COMPUTER ARCHITECTURE (PARALLELISM & LOCALITY)

Project 1.1
Analytical Models of Locality within Matrix Multiply Algorithms

Emily Bragg, Cagri Eryilmaz, Ali Fakhrzadehgan

February 2, 2015

1 Single Level Analytical Models

1.1 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.

1.2 Multiplication Models

1.2.1 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 replacement policy[2].

[1] One + and one ×.

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

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:

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

In other words, for accesses of A in the most inner loop from each L number of accesses, $L-1$ of them is services by the first level of cache ($(L-1) * \frac{N}{L}$). So in total, $(L-1) * \frac{N^3}{L}$ of accesses are serviced via L1 and $\frac{N^3}{L}$ of them are missed (initiating a new transfer between memory and L1).

Obviously, since B's access pattern is in column-major, it will not utilize its available locality so 0.

Finally for C, from each L iterations of the middle loop, $L-1$ of are serviced via L1. So, cache effect would be $(L-1) * \frac{N^2}{L}$.

(This should be improved)This formulas make sense, since if we set $L = 1$ then number of useful accesses to L1 would be zero for all of them, which means we are completely loosing spatial locality, and of course since Z is small we cannot benefit from temporal, too. And making L larger and larger (obviously, assuming Z grows at the same time), the limit would be total number of access, which is true and means we are utilizing the whole available spatial-temporal locality.

1.2.2 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.

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

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 once[3]. 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 preprocess 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.

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

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

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

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
```

1.2.3 Partitioning Matrixes and Divide&Conquer (Cache aware and Cache Oblivious)

If consider the case which all of the matrixes are small enough which cache can hold them all (i.e. $3N^2 < Z$), then number of data transfers (so amount of cache block which is transferred between memory and cache) will be reduced. In this case, the arithmetic Intensity would be:

$$\text{Arithmetic Intensity} = \frac{2N^3}{\frac{4N^2}{L}}(Hmm?)$$

Thus, having small matrixes (in comparison with cache size) can improve locality or more accurately can improve utilizing the available locality. Smaller Matrixes can remain cache while they are being processes, so there is no extra overhead on transferring data (which reduces the denominator). The first idea to utilize this property is dividing each matrix to smaller sub-blocks which can be stored in cache completely (assuming that we know the size of the cache). Algorithm for this model of computation is similar to the base model with this difference that is being executed on $n \times n$ matrixes (Algorithm 4).

Algorithm 4 Matrix Multiplication with $n \times n$ blocks

```
for $I = 0; I < \frac{N}{n}; I++$ do
    for $J = 0; J < \frac{N}{n}; J++$ do
        $C_{IJ} = 0;$
        for $K = 0; K < \frac{N}{n}; K++$ do
            $C_{IJ}$ += MatMul( $A_{IK}$, $B_{KJ}$ );
        end for
    end for
end for
```

If we consider the size of the cache to be Z, $n \times n$ which can be completely stored in cache should have the property that $n < \sqrt{\frac{Z}{3}}$. Notice that, each of C_{IJ}, A_{IK}, and B_{KJ} are chunks of original matrix and + in the most inner loop is a Matrix Sum.

Assuming that $n < \sqrt{\frac{Z}{3}}$, for computing each sub-matrix we consume $\frac{N}{n} \times (2 \times n^2)$ operations, which in total that would be $\frac{N}{n} \times (2 \times n^2) \times (\frac{N}{n})^2 = \frac{2N^3}{n}$. Using our assumption we have $\frac{2N^3}{\sqrt{\frac{Z}{3}}}$.

Number of data transfers (which directly depends on number of cache misses) in this model can be calculated as follow. For each iteration of the middle loop we have: 1. Gathering C_{IJ} with size of n^2 and takes $\frac{n^2}{L}$ lines, 2. Gathering A_{IK} and B_{KJ} each of them for $\frac{N}{n}$ times. Hence, number of data transfers for the middle loop would be $\frac{2n^2}{L} + 2 \times \frac{N}{n} \times \frac{n^2}{L}$. This happens for $(\frac{N}{n})^2$, so $(\frac{N}{n})^2 \times (\frac{2n^2}{L} + 2 \times \frac{N}{n} \times \frac{n^2}{L}) = \frac{2N^2}{L} + \frac{2N^3}{nL} = \frac{2N^2}{L} + \frac{2N^3}{L\sqrt{\frac{Z}{3}}}$.

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

As you can see, assumption here is based on the size of cache Z. However, we can go further and also eliminate this assumption (Cache Oblivious). We can do so, if we divide matrixes repeatedly to finer block. Obviously from some point on, we will pass the unknown cache size and every variable would be accessible from cache. This leads us to the Divide&Conquer model which divides matrixes until it reaches some base case. Notice that, after passing the cache size further divisions does not have any impact.

1.3 Graphs

2 Model with Two-Level Cache Hierarchy

2.1 Assumptions

2.2 Model

2.3 Graphs

3 Model with Two-Level Cache Hierarchy and Register File

3.1 Assumptions

3.2 Model

3.3 Graphs

$$\begin{aligned}(x+y)^3 &= (x+y)^2(x+y) \\ &= (x^2+2xy+y^2)(x+y) \\ &= (x^3+2x^2y+xy^2)+(x^2y+2xy^2+y^3) \\ &= x^3+3x^2y+3xy^2+y^3\end{aligned} \tag{3.1}$$

Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

$$A = \begin{bmatrix} A_{11} & A_{21} \\ A_{21} & A_{22} \end{bmatrix} \tag{3.2}$$

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.

3.3.1 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.

4 Lists

4.1 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

4.2 Example of list (enumerate)

1. First item in a list

2. Second item in a list

3. Third item in a list