ZKML 가속을 위한 고성능 GPU 행렬 곱셈 설계 및 Montgomery 곱셈 최적화

ZKML은 데이터의 프라이버시를 보호하면서도 인공지능 모델의 추론 결과를 신뢰할 수 있게 해주는 핵심 기술로 주목받고 있다. 그러나 ZKML의 증명 생성 과정은 거대한 소수 체 위에서의 수많은 행렬 곱셈 연산을 수반하기 때문에, 극심한 연산 및 메모리 병목 현상을 겪게 된다. 기존의 행렬 연산은 메모리 크기가 정해지면 총 연산 횟수가 고정되므로, 연산 유닛의 처리량을 극대화하는 것과 더불어 메모리 접근 빈도를 최소화하는 것이 필수적이다. 이에 본 연구에서는 ZKML 가속을 목표로 GPU 아키텍처 기반의 다각적인 최적화 기법을 적용 및 분석하였다. 구체적으로, 하드웨어 계층에서는 메모리/레지스터 Bank Conflict 분석과 하드웨어 가속기(Tensor Core)의 특성을 파악하였다. 알고리즘 계층에서는 Dense Matrix에 대한 Tiling 및 Packing 기법을 검증하고, 동적 업데이트가 빈번한 Sparse Matrix을 위한 COO 버퍼 및 Lazy Update 전략을 제안한다. 마지막으로 암호학적 연산 계층에서는 ZKML 특유의 모듈러 나눗셈 병목을 해소하기 위해 Montgomery Multiplication을 도입하였다. 본 리포트는 이러한 다계층(Multi-layer) 최적화가 전체 시스템 성능에 미치는 영향을 정량적으로 입증하고자 한다.

1.1 Memory Hierarchy

gpu는 global memory와 shared memory, register 이렇게 3개로 나누어져 있다. Global memory는 모든 block이 공유하는 memory로 크기는 가장 크지만 가장 느리다. 반면 shared memory는 block간의 thread들이 공유할 수 있는 memory로 global memory보다는 크기가 작지만 global memory보다 훨씬 빠르게 접근할 수 있다. register는 각각의 thread마다 가지고 있는 memory로, 가장 빠르지만 크기가 매우 작다.

1.2 Bank conflict

Shared memory는 32개의 bank로 나누어져있다. 하나의 thread 만이 하나의 bank에 접근할 수 있다. 만약 여러 개의 thread가 하나의 bank에 접근한다면 동시에 접근하지 못해 성능이 떨어지게 된다.

성능을 측정한 코드는 다음과 같다.

```
template <int STRIDE>
__global__ void shared_bank_conflict_kernel(float* out) {
    volatile __shared__ float s_data[8192];

    int tid = threadIdx.x;

    int idx = tid * STRIDE;

    s_data[idx] = idx;
    __syncthreads();

    for(int i = 0; i < 100000; i++) s_data[idx] = s_data[idx] + 0.01f;

    out[tid] = s_data[idx];
}
```

STRIDE가 1인 경우에는 한 warp에 있는 thread들이 서로 다른 bank에 접근하여 shared memory bank conflict가 발생하지 않지만, STRIDE가 32인 경우, 같은 bank에 접근하여 shared memory bank conflict가 접근한다.

bank conflict가 없는 경우에는 19.01ms가, bank conflict가 있는 경우에는 581.22ms로 약 30배 정도 차이가 났다.

1.3 register conflict

Volta 아키텍처의 레지스터 파일은 두 개의 뱅크로 나뉘어 관리된다. 각 스레드는 한 번에 각 뱅크에서 하나의 레지스터만 읽어올 수 있기 때문에, 만약 동일한 뱅크에 속한 여러 레지스터에 동시에 접근하면 Bank Conflict이 발생하여 Latency가 생기게 된다.

Register conflict를 측정하기 위해 발생한 troubleshooting은 다음과 같다.

1. PTX inline assembly를 통한 직접 할당 시도

처음에는 cuda 코드 내에서 asm 구문을 사용하여 레지스터 번호를 직접 지정하는 방식을 시도하였다. 하지만 PTX에서는 의도한 대로 레지스터가 매핑된 것처럼 보였으나, SASS 수준에서 확인해 본 결과, ptxas 컴파일러의 레지스터 할당기가 자체적인 최적화를 수행하여 사용자가 지정한 레지스터 번호를 임의로 재배치하는 문제가 발생하였다.

2. Assembly 직접 수정 시도 및 에러

컴파일러의 개입을 막기 위해, 컴파일된 어셈블리 코드를 직접 수정하여 물리적 레지스터 번호를 강제로 맞추고자 하였다. 그러나 하드웨어가 요구하는 스레드당 레지스터 할당 개수가 메타데이터로 엄격하게 관리되고 있어, 임의로 레지스터 매핑을 변경하거나 특정 뱅크의 레지스터만 집중적으로 사용하도록 개수를 수정할 경우 바이너리 구조가 손상되어 에러가 발생하는 한계에 부딪혔다.

```
// ---------------------- .text._Z18no_conflict_kernelPf   --------------------------
        .section        .text._Z18no_conflict_kernelPf,"ax",@progbits
        .__section_name         0xda    // offset in .shstrtab
        .__section_type         SHT_PROGBITS
        .__section_flags        0x6
        .__section_addr         0x0
        .__section_offset       0xd80   // maybe updated by assembler
        .__section_size         0x300   // maybe updated by assembler
        .__section_link         3
        .__section_info         0x12000009
        .__section_entsize      0
        .align                128       // equivalent to set sh_addralign
        .sectioninfo    @"SHI_REGISTERS=40"
        .align  128
          .global         _Z18no_conflict_kernelPf
          .type           _Z18no_conflict_kernelPf,@function
          .size           _Z18no_conflict_kernelPf,(.L_x_5 - _Z18no_conflict_kernelPf)
          .other          _Z18no_conflict_kernelPf,@"STO_CUDA_ENTRY STV_DEFAULT"
```

SHI_REGISTERS로 thread당 할당하는 register를 명시한다.

3. 레지스터 강제 확장 및 SASS 직접 수정

최종적으로, thread당 레지스터 수를 강제로 늘려 충분한 레지스터를 확보한 후, assembly 파일에서 임의의 FFMA 명령어를 넣어, 실행파일을 만들었다. 이를 통해 컴파일러의 최적화를 우회하고 성공적으로 bank conflict를 유발할 수 있었다.

```
  FFMA R28, R21, R22, R28 ;
  FFMA R29, R22, R23, R29 ;
  ISETP.NE.AND P0, PT, R11,
  FFMA R30, R23, R24, R30 ;
  FFMA R31, R20, R21, R31 ;
FFMA R28, R21, R22, R28 ;
FFMA R29, R22, R23, R29 ;
FFMA R30, R23, R24, R30 ;
FFMA R31, R20, R21, R31 ;

FFMA R28, R21, R22, R28 ;
FFMA R29, R22, R23, R29 ;
FFMA R30, R23, R24, R30 ;
```

```
  FFMA R28, R20, R24, R28 ;
  FFMA R29, R21, R25, R29 ;
  ISETP.NE.AND P0, PT, R11, 0xf4240
  FFMA R30, R22, R26, R30 ;
  FFMA R31, R23, R27, R31 ;
FFMA R28, R20, R24, R28 ;
FFMA R29, R21, R25, R29 ;
FFMA R30, R22, R26, R30 ;
FFMA R31, R23, R27, R31 ;


FFMA R28, R20, R24, R28 ;
FFMA R29, R21, R25, R29 ;
FFMA R30, R22, R26, R30 ;
FFMA R31, R23, R27, R31 ;


FFMA R28, R20, R24, R28 ;
FFMA R29, R21, R25, R29 ;
FFMA R30, R22, R26, R30 ;
FFMA R31, R23, R27, R31 ;
```

왼쪽은 bank conflict가 발생하지 않는 코드이고, 오른쪽은 발생하는 코드이다.

실험 결과, bank conflict가 발생할 때는 56.534 ms가, 발생하지 않을 때는 41.687 ms가 걸렸다.

1.4 tensor core

기존의 CUDA Core를 사용한 연산이 1개의 스레드가 1개의 요소를 독립적으로 계산하는 방식이었다면, tensor core는 하나의 warp가 16x16 행렬의 곱셈을 단일 hardware 명령어로 처리하는 것이다. 이러한 tensor core의 연산은 WMMA API를 통해 이루어지며, 크게 3가지 과정으로 실행된다.

1. 레지스터로의 데이터 적재 (wmma::load_matrix_sync)

연산을 수행하기 전, global memory나 shared memory에 위치한 행렬 데이터를 tensor core 레지스터로 적재한다.

2. 하드웨어 가속 행렬 곱셈 (wmma::mma_sync)

레지스터에 적재된 행렬을 바탕으로 곱셈을 진행한다. Tensor core는 단 한 번의 호출로 16x16행렬끼리의 곱을 한 번에 한다.

3. 연산 결과 저장 (wmma::store_matrix_sync)

연산 결과를 global memory나 shared memory에 저장한다.

텐서 코어 연산에서 성능을 극대화하는 또 다른 핵심은 Mixed Precision의 활용이다. WMMA 연산 시, 입력 행렬 A와 B는 메모리 대역폭을 절약하고 연산 속도를 극대화하기 위해 16비트 반정밀도 부동소수점으로 입력받는다. 그러나 이들을 곱하고 더하는 과정에서 발생할 수 있는 Overflow를 방지하기 위해, Accumulator는 32비트 단정밀도를 사용하도록 설계되었다.

```cpp
__global__ void naiveGemm(half *A, half *B, float *C, int M, int N, int K) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < M && col < N) {
        float sum = 0.0f;
        for (int i = 0; i < K; ++i) {
            sum += __half2float(A[row * K + i]) * __half2float(B[i * N + col]);
        }
        C[row * N + col] = sum;
    }
}

__global__ void wmmaGemm(half *A, half *B, float *C, int M, int N, int K) {
    int warpM = (blockIdx.y * blockDim.y + threadIdx.y) / 32;
    int warpN = (blockIdx.x * blockDim.x + threadIdx.x);

    int row = warpM * 16;
    int col = warpN * 16;

    if (row < M && col < N) {
        wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a_frag;
        wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::col_major> b_frag;
        wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

        wmma::fill_fragment(c_frag, 0.0f);

        for (int i = 0; i < K; i+= 16) {
            wmma::load_matrix_sync(a_frag, A + row * K + i, K);
            wmma::load_matrix_sync(b_frag, B + i * N + col, N);

            wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
        }

        wmma::store_matrix_sync(C + row * N + col, c_frag, N, wmma::mem_row_major);
    }
}
```

4096x4096 행렬곱을 위와 같은 코드로 실험한 결과, Tensor core를 사용한 것(0.69ms)이 사용하지 않은 것(81.429ms)보다 약 116배 더 빨랐다. 이는 tensor core에 맞게 데이터를 배치하면 성능을 극대화할 수 있음을 보여준다.

2. 행렬 연산 최적화 기법

2.1 Dense matrix

행렬 곱셈 연산 ($C = A \times B$) 연산에서 행렬의 크기(N)이 정해지면 총 연산 횟수는 $O(N^3)$으로 고정된다. 따라서 연산 자체의 속도를 높이는 것뿐만 아니라 memory bound을 해결하는 것도 중요해진다.

이를 위해서는 global memory에 대한 총 접근 횟수를 낮추고, 한 번 가져온 메모리당 수행하는 연산의 비율을 극대화해 overhead를 낮춰야 한다. 이를 달성하기 위해 tiling, packing 이라는 두 가지 핵심 기법을 단계적으로 적용하였다.

1. Tiling: 메모리 접근 overhead 감소

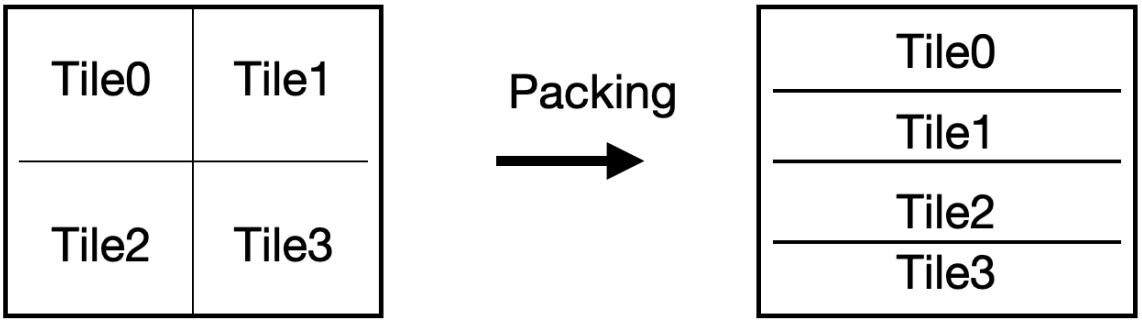
Global memory에서 thread가 매번 데이터를 직접 읽어와 연산하는 구조는 느린 메모리 접근을 유발한다. 이를 해결하기 위해 GPU 내부의 shared memory에 데이터를 한 번 가져온 뒤 연산을 하는 Tiling 기법을 적용하였다.

행렬을 $T \times T$ 크기의 Tile로 나누어 shared memory에 한 번 적재하면, $2T^2$개의 데이터를 가져와 $2T^3$번의 연산을 수행할 수 있다. 즉, 한 번 가져온 데이터를 T번 사용하게 되며, 이를 통해 글로벌 메모리 접근 횟수를 T배 낮추어 메모리 지연에 대한 overhead를 낮출 수 있다.

2. Packing: memory coalescing 극대화

Tiling을 위해 global memory에서 tile 크기만큼 데이터를 가져올 때, 원본 행렬이 일반적인 형태로 길게 나열되어 있으면 메모리 주소에 대한 비연속 접근이 발생한다. 이는 TLB Miss를 유발하는 치명적인 병목이다. 이를 해결하기 위해 타일을 일자로 피는 packing을 사용하였다.

이렇게 하게 될 경우, 하나의 warp가 tile 데이터에 접근할 때 32개의 thread가 정확히 연속된 32개의 메모리 주소를 동시에 요청하게 된다. GPU는 연속된 메모리 접근을 하나의 트랜잭션으로 묶어서 처리하므로(memory coalescing), 성능을 향상시킬 수 있다.



구현 코드와 결과는 다음과 같다.

```
#define TILE_SIZE 32
__global__ void tiled_gemm_kernel(float* A, float* B, float* C, int n) {
    __shared__ float tile_A[TILE_SIZE][TILE_SIZE];
    __shared__ float tile_B[TILE_SIZE][TILE_SIZE];
    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int row = by * TILE_SIZE + ty;
    int col = bx * TILE_SIZE + tx;
    float val = 0.0f;

    for (int k = 0; k < n; k += TILE_SIZE) {
        // 크기가 넘어갈 때에 대한 예외처리
        if (row < n && (k + tx) < n) tile_A[ty][tx] = A[row * n + (k + tx)];
        else tile_A[ty][tx] = 0.0f;
        if (col < n && (k + ty) < n) tile_B[ty][tx] = B[(k + ty) * n + col];
        else tile_B[ty][tx] = 0.0f;
        __syncthreads();

        for (int i = 0; i < TILE_SIZE; i++) {
            val += tile_A[ty][i] * tile_B[i][tx];
        }
        __syncthreads();
    }
    if (row < n && col < n) C[row * n + col] = val;
}
```

Tiling 구현한 코드. __synthreads() 전의 코드는 데이터를 shared memory로 가져온다.

```
__global__ void pack_matrix_kernel(float* src, float* packed_dst, int n) {
    int row = blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;

    if (row < n && col < n) {
        int src_idx = row * n + col;
        int grid_width = n / TILE_SIZE;
        int tile_idx = blockIdx.y * grid_width + blockIdx.x;
        int in_tile_idx = threadIdx.y * TILE_SIZE + threadIdx.x;
        int dst_idx = tile_idx * (TILE_SIZE * TILE_SIZE) + in_tile_idx;

        packed_dst[dst_idx] = src[src_idx];
    }
}
```

Packing 구현한 코드

4096x4096 행렬 곱을 실험해본 결과, Tiled를 적용한 코드(41.79ms)가 적용하지 않은 코드(73.88ms)보다 1.7배 빨랐다. Packing과 Tiled를 적용한 코드(40.7ms)는 tiled만 적용한 코드(41.79ms)와 거의 비슷했다. 이유는 Tiling할 때 이미 global memory에 대한 memory coalescing이 적용되었기 때문이다.

추가적으로, tensor core를 적용해본 결과, 13.68ms가 나와 기존의 코드보다 2.97배 빨랐다. 하지만 단순히 tensor core만 쓴 것(0.69ms)보다 19.8배 느렸다. 이는 packing과 tiling하는 과정에서 global memory에서 shared memory로 데이터를 가져오는 overhead가 전체 연산 시간을 지연시키는 현상이 발생했기 때문이다. 이러한 이유는 tensor core 연산이 너무 빨라서 메모리 접근에 대한 overhead를 상쇄하지 못했기 때문이다.

2.2 sparse matrix

Sparse matrix 곱셈에서는 sparsity 패턴과 이를 메모리에 올리는 format에 의해 성능이 좌우된다. 이 리포트에서는 pattern과 상관없이 메모리 공간 효율성이 높고 memory coalescing에 유리한 CRS (Compressed Row Storage) 포맷을 기본 구조로 채택하였다.

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\begin{aligned} \mathtt{ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ \mathtt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \mathtt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

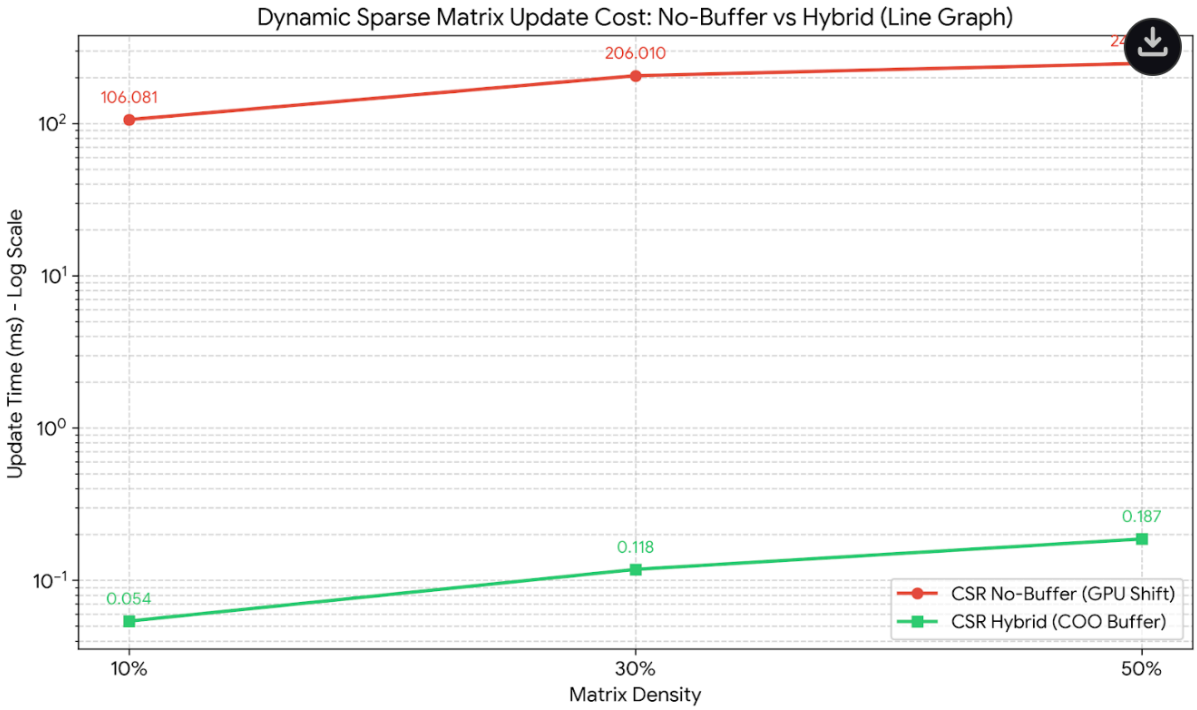
indices는 열의 위치를, ptr은 행의 위치에 대한 정보를 나타낸다.(ptr[i]부터 ptr[i+1]까지는 i번째 행에 있음을 나타낸다.) Ptr를 저렇게 표현하면 행렬의 Non zero 요소 개수와 상관없이 row+1개만큼의 공간만 필요해 공간을 절약할 수 있다.

이러한 CRS 포맷은 행렬이 바뀌지 않는 연산에서는 최적화되어 있으나, 새로운 0이 아닌 요소가 추가되거나 기존 값이 빈번하게 업데이트되는 동적 환경에서는 치명적인 성능 저하를 유

발한다. 이유는 해당 행 이후의 모든 indices와 ptr 값, data 값을 한 칸씩 밀어내야 하므로, 단 한 번의 업데이트에 최악의 경우 O(NNZ)의 막대한 데이터 이동 오버헤드가 발생한다. (NNZ: Non Zero 요소의 총개수)

이러한 문제를 해결하기 위해, 본 리포트에서는 COO 포맷을 buffer로 활용하는 구조를 제시한다. 행렬에 새로운 데이터가 추가되거나 변경될 때, CRS를 바꾸지 말고, COO 버퍼에 저장한다. 그리고 행렬곱을 구할 때, CRS와 COO에 각각 적용해서 구한 뒤, 결과를 합한다. 또한, Lazy update 전략을 이용해 나중에 버퍼에 쌓인 데이터가 일정 크기를 넘어갈 경우, 해당 데이터를 CRS에 반영함으로써 업데이트에 대한 비용을 감소시킨다.

다음은 4096x4096 sparse matrix의 density에 따른 실험 결과이다.



buffer를 사용하는 것이 그렇지 않은 것보다 약 1500배정도 빨랐다. 이유는 buffer를 사용하지 않을 경우, CRS를 update하기 위해 global memory에 순차적으로 접근해 데이터를 바꿔야하기 때문이다.

2.3 Cryptographic 가속: Montgomery Multiplication

ZKML에서 증명의 생성 및 검증을 위한 모든 산술 연산은 거대한 소수 p를 module로 하는 유한체위에서 수행된다. 따라서 신경망 모델의 가중치와 활성화 값은 모두 이 유한체 내의 원소로 매핑되어 행렬 곱셈 연산을 거치게 된다. 문제는 유한체 연산의 핵심인 모듈러 곱셈

($A \times B \pmod{p}$)에 있다. 두 거대 정수를 곱한 뒤 소수 p로 나누어 나머지를 구하는 전통적인 나눗셈 기반의 모듈러 연산(% 연산자)은 하드웨어와 소프트웨어 레벨에서 극심한 클럭 사이클을 소모한다. 수백만에서 수십억 번의 곱셈이 동반되는 ZKML 환경에서, 이 나눗셈 연산은 증명 생성 시간을 지연시키는 가장 치명적인 병목으로 작용한다. 이러한 무거운 나눗셈 오버헤드를 근본적으로 제거하기 위해, 본 구조에서는 Montgomery Multiplication 알고리즘을 도입하여 소수 체 연산 파이프라인을 최적화하였다. 몽고메리 곱셈은 값들을 특수한 Montgomery Domain으로 미리 변환하여 연산하는 수학적 기법이다. 피연산자를 R이라는 상수와 결합하여 $A \cdot R \pmod{p}$형태로 변환한 뒤 연산을 수행한다. 이 도메인 내부에서 곱셈을 수행할 경우, 값비싼 나눗셈(% p)을 수행할 필요가 완전히 사라진다. 대신 컴퓨터 하드웨어가 가장 빠르게 처리할 수 있는 단순 곱셈, 덧셈, 그리고 비트 시프트 및 논리곱연산만으로 모듈러 축소 과정을 완벽하게 대체할 수 있다.

thread당 1000번 연산을 진행하는 실험을 한 결과, montgomery 연산을 사용한 경우(37.64ms)가 그렇지 않은 경우(251.50ms)보다 약 6배 빨랐다. 이는 zkml에서는 %대신 shift 연산을 사용하는 것이 필수적임을 보인다.

Reference

[1]: Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking

[2]: Efficient Sparse Matrix-Vector Multiplication on CUDA

[3]: Anatomy of High-Performance Matrix Multiplication