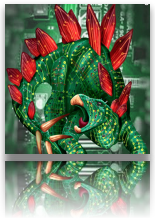
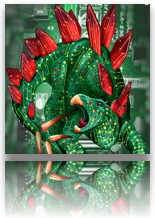


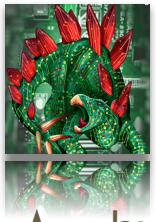
SIMD & Vectorization

Lecture 3

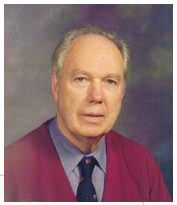


SIMD & Vectorization

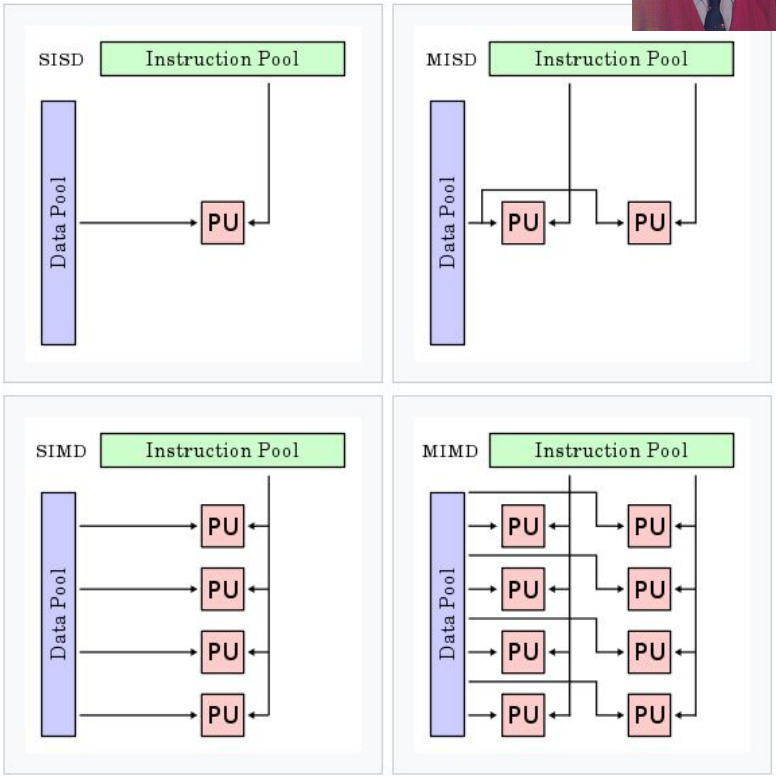
- ★ FLYNN's Taxonomy revisit
- ★ SIMD concepts
- ★ History of Vectorize Instructions
  - ○ VMX/Altivec
    (Apply, IBM, Freescale/Motorola)
  - ○ MMX, SSE, AVX, AVX2 (Intel)
  - ○ 3DNow! (Intel)
- ★ Architecture
  - ○ ALU & Registers
- ★ GPU
  - ○ CUDA
  - ○ OpenCL
- ★ AVX programming
- ★ CUDA programming
- ★ AVX in C
- ★ Vectorization in Python



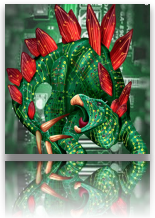
Flynn's Taxonomy



- ★ by Michael Flynn (1934-) in 1966
- ★ Partition by Instruction Stream and Data Stream
  - Single Instruction stream, Single Data stream (SISD) - **uniprocessor**
  - Single Instruction stream, Multiple Data stream (SIMD) - **vector processor**
  - Multiple Instruction stream, Single Data stream (MISD) - **Fault-Tolerant Computing** (The course is available as approved from A. Arthit.)
  - Multiple Instruction stream, Multiple Data stream (MISD) - **Multicore, Distributed Systems**



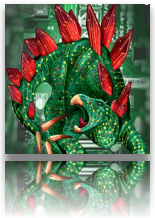
Pictures from wikipedia



SIMD Overview

- ★ A method of improving performance in applications
- ★ a single operation to be performed across a large dataset

- ★ World before SIMD,
  - Loop to iterate through each element in a dataset
  - One way to reduce the number of iterations, **loop unrolling**
- ★ SIMD packs multiple actions in each loop to parallelise them (run simultaneously).



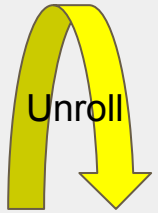
SIMD concepts

Try a simple loop

```
#define MAX 100000000

int a[MAX];
int b[MAX];
int c[MAX];

for (int i=0;i<MAX;i++) {
 c[i]=a[i]+b[i];
}
```

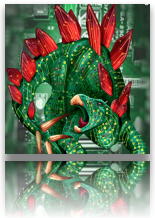
Loop unroll

```
for (int i=0;i<MAX-4;i+=4) {
 c[i]=a[i]+b[i];
 c[i+1]=a[i+1]+b[i+1];
 c[i+2]=a[i+2]+b[i+2];
 c[i+3]=a[i+3]+b[i+3];
}
```



Use packed registers and vector instructions

```
for (int i=0;i<MAX-4;i+=4) {
 {c[i], c[i+1], c[i+2], c[i+3]} =
  {a[i], a[i+1], a[i+2], a[i+3]} +
  {b[i], b[i+1], b[i+2], b[i+3]}
}
```



SIMD concepts in ASM

```
#define MAX 100000000

int a[MAX];
int b[MAX];
int c[MAX];

for (int i=0;i<MAX;i++) {
 c[i]=a[i]+b[i];
}
```

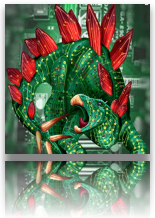
```
.L3:
    .loc 1 12 8 c
    movl    -4(%r
    cltq
    leaq    0(,%r
    leaq    a(%ri
    movl    (%rd>
    .loc 1 12 13
    movl    -4(%rbp), %eax
    cltq
    leaq    0(,%rax,4), %rcx
    leaq    b(%rip), %rax
    movl    (%rcx,%rax), %eax
    .loc 1 12 11 discriminator 3
    leal    (%rdx,%rax), %ecx
    .loc 1 12 6 discriminator 3
    movl    -4(%rbp), %eax
    cltq
    leaq    0(,%rax,4), %rdx
    leaq    c(%rip), %rax
    movl    %ecx, (%rdx,%rax)
    .loc 1 11 21 discriminator 3
    addl    $1, -4(%rbp)
.L2:
    .loc 1 11 1 discriminator 1
    cmpl    $99999999, -4(%rbp)
    jle     .L3
```

gcc 8 support for AVX

 # gcc -O3

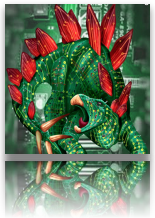
```
for (int i=0;i<MAX-4;i+=4) {
 {c[i], c[i+1], c[i+2], c[i+3]} =
  {a[i], a[i+1], a[i+2], a[i+3]} +
  {b[i], b[i+1], b[i+2], b[i+3]}
}
```

```
.L2:
.LBB3:
    .loc 1 12 2 is_stmt 1 discriminator 3 view
.LVU5
    .loc 1 12 11 is_stmt 0 discriminator 3 view
.LVU6
    movdqa    (%rcx,%rax), %xmm0
    paddd     (%rdx,%rax), %xmm0
    .loc 1 12 6 discriminator 3 view .LVU7
    movaps    %xmm0, (%rsi,%rax)
    .loc 1 12 6 discriminator 3 view .LVU8
    addq      $16, %rax
    cmpq      $400000000, %rax
    jne       .L2
```



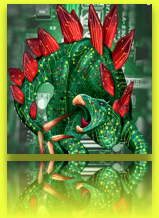
(Brief) History of SIMD instructions

- ★ 1996 - Intel Pentium MMX (unused due to technical limitations)
- ★ 1997 - PowerPC G3 AVX/Altivec Motorola (with Apple and IBM)
- ★ 1997/1998 - Intel Pentium 2 (slow adoption rate)
- ★ 1999 - Intel Pentium 3 SSE (more success, small usage compared to Altivec)
- ★ 2000 - AMD Athlon 3DNow! (proprietary/small adoption)
- ★ 2002 - Pentium 4 SSE2 (used across many multimedia applications)
- ★ 2002 - PowerPC G5 AVX/Altivec
- ★ 2003 - AMD Advanced 3DNow!/3DNow!2
- ★ 2004 - Intel Pentium 4 SSE3 (Claim to be better than Altivec)
- ★ 2004 - AMD license SSE and SSE2 for Athlon and Athlon 64

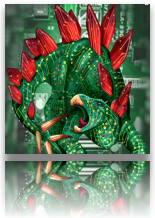


(Brief) History of SIMD instructions

- 2006 - Intel Core SSE 4
- 2007 - AMD introduces SSE5
- 2008 - Intel AVX
- ...
- 2012 - Intel/AMD FMA
- ...
- 2013 - Intel AVX2
- 2015 - Intel AVX-512
- ...
-



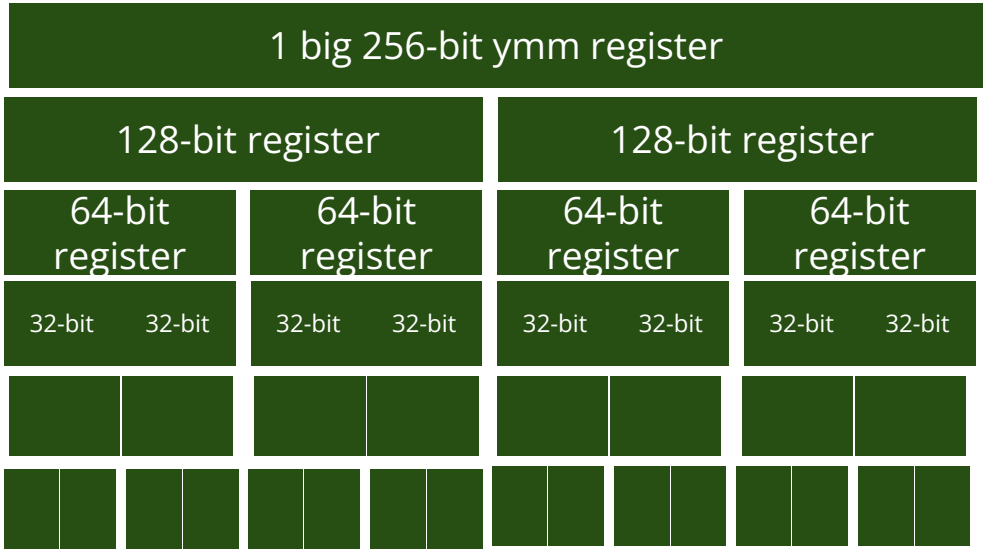
In 1990s, Altivec made Apple Macintosh a preferred choices for graphics/multimedia.

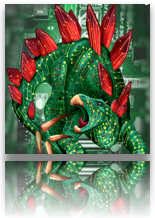


Conceptual Design

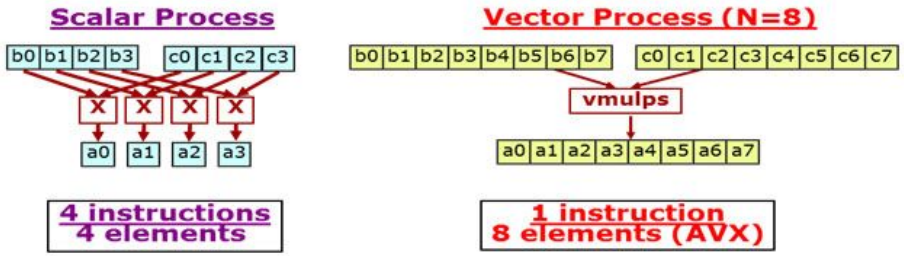
★ One (big) ALU that can partition into parts. (Think ***union*** in C)

```
union {
    unsgined __int256 u256;
    unsigned __int128[2] u128;
    unsigned long[4] u64;
    unsigned int[8] u32;
    unsigned short[16] u16;
    unsigned char[32] u8;
}
```



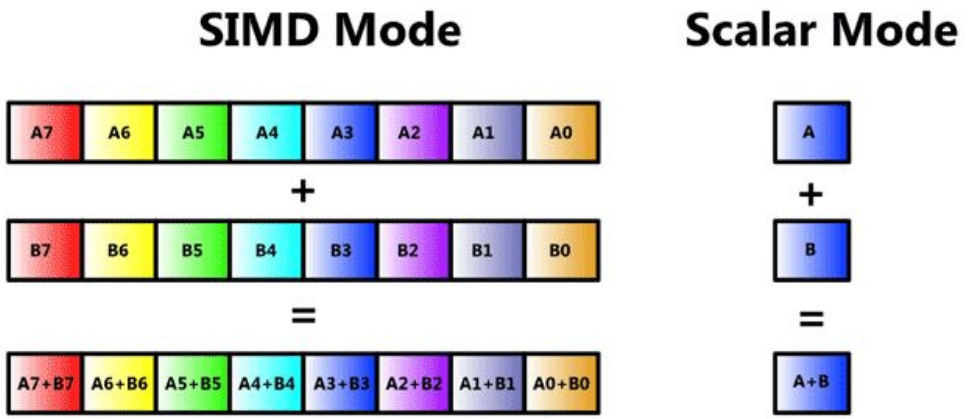


ALU design

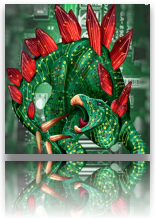


- ★ If we ignore the half-carry flag (aka nibble-carry flag), an 16-bit adder is just 2x 8-bit adders
- ★ Similar concept for other operators

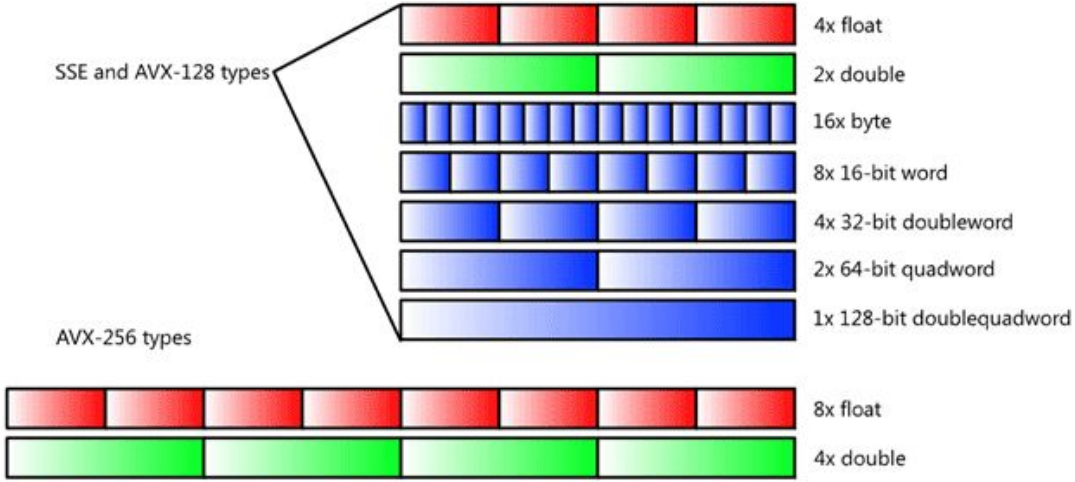
```
1010 0101_0101 0011
                   +
0100 1011_0010 0101
---------------------------
1111 0000?0111 1000
```

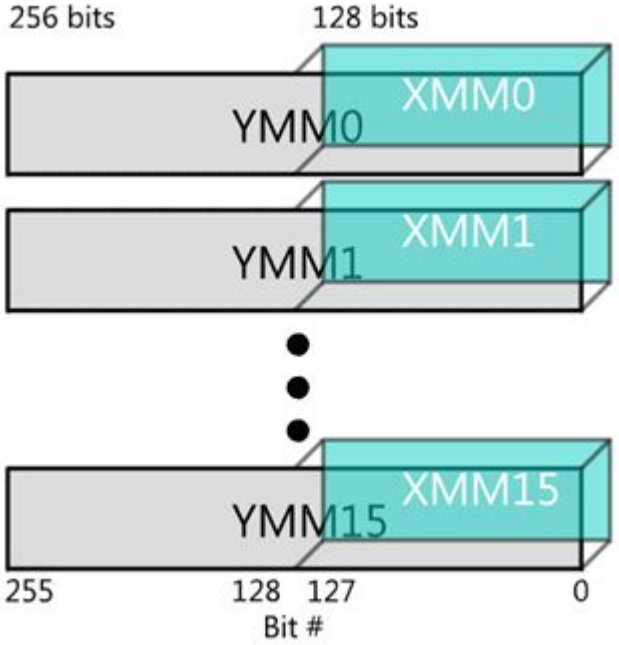


https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html

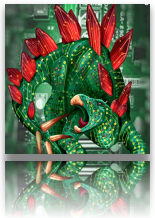


AVX registers



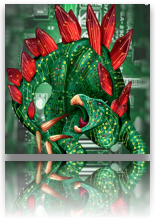


https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html



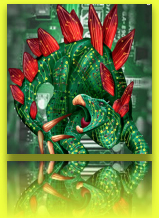
AVX Instructions

Arithmetic	Description
(V)[ADD/SUB/MUL/DIV][P/S][D/S]	Add/subtract/multiply/divide packed/scalar double/single
(V)ADDSUBP[D/S]	Packed double/single add and subtract alternating indices
(V)DPP[D/S]	Dot product, based on immediate mask
(V)HADDP[D/S]	Horizontally add
(V)[MIN/MAX][P/S][D/S]	Min/max packed/scalar double/single
(V)MOVMSKP[D/S]	Extract double/single sign mask
(V)PMOVMSKB	Make a mask consisting of the most significant bits
(V)MPSADBW	Multiple sum of absolute differences
(V)PABS[B/W/D]	Packed absolute value on bytes/words/doublewords
(V)P[ADD/SUB][B/W/D/Q]	Add/subtract packed bytes/words/doublewords/quadwords
(V)PADD[S/U]S[B/W]	Add packed signed/unsigned with saturation bytes/words
(V)PAVG[B/W]	Average packed bytes/words
(V)PCLMULQDQ	Carry-less multiplication quadword
(V)PH[ADD/SUB][W/D]	Packed horizontal add/subtract word/doubleword
(V)PH[ADD/SUB]SW	Packed horizontal add/subtract with saturation
(V)PHMINPOSUW	Min horizontal unsigned word and position
(V)PMADDWD	Multiply and add packed integers
(V)PMADDUBSW	Multiply unsigned bytes and signed bytes into signed words
(V)P[MIN/MAX][S/U][B/W/D]	Min/max of packed signed/unsigned integers

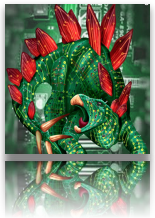


AVX Instructions (ctd.)

(V)PMUL[H/L][S/U]W	Multiply packed signed/unsigned integers and store high/low result
(V)PMULHRSW	Multiply packed unsigned with round and shift
(V)PMULHW	Multiply packed integers and store high result
(V)PMULL[W/D]	Multiply packed integers and store low result
(V)PMUL(U)DQ	Multiply packed (un)signed doubleword integers and store quadwords
(V)PSADBW	Compute sum of absolute differences of unsigned bytes
(V)PSIGN[B/W/D]	Change the sign on each element in one operand based on the sign in the other operand
(V)PS[L/R]LDQ	Byte shift left/right amount in operand
(V)SL[L/AR/LR][W/D/Q]	Bit shift left/arithmetic right/logical right
(V)PSUB(U)S[B/W]	Packed (un)signed subtract with (un)signed saturation
(V)RCP[P/S]S	Compute approximate reciprocal of packed/scalar single precision
(V)RSQRT[P/S]S	Compute approximate reciprocal of square root of packed/scalar single precision
(V)ROUND[P/S][D/S]	Round packed/scalar double/single
(V)SQRT[P/S][D/S]	Square root of packed/scalar double/single
VZERO[ALL/UPPER]	Zero all/upper half of YMM registers



With vectorization, each iteration of a loop can be executed in few cycles.



Should we use vectorization? YES!

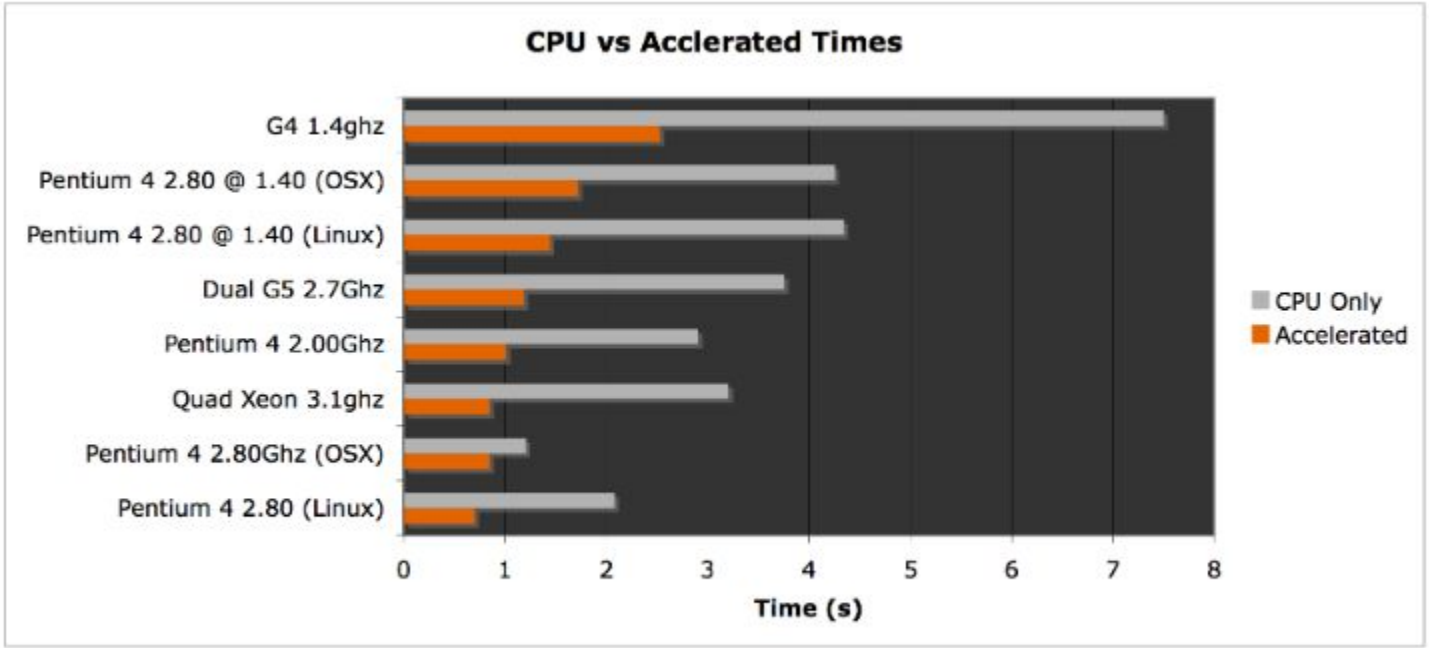
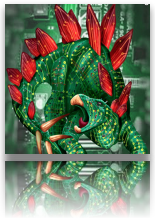
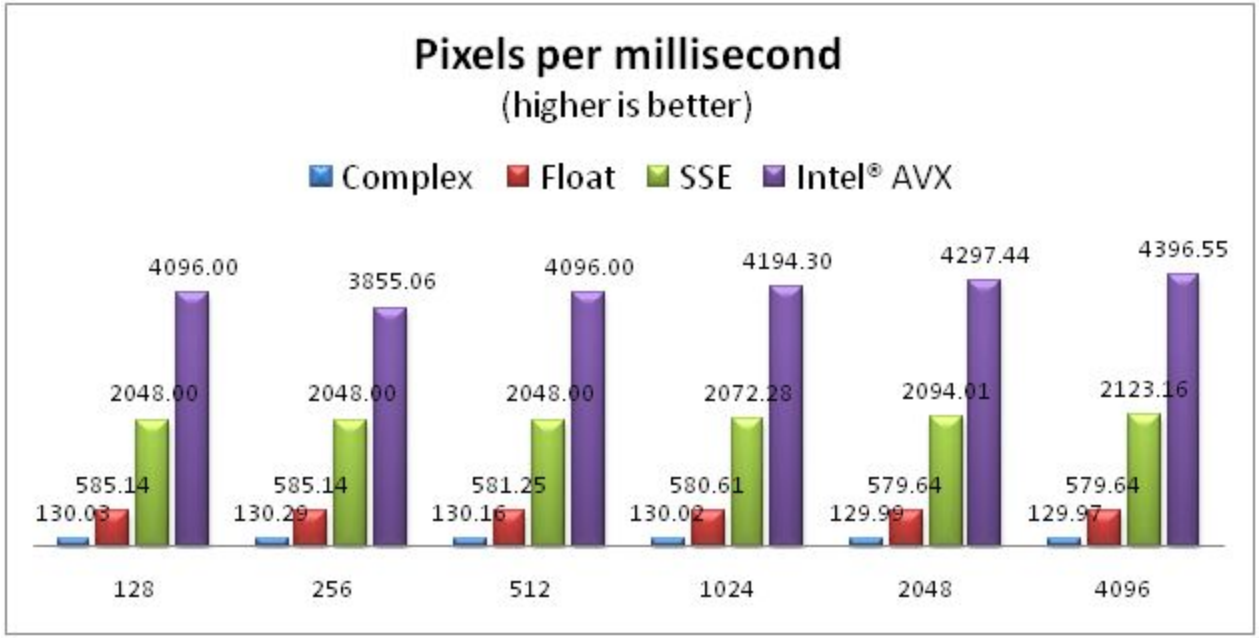


Figure 1 – Combined Results

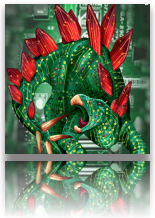
http://noisymime.org/blogimages/SIMD.pdf



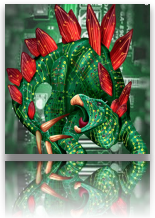
Should we use vectorization? YES!



https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html



How about GPU?



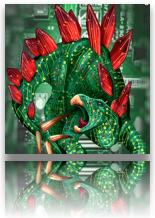
Single Instruction, Multiple Threads (SIMT)

★ Introduced by Nvidia Tesla GPU for General-Purpose Computing on GPU (GPGPU)

SIMT Terminology

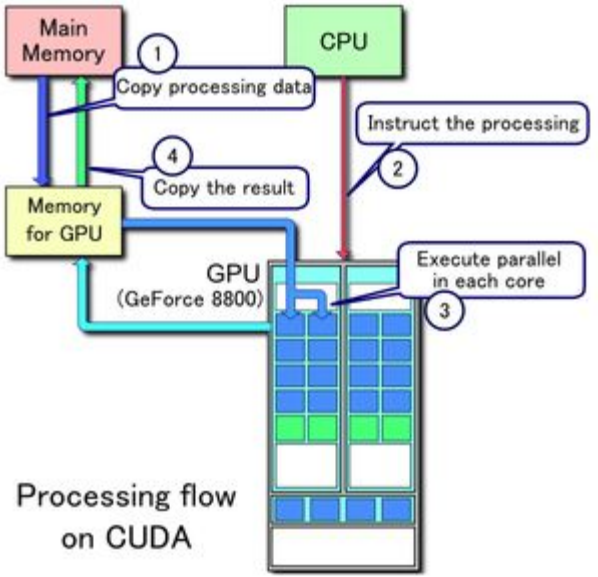
Nvidia CUDA	OpenCL	Hennessy & Patterson[7]
Thread	Work-item	Sequence of SIMD Lane operations
Warp	Wavefront	Thread of SIMD Instructions
Block	Workgroup	Body of vectorized loop
Grid	NDRange	Vectorized loop

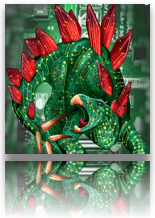
https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads



CUDA

- ★ Compute Unified Device Architecture
- ★ NVIDIA API for GPGPU
- ★ Use CUDA to abstract GPU hardware

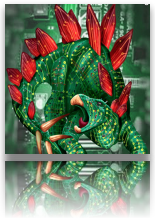




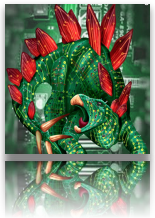
OpenCL

- ★ Open Computing Language
- ★ Apple (later Khronos Group) to create open standard for GPGPU
- ★ (In 2017, Apple deprecated OpenCL and OpenGL in favor of Metal2.)
- ★ Basically standard API with drivers for each hardware
  - Supports ARM, IBM, Intel, AMD, Nvidia, Xilinx, ...
- ★ Available on Windows, Linux, Mac OS, Androids

- ★ On September 30, 2020, OpenCL 3.0 specification was released.



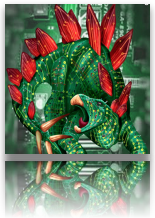
How to explicitly use vectorization in our code?



Explicitly use of vectorization in C

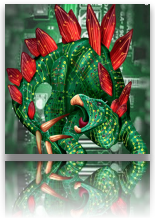
- ★ Modern gcc knows avx, avx2, avx512
- ★ Use -m to specify your architecture
- ★ Eg.
  - $ gcc -mavx -o test_avx test_avx.c
  - $ gcc -mavx2 -o test_avx test_avx.c
- ★ To learn more, see
  https://software.intel.com/content/www/us/en/develop/articles/using-avx-without-writing-avx-code.html
  https://www.codeproject.com/articles/874396/crunching-numbers-with-avx-and-avx

Well, you can also use inline assembly.



Let's try it with gcc

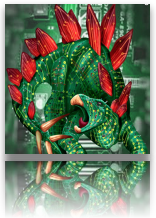
- ★ Headers
  - #include <smmintrin.h>
  - #include <immintrin.h>
- ★ Representing vectors
  - __m256 (for eight floats)
  - __m256d (for four doubles)
  - __m256i (for integers, no matter the size)
  - (For 128-bit types, replace **__m256** with **__m128** and **_mm256_** with **_mm_**)



Types of values in vectors

```
//_mm<bit_width>_<name>_<data_type>
__m256 evens = _mm256_set_ps(2.0, 4.0, 6.0, 8.0, 10.0,
12.0, 14.0, 16.0);
```

- ★ si256 – signed 256-bit integer
- ★ si128 – signed 128-bit integer
- ★ epi8, epi32, epi64 — an vector of signed 8-bit integers (32 in a __m256 and 16 in a __m128) or signed 32-bit integers or signed 64-bit integers
- ★ epu8 — an vector of unsigned 8-bit integers (when there is a difference between what an operation would do with signed and unsigned numbers, such as with conversion to a larger integer or multiplication)
- ★ epu16, epu32 — an array of unsigned 16-bit integers or 8 unsigned 32-bit integers (when the operation would be different than signed)
- ★ ps — “packed single” — 8 single-precision floats
- ★ pd — “packed double” — 4 doubles
- ★ ss — one float (only 32-bits of a 256-bit or 128-bit value are used)
- ★ sd — one double (only 64-bits of a 256-bit or 256-bit value are used

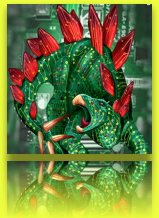


Example in C

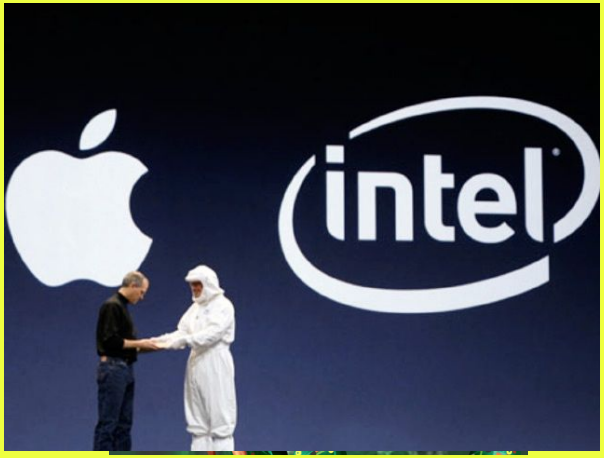
- ★ Assuming that we want to add two set of integers together
- ★ Use “gcc -mavx2”

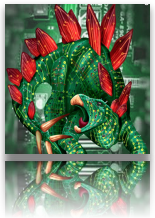
```
// No AVX
void add(int size, int *a,int *b) {

for (int i=0;i<size;i++) {
     a[i] += b[i];
}

}
```

```
// with AVX2
void add_avx(int size, int *a, int*b) {
int i=0;
for (;i<size;i+=8) {
// load 256-bit chunks of each array
__m256i av = _mm256_loadu_si256((__m256i*) &a[i]);
__m256i bv = _mm256_loadu_si256((__m256i*) &b[i]);

// add each pair of 32-bit integers in chunks
av = _mm256_add_epi32(av, bv);

// store 256-bit chunk to a
_mm256_storeu_si256((__m256i*) &a[i], av);
}
// clean up
for(;i<size;i++) {
   a[i] += b[i];
}
}
```

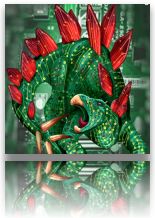


Intel-based Mac OS X enforces 16-byte memory alignment. Why?





How about other programming languages?



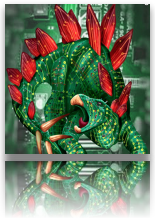
Try python

- ★ Use special libraries
- ★ Numpy for vectorization (eg. AVX)
- ★ CuPy for CUDA

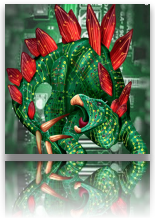
```
a=list(range(6400000))
b=list(range(6400000))
for i in range(size):
    a[i]+=b[i]
```

```
import numpy as np

na=np.array(a)
nb=np.array(b)

na+=nb
```

```
import cupy as np

na=np.array(a)
nb=np.array(b)

na+=nb
```



Exercise

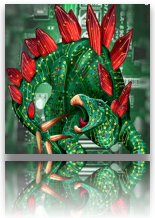


Exercise I

Design an experiment to measure the speed up of this vectorize code.

```
// No AVX
void add(int size, int *a,int *b) {

for (int i=0;i<size;i++) {
     a[i] += b[i];
}

}
```

```
// with AVX2
void add_avx(int size, int *a, int*b) {
int i=0;
for (;i<size;i+=8) {
// load 256-bit chunks of each array
__m256i av = _mm256_loadu_si256((__m256i*) &a[i]);
__m256i bv = _mm256_loadu_si256((__m256i*) &b[i]);

// add each pair of 32-bit integers in chunks
av = _mm256_add_epi32(av, bv);

// store 256-bit chunk to a
_mm256_storeu_si256((__m256i*) &a[i], av);
}
// clean up
for(;i<size;i++) {
   a[i] += b[i];
}
}
```

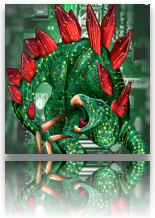


Exercise II

Design an experiment to measure the speed up of this in numpy (and cupy it you have a GPU)

```
a=list(range(6400000))
b=list(range(6400000))
for i in range(size):
    a[i]+=b[i]
```

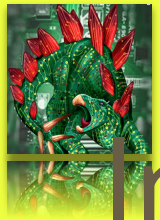
```
import numpy as np

na=np.random.randint(1,1000, 6400000)
nb=np.random.randint(1,1000, 6400000)

na+=nb
```



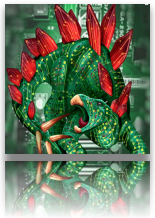
Exercise III

While vectorization is powerful, please explain a situation when it may not be beneficial.

Hint 1 - Compiler support
Hint 2 - Vectorizability of Software



In python, numpy can be used to eliminate loop in a program. (Machine Learning uses it)



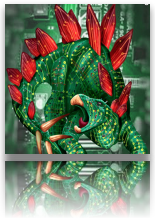
Vectorize Algorithms

Trends in optimization

For some difficult algorithms, vectorization can gain more performance improvement.

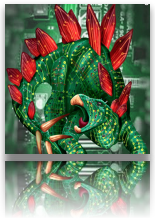
Eg. tensorflow, pytorch, etc...

I have seen a lot of graduate students do it as theses. (Both AVX and CUDA)



Just for fun

- ★ AVX512 - https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-animation.html



End of Lecture 3