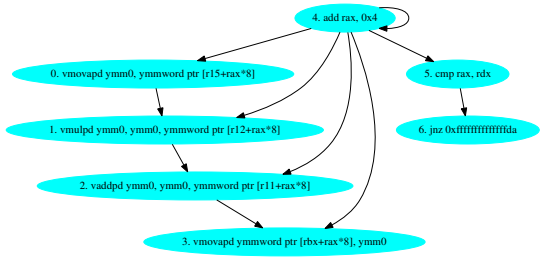
IACA: stream_avx vTriad comp. part (gcc -O3; intel Sandy Bridge)

```
$ iaca -64 -arch SNB stream_avx.iaca
Intel(R) Architecture Code Analyzer Version - 2.2 build:1aef335 (Wed, 28 Dec 2016 15:14:25 +0200)
Analyzed File - stream_avx.iaca
Binary Format - 64Bit
Architecture  - SNB
Analysis Type - Throughput

Throughput Analysis Report
--------------------------
Block Throughput: 3.00 Cycles       Throughput Bottleneck: Port2_DATA, Port3_DATA

Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------------------------
|  Port  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |
---------------------------------------------------------------------------------------
| Cycles | 1.0   0.0 | 1.0 | 2.0   3.0 | 2.0   3.0 | 2.0 | 2.0 |
---------------------------------------------------------------------------------------

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion happened
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256/AVX512 instruction, dozens of cycles penalty is expected
X - instruction not supported, was not accounted in Analysis

| Num Of |                    Ports pressure in cycles                     |     |
|  Uops  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |     |
---------------------------------------------------------------------------------------
|   1    |           |     | 0.5   1.0 | 0.5   1.0 |     |     | CP | vmovapd ymm0, ymmword ptr [r15+rax*8]
|   2    | 1.0       |     | 0.5   1.0 | 0.5   1.0 |     |     | CP | vmulpd ymm0, ymm0, ymmword ptr [r12+rax*8]
|   2    |           | 1.0 | 0.5   1.0 | 0.5   1.0 |     |     | CP | vaddpd ymm0, ymm0, ymmword ptr [r11+rax*8]
|   2    |           |     | 0.5       | 0.5       | 2.0 |     |    | vmovapd ymmword ptr [rbx+rax*8], ymm0
|   1    |           |     |           |           |     | 1.0 |    | add rax, 0x4
|   1    |           |     |           |           |     | 1.0 |    | cmp rax, rdx
|   0F   |           |     |           |           |     |     |    | jnz 0xffffffffffffffda
Total Num Of Uops: 9
```



IACA: stream_avx vTriad comp. part (gcc -O3; intel Haswell)

```
$ iaca -64 -arch HSW stream_avx.iaca
Intel(R) Architecture Code Analyzer Version - 2.2 build:1aef335 (Wed, 28 Dec 2016 15:14:25 +0200)
Analyzed File - stream_avx.iaca
Binary Format - 64Bit
Architecture  - HSW
Analysis Type - Throughput

Throughput Analysis Report
--------------------------

Block Throughput: 2.25 Cycles       Throughput Bottleneck: FrontEnd

Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------------------------
|  Port  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |  6  |  7  |
---------------------------------------------------------------------------------------
| Cycles | 1.0   0.0 | 1.0 | 2.0   1.5 | 2.0   1.5 | 1.0 | 1.0 | 1.0 | 0.0 |
---------------------------------------------------------------------------------------

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion happened
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256/AVX512 instruction, dozens of cycles penalty is expected
X - instruction not supported, was not accounted in Analysis

| Num Of |                    Ports pressure in cycles                     |     |
|  Uops  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |  6  |  7  |     |
---------------------------------------------------------------------------------------
|   1    |           |     | 0.5   0.5 | 0.5   0.5 |     |     |     |     | vmovapd ymm0, ymmword ptr [r15+rax*8]
|   2    | 1.0       |     | 0.5   0.5 | 0.5   0.5 |     |     |     |     | vmulpd ymm0, ymm0, ymmword ptr [r12+r
|   2    |           | 1.0 | 0.5   0.5 | 0.5   0.5 |     |     |     |     | vaddpd ymm0, ymm0, ymmword ptr [r11+r
|   2    |           |     | 0.5       | 0.5       | 1.0 |     |     |     | vmovapd ymmword ptr [rbx+rax*8], ymm0
|   1    |           |     |           |           |     | 1.0 |     |     | add rax, 0x4
|   1    |           |     |           |           |     |     | 1.0 |     | cmp rax, rdx
|   0F   |           |     |           |           |     |     |     |     | jnz 0xffffffffffffffda
Total Num Of Uops: 9
```



IACA: stream vTriad comp. part (gcc -O3; intel Skylake)

```
$ iaca -arch SKL -trace iaca_stream.trace stream
Intel(R) Architecture Code Analyzer Version -  v3.0-28-g1ba2cbb build date: 2017-10-23;16:42:45
Analyzed File -  stream
Binary Format - 64Bit
Architecture  -  SKL
Analysis Type - Throughput

Throughput Analysis Report
--------------------------
Block Throughput: 2.00 Cycles       Throughput Bottleneck: Dependency chains
Loop Count:  22
Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------------------------
|  Port  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |  6  |  7  |
---------------------------------------------------------------------------------------
| Cycles | 0.5   0.0 | 0.5 | 2.0   2.0 | 2.0   1.0 | 1.0 | 0.5 | 0.5 | 0.0 |
---------------------------------------------------------------------------------------

DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3)
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion occurred
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256/AVX512 instruction, dozens of cycles penalty is expected
X - instruction not supported, was not accounted in Analysis

| Num Of |                    Ports pressure in cycles                     |     |
|  Uops  | 0  - DV | 1  | 2  -  D  | 3  -  D  | 4  | 5  | 6  | 7  |
---------------------------------------------------------------------------------------
|   1    |         |    | 1.0  1.0 |          |    |    |    |    | vmovsd xmm0, qword ptr [r14+rax*8]
|   1    |         |    |          | 1.0  1.0 |    |    |    |    | vmovsd xmm3, qword ptr [r12+rax*8]
|   2    | 0.5     | 0.5| 1.0  1.0 |          |    |    |    |    | vfmadd132sd xmm0, xmm3, qword ptr [r13+rax*8]
|   2    |         |    |          | 1.0      | 1.0|    |    |    | vmovsd qword ptr [r15+rax*8], xmm0
|   1    |         |    |          |          |    | 0.5| 0.5|    | inc rax
|   1*   |         |    |          |          |    |    |    |    | cmp rax, rbp
|   0*F  |         |    |          |          |    |    |    |    | jb 0xffffffffffffffd9
Total Num Of Uops: 8
```

IACA trace: stream vTriad comp. part (gcc -O3; intel Skylake)

```
$ head -n 35 iaca_stream.trace | cut -c -99
it|in|Dissasembly                                 :0123456789012345678901234567890123456789012345678901
 0| 0|vmovsd xmm0, qword ptr [r14+rax*8]          :           |           |           |           |
 0| 0|     TYPE_LOAD (1 uops)                     :s----deeeew-----R---------p           |           |
 0| 1|vmovsd xmm3, qword ptr [r12+rax*8]          :           |           |           |           |
 0| 1|     TYPE_LOAD (1 uops)                     :s----deeeew-----R---------p           |           |
 0| 2|vfmadd132sd xmm0, xmm3, qword ptr [r13+rax*8] :           |           |           |           |
 0| 2|     TYPE_LOAD (1 uops)                     :s----cdeeeew-----R---------p           |           |
 0| 2|     TYPE_OP (1 uops)                       :A----------deeew-----R---------p       |           |
 0| 3|vmovsd qword ptr [r15+rax*8], xmm0          :           |           |           |           |
 0| 3|     TYPE_STOREDATA (1 uops)                : A----------dw-----R---------p         |           |
 0| 3|     TYPE_STOREADDRESS (1 uops)             : s----deeeew----------R---------p     |           |
 0| 4|inc rax                                     :           |           |           |           |
 0| 4|     TYPE_OP (1 uops)                       : sdw----------------R---------p       |           |
 0| 5|cmp rax, rbp                                :           |           |           |           |
 0| 5|     TYPE_OP (1 uops)                       : A-dw----------------R---------p      |           |
 0| 6|jb 0xffffffffffffffd9                       :           |           |           |           |
 0| 6|     TYPE_OP (0 uops)                       :  w----------------------R---------p  |           |
 1| 0|vmovsd xmm0, qword ptr [r14+rax*8]          :           |           |           |           |
 1| 0|     TYPE_LOAD (1 uops)                     :  As----deeeew----------R---------p   |           |
 1| 1|vmovsd xmm3, qword ptr [r12+rax*8]          :           |           |           |           |
 1| 1|     TYPE_LOAD (1 uops)                     :  As----deeeew----------R---------p   |           |
 1| 2|vfmadd132sd xmm0, xmm3, qword ptr [r13+rax*8] :           |           |           |           |
 1| 2|     TYPE_LOAD (1 uops)                     :  As----cdeeeew----------R---------p  |           |
 1| 2|     TYPE_OP (1 uops)                       :  A----------deeew-----R---------p    |           |
 1| 3|vmovsd qword ptr [r15+rax*8], xmm0          :           |           |           |           |
 1| 3|     TYPE_STOREDATA (1 uops)                :   A----------dw-----R---------p      |           |
 1| 3|     TYPE_STOREADDRESS (1 uops)             :   s----deeeew----------R---------p  |           |
 1| 4|inc rax                                     :           |           |           |           |
 1| 4|     TYPE_OP (1 uops)                       :   sdw----------------R---------p    |           |
 1| 5|cmp rax, rbp                                :           |           |           |           |
 1| 5|     TYPE_OP (1 uops)                       :   A-w----------------R---------p    |           |
 1| 6|jb 0xffffffffffffffd9                       :           |           |           |           |
 1| 6|     TYPE_OP (0 uops)                       :   w----------------------R---------p|           |
 2| 0|vmovsd xmm0, qword ptr [r14+rax*8]          :           |           |           |           |
 2| 0|     TYPE_LOAD (1 uops)                     :    As----deeeew----------R---------p  |           |
```

OSACA, LLVM-MCA

- alternatives to IACA (reaching EOL April 2019):
  - **O**pen **S**ource **A**rchitecture **C**ode **A**nalyzer
    (using kerncraft)
  - **L**ow **L**evel **V**irtual **M**achine - **M**achine **C**ode **A**nalyzer
  - also pmu-tools

ARM variant

- compile under specification of the architecture (on "CubieTower"):

```
gcc-4.7 -march=armv7 -mfpu=neon -O3 \
        -funsafe-math-optimizations -lm \
        stream_ref_float.c -o stream_ref_float
```

- check CPU frequency:

```
$ dmesg | grep freq
$ cd /sys/devices/system/cpu/cpu0/cpufreq
$ cat scaling_available_governors scaling_governor
$ echo 'performance' > scaling_governor
$ cat cpuinfo_cur_freq
$ echo 1008000 > scaling_setspeed
```

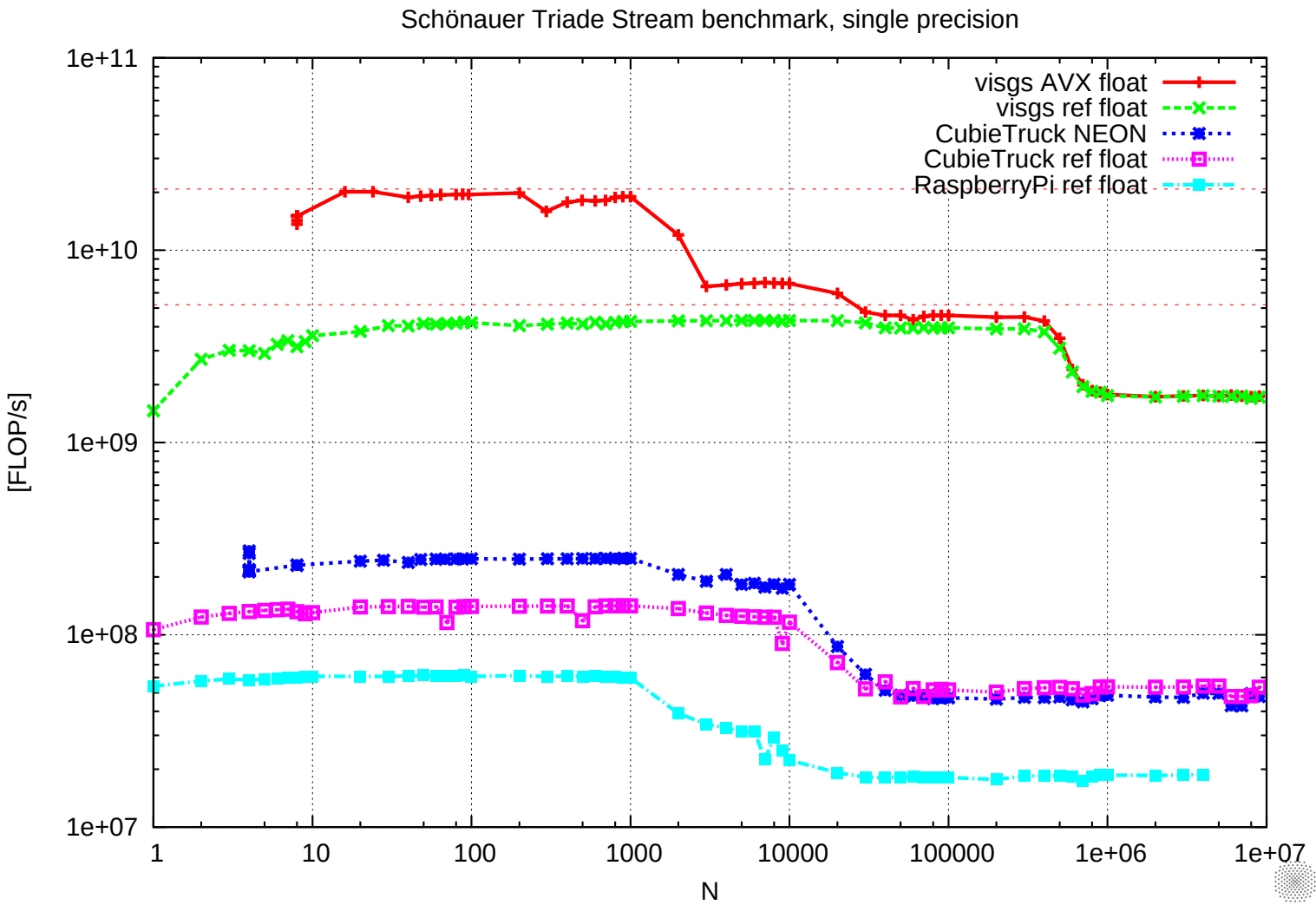
ARM NEON intrinsics variant (single precision)

- header file: **#include** <arm_neon.h>

SSE2	NEON
__m128 v;	float32x4_t v;
v=_mm_set_ps(s,s,s,s);	v=vdupq_n_f32(s)
v=_mm_set_ps(s0,s1,s2,s3);	v=vsetq_lane_f32(s0,v,0);
	v=vsetq_lane_f32(s1,v,1);
	v=vsetq_lane_f32(s2,v,2);
	v=vsetq_lane_f32(s3,v,3);
v=_mm_load_ps(p);	v=vld1q_f32(p);
_mm_store_ps(p,v);	vst1q_f32(p,v);
v=_mm_add_ps(va,vb);	v=vaddq_f32(va,vb);
v=_mm_mul_ps(va,vb);	v=vmulq_f32(va,vb);

(s. e.g. ARM® NEON™ Intrinsics Reference)

vTriad Stream Benchmark on a Cubietruck (SP)



PThread variant (1)

- introducing an array with elements of a structure, which contain the necessary data for each thread, e.g.

```
typedef struct {
 unsigned int p,numthreads; /* ThreadID, Number of threads */
 unsigned int N;            /* Number of elements */
 double *a,*b,*c,*d;        /* arrays */
} t_thread_work;
```

and in the main program

```
t_thread_work thread_work[NUMTHREADS];
for(p=0;p<NUMTHREADS;++p) {
 thread_work[p].N=N;
 thread_work[p].p=p;
 thread_work[p].numthreads=NUMTHREADS;
 thread_work[p].a=a; thread_work[p].b=b;
 thread_work[p].c=c; thread_work[p].d=d;
}
```

PThread variant (2)

- introducing an array with elements of a structure, which contain the necessary data for each thread
- function with a pointer to such a structure as parameter, which contains the core functionality, e.g.

```
void thread_stream(t_thread_work *w) {
 /* set workpackage for each thread separately,
    dependent on p,... */
 for(i=i1;i<i2;i+=di)
  w->a[i]=w->b[i]*w->c[i]+w->d[i];
}
```

PThread variant (3)

- introducing an array with elements of a structure, which contain the necessary data for each thread
- function with a pointer to such a structure as parameter, which contains the core functionality
- creating new threads, to let them execute this function threads will join again afterwards → barrier

```
for(p=0;p<NUMTHREADS;++p) {
  pthread_create(&thread[p],NULL
                 ,(void*(*)(void*))&thread_stream,(void*)&(thread_work[p]));
}
for(p=0;p<NUMTHREADS;++p) pthread_join(thread[p],NULL);
```

include the function & datatype definitions before through **#include** <pthread.h>

PThread variant (4)

- introducing an array with elements of a structure, which contain the necessary data for each thread
- function with a pointer to such a structure as parameter, which contains the core functionality
- creating new threads, to let them execute this function

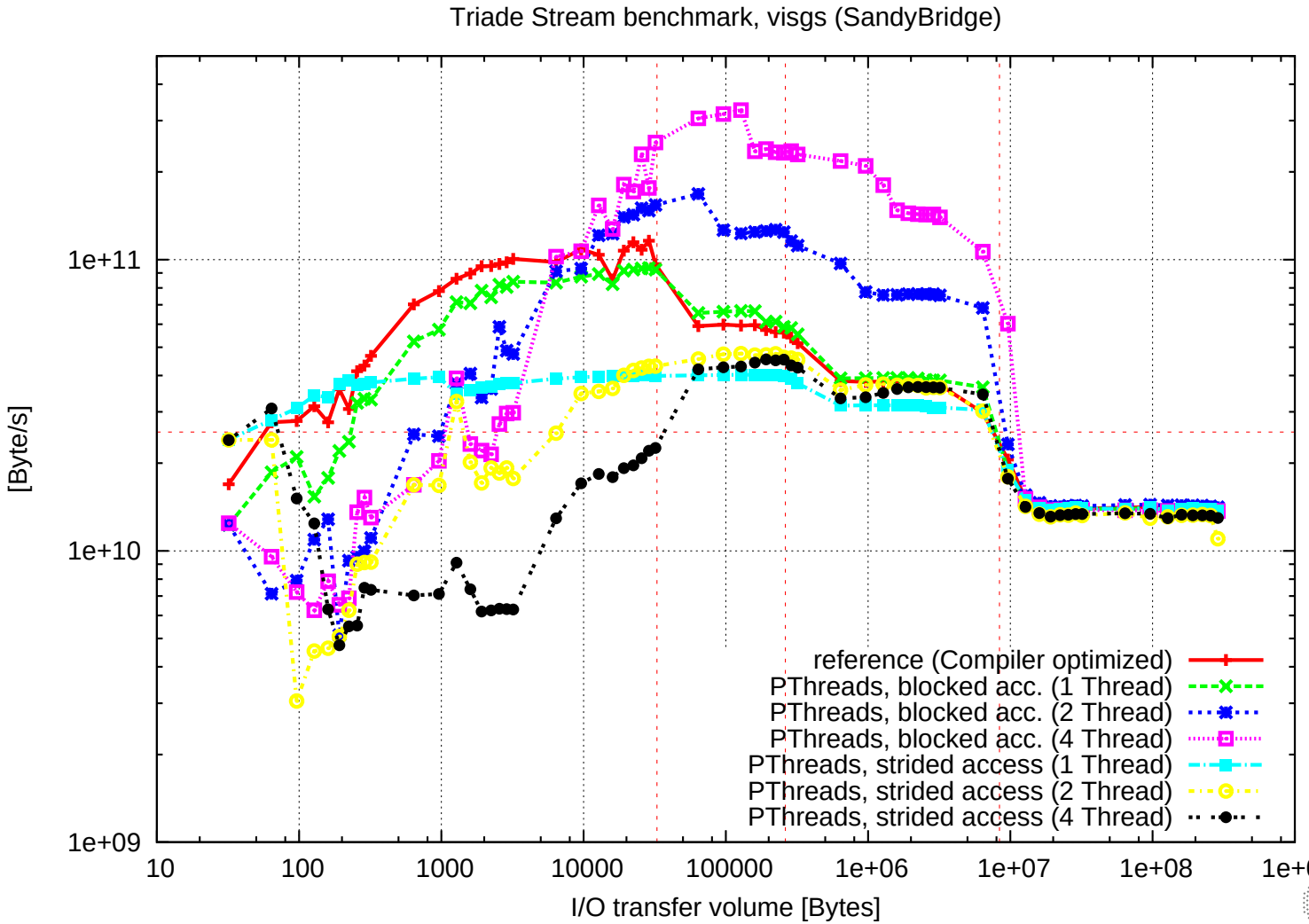
```
int pthread_create( pthread_t *thread, NULL
                  , void *(*start_routine)(void*), void *arg );
```

  threads will join again afterwards → barrier

```
int pthread_join( pthread_t thread, NULL );
```

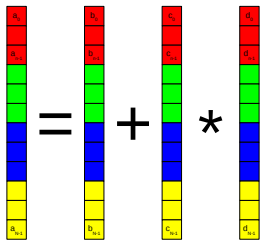
- translate with necessary options (e.g. gcc −pthread)
- test run with valgrind −−tool=helgrind (or −−tool=drd)
- PThreads can be combined with SIMD-variants
- optional: C++11 (or C+17) and JavaThreads variants (runtime comparison)

PThread vTriad Stream Benchmark @VIS-SgS



OpenMP variant

- initial OpenMP version starting from the reference-C-variant:
  - the most inner vector based loop is parallelized:
    #pragma omp **parallel for**



  - utilization of omp_get_wtime() to measure the time
    the usage of library functions requires to include the header file:

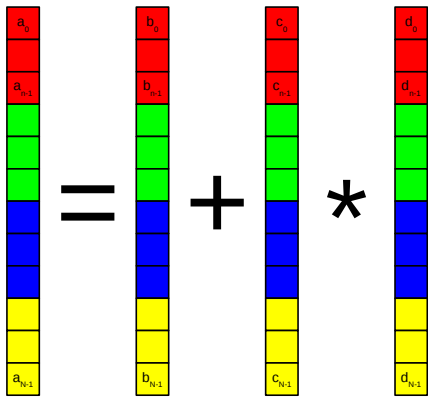
```
#ifdef _OPENMP
#include <omp.h>
#endif
```

  - compile with necessary options (e.g. gcc −fopenmp)
  - set environment variable OMP_NUM_THREADS

OpenMP variant (2)

- first modification:
  - parallel block including repetition loop:
    #pragma omp **parallel**

    (Which variables have to be declared as "private"? Synchronization? Barriers required?)
  - inner *i*-loop is executed collectively
    #pragma omp **for**

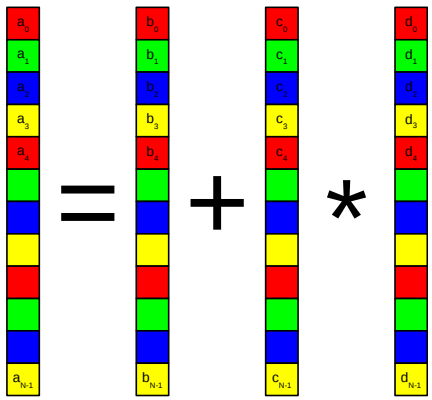


  - Impact of the "schedule" directive to the runtime?
  - "if" directive for the parallel block?

OpenMP variant (2)

- first modification: alternative version
  - parallel block including repetition loop:
    #pragma omp **parallel**

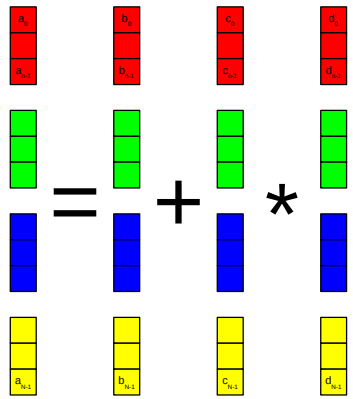
    (Which variables have to be declared as "private"? Synchronization? Barriers required?)
  - inner *i*-loop is executed collectively
  - each thread executes only a part of the loop iterations:
    if (i%ompnumthreads==ompthreadid) ... with
    int ompnumthreads=omp_get_num_threads();
    int ompthreadid=omp_get_thread_num();



  - Change of the runtime? Reasons?

OpenMP variant (3)

- second modification for NUMA systems (and to avoid "false share") using **#pragma** omp **for** again (e.g. to benchmark shared L3 Cache):
  - Data partitioning
    (each thread initializes and computes its own vectors)
  - Expand parallel region to the initialization
    (alternatively another parallel region can be used)
    → "first touch" by executing thread
    How do we measure the time now? → explicit barriers
  - output through a single thread (e.g. with
    if (omp_get_thread_num()==0)...,
    #pragma omp single, **#pragma** omp master)



- Thread-pinning (to prevent the OS to migrate thread execution to other Cores)
  - with tools like "taskset", "numactl", "likwid-pin",... or
  - defined by environment variables
    - GOMP_CPU_AFFINITY (gcc), KMP_AFFINITY (icc),...
    - OMP_PROC_BIND, OMP_PLACES (since OpenMP 4.0)

OpenMP SIMD variant

- additional SIMD optimization possible...
- OpenMP also supports SIMD vectorization (since version 4)
- the most inner loop is vectorized with:
  #pragma omp simd
- Vectorization can be combined with thread parallelization:
  #pragma omp **parallel for** simd

  (or just **#pragma** omp **for** simd within a parallel region)
- How close can we get to the peak performance?
  Is there a saturation of the memory bus using multiple cores?

Likwid-Bench

Likwid-Bench is part of the Likwid Performance Tools and a tool to perform streaming micro-benchmarks.

```
$ likwid-bench -a
```

will list the available benchmark kernels.

```
$ likwid-bench -l triad_avx
```

shows the properties of the triad_avx benchmark, and

```
$ likwid-bench -p
```

the available domains (CPU sockets, NUMA domains) Now e.g.

```
$ likwid-bench -t triad_avx -w S0:10kB
```

will e.g. run the Schönauer vector triad on the first socket (with one thread per core) for a working set size of 10 kB.

Part II

Tutorial: OpenMP

omp parallel

```
#include<stdio.h>
int main() {
#pragma omp parallel
 {
printf("1");
printf("2");
printf("3");
printf("4");
 } /* omp parallel */
printf("\n");
return 0;
}
```

- How about the printout of this simple OpenMP program? Is it deterministic? Are there constraints?

omp for schedule variants

```
#define N 100
#include<stdio.h>
#include<unistd.h> /* usleep() */
#include<omp.h> /* omp_get_thread_num() */
int main() {
  int i, t[N];
/* Experiment with...
 schedule(static), schedule(static,10),
 schedule(dynamic), schedule(dynamic,10),
 schedule(guided), schedule(guided,10),
 schedule(auto), schedule(nonmonotonic:dynamic),...
...and optionally with nested loops and collapse.
Alternatively use the OMP_SCHEDULE environment variable: */
#pragma omp parallel for schedule(runtime)
  for(i=0;i<N;++i) {
    t[i]=omp_get_thread_num();
    usleep(t[i]+i); /* wait for (t[i]+i)*1E-6 sec */
  }
  for(i=0;i<N;++i) printf("%d ",t[i]); printf("\n");
  return 0;
}
```

omp barrier overhead

```
#include<stdio.h> /* printf() */
int main() {
unsigned long nrepeat=100;
double ompwt_max_diff=0.;
#pragma omp parallel
 { unsigned long i;
#pragma omp barrier
double ompwt_start=omp_get_wtime(); /*-------------*/
for(i=0;i<nrepeat;++i)
{
/* should do some work here with well-known,separately-measured
   runtime to be subtracted at the end */
#pragma omp barrier
}
double ompwt_end=omp_get_wtime(); /*-------------*/
double ompwt_diff=ompwt_end-ompwt_start;
#pragma omp critical
if(ompwt_diff>ompwt_max_diff) ompwt_max_diff=ompwt_diff;
 } /* omp parallel */
printf("%lu\t%g\n",nrepeat,ompwt_max_diff/nrepeat);
return 0;
}
```

Calculation of π: A buggy version

- The following OpenMP program aims to compute π via a summation formula originating from Euler (solution for the Basel problem):
  $\pi = \sqrt{6 \cdot \sum_{i \in \mathbb{Z}^+} \frac{1}{i^2}}$

```
unsigned long n=125000;
double addend, pi;
unsigned long i;
#pragma omp parallel
{
  pi=0.;
#pragma omp for nowait
  for(i=1;i<=n;++i) {
    addend=1./(i*i);
    pi+=addend;
  }
  pi*=6.;
  pi=sqrt(pi);
} /* omp parallel */
  printf("pi = %g\n",pi);
```

- The program doesn't work correctly. Eliminate the "bugs"!

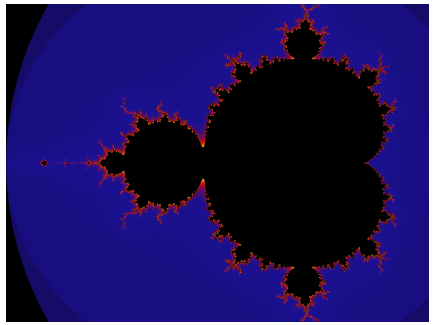
Calculation of π: A first correction

- After some changes the program looks like that:

```
unsigned long n=125000, i; double addend, pi;
#pragma omp parallel
{
#pragma omp master
  pi=0.;
#pragma omp for private(addend)
  for(i=1;i<=n;++i) {
    addend=1./i/i; /* possible overflow for (i*i) */
#pragma omp critical
    pi+=addend;
  } /* omp for (implicit barrier) */
#pragma omp single nowait
  pi=sqrt(6.*pi);
} /* omp parallel */
```

- Is the program correct now? Which changes are (un)necessary or (still) buggy?
  (If everything is correct(ed): How about Strong Equivalence to the sequential version?)
- Evaluate alternatives for `omp critical`.

Mandelbrot set



- Regarding the sequence of complex numbers

$$z_n = z_{n-1}^2 + c \quad with\ c, z \in \mathbb{C}$$

for certain c and starting numbers z_0 this sequences will be bounded. It will diverge, if $|z_n| > 2$

- The Mandelbrot set M is the union of all c with $z_0 = 0 + 0i$, where the sequence will not diverge:

$$c \in M \iff \limsup_{n \to \infty} |z_n| \leq 2$$

(For a given c, there's a Julia set including all numbers z_0, where the sequence will not diverge and therefore Mandelbrot (with variable c) and Julia sets (with variable z_0) might be seen as twodimensional "slices" of a common fourdimensional space.)

Mandelbrot set: implementation

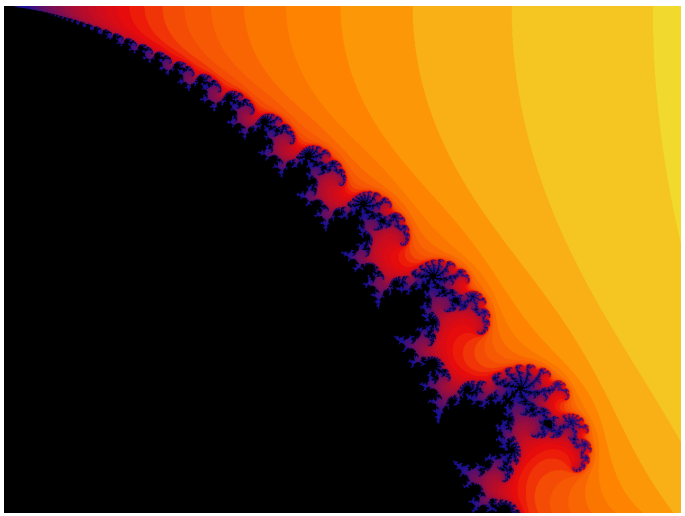
- An implementation to calculate the Mandelbrot (or Julia) set will check for a limited maximal iteration count, if the sequence diverges:

```
/* determine Creal,Cimg for pixel */
Zreal=Zimg=0.;
for (i=0;i<maxiterations;++i) {
  Zreal_n = Zreal*Zreal-Zimg*Zimg+Creal;
  Zimg_n  = 2*Zreal*Zimg+Cimg;
  if (Zreal_n*Zreal_n+Zimg_n*Zimg_n>2.*2.) break;
  Zreal=Zreal_n; Zimg=Zimg_n;
}
/* save i for pixel */
```

- the number of iterations until the sequence diverges will be stored for each c and form the basis for a (greyscale) image. (The image can also be colorized using e.g. gimp, ImageMagick,...).

Mandelbrot set: parallelization

- the calculations for each c ($< c_{real}, c_{img} > \rightarrow$ pixel) are embarrassingly parallel!
- use OpenMP to parallelize the given (sequential) code
  - omp **parallel for** to parallelize outer loop
    (also test collapse (2) −− is is applicable ?)
  - test different scheduling variants
    using schedule(runtime) and setting OMP_SCHEDULE, e.g.
    export OMP_SCHEDULE="guided,2"
  - You might also additionally use AVX (simd)



Matrix multiplication

- A matrix multiplication $C = A \cdot B$ should be parallized with OpenMP (BLAS3 $\rightarrow$ PBLAS3).
- The given test program initializes the matrices A and B with fixed values. To read the two matrices stored e.g. in Matrix Market Exchange Format files, the ANSI C library for Matrix Market I/O could be used. (MatrixMarket offers a selection of matrices & generators.)
- the (first) algorithm should calculate the results C_{ij} in parallel. (parallel calculation of the correspondent sum elements might be a further step to tackle)
- if in the most inner loop the expression $(i + j + k)\%n2$ instead of the loop interator k is used, this corresponds to the EREW-PRAM algorithm of the lecture
- for a cache-optimized memory access the matrix B should be stored in column order. A row-oriented storage of the elements of B will induce a strided memory access.

A test with tasks

```
#include<stdio.h>

int main() {
#pragma omp parallel
 {
#pragma omp single
  {
  printf("1");
#pragma omp task
  printf("2");
#pragma omp task
  printf("3");
/*#pragma omp taskwait*/
  printf("4");
  } /* omp single */
 } /* omp parallel */
 printf("\n");
 return 0;
}
```

- How does the printout look like?
- Why is there a
  omp single (or likewise omp master) block
  spanning the whole omp **parallel** region?
  (alternative: omp **parallel** sections with a single omp section)
  What happens without this construct?
- Are there any changes if the
  omp taskwait is uncommented?

Task-based recursive Fibonacci function with errors

```
unsigned long fibonacci(unsigned int n) {
 if (n<=1) {
  return (unsigned long)n;
 } else {
#pragma omp task
  unsigned long f1=fibonacci(n-1);
#pragma omp task
  unsigned long f2=fibonacci(n-2);

  return f1+f2;
 }
}
```

- This ficonacci function initially is called once from a parallel region in main.
- Running as a sequential program with correct results, the program doesn't even compile with OpenMP...
  Correct the function!
- Optimize the efficiency (for this parallel recursive implementation)

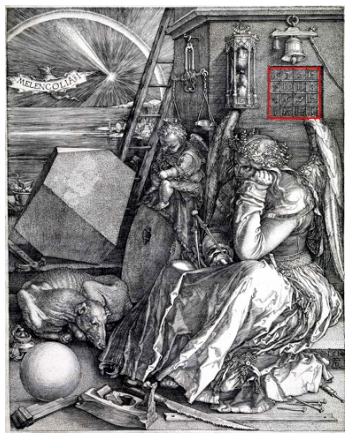
Quicksort

- Steps for a OpenMP parallelization:
  1. a parallel region contains the initial call of the quicksort function, which should however be executed only once (using e.g. omp single or omp master)
  2. further recursive calls will be executed in parallel through omp task
- Increase of efficiency, by sorting sequentially for smaller array sizes. Try...
  - C **if**-statement, which selects between a (sequential) quicksort and OpenMP-extented quicksort call
  - **if** extension of the omp task construct
  - final extension (since OpenMP 3.1)

Magic Square

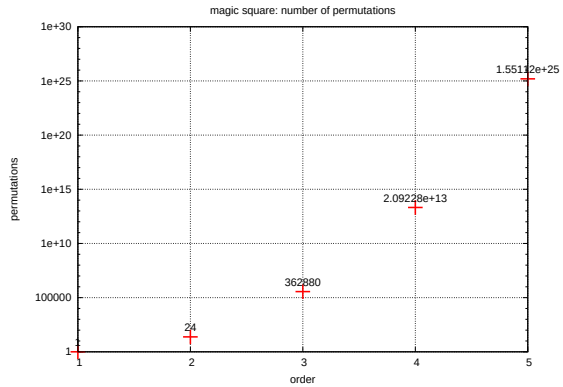
- for a magic square of order o the $n = o^2$ integer values $1, 2, \ldots, n$ will be put in place in such a way, that there's the same sum $\frac{o(o^2+1)}{2}$ for all rows, columns and diagonals.

  (well-know example: Dürer's magic square of order 4 within his Melencolia I)



- there exist methods to construct a magic square for odd or doubly-even (divisible by four) o

Magic Square: algorithm



- out of the possible $n!$ permutations of the $o \cdot o = n$ matrix values, the magic squares should be filtered out.
  ($o^2!$ tests with $2o + 2$ sums for o elements necessary?)
- an algorithm to create permuations can be set up recursively, by iterating over all elements for the first positions and run through all permutions of the remaining elements for the following positions. A recursion branch can be cancelled, if a condition (e.g. row sum) is not satisfied for all (from the beginning).
- an optimized algorithm might also take advantage of symmetry (rotational(+3) and reflection(+1+1+1+1) $\Rightarrow$ factor 8)
- an efficient sequential algorithm should be the starting point for an OpenMP parallelization
- parallize with OpenMP tasks and make a private copy of data where needed

A test with dependend tasks

```
#include<stdio.h>

int main() {
int i1=1,i2,i3;
#pragma omp parallel sections
 {{
#pragma omp task
printf("1");
#pragma omp task depend(in:i1) depend(out:i2)
  {
printf("2");
i2=i1+2;
  }
#pragma omp task depend(inout:i1)
  {
i1++;
printf("3");
  }
#pragma omp task depend(in:i2) depend(out:i3)
  {
i3=i2+1;
printf("4");
  }
 }} /* omp parallel sections */
printf("\n%d%d%d\n",i1,i2,i3);
return 0;
}
```

- Can the third task be executed before the second one?
- Is the result of i1, i2 and i3 deterministic? (Why?)
- What is printed?
  (List all possibilities.)