Credits

• Marat Dukhan (Georgia Tech)
  – http://www.cc.gatech.edu/grads/m/mdukhan3/
• Salvatore Orlando (Univ. Ca' Foscari di Venezia)
Single Instruction Multiple Data

- In the SIMD model, the same operation can be applied to multiple data items.
- This is usually realized through special instructions that work with short, fixed-length arrays.
  - E.g., SSE and ARM NEON can work with 4-element arrays of 32-bit floats.

**Scalar instruction**

<table>
<thead>
<tr>
<th>a</th>
<th>13.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>12.3</td>
</tr>
<tr>
<td>a + b</td>
<td>25.3</td>
</tr>
</tbody>
</table>

**SIMD instruction**

<table>
<thead>
<tr>
<th>a</th>
<th>13.0</th>
<th>7.0</th>
<th>-3.0</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>12.3</td>
<td>4.4</td>
<td>13.2</td>
<td>-2.0</td>
</tr>
<tr>
<td>a + b</td>
<td>25.3</td>
<td>11.4</td>
<td>10.2</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Programming for SIMD

- Compiler auto-vectorization
- Optimized SIMD libraries
  - ATLAS, FFTW
- Domain Specific Languages (DSLs) for SIMD programming
  - E.g., Intel SPMD program compiler
- Compiler-dependent vector data types
- Compiler SIMD intrinsics
- Assembly language
A note of caution

- Most (all?) modern processors have SIMD instructions that can provide a considerable performance improvement
  - Up to 4x / 8x theoretical speedup for 4/8 lanes SIMD
  - Superlinear speedup can be observed in some cases
- However
  - Compilers are not very smart at auto-vectorization
  - Using SIMD instructions by hand is cumbersome, error-prone and non-portable
    - Each processor has its own set of SIMD instructions
    - Sometimes, different models from the same processor family have different SIMD instruction sets
    - Yes, I'm looking at you Intel
Intel SIMD extensions timeline

**MMX [1993]:** 8 64-bit vector registers, supports only integer operations

**SSE [1999]:** 8 or 16 128-bit vector registers, supports single-precision floating point operations

**SSE2 [2000]:** 8 or 16 128-bit vector registers, supports double-precision floating point operations – later extensions: SSE3, SSSE3, SSE4.1, SSE4.2

**AVX [2008]:** supported in the Intel Sandy Bridge processors, and later extends the vector registers to 16 registers of length 256 bits

**AVX2 [2013]:** introduced in the Haswell microarchitecture, extends the vector registers to 16 256-bit registers

**AVX-512 [2015]:** proposed in July 2013 and supported since 2015 with Intel's Knights Landing processor
Intel SSE/AVX

- **SSE**
  - 16 × 128-bit SIMD registers
    - XMM0—XMM15 (in x86-64 mode)
- **AVX2**
  - 16 × 256-bit SIMD registers
    - YMM0—YMM15
- **AVX-512**
  - 32 × 512-bit SIMD registers
    - ZMM0—ZMM31

**SSE/AVX types**

- 4x floats: 32
- 2x double: 64
- 16x chars: 8
- 8x 16-bit words: 16
- 4x 32-bit doublewords: 32
- 2x 64-bit quadwords: 64
- 1x 128-bit doublequadword: 128

**AVX2 types**

- 8x floats: 32
- 4x double: 64

**AVX2 types**

**Source:** [https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf](https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf)
ARM NEON

- The NEON register bank consists of 32 64-bit registers
- The NEON unit can view the same register bank as:
  - $16 \times 128$-bit quadword registers $Q0$–$Q15$
  - $32 \times 64$-bit doubleword registers $D0$–$D31$

Is it worth the effort?

• In some cases, yes
  - Example: BML cellular automaton ($N = 1024$, $\rho = 0.3$, steps = 1024), CPU Xeon E3-1220 @ 3.10GHz 4 cores (no ht) + GCC 4.8.4, 16GB RAM; GPU nVidia Quadro K620 (384 CUDA cores) + nvcc V8.0.61

<table>
<thead>
<tr>
<th>Method</th>
<th>Speedup with respect to serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>1</td>
</tr>
<tr>
<td>OpenMP (4 cores)</td>
<td>2.76</td>
</tr>
<tr>
<td>OpenMP + halo (4 cores)</td>
<td>5.16</td>
</tr>
<tr>
<td>CUDA (no shmem)</td>
<td>8.42</td>
</tr>
<tr>
<td>CUDA (shmem + optim)</td>
<td>34.53</td>
</tr>
<tr>
<td>SIMD x16 (1 core)</td>
<td>66.52</td>
</tr>
<tr>
<td>SIMD x16 (4 cores)</td>
<td>180.56</td>
</tr>
</tbody>
</table>

Speedup with respect to serial (higher is better)
Superlinear speedup?

```c
for (i=0; i<n; i++)
    Compute(i);

SIMD Programming
```
Superlinear speedup?

```c
for (i=0; i<n-3; i+=4)
    Compute(i)
    Compute(i+1)
    Compute(i+2)
    Compute(i+3)
}
```

handle leftovers...
Superlinear speedup?

```c
for (i=0; i<n-3; i+=4)
    SIMD_Compute(i, ..., i+3)
}
handle leftovers...
```
Checking for SIMD support on Linux

cat /proc/cpuinfo

- On Intel

flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpre pdcm pciid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_l1m abm 3dnowprefetch epb intel_pt tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm rdseed adx smap xsaveopt cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm arat pln pts

- On ARM

Features : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpu32 lpaes ept vstrm crc32
Vectorization opportunities

```c
float vsum(float *v, int n) {
    float s = 0.0; int i;
    for (i=0; i<n; i++)
        s += v[i];
    return s;
}
```

3  -1  2  0  -4  7  -5  11  2  8  21  3  -3  7  11  12
Vectorization opportunities

```c
float vsun(float *v, int n)
{
    float s = 0.0; int i;
    for (i=0; i<n; i++)
        s += v[i];
    return s;
}
```
Vectorization opportunities

```c
float vs_sum(float *v, int n)
{
    float s = 0.0; int i;
    for (i=0; i<n; i++)
        s += v[i];
    return s;
}
```

---

SIMD Programming
Vectorization opportunities

```c
float vsum(float *v, int n) {
    float s = 0.0; int i;
    for (i=0; i<n; i++)
        s += v[i];
    return s;
}
```

v 3 -1 2 0 -4 7 -5 11 2 8 21 3 -3 7 11 12

s -1 6 -3 11

s' 1 14 18 14
Vectorization opportunities

float vsum(float *v, int n)
{
    float s = 0.0; int i;
    for (i=0; i<n; i++)
        s += v[i];
    return s;
}
Vectorization opportunities

float vsun(float *v, int n)
{
    float s = 0.0; int i;
    for (i=0; i<n; i++)
        s += v[i];
    return s;
}
Vectorization opportunities

- In our example, care must be taken if the array length is not multiple of SIMD vector length

- **Padding**
  - Add “dummy” elements at the beginning/end to make the array length multiple of SIMD vector length
  - Not always feasible (requires to modify the input; choosing the values of the extra elements can be tricky and is problem-dependent)

- **Handle the leftovers with scalar operations**
  - Possible (small?) performance hit
  - Redundant code – therefore error-prone – if done by hand
```c
float vsum(float *v, int n) {
    float s = 0.0; int i;
    for (i=0; i<n; i++)
        s += v[i];
    return s;
}

float vsum(float *v, int n) {
    float vs[4] = {0.0, 0.0, 0.0, 0.0};
    float s = 0.0;
    int i;
    for (i=0; i<n-4; i += 4) {
        vs[0] += v[i ];
        vs[1] += v[i+1];
        vs[2] += v[i+2];
        vs[3] += v[i+3];
    }
    /* Handle leftover */
    for (; i<n; i++) {
        s += v[i];
    }
    return s;
}
```

Not really SIMD; assumes 4-lanes SIMD
Auto-vectorization
Compiler auto-vectorization

• To enable auto-vectorization with GCC
  
  `-O2`
  `-ftree-vectorize`
  `-fopt-info-vec-optimized`
  `-fopt-info-vec-missed`
  `-march=native`

  You might need to turn on optimization to enable auto-vectorization
  Enable debugging output to see what gets vectorized and what does not

  Autodetect and use SIMD instructions available on your platform

• Warning: if you enable platform-specific optimizations, your binary will not run on less capable processors!

• To see which flags are enabled with `-march=native`
  
  `gcc -march=native -Q --help=target`
Selection of the target architecture
the hard way

• On Intel:
  – -msse emits SSE instructions
  – -msse2 emits SIMD instructions up to SSE2
  – ...
  – -msse4.2 emits SIMD instructions up to SSE4.2
  – -mavx emits SIMD instructions up to AVX (incl. SSEx)
  – -mavx2 emits SIMD instructions up to AVX2

• On ARM (RaspberryPI2/3)
  – -march=armv7-a -mfpu=neon \  
    -mvectorize-with-neon-quad

• On ARM (NVIDIA Jetson TK1 dev. board)
  – -march=armv7 -mfpu=neon \  
    -mvectorize-with-neon-quad
Auto-vectorization

• Let's try GCC auto-vectorization capabilities

```
gcc -march=native -O2 -ftree-vectorize -fopt-info-vec-optimized -fopt-info-vec-missed simd-vsum-auto.c -o simd-vsum-auto
```

```
simd-vsum-auto.c:45:5: note: step unknown.
simd-vsum-auto.c:45:5: note: reduction: unsafe fp math optimization:
s_10 = _9 + s_14;
...
...
simd-vsum-auto.c:54:5: note: Unknown misalignment, is_packed = 0
simd-vsum-auto.c:54:5: note: loop vectorized
```
Auto-vectorization

• GCC was unable to vectorize this loop:

```c
float vsum(float *v, int n)
{
    float s = 0.0; int i;
    for (i=0; i<n; i++)
        s += v[i];
    return s;
}
```

• The output gives us a hint:

```
simd-vsum-auto.c:45:5: note: reduction: unsafe fp math
optimization: s_10 = _9 + s_14;
```
Auto-vectorization

• In this case, GCC is smarter than we are
• The familiar rules of infinite-precision math do not hold for finite-precision math
  – FP addition is not associative, i.e., \((a + b) + c\) could produce a different result than \(a + (b + c)\)
  – Therefore, GCC by default does not apply optimizations that violate FP safety
• We can tell GCC to optimize anyway with \(-\text{funsafe-math-optimizations}\)
Bingo!


... 
simd-vsum-auto.c:45:5: note: loop vectorized 
...
simd-vsum-auto.c:54:5: note: loop vectorized 
...
Auto-vectorization

- You can look at the generated assembly output with

```bash
gcc -S -c -funsafe-math-optimizations -march=native -O2 -ftree-vectorize simd-vsum-auto.c -o simd-vsum-auto.s
```

```
.L4:
    movq   %rcx, %r8
    addq   $1, %rcx
    salq   $5, %r8
    vaddps (%rdi,%r8), %ymm0, %ymm0
    cmpl   %ecx, %edx
    ja     .L4
    vhaddps %ymm0, %ymm0, %ymm0
    vhaddps %ymm0, %ymm0, %ymm1
    vperm2f128 $1, %ymm1, %ymm1, %ymm0
    vaddps %ymm1, %ymm0, %ymm0
    cmpl   %esi, %eax
    je     .L17
```

*xxxps instructions are those dealing with Packed Single-precision SIMD registers*
Auto-Vectorization
the Good, the Bad, the Ugly

In theory, auto-vectorization is totally transparent. However, in might be beneficial to provide hints to the compiler (e.g., alignment info, usage of C99 restrict keyword) to help it do a better job.

Even when auto-vectorization is possible, the compiler often generates sub-optimal code.

Regardless how good the compiler is, there are many cases where auto-vectorization fails. Then you are on your own.
Vector data type
Vector data types

- Some compilers support vector data types
  - Vector data types are non portable, compiler-specific extensions
- Vector data types as (as the name suggests) small vectors of some numeric type
  - typically, char, int, float, double
- Ordinary arithmetic operations (sum, product...) can be applied to vector data types
- The compiler emits the appropriate SIMD instructions for the target architecture if available
  - If no appropriate SIMD instruction is available, the compiler emits equivalent scalar code
Definition

• Defining vector data types

```c
/* v4i is a vector of elements of type int; variables of type v4i occupy 16 bytes of contiguous memory */
typedef int v4i __attribute__((vector_size(16)));

/* v4f is a vector of elements of type float; variables of type v4f occupy 16 bytes of contiguous memory */
typedef float v4f __attribute__((vector_size(16)));
```

• The length (num. of elements) of v4f is

```c
#define VLEN (sizeof(v4f)/sizeof(float))
```
Definition

• Is it possible to define vector types or “arbitrary” length

/* v8f is a vector of elements of type float; variables of type v8f occupy 32 bytes of contiguous memory */
typedef float v8f __attribute__((vector_size(32)));

• If the target architecture does not support SIMD registers of the specified length, the compiler takes care of that
  – e.g., using multiple SIMD instructions on shorter vectors
/* simd-vsum-vector.c */

typedef float v4f __attribute__((vector_size(16)));
#define VLEN (sizeof(v4f)/sizeof(float))

float vsum(float *v, int n)
{
    v4f vs = {0.0f, 0.0f, 0.0f, 0.0f};
    v4f *vv = (v4f*)v;
    int i; float s = 0.0f;
    for (i=0; i<n-VLEN+1; i += VLEN) {
        vs += *vv;
        vv++;
    }
    for ( ; i<n; i++) {
        s += v[i];
    }
    return s;
}
Vector data types

• GCC allows the following operators on vector data types
  +, -, *, /, unary minus, ^, |, & , ~, %

• It is also possible to use a binary vector operation where one operand is a scalar

```c
typedef int v4i __attribute__((vector_size (16)));

v4i a, b, c;
long x;

a = b + 1;  // OK: a = b + {1,1,1,1};
a = 2 * b;  // OK: a = {2,2,2,2} * b;
a = 0;      // Error: conversion not allowed
a = x + a;  // Error: cannot convert long to int
```
Vector data types

• Vector comparison is supported with standard comparison operators: ==, !=, <, <=, >, >=
• Vectors are compared element-wise
  – 0 when comparison is false
  – -1 when comparison is true

```c
typedef int v4i __attribute__((vector_size (16)));

v4i a = {1, 2, 3, 4};
v4i b = {3, 2, 1, 4};
v4i c;

c = (a > b);    /* Result {0, 0,-1, 0} */
c = (a == b);   /* Result {0,-1, 0,-1} */
```

https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html
Note on memory alignment

- Some versions of GCC emit assembly code for dereferencing a pointer to a vector datatype that only works if the memory address is 16B aligned
  - More details later on

- `malloc()` may or may not return a pointer that is properly aligned. From the man page:
  
  "The malloc() and calloc() functions return a pointer to the allocated memory, which is suitably aligned for any built-in type."
  

?????
Ensuring proper alignment

- For data on the stack:

```c
/* __BIGGEST_ALIGNMENT__ is 16 for SSE, 32 for AVX; it is therefore the preferred choice as it is automatically defined to suit the target */

float v[1024] __attribute__((aligned(__BIGGEST_ALIGNMENT__)));
```

- For data on the heap:

```c
#define _XOPEN_SOURCE 600
#include <stdlib.h>

float *v;
posix_memalign(&v, __BIGGEST_ALIGNMENT__, 1024);
```

Do this at the very beginning (before including anything else); better yet, compile with the `-D_XOPEN_SOURCE=600` flag to define the symbol compilation-wide.
SIMDizing branches

• Branches (if-then-else) are difficult to SIMDize, since the SIMD programming model assumes that the same operation is applied to all elements of a SIMD register.

• How can we SIMDize the following code fragment?

```c
int a[4] = { 12, -7, 2, 3 };
int i;

for (i=0; i<4; i++) {
    if ( a[i] > 0 ) {
        a[i] = 2;
    } else {
        a[i] = 1;
    }
}
```
```c
v4i a = { 12, -7, 2, 3 };
v4i vtrue = {2, 2, 2, 2};
v4i vfalse = {1, 1, 1, 1};

v4i mask = (a > 0); /* mask = {-1, 0, -1, -1} */
a = (vtrue & mask) | (vfalse & ~mask);
```
Data layout: SoA vs AoS

- **Arrays of Structures (AoS)**

```c
typedef point3d {
    float x, y, z, t;
};
#define NP 1024
point3d particles[NP];
```

- **Structures of Arrays (SoA)**

```c
#define NP 1024
float particles_x[NP];
float particles_y[NP];
float particles_z[NP];
float particles_t[NP];
```
Programming with SIMD intrinsics
SIMD intrinsics

• The compiler exposes all low-level SIMD operations through C functions
  – Each function maps to the corresponding low-level SIMD instruction
  – In other words, you are programming in assembly
• SIMD intrinsics are platform-dependent
  – since different processors have different instruction sets
• However
  – SIMD intrinsics are machine-dependent but compiler-independent
  – Vector data types are machine-independent but compiler-dependent
Vector sum with SSE intrinsics

/* simd-vsum-intrinsics.c */
#include <x86intrin.h>

float vsum(float *v, int n)
{
    __m128 vv, vs;
    float s = 0.0f;
    int i;

    vs = _mm_setzero_ps();
    for (i=0; i<n-4+1; i += 4) {
        vv = _mm_loadu_ps(&v[i]);
        vs = _mm_add_ps(vv, vs);
    }
    for (; i<n; i++) {
        s += v[i];
    }
    return s;
}
SSE intrinsics

- `#include <x86intrin.h>`
- Three new datatypes
  - `__m128`
    - Four single-precision floats
  - `__m128d`
    - Two double-precision floats
  - `__m128i`
    - Two 64-bit integers
    - Four 32-bit integers
    - Eight 16-bit integers
    - Sixteen 8-bit integers
SSE memory operations

- `__m128 _mm_loadu_ps(float *aPtr)`
  - Load 4 floats starting from memory address `aPtr`
- `__m128 _mm_load_ps(float *aPtr)`
  - Load 4 floats starting from memory address `aPtr`
  - `aPtr` must be a multiple of 16
- `_mm_storeu_ps(float *aPtr, __m128 v)`
  - Store 4 floats from `v` to memory address `aPtr`
- `_mm_store_ps(float *aPtr, __m128 v)`
  - Store 4 floats from `v` to memory address `aPtr`
  - `aPtr` must be a multiple of 16

- There are other intrinsics for load/store of doubles/ints

\[ u = \text{unaligned} \]
How values are stored in memory

- SSE SIMD registers are stored in memory as a0, a1, a2, a3 in increasing memory locations.
Some SSE arithmetic operations

- Do operation <op> on a and b, write result to c:
  
  ```c
  __m128 c = _mm_<op>_ps(a, b);
  ```

- Addition: c = a + b
  - c = _mm_add_ps(a, b);

- Subtraction: c = a − b
  - c = _mm_sub_ps(a, b);

- Multiplication: c = a * b
  - c = _mm_mul_ps(a, b);

- Division: c = a / b
  - c = _mm_div_ps(a, b);

- Minimum: c = fmin(a, b)
  - c = _mm_min_ps(a, b);

- Maximum: c = fmax(a, b)
  - c = _mm_max_ps(a, b);

- Square root: c = sqrt(a)
  - c = _mm_sqrt_ps(a);
Load constants

\[ v = \_\text{mm\_set\_ps}(a, b, c, d) \quad \text{float} \ d = \_\text{mm\_cvts\_f32}(v) \]

\[
\begin{array}{cccc}
a & b & c & d \\
\end{array}
\]

\[ v = \_\text{mm\_set1\_ps}(a) \]

\[
\begin{array}{cccc}
a & a & a & a \\
\end{array}
\]

\[ v = \_\text{mm\_setzero\_ps}() , \]

\[
\begin{array}{cccc}
0.0 & 0.0 & 0.0 & 0.0 \\
\end{array}
\]
Resources

- Intel Intrinsics Guide