Vectorization

Oliver Heidmann

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

2016-12-22

Structure

The problem

The Program:
Simulation/Game/Analytics which processes huge amounts of data.

The Problem:
The execution time is too high.
From benchmarks we see that we are not using
100% of our CPU capability.
What can we do?

What we can do

- manual optimizations
- parallelization

But we still are below our CPU capabilities.

What we forgot

- manual optimizations
- parallelization
- => vectorization <=

Flops calculation:
#Cores * Clock * #Operations per Cycle * VectorSize/32bit * 2

Vectorization

What is Vectorization?

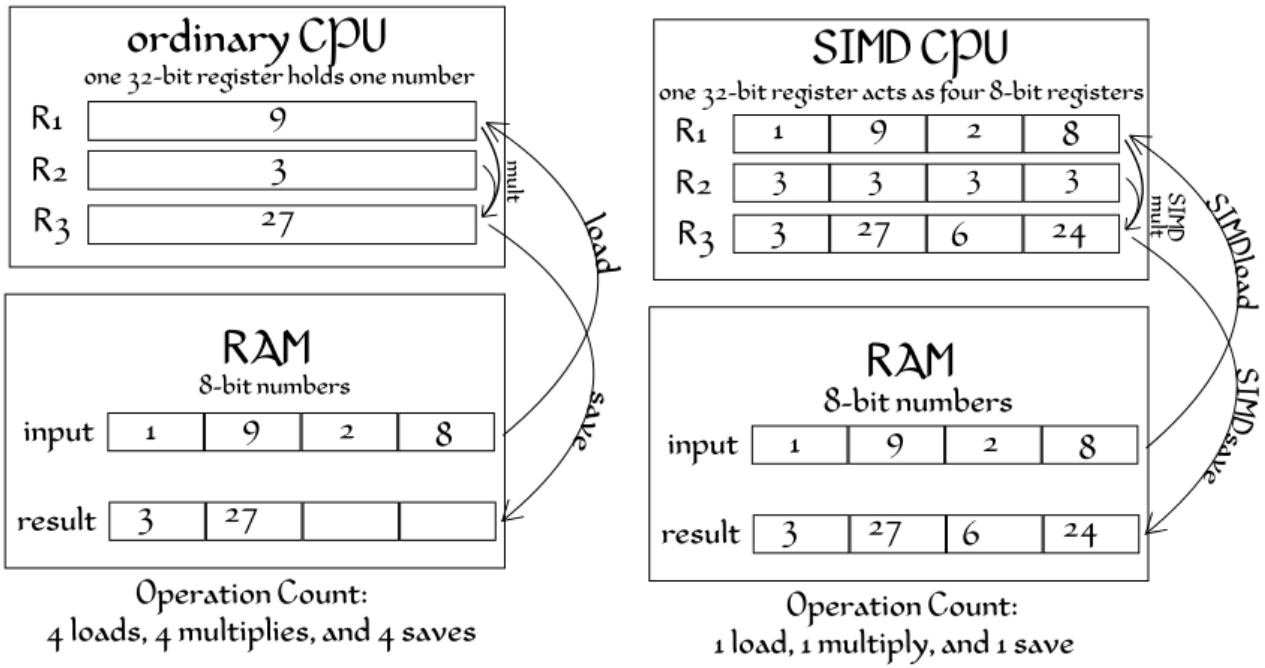
Vectorization allows us to process multiple values in one instruction.

How is that possible?

- vector units

What are those units?

- special computation units
- based on the SIMD (Single Instruction Multiple Data) principle
- calculate multiple results from multiple inputs in one instruction



Vectorization

What is Vectorization?

Vectorization allows us to process multiple values in one instruction.

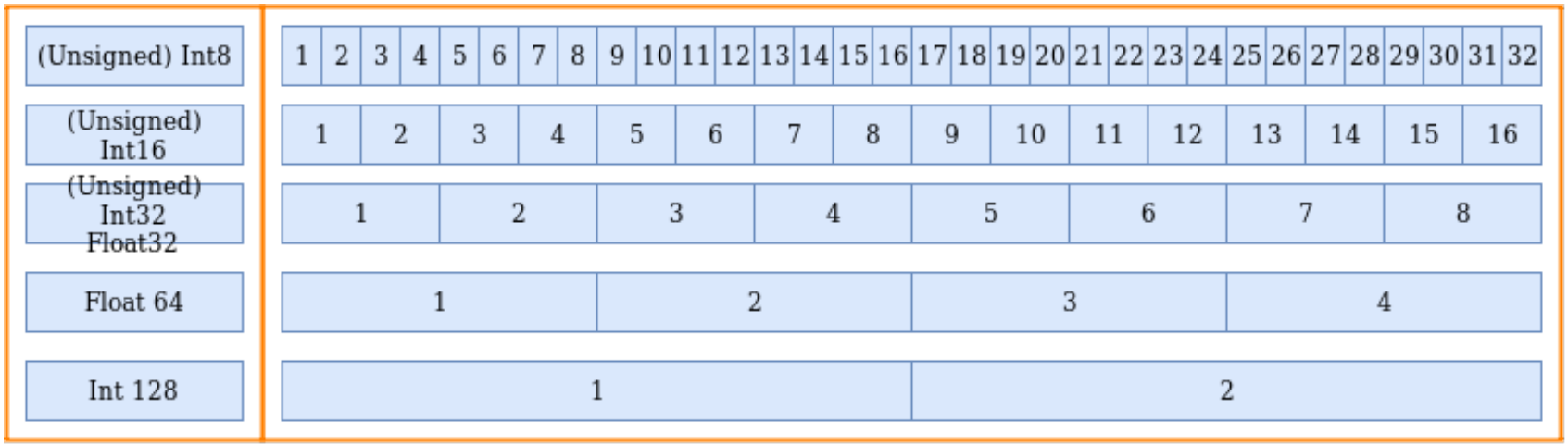
How is that possible?

- vector units
- vector registers

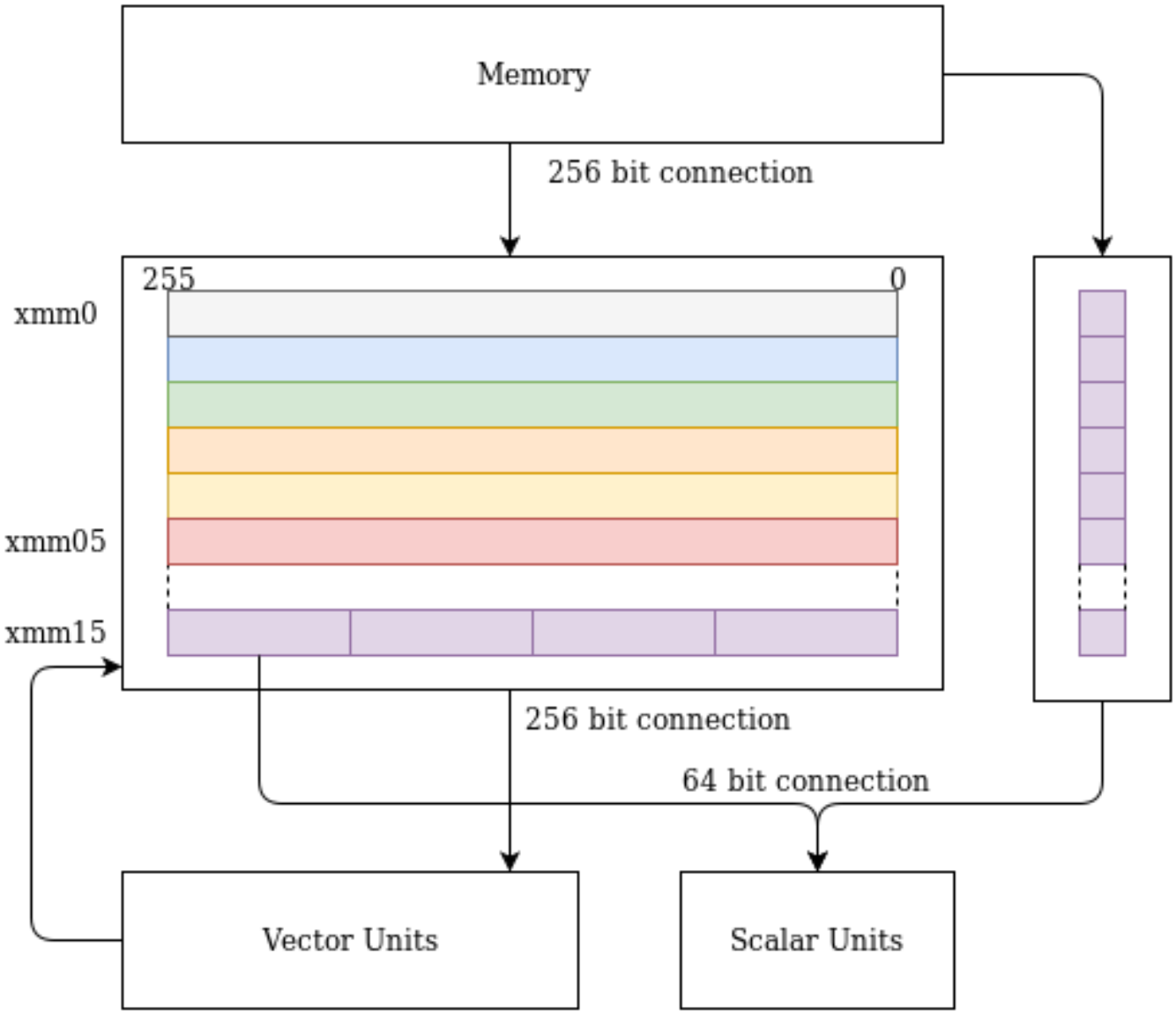
Vector Registers

- extra registers on the CPU
- can store and load multiple values at once

For 256-bit wide vector registers



Vector Registers



Vectorization

What is Vectorization?

Vectorization allows us to process multiple values in one instruction.

How is that possible?

- vector units
- vector registers
- extended set of CPU instructions

The vector instructions

- extra instructions to use the vector units/registers
- depend on CPU architecture
- different register widths per architecture
- different extensions through time

Important differences between instruction sets:

- SSE(Streaming SIMD Extensions)
  - only single precision floats
  - 8 128-bit vector registers
  - first supported by intel pentium 3

Important differences between instruction sets:

- SSE(Streaming SIMD Extensions)
  - only single precision floats
  - 8 128-bit vector registers
  - first supported int intel pentium 3
- SSE2
  - added support for 16-bit short, 32-int, 64-double-precision and 64-int
  - added 8 new vector registers for x64

Important differences between instruction sets:

- SSE(Streaming SIMD Extensions)
  - only single precision floats
  - 8 128-bit vector registers
  - first supported int intel pentium 3
- SSE2
  - added support for 16-bit short, 32-int, 64-double-precision and 64-int
  - added 8 new vector registers for x64
- AVX/AVX2(Advanced Vector Extensions)
  - now 256-bit registers
  - added three-operand SIMDs
  - added gather support

Vectorization

What is Vectorization?

Vectorization allows us to process multiple values in one instruction.

How is that possible?

- vector units
- vector registers
- extended set of CPU instructions

What speedup can we expect?

type-width	128-bit	256-bit
8	1600%	3200%
16	800%	1600%
32	400%	800%
64	200%	400%

Real speedup will not be as huge

- overhead from loops
- cache misses/ memory access times
- data layout not perfect

Vectorization

What is Vectorization?

Vectorization allows us to process multiple values in one instruction.

How is that possible?

- vector units
- vector registers
- extended set of CPU instructions
- everything implemented in silicon

The effect:

- huge speedups

What makes my code eligible for vectorization?

- calculations over arrays
- code must be in the innermost loop
- no if branches
- only inlined functions
- continuous data chunks

Data organisation

Context: distance vector = pos1 - pos2

```
struct vector
{
    float x;
    float y;
    float z;
}
```

```
struct particle
{
    vector pos;
    vector velo;
    vector accel;
}
```

This will not work well

- data is not coherent

Data organisation

Context: distance vector = pos1 - pos2

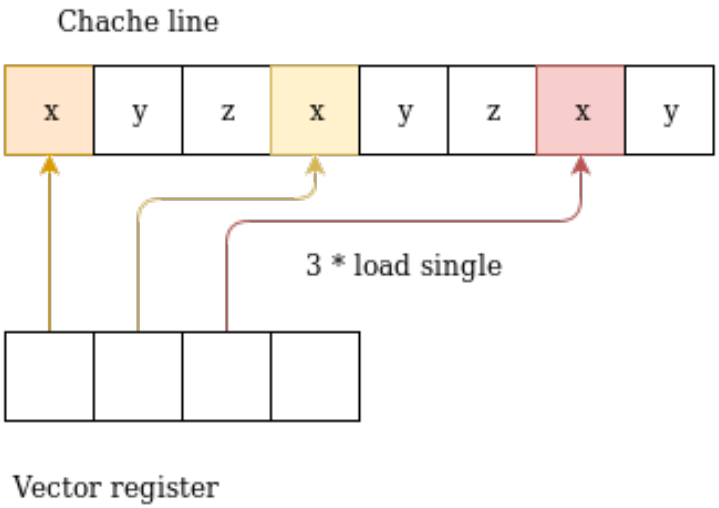
```
struct vectors
{
    float x[particle_cnt];
    float y[particle_cnt];
    float z[particle_cnt];
}
```

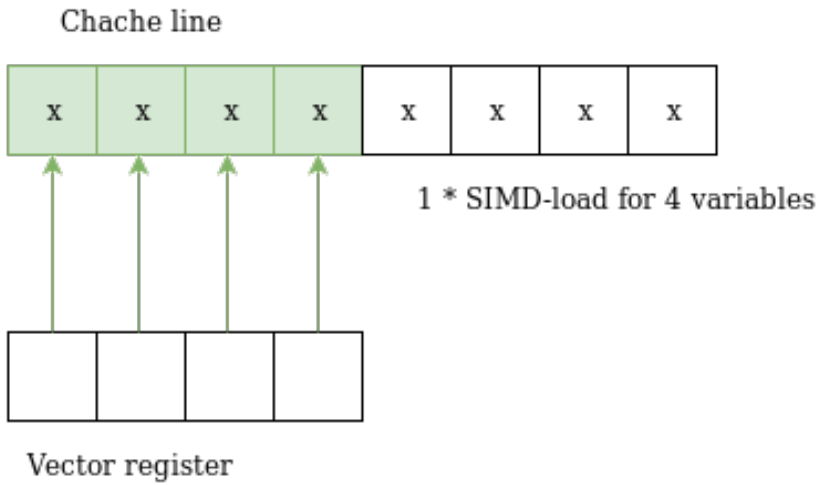
```
struct particles
{
  vectors pos;
  vectors velo;
  vectors accel;
}
```

This will work well

- data is now coherent

data organisation





How can I use vectorization?

The compiler does that for us if we tell it.
Example for gcc:

- gcc standard optimizations do not vectorize
- -O3 enables auto vectorization
- -O3 does it by using the -ftree-vectorize flag
- -fopt-info-vec enables vectorization report
- -save-temps saves the temporary files eg. assembler code

```
void test(float * vec1, float * vec2, float * res) {
    for (unsigned long i = 0; i < vector_size; i++) {
        res[i] += vec2[i] * vec1[i];
    }
}
```

- program checks for overlapping arrays parts
- program needs to check for aliasing

The restrict keyword:

Tells the compiler that the pointers are not aliased.
Meaning that the (sub)arrays are not overlapping or the same.

```
void test(float *__restrict vec1,
          float *__restrict vec2,
          float *__restrict res) {
    for (unsigned long i = 0; i < vector_size; i++) {
        res[i] += vec2[i] * vec1[i];
    }
}
```

- needs information about type boundaries

__attribute__((__aligned__(type_size))) :
Tells the compiler the size of the type in bit.
So that it is known how big a to be loaded bit word is.
Otherwise size will be checked at runtime.

```
constexpr size_t float_size = sizeof(float) * 8;
typedef float float_32 __attribute__((__aligned__(float_size)));

float_32 *vec1;
float_32 *vec2;
float_32 *res;

void test(float_32 *__restrict vec1,
          float_32 *__restrict vec2,
          float_32 *__restrict res)
{
    for (unsigned long i = 0; i < vector_size; i++) {
        res[i] += vec2[i] * vec1[i];
    }
}
```

```
void test(float_32 * vec1,
          float_32 * vec2,
          float_32 * res)
{
#pragma omp simd aligned(vec1, vec2, res:32)
    for (unsigned long i = 0; i < vector_size; i++) {
        res[i] += vec2[i] * vec1[i];
    }
}
```

Some other usefull omp commands

- collapse(x)
  collapses nested for loops into one loop
- unroll(x)
  loop unrolling hint

commands can be combined with parallelization pragmas
e.g:

```
#pragma omp for simd aligned(var, var2:32)
```

Vectorization: Pros/Cons

Pros:

- depending on numeric type we can gain huge to immense speedup
- most modern systems support vectorization
- no extra cost for new hardware
- no extra software needed

Cons:

- complicated to implement for object oriented design
- exact result only visible in assembler code

Conclusion

- vectorization is a from of optimization
  - supported by modern compilers (gcc 4.6 and onward)
  - supported in modern hardware
  - when done right gives immense speedup
- Vectorizing
  - compiler does it for us
  - if it gets enough info
  - needs coherent data layout

- talk about vectorization by Ulrich Drepper
  `https://www.youtube.com/watch?v=DXPfE2jGqg0`
- talk about vectorization by James Reinders
  `https://www.youtube.com/watch?v=hyZMssi_gZY&t=1640s`
- Article about auto vectorization (caution! for gcc 4.7)
  `https://locklessinc.com/articles/vectorize/`
- AMD's 3DNow! wikipedia page
  `https://en.wikipedia.org/wiki/3DNow!`
- SSE2 wikipedia page
  `https://en.wikipedia.org/wiki/SSE2`
- AVX2 wikipedia page
  `https://en.wikipedia.org/wiki/Advanced_Vector_`