JÜLICH
FORSCHUNGSZENTRUM

OpenCL Vectorising Features

Andreas Beckmann

November 21, 2017

Levels of Vectorisation

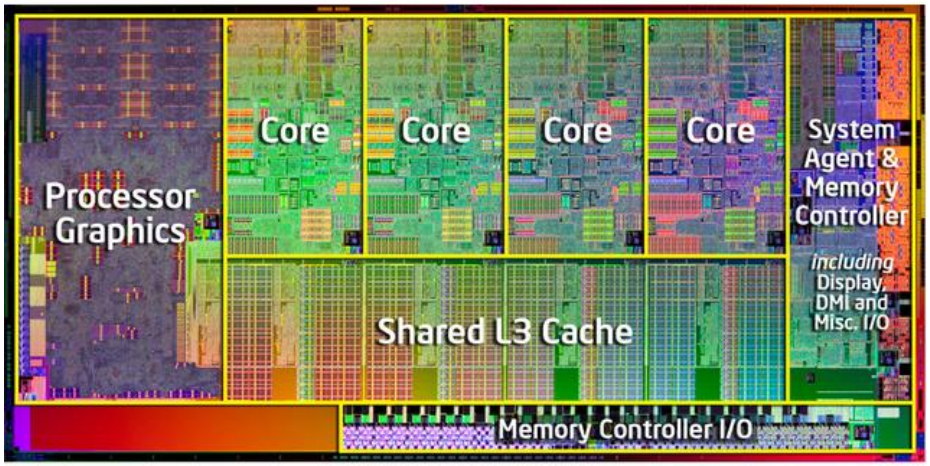
- vector units, SIMD devices
  - width, instructions
  - SMX, SP cores
  - Cus, PEs
- vector operations within kernels

Intel Legacy Vector Features

MMX (1997) Pentium MMX
+ 64-bit registers: 2x32/4x16/8x8 integer operations only

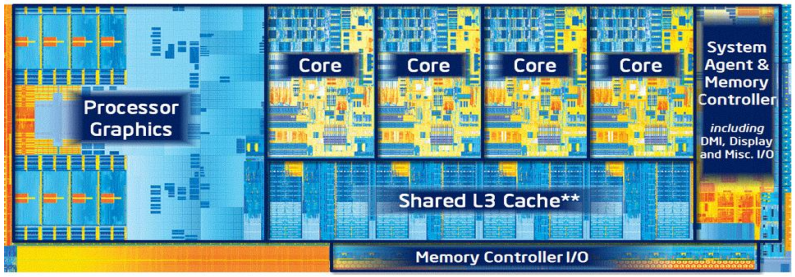
Streaming SIMD Extensions (SSE) introduced in 1999 with Pentium III
+ Maximum vector size = 128 bits
+ Maximum number of registers = 2 (e.g. A = A + B)
+ Support for only one data type: 4 32-bit FP numbers

SSE2 - SSE4
+ More data types and instructions
+ Still only 128-bit, two register



Intel Sandy Bridge / Ivy Bridge Vector Features

Sandy Bridge introduction
+ Doubled size to 256-bit vectors
+ New instruction encoding, also for 128-bit vector operations
+ Support for 3 registers per instruction
+ e.g. C = A + B
+ Allows non-destructive operations
+ legacy SSE supported with switching penalties
+ 1 Addition + 1 Multiplication per cycle



Improved in Ivy Bridge
+ Conversion between compressed 16-bit FP format and 32-bit single precision format
+ Latter is used for AVX, SSE
+ Enables higher precision calculations
+ Future support for up to 1024 FP vector support
+ More operations per instruction -> less power usage

Intel Haswell/Broadwell Vector Features

FMA – fused multiply-add

A = A * B + C B = A * B + C C = A * B +C (Intel)

D = A * B + C (AMD)

Two FMA instructions per cycle!

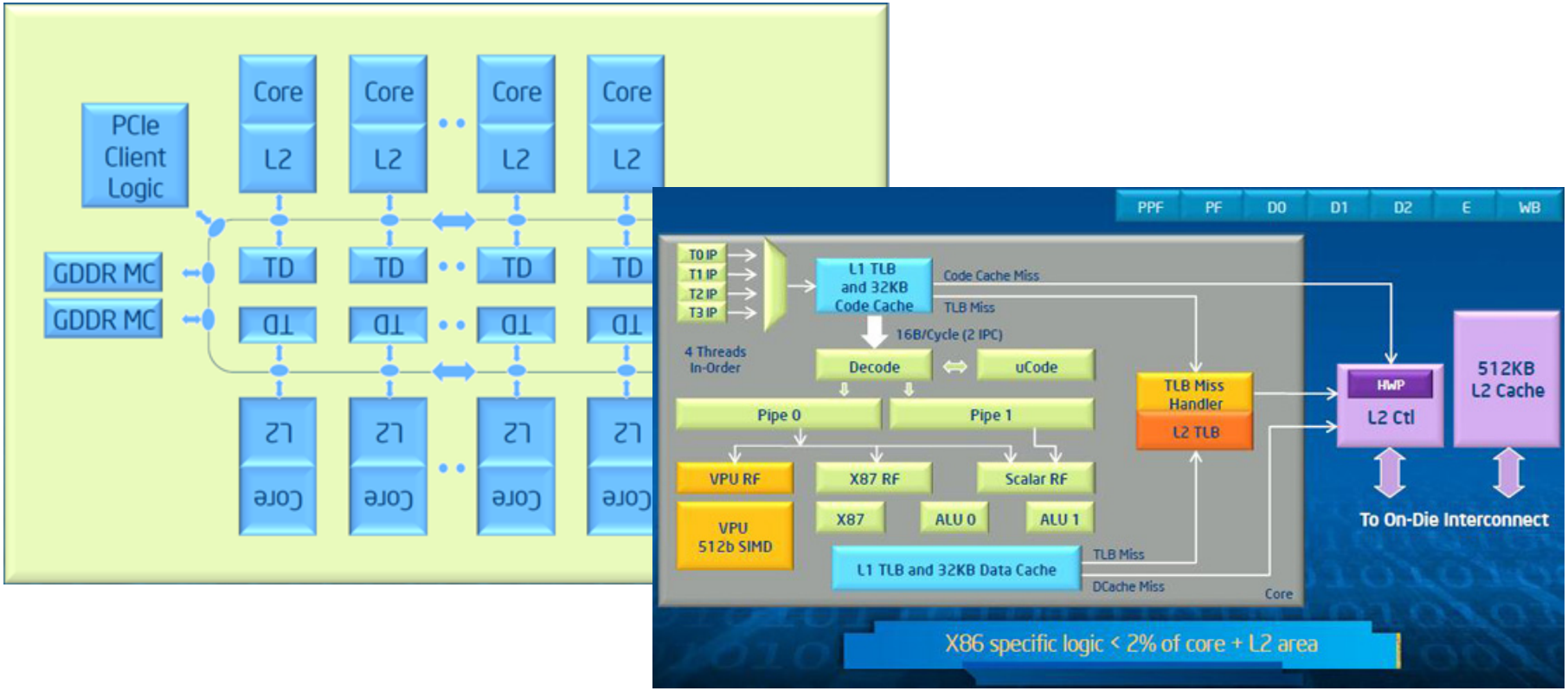
More instructions incl. Cross-lane shuffling

Additional special purpose feature sets (crypto etc.)

Intel Skylake Vector Features

Skylake-SP Xeon with AVX512
(a subset of the AVX512 in KNL)

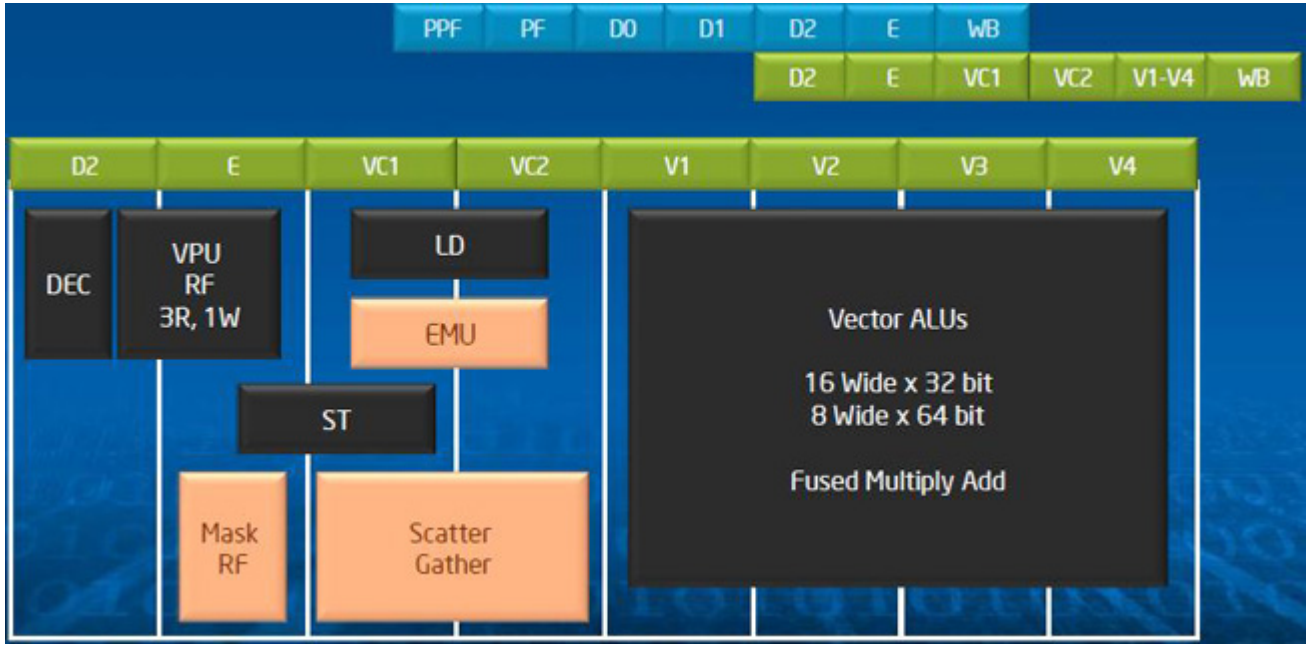
Intel Xeon Phi Coprocessor



General purpose programming environment

- runs Linux (full service, open Source OS)
- runs applications written in Fortran, C, C++, OpenMP, **OpenCL,** …
- runs x86 ISA + SIMD extension, supports x86 coherent memory model
- x86 collateral (libraries, compilers, debuggers, ...)

Intel Xeon Phi Coprocessor Vector Processing Unit (VPU)



- 512-bit SIMD instruction set
- can execute 16 single-precision (SP) or 8 double-precision (DP) operations per cycle and supports Fused-Multiply-ADD (FMA)
- mask register allows per lane predicated execution
- supports gather / scatter instructions (non-unit stride vectorisation)
- Extended Math Unit can execute vector operations as square root, log, ...

GPU Computing

Fig.: Nvidia

Kepler GPU (GK110):

- Each green square = single FPU
- Each FPU **(about 2700)** available for a different thread
- Overall, GK110 can handle **more than 30000 threads** simultaneously…
- …and even better, in our program we can send **billions of threads** to the GPU!

Vector Operations

- modern microprocessors include vector units:
  - functional units that carry out operations on blocks of numbers
- for example, x86 CPUs have over the years introduced MMX, SSE, AVX, AVX2, AVX512 instruction sets …
  - characterized in part by their widths (e.g. SSE operates on 128 bits at a time, AVX 256 bits etc)
- to gain full performance from these processors it is important to exploit these vector units
- compilers can sometimes automatically exploit vector units.
  - experience over the years has shown, however, that you all too often have to code vector operations by hand.

Vector intrinsic challenges

- example using 128 bit wise SSE:

```
#include "xmmintrin.h "     // vector intrinsics from gcc for SSE (128 bit wide)
__m128 ramp = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);   // pack 4 floats into vector
register
__m128 vstep = _mm_load1_ps(&step);              // pack step into a vector register
__m128 xvec; = _mm_mul_ps(ramp,vstep);           // multiply corresponding 32 bit
                                                 //  floats and assign to xvec
```

- requires an assembly code style of programming:
  - Load into registers
  - Operate with register operands to produce values in another vector register
- Non portable: Change vector instruction set (even from the same vendor) and code must be re-written
- Consequences:
  - very few programmers are willing to code with intrinsics, Most programs only exploit vector instructions that the compiler can automatically generate
  - most programs grossly under exploit available performance.

Vector types

OpenCL provides a high-level portable vector instruction set

- a set of vector instructions in C kernel programming language
- portable between different vector instruction sets
- instructions support vector lengths of 2, 4, 8, and 16 … for example:
  - **char2, ushort4, int8, float16, double2**, …
- properties of these types include:
  - endian safe
  - aligned at vector length
  - vector operations (elementwise) and built-in functions

Vector operations

- Vector literal

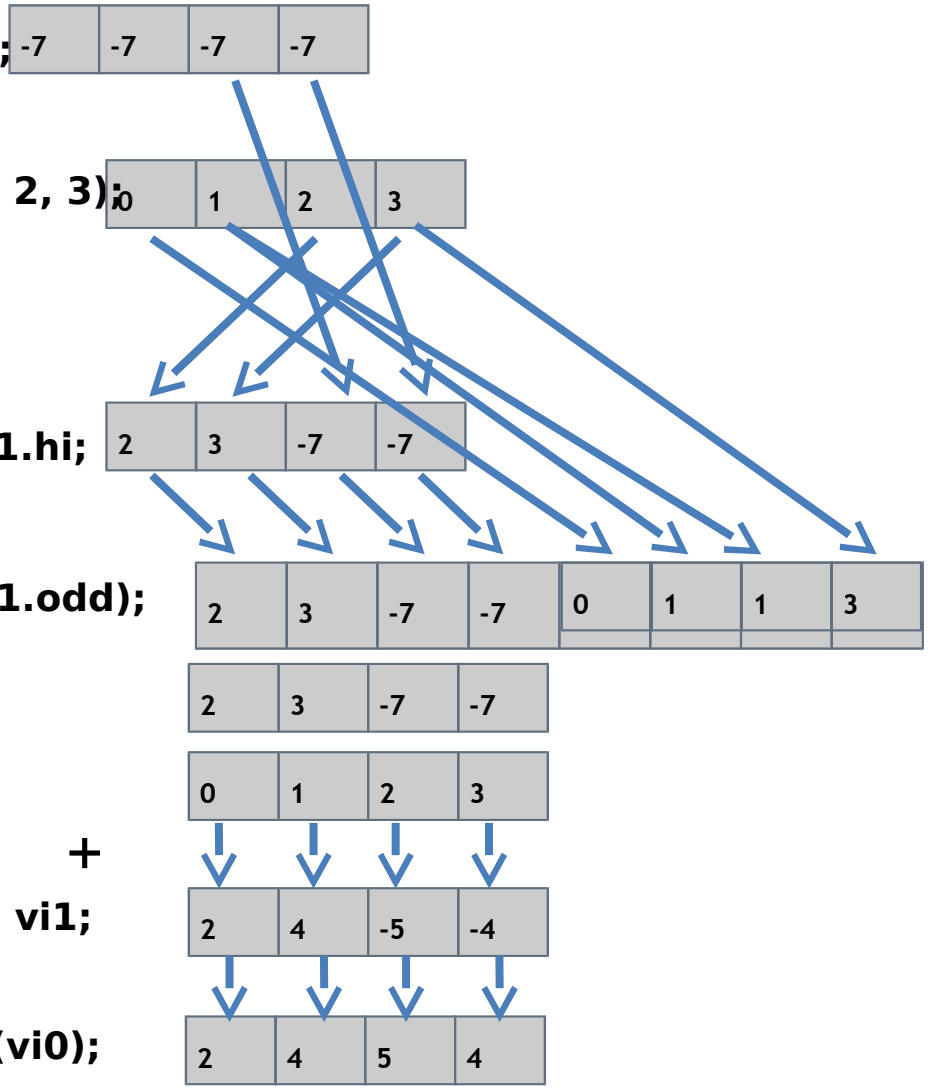
```
int4 vi0 = (int4) -7;
int4 vi1 = (int4) (0, 1, 2, 3);
```

- Vector components

```
vi0.lo = vi1.hi;
int8 v8 = (int8) (vi0, vi1.s01, vi1.odd);
```

- Vector ops

```
vi0 += vi1;
vi0 = abs(vi0);
```



Using vector operations

You can convert a scalar loop into a vector loop using the following steps:

- based on the width of your vector instruction set and your problem, choose the number of values you can pack into a vector register (the width):
  - e.g. for a 128 bit wide SSE instruction set and float data (32 bit), you can pack four values (128 bits = 4*32 bits) into a vector register
- unroll the loop to match your width (in our example, 4)
- set up the loop preamble and postscript, e.g. if the number of loop iterations doesn't evenly divide the width, you'll need to cover the extra iterations in a loop postscript or pad your vectors in a preamble
- replace instructions in the body of the loop with their vector instruction counter parts

Vector instructions example

- scalar loop:

```
for (i = 0; i < 34; i++) x[i] = y[i] * y[i];
```

- width for a 128-bit SSE is 128/32=4
- unroll the loop, then add postscript and premable as needed:

```
NLP = 34+2; x[34]=x[35]=y[34]=y[35]=0.0f // preamble to zero pad
for (i = 0; i < NLP; i = i + 4) {
  x[i] = y[i] * y[i];  x[i+1] = y[i+1] * y[i*1];
  x[i+2] = y[i+2] * y[i*2];  x[i+3] = y[i+3] * y[i*3];
}
```

- replace unrolled loop with associated vector instruction

```
float4 x4[DIM], y4[DIM];
// DIM set to hold 34 values extended to multiple of 4 (36)
float4 zero = {0.0f, 0.0f, 0.0f, 0.0f};
NLP = 34 % 4 + 1; // 9 values (as 34 isn't a multiple of 4)
x4[NLP-1] = 0.0f; y4[NLP-1] = 0.0f; // zero pad arrays

for (i = 0; i < NLP; i++)
x4[i] = y4[i] * y4[i]; // actual vector operations
```

Exercise OpenCL_Vectorising_Features/dot

- inspect the provided dot programs
  - dot_product.cc (host part)
  - DotProduct1.cl (device part)
- and execute it on one of the course test platforms
  - make
  - ./dotproduct cpu | gpu | acc [<kernel.cl>]

- complete the TODO section in the vectorised-kernel
  - DotProduct2.cl
- by using vector instructions (cf. slide 12 or reference card)
- activate the new kernel in the host program and execute

Auto-vectorisation

- the OpenCL device compilers are good at auto-vectorising your code
- adjacent work-items may be packed to produce vectorized code

- by using vector operations the compiler may not optimize as successfully
- so ***think twice*** before you explicitly vectorize your OpenCL kernels, you might end up hurting performance!