Matrix-Martrix Multiply: Tiled and Data Parallel Approaches

Anthony Troy 14212116

April 26, 2015

CONTENTS

Disclaimer: Submitted to Dublin City University, School of Computing for module CA670: Concurrent Programming, 2015. I hereby certify that the work presented and the material contained herein is my own except where explicitly stated references to other material are made.

1 INTRODUCTION

As identified in our previous report, dense linear algebraic algorithms rely on the consideration of spatial and temporal locality. Our candidate algorithm, dense Matrix-Matrix multiplication (MMM), requires "abundant parallel computation", "frequent memory access", and features "O(n) data reuse". Thus it is again fitting for us to demonstrate the performance of a machine's memory hierarchy and to now explore how it features across different computing units. (Fatahalian et al. 2004).

We know that MMM exhibits computations that are highly parallelisable and in our Java programs we forwarded our tile "task parallel" approach. In this brief report, and accompanying OpenMP and OpenCL programs, we investigate how the parallelised blocking or tiling of MMM on OpenMP measures up against a "data parallel" approach on OpenCL.

First we briefly revisit the concepts of Matrix multiplication and cache-awareness, following such we describe how to bootstrap the accompanying OpenMp and OpenCL programs. Subsequently, we outline the included algorithms and their design. To conclude we outline our benchmarking process and findings.

2 MMM AND CACHE-AWARENESS

Performant dense MMM is in most cases related to the memory layout of the arrays. Even on modern CPUs with multi-level memory hierarchies, naive access patterns in the memory hierarchy can commonly incur cache capacity misses. We know that cache-oblivious algorithm to multiply matrices simply mimics computing the product by hand.

```
for (i = 0; i < rowAmount1; i++)
    for (j = 0; j < colAmount1; j++)
        for (k = 0; k < rowAmount2; k++)
            C[i][j] += (A[i][k] * B[k][j]);
                    ....
```

Elements of B are accessed column-wise and are not in sequential order in memory. Meaning each iteration in j reuses row i of A, that row may have been evicted from the cache by the time the inner-most loop completes.

The tall cache assumptions poses that a cache can store up to M/B blocks, for a total size of M elements, $M \geqslant B^2$ (Frigo et al. 2012). Therefore, under a cache-oblivious algorithm, a large enough matrix will not fit into a tall cache.

$$n = rowAmountOfA = colAmountOfA = rowAmountOfB$$

$$n > M/B$$

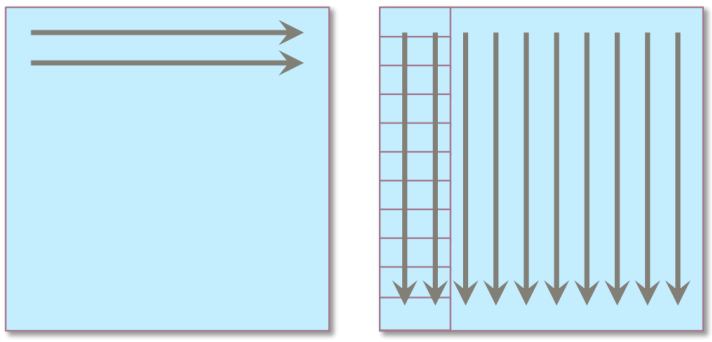

Figure 1: Naive row-major layout of A and B

Lending from our previous report, our illustration (figure 1) depicts the red rectangles in matrix B as our blocks of cache. When traversing B the accessing of each element is cached. Meaning if $n = colAmountOfA$ and $n > M/B$ then before the traversal of a column B completes the first, and possibly several subsequent, cached blocks of B will be evicted. Hence, upon the next and remaining traversals of B we may incur cache-misses on each block. Therefore a cache-oblivious algorithm with $n > M/B$ results in the following work done

$$Q(n) = \Theta(n^3)$$

Additionally a cache-oblivious algorithm with $n \leqslant M/B$ will get the bellow work done since the B exploits spatial locality

$$Q(n) = \Theta(n^3)/B$$

3 DELIVERABLES

3.1 Accompanying Files

File	Description
matrix_utils	Application folder for a shared utilities library
matrix_utils/src	Source folder for shared utilities
matrix_utils/src/dbg.h	Macro functions for logging
matrix_utils/src/miniunit.h	Macro functions and variables for unit testing
matrix_utils/src/matrixutils.h	Header for matrix utilities
matrix_utils/src/matrixutils.c	Implementation of matrix utilities
matrix_utils/tests	Test folder for shared utilities
matrix_utils/tests/dbg_test.c	Unit tests for dbg.h
matrix_utils/tests/matrixutils_test.c	Unit tests for matrixutils.c
openMP_app	Application folder for OpenMP
openMP_app/src	Source folder for application
openMP_app/src/mp_mm.h	Header for matrix multiplication
openMP_app/src/mp_mm.c	Implementation of matrix multiply
openMP_app/src/profiling	Source profiling folder
openMP_app/../mp_profiling.c	Main script for running benchmarks
openMP_app/tests	Test folder for OpenMP application
openMP_app/tests/mp_mm_test.c	Unit tests for mp_mm.c
openCL_app	Application folder for openCL
openCL_app/src	Source folder for application
openCL_app/src/cl_mm.h	Main OpenCL host implementation
openCL_app/src/mm.cl	OpenCL kernel for matrix multiply
openCL_app/src/profiling	Source profiling folder
openCL_app/../cl_profiling.c	Main script for running benchmarks
Makefile	commands for building, linking, cleaning, testing and profiling matrix_utils, openMP_app and openCL_app
runtests.sh	script for running units
openMP_results.csv	Results outputted from openMP profile Make task
openCL_results.csv	Results outputted from openCL profile Make task

3.2 Bootstraping

The following section outlines instructions to guide one in configuring, testing and running the accompanying OpenMP and OpenCL implementations on a OS X (10.10) environment. Before proceeding one should note that both the Makefile and runtests.sh have been tested only on Unix Z shell, to respect other shells some alterations may need to be made.

Additionally, all sources in matrix_utils and openMP_app have been successfully compiled and linked with GCC 4.9.2 (Make variable $(CC)). Bellow outlines the full version information of this GCC.

```
COLLECT_GCC=gcc-4.9
Target: x86_64-apple-darwin14.1.0
Configured with: ../configure --build=x86_64-apple-darwin14.1.0
--prefix=/usr/local/Cellar/gcc/4.9.2_1
--libdir=/usr/local/Cellar/gcc/4.9.2_1/lib/gcc/4.9
--enable-languages=c,c++,objc,obj-c++,fortran
--program-suffix=-4.9
Thread model: posix
gcc version 4.9.2 (Homebrew gcc 4.9.2_1)
```

However the sources in openCL_app have been successfully compiled with the native OS X GCC bundled with xCode 6.3.1, the LLVM front-end 6.1.0 (Make variable $(CL_CC)). Bellow we can see the full version of this GCC.

```
Configured with:
--prefix=/Applications/Xcode.app/Contents/Developer/usr
--with-gxx-include-dir=/usr/include/c++/4.2.1
Apple LLVM version 6.1.0
(clang-602.0.49) (based on LLVM 3.6.0svn)
Target: x86_64-apple-darwin14.3.0
Thread model: posix
```

Using older GCC versions should be fine as our sources only use basic features from the OpenMP and OpenCL libraries. Given the case that one wishes to use a different GCC version and/or use new compiler flags, simply override the Makefile variables as follows:

```
make footask CC=glol CFLAGS=-rofl
```

Given that we have our respecting compilers and flags in place, we can now configure, test and run our accompanying OpenMP and OpenCL implementations. Such is acheivable through the following Make tasks:

1. Building and testing the applications

   ```
   make clean all
   ```

2. Build, run and benchmark the openMP application

   ```
   make clean profile-openMP
   ```

3. Build, run and benchmark the openCL application

   ```
   make clean profile-openCL
   ```

4 ALGORITHM DESIGN

4.1 CPU Tiled Matrix-Matrix Multiply

Our OpenMP solution uses "tiling" to attempt to resolve cache capacity misses. Tiling is a type of loop transformation on which many cache blocking algorithms are built. It involves a "combination of strip-mining and interchange in multiply-nested loops", which in turn "strip-mines several nested loops and performs interchanges to bring the strip-loops inward" (Murphy 2015).

Our tiling solution aims to ensure good spatial locality for accesses across matrix B. Simple but effective, our implementation simply partitions the matrices' iteration space though such strip mining and loop interchange.

Although not implemented recursively, tiling shares some similarities to the divide-and-conquer approach in that it aims to refine the problem size through partitioning the matrices into submatrices (Frigo et al. 2012). Thus the design of the tiling solution quite naturally lends itself to parallelisation, and we achieve this as follows:

```
s = (int) ceil((rowAmount1 * 1.0)/30);
s2 = (int) ceil((colAmount1 * 1.0)/30);
s3 = (int) ceil((rowAmount2 * 1.0)/30);
#pragma omp parallel for \
        shared(A_p,B_p,C_p,aRows,aCols,bRows,bCols) \
        private(i,j,k,i2,k2,j2)
for (i = 0; i < rowAmount1 / s; i+=s)
 for (j = 0; j < colAmount1 / s2; j+=s2)
  for (k = 0; k < rowAmount2 /s3; k+=s3)
   for (i2 = i; i2 < (i+s) && i2 < rowAmount1; i2++)
    for (j2 = j; j2 < (j+s2) && j2 < colAmount1; j2++)
     for (k2 = k; k2 < (k+s3) && k2 < rowAmount2; k2++)
       C[i2][j2] += (A[i2][k2] * B[k2][j2]);
                          ....
```

Similar to our Java tiling solution, we use s, s2 and s3 parameters in our tiling algorithm. We can see that they are derived respectively from a matrix dimension divided by 30. They are then later utilised to determine the dimensions (size s2xs3) of the submatrix-submatrix multiplications to be ran. In the case of a square MMM the submatrices' dimensions would be sxs. We also use the expected "parallel for" OpenMP clauses to parallelise the algorithm.

Determining values for s, s2 and s3 is not straight-forward, they ultimately determine the tiling size which in turn demands an accurate estimate of the cache size on your target machine. Assuming square matrices which do not fit into a tall cache then

$$n = rowAmountOfA = colAmountOfA = rowAmountOfB$$

$$n > M/B$$

Once we determine $s \leqslant M/B$ we can easily decompose submatrices or tiles from matrix B that respect the target cache size (Figure 2).

As you would expect the tiling optimisation still results in $\Theta(n^3)$ computations. With an optimal s value, $\Theta(M^{1/2})$, we see an improvement in the amount work done to that of our naive approach. The tall-cache assumption implies that each submatrix and its data will fit in cache (Frigo et al. 2012), meaning only cold caches misses should occur per submatrix ($\Theta(s^2/B)$).

$$\Theta(n) = \Theta((n/s)^3(s^3)) = \Theta(n^3)$$

$$Q(n) = \Theta((n/s)^3(s^2/B)) = \Theta(n^3/BM^{1/2})$$

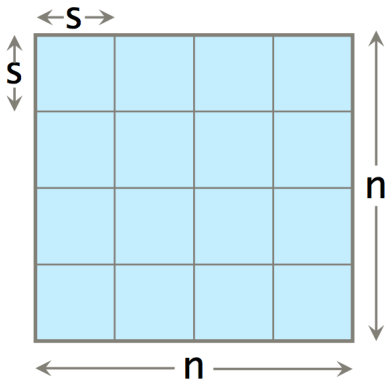

Figure 2: Submatrices of matrix B based on s

4.2 GPU Matrix Multiply

Conceptually, a GPU implements the master-slave programming model whereby the GPU operates as a slave under the control of a master or host processor. Thus, under OpenCL we are required to write a host program that runs the master processor. Our host program sends the matrices' data to the slaves, invoking a kernel/function that runs on the slave which processes the sent data, and finally receiving the results back from the slave.

Another programming model posed by GPUs is the Single Instruction Multiple Thread model. Such involves the GPU accomplishing a computational task using thousands of light weight threads. Simply, these threads are grouped into blocks and the blocks are organised as a grid. While a block of threads may be 1-, 2-, or 3-dimensional, the grid of blocks can only be 1- or 2-dimensional. As you would expect, for MMM we only require a 2-dimensional block of threads.

Kernel invocation require the specification of the block and grid dimensions along with any parameters the kernel may have. Illustrated bellow in our OpenCL MMM kernel, mm.cl.

```
__kernel void cl_mm(__global float *A_mem, __global float *B_mem,
                    __global float *C_mem, int a_rows, int b_rows)
{
    int i = get_global_id(0);
    int j = get_global_id(1);

    int k;
    float value = 0.0;
    for (k=0; k<a_rows; k++)
        value += A_mem[j*a_rows+k] * B_mem[k*b_rows+i];
    C_mem[j*a_rows+i] = value;
}
```

Each kernel execution performs one dot-product associated with one output matrix element. Contrasting our OpenMP solution, we do not at all change the asymptotic complexity, instead we reduce the constant factor significantly. As each thread in a work-group executes in parallel, we expect larger work-group sizes to take advantage of maximum concurrency.

It is important to note that the overhead involved in moving data on and off the GPU (including the matrices themselves, the Kernel instructions, and the work-queue information) is significant. Not covered by our implemen-

tation, but worth noting, are the memory types in the OpenCL memory hierarchy. In OpenCL the slowest memory type is global memory, which is available to all threads in all work-groups. A faster memory type is local memory, which is only visible to threads of the work-group it belongs to.

Unlike global memory, local memory is not initialised and transferred to the device from the host. Work-item threads must fill local memory before they can use it, a kernel can then fill one location per thread and implement a synchronisation barrier to wait for all of the other threads to fill up the rest of the data. Therefore the memory write pattern is heavy on local memory and only one write to the global memory occurs.

5 FINDINGS

5.1 Benchmarking Results

Our benchmarking goal was to analyse the trends displayed across the parallelised executions of CPU-bound and GPU-bound MMM. Thus our benchmarking process undertaken had the following features:

1. A varying set of dimensions for A and B respectively on both algorithms were used to identify trends. The dimensions (200x200), (200x300), (1000x1000) and (1000x1200) were used.
2. Each benchmark was ran five times per dimension in efforts to identify out outliers.
3. To maintain measurement accuracy the elements held by A and B needed to be the same for each measurement. Every element held by A is always 3, while each element in B is always 3.
4. Each benchmark was conducted on one machine under the same conditions (no other user processes running, machines connected to power outlets, etc.)

The accompanying open*_results.csv files contain the outputs of the OpenMP and OpenCL benchmarks ran respectively. The machine on which these were ran holds a four core Intel i5 CPU with an Intel HD Graphics 4000 GPU.

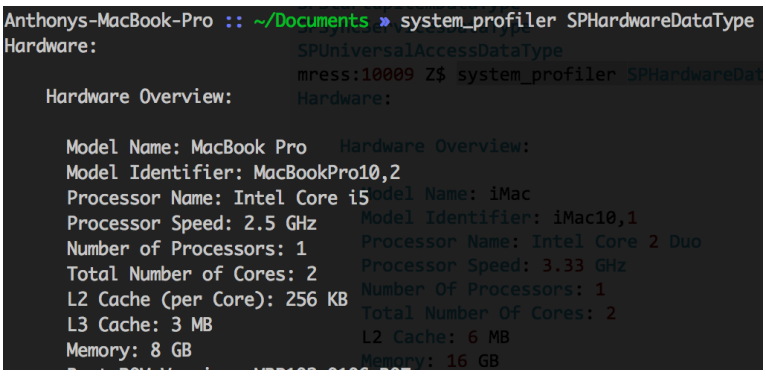

Figure 3: i5 CPU specification

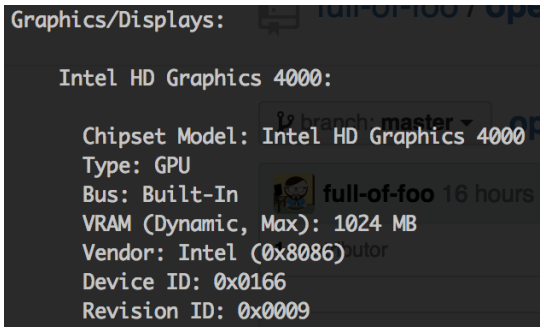

Figure 4: Intel HD Graphics 4000 GPU specification

Bellow we see the results per second of the OpenMP (Figure 5) and OpenCL (Figure 6) benchmark results over five runs.

	1	2	3	4	5
200x200	0.000083	0.000069	0.000074	0.000071	0.000051
200x300	0.000071	0.000098	0.000082	0.000084	0.000074
1000x1000	0.000125	0.000160	0.000917	0.000196	0.000210
1000x1200	0.000205	0.000160	0.000193	0.000140	0.000180

Figure 5: OpenMP benchmarks

	1	2	3	4	5
200x200	0.003738	0.003752	0.003723	0.003755	0.003742
200x300	0.003745	0.003736	0.003768	0.003759	0.003752
1000x1000	0.313471	0.214306	0.213094	0.212716	0.213695
1000x1200	0.210525	0.211589	0.208697	0.218218	0.215472

Figure 6: OpenCL benchmarks

Our results forward some interesting discussion points. On the surface it appears that one should favour a CPU-bound solution for MMM over a GPU-bound counterpart. However, as previously highlighted, we must consider that our OpenCL solution is somewhat naive in that it shares global memory rather than utilising OpenCL local memory.

REFERENCES

Fatahalian, K., Sugerman, J. & Hanrahan, P. (2004), Understanding the efficiency of gpu algorithms for matrix-matrix multiplication, *in* 'Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware', HWWS '04, ACM, New York, NY, USA, pp. 133–137.
URL: *http://doi.acm.org/10.1145/1058129.1058148*

Frigo, M., Leiserson, C. E., Prokop, H. & Ramachandran, S. (2012), 'Cache-oblivious algorithms', *ACM Trans. Algorithms* **8**(1), 4:1–4:22.
URL: *http://doi.acm.org/10.1145/2071379.2071383*

Murphy, M. (2015), 'Loop parallelism'.
URL: *https://patterns.eecs.berkeley.edu/?page*$_{\mathrm{i}}$d = 562