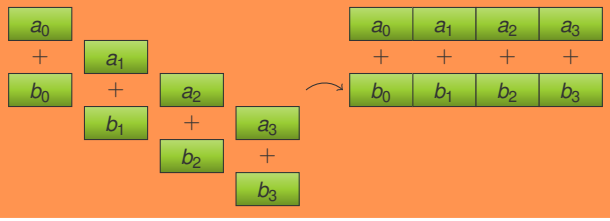
SIMD

Single Instruction Multiple Data

You program *one instruction stream*.
It is executed on *more than one datum* at the same time.



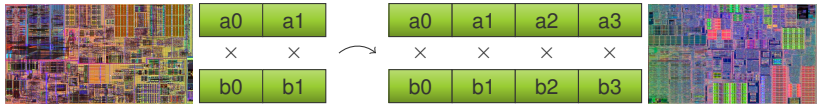
- Less transistors (⇒ power) for more Flops
- Different implementations exist:
  - SIMD registers with N bytes ⇒ stores N/sizeof(T) values
  - Instruction decoder feeds several ALUs in parallel

SIMD for Real

- 64 bit: x86: MMX
- 128 bit:
  - x86: SSE, SSE2, SSE3, SSSE3, SSE4a, SSE4.1, SSE4.2
  - Power: AltiVec / Velocity Engine / VMX
  - ARM: NEON
- 256 bit: x86: AVX, FMA4, XOP (, AVX2)
- 512 bit: part of the AVX spec
- 1024 bit: part of the AVX spec
- GPUs

SIMD in the future

wider vectors and *more instructions*
lead to highly efficient SIMD code
at relatively *low hardware costs*



I thought I am using it already?

If you made use of it you'd know.

- We read
  - OS abc now supports AVX
  - Compiler xyz now supports AVX
- If you allow the compiler (via flags), it will try to auto-vectorize
- Auto-vectorization is a very hard problem
- Only count on it when you know what you're doing
- When you know that already why not make it explicit?

Synchronous Parallel

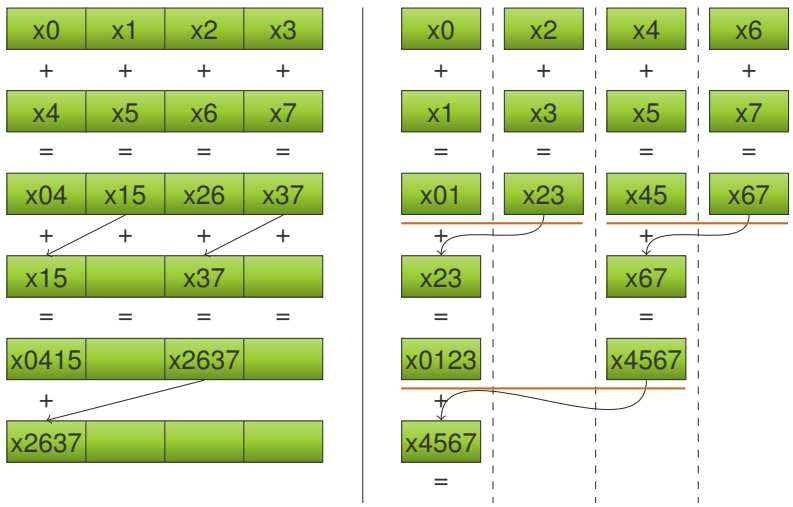
Compare with multithreading

- exchanging data between threads is non-trivial
- you can not know the progress of other threads except at explicit synchronization points
- if you forget to place a synchronization point your code might exhibit races

SIMD parallelism is synchronously parallel

Imagine N threads running in perfect sync. . .

Example: Summation



Vc to the rescue

Vc is a zero-overhead wrapper around intrinsics
with greatly improved syntax and semantics

- recognizable types:
  `float_v double_v int_v ushort_v` . . .
- infix operators:
  `a + b * b`
- masks and value vectors are different types:
  `float_m mask = a < b;`
- masked ops syntax:
  `a(a < b) += b`

Marketing Speak

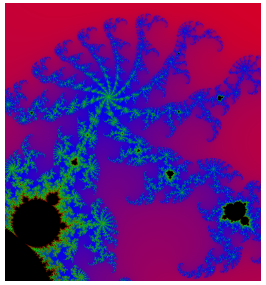
Vc is intuitive, portable, and fast!

Example: Mandelbrot

Mandelbrot Iteration

$z_{n+1} = z_n^2 + c$

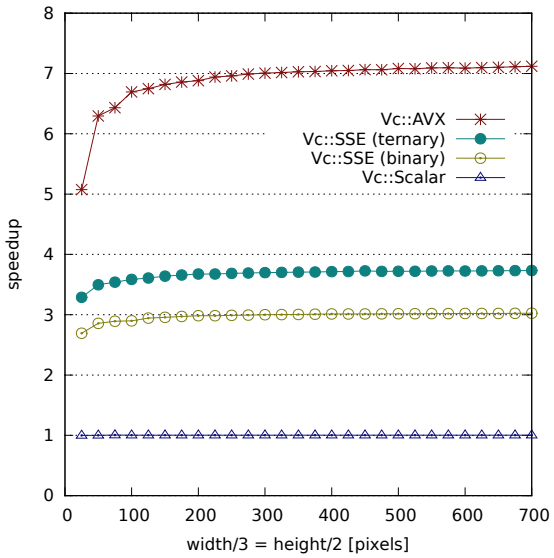
- $z_0 = 0$ and $c =$ canvas coordinates
- Iterate until either $|z_n|^2 \geq 4$ or $n > n_{max}$
- n determines color



Mandelbrot with Vc

```
typedef std::complex<float_v> Z;
for (int y = 0; y < imageHeight; ++y) {
  const float_v c_imag = y0 + y * scale;
  for (int_v x = int_v::IndexesFromZero();
       !(x < imageWidth).isEmpty(); x += float_v::Size) {
    Z c(x0 + static_cast<float_v>(x) * scale, c_imag);
    Z z = c;
    int_v n = int::Zero();
    int_m inside = fastNorm(z) < S;
    while (!(inside && n < maxIt).isEmpty()) {
      z = z * z + c;
      ++n(inside);
      inside = fastNorm(z) < S;
    }
    int_v colorValue = (maxIt - n) * 255 / maxIt;
    int maxJ = min(int_v::Size, imageWidth - x[0]);
    for (int j = 0; j < maxJ; ++j) {
      colorizeNextPixel(colorValue[j]);
    }
  }
}
```


Mandelbrot Speedup



The Performance Monitoring Unit (PMU)

- a few special registers to count selected events
- interrupt at a selected value
- kernel (module) can note the instruction pointer
- sampling of many different events requires multiplexing

⇒ very precise (not 100%) sampling down to the Assembly level

Tools

- VTune: probably best known tool for the PMU
- perf: bundled with the Linux kernel

Incomplete List of Events

- MEM_INST_RETIRED.{LOADS,STORES}
- MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM
- MEM_UNCORE_RETIRED.REMOTE_CACHE_LOCAL_HOME_HIT
- MEM_UNCORE_RETIRED.{REMOTE,LOCAL}_DRAM
- FP_COMP_OPS_EXE.X87
- FP_COMP_OPS_EXE.MMX
- FP_COMP_OPS_EXE.SSE_FP
- FP_COMP_OPS_EXE.SSE2_INTEGER
- FP_COMP_OPS_EXE.SSE_FP_{PACKED,SCALAR}
- FP_COMP_OPS_EXE.SSE_{SINGLE,DOUBLE}_PRECISION
- SSEX_UOPS_RETIRED.{PACKED,SCALAR}_{SINGLE,DOUBLE}
- BR_INST_EXEC.*
- BR_MISP_EXEC.*
- MEM_LOAD_RETIRED.{L1D,L2,L3_UNSHARED}_HIT ...
- L2_RQSTS.{LOADS,IFETCHES,MISS,REFERENCES}
- L2_DATA_RQSTS.*
- L2_WRITE.*
- L2_LINES_IN.*
- L2_LINES_OUT.*
- UNC_L3_HITS.*
- UNC_L3_MISS.*
- UNC_L3_LINES_IN.*
- UNC_L3_LINES_OUT.*

SIMD Example

```
enum { N = 10000 };

float x[N];
for (int i = 0; i < N; ++i)
  x[i] = i;

for (int n = N - 1; n > 0; --n)
  for (int i = 0; i < n; ++i)
    x[i] += x[i + 1];
```

$$N_{cvt} = 10000$$

$$N_{add} = \sum_{k=1}^{9999} k = \frac{9999 \cdot 9998}{2} = 49\,985\,001 \approx 50e6$$

$$\frac{N_{add}}{4} \approx 12.5e6$$

Result

```
mkretz@Trigati% profile.sh SIMD ./test-O2
107,760,320 cycles
212,761,004 instructions     1.97  insns per cycle
 50,008,232 FP_COMP_OPS_EXE.SSE_FP
          0 FP_COMP_OPS_EXE.SSE_FP_PACKED
 50,009,426 FP_COMP_OPS_EXE.SSE_FP_SCALAR
 49,999,156 FP_COMP_OPS_EXE.SSE_SINGLE_PRECISION
     10,270 FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION
          0 SIMD_INT_128.PACKED_ARITH
        370 SIMD_INT_128.SHUFFLE_MOVE
          1 SSEX_UOPS_RETIRED.PACKED_SINGLE
249,972,513 SSEX_UOPS_RETIRED.SCALAR_SINGLE
          0 SSEX_UOPS_RETIRED.PACKED_DOUBLE
          0 SSEX_UOPS_RETIRED.SCALAR_DOUBLE
         39 SSEX_UOPS_RETIRED.VECTOR_INTEGER
```


Result when Auto-Vectorized

```
mkretz@Trigati% profile.sh SIMD ./test-O3
 44,128,560 cycles
 69,386,657 instructions     1.57  insns per cycle
 12,513,484 FP_COMP_OPS_EXE.SSE_FP
 12,500,449 FP_COMP_OPS_EXE.SSE_FP_PACKED
     15,178 FP_COMP_OPS_EXE.SSE_FP_SCALAR
 12,513,019 FP_COMP_OPS_EXE.SSE_SINGLE_PRECISION
      2,608 FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION
      2,666 SIMD_INT_128.PACKED_ARITH
      2,680 SIMD_INT_128.SHUFFLE_MOVE
 24,992,501 SSEX_UOPS_RETIRED.PACKED_SINGLE
     67,500 SSEX_UOPS_RETIRED.SCALAR_SINGLE
          0 SSEX_UOPS_RETIRED.PACKED_DOUBLE
          0 SSEX_UOPS_RETIRED.SCALAR_DOUBLE
      4,726 SSEX_UOPS_RETIRED.VECTOR_INTEGER
```

L1 Cache Results

```
mkretz@Trigati% profile.sh L1-load-store ./test-O3
 44,656,800 cycles
 69,386,652 instructions
 37,530,486 L1-dcache-loads
    895,304 L1-dcache-load-misses 2.39% of all L1-dcache hits
 12,522,243 L1-dcache-stores
        337 L1-dcache-store-misses

mkretz@Trigati% profile.sh L1-load-store ./test-O2
103,356,861 cycles
212,760,993 instructions
100,013,036 L1-dcache-loads
    824,845 L1-dcache-load-misses 0.82% of all L1-dcache hits
 50,014,795 L1-dcache-stores
        344 L1-dcache-store-misses
```


Branch Prediction Results

```
mkretz@Trigati% profile.sh branches ./test-O3
 43,496,620 cycles
 69,386,651 instructions
  1,705,984 branches
     17,346 branch-misses         1.02% of all branches

mkretz@Trigati% profile.sh branches ./test-O2
112,102,023 cycles
212,760,956 instructions
  6,334,488 branches
     15,366 branch-misses         0.24% of all branches
```


Typical Bottlenecks

- compute bound
- memory (bandwidth) bound
- I/O bound (communication, storage)

Example: Matrix-Matrix multiply (DGEMM/SGEMM)

- every naïve implementation will be mem bound
- fully optimized implementation is compute bound

Strategy for Memory-Bound Code

- blocking
- streaming stores
- prefetching
- non-temporal prefetching
- substitute memory accesses with computation

Releases

- 0.5 done
- 0.6 (AVX support) released *today*
- 0.7 aims for much better ICC and MSVC support and hopefully MIC support
- ROOT. . . still an open task

Compiler Support

- GCC support is very good
- ICC
  - less errors (incompatibilities or miscompiles)
  - still some very bad performance issues
    (e.g. Mandelbrot Vc::SSE slower than Vc::Scalar)
- MSVC
  - I recently ported all of Vc to latest MSVC on WinXP (32bit)
  - 64-bit support probably broken
  - AVX support certainly broken
- Clang (LLVM)
  - on my list to test
  - I never used clang before (what a shame)