Recitation 4

Announcements

- Problems set 4 is out, it consists of two parts:
  - Theory, deadline 12.10, graded.
  - Code, deadline 19.10, pass/fail.
- Problem set 2 will be graded soon.

Performance competition

- Tune/optimize serial matrix multiplication.
- The one with the fastest solution will get an AMAZING price.
- A certain speedup is required to pass the assignment.
- You can also compare your results with ATLAS, a auto-tuned BLAS implementation.

GEMM

- BLAS (Basic Linear Algebra Subprograms) is a much used library.
- It contains the function `gemm()`, general matrix multiply.
- Specifically, it computes:

$$C = \alpha AB + \beta C$$

  where A is a $m \times k$ matrix, B is a $k \times n$ matrix, C is a $m \times n$ matrix and α and β are scalars.
- (If we set α to 1, and β to 0, this becomes plain matrix multiplication)

HEMM

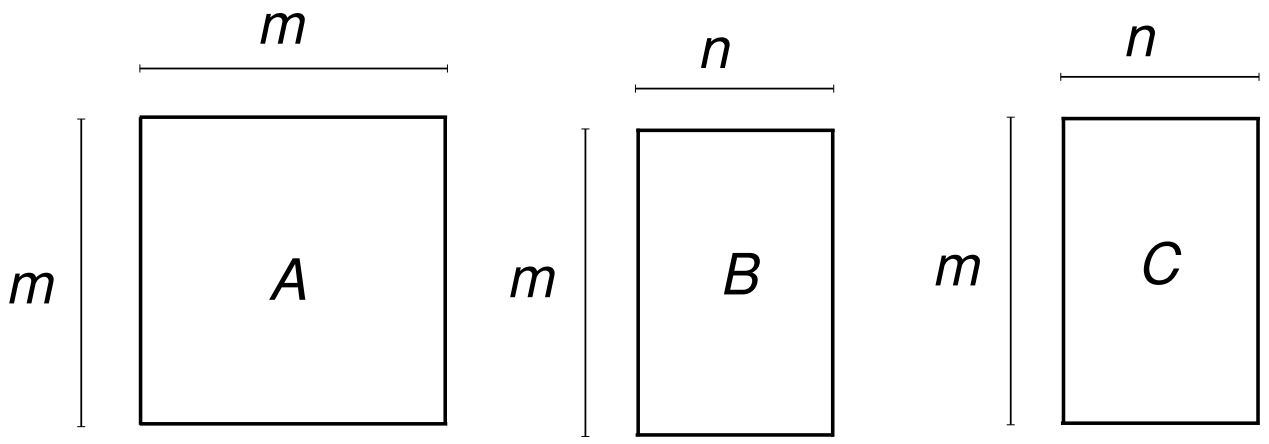
- We will implement a variation, where A is *Hermitian*
- A Hermitian matrix is a complex square matrix equal to its own conjugate transpose.
- That is, the element in the i-th row and j-th column is equal to the complex conjugate of the element in the j-th row and i-th column.
- The complex conjugate of a complex number is the number with the same real part, but where the imaginary part has oposite sign.
- E.g. the complex conjugate of 2 + 3i is 2 - 3i.

Example

$$\begin{bmatrix} 3 & 2+4i & 8-9i & -2+2i \\ 2-4i & 7 & -5-4i & 1+i \\ 8+9i & -5+4i & 8 & -3+2i \\ -2-2i & 1-i & -3-2i & 1 \end{bmatrix}$$

- For example, row 3, column 1: 8 + 9i is the complex conjugate of row 1, column 3: 8 - 9i
- The elements on the diagonal are real, they are their own complex conjugates.

Matrices



Only A is Hermitian (and therefore square)

Naive implementation

gemm

```
for(int x = 0; x < n; x++){
  for(int y = 0; y < m; y++){
    C[y*n + x] *= beta;
    for(int z = 0; z < m; z++){
      C[y*n + x] += alpha*A[y*m+z]*B[z*n + x];
    }
  }
}
```

This is just a plain matrix multiply, it does not take advantage of the fact that A is Hermitian.

Complex numbers

- As mentioned, we'll use complex numbers.
- C99 includes support for complex numbers, addition, multiplication etc works as expected.

```
#include <complex.h>
...
complex double a = 5 + 3*I;
complex double b = 4 + 1*I;

complex double c = a * b;

printf("%f, %f\n", creal(c), cimag(c));
```

Complex numbers

C99s complex numbers are implemented with structs. So we can do stuff like this:

```
typedef struct{
    double real; double imag;
} my_complex;

int main(){
    complex double a = 4 + 3*I;

    double* b = (double*)&a;
    my_complex* c = (my_complex*)&a;

    //Both will print 4.0,3.0
    printf("%f,%f\n", b[0], b[1]);
    printf("%f,%f\n", c->real, c->imag);
}
```

SIMD instructions

- The *Streaming SIMD Extensions* (SSE) were introduced by Intel for the Pentium III back in ’99.
- Several updates (SSE2, SSE3, SSSE3, AVX, AVX2 etc) have been added since.
- The course servers support AVX2, if you have an older CPU, it might not support the latest versions.
- Extend x86 with intructions that use special 128 (SSE) or 256 (AVX) bit registers.
- One such register can for instance hold 4 or 8 floats, or 2 or 4 doubles (for 128 and 256 bits, respectively).
- Using SSE/AVX instructions, we can therefore multiply 4 or 8 floats in a single instruction.
- C doesn’t support SSE/AVX directly. We must use complier intrinsics. (Or inline assembly).

Intrinsics

A intrinsic function is a function that the compiler translates directly. This code performes elementwise multiplication of two double arrays, using SSE.

```
for(int i = 0; i < 1024; i += 4){
    __m128 x = _mm_load_ps(&(a[i]));
    __m128 y = _mm_load_ps(&(b[i]));

    __m128 z = _mm_mul_ps(x,y);

    _mm_store_ps(&(c[i]), z);
}
```

Complex multiplication with intrinsics

	x	a.r	a.i	b.r	b.i
	y	c.r	c.i	d.r	d.i
`t = MOVSLDUP(x);`	t	a.r	a.r	b.r	b.r
`t2 = t * y;`	t2	a.r * c.r	a.r*c.i	b.r*d.r	b.r*d.i
`y = SHUFPS(y, y, 0xb1);`	y	c.i	c.r	d.i	d.r
`t = MOVSHDUP(x);`	t	a.i	a.i	b.i	b.i
`t = t * y;`	t	a.i*c.i	a.i*c.r	b.i*d.i	b.i*d.r
`x = ADDSUBPS(t2, t);`	t2	a.r*c.r-a.i*c.i	a.r*c.i+a.i*c.r	b.r*d.r-b.i*d.i	b.r*d.i+b.i*d.r

(Here, short versions of the instrinsic names are used, again using 128 bit SSE)

Intrinsics

- Intel has a good guide to the available intrinsics at `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`
- To use intrinsics, you need to include <x86intrin.h> and use the -mavx2 (or e.g. -msse3 for earlier versions) for gcc.
- You also need aligned memory, this can be allocated with `posix_memalign()`.
- These details might vary with other operating systems/compilers.

Practicalities

- The matrix multiplication code consists of 4 files: *chemm.c* which contains the main function, and *naive.c*, *fast.c* and *atlas.c*, each of wich contains a implementation of the `chemm()` function.
- Currently, the implementations in *naive.c* and *fast.c* are the same, you should improve the implementation in *fast.c*.

Compilation

- The provided makefile will build three different programs *chemm_naive*, *chemm_fast* and *chemm_atlas*, one for each implementation.
- In addition to the necessary linking flags, the only flags used are `-O3 -mavx2`. Other flags might be useful.

Problem size

- You may assume that *m*, *n* are powers of two (2,4,8,16...).
- These sizes are specified as command line arguments.
- When testing the performance of your code, you should use matrices big enough to make your code run some seconds.
- If you're doing cache related optimizatins, keep in mind that these will only have an effect if the matrices doesn't fit in the cache.
- The content of the matrices, as well as α and β are random (complex) numbers.

Correctness

- The output will be compared with the output of atlas to check for correctness.
- Small errors are expected, due to roundoff/floating point problems, your code shouldn’t be any worse than the naive implementation.
- There are also functions to print the matrices for debugging.

Timing

- Keep in mind that the performance of your program will depend upon the other activity on the computer you're using. The course servers will probably be busy on the night of the deadline, which might influence your results. Run the program a few times to make sure you get stable timings.
- For the performance competition, we will run the programs with a few different problem sizes on `tdt4200s1`.
- For the very best performance, you need to tune the porgram to the specific CPU used, `tdt4200s1` has a Xeon E5-2640

Hints

- Take advantge of the fact that the matrix is Hermitian.
- Use vector/SIMD instructions.
- Try cache blocking.
- Loop unrolling, reordering, tiling.