High Performance Programming, Lecture 8

SIMD/vectorization, performance analysis

Assignments

Notes about debugging:

- understand the problem
- test often! (write separate tests for your functions, test each smallest testable part of an application)
- use the smallest possible test case
- use debugger
- try to explain each step in your code to someone (of at least say it aloud)

Assignments

Don't forget Assignment 4!

Deadline for Assignment 4: **Friday, Feb 22**.

You are welcome to ask questions now, or by e-mail later.

n-body problem

Evolution of the galaxy (N stars) in a selected space region.

Simple approach: each star acts with some force to other N-1 stars. Then the total number of force calculations is $\mathcal{O}(N^2)$.

Use approximation! A tree based approximation scheme reduces the computational complexity of the problem from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$ (The Barnes-Hut algorithm).

The basic idea: if the group is sufficiently far away, we can approximate its gravitational effects by using its center of mass.

Assignment issues

In each timestep

In each timestep:

1. **First** compute forces for all particles
2. **Then** use the computed forces to update velocities and positions of all particles

In Assignment 4, the Barnes-Hut method for computing the forces is to be used at each timestep. It is OK to delete the old tree and create a new tree structure in each timestep.

If you want a hint about how a tree structure can be created in C, look at “`tree_example.c`” available under “Example programs” in Student Portal. See also BST part in Lab 2.

Questions about assignments?

Questions now?

Reading

About **SIMD/vectorization**:

- Fog 12 : "Using vector operations"
- Fog 13 : "... different instruction sets" (mostly look at first part there, page 125, before 13.1)

About **performance analysis**:

- Fog 3.1 : "How much is a clock cycle?"
- Fog 3.2 : "Use a profiler to find hot spots"

For next module, about **Pthreads**:

- Pacheco chapter 1: "Why Parallel Computing?"
- Pacheco chapter 2: "Parallel Hardware and Parallel Software"
- Pacheco chapter 4: "Shared-Memory Programming with Pthreads"

Parallelism

Modern computer systems have **several levels of parallelism**:

- process level parallelism (PLP, message passing) – *other course*
- thread level parallelism (TLP) – *next lectures*
- instruction level parallelism (ILP) – *previous lectures*
- data level parallelism (data processing by several vector arithmetic logic units) – *today!*

SIMD instructions

Single instruction, multiple data (SIMD)

Allows increased performance by doing the same operation for several data entries at once.

Instructions must be issued that explicitly use vector registers and vector operations.

Examples: image and sound processing, mathematical operations on vectors and matrices

Vector operations

Example (1)

Example: floating-point multiplication with 4 numbers in each vector. Operation needs vectors of 4 elements to operate on.

For example, 4 elements a1, a2, a3, a4 multiplied by 4 elements b1, b2, b3, b4:

```
------------------------------------
    vector vector          vector
    a      b               result
------------------------------------
    a1     b1     --->     a1*b1
    a2     b2     --->     a2*b2
    a3     b3     --->     a3*b3
    a4     b4     --->     a4*b4
------------------------------------
```

Vector operations

Example (2)

If each vector register can fit 8 numbers, the operation becomes:

```
------------------------------------
    vector vector          vector
    a      b               result
------------------------------------
    a1     b1     --->     a1*b1
    a2     b2     --->     a2*b2
    a3     b3     --->     a3*b3
    a4     b4     --->     a4*b4
    a5     b5     --->     a5*b5
    a6     b6     --->     a6*b6
    a7     b7     --->     a7*b7
    a8     b8     --->     a8*b8
------------------------------------
```

How can we use vector instructions?

- **inline assembler** – full control of vectorization usage, least portable approach
- **intrinsics** (*today and lab 4*) – set of data types and internal compiler functions, directly mapping to processor instructions (vector registers are allocated by compiler)
- **SIMD directives** of compilers (OpenMP and OpenACC standards)
- **language extensions** (Intel Array Notation, Intel ISPC, Apple Swift SIMD) and **libraries** (C++17 SIMD Types, Boost.SIMD, SIMD.js)
- **libraries** (MKL, OpenBLAS, BLIS, fftw, numpy, OpenCV)
- **automatic vectorizing compiler** (*today and lab 4*) – ease of use, high code portability.

http://2017.russianscdays.org/files/pdf17/55.pdf

SIMD/vectorization

Streaming SIMD Extensions (SSE)
Advanced Vector Extensions (AVX) an advanced version of SSE

The vector operations use a set of special vector registers:

- 64 bit (MMX)
- 128 bit (XMM) – SSE
- 256 bit (YMM) – AVX
- 512 bit (ZMM) – AVX-512

SIMD/vectorization

Different SSE versions (1)

Instruction set	Important features
SSE ($\sim$ 1999)	128-bit float vectors
SSE2 ($\sim$ 2001)	128-bit int and double vectors
SSE3 ($\sim$ 2004)	horizontal add, etc.
SSSE3 ($\sim$ 2006)	a few more int vector instructions
SSE4.1	some more vector instructions
SSE4.2	string search instructions
AVX ($\sim$ 2008)	256-bit float and double vectors
AVX2 ($\sim$ 2013)	256-bit int vectors
FMA3	floating-point multiply-and-add
AVX-512 ($\sim$ 2017)	512-bit int and floating-point vectors

(Info mostly from Table 13.1 in Fog book)

SIMD/vectorization

Different SSE versions (2)

Header file	Extension name	Abbrev.
`xmmintrin.h`	Streaming SIMD Extensions	SSE
`emmintrin.h`	Streaming SIMD Extensions 2	SSE2
`pmmintrin.h`	Streaming SIMD Extensions 3	SSE3
`tmmintrin.h`	Supplemental Streaming SIMD Extensions 3	SSSE3
`smmintrin.h`	Streaming SIMD Extensions 4 (Vector math)	SSE4.1
`nmmintrin.h`	Streaming SIMD Extensions 4 (String proc.)	SSE4.2
`immintrin.h`	Advanced Vector Extensions Instructions	AVX

Need to `#include` an appropriate header file to be able to use corresponding vector types and intrinsic functions.

SIMD/vectorization

Intel Intrinsics Guide

Good resource for looking up specific intrinsic functions:
Intel Intrinsics Guide:
`https://software.intel.com/sites/landingpage/IntrinsicsGuide/`

(Used in Lab 8)

SIMD/vectorization

```
__m128i _mm_loadu_si128 (__m128i const* mem_addr)

#include <emmintrin.h>
CPUID Flags: SSE2

Description:
Load 128-bits of integer data from memory into dst.
   mem_addr does not need to be aligned on any
   particular boundary.

Operation: dst[127:0] := MEM[mem_addr+127:mem_addr]
```

Here, "`__m128i`" is a **vector type** and the function `_mm_loadu_si128(...)` is an example of an **intrinsic function**.

SIMD/vectorization

```
__m256i _mm256_loadu_si256 (__m256i const * mem_addr)

#include <immintrin.h>
CPUID Flags: AVX

Description:
Load 256-bits of integer data from memory into dst.
   mem_addr does not need to be aligned on any
   particular boundary.

Operation:
dst[255:0] := MEM[mem_addr+255:mem_addr]
dst[MAX:256] := 0
```

Example load/store SSE3

```
#include <pmmintrin.h>   /* SSE3 */

#define VECT_LEN 16   /* the length of the vector */
...
/* length of the array must be a multiple of the vector
    size VECT_LEN */
for(i = 0; i < len; i += VECT_LEN )
{
    __m128i v = _mm_loadu_si128((__m128i *) (src+i));
    _mm_storeu_si128((__m128i *)(dst+i), v);
}
...
```

See folder `test_load_store`.
Compile: `gcc -o main_sse main_sse.c` and run `./main_sse`.

Note: gcc compiler implicitly supports SSE instruction set.

Example load/store AVX

```
#include <immintrin.h>   /* AVX */

#define VECT_LEN 32   /* the length of the vector */
...
/* length of the array must be a multiple of the vector
    size VECT_LEN */
for(i = 0; i < len; i += VECT_LEN )
{
    __m256i v = _mm256_loadu_si256((__m256i *) (src+i));
    _mm256_storeu_si256((__m256i *)(dst+i), v);
}
...
```

Try to compile: `gcc -o main_avx main_avx.c` and run `./main_avx`.

Example load/store AVX

```
#include <immintrin.h>   /* AVX */

#define VECT_LEN 32   /* the length of the vector */
...
/* length of the array must be a multiple of the vector
    size VECT_LEN */
for(i = 0; i < len; i += VECT_LEN )
{
    __m256i v = _mm256_loadu_si256((__m256i *) (src+i));
    _mm256_storeu_si256((__m256i *)(dst+i), v);
}
...
```

Try to compile: `gcc -o main_avx main_avx.c` and run `./main_avx`. Oh, no! We get "... target specific option mismatch" error. To make it work we should allow to the compiler to use AVX instruction set. Add `-mavx`: `gcc -mavx -o main_avx main_avx.c`

Example load/store AVX512

```
#include <immintrin.h>   /* AVX and AVX512*/

#define VECT_LEN 64   /* the length of the vector */
...
/* length of the array must be a multiple of the vector
    size VECT_LEN */
for(i = 0; i < len; i += VECT_LEN )
{
    __m512i v = _mm512_loadu_si512((__m512i *) (src+i));
    _mm512_storeu_si512((__m512i *)(dst+i), v);
}
...
```

Try to compile: `gcc -o main_avx512 main_avx512.c` and run `./main_avx512`.

Example load/store AVX512

```
#include <immintrin.h>   /* AVX and AVX512*/

#define VECT_LEN 64   /* the length of the vector */
...
/* length of the array must be a multiple of the vector
    size VECT_LEN */
for(i = 0; i < len; i += VECT_LEN )
{
    __m512i v = _mm512_loadu_si512((__m512i *) (src+i));
    _mm512_storeu_si512((__m512i *)(dst+i), v);
}
...
```

Try to compile: `gcc -o main_avx512 main_avx512.c` and run `./main_avx512`.
We get "... target specific option mismatch" error again. To make it work we should allow to the compiler to use AVX512 instruction set. Add `-mavx512f`: `gcc -mavx512f -o main_avx512 main_avx512.c`

SIMD/vectorization

gcc compiler options

Note that you can compile with e.g. `-mavx512f` even if that instruction set is not supported by the computer you are using to compile the code.

Then you get an executable that will not run on that computer. Trying to run it will give “illegal instruction” errors. But, it will work on other computers supporting that instruction set.

Compiling for another architecture is called *cross compiling*.

SIMD/vectorization

Checking what is supported on your computer

On Linux systems, check which instruction sets are available by looking at flags in `/proc/cpuinfo` (sometimes also given by `lscpu` command, but not always)

On Mac OS:

```
sysctl -a | grep machdep.cpu
```

Example (my computer Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz):

flags : ... **mmx** fxsr **sse sse2** ss ht ... tm2 **ssse3** sdbg **fma** cx16 xtpr pdcm pcid **sse4_1 sse4_2** x2apic movbe ... aes xsave **avx** f16c rdrand lahf_lm abm ... fsgsbase tsc_adjust bmi1 **avx2** smep bmi2 erms ...

(no explicit SSE3 flag, it is implied by **ssse3**)

SIMD/vectorization

Aligned vs unaligned load/store functions

Alignment is a property of a memory address. Data are **naturally aligned** if the address is a multiple of the data size.

Misaligned memory accesses can incur large performance losses on certain target processors that do not support them in hardware.

There are different variants of load/store functions depending on if the data are aligned.

Example:
`__m128i _mm_loadu_si128 (__m128i const* mem_addr)`
"Load 128-bits of integer data from memory into dst. `mem_addr` **does not need to be aligned** on any particular boundary."

Alternative:
`__m128i _mm_load_si128 (__m128i const* mem_addr)`
"Load 128-bits of integer data from memory into dst. `mem_addr` **must be aligned on a 16-byte boundary** or a general-protection exception may be generated."

GCC autovectorization flags

Enable autovectorization using GCC optimization flags `-O3` or `-ftree-vectorize`.

Get **details of autovectorization results** using the compiler flags:

- `-fopt-info-vec` or `-fopt-info-vec-optimized`: The compiler will log which loops are being vector optimized.
- `-fopt-info-vec-missed`: Detailed info about loops not being vectorized, and a lot of other detailed information.
- `-fopt-info-vec-note`: Detailed info about all loops and optimizations being done.
- `-fopt-info-vec-all`: All previous options together.

Note: use `-ftree-vectorizer-verbose=2` instead of `-fopt-info-vec` for **older gcc versions**.

Disable vectorization: `-fno-tree-vectorize`.

clang autovectorization flags

Get **details of autovectorization results** using the compiler flags:

- `-Rpass=loop-vectorize` identifies loops that were successfully vectorized.
- `-Rpass-missed=loop-vectorize` identifies loops that failed vectorization and indicates if vectorization was specified.
- `-Rpass-analysis=loop-vectorize` identifies the statements that caused vectorization to fail.

auto-vectorization

– The compiler needs to prove that the optimization is legal
– And the compiler needs to prove that the optimization is beneficial (under almost all circumstances)

Criteria for loop vectorization:

- The loop count can't change once the loop starts (but can be unknown at the compile time).
- Conditionals sentences can be used if they don't change the control flow
- If a loop is part of a loop nest, it should be the inner loop.
- No function calls (except built-in math functions and inlined functions)

auto-vectorization

Obstacles to Vectorization:

- Uncountable loop
- Non-contiguous memory access (data must be loaded separately using multiple instructions)
- Data dependencies (vectorization changes the order of operations within a loop)
- External functions inside a loop (unless these are inlined)
- Conditionals sentences (if/else, break, continue) which cannot be removed by the compiler (cannot be rewritten using bitwise operations)

Advise: use the latest compiler version.

auto-vectorization

See code in the `vectorization` folder.

Compile it using `make`. See compiler flags CFLAGS inside make: `-Wall -O3 -fopt-info-vec`.

```
main.c:38:9: note: loop vectorized
function.c:5:9: note: loop vectorized
function.c:5:9: note: loop versioned for
   vectorization because of possible aliasing
```

The last note means that the loop is scheduled for the *runtime alias check* (check if the arrays overlap).

Try to add the `restrict` keyword to the vectors in the function `sum_arrays` in `function.c`. Recompile and run. Any difference with the vectorization output?

auto-vectorization

Try to change the for loop conter to
`for (size_t i=0; i<N; i+=2)` in `sum_arrays`.

Enable matrix multiplication changing `#if 0` to `#if 1` in *all* files.
Is `mult_kij` vectorized? What is you change loop order (exchange places of i and j loops)?

Try to add `if(a[i] < 0.1)` inside for loop in `sum_arrays`.
Vectorization?
Also, try to change 0.1 to 50. Any difference in time?

auto-vectorization

What of you add this instead:

```
if(a[i] < 0.1)
   c[i] = a[i];
else
   c[i] = 0;
```

Check also loops in the main function:
`for(i = 0; i < N; i++) sum1 += dst1[i];`.
`for(i = 0; i < N; i++) src1[i] = sin(dst1[i]);`. Is it vectorized (depends on the compiler)? If not, how can we allow its vectorization? Also, try to use technique as loop fusion.

Enable string operations changing `#if 0` to `#if 1` in *all* files. Are loop inside `lowercase1` and `lowercase2` vectorized?

SIMD/vectorization

Aligning data using attributes (gcc)

Static arrays:

```
float foo[SIZE] __attribute__((aligned (16)));
```

Dynamic arrays: (use SSE intrinsics)

```
void _mm_free (void * mem_addr)  // Allocate size bytes of
     memory, aligned to the alignment specified in align
void* _mm_malloc (size_t size, size_t align)  // Free
    aligned memory that was allocated with _mm_malloc
```

The preferable alignment for 128-bit vectors is 16 and for 256-bit vectors is 32.

Data structure alignment example

```
void fill(char *x){
   for (int i = 0; i < 1024; i++)
     x[i] = 1;}
```

The compiler may decide to vectorize this loop using unaligned data movement instructions or generate the run-time alignment optimization (loop peeling):

```
peel = x & 0x1f;
if (peel != 0) {
 peel = 32 - peel;
/* runtime peeling loop */
for (i = 0; i < peel; i++)
   x[i] = 1;
}
/* aligned access */
for (i = peel; i < 1024; i++)
  x[i] = 1;
```

Effective way to obtain aligned access patterns at the expense of a slight increase in code size and testing.

https:

Data structure alignment example

To instruct gcc compiler that data are aligned you can use:

```
void fill(char *x)
{
   float *y =__builtin_assume_aligned(x, 32);
   for (int i = 0; i < 1024; i++)
     y[i] = 1;
}
```

SIMD/fma

Floating-point arithmetic is not associative. The ordering of the operations will affect results due to round-off.

`-ffast-math` compiler option yield faster code for programs that *do not require the guarantees of IEEE or ISO specifications*. It enables `-funsafe-math-optimizations` flag.

```
__m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c)
```

Multiply packed single-precision (32-bit) floating-point elements in a and b, add the intermediate result to packed elements in c, and store the results in dst.

`_mm_fmadd_ps` generates one of the assembly intructions: `vfmadd231ps`, `vfmadd132ps`, `vfmadd213ps`.

Scalar product

Compiler explorer: `https://godbolt.org/z/XV8gG3`

```
float scalarproduct(float * array1, float * array2,
    size_t length) {
      float sum = 0.0f;
      for (size_t i = 0; i < length; ++i) {
              sum += array1[i] * array2[i];
      }
      return sum;
}
```

Try to add flags: `-O3`, `-ffast-math`, `-mfma`

Check code in the folder `scalar_product`.

Performance analysis

What is performance?

“**Performance** in general refers to the total effectiveness of a program.
What are the measures of the performance?

- execution time
- FLOPS (floating point operations per second)
- memory usage
- power consumption
- response time (time from request until result)
- throughput (number of operations per unit time)
- bandwidth (number of bits transmitted per unit time)

Measuring time

What is taking so much time?

Which section of my program should be made better?

How can you figure out why it is so slow?

What are possible ways to speed it up?

Speedup

When evaluating performance of programs we often talk about the "**speedup**" S that we get from some optimization we have done:

$$S = \frac{T_{original}}{T_{optimized}}$$

Reporting the speedup, defined in this way, is good because it is **well-defined** and clear what you mean.

Reporting the speedup is better than writing e.g. "performance increased by 20%" which would be less clear, readers may not understand exactly what that means.

CPU clock rate

CPU clock speed or rate is the speed at which a microprocessor executes instructions.

number of cycles per second - Hz

- Today's processors – 2-5 GHz (thousand million cycles per second)
- Record? (2013): IBM zEC12 with 5.5 GHz (using "overclocking" may give higher rates like 7 or 8 GHz but perhaps not working reliably)

Amount of work different CPUs can do in one cycle varies a lot.

Profiling

Profiling is a process of analyzing the behavior of the program (CPU performance, memory, I/O) **while the program is running**.

Gives a detailed description of execution times for each part of the program

Allows to identify:
- hot spots - big amount of executed instructions or place in the program where it uses a lot of CPU time.
- bottlenecks - parts of the program which uses processor resourses inefficiently.

Profiling is done on working code!

Profiling

Statistical Sampling and Code Instrumentation

Sampling means polling the status of the program at regular time intervals.

- Resulting profiles are statistical — not exact
- Usually has minimal interference with program's runtime
- The Unix/Linux `time` command works in this way

Instrumenting (tracing) means interposing profiling calls within your program's regular calls.

- More exact, but also more resource-intensive
- Reproducible results for e.g. number of calls to a function
- The `gprof` utility works in this way

That's all from me!