1. Mandatory Assignment

Aligned SSE: yes

Unaligned SSE: no

Mark Jervelund
(Mjerv15)

Parallel Computing (DM818)

UNIVERSITY OF SOUTHERN DENMARK

IMADA

October 20, 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 and the remainder that i talked to were phds as were not allowed to form groups.

Issues

There is currently a bug with the freeing the memory allocation, this means that the program will work one iteration. I've tried debugging with gdb and Valgrind, but i haven't been able to figure out where the bug is yet. I assume it's an overflow, or an out of place memory happening in the B Array.

Design

Advice from lecturer

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

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

and as a third helping hand 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 the design choice to 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.

Block size and Cache

For figuring out the block size, the size of the largest possible matrix that can fit in level 2 cache was calculated.

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

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

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

An equation to solve this was also shown doing the lecture. The this is used the 128 by 128 for the main matrix, and 4 by 128 for the remaining.

$$8 * (m_c * k_c + m_c * n_c + N_c * k_c) = 8 * (128 * 128 + 128 * 4 + 4 * 128) = 139264bytes\ 136Kbytes \tag{3}$$

After doing the calculations we can see that the program will need at least 136 KByte of cache to store the data needed. level2 cache in most modern cpu's is 256 KB, so this will suffice. A reason for not pushing it a using maybe a 256 by 128 block will mean that all 256 KB of the level2 cache would be used by the main matrix, this could cause some severed trotting issues due to it being pushed to level3 cache or main memory..

Intrinsic

I only decided on only implementing intrinsic for a core of 4x4, and just using a naive for loop implementation for the smaller blocks, as programming all the possible different blocks alone would take too much time.

This project was designed using avx2 instruction set, this set contains 6 data types

1. __m128 contains 4 floats
2. __m128d contains 2 doubles
3. __m128i contains 8 integers
4. __m256 contains 8 floats
5. __m256d contains 4 doubles
6. __m256i contains 16 integers

A generalization for the naming scheme of data types is __m<size><type> where size is 128, 256 or 512 when you count the newest avx512 in.

The naming convention on the functions for the instructions follow the same scheme.
<size><name>_<type> Its almost the same except a few more data types can be used here.
The data type the project will use is pd which is packed doubles.
However a large array of data types are supplied to use with the intrinsic instructions.

1. ps packed floats
2. pd packed doubles
3. epi8/16/32/64 vectors containing signed integers of size 8 to 84
4. epu8/16/32/64 vectors containing unsigned integers of size 8 to 84
5. si128/si256/si512 unspecified vector of size 128, 256, or 512
6. m<size><type> when using input vectors that are different than return type.

There are too many different functions to list so I'll only list the ones i have used in this project

1. mul - Multiplication
2. add - Adds two vectors
3. store - Stores a vector from registers to memory
4. setzero - sets all entries in a vector to 0.0
5. load - loads data from memory into registers.
6. set1 - sets all values in a vector to specified value.

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 when storing or loading information.

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

The program has 2 block packing functions. and 1 unpacking, the 2 packing functions are quite close in usecase and what they do.
PackA packs up to 128x128 of matrix A into Ablock, and Packb packs up to 128*lda into BlockB.
The last packing function is for unblocking is for unpacking Cblock into C

do-block

The do block functions contains the 2 inner for loops that split A into smaller blocks, and the logic that decides if its uses the high performance multiplication function, or the naive dynamic implementation.

```
void Do_block(double *C, unsigned int M, unsigned int N, unsigned int K) {
    double *B = Bblock;
    for (unsigned int n = 0; n < N; n += NR) {
        for (unsigned int m = 0; m < M; m += MR) {
            unsigned int Max_M = std::min(NR, M - m);
            unsigned int Max_N = std::min(MR, N - n);
            if (Max_M == MR && Max_N == NR) {
                core_4_4(Ablock + m * K, B, Cblock + m, K);
            } else {
                core_dyn(Ablock + m, B, Cblock +m, M, N, K);
            }
        }
        B += NR * K;
        unpackCBlock(C + n * lda, M, std::min(NR, N - n));
    }
}
```

Core-dyn

This is simply a stard matrix matrix multiplication function using 3 loops, and no intrinsics.

Core-4x4

The core 4x4 function is the high performance implementation of matrix matrix multiplication. Its performs 32 double precision floating point operations per cycle using fewer operations.
The first optimization that the intrinsic functions does for us is that we load 4 doubles into registers from memory at the time.
after this we load a single double from B into 4 positions in a vector again only using one instruction. this pattern keeps going with all out operations. we can load blocks into registers a lot faster, and we can compute 4 numbers at the time using this function.

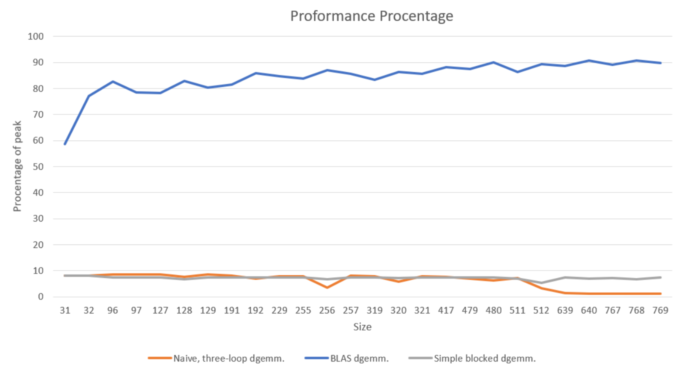
```
    __m256d c0 = _mm256_setzero_pd();
    __m256d a1 = _mm256_load_pd(A + k * MR);
    __m256d b00 = _mm256_set1_pd(B[k]);
    c0 = _mm256_add_pd(c0, _mm256_mul_pd(b00, a1));
    _mm256_store_pd(C, c0);
```

Testing

For testing the program i was unable to get the benchmark to work due to a memory allocation/free error in the code. i had it run once but there ware some problems in other parts of the code, but the performance was around 5000 mflops. but I'm currently unsure if that result could be trusted so I'm only mentioning it here.
The only testing I've done is to check if it returns the correct values, this was done using a testing program i made by removing some parts of the Benchmark program and changing it so it compared to the other implementations we were given as well as printing the output to the terminal for manual verification.

Comparison

The comparison of the 3 implementations we were given.



Conclusion

The Matrix matrix multiplication assignment proved to be a bit harder than anticipated. I thought I had it working, but found out with 2 days left that there was an error i was unable to find. I did however get to learn a lot of new things, and i got to get better at using the tools for debugging and finding out where the issue in the code is. furthermore I've learned how use intrinsic and how it can boost a programs run time by a large factor.

appendix

Goto paper

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

Error, Valgrind result and stack overflow post

```
*** Error in `./benchmark-dgemm': munmap_chunk(): invalid pointer: 0x0000000000eca9b0 ***
benchmark-dgemm: malloc.c:2405: sysmalloc: Assertion `(old_top == initial_top (av) && old_size
    == 0) || ((unsigned long) (old_size) >= MINSIZE && prev_inuse (old_top) && ((unsigned long)
     old_end & (pagesize - 1)) == 0)' failed.


==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.

https://stackoverflow.com/questions/2987207/why-do-i-get-a-c-malloc-assertion-failure
```

Code

```
//
// Created by jervelund on 9/21/17.
//


#include <immintrin.h>
#include <stdio.h>
#include <algorithm>
#include <malloc.h>
#include <iostream>

/*
  In case you're wondering, dgemm stands for Double-precision, GEneral Matrix-Matrix
      multiplication.
*/

const char *dgemm_desc = "mjerv15 blocked dgemm.";

//#define min(a,b) (((a)<(b))?(a):(b))

unsigned int NR = 4;
unsigned int MR = 4;

unsigned int KC = 128;
unsigned int MC = 128;
unsigned int lda; //size lda*lda of matrix

double *Ablock;
double *Bblock;
double *Cblock;

```

```
//              From Lecture by Jacob
//                                                                    N
//                                +----------------------------+
//                                |                         |
//                                |                         |
//                                |                         |
//                                |                         |
//                                |                         |
//                              K |            B            |
//                                |                         |
//                                |                         |
//                                |                         |
//                                |                         |
//                                |                         |
//                                |                         |
//                                                +----------------------------+
//                      K
//  +----------------------------+ +----------------------------+
//  |                         |   |                         |
//  |                         |   |                         |
//  |                         |   |                         |
//  |                         |   |                         |
//  |                         |   |                         |
//  |           A             |   |                         |
// M|                         |   |            C            |
//  |                         |   |                         |
//  |                         |   |                         |
//  |                         |   |                         |
//  |                         |   |                         |
//  |                         |   |                         |
//  +----------------------------+ +----------------------------+

//debugging
void printMatrix2(double *matrix, int MatrixSize) {
    std::cout << ("Starting Matrix") << '\n';
    for (int i = 0; i < MatrixSize * MatrixSize; i++) {
        std::cout << matrix[i] << ' ';
        if ((i + 1) % MatrixSize == 0) {
            printf("\n");
        }
    }
    std::cout << ("Ending matrix") << '\n';
}

void printdouble(char *text, double *X, int arraySize) {
    std::cout << text << '\n';
    for (int i = 0; i < arraySize; i++) {
        double a = X[i];

        std::cout << a << ' ';
    }
    std::cout << '\n';
}


void packAblock(double *A, unsigned int M, unsigned int K) {
    unsigned int a = 0;
    for (unsigned int m = 0; m < M; m += MR) {
        unsigned int MMax = std::min(MR, M - m);
        for (unsigned int k = 0; k < K; k++) {
            for (unsigned int i = 0; i < MMax; i++) {
                Ablock[a++] = A[m + i + k * lda];
            }
```

```
        }
    }
}

void packBBlock(double *B, unsigned int K) {
    unsigned int b = 0;
    for (unsigned int n = 0; n < lda; n++) {
        for (unsigned int k = 0; k < K; k++) {
            Bblock[b++] = B[k + n * lda];
        }
    }
}


//TODO currently broken, currently fixed
void unpackCBlock(double *C, unsigned int M, unsigned int N) {
//   printMatrix2(Cblock,4 );
    for (unsigned int n = 0; n < NR; n++) {
        for (unsigned int i = 0; i < M; i++) {
//           int lookingat = i + n * MC; ///Debugging value used when debugging with gdb
//           int storeingat = i + n * lda; ////Debugging value used when debugging with gdb
//           double x = Cblock[lookingat]; ////Debugging value used when debugging with gdb
            C[i + n * lda] = Cblock[i + n * MC];
        }
    }
}


//This is pretty what much Jacob did, except i wrote it with AVX2
//He said we should let the compiler take care of how to handle this. eg.
// __m256d c01x0-3 could be written as __m256d c23 = _mm_set_pd(0.0,0.0,0.0,0.0); and same goes
     for most of this function.
void core_4_4(double *A, double *B, double *C, unsigned int K) {
//    std::cout << "K is = " << K << '\n';
    __m256d c0 = _mm256_setzero_pd();
    __m256d c1 = _mm256_setzero_pd();
    __m256d c2 = _mm256_setzero_pd();
    __m256d c3 = _mm256_setzero_pd();

    for (unsigned int k = 0; k < K - 1; k += 2) {
//       printdouble(A+k*MR,4);
//       printdouble(A+(2+k)*MR,4);

        __m256d a1 = _mm256_load_pd(A + k * MR);
//       printdouble("A0123k0",A + (k) * MR, 4);

        __m256d a2 = _mm256_load_pd(A + (k + 1) * MR);

//       printdouble("A0123k1",A+ (k+1) * MR, 4);

        __m256d b00 = _mm256_set1_pd(B[k]);
        __m256d b01 = _mm256_set1_pd(B[k + K]);
        __m256d b02 = _mm256_set1_pd(B[k + K * 2]);
        __m256d b03 = _mm256_set1_pd(B[k + K * 3]);

        __m256d b10 = _mm256_set1_pd(B[k + 1]);
        __m256d b11 = _mm256_set1_pd(B[k + K + 1]);
        __m256d b12 = _mm256_set1_pd(B[k + K * 2 + 1]);
        __m256d b13 = _mm256_set1_pd(B[k + K * 3 + 1]);

        //do left
        c0 = _mm256_add_pd(c0, _mm256_mul_pd(b00, a1));
```

```
        c1 = _mm256_add_pd(c1, _mm256_mul_pd(b01, a1));
        c2 = _mm256_add_pd(c2, _mm256_mul_pd(b02, a1));
        c3 = _mm256_add_pd(c3, _mm256_mul_pd(b03, a1));

        c0 = _mm256_add_pd(c0, _mm256_mul_pd(b10, a2));
        c1 = _mm256_add_pd(c1, _mm256_mul_pd(b11, a2));
        c2 = _mm256_add_pd(c2, _mm256_mul_pd(b12, a2));
        c3 = _mm256_add_pd(c3, _mm256_mul_pd(b13, a2));

    }
    //edge case.
    if (K % 2 == 1) {
        unsigned int k = K - 1;
        __m256d a0 = _mm256_load_pd(A + k * MR);

        __m256d b00 = _mm256_set1_pd(B[k]);
        __m256d b01 = _mm256_set1_pd(B[k + K]);
        __m256d b02 = _mm256_set1_pd(B[k + K * 2]);
        __m256d b03 = _mm256_set1_pd(B[k + K * 3]);

        c0 = _mm256_add_pd(c0, _mm256_mul_pd(b00, a0));
        c1 = _mm256_add_pd(c1, _mm256_mul_pd(b01, a0));
        c2 = _mm256_add_pd(c2, _mm256_mul_pd(b02, a0));
        c3 = _mm256_add_pd(c3, _mm256_mul_pd(b03, a0));
    }

    //TODO something is broken here. its been fixed
    ////Debugging value used when debugging with gdb
//    double *tmp = (double *) _mm_malloc(NR * sizeof(double), 32);
//
//    _mm256_store_pd(tmp, c0123x0);
//
//    _mm256_store_pd(tmp, c0123x1);
//
//    _mm256_store_pd(tmp, c0123x2);
//
//    _mm256_store_pd(tmp, c0123x3);

//    _mm_free(tmp);

    _mm256_store_pd(C, c0);
    _mm256_store_pd(C + MC, c1);
    _mm256_store_pd(C + 2 * MC, c2);
    _mm256_store_pd(C + 3 * MC, c3);


}

//works.
void core_dyn(double *A, double *B, double *C, unsigned int M, unsigned int N, unsigned int K)
 {
    for (unsigned int j = 0; j < N; j++) {
        for (unsigned int i = 0; i < M; i++) {
            double cij = C[j * lda + i];
            for (int k = 0; k < K; ++k) {
                cij += A[k * lda + i] * B[j * lda + k];
            }
            C[j * MC + i] += cij;
        }
    }
}

void Do_block(double *C, unsigned int M, unsigned int N, unsigned int K) {
```

```
    double *B = Bblock;
    for (unsigned int n = 0; n < N; n += NR) {
        for (unsigned int m = 0; m < M; m += MR) {
            unsigned int Max_M = std::min(NR, M - m);
            unsigned int Max_N = std::min(MR, N - n);
            if (Max_M == MR && Max_N == NR) {
                core_4_4(Ablock + m * K, B, Cblock + m, K);
            } else {
                core_dyn(Ablock + m, B, Cblock +m, M, N, K);
            }
        }
        B += NR * K;
        unpackCBlock(C + n * lda, M, std::min(NR, n - n));
    }
}

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));
            Do_block(C + i, std::min(MC, lda - i), lda, std::min(KC, lda - k));
        }
    }
    _mm_free(Ablock);
    free(Bblock);
    _mm_free(Cblock);
}
```