1. Mandatory Assignment

Mark Jervelund
(Mjerv15)

Parallel Computing (DM818)

UNIVERSITY OF
SOUTHERN DENMARK

IMADA

October 19, 2017 at 10

Contents

Project description

To describe the services an operating system provides to users, processes, and other systems
To discuss the various ways of structuring an operating system
To explain how operating systems are installed and customized and how they boot

Introduction

This report contains documentation for the first mandatory assignment for DM818 Parallel Computing, in the first assignment we were tasked with designing a high performance singled threaded matrix matrix multiplication program.

Work load

For the following projected i ended up working alone as i didn't really know anyone in the course as I'm a bachelor student, the people i knew from other courses were already in groups.

Issues

There is currently a bug with the freeing the memory allocation in the program, this means that the program will work one iteration. I've tried bugging with gdb and Valgrind, but i haven't been able to figure out what the bug is yet. but i assume it's a overflow, or our of place memory happening in the B Array, see the appendix for Valgrind and a link to a post on stack overflow which could be the same as the issue I'm experiencing.

Design

Advice from lecturer

For the designing the algorithm we were given 2 examples of how its normally naively implemented, a naive 3 for loop implementation, and a naive blocked version.

Furthermore we were pointed heavy towards the Goto paper on high performance matrix multiplication, which describes how you should design the blocking for the best possible performance.

and as a third pointer we were given a lecture by Jacob who explained how he implemented it, and what we should do to get the highest performance,

Own design choices

I made a design choice follow Jacobs implementation. this meant writing functions that pack the larger matrix into smaller sub-matrices that are then used for the matrix matrix multiplication.
For figuring out the block size i calculated the largest n by n matrix possible,

$$sqr((256 * 1024byte)/64bit) = 181.019335984 \tag{1}$$

as 181 isn't a good size when handing matrix blocking, i rounded down to nearest 2^n which is 128.

$$128 * 128 * 64bit = 128Kbytes \tag{2}$$

I could also have attempted to implement a 256 by 128 matrix, but this would cause the level2 cache to always be filled to the brim and would most likely be dumped to memory which would "increase calculation time by a factor of 100s or 1000s" so i decided to not even attempt this.
I only decided on only implementing intrinsic for a core of 4x4, and just using a naive for loop implementation for the smaller blocks.

Implementation

Square-dgemm

The initial function square_dgemm gets the 3 matrices and M in as the 4 arguments. M is directly saved as a global variable.
3 sub matrices are then reserved, Ablock, Bblock, and C, block. Ablock and Cblock and allocated in a way that is aligned in memory by 32-bytes. this is done so the intrinsic instructions will run faster.

The first for loop blocks B into blocks of of size KC * M(128 * size of matrix). The second for loop blocks A into blocks of 128 by 128 so we keep as much of A in cache as possible.
Prepare block is then called with the blocks we want to multiply.

```
void square_dgemm(int M, double *A, double *B, double *C) {
    lda = M;
    Ablock = (double *) _mm_malloc(MC * KC * sizeof(double), 32); //128*128
    Bblock = (double *) malloc(lda * KC * sizeof(double)); // M * 128
    Cblock = (double *) _mm_malloc(MC * NR * sizeof(double), 32); //128*4

    for (unsigned int k = 0; k < lda; k += KC) {
        packBBlock(B + k, std::min(KC, lda - k));
        for (unsigned int i = 0; i < lda; i += MC) {
            packAblock(A + i + k * lda, std::min(MC, lda - i), std::min(KC, lda - k));
            Prepare_block(C + i, std::min(MC, lda - i), lda, std::min(KC, lda - k));
        }
    }
    _mm_free(Ablock);
    free(Bblock);
    _mm_free(Cblock);
}
```

Blocking functions

Prepare-block

testing

Comparison

Conclusion

appendix

Goto paper

https://dl.acm.org/citation.cfm?id=1356053

Code Protect

https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX

Valgrind result and stackoverflow post

```
https://stackoverflow.com/questions/2334352/why-do-i-get-a-sigabrt-here

==13219== Your program just tried to execute an instruction that Valgrind
==13219== did not recognise.  There are two possible reasons for this.
==13219== 1. Your program has a bug and erroneously jumped to a non-code
==13219==    location.  If you are running Memcheck and you just saw a
==13219==    warning about a bad jump, it's probably your program's fault.
==13219== 2. The instruction is legitimate but Valgrind doesn't handle it,
==13219==    i.e. it's Valgrind's fault.  If you think this is the case or
==13219==    you are not sure, please let us know and we'll try to fix it.
==13219== Either way, Valgrind will now raise a SIGILL signal which will
==13219== probably kill your program.
```

other

```
 https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX_512&cats=Elemen

https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX

_mm<bit_width>_<name>_<data_type>

The parts of this format are given as follows:

<bit_width> identifies the size of the vector returned by the function. For 128-bit vect

<name> describes the operation performed by the intrinsic
<data_type> identifies the data type of the function's primary arguments


Data Type       Description
__m128i 128-bit vector containing 8 integers
__m128  128-bit vector containing 4 floats
__m128d 128-bit vector containing 2 doubles
__m256i 256-bit vector containing 16 integers
__m256  256-bit vector containing 8 floats
__m256d 256-bit vector containing 4 doubles
__m512i 512-bit vector containing 32 integers
__m512  512-bit vector containing 16 floats
__m512d 512-bit vector containing 8 doubles



ps - vectors contain floats (ps stands for packed single-precision)
pd - vectors contain doubles (pd stands for packed double-precision)
epi8/epi16/epi32/epi64 - vectors contain 8-bit/16-bit/32-bit/64-bit signed integers
epu8/epu16/epu32/epu64 - vectors contain 8-bit/16-bit/32-bit/64-bit unsigned integers
```

```
si128/si256 - unspecified 128-bit vector or 256-bit vector
m128/m128i/m128d/m256/m256i/m256d - identifies input vector types when they're different
```

notes from lecture with Jakob

Our cache use, calculate it to be = to 256 or maybe a bit less. $kcandmcshouldbelarge.n_cislessimportant.$
kc = 128 MC = 128, $8 * (m_c * k_c + m_c * n_c + N_c + k_c)$
$avx_256registersarecalledymm0to15$ $avx_512registersarecalledzmm0to31$
Make function for each size of slices.. 128 = 4.4 256 = 8.4 512 = 8.8