Vector units

Oguz Kaya

University' Paris-Saclay

Orsay, France

université PARIS-SACLAY

Outline

université PARIS-SACLAY

Outline

université PARIS-SACLAY

Outline

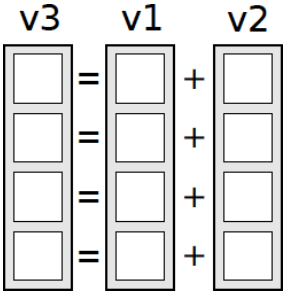
université PARIS-SACLAY

Scalar

A scalar processor performs the operations one by one each operation on scalar data (a scalar)

```
v3[0] = v1[0] + v2[0];
v3[1] = v1[1] + v2[1];
v3[2] = v1[2] + v2[2];
v3[3] = v1[3] + v2[3];
```

Scalar

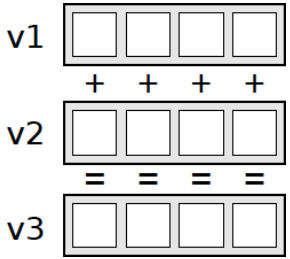


Vector Calculus

A vector processor performs operations in parallel on a set of scalars, each operation being on vectors

```
v3 = v1 + v2;
```

Vector Calculus



Advantages/inconveniences of the vector

1. Advantage:
   - Instructions n fois plus rapide, n la largeur de l'unite' vectorielle si l'algorithme se pe au paradigme SIMD: une me e nstruction simultane'ment sur plusieurs de es
     - vector calculus scientific
     - simulations image
     - processing
     - automatic learning (neural networks)
2. Inconve'nient:
   - Gain only if the algorithm se p o the SIMD paradigm.
   - Otherwise it is a loss of circuitry that could e sed' for other purposes (cores, cache, register, . . .).
   - Sometimes difficult (and dirty) to use by hand (automation possible).

First implementations

- Abes 60: first Solomon project
- 1970s: ILLIAC IV, University of Illinois, then CDC,
- Cray, NEC, Hitachi, Fujitsu
- since 1996: scalar + vector processors, Pentium (MMX, SSE, AVX), PowerPC (AltiVec)
- now: in almost all architectures (x86, ARM, RISC-V)

General operation of SIMD units

Same as the operation of scalar registers, except that each operation is done on a set of data

1. Loading data from memory to SIMD registers
2. Ope'ration on SIMD registers
3. Placement of the result in memory

Programming

- **Automatic vectorization:** The compiler tries to vectorize the code automatically.
- **Assembly programming:** Low-level programming, direct manipulation of registers and instructions.
- **Intrinsic:** Manipulation of variables by finite intrinsic functions in a ?mmintrin.h file.
- **High-level libraries (OpenMP):** Self-vetted using the tips into the library.

Outline

université PARIS-SACLAY

Streaming SIMD Extensions (SSE)

- SSE - Pentium III, Athlon XP
- SSE2 - Pentium 4
- Willamette SSE3 - Pentium 4 Prescott
- SSSE3 - (Supplemental SSE3) Core Woodcrest
- SSE4.1 - Core 2 Penryn
- SSE4.2 - Core i7 Nehalem
- AVX - Sandy Bridge
- **AVX2** - Haswell
- AVX512 - Skylake and beyond

Streaming SIMD Extensions (SSE)

Each new version of SSE/AVX contains the previous SSE(N) = new instructions + SSE(N-1)

Intrinsics

The intrinsics are accessible through .h files that define: variable types

- functions operating on these variables

- SSE: xmmintrin.h
- SSE2: emmintrin.h
- SSE3: pmmintrin.h
- SSSE3: tmmintrin.h
- SSE4.1: smmintrin.h
- SSE4.2: nmmintrin.h
- **AVX/AVX2: immintrin.h**

Run **lscpu** under Linux to display the supported version of e.

Advanced Vector Extensions (AVX)

16 256bit registers on x86-64: YMM0 - > YMM15

- memory transfers < - > AVX registers arithmetic
- operations
- ope'rations logic tests
- permutations
-

Compilation with AVX

```
gcc program.c -o program -mavx2
```

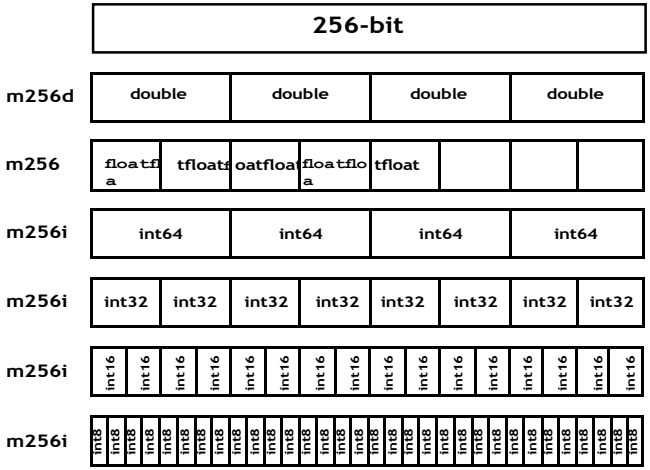
```
./program
```

Types of variables

- __m256: 8 floats
- __m256d: 4 doubles
- __m256i: 256/width integers ($width = 8, 16, 32, 64$)

There is a set of 'finite' instructions for each type.

AVX vectors


256-bit
m256d
double
double
double
double
m256
m256i
int64
int64
int64
int64
m256i
int32
int32
int32
int32
int32
int32
int32
int32
m256i
int16
m256i
int8

Functions

_mm256 {*operation*}{*non-alignment*} {*dataorganization*}{*datatype*}. . .

__m256 mm256 add ps(__m256 A, __m256 B)

Terminology

_mm256_loadu_ps256

- s (single): single precision float (32bits)
- d (double): float double precision (64bits) i...
- (integer): integer
- p (packed): contiguous, operate on the whole
- vector s (scalar): operate only on one element
- (obsolete!) u (unaligned): data not aligned in
- memory l (low): low order bits
- h (high): most significant bits
- r (reversed): in reverse order

General Operation

1. AVX variables declaration (registers): __m256 r1;

2. loading data from the memory to the AVX registers: float A[8] __attribute__ ((aligned(16)));
   ...
   r1 = _mm256_load_ps(&A[0]);

3. operations on SIMD registers r1 = _mm256_add_ps(r1, r1);

4. placing the content of the registers in memory: _mm256_store_ps(&A[0], r1)

Transfers between the memory and the AVX

- **Floating:**
  __m256 _mm256_load_ps(float *addr)
  void _mm256_store_ps(float *addr, __m256 a)
- **Doubles:**
  __m256d _mm256_load_pd(double *addr)
  void _mm256_store_pd(double*addr, __m256 a)
- **Entity:**
  __m256i _mm256_load_si256(__m256i const * addr)
  void _mm256_store_si256(__m256i const * addr, __m256i a)

Transfers from memory to AVX registers

- aligned or not aligned:
  _mm256_load. ... or _mm256_loadu. ...
  _mm256_blind. ... or mm256 blindu. . .

- per vector: 8 × *SP*, 4 × *DP*, ... _mm256
  _load ps, _mm256 load pd, ... _mm256
  _store ps, mm256 store pd, ...

Aligned

- The access to a variable is **aligned** if the address of this variable is a multiple of its size. This constraint is material.
- For AVX variables, the processor can perform efficient transfers of 32-byte (256-bit) AVX variables if it is aligned to 32 bytes.
- To get the alignment of a type: `__alignof__(type)`

Aligned

A cache line is usually 32 to 64 bytes in size.

If the data loaded into an SSE register comes from a memory area that "straddles" 2 cache lines, then the 2 cache lines must be read in order to fill the SSE register, which implies a decrease in performance

Allocation of aligned donors

- 32 bytes static alignment: int x __attribute__((aligned (32)))
  float y[8] __attribute__ ((aligned (32)))

- Dynamic alignment on 32 bytes:
  type* x = (type*) _mm_malloc(size*sizeof(type), 32);
- Non-aligned accesses can be much slower (but not as much in the new generations).

Arithmetic operations

- _mm256_add_pd(__m256d A, __m256d B) - (Add-Packed-Double)
  - Input: [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Output: [A0 + B0, A1 + B1, A2 + B2, A3 + B3]
- _mm256_add_ps(__m256 A, __m256 B) - (Add-Packed-Single)
  - Input: [A0, ... , A7], [B0, ... , B7]
  - Output: [A0 + B0, ... , A7 + B7]
- _mm256_add_epi64(__m256i A, __m256i B) - (Add-Packed-Int64)
  - Input: [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Output: [A0 + B0, A1 + B1, A2 + B2, A3 + B3]
- _mm256_add_epi32(__m256i A, __m256i B) - (Add-Packed-Int32)
- _mm256_add_epi16(__m256i A, __m256i B) - (Add-Packed-Int16)
- _ _ _ -- --

Arithmetic operations

mm256 add epi8(m256i A, m256i B) - (Add-Packed-Int8)

Same for mm256 sub . _

Arithmetic operations

- _mm256_mul_pd(__m256d A, m256d B) - (Multiply-Packed-Double)
  - Input: [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Output: [A0 * B0, A1 * B1, A2 * B2, A3 * B3]
- _mm256_mul_ps(__m256 A, m256 B) - (Multiply-Packed-Single)
  - Input: [A0, ... , A7], [B0, ... , B7]
  - Output: [A0 * B0, ... , A7 * B7]
- _mm256_mul_epi64(__m256i A, m256i B) - (Multiply-Packed-Int64)
  - Input: [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Output: [A0 * B0, A1 * B1, A2 * B2, A3 * B3]
- _mm256_mul_epi32(__m256i A, m256i B) - (Multiply-Packed-Int32)
- _mm256_mul_epi16(__m256i A, m256i B) - (Multiply-Packed-Int16)
- _ _ _ -- --

Arithmetic operations

mm256 mul epi8(m256i A, m256i B) - (Multiply-Packed-Int8)

Ditto for mm256 div ... ; beware, the division is **very slow**!

Arithmetic operations

- _mm256_fmadd_pd(__m256 A, __m256 B, __m256 C) - (Fused-Multiply-Add-Packed-Double)
  - Input: [A0, A1, A2, A3], [B0, B1, B2, B3], [C0, C1, C2, C3] Output: [
  - A0 * B0 + C0, A1 * B1 + C1, A2 * B2 + C2, A3 * B3 + C3]
- _mm256_fmadd_ps(__m256 A, __m256 B, __m256 C) - (Fused-Multiply-Add-Packed-Single)
  - Input: [A0, ... , A7], [B0, ... , B7], [C0, ... , C7] Output:
  - [A0 * B0 + C0, ... , A7 * B7 + C7]

Same for _mm256_fmsub_ps.

Horizontal operations

- _mm256_hadd_pd(A, B) - (Horizontal-Add-Packed-Double)
  - Input: [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Output: [A0 + A1, B0 + B1, A2 + A3, B2 + B3]
- _mm256_hadd_ps(A, B) (Horizontal-Add-Packed-Single)
  - Input: [A0, . . . , A7], [B0, . . . , B7]
  - Output: [A0 + A1, A2 + A3, B0 + B1, B2 + B3, A2 + A3, . . . , B6 + B7]
- _mm256_hadd_epi32(_m256i A, _m256i B, m256i C) - (Horizontal-Add-Packed-Int32)
- mm256 hadd epi16(_m256i A, m256i B, m256i C) - (Horizontal-Add-Packed-Int16)
- Same as mm256 hsub . _

Logical and comparative operations

- _mm256 _{and , or , xor, . . . } {ps, pd, si256}(A,
- _B) mm256 cmp {ps, pd}(A, B, op)
  - op = CMP LT OS: Comparison A < B op
  - = CMP LE OS: Comparison A :(B op =
  - CMP EQ OS: Comparison A == B op =
  - CMP GT OS: Comparison A > B op =
  - CMP GE OS: Comparison A ? B
- _mm256 cmpeq epi{64, 32, 16, 8}(m256i A, __m256i B)
- _mm256 cmpgt epi{64, 32, 16, 8}(m256i A, __m256i B)

For C = mm256 cmp. . (A, B), if A[i] op B[i] is correct, C[i] = 0b11. ..... 1. If it is false, C[i] = 0b00. ... 0.

Pmix: Permutation

The instruction allows to obtain any parmutation of data in a register AVX. The permutation is specified by an immediate entry of which bits are used as indices.

- __m256d B = mm256 permute4x64 pd(m256d A, const int imm8)
  B[0] = A[imm8[1:0]]
  B[1] = A[imm8[3:2]]
  B[2] = A[imm8[5:4]]
  B[3] = A[imm8[7:6]]
- __m256i B = mm256 permute4x64 epi64(m256i A, const int imm8)
  B[0] = A[imm8[1:0]]; B[1] = A[imm8[3:2]]
  B[2] = A[imm8[5:4]]; B[3] = A[imm8[7:6]]
- __m256 B = mm256 permutevar8x32 ps(m256 A, __m256i idx) B[i] = A[idx[(32 * i) + 2 : (32 * i)]]
- __m256i B = mm256 permutevar8x32 epi32(m256i A, __m256i idx) B[i] = A[idx[(32 * i) + 2 : (32 * i)]]

Assignment of constants

- _mm256_set_ps(float f0, . . . , float f7)
- _mm256_set_pd(double d0, double d1, double d2, double d3)
- _mm256_set_epi64x(int64_t i0, . . . , int64_t i3)
- _mm256_set_epi32(int i0, . . . , int i7)
- _mm256_set_epi16(short i0, . . . , short
- _i15) mm256_set_epi8(char i0, . . . , char

  i31)
- There is also the variant mm256_set1_ps(float f) which assigns the new value to all the elements of the vector (same for double, int{64, 32, 16, 8})

Tips for good performance

- Organize data correctly to avoid non-contiguous access.
- Stay in the cache as much as possible.
- Align the data (less important in the [illegible]versions).
- Stay as long as possible in the registers (i.e., avoid reading/writing on the memory)

Organization of data for verification

The initial structure could not be adapted to the vectorization. In this case it would be necessary to rethink it.

- Array of structures (AoS): xyzwxyzwxyzw
- Structure of Arrays (SoA): xxxxxxxxx . . . yyyyyyyy . . . zzzzzzzzz . . wwwwwwwwwww . . .
- Hybrid structure: xxxxyyyzzzzwwwwxxxxyyyyzzzzwwww .

Conclusion

- Vector units allow SIMD operation mode on a data vector. The state of the AVX
- instruction set is rather chaotic (just like x86-64) Instructions not compatible with all
- data types
- Substantial gain potential (16-32x with AVX2)