CS61C
Lab3a: Cache Blocking
Fall 2010

TA: Andrew Gearhart

September 30, 2010

1 Goals

- Obtain an introduction to the optimization technique of cache blocking
- Develop a better intuition for how memory access patterns affect code performance

2 Lab Materials

NOTE: This lab requires the use of machines in 200 SD!!!!

Copy ~**cs61c/labs_resources/3a/matmul.tar.gz** into your home directory. Untar and unzip the folder with the command **"tar -zxvf ./matmul.tar.gz"**. Inside the newly-created folder "matmul", you should find a Makefile, benchmark.cpp, dgemm-naive.cpp, dgemm-blocked.cpp and dgemm-blas.cpp. Make sure to **execute "export GOTO_NUM_THREADS=1"** to get accurate results from the tuned library.

3 Background

Cache blocking is a technique that attempts to reduce the cache miss rate by **improving the temporal and spacial locality of data accesses**. More intuitively, blocking tries to keep memory accesses confined in a small area of memory so that the amount of useful data in cache is improved. For this lab, you will be implementing a cache blocking scheme for a dense matrix multiply algorithm and explore some interesting patterns in code performance. The definition of the matrix multiply operation (for square matrices A,B of size $N \times N$) is

$$(AB)_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$$

where $1 \leq i \leq N$ and $1 \leq j \leq N$. Less formally, one can see that each element of the matrix product ($(AB)_{ij}$) requires the multiplication and sum of row i of A and column j of B. From a computational standpoint, we realize that a matrix multiply function can be implemented by 3 nested loops. This basic implemention is shown in dgemm-naive.cpp. Note that even though A and B are two-dimensional in dgemm-naive.cpp they are stored in a 1D array. Basically, the arrays are formed by making a list of the columns of A and B. This is called "column-major" representation. For example:

$$C = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \rightarrow [c_{11}, c_{21}, c_{31}, c_{21}, c_{22}, c_{23}, c_{31}, c_{32}, c_{33}]$$

If we think critically about the basic/naive implementation, we realize that the function in dgemm-naive.cpp moves through a very large space of memory to compute a single element of the product. As such, we are constantly accessing new values from memory and obtain very little reuse of cached data!

Thankfully, the structure of matrix multiply (and many other linear algebra routines) provides us with an alternative approach. For example, if we consider square A and B with N=6, we can define the matrices in this fashion:

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix}$$

where $A_{11} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$. From this "blocked" representation, we can now calculate the matrix product as such

$$AB = \begin{bmatrix} (A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}) & (A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32}) & (A_{11}B_{13} + A_{12}B_{23} + A_{13}B_{33}) \\ (A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}) & (A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32}) & (A_{21}B_{13} + A_{22}B_{23} + A_{23}B_{33}) \\ (A_{31}B_{11} + A_{32}B_{21} + A_{33}B_{31}) & (A_{31}B_{12} + A_{32}B_{22} + A_{33}B_{32}) & (A_{31}B_{13} + A_{32}B_{23} + A_{33}B_{33}) \end{bmatrix}$$

In this form, the calculation of the N=6 product is the same as applying the N=3 naive approch to submatrices of size N=2! In the calculation of each smaller matrix multiply (the $A_{ik}B_{kj}$) we are only accessing a small number of values, so the amount of cache reuse is significantly improved. If we make the size of the blocks to be $O(cachesize)$, we can significantly improve code performance.

In summation, applying cache blocking to matrix multiply involves a triply-nested loop over blocks that computes a BLOCKSIZE*BLOCKSIZE submatrix multiplication at each step. This technique can be applied multiple times to block for successive levels of cache.

4 Processor Characteristics

The desktop machines in 200 SD are dual-socket (two physical processor chips) with quad-core Intel Xeon E5520 processors running at 2.26 Ghz (for a total of 8 cores). These processors are based upon the Nehalem microarchitecture (codename Gainestown) and have 3 levels of cache: L1 Data (32 KBytes/core), L1 Instruction (32 KBytes/core), L2 (256 KBytes/core), L3 (8 MBytes shared). The machines currently have 12 GBytes of DRAM (main memory).

5 Exercises/Questions

Type "make benchmark-naive" and "make benchmark-blas" to build the naive and tuned BLAS (Basic Linear Algebra Subprograms) matrix multiply codes. If you run these programs, you can see an output of machine performance for various matrix sizes.

1. **Looking at the output, why do you think performance degrades significantly between matrices of size 512 and size 639?**

2. The executable benchmark-blas calls a highly-optimized matrix multiply routine that can be considered a practical upper bound on serial performance. Note that the perforance of this routine is significantly higher than the naive case.

   Look at the file dgemm-blocked.cpp. This is a sketch of a blocked matrix multiply, but with some portions incomplete.

   Replace the "?" portions of dgemm-blocked.cpp to create a working blocked matrix multiply. Show your TA that this code compiles and runs correctly (compile with "make benchmark-blocked"). What performance differences can be seen between the naive and blocked codes?

3. Once you have finished the blocked version of matrix multiply, try a couple different values of BLOCK_SIZE and consider differences in performance.

   From the above processor description, what would be some good block sizes to try? Explain your reasoning to your TA.

4. If we use BLOCK_SIZE=43, there are serious performance dips at 256,512 and 768.

   Theorize why performance is so bad for such cases. Explain to the TA.