



CUDA Optimization with Matrix Multiplication


이정근 (Jeong-Gun Lee)

한림대학교 소프트웨어융합대학, 임베디드 SoC 연구실
www.onchip.net
 Email: Jeonggun.Lee@hallym.ac.kr

Contents

- Motivation
- Environment
- Implementations on **CPU**
- Implementations on **GPU**
 - Initial Code
 - Tiling
 - Memory Coalescing
 - Avoid Bank Conflict
 - Loop Unrolling
- Performance





Motivation

- In this talk, “**matrix multiplication**” is optimized on a GPU with various optimization strategies in CUDA
- “Matrix multiplication” is a very fundamental component in many scientific applications and is one of the most important examples of high performance parallel programming.

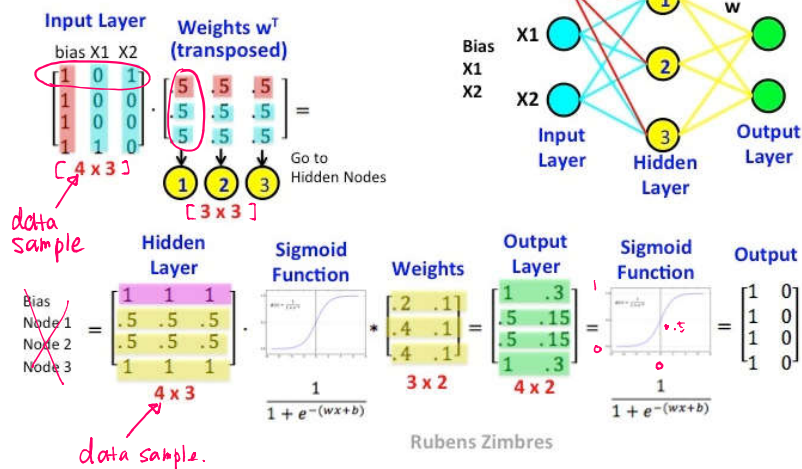
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 4 & 4 \\ 10 & 8 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 7 & 10 \end{bmatrix}$$



Motivation: MM in Neural Networks (신경망)

Color Guided Matrix Multiplication for a Binary Classification Task with N = 4



<https://www.datasciencecentral.com/profiles/blogs/matrix-multiplication-in-neural-networks>



CUDA Working Environment

- The project used Intel i7-3770 CPU and NVIDIA GeForce 690 GTX architecture.

	GeForce GTX 690
Global Memory	2GB
Number of cores	✓ 1536
Warp Size	✓ 32
Maximum number of threads per block	1024
Clock rate	1.02Ghz

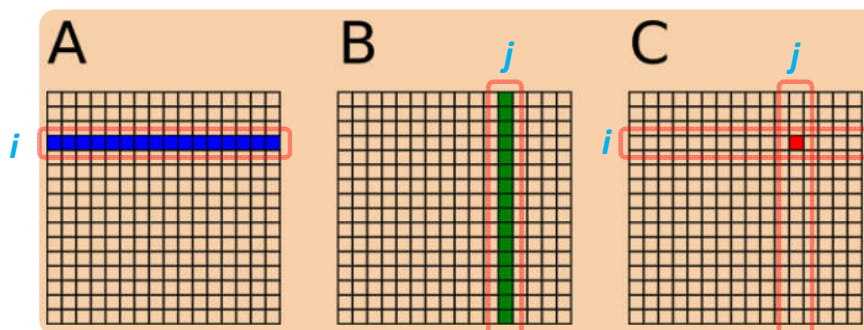
*low cost
← small for desktop.*



Implementation on CPU

- $n \times n$ matrix multiplication is easy !

$$\sum_{i=0}^n \sum_{j=0}^n C[i,j] = \sum_{i=0}^n \sum_{j=0}^n \sum_{k=0}^n A[i,k] * B[k,j]$$





Implementation on CPU

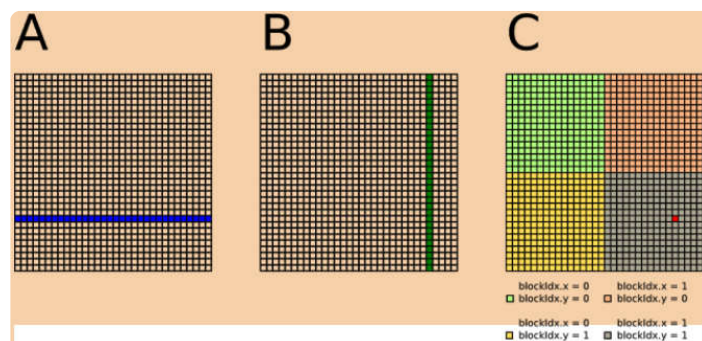
- Code ...

```
void MatrixMulC(float *M, float *N, float *P, int width)
{
    for (unsigned int i = 0; i < width; ++i)
    {
        for (unsigned int j = 0; j < width; ++j)
        {
            float sum = 0.0;
            for (unsigned int k = 0; k < width; ++k)
            {
                float a = M[i * width + k];
                float b = N[k * width + j];
                sum += a * b; // 한 행과 한 열의 곱에 대한 누적 합 계산
            }
            P[i * width + j] = sum; // 결과 행렬에 결과 대입
        }
    }
}
```



Implementation on GPU

- An initial GPU CUDA code allocates **one thread to compute one element of the matrix C**.
- Each thread reads one row of matrix A and one column of B from a global memory, proceeds with each multiplication and stores the result in C.





Implementation on GPU

- Initial Code working on a Global Memory

```
__global__ void MatrixMul(float *M, float *N, float *P, int width)
{
    int k = 0;                float accu = 0.0;
    // Block index
    int bx = blockIdx.x;      int by = blockIdx.y;
    // Thread index
    int tx = threadIdx.x;     int ty = threadIdx.y;

    // i, j는 행렬 M과 N에서의 각 스레드에 대한 시작 위치를 계산
    int i = by * blockDim.y + ty;    int j = bx * blockDim.x + tx;

    for(int k=0; k<width; k++)
    {
        // 한 스레드가 한 줄의 행과 한 줄의 열을 곱하여 누적 합을 계산
        accu = accu + M[ i * width + k ] * N[ k * width + j ];
    }
    P[ i * width + j ] = accu; // 최종 누적 합을 결과 매트릭스에 저장
}
```



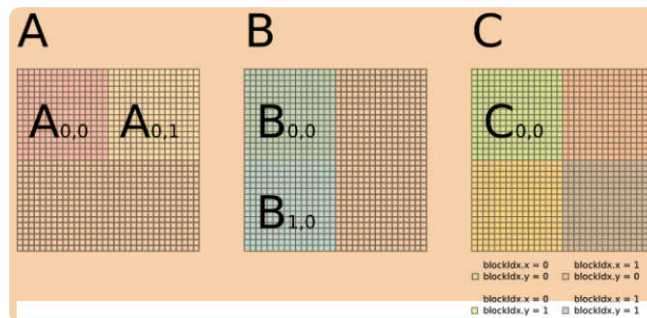
Implementation on GPU - *Tiling*

- In the initial code,
 - all threads load data from global memory, so performance is poor.
- In a tiled approach,
 - one thread block computes one tile of the matrix C.**
- One thread in the thread block calculates one element of the tile.



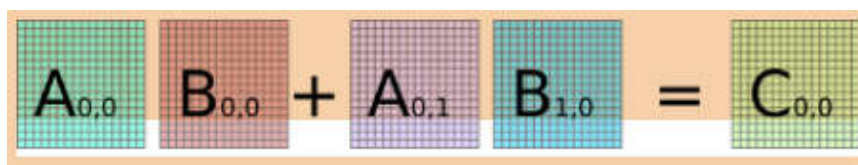
Implementation on GPU - *Tiling*

- One thread block computes one tile of the matrix C.
- One thread in the thread block calculates one element of the tile.
- The figure shows a **32 * 32 matrix** divided by **FOUR 16 * 16 submatrices**.
- To calculate this, you can create four thread blocks each with 16 * 16 threads.



Implementation on GPU - *Tiling*

- In each iteration, a thread block loads one tile of A and one tile of B from global memory into shared memory and performs the computation.
- At this time, the accumulated value is stored in the register.
- After all iterations, the thread block stores a tile of C in global memory.
- For example, a thread block may be repeated twice to compute C (0, 0).
 - $C(0,0) = A(0,0) * B(0,0) + A(0,1) * B(1,0)$



Implementation on GPU - *Tiling*

- In the first iteration, the thread block loads the A (0,0) tile and the B (0,0) tile into shared memory.
- Each thread creates an element of C and stores it in a register.** This accumulates in the next iteration.

The diagram illustrates the first iteration of GPU matrix multiplication tiling. It shows three matrices: A, B, and C. Matrix A is divided into tiles A_{0,0} and A_{0,1}. Matrix B is divided into tiles B_{0,0} and B_{1,0}. Matrix C is divided into tiles C_{0,0} and C_{1,0}. The Shared Memory contains tiles A_{0,0} and B_{0,0}. The Register File contains the accumulated result C_{0,0}. Arrows indicate the flow of data from Global Memory to Shared Memory and then to the Register File.

Implementation on GPU - *Tiling*

- In the second iteration, the thread block loads the A (0,1) tile and the B (1,0) tile into shared memory.
- Each thread accumulates after the operation.
- At the end of the final iteration, the C element of the register is stored in global memory.

The diagram illustrates the second iteration of GPU matrix multiplication tiling. It shows three matrices: A, B, and C. Matrix A is divided into tiles A_{0,0} and A_{0,1}. Matrix B is divided into tiles B_{0,0} and B_{1,0}. Matrix C is divided into tiles C_{0,0} and C_{1,0}. The Shared Memory contains tiles A_{0,1} and B_{1,0}. The Register File contains the accumulated result C_{0,0}. Arrows indicate the flow of data from Global Memory to Shared Memory and then to the Register File.



Implementation on GPU - *Tiling*

```

__global__ void MatrixMul(float *M, float *N, float *P, int width)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

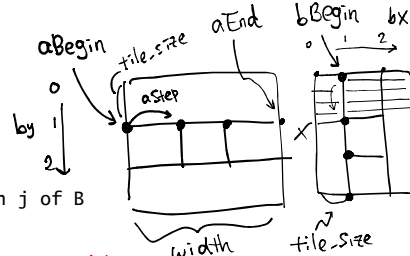
    // Accumulate row i of A and column j of B
    int i = by * blockDim.y + ty;
    int j = bx * blockDim.x + tx;

    const int tile_size = 16; // tile size

    // shared memory define
    __shared__ float As[tile_size][tile_size];
    __shared__ float Bs[tile_size][tile_size];

    // M 행렬에서의 시작 인덱스와 끝 인덱스, 그리고 증가량
    int aBegin = width * tile_size * by;
    int aEnd = aBegin + width - 1;
    int aStep = tile_size;

```



$bBegin = tile_size \times bx$
 $bStep = tile_size \times width$
 $bEnd = bBegin + width \times (width - 1)$



Implementation on GPU - *Tiling*

```

// N 행렬에서의 시작 인덱스와 증가량
int bBegin = tile_size * bx;
int bStep = tile_size * width;

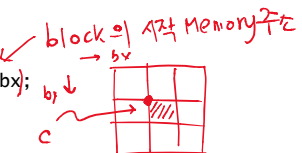
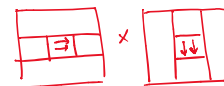
float Csub = 0;    int a, b, k;

for (a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
{
    // Shared Memory로 tile의 크기만큼 Load
    As[ty][tx] = M[a + width * ty + tx];
    Bs[tx][ty] = N[b + width * tx + ty];
    __syncthreads();

    // 각 스레드가 공유 메모리에서 해당 타일의 크기만큼 누적합 계산
    for (int k = 0; k < tile_size; ++k)
    {
        Csub += As[ty][k] * Bs[k][tx];
    }
    __syncthreads();
}

// 결과 매트릭스의 최종 인덱스에 저장
int c = (width * tile_size * by) + (tile_size * bx);
P[c + width * ty + tx] = Csub;

```



Opt: *Memory Coalescing*

- The condition that maximum bandwidth can be used when reading global memory is called **memory coalescing**.
- The way to access memory in CUDA is to split the warp in half and transfer 16 threads at the same time.

```

for (a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
{
    // Shared Memory로 tile의 크기만큼 Load
    As[ty][tx] = M[a + width * ty + tx];

    //Memory Coalescing : N[b + width * tx + ty] -> N[b + width * ty + tx]
    Bs[tx][ty] = N[b + width * ty + tx];
    __syncthreads();

    // 각 스레드가 공유 메모리에서 해당 타일의 크기만큼 누적합 계산
    for (int k = 0; k < tile_size; ++k)
    {
        //Memory Coalescing : Bs[k][tx] -> Bs[tx][k]
        Csub += As[ty][k] * Bs[tx][k];
    }
    __syncthreads();

    // 결과 매트릭스의 최종 인덱스에 저장
    int c = width * tile_size * by + tile_size * bx;
    P[c + width * ty + tx] = Csub;
}

```

Handwritten notes and diagrams:

- Shared Memory**: A diagram showing a grid of memory banks. Arrows indicate data flow from global memory to shared memory and then to the processing unit.
- Bank Conflict!**: A diagram showing a grid of memory banks. Arrows indicate data flow from global memory to shared memory and then to the processing unit. A red star marks a conflict point.
- Warp**: A diagram showing a grid of memory banks. Arrows indicate data flow from global memory to shared memory and then to the processing unit. A red star marks a conflict point.
- ty = 0**: A handwritten note indicating the row index.
- tx, k**: Handwritten notes indicating the column index and the inner loop index.

- The above code modifies the index to read contiguous memory areas from global memory.

Opt: *Memory Coalescing*

- Each SM has 16 KB of shared memory.
- Each shared memory consists of **16 banks**, each of which has 1 KB of memory.
- You can access once per GPU cycle for each bank, and you get the **greatest efficiency when the threads access the 16 banks in parallel at the same time**.
- Shared memory, like global memory, is accessed by dividing warps consisting of 32 threads into 16 threads (**Fermi**).
- When 16 threads read or write shared memory at one time, a bank conflict occurs when multiple threads attempt to access a bank.
- When 16 threads access the shared memory consisting of 16 * 16 in the column direction, a 16-way bank conflict occurs and the call rate is 1/16.
- Thus, simply accessing the threads in the row direction avoids bank conflicts.



Opt: *Avoiding Bank Conflict*

bank0	address	0	64	128	16320
bank1	address	4	68	132	16324
bank2	address	8	72	136	16328
...								
bank15	address	60	124	188	16380

```

for (a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
{
    As[ty][tx] = M[a + width * ty + tx];    Bs[ty][tx] = N[b + width * ty + tx];
    __syncthreads();

    for (int k = 0; k < tile_size; ++k)
    {
        Csub += As[ty][k] * Bs[k][tx];
        __syncthreads();
    }

    int c = width * tile_size * by + tile_size * bx;
    P[c + width * ty + tx] = Csub;
  
```

Handwritten annotations:

- Diagrams above the code show memory access patterns for M , As , N , and Bs with arrows indicating the direction of access.
- Red arrows point from $As[ty][k]$ and $Bs[k][tx]$ to the $__syncthreads();$ line, with the word "broadcast" written below.
- A red star is next to the inner loop's closing brace.
- Red text "No bank conflict!" is written next to the inner loop.



Opt: *Loop Unrolling*

- Loop unrolling is to determine if the same operation can be performed with fewer instructions.
- When the loop is removed, the offset is a constant.
- This has the effect of internally eliminating the loop branch, incrementing the induction variable, and calculating the internal iteration address.

```

for (int k = 0; k < tile_size; ++k)
{
    Csub += As[ty][k] * Bs[k][tx];
}
  
```

Handwritten annotations:

- Red text "sequential!" is written below the code.
- Red arrows point from $As[ty][k]$ and $Bs[k][tx]$ to the $__syncthreads();$ line, with the word "broadcast" written below.


Opt: *Loop Unrolling*

- Unrolling

```

for (int k = 0; k < tile_size; ++k)
{   Csub += As[ty][k] * Bs[k][tx];   }

```



Step

1 Csub += As[ty][0] * Bs[0][tx];

2 Csub += As[ty][1] * Bs[1][tx];

3 Csub += As[ty][2] * Bs[2][tx];

⋮ Csub += As[ty][3] * Bs[3][tx];

⋮ Csub += As[ty][4] * Bs[4][tx];

⋮ Csub += As[ty][5] * Bs[5][tx];

⋮ Csub += As[ty][6] * Bs[6][tx];

⋮ Csub += As[ty][7] * Bs[7][tx];

⋮ Csub += As[ty][8] * Bs[8][tx];

⋮ Csub += As[ty][9] * Bs[9][tx];

⋮ Csub += As[ty][10] * Bs[10][tx];

⋮ Csub += As[ty][11] * Bs[11][tx];

⋮ Csub += As[ty][12] * Bs[12][tx];

⋮ Csub += As[ty][13] * Bs[13][tx];

⋮ Csub += As[ty][14] * Bs[14][tx];

⋮ Csub += As[ty][15] * Bs[15][tx];

Performance Comparison

- Pthread .vs. AVX .vs. OpenMP .vs. GPU

Matrix Size	CPU	pthread	AVX	openMP		
256	0.06	0.04	0.09	0.04		
512	0.64	0.20	0.48	0.21		
1024	7.17	2.33	3.25	2.38		
2048	156.19	36.59	25.50	38.05		
4096	1311.08	333.69	206.35	349.36		
		GPU				
Matrix Size	CPU	Initial	Tiling	Coalescing	Bank Conflict	Loop Unrolling
256	0.06	0.13	0.11	0.11	0.11	0.11
512	0.64	0.13	0.11	0.11	0.11	0.11
1024	7.17	0.19	0.15	0.15	0.14	0.12
2048	156.19	0.45	0.41	0.43	0.38	0.26
4096	1311.08	3.60	3.50	3.60	3.30	2.20



Reference

- <https://github.com/lzhengchun/matrix-cuda>
- <https://piazzaresources.s3.amazonaws.com/i48o74a0lqu0/i5c0b5lnj4i5kf/7CUDAOptimization1.pdf?AWSAccessKeyId=AKIAIEDNRLJ4AZKBW6HA&Expires=1500576130&Signature=GO6IS%2FUz6gfSgXGQml2A9aaB%2B04%3D>