

Project 1.1
Analytical Models of Locality within Matrix
Multiply Algorithms

Emily Bragg, Cagri Eryilmaz, Ali Fakhrazadehgan

January 30, 2015

1 INTRODUCTION

2 SINGLE LEVEL ANALYTICAL MODELS

2.1 ASSUMPTIONS

In this model there is only level of cache (L1) and there is no Register File in the architecture, so that the processor only operates on the data in memory, which obviously if it is present in L1 (Hit) it can be used directly and if it is not (Miss) it should be gathered first.

Cache organization is considered to be a *fully associative*, ideal *LRU* replacement policy, with cache-line/blocks size of L and total cache size is Z .

The analytical metric used here is the Arithmetic Intensity of the Matrix Multiplication. Basically, the multiplication algorithm is the well-known $O(N^3)$ algorithm and in each step we are trying to optimize this procedure. Notice that, optimization can be done both on computation complexity and cache complexity of the algorithm, but here the main focus is to improve the cache complexity.

2.2 MULTIPLICATION MODELS

2.2.1 BASELINE MATRIX MULTIPLICATION ALGORITHM

As it is provided in the attachments of this report, baseline core computation of matrix multiply looks like below.

Algorithm 1 Baseline Matrix Multiplication

```
for  $i = 0; i < N; i++$  do
  for  $j = 0; j < N; j++$  do
     $C[i][j] = 0;$ 
    for  $k = 0; k < N; k++$  do
       $C[i][j] += A[i][k] \times B[k][j];$ 
    end for
  end for
end for
```

By looking the Algorithm 1, the asymptotic computation complexity could be figured out, which is $O(N^3)$. However, since we need a more accurate metric here, we consider total number of arithmetic operations as $2N^3$ ¹. For this point on, since core of the computation is not changed, we consider number of operations to be the same.

Now we focus on the cache complexity of the computation. If we start from the third loop (the most inner one), we can see there are three data access. However, between these three, hopefully $C[i][j]$ is almost always present in the cache. This statement is true because before entering this loop, $C[i][j]$ is cached and it will resident for this whole loop according to the

¹One + and one \times .

replacement policy².

An important thing about the A is, it is being traced in a row-major order, hence we can utilize the locality in each cache-line. Since each row of A is N words, it will require $\frac{N}{L}$ transfers of cache-lines.

Matrix B is being trace in the column-major in this algorithm and since we are considering matrixes to be large in comparison with cache-line and cache size, there is almost no chance to observe any locality between consecutive access. Therefor, in each iteration B will experience almost N misses.

Putting it all together, B will produce total number of N^3 . However, A will produce $\frac{N^3}{L}$ misses. Why? If we look closer to the access pattern of B this accesses are re-writing the whole cache (with this assumption that B is much larger than cache). Thus, although there is this opportunity of having *Hits* from A in the next iteration of middle loop, B 's access pattern removes these blocks. The important thing about C is that it is being accessed in the second loop (the middle one) and begin traced in row-major order. Consequently, the total number of misses from C would be $\frac{2N^2}{L}$. Using the arithmetic intensity of the operations, we will get:

$$\text{Arithmetic Intensity} = \frac{2N^3}{\frac{N^3}{L} + N^3 + \frac{2N^2}{L}}$$

2.2.2 REORDERED MULTIPLICATION AND USING TRANSPOSITION

In the last part the main reason for losing benefits of locality in consecutive accesses of B was its access pattern. As mentioned above, this matrix is being access in column-major, so every access brings a new cache-block to the cache which will be used only once. This part is about improving this behavior.

As the first step, we can change the access pattern of B from column-major to row-major. By doing so, we can benefit from using all data words available in the cache-block brought into the cache in the previous iterations. This can simply done, by the following algorithm.

Algorithm 2 Matrix Multiplication with Reordering B Accesses

```

for  $i = 0; i < N; i++$  do
  for  $k = 0; k < N; k++$  do
    for  $j = 0; j < N; j++$  do
       $C[i][j] += A[i][k] \times B[k][j];$ 
    end for
  end for
end for

```

What is being done in this algorithm is like this. Consider A , B , and C as below.

$$\begin{bmatrix} C_{00} & \dots & C_{0(N-1)} \\ \dots & \dots & \dots \\ C_{(N-1)0} & \dots & C_{(N-1)(N-1)} \end{bmatrix} = \begin{bmatrix} A_{00} & \dots & A_{0(N-1)} \\ \dots & \dots & \dots \\ A_{(N-1)0} & \dots & A_{(N-1)(N-1)} \end{bmatrix} \times \begin{bmatrix} B_{00} & \dots & B_{0(N-1)} \\ \dots & \dots & \dots \\ B_{(N-1)0} & \dots & B_{(N-1)(N-1)} \end{bmatrix}$$

²Since every assignment in this loop is addressing $C[i][j]$ repeatedly, so LRU will not remove it from the cache.

Here, instead of producing each element of C , this algorithm tries to first produce partial multiplications of each element and then, in each step aggregates the values. For instance, one complete execution of the most inner loop (i.e. one iteration of middle loop), all of the first partial multiplications of first row of the C is done.

This approach is good and bad. This is good, because accesses to B become very similar to accesses to A . In other words, accessing B in row-major results in less number of misses when accessing B , which is $\frac{N^3}{L}$. And it also may help removing A block from the cache, because there are less number of blocking coming in/out to cache, so for A we have $\frac{N^2}{L}$.

Although this reordering decreased number of B 's misses, but this can be harmful for C . Now, we have overhead on C . Why? Because computation of each element of C is divided to many partial expressions which are being calculated/stored not necessarily on the same element/cache-block of C . In other words, when each of the C 's cache-lines are being fetched more than once³. Thus, the inner loop on C looks very similar to access patterns of A and B , and produces total number of $\frac{2N^3}{L}$ misses (which means data movement). We can say:

$$\text{Arithmetic Intensity} = \frac{2N^3}{\frac{2N^3}{L} + \frac{N^2}{L} + \frac{N^3}{L}}$$

But still there is another idea, which instead of reordering the accesses to B , with some pre-process we can produce B^T . Using B^T instead of B and reordering the loops is another approach that can improve benefiting from the locality of B . Algorithm for this model using B^T is like below.

Algorithm 3 Matrix Multiplication using B^T

```

for  $i = 0; i < N; i++$  do
  for  $j = 0; j < N; j++$  do
     $C[i][j] = 0;$ 
    for  $k = 0; k < N; k++$  do
       $C[i][j] += A[i][k] \times B^T[j][k];$ 
    end for
  end for
end for

```

Similarly, this algorithm has same access patterns for A and B , however, it also benefits from the locality in C access. So:

$$\text{Arithmetic Intensity} = \frac{2N^3}{\frac{2N^2}{L} + \frac{N^2}{L} + \frac{N^3}{L}}$$

³it is better to complete to whole operation on one cache-line instead of bringing back and forth.

2.2.3 PARTITIONING THE MATRIX TO SPECIFIED SUB-MATRIXES

2.2.4 DIVIDE & CONQUER

2.3 GRAPHS

3 MODEL WITH TWO-LEVEL CACHE HIERARCHY

3.1 ASSUMPTIONS

3.2 MODEL

3.3 GRAPHS

4 MODEL WITH TWO-LEVEL CACHE HIERARCHY AND REGISTER FILE

4.1 ASSUMPTIONS

4.2 MODEL

4.3 GRAPHS

$$\begin{aligned}(x+y)^3 &= (x+y)^2(x+y) \\ &= (x^2+2xy+y^2)(x+y) \\ &= (x^3+2x^2y+xy^2) + (x^2y+2xy^2+y^3) \\ &= x^3+3x^2y+3xy^2+y^3\end{aligned}\tag{4.1}$$

Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

$$A = \begin{bmatrix} A_{11} & A_{21} \\ A_{21} & A_{22} \end{bmatrix}\tag{4.2}$$

Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem.

4.3.1 HEADING ON LEVEL 3 (SUBSUBSECTION)

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

HEADING ON LEVEL 4 (PARAGRAPH) Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

5 LISTS

5.1 EXAMPLE OF LIST (3*ITEMIZE)

- First item in a list
 - First item in a list
 - * First item in a list
 - * Second item in a list
 - Second item in a list
- Second item in a list

5.2 EXAMPLE OF LIST (ENUMERATE)

1. First item in a list
2. Second item in a list
3. Third item in a list