Recitation 3

Announcements

- Problems set 3 is out, deadline 3.10.
- Problem set 2 will be graded soon.

PS 3

- Serial optimization.
- Some theory.
- Some code reading/analysis.
- Finding bugs with Valgrind (cf. Rune's lecture)
- Performance competition.

Valgrind

- The Valgrind part of the assignment is based on Runes lecture on Monday.
- In all the exercises, what we're asking for is a list of bugs/errors/issues/comments.
- The exercise texts also contains a lot of instructions and questions. These are only intended to help you find bugs/problems, and should not be answered directly, the only thing we want is the list.
- Short one-line answers are acceptable, eg:
  Line 41: "No newline at end of file". And ends with a comment too!
- (See http://stackoverflow.com/questions/72271/no-newline-at-end-of-file-compiler-warning to read a why it is bad.)

KCachegrind/QCachegrind

- For one of the problems, Kcahcegrind is great tool to find problmes.
- A Windows version, Qcachegrind, exists.
- According to the internet, it is possible to make it work on Macs too, but I have not tested this myself.
- If you find the instalation too difficult, or don't want to install random third party programs, you can use the text based valgrind tools to find problems instead.
- You'll typically find different things with these two approaches.

Performance competition

- Tune/optimize 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.

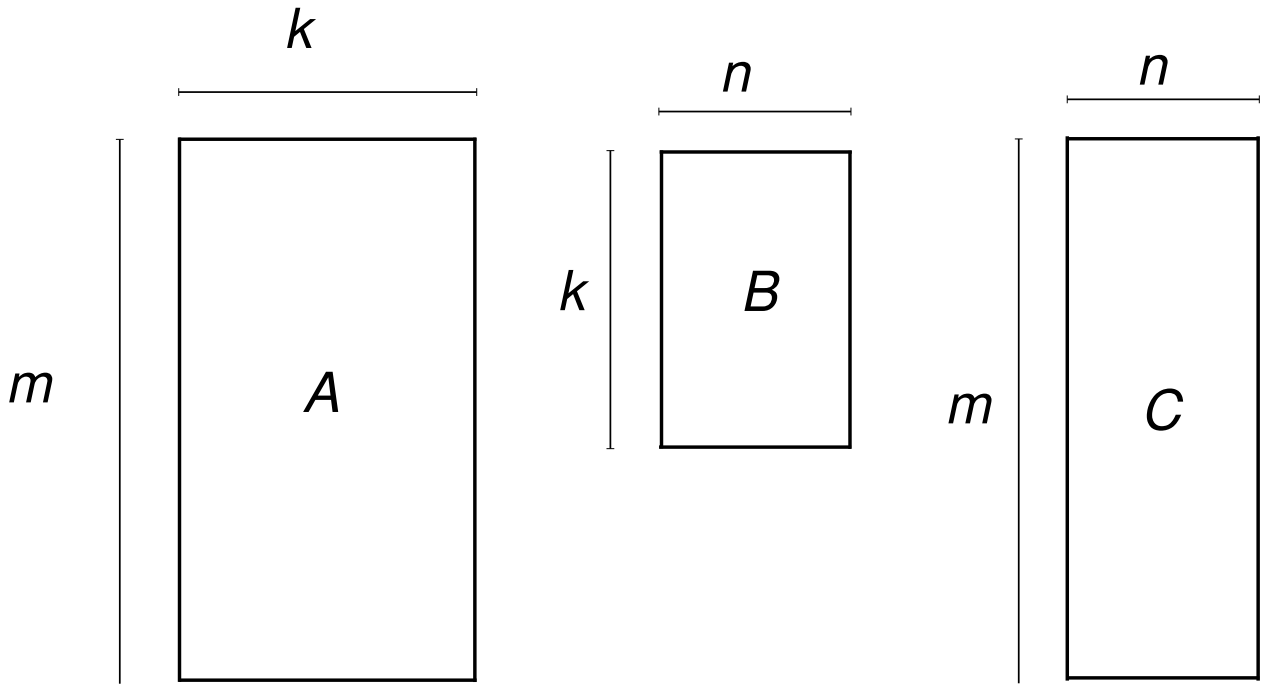
GEMM

- 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)
- We'll implement this for complex numbers.

Matrices



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 < k; z++){
      C[y*n + x] += alpha*A[y*k+z]*B[z*n + x];
    }
  }
}
```

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, etc) have been added since.
- Extend x86 with intructions that use special 128 bit registers.
- One such register can for instance hold four floats, or two doubles.
- Using SSE instructions, we can therefore multiply four floats in a single instruction.
- C doesn’t support SSE 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.

```
for(int i = 0; i < 1024; i+=2){
  x = _mm_loadu_pd(&a[i]);
  y = _mm_loadu_pd(&b[i]);

  // z = x * y;
  z = _mm_mul_pd(x,y);

  _mm_storeu_pd(&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)

Practicalities

- You'll get two archives, one with code for the Valgrind part, the other with code for the matrix multiplication.
- The matrix multiplication code consists of 4 files: *gemm.c* which contains the main function, and *naive.c*, *fast.c* and *atlas.c*, each of wich contains a implementation of the `gemm()` 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 *gemm_naive*, *gemm_fast* and *gemm_atlas*, one for each implementation.
- In addition to the necessary linking flags, the only flag used is `-O3`. Other flags might be useful.

Problem size

- You may assume that *m*, *n* and *k* 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 for some seconds.
- 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.