Lecture 7: More on performance

- Last lecture we focused on memory management and the cache hierarchy in modern CPUs
- A simply understanding of these things got us better performance
- We can go further!

BLAS/LAPACK again

www.netlib.org/blas/#_level_1

LEVEL 1

- SINGLE
  - SROTG - setup Givens rotation
  - SROTMG - setup modified Givens rotation
  - SROT - apply Givens rotation
  - SROTM - apply modified Givens rotation
  - SSWAP - swap x and y
  - SSCAL - x = a*x
  - SCOPY - copy x into y
  - SAXPY - y = a*x + y
  - SDOT - dot product
  - SDSDOT - dot product with extended precision accumulation
  - SNRM2 - Euclidean norm
  - SCNRM2- Euclidean norm
  - SASUM - sum of absolute values
  - ISAMAX - index of max abs value
- DOUBLE
  - DROTG - setup Givens rotation
  - DROTMG - setup modified Givens rotation
  - DROT - apply Givens rotation
  - DROTM - apply modified Givens rotation
  - DSWAP - swap x and y
  - DSCAL - x = a*x
  - DCOPY - copy x into y
  - DAXPY - y = a*x + y
  - DDOT - dot product
  - DSDOT - dot product with extended precision accumulation
  - DNRM2 - Euclidean norm
  - DZNRM2 - Euclidean norm

BLAS/LAPACK again

- We’re going to focus on implementing a simple level 1 BLAS routine
- daxpy – y = ax + y, where x,y are vectors, a is a scalar
- Last lecture we understood how important it was to know how your data is stored and how you access it
- C++ uses row-major storage by default to store 2D arrays

daxpy

◆ daxpy()

subroutine daxpy (	integer	N,
	double precision	DA,
	double precision, dimension(*)	DX,
	integer	INCX,
	double precision, dimension(*)	DY,
	integer	INCY
)		

DAXPY

Purpose:

```
    DAXPY constant times a vector plus a vector.
    uses unrolled loops for increments equal to one.
```

Parameters

[in]	**N**	N is INTEGER number of elements in input vector(s)
[in]	**DA**	DA is DOUBLE PRECISION On entry, DA specifies the scalar alpha.
[in]	**DX**	DX is DOUBLE PRECISION array, dimension (1 + (N - 1)*abs(INCX))
[in]	**INCX**	INCX is INTEGER storage spacing between elements of DX
[in,out]	**DY**	DY is DOUBLE PRECISION array, dimension (1 + (N - 1)*abs(INCY))
[in]	**INCY**	INCY is INTEGER storage spacing between elements of DY

Author

Time to write some code!

- Name your file main_daxpy.cpp
- We're going to implement our own version of a daxpy
- What are the incx and incy parameters?
- If you did the "ADVANCED" part of homework from Lecture 2, you may have seen a bit of this already

Assembly

- We now want to look at the assembly code generated
- Assembly are the instructions that our CPU accepts as commands
- It is a “low-level” language, C++ is a “high-level” language
- A compiler turns C++ into assembly, so the CPU can understand it
- We can have a look at the executable file our compiler produces to see the assembly it spits out

Follow the instructions at (or for gcc objdump -CdlS a.out > main_daxpy.asm):

https://docs.microsoft.com/en-us/visualstudio/debugger/how-to-use-the-disassembly-window?view=vs-2017

Assembly

These are lines 198-233 in main_daxpy_O0.asm

```
main_daxpy.cpp:13

   // Let's ignore the incx and incy for now
   /////#pragma loop(no_vector)
   for (int i = 0; i < n; i++)
  400a01:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
main_daxpy.cpp:13 (discriminator 3)
  400a08:       8b 45 fc                mov    -0x4(%rbp),%eax    // THESE ARE INSTRUCTIONS TO DO THE COMPARISON BETWEEN i AND n
  400a0b:       3b 45 ec                cmp    -0x14(%rbp),%eax
  400a0e:       7d 59                   jge    400a69 <daxpy(int, double, double*, int, double*, int)+0x83>
main_daxpy.cpp:15 (discriminator 2)
   {
      dy[i] += alpha * dx[i];
  400a10:       8b 45 fc                mov    -0x4(%rbp),%eax
  400a13:       48 98                   cltq
  400a15:       48 8d 14 c5 00 00 00    lea    0x0(,%rax,8),%rdx
  400a1c:       00
  400a1d:       48 8b 45 d0             mov    -0x30(%rbp),%rax
  400a21:       48 01 d0                add    %rdx,%rax        // HERE IS WHERE WE ARE DOING OUR ADDS
  400a24:       8b 55 fc                mov    -0x4(%rbp),%edx
  400a27:       48 63 d2                movslq %edx,%rdx
  400a2a:       48 8d 0c d5 00 00 00    lea    0x0(,%rdx,8),%rcx
  400a31:       00
  400a32:       48 8b 55 d0             mov    -0x30(%rbp),%rdx
  400a36:       48 01 ca                add    %rcx,%rdx
  400a39:       f2 0f 10 0a             movsd  (%rdx),%xmm1
  400a3d:       8b 55 fc                mov    -0x4(%rbp),%edx
  400a40:       48 63 d2                movslq %edx,%rdx
  400a43:       48 8d 0c d5 00 00 00    lea    0x0(,%rdx,8),%rcx
  400a4a:       00
  400a4b:       48 8b 55 d8             mov    -0x28(%rbp),%rdx
  400a4f:       48 01 ca                add    %rcx,%rdx
  400a52:       f2 0f 10 02             movsd  (%rdx),%xmm0
  400a56:       f2 0f 59 45 e0          mulsd  -0x20(%rbp),%xmm0        // AND OUR MULTIPLIES
  400a5b:       f2 0f 58 c1             addsd  %xmm1,%xmm0              // SOME MORE ADDS
```

Assembly

- You see “instructions: like mov, add, lea
- These are the CPU actually doing things, like incrementing the value of i in our for loop, moving things in and out of different areas of memory in the CPU
- Rather than write code in C++, you can go and write in assembly if you like
- Tedious, but it’s what we had to do before “high level” languages were invented

Assembly

- Now we’re going to turn on “optimisations” in our compiler, using the compiler flag “-O3”
- Note you may have to use the flag “-fno-tree-vectorise” to get assembly shown previously
- The compiler will try and do the best job to spit out the assembly that results in the fastest/smallest memory use code

Assembly

These are lines 322-354 in main_daxpy_O3.asm

```
main_daxpy.cpp:13
void daxpy(int n, double alpha, double *dx, int incx, double *dy, int incy)
{

   // Let's ignore the incx and incy for now
   /////#pragma loop(no_vector)
   for (int i = 0; i < n; i++)
  4009de:       31 d2                   xor    %edx,%edx
  4009e0:       31 c0                   xor    %eax,%eax
  4009e2:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
main_daxpy.cpp:15
   {
      dy[i] += alpha * dx[i];
  4009e8:       66 0f 10 04 06          movupd (%rsi,%rax,1),%xmm0
  4009ed:       83 c2 01                add    $0x1,%edx
  4009f0:       66 0f 59 c1             mulpd  %xmm1,%xmm0                    // THESE MULTIPLY INSTRUCTIONS LOOK DIFFERENT
  4009f4:       66 0f 58 04 01          addpd  (%rcx,%rax,1),%xmm0
  4009f9:       0f 29 04 01             movaps %xmm0,(%rcx,%rax,1)
  4009fd:       48 83 c0 10             add    $0x10,%rax
  400a01:       39 fa                   cmp    %edi,%edx
  400a03:       72 e3                   jb     4009e8 <main+0x188>
  400a05:       45 39 ca                cmp    %r9d,%r10d
  400a08:       43 8d 04 01             lea    (%r9,%r8,1),%eax
  400a0c:       74 1d                   je     400a2b <main+0x1cb>
  400a0e:       f2 0f 10 05 9a 03 00    movsd  0x39a(%rip),%xmm0        # 400db0 <_IO_stdin_used+0x40>
  400a15:       00
  400a16:       48 98                   cltq
  400a18:       49 8d 14 c4             lea    (%r12,%rax,8),%rdx
  400a1c:       f2 41 0f 59 44 c5 00    mulsd  0x0(%r13,%rax,8),%xmm0
  400a23:       f2 0f 58 02             addsd  (%rdx),%xmm0
  400a27:       f2 0f 11 02             movsd  %xmm0,(%rdx)
  400a2b:       49 8d ac 24 a0 0f 00    lea    0xfa0(%r12),%rbp
  400a32:       00
  400a33:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

Assembly

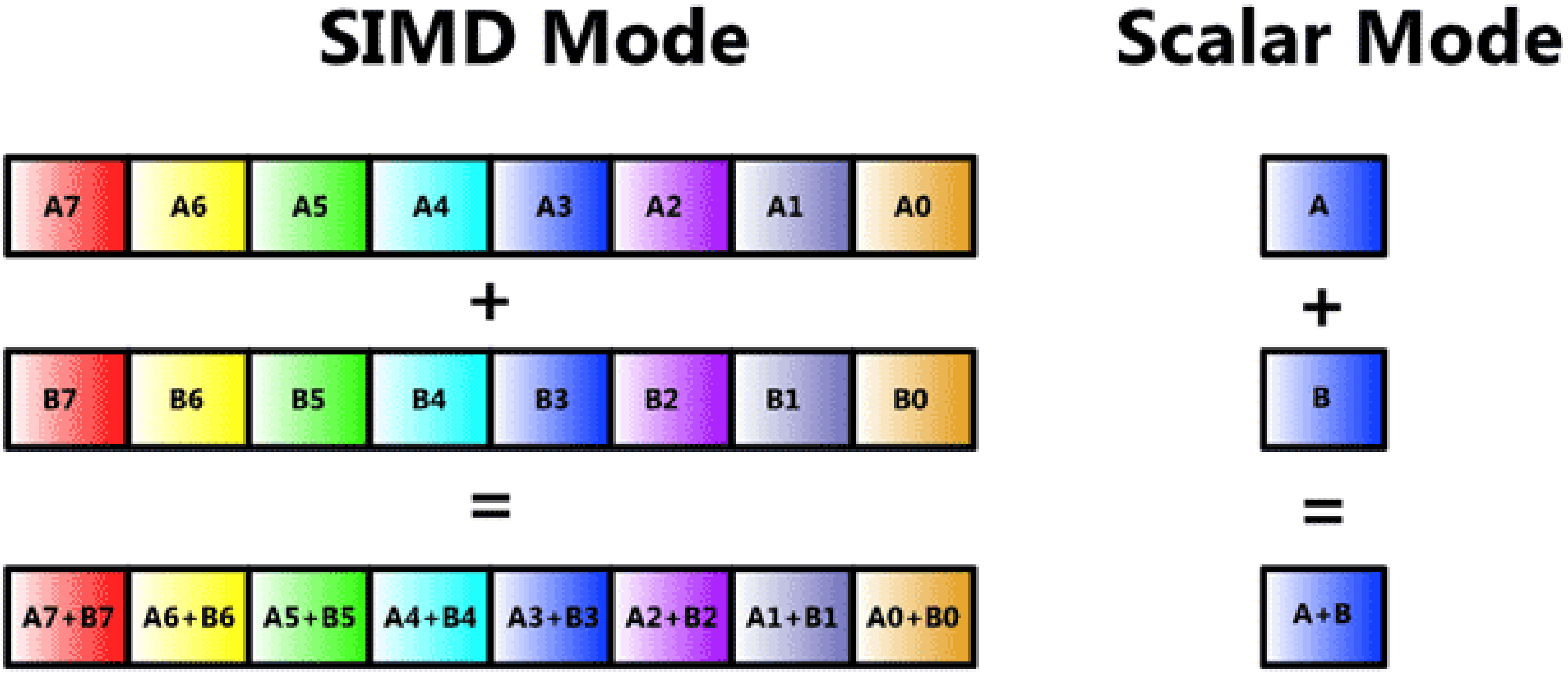
- Some of our instructions look different
- Particularly the appearance of “mulpd”

https://www.felixcloutier.com/x86/mulpd

Vectorisation, SSE, SSE2, AVX??

- In 1999, Intel introduced a new set of SIMD instructions to their processors
- Single instruction, multiple data
- Previous chips had a “floating point unit” in them that could do things like y = a * x, where x,y and a are doubles
- Intel introduced new hardware to do things like y[] = a * x[], so you could do multiplication across an entire vector
- They then created new instructions to control this hardware
- These instructions were called “SSE”, and if your chip has the hardware and instructions present, you could get big speedups by using these in your code

Vectorisation, SSE, SSE2, AVX??



Vectorisation, SSE, SSE2, AVX??

- If you write code in assembly, great time to learn some new instructions!
- If you write code in a “high-level” language like C++, it is the job of your compiler to turn your code into “fast” code
- So turning on optimisations like “-O3” tell the compiler to use things like these “vectorised” instructions
- Good news, means we don’t need to do much to benefit

Vectorisation, SSE, SSE2, AVX??

- Compilers are only so clever however
- Simple loops, not much pointer indirection, __restrict__ keyword, contiguous memory access!
- Compiler flags like "-free-vectorize -fopt-info-vec-missed" (gcc) will tell you when a compiler hasn't been able to vectorise a loop
- Then you can try rewriting your loops
- OR
- If you're doing things like daxpy, use BLAS/LAPACK functions whenever possible
- Whoever wrote your BLAS/LAPACK routines (hopefully) wrote vectorised code themselves (maybe in assembly)

Back to our daxpy

- What were those parameters incx and incy
- They are the increments to get to the next entry
- If they are not 1 your vector is not contiguous
- Why would anyone ever need this parameter?
- BLAS level 2/3 functions have similar parameters
- For example, if we look at dgemv
- lda is the size of the leading dimension
- We may only want to do a matrix-vector product on a subset of a matrix
- Given we’re doing non-contiguous memory access, we know this may will have a performance penalty!

dgemv

◆ dgemv()

```
subroutine dgemv ( character                            TRANS,
                   integer                              M,
                   integer                              N,
                   double precision                     ALPHA,
                   double precision, dimension(lda,*)   A,
                   integer                              LDA,
                   double precision, dimension(*)       X,
                   integer                              INCX,
                   double precision                     BETA,
                   double precision, dimension(*)       Y,
                   integer                              INCY
                 )
```

DGEMV

Purpose:

```
DGEMV  performs one of the matrix-vector operations

   y := alpha*A*x + beta*y,   or   y := alpha*A**T*x + beta*y,

where alpha and beta are scalars, x and y are vectors and A is an
m by n matrix.
```

Introduction to templating

- C++ has a nice way to write general code, called templating
- It lets you write “generic” code that depends on a type that is explicit at compile time
- For example, if you want to write a method that adds two values called add_values(int a, int b)
- If you want to change the type of a and b, you would have to write another method with add_values(double a, double b)
- Templating lets you write generic code, like add_values(Type a, Type b)
- Then when you call add_values(a,b) it works out what type a and b are, and the compiler builds a type specific method

Templating

- You’ve already used a templated library
- The stl (standard TEMPLATE library)
- Specifically the stl containers
- When you call vector<int>, or vector<double>, you are using templates
- We can even write out own templated methods/classes

Time to write some code!

- Get your Matrix.h and Matrix.cpp from last lecture
- There are fresh copies on the github if you need
- Write a main class test_main_template.cpp
- We are going to convert our Matrix class to be templated
- That way we don’t have to write any more code if you wanted a Matrix class that held integers, instead of doubles

Homework

- Try and see if the use of the incx and incy parameters changes your assembly code in our daxpy
- Understand what “registers” on the CPU and where you can see them being used in the assembly code
- Try doing some timing tests between the vectorised and non-vectorised daxpy
- ADVANCED – Install and link to a BLAS library (e.g., ATLAS, OpenBLAS)
- SUPER ADVANCED – Do timing tests for BLAS vs our dgemm as the matrix size increases
- SUPER ADVANCED – Read up on the hardware differences between MMX/SSE/SSE2/AVX.. etc