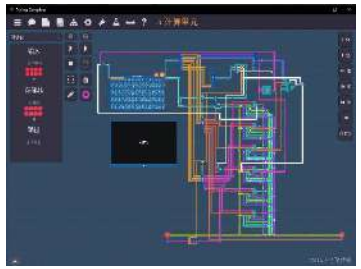
HGEMM

赖海斌

12211612

HPC Learning Path



- CSAPP
- Computer Arch: A Quantitative Approach
- Parallel Programming
- OSTEP
- Computer Network
- Compiler
- ----
- CMU 445 Database
- MIT 6.824 Distributed System

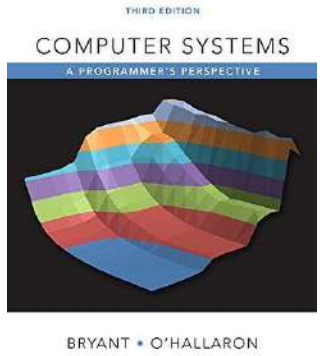
- Linux/git/C++/Python/slurm
- MPI/OpenMP
- CUDA
- Pytorch/Triton

- asm, ptx, Tensor Core, NPU
- Intel VTune / Nvidia Profiler / TAU
- gdb/perf/ebpf

- K8s, docker, MIG
- RDMA, All-to-All, dragonfly, CXL

- MoE, Diffusion
- Distributed Training, MegatronLM
- SFT, LoRA, RAG
- Inference, KVCache

- Simulator
- Paper



GEMM Learning

Content

- 1. What is HGEMM
  - GEMM -> SGEMM -> HGEMM (Lab1)
  - Parallel SGEMM (Lab2)
- 2. CUDA
  - CUDA 101 (Lab3)
  - Tensor Core (Lab4)
- 3. Profiler: Nsight
  - Nsys
  - NCU (Lab5)

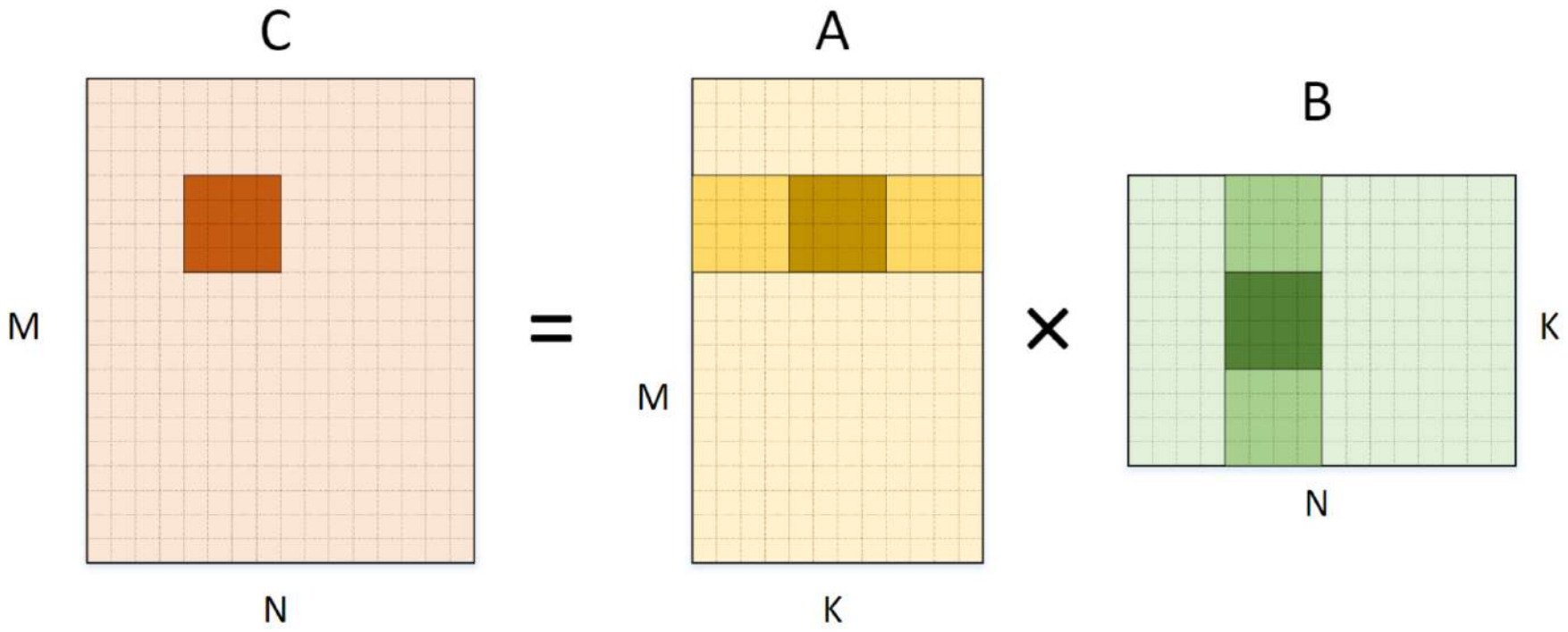
Level

- Beginner
- Medium
- Hard

GEMM

- General Matrix Multiplication

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$



GEMM's history: BLAS

Jack Dongarra

Cedar Computer 1985

BLAS level 1 (1979) *vector-vector operations*

$$\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$$

BLAS level 2 (1988) *matrix-vector operations*

$$\boldsymbol{y} \leftarrow \alpha \boldsymbol{A}\boldsymbol{x} + \beta \boldsymbol{y}$$

GEMV

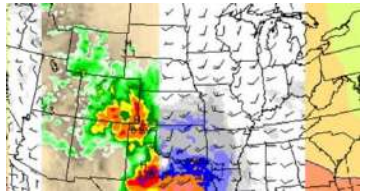
BLAS level 3 (1990) *matrix-matrix operations*

$$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B} + \beta \boldsymbol{C},$$

GEMM

Basic Linear Algebra Subprograms (BLAS) is a specification that prescribes a set of low-level routines for performing **common linear algebra operations** such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication.

MATLAB®



LLaMA C++

NumPy

PyTorch

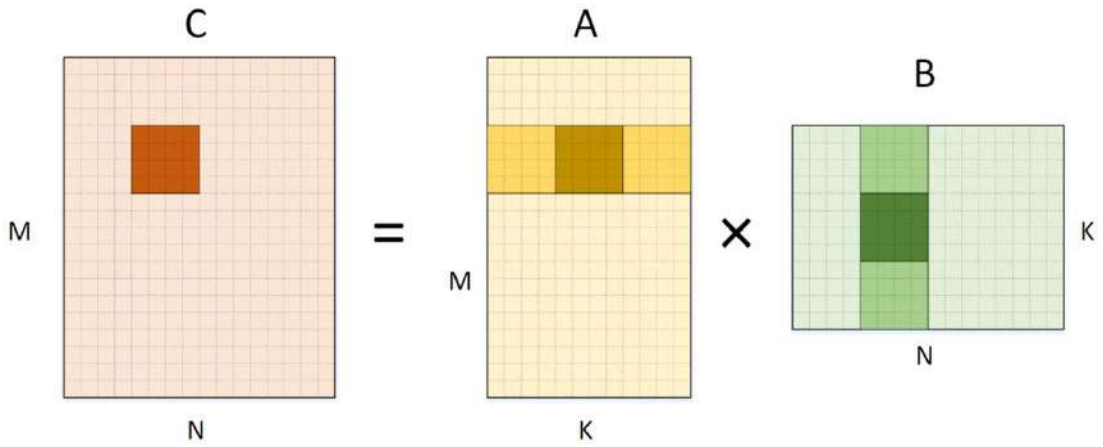
intel Intel MKL

GEMM's API

Level 3 BLAS: matrix-matrix, $O(n^3)$ operations

types	name	(options	size	arguments	)	description	equation	flops	data
s, d, c, z	gemm	(	transA, transB,	m, n, k, alpha, A, ldA, B, ldB, beta, C, ldC)		general matrix-matrix multiply	$C = \alpha A^* B^* + \beta C$	$2mnk$	$mk + nk + mn$

Parameter	Meaning
A, B	Input matrices
C	Output matrix (may contain initial values)
M, N, K	Matrix dimensions: A[M×K], B[K×N], C[M×N]
α, β	Scalars: C = αAB + βC
lda, ldb, ldc	Leading dimensions (row stride for 2D arrays)



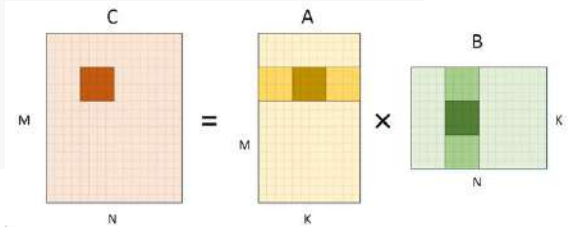
SGEMM

- **Single precision** General Matrix Multiplication

```java
public static void sgemm(int M, int N, int K,
                         float alpha, float[][] A,
                         float[][] B, float beta,
                         float[][] C) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            float sum = 0.0f;
            for (int k = 0; k < K; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = alpha * sum + beta * C[i][j];
        }
    }
}
```

$$(AB)_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj}.$$

```java
public static void main(String[] args) {
    int M = 2, K = 3, N = 2;
    float alpha = 1.0f;
    float beta = 0.0f;

    float[][] A = {
        {1f, 2f, 3f},
        {4f, 5f, 6f}
    };

    float[][] B = {
        {7f, 8f},
        {9f, 10f},
        {11f, 12f}
    };

    float[][] C = new float[M][N]; // Zero-initialized

    sgemm(M, N, K, alpha, A, B, beta, C);

    // Print result
    System.out.println("Result C =");
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            System.out.printf("%8.2f", C[i][j]);
        }
        System.out.println();
    }
}
```



Double precision General Matrix Multiplication

```java
/**
 * Compute C = alpha * A * B + beta * C
 * A: M x K matrix
 * B: K x N matrix
 * C: M x N matrix
 */
public static void dgemm(int M, int N, int K,
                         double alpha, double[][] A,
                         double[][] B, double beta,
                         double[][] C) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            double sum = 0.0;
            for (int k = 0; k < K; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = alpha * sum + beta * C[i][j];
        }
    }
}
```

Simplified DGEMM

```java
/**
 * Compute C = A * B
 * A: M x K matrix
 * B: K x N matrix
 * C: M x N matrix (output)
 */
public static void dgemm(int M, int N, int K,
                         double[][] A, double[][] B, double[][] C) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0.0;
            for (int k = 0; k < K; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Important Note

- Note: In this competition, you are allowed to use the simplified version of matrix multiplication: $C = A \times B$
- That means you do not need to include the scalar values alpha and beta as in the full GEMM formula ($C = \alpha AB + \beta C$). Just compute the product of matrix A and matrix B directly and store the result in matrix C.
- Whether M, N, and K are passed as function parameters is not required

What's the difference between float and double?

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

What's the difference between float and double?

IEEE Floating-Point Format

$$\pm 1.xxxxxxx_2 \times 2^y$$

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent (y+Bias)	Fraction (xxxx)

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})}$$

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand(规约化有效数字)
  - 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden one)
  - Significand is Fraction with the "1." restored
- Exponent: excess representation: actual exponent (y) + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

Example 1: Decimal to FP

- Represent –0.75
  - $-0.75_{ten} = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 1_01111110_1000…00 (23bits)
- Double: 1_01111111110_1000…00 (52bits)

Exercise: Represent 24.5_{ten} in single-precision FP
- Sign bit = ?
- Fraction = ?
- Exponent = $?_{ten}$

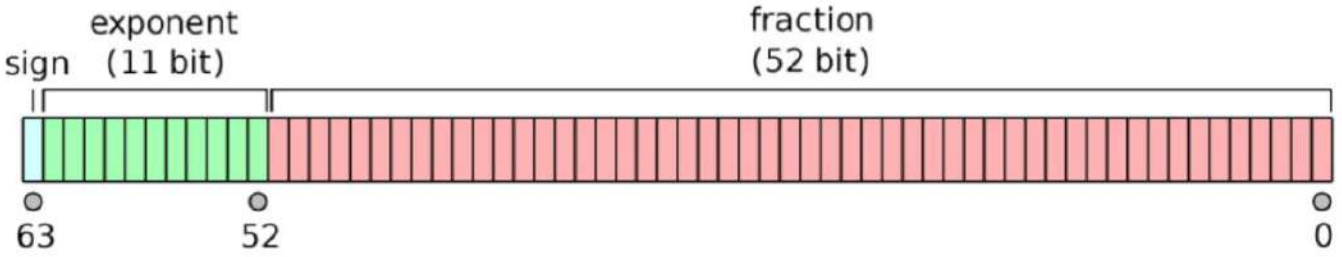
Recall:

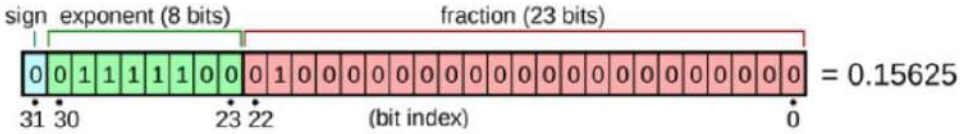
True exponent ⇄ Exponent in register (+127 / -127)

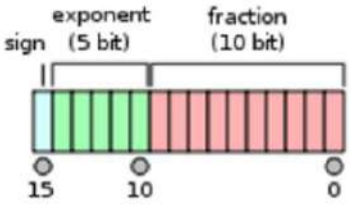
Lab01 Float Converter

FP64, FP32, FP16

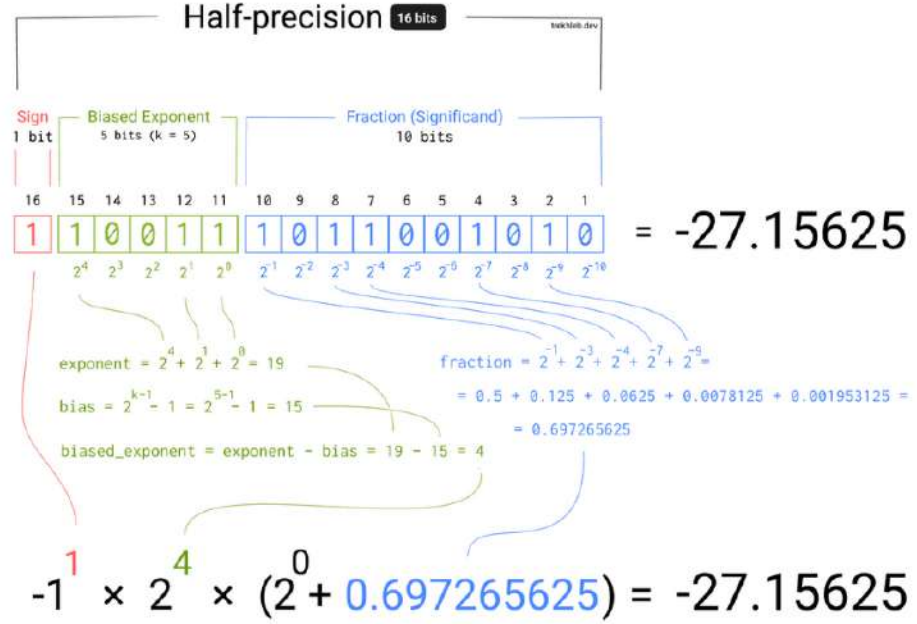
Format of Floating points IEEE754







半精度浮点数 16 bit 浮点数计算过程示例：



FP16		
Sign	**Exponent**	**Mantissa/Significand**
+	2^4	1.53125
0	19	544
☐	☑ ☐ ☐ ☑ ☑	☑ ☐ ☐ ☐ ☑ ☐ ☐ ☐ ☐ ☐

What's HGEMM

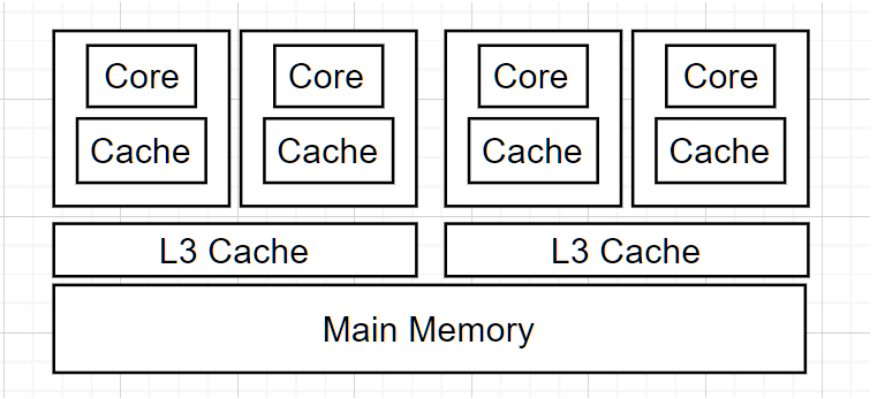
```java
public static void hgemm(short[][] A, short[][] B, short[][] C, int M, int N, int K) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            float sum = 0.0f;
            for (int k = 0; k < K; k++) {
                float a = toFloat(A[i][k]);
                float b = toFloat(B[k][j]);
                sum += a * b;
            }
            C[i][j] = toFP16(sum);
        }
    }
}
```

```cpp
#include "half.hpp"

//  g++ -O2 -std=c++11 origin.cpp -o origin_hgemm

using half_float::half;

void hgemm(int M, int N, int K,
           const half* A, const half* B, half* C) {
    for (int i = 0; i < M; ++i){
        for (int j = 0; j < N; ++j) {
            float sum = 0.0f;
            for (int k = 0; k < K; ++k)
                sum += static_cast<float>(A[i * K + k])
                     * static_cast<float>(B[k * N + j]);
            C[i * N + j] = half(sum);
        }
    }
}
```

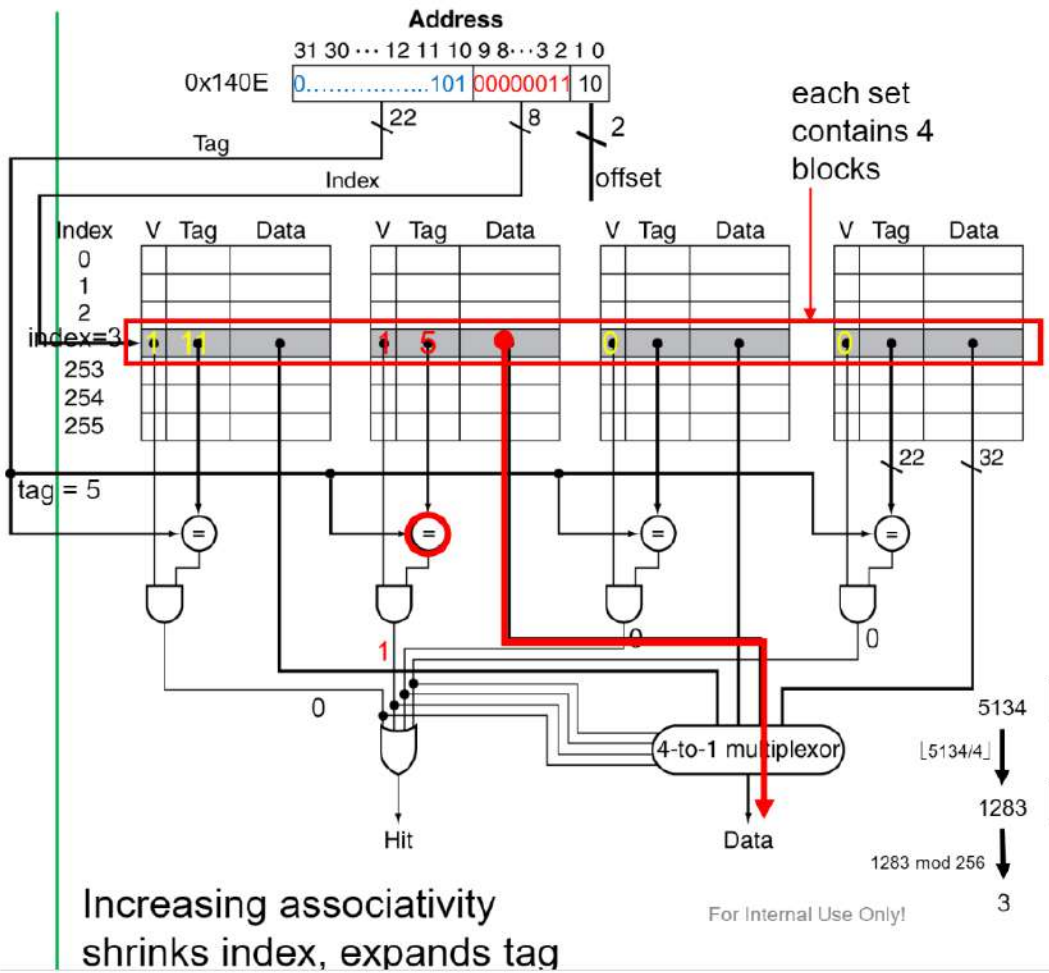
[half: Half-precision floating-point library]

Parallelizing HGEMM using OpenMP 🌶️

```cpp
#include <omp.h>

//  g++ -O2 -std=c++11 origin_omp.cpp -o openmp_hgemm -fopenmp

using half_float::half;

void hgemm(int M, int N, int K,
           const half* A, const half* B, half* C) {
            // using openmp
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < M; ++i){
        for (int j = 0; j < N; ++j) {
            float sum = 0.0f;
            for (int k = 0; k < K; ++k)
                sum += static_cast<float>(A[i * K + k])
                     * static_cast<float>(B[k * N + j]);
            C[i * N + j] = half(sum);
        }
    }
}
```



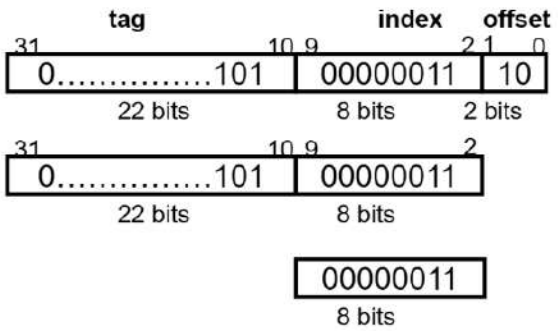
Set Associative Cache Organization



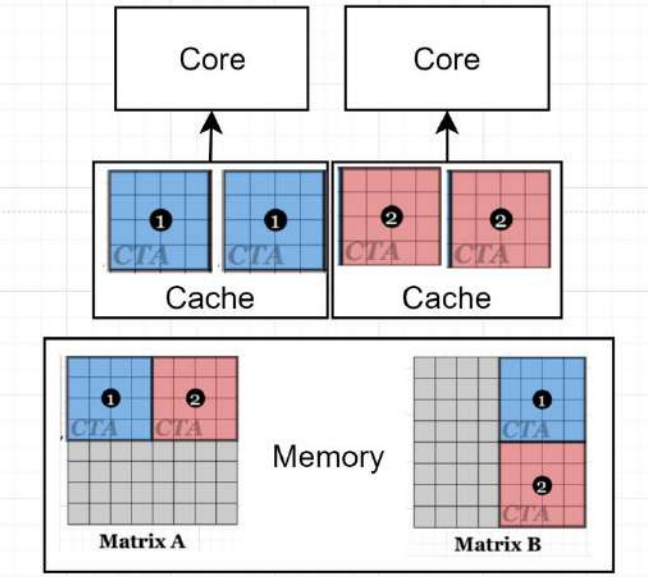
4KB cache, 1word/block
To what **set** number does address 0x140E map?

offset: $\log_2$(4byte) = 2bits
#blocks: 4KB/4B = 1K blocks
#sets: 1K blocks/4 way = 256 sets
index: $\log_2$(256sets) = 8bits

$0x140E_{hex} = 10100000011110_{two}$
tag index offset



B[k][j], B[k][j+1], ..., B[k][j+15]



Find a balance:

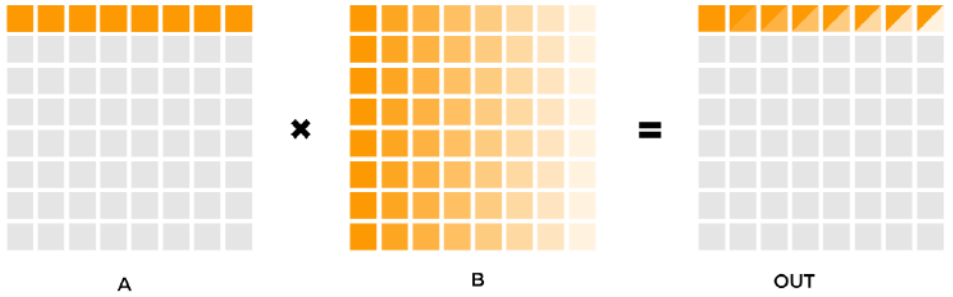
Block size ↓ cache hit ↓ memory ↓
Block size ↑ cache hit ↑ memory ↑

Matrix Tiling technique

```
void hgemm_blocked(const __fp16* A, const __fp16* B, __fp16* C,
                   int M, int N, int K) {
    for (int i0 = 0; i0 < M; i0 += BLOCK_SIZE) {
        for (int j0 = 0; j0 < N; j0 += BLOCK_SIZE) {
            for (int k0 = 0; k0 < K; k0 += BLOCK_SIZE) {
                // 遍历当前 block 区域
                for (int i = i0; i < std::min(i0 + BLOCK_SIZE, M); ++i) {
                    for (int j = j0; j < std::min(j0 + BLOCK_SIZE, N); ++j) {
                        float sum = 0.0f;
                        for (int k = k0; k < std::min(k0 + BLOCK_SIZE, K); ++k) {
                            float a = static_cast<float>(A[i * K + k]);
                            float b = static_cast<float>(B[k * N + j]);
                            sum += a * b;
                        }
                        if (k0 == 0) {
                            C[i * N + j] = static_cast<__fp16>(sum);
                        } else {
                            float c = static_cast<float>(C[i * N + j]);
                            C[i * N + j] = static_cast<__fp16>(c + sum);
                        }
                    }
                }
            }
        }
    }
}
```

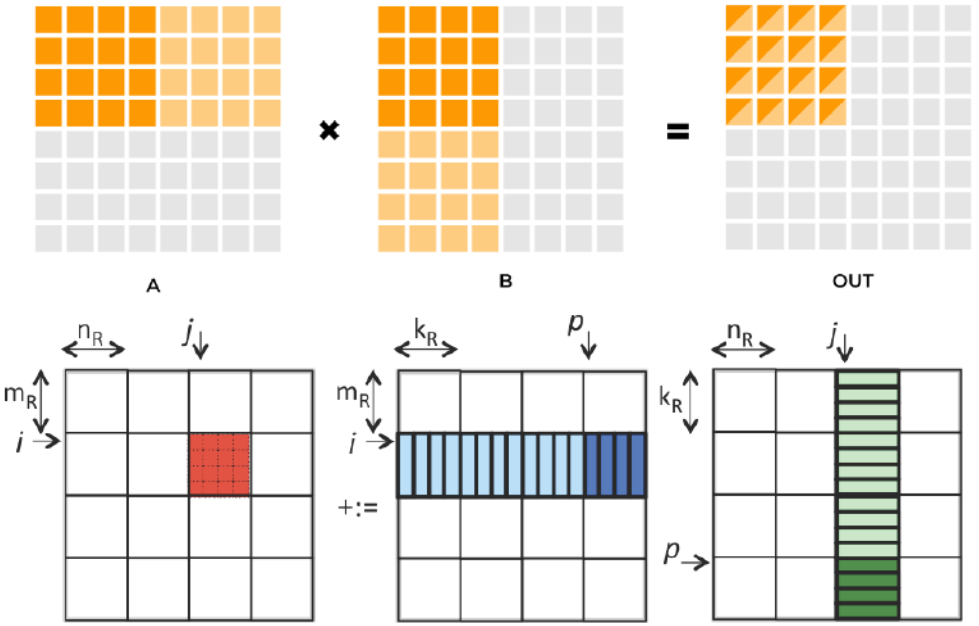
B[kk][jj], B[kk][jj+1], B[kk][jj+2], B[kk][jj+3],
B[kk+1][jj], B[kk+1][jj+1], ...

Original

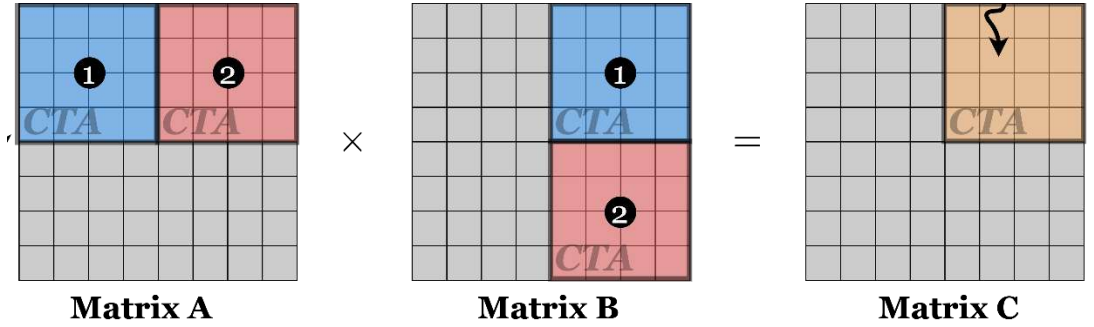


C[i][j] += A[i][k] * B[k][j];

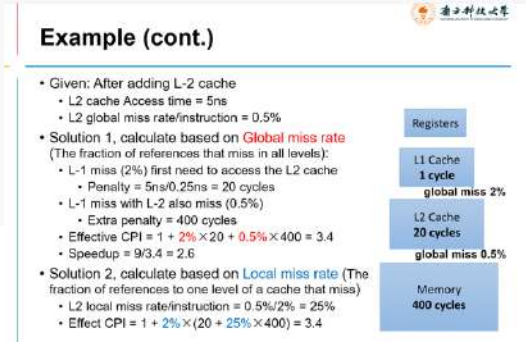
Matrix Tiling

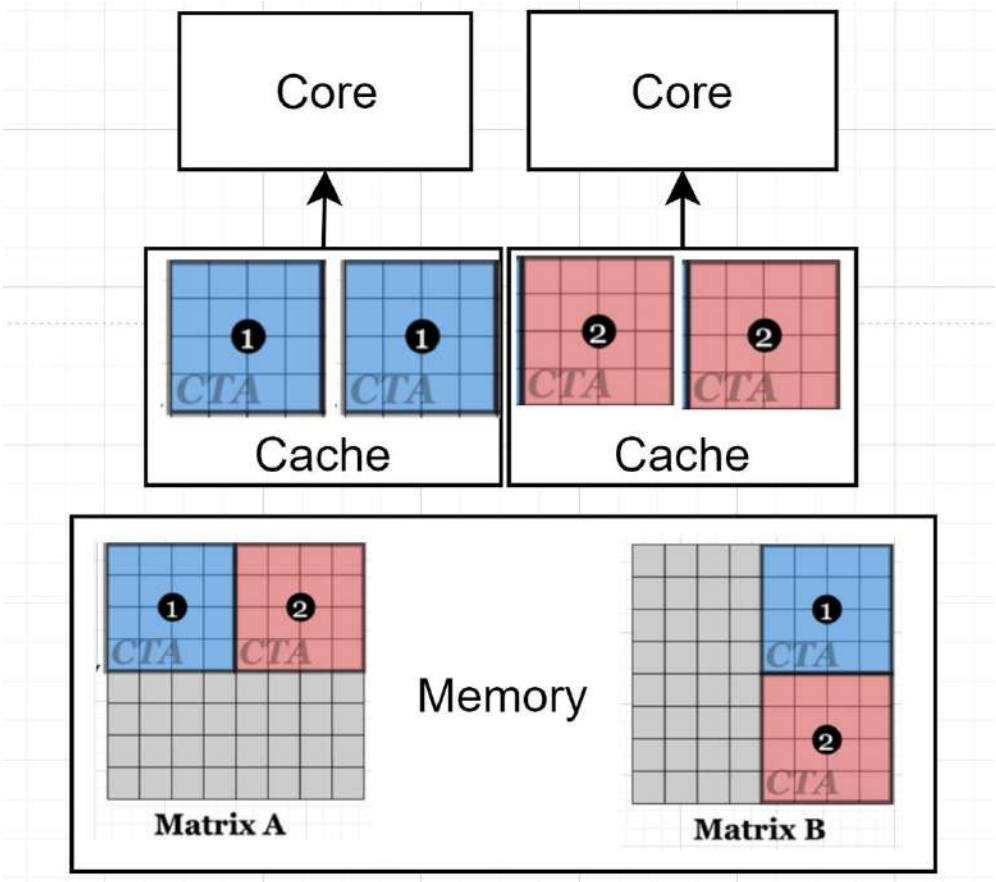


Utilize Memory Access Pattern For Raising Cache hit closer




```
void hgemm_blocked(const __fp16* A, const __fp16* B, __fp16* C,
                   int M, int N, int K) {
    for (int i0 = 0; i0 < M; i0 += BLOCK_SIZE) {
        for (int j0 = 0; j0 < N; j0 += BLOCK_SIZE) {
            for (int k0 = 0; k0 < K; k0 += BLOCK_SIZE) {
                // 遍历当前 block 区域
                for (int i = i0; i < std::min(i0 + BLOCK_SIZE, M); ++i) {
                    for (int j = j0; j < std::min(j0 + BLOCK_SIZE, N); ++j) {
                        float sum = 0.0f;
                        for (int k = k0; k < std::min(k0 + BLOCK_SIZE, K); ++k) {
                            float a = static_cast<float>(A[i * K + k]);
                            float b = static_cast<float>(B[k * N + j]);
                            sum += a * b;
                        }
                        if (k0 == 0) {
                            C[i * N + j] = static_cast<__fp16>(sum);
                        } else {
                            float c = static_cast<float>(C[i * N + j]);
                            C[i * N + j] = static_cast<__fp16>(c + sum);
                        }
                    }
                }
            }
        }
    }
}
```

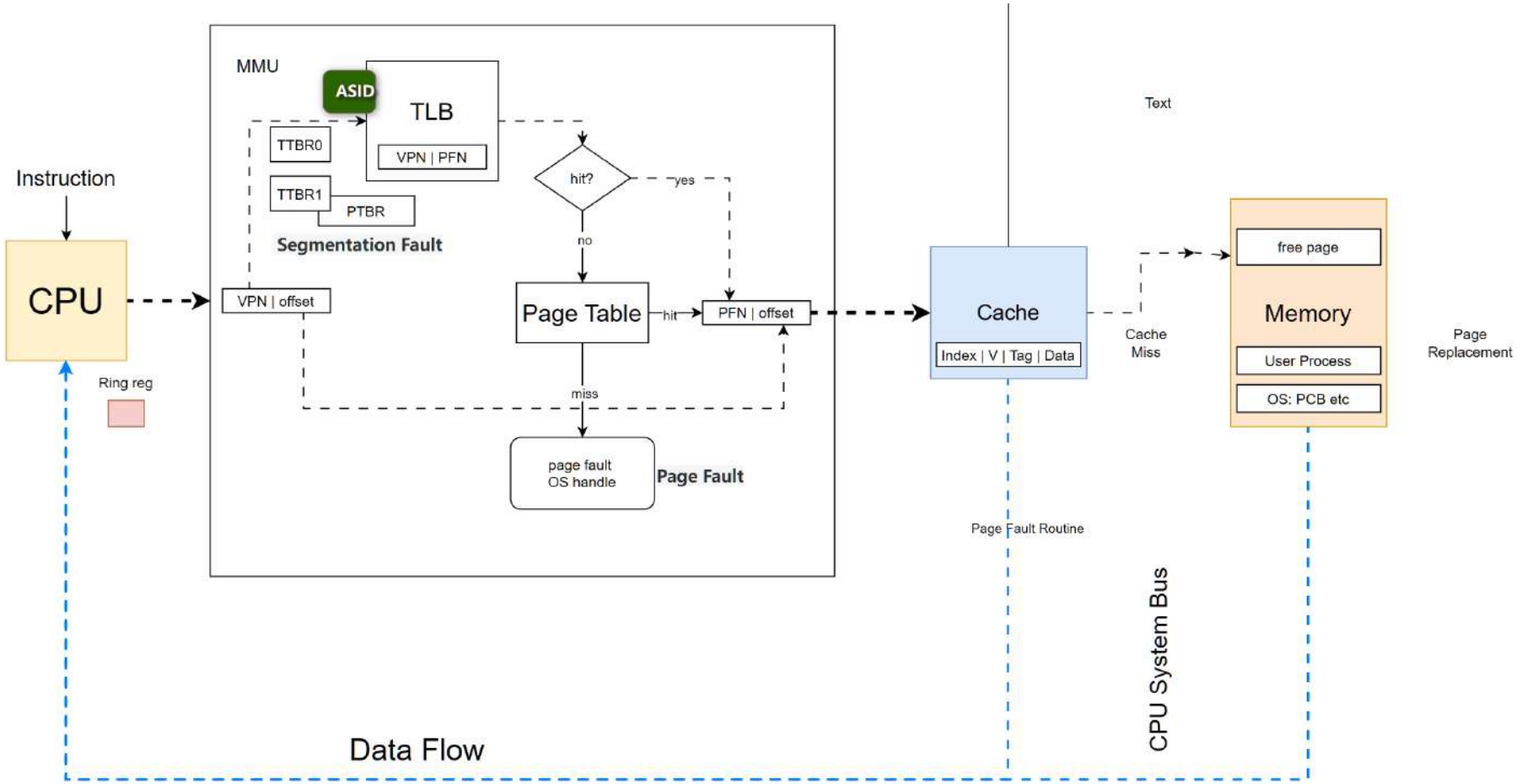


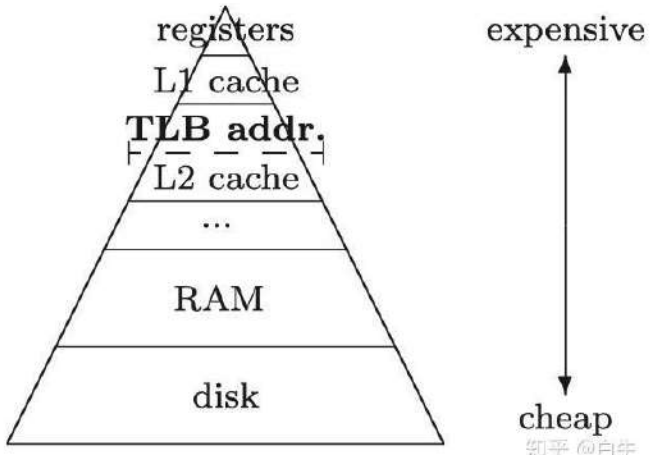


Lab02: perf

perf stat -e cache-misses,cache-references,dTLB-load-misses

TLB sometimes can be a bound!





GEMM caching - 知乎

SOSP24: Powerinfer
Utilize Memory Access Pattern

PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU

Yixin Song, Zeyu Mi, Haotong Xie and Haibo Chen
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University

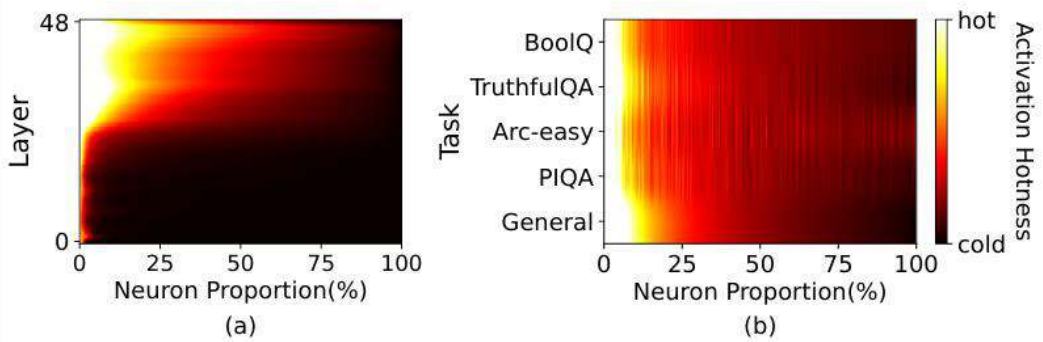


Figure 5. Activation frequency in OPT-30B. (a) The activation frequency of neurons in different layers. The Y-axis represents the layer id. (b) The activation frequency of neurons in different tasks for 30th layer. The Y-axis represents the tasks. The X-axis shows neuron proportion.

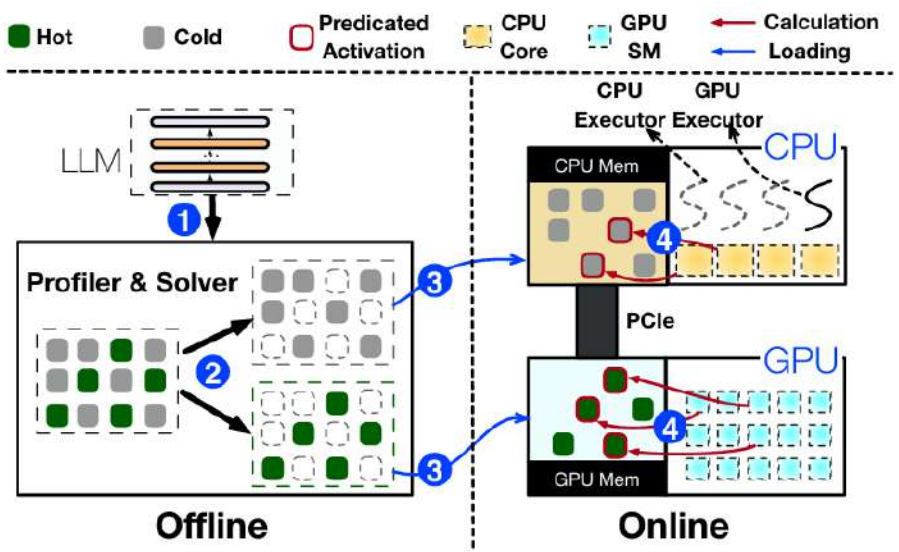


Figure 7. The architecture and inference workflow of PowerInfer.

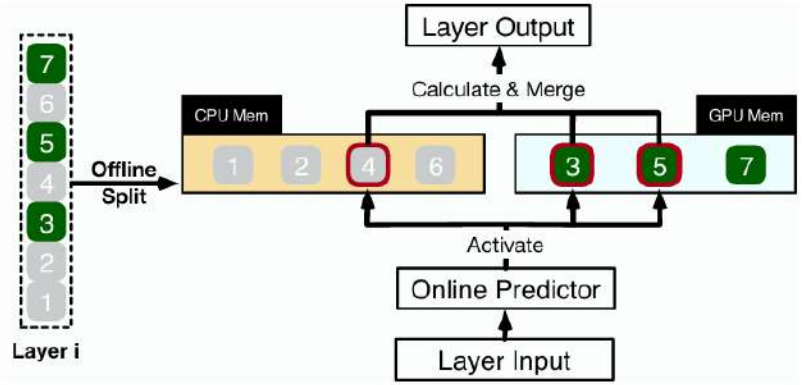
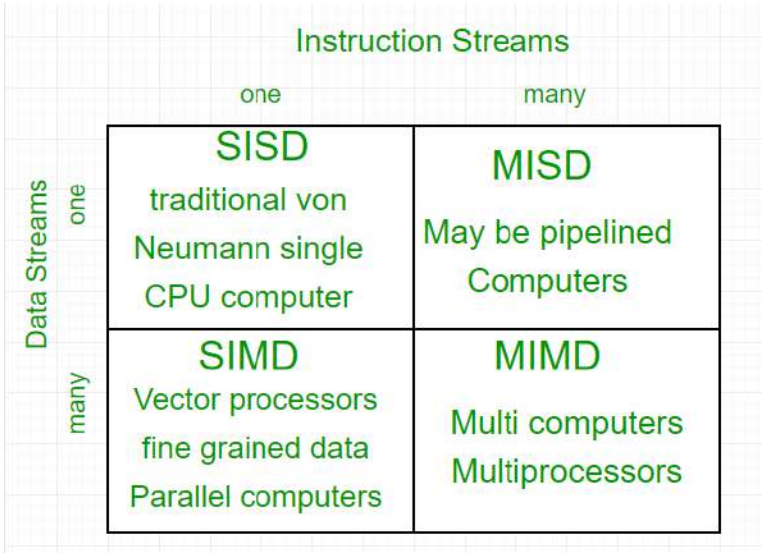
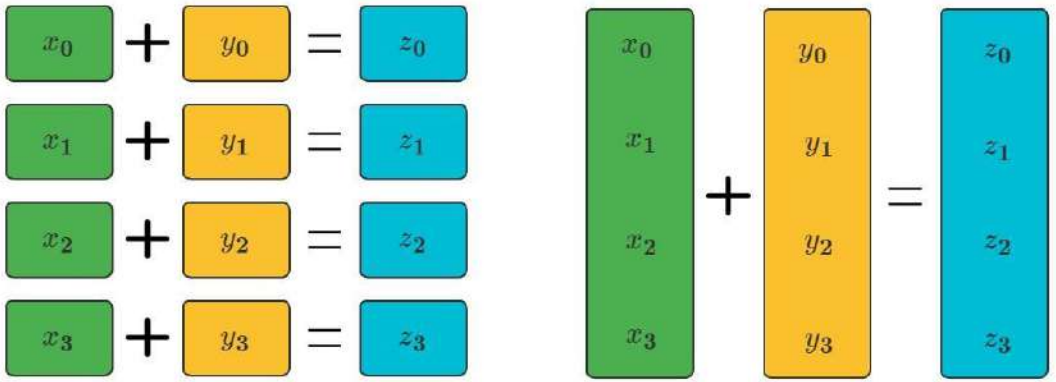


Figure 8. An illustrative example shows how PowerInfer calculates different neurons for one LLM layer.

[PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU]

Using specific hardware: SIMD





SISD SIMD

```cpp
// AVX2内核：计算C[m:m+8, n:n+8] += A[m:m+8, k:k+K] × B[k:k+K, n:n+8]
void hgemm_kernel_avx2(const float16_t* A, const float16_t* B, float* C,
                       int M, int N, int K, int lda, int ldb, int ldc,
                       int m_start, int m_end, int n_start, int n_end) {
    for (int m = m_start; m < m_end; m += 8) { // 每次处理8行
        for (int n = n_start; n < n_end; n += 8) { // 每次处理8列
            // 累加器：FP32，8x8子块
            __m256 acc[8][8];
            for (int i = 0; i < 8; ++i)
                for (int j = 0; j < 8; ++j)
                    acc[i][j] = _mm256_setzero_ps();

            // 沿K维度计算
            for (int k = 0; k < K; ++k) {
                // 加载A的8个FP16值（一行）
                __m128i a_ph[8];
                for (int i = 0; i < 8 && m + i < M; ++i) {
                    a_ph[i] = _mm_loadu_si128(
                        reinterpret_cast<const __m128i*>(&A[(m + i) * lda + k]));
                }

                // 加载B的8个FP16值（一列）
                __m128i b_ph[8];
                for (int j = 0; j < 8 && n + j < N; ++j) {
                    b_ph[j] = _mm_loadu_si128(
                        reinterpret_cast<const __m128i*>(&B[k * ldb + n + j]));
                }

                // 转换为FP32并计算
                for (int i = 0; i < 8 && m + i < M; ++i) {
                    __m256 a_ps = _mm256_cvtph_ps(a_ph[i]); // FP16 -> FP32
                    for (int j = 0; j < 8 && n + j < N; ++j) {
                        __m256 b_ps = _mm256_cvtph_ps(b_ph[j]);
                        acc[i][j] = _mm256_fmadd_ps(a_ps, b_ps, acc[i][j]);
                    }
                }
            }

            // 存储结果到C
            for (int i = 0; i < 8 && m + i < M; ++i) {
                for (int j = 0; j < 8 && n + j < N; ++j) {
                    float* acc_ptr = reinterpret_cast<float*>(&acc[i][j]);
                    for (int l = 0; l < 8; ++l) {
                        C[(m + i) * ldc + (n + j)] += acc_ptr[l];
                    }
                }
            }
        }
    }
}
```

Intel® 64 and IA-32 Architectures Software Developer's Manual

Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4

CHAPTER 10
PROGRAMMING WITH INTEL® STREAMING SIMD EXTENSIONS (INTEL® SSE)

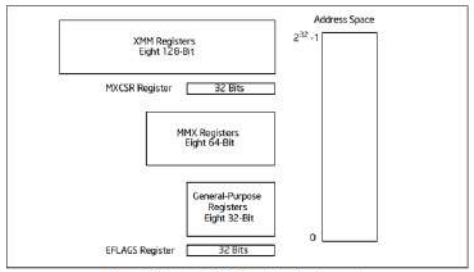


Figure 10-1. Intel® SSE Execution Environment

```cpp
// AVX2内核：计算C[m:m+8, n:n+8] += A[m:m+8, k:k+K] × B[k:k+K, n:n+8]
void hgemm_kernel_avx2(const float16_t* A, const float16_t* B, float* C,
                       int M, int N, int K, int lda, int ldb, int ldc,
                       int m_start, int m_end, int n_start, int n_end) {
    for (int m = m_start; m < m_end; m += 8) { // 每次处理8行
        for (int n = n_start; n < n_end; n += 8) { // 每次处理8列
            // 累加器：FP32，8x8子块
            __m256 acc[8][8];
            for (int i = 0; i < 8; ++i)
                for (int j = 0; j < 8; ++j)
                    acc[i][j] = _mm256_setzero_ps();

            // 沿K维度计算
            for (int k = 0; k < K; ++k) {
                // 加载A的8个FP16值（一行）
                __m128i a_ph[8];
                for (int i = 0; i < 8 && m + i < M; ++i) {
                    a_ph[i] = _mm_loadu_si128(
                        reinterpret_cast<const __m128i*>(&A[(m + i) * lda + k]));
                }

                // 加载B的8个FP16值（一列）
                __m128i b_ph[8];
                for (int j = 0; j < 8 && n + j < N; ++j) {
                    b_ph[j] = _mm_loadu_si128(
                        reinterpret_cast<const __m128i*>(&B[k * ldb + n + j]));
                }

                // 转换为FP32并计算
                for (int i = 0; i < 8 && m + i < M; ++i) {
                    __m256 a_ps = _mm256_cvtph_ps(a_ph[i]); // FP16 -> FP32
                    for (int j = 0; j < 8 && n + j < N; ++j) {
                        __m256 b_ps = _mm256_cvtph_ps(b_ph[j]);
                        acc[i][j] = _mm256_fmadd_ps(a_ps, b_ps, acc[i][j]);
                    }
                }
            }

            // 存储结果到C
            for (int i = 0; i < 8 && m + i < M; ++i) {
                for (int j = 0; j < 8 && n + j < N; ++j) {
                    float* acc_ptr = reinterpret_cast<float*>(&acc[i][j]);
                    for (int l = 0; l < 8; ++l) {
                        C[(m + i) * ldc + (n + j)] += acc_ptr[l];
                    }
                }
            }
        }
    }
}
```

Loop unrolling

CIS 662 – Computer Architecture – Fall 2004 - Class 16 – 11/09/04

Loop Unrolling

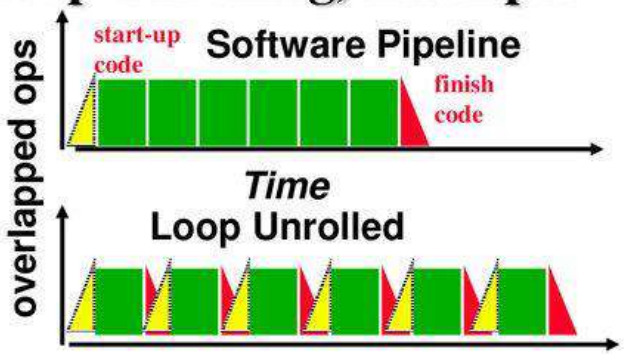
- Step 3: Rename registers

```
Loop:  L.D F0,0(R1)                 Loop:  L.D F0,0(R1)
       ADD.D F4, F0, F2                    ADD.D F4, F0, F2
       S.D F4, 0(R1)                       S.D F4, 0(R1)
       L.D F0,-8(R1)                       L.D F6,-8(R1)
       ADD.D F4, F0, F2                    ADD.D F8, F6, F2
       S.D F4, -8(R1)                      S.D F8, -8(R1)
       L.D F0,-16(R1)                      L.D F10,-16(R1)
       ADD.D F4, F0, F2                    ADD.D F12, F10, F2
       S.D F4, -16(R1)                     S.D F12, -16(R1)
       L.D F0,-24(R1)                      L.D F14,-24(R1)
       ADD.D F4, F0, F2                    ADD.D F16, F14, F2
       S.D F4, -24(R1)                     S.D F16, -24(R1)
       DADDUI R1, R1, #-32                 DADDUI R1, R1, #-32
       BNE R1, R2, Loop                    BNE R1, R2, Loop
```


Software Pipelining (Symbolic Loop Unrolling) Example

Show a software-pipelined version of the code:

```
Loop:  L.D     F0,0(R1)
       ADD.D   F4,F0,F2
       S.D     F4,0(R1)
       DADDUI  R1,R1,#-8
       BNE     R1,R2,LOOP
```



3 times because chain of dependence of length 3 instructions exist in body of original loop i.e. L.D → ADD.D → S.D

Iteration

Before: Unrolled 3 times

```
1   L.D     F0,0(R1)
2   ADD.D   F4,F0,F2
3   S.D     F4,0(R1)
4   L.D     F0,-8(R1)
5   ADD.D   F4,F0,F2
6   S.D     F4,-8(R1)
7   L.D     F0,-16(R1)
8   ADD.D   F4,F0,F2
9   S.D     F4,-16(R1)
10  DADDUI  R1,R1,#-24
11  BNE     R1,R2,LOOP
```

Time

After: Software Pipelined Version

```
        L.D     F0,0(R1)      } start-up code
        ADD.D   F4,F0,F2      }
        L.D     F0,-8(R1)     }
LOOP:
1       S.D     F4,0(R1)      ;Stores M[i]
2       ADD.D   F4,F0,F2      ;Adds to M[i-1]
3       L.D     F0,-16(R1)    ;Loads M[i-2]
4       DADDUI  R1,R1,#-8
5       BNE     R1,R2,LOOP
        S.D     F4, 0(R1)     } finish code
        ADDD    F4,F0,F2      }
        S.D     F4,-8(R1)     }
```

New Loop

2 fewer loop iterations

No Branch delay slot in this example

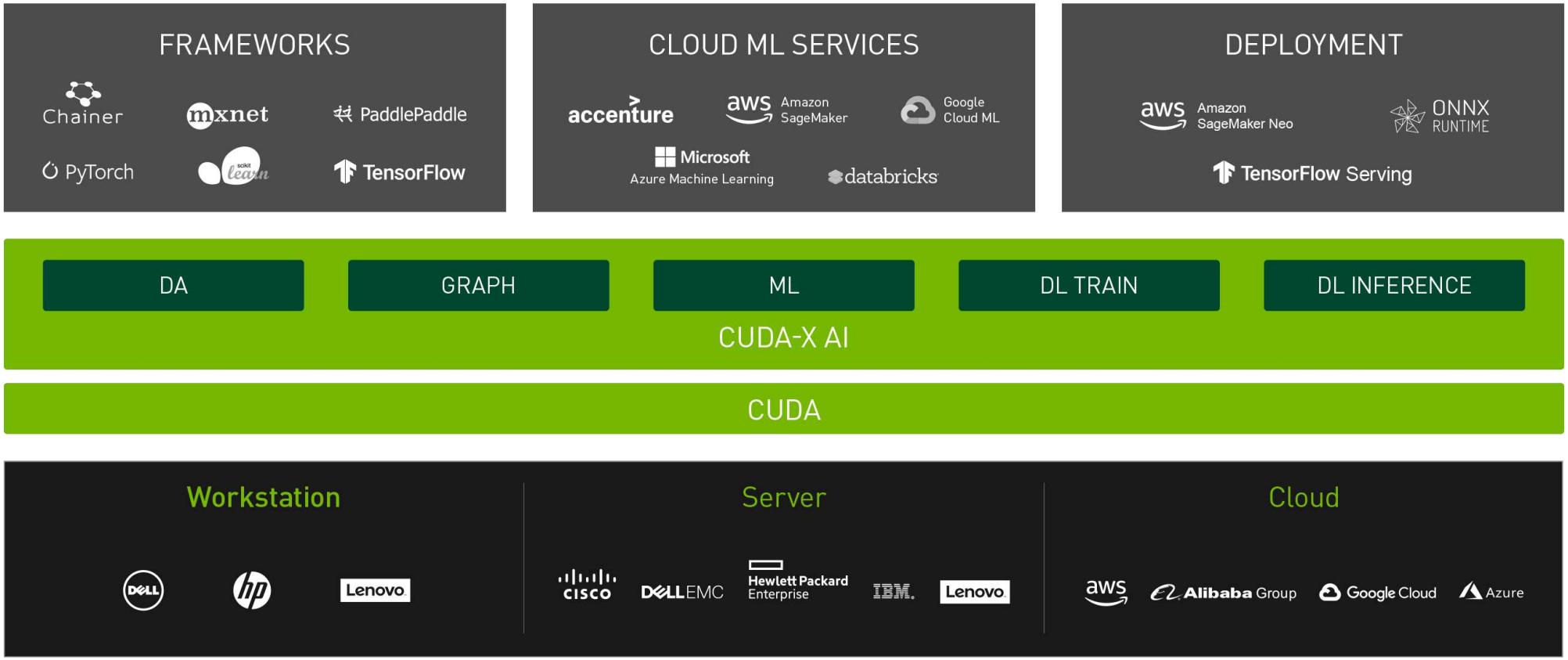
No actual loop unrolling is done (do not rename registers)

CMPE550 - Shaaban

Superscalar Processor
ILP: Instruction-Level Parallelism

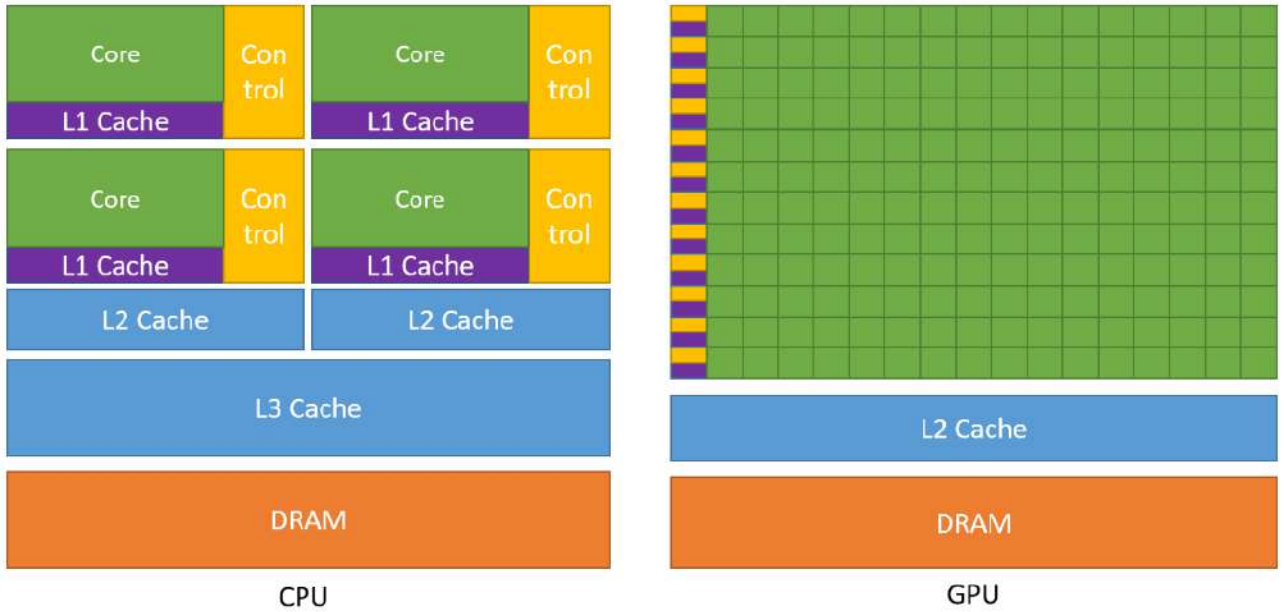
What is CUDA?

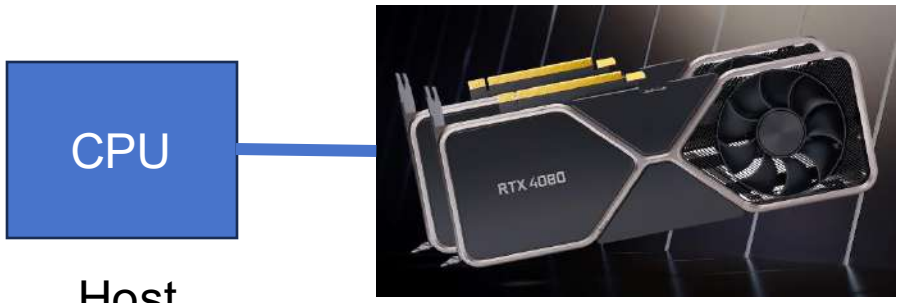
A development environment for creating high performance GPU-accelerated applications.



[CUDA Toolkit Documentation 12.9 Update 1]

GPU





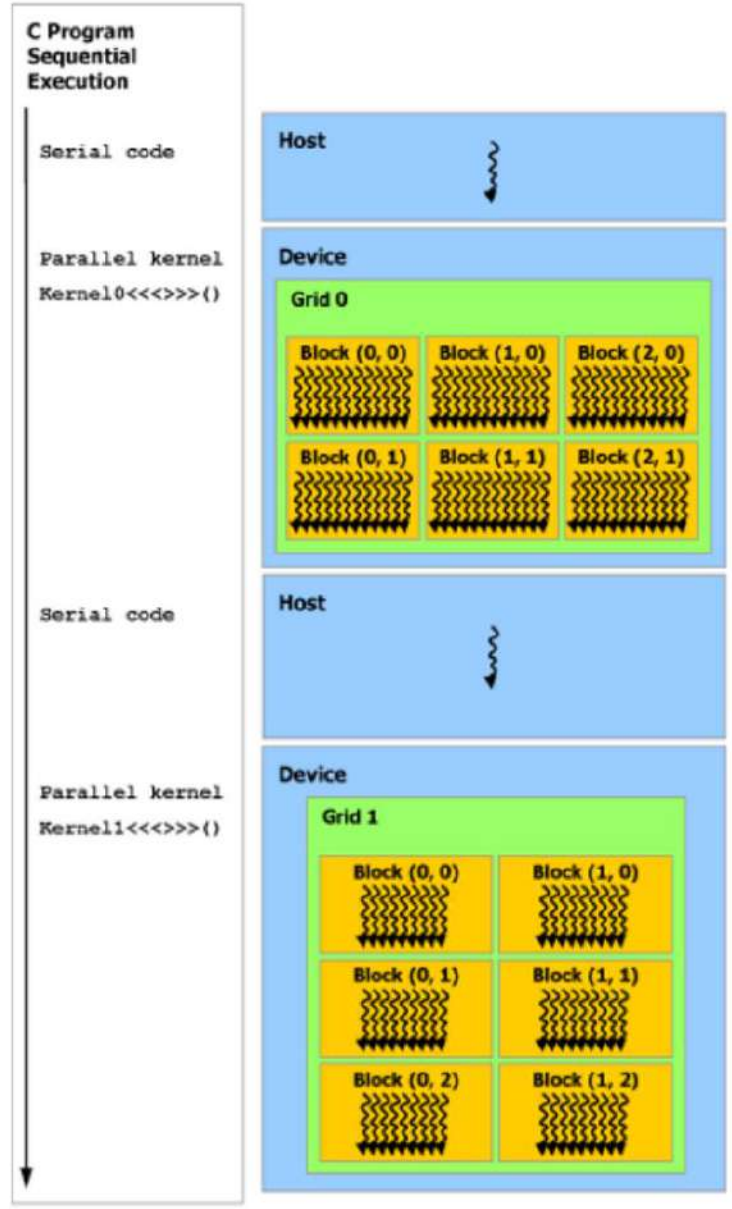


Figure 7: *Heterogeneous Programming*

Hello, CUDA!

```cpp
#include <stdio.h>
#include <cuda_runtime.h>

// CUDA kernel for vector addition
__global__ void vectorAdd(const float *a, const float *b, float *c, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}

int main() {
    const int N = 1024; // Vector size
    size_t size = N * sizeof(float);

    // Host vectors
    float *h_a = (float *)malloc(size);
    float *h_b = (float *)malloc(size);
    float *h_c = (float *)malloc(size);

    // Initialize input vectors
    for (int i = 0; i < N; i++) {
        h_a[i] = rand() / (float)RAND_MAX;
        h_b[i] = rand() / (float)RAND_MAX;
    }

    // Device vectors
    float *d_a, *d_b, *d_c;
    cudaError_t err;
    cudaMalloc(&d_a, size);
    cudaMalloc(&d_b, size);
    cudaMalloc(&d_c, size);

    // Copy inputs to device
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

    // Launch kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, N);

    // Copy result back to host
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
```

Lab03: CUDA

GPU Hierarchy

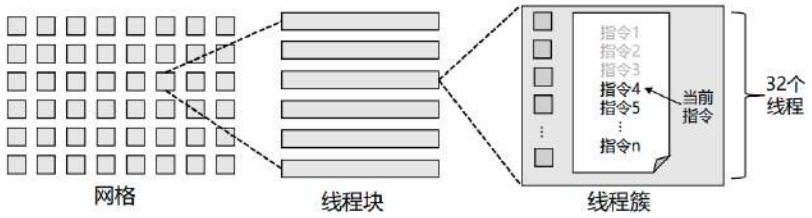
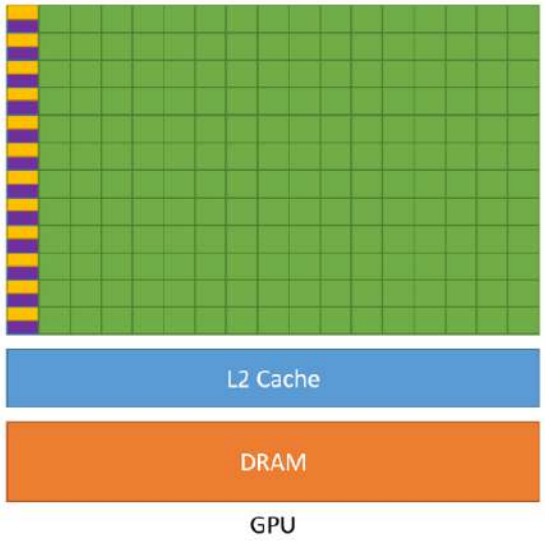


图 7-7　网格、线程块与线程簇之间的逻辑关系



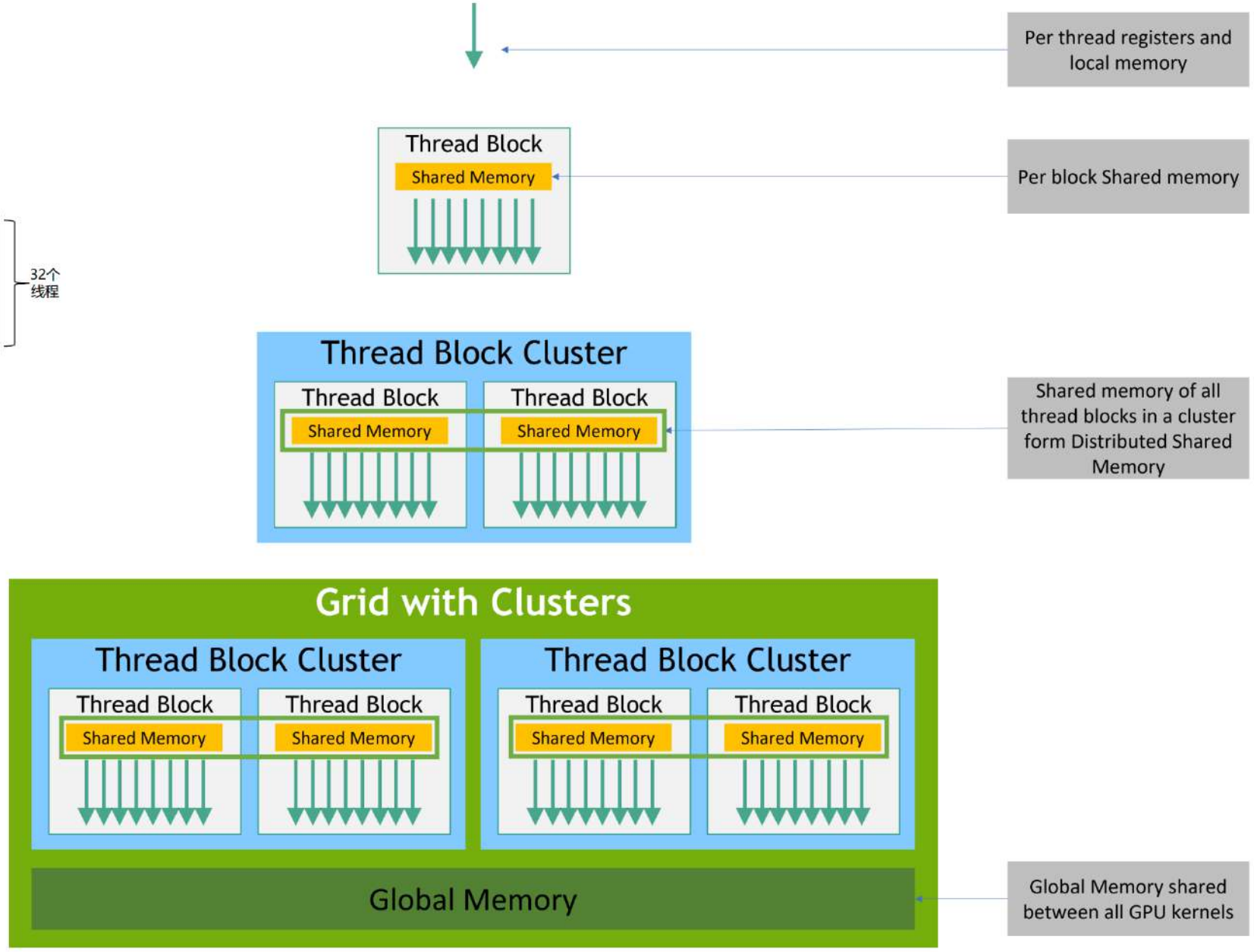
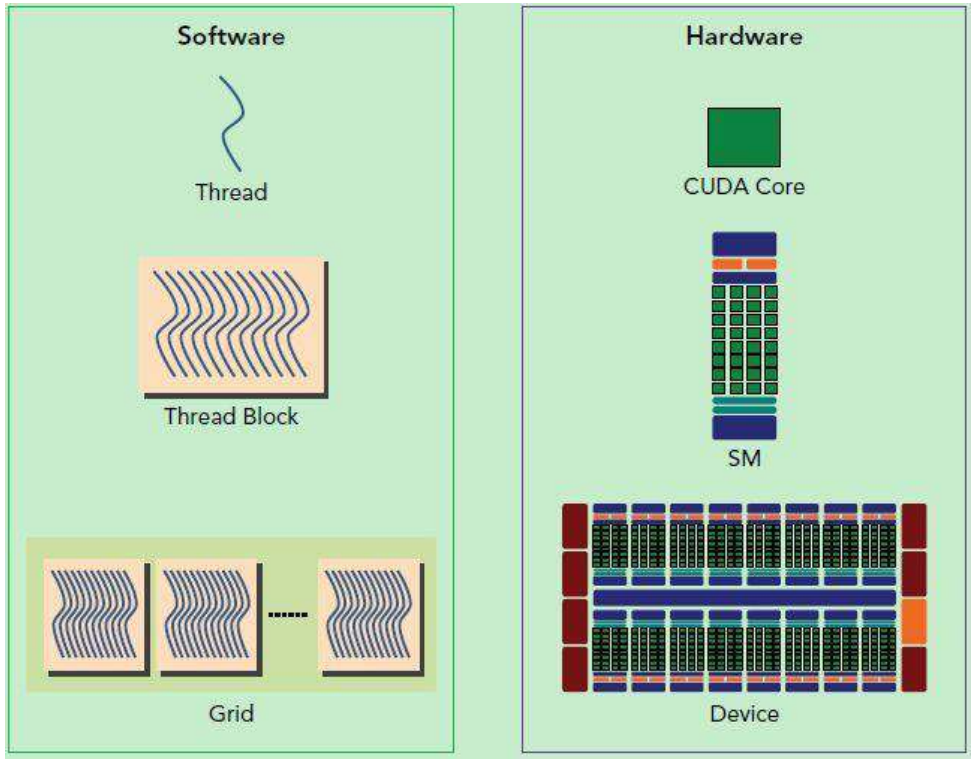


Figure 6: *Memory Hierarchy*



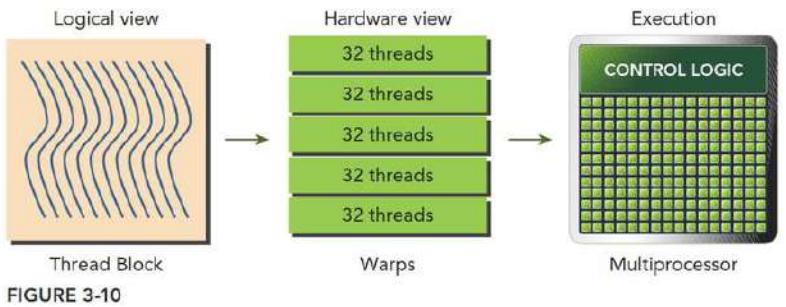
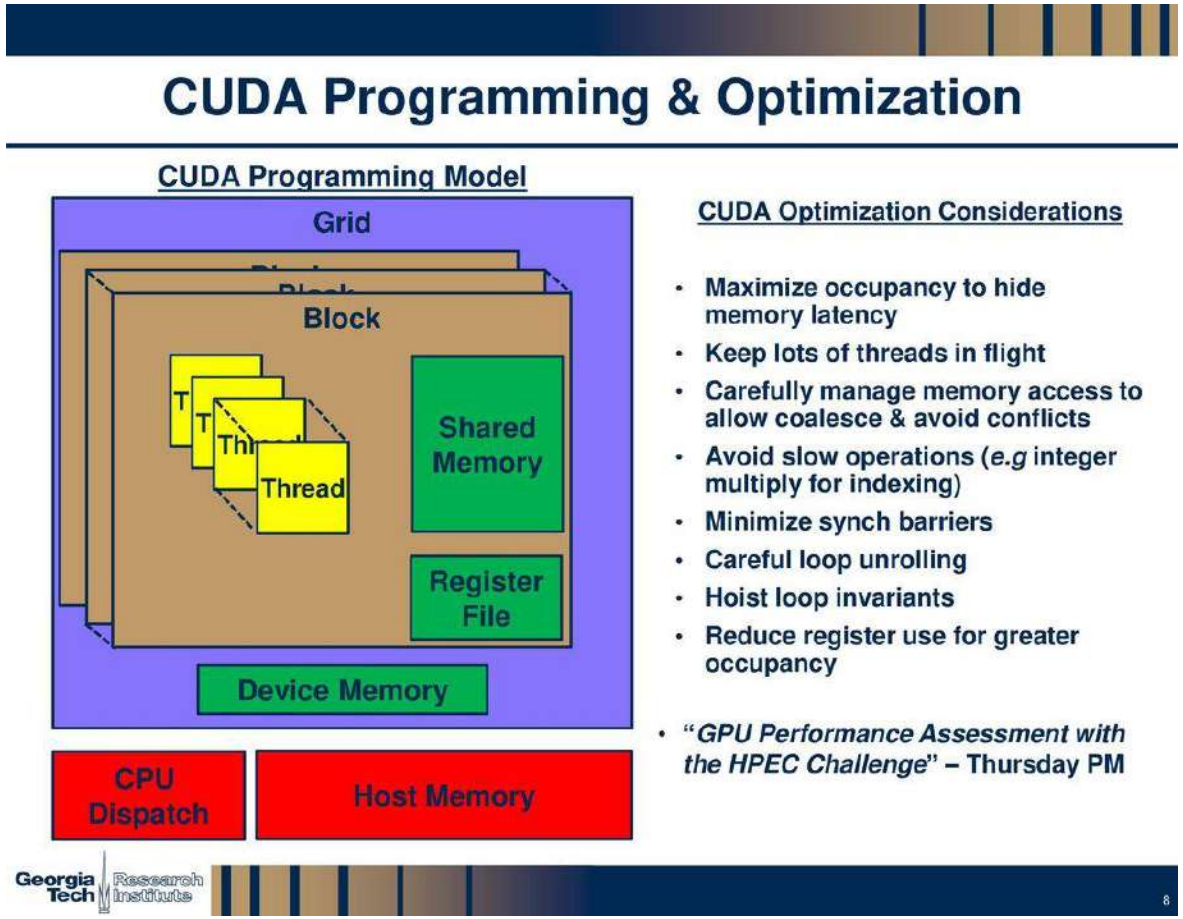


FIGURE 3-10



Warp 🌶：the smallest scheduling unit in GPU execution.

HGEMM in CUDA

Algorithm 1: Blocked HGEMM. Fragments (A_frag, B_frag, C_frag) reside in registers

```
1  __shared__ A_smem[b_m][b_k];
2  __shared__ B_smem[b_n][b_k];
3  A_frag[w_m][w_k];
4  B_frag[w_n][w_k];
5  C_frag[w_m][w_n];
6  for iter←0 to k by b_k do
7      A_smem ← b_m × b_k of A;
8      B_smem ← b_n × b_k of B;
9      for i←0 to b_k by w_k do
10         foreach 16 × 8 matrix in C_frag do
11             C_frag[][] ← C_frag[][] + A_frag[][i]×
                B_frag[][i];
12         end
13     end
14 end
```

```cpp
// Launch kernel
dim3 blockDim(TILE_WIDTH, TILE_WIDTH);
dim3 gridDim((N + TILE_WIDTH - 1) / TILE_WIDTH, (M + TILE_WIDTH - 1) / TILE_WIDTH);
hgemm_kernel<<<gridDim, blockDim>>>(d_A, d_B, d_C, M, N, K);
cudaErrCheck(cudaGetLastError());

// Copy result back to host
cudaErrCheck(cudaMemcpy(h_C, d_C, size_C, cudaMemcpyDeviceToHost));
```

```cpp
// CUDA kernel for FP16 matrix multiplication C = A * B
__global__ void hgemm_kernel(const half *__restrict__ A, const half *__restrict__ B, half *__restrict__ C,
                             int M, int N, int K) {
    __shared__ half As[TILE_WIDTH][TILE_WIDTH];
    __shared__ half Bs[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int row = by * TILE_WIDTH + ty;
    int col = bx * TILE_WIDTH + tx;

    half sum = __float2half(0.0f);

    for (int m = 0; m < (K + TILE_WIDTH - 1) / TILE_WIDTH; ++m) {
        // Load tiles into shared memory
        if (row < M && m * TILE_WIDTH + tx < K) {
            As[ty][tx] = A[row * K + m * TILE_WIDTH + tx];
        } else {
            As[ty][tx] = __float2half(0.0f);
        }

        if (m * TILE_WIDTH + ty < K && col < N) {
            Bs[ty][tx] = B[(m * TILE_WIDTH + ty) * N + col];
        } else {
            Bs[ty][tx] = __float2half(0.0f);
        }
        __syncthreads();

        // Compute partial sum for this tile
        for (int k = 0; k < TILE_WIDTH; ++k) {
            sum = __hadd(sum, __hmul(As[ty][k], Bs[k][tx]));
        }
        __syncthreads();
    }

    // Write result to global memory
    if (row < M && col < N) {
        C[row * N + col] = sum;
    }
}
```

Tiling in CUDA

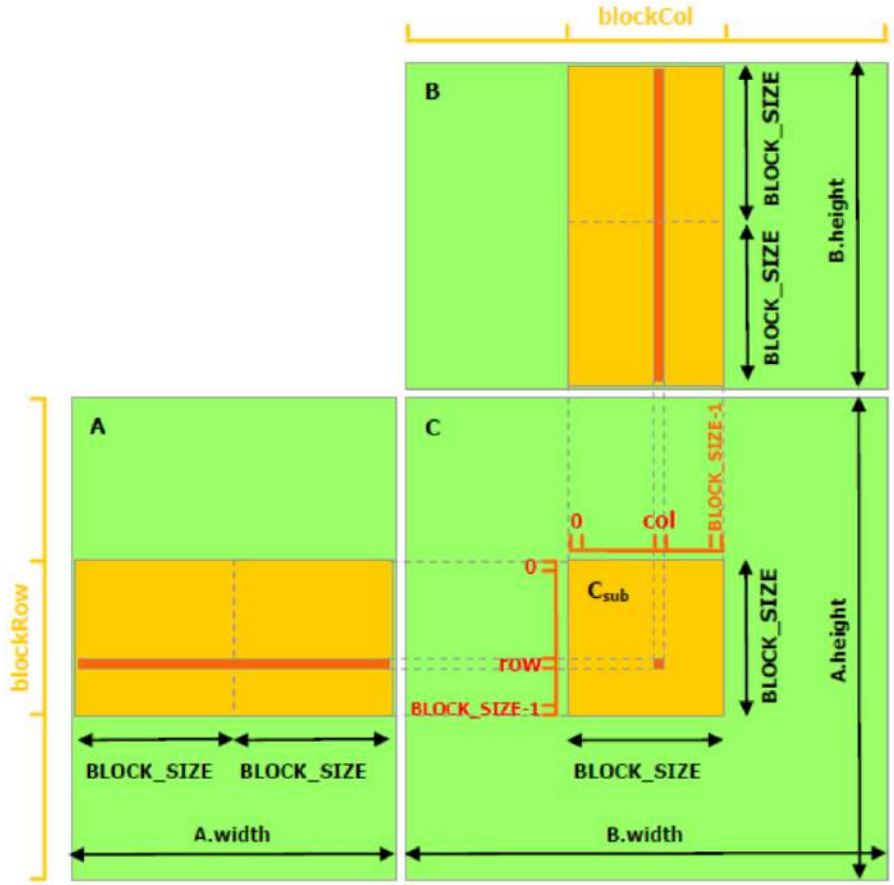
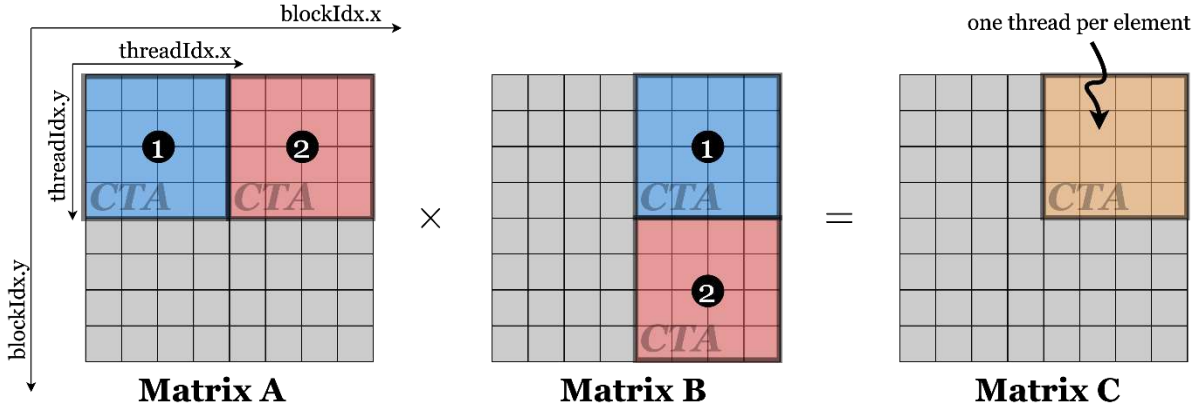
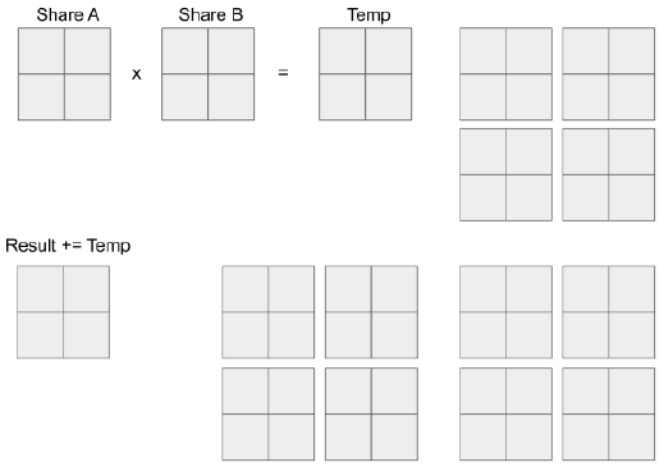


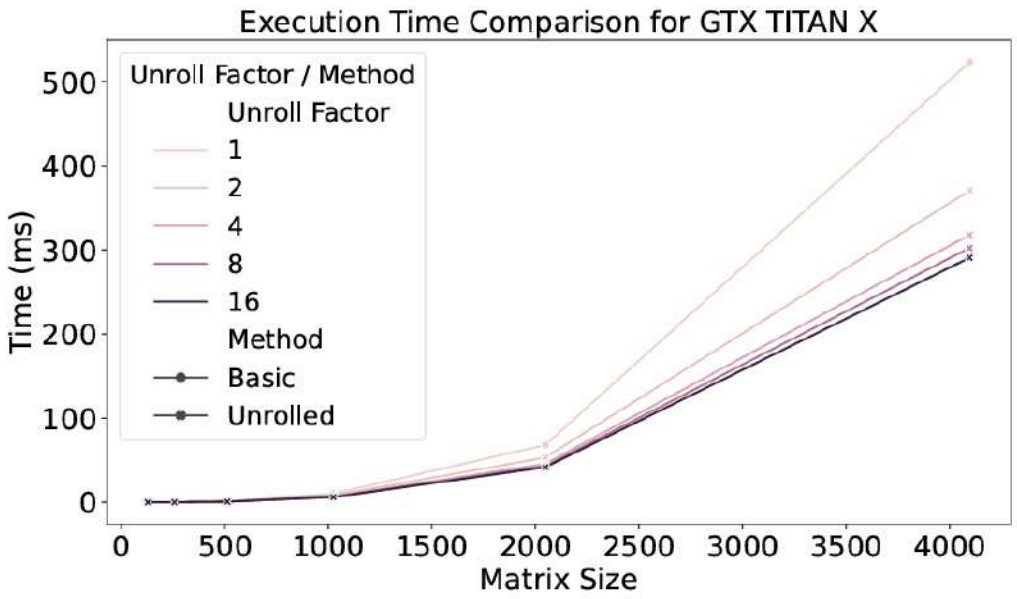
Figure 9: *Matrix Multiplication with Shared Memory*





penny-xu.github.io/blog/tiled-matrix-multiplication/

Loop Unrolling in CUDA



[Loop Unrolling Impact on CUDA Matrix Multiplication Operations | IEEE Conference Publication | IEEE Xplore]

[CUDA ---- Branch Divergence and Unrolling Loop - 苹果妖 - 博客园]

```cuda
__global__ void reduceUnrollWarps8 (int *g_idata, int *g_odata, unsigned int n) {
    // set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x*blockDim.x*8 + threadIdx.x;

    // convert global data pointer to the local pointer of this block
    int *idata = g_idata + blockIdx.x*blockDim.x*8;

    // unrolling 8
    if (idx + 7*blockDim.x < n) {
        int a1 = g_idata[idx];
        int a2 = g_idata[idx+blockDim.x];
        int a3 = g_idata[idx+2*blockDim.x];
        int a4 = g_idata[idx+3*blockDim.x];
        int b1 = g_idata[idx+4*blockDim.x];
        int b2 = g_idata[idx+5*blockDim.x];
        int b3 = g_idata[idx+6*blockDim.x];
        int b4 = g_idata[idx+7*blockDim.x];
        g_idata[idx] = a1+a2+a3+a4+b1+b2+b3+b4;
    }
    __syncthreads();

    // in-place reduction in global memory
    for (int stride = blockDim.x / 2; stride > 32; stride >>= 1) {

        if (tid < stride) {
            idata[tid] += idata[tid + stride];
        }

        // synchronize within threadblock
        __syncthreads();
    }

    // unrolling warp
    if (tid < 32) {
        volatile int *vmem = idata;
        vmem[tid] += vmem[tid + 32];
        vmem[tid] += vmem[tid + 16];
        vmem[tid] += vmem[tid + 8];
        vmem[tid] += vmem[tid + 4];
        vmem[tid] += vmem[tid + 2];
        vmem[tid] += vmem[tid + 1];
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = idata[0];
}
```

V100

Figure 4. Volta GV100 Full GPU with 84 SM Units

Table 1. Comparison of NVIDIA Tesla GPUs

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS[1]	5	6.8	10.6	15.7
Peak FP64 TFLOPS[1]	1.7	.21	5.3	7.8
Peak Tensor TFLOPS[1]	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm²	601 mm²	610 mm²	815 mm²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

[1] Peak TFLOPS rates are based on GPU Boost Clock

Tensor Cores

Tensor Core Unit (TCU) is increasingly integrated into modern high-performance processors to enhance matrix multiplication performance.

NVIDIA Tensor Core

为生成式 AI 实现大规模加速

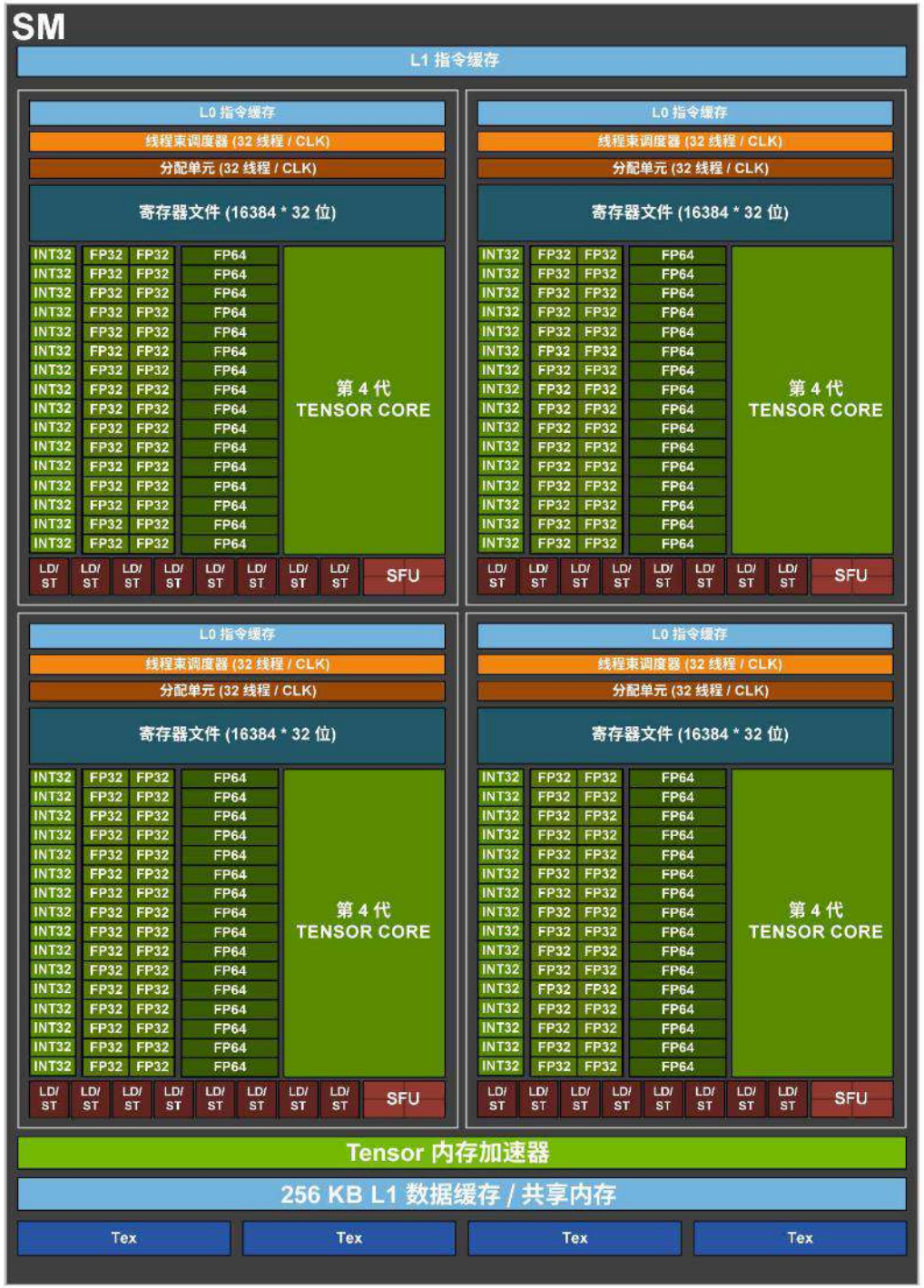
Tensor Core 可实现混合精度计算，动态调整算力，从而在保持准确性和提供更强安全性的同时提高吞吐量。在应对广泛的 AI 和高性能计算 (HPC) 任务时，新一代 Tensor Core 的速度更胜以往。NVIDIA Tensor Core 可将万亿级参数生成式 AI 模型的训练速度提高 4 倍，将推理性能提升 30 倍，并加速现代 AI 工厂的所有工作负载。

Table 1. NVIDIA A100 Tensor Core GPU Performance Specs

Peak FP64[1]	9.7 TFLOPS
Peak FP64 Tensor Core[1]	19.5 TFLOPS
Peak FP32[1]	19.5 TFLOPS
Peak FP16[1]	78 TFLOPS
Peak BF16[1]	39 TFLOPS
Peak TF32 Tensor Core[1]	156 TFLOPS | 312 TFLOPS[2]
Peak FP16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak BF16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak INT8 Tensor Core[1]	624 TOPS | 1,248 TOPS[2]
Peak INT4 Tensor Core[1]	1,248 TOPS | 2,496 TOPS[2]

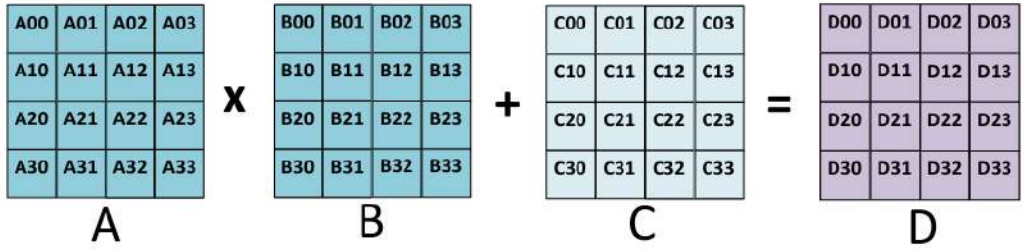
1 - Peak rates are based on GPU Boost Clock.
2 - Effective TFLOPS / TOPS using the new Sparsity feature

4x108 = 432 Tensor Cores in A100



Tensor Cores

Each tensor core can complete a single 4x4 matrix-multiply-and-accumulation (MACC) each clock cycle



3: Tensor cores complete one 4×4 MACC operation
le ($D = A * B + C$). Reproduces Figure 8 in [26].

A100 Tensor Cores Accelerate HPC

The performance needs of High-Performance Computing (HPC) applications are growing rapidly. Many applications from a wide range of scientific and research disciplines rely on double precision (FP64) computations. To meet the rapidly growing compute needs of HPC computing, A100 Tensor Cores support acceleration of IEEE-compliant FP64 computations, delivering up to 2.5x the FP64 performance of the NVIDIA Tesla V100 GPU. The new Double Precision Matrix Multiply Add instruction on A100 replaces 8 DFMA instructions on V100, reducing instruction fetches, scheduling overhead, register reads, datapath power, and shared memory read bandwidth. Using Tensor Cores, each SM in A100 computes a total of 64 FP64 FMA operations/clock (or 128 FP64 operations/clock), which is twice the throughput of Tesla V100. The A100 Tensor Core GPU with 108 SMs delivers a peak FP64 throughput of 19.5 TFLOPS, which is 2.5x that of Tesla V100.

With support for these new formats, the A100 Tensor Cores can be used to accelerate HPC workloads, iterative solvers, and various new AI algorithms.

UT OPERANDS	ACCUMULATOR	TOPS	X-factor vs. FFMA	SPARSE TOPS	SPARSE X-factor vs. FFMA
	FP32	15.7	1x	-	-
	FP32	125	8x	-	-
	FP32	19.5	1x	-	-
	FP32	156	8x	312	16x
	FP32	312	16x	624	32x
	FP32	312	16x	624	32x
	FP16	312	16x	624	32x
	INT32	624	32x	1248	64x
	INT32	1248	64x	2496	128x
Y	INT32	4992	256x	-	-
P64		19.5	1x	-	-

M. A. Raihan, N. Goli and T. M. Aamodt, "Modeling Deep Learning Accelerator Enabled GPUs," 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Madison, WI, USA, 2019, pp. 79-92, doi: 10.1109/ISPASS.2019.00016.

Hello, Tensor Core!

```cpp
// Kernel using WMMA to perform matrix multiplication C = A * B
__global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
   int warpM = (blockIdx.x * blockDim.x + threadIdx.x) / 32;
   int warpN = blockIdx.y * blockDim.y + threadIdx.y;

   nvcuda::wmma::fragment<nvcuda::wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, nvcuda::wmma::row_major> a_frag;
   nvcuda::wmma::fragment<nvcuda::wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, nvcuda::wmma::col_major> b_frag;
   nvcuda::wmma::fragment<nvcuda::wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;

   nvcuda::wmma::fill_fragment(c_frag, 0.0f);

   int aRow = warpM * WMMA_M;
   int aCol = 0;
   int bRow = 0;
   int bCol = warpN * WMMA_N;
   int cRow = warpM * WMMA_M;
   int cCol = warpN * WMMA_N;

   if (aRow < M && bCol < N && aCol < K) {
      for (int i = 0; i < K; i += WMMA_K) {
         nvcuda::wmma::load_matrix_sync(
         nvcuda::wmma::load_matrix_sync(
         nvcuda::wmma::mma_sync(
      }
      nvcuda::wmma::store_matrix_sync(
   }
}
```

Lab04: Tensor Core

yan-ipdps20.pdf

https://github.com/Bruce-Lee-LY/cuda_hgemm

How to use Tensor Core: WMMA

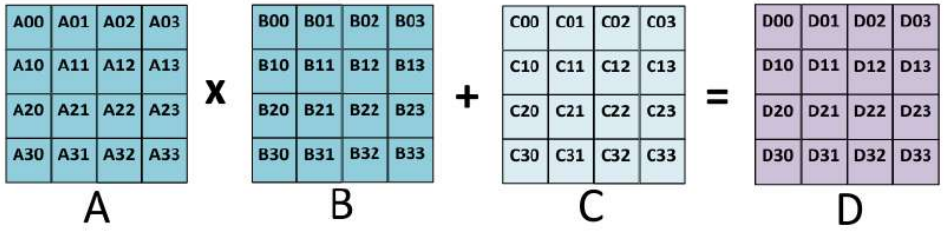


Figure 3: Tensor cores complete one 4×4 MACC operation per cycle ($D = A * B + C$). Reproduces Figure 8 in [26].

GPU programmer as warp-wide operations for performing the computation $D = A \times B + C$, where A, B, C and D can be tiles of larger matrices. Using the WMMA API, all threads in a warp cooperatively work together to perform a matrix-multiply and accumulate operation on these tiles. NVIDIA's WMMA API currently specifies a limited set of tile sizes. The sizes for tiles A, B, C and D are represented

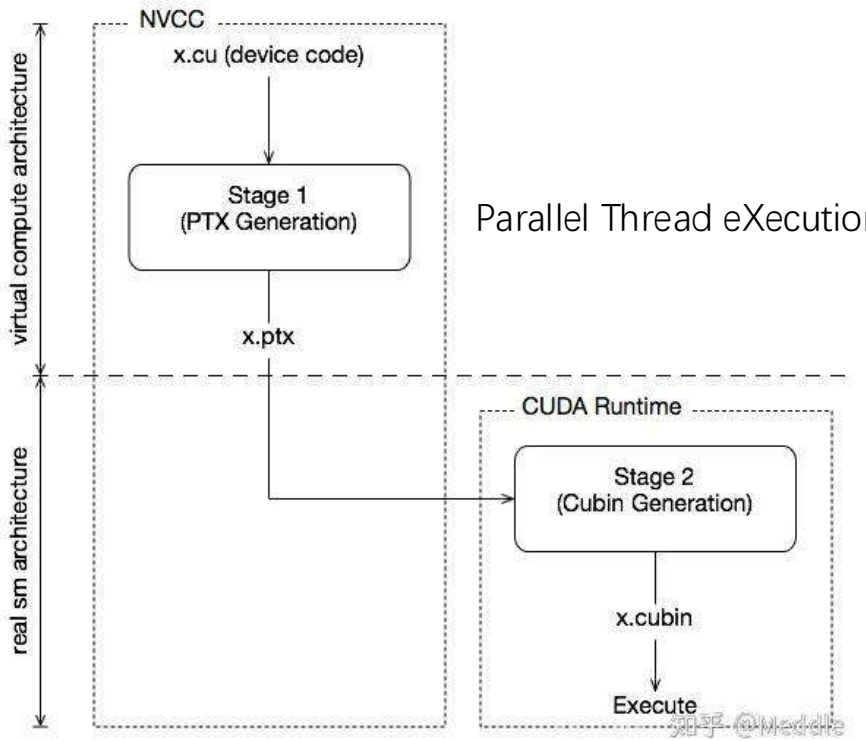
using the notation $M \times N \times K$, where $M \times K$ is the dimension of Tile A, $K \times N$ is the dimension of Tile B and thus C and D have dimension $M \times N$. CUDA 9.0 supports only one tile sizes, $16 \times 16 \times 16$, while later versions allow additional flexibility.

[code-samples/posts/tensor-cores/simpleTensorCoreGEMM.cu at master · NVIDIA-developer-blog/code-samples]

https://cx9898.github.io/post/CUDA%20-bian-cheng-shi-yong-%20Tensor%20core%20-xiang-jie.html

```cpp
__global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
   // Leading dimensions. Packed with no transpositions.
   int lda = M;
   int ldb = K;
   int ldc = M;

   // Tile using a 2D grid
   int warpM = (blockIdx.x * blockDim.x + threadIdx.x) / warpSize;
   int warpN = (blockIdx.y * blockDim.y + threadIdx.y);

   // Declare the fragments
   wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
   wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
   wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
   wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;

   wmma::fill_fragment(acc_frag, 0.0f);

   // Loop over k
   for (int i = 0; i < K; i += WMMA_K) {
      int aRow = warpM * WMMA_M;
      int aCol = i;

      int bRow = i;
      int bCol = warpN * WMMA_N;

      // Bounds checking
      if (aRow < M && aCol < K && bRow < K && bCol < N) {
         // Load the inputs
         wmma::load_matrix_sync(a_frag, a + aRow + aCol * lda, lda);
         wmma::load_matrix_sync(b_frag, b + bRow + bCol * ldb, ldb);

         // Perform the matrix multiplication
         wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);

      }
   }

   // Load in the current value of c, scale it by beta, and add this our result scaled by alpha
   int cRow = warpM * WMMA_M;
   int cCol = warpN * WMMA_N;

   if (cRow < M && cCol < N) {
      wmma::load_matrix_sync(c_frag, c + cRow + cCol * ldc, ldc, wmma::mem_col_major);

   #pragma unroll
      for(int i=0; i < c_frag.num_elements; i++) {
         c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];
      }

      // Store the output
      wmma::store_matrix_sync(c + cRow + cCol * ldc, c_frag, ldc, wmma::mem_col_major);
   }
}
```

How WMMA works



Parallel Thread eXecution (PTX)

A100
3060

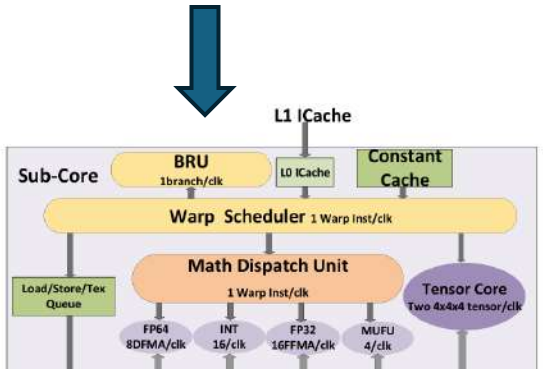
device-specific machine code (SASS)

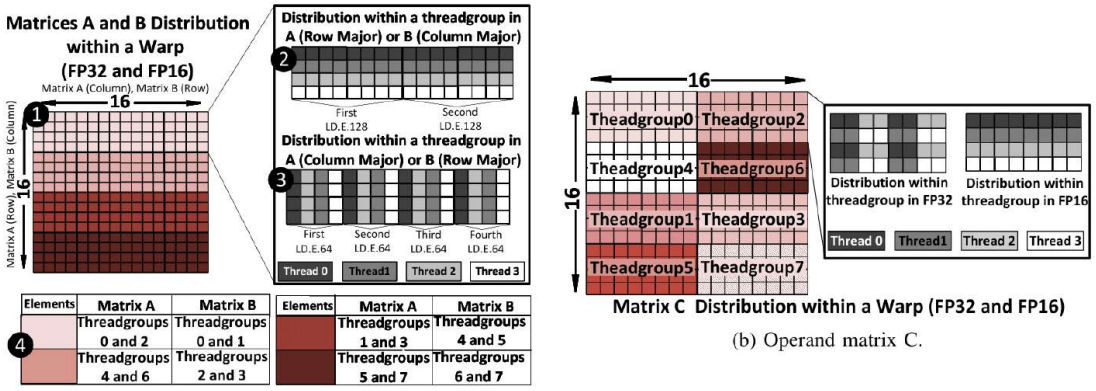
```
wmma.load.a.sync.layout.shape.type ra, [pa] {stride};
wmma.load.b.sync.layout.shape.type rb, [pb] {stride};
wmma.load.c.sync.layout.shape.type rc, [pc] {stride};
wmma.mma.sync.alayout.blayout.shape.dtype.ctype rd, ra, rb, rc;
wmma.store.d.sync.layout.shape.type rd, [pd] {stride};
```

Figure 2: Tensor Core PTX instructions

device-specific machine code (SASS)

[理解Tensor Core - 知乎]

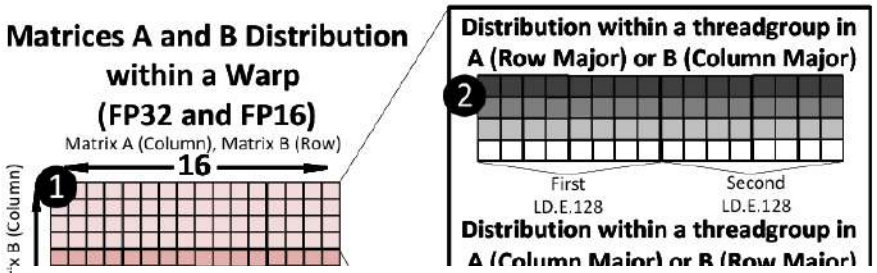




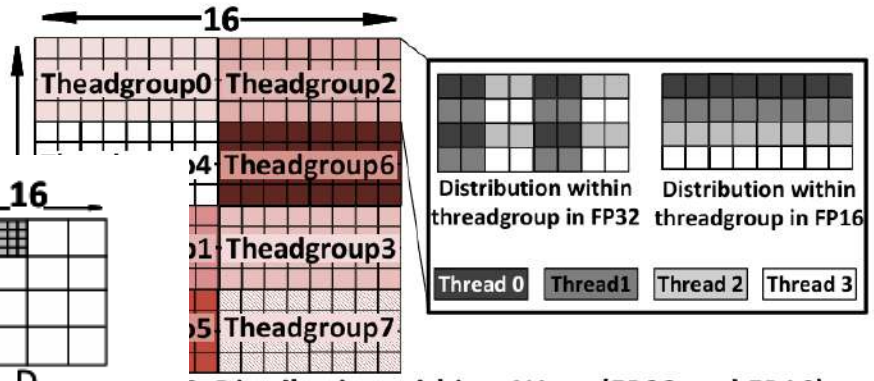
(a) Operand matrices A and B.

(b) Operand matrix C.

Figure 7: Distribution of operand matrix elements to threads for Tensor Cores in the Titan V (Volta).



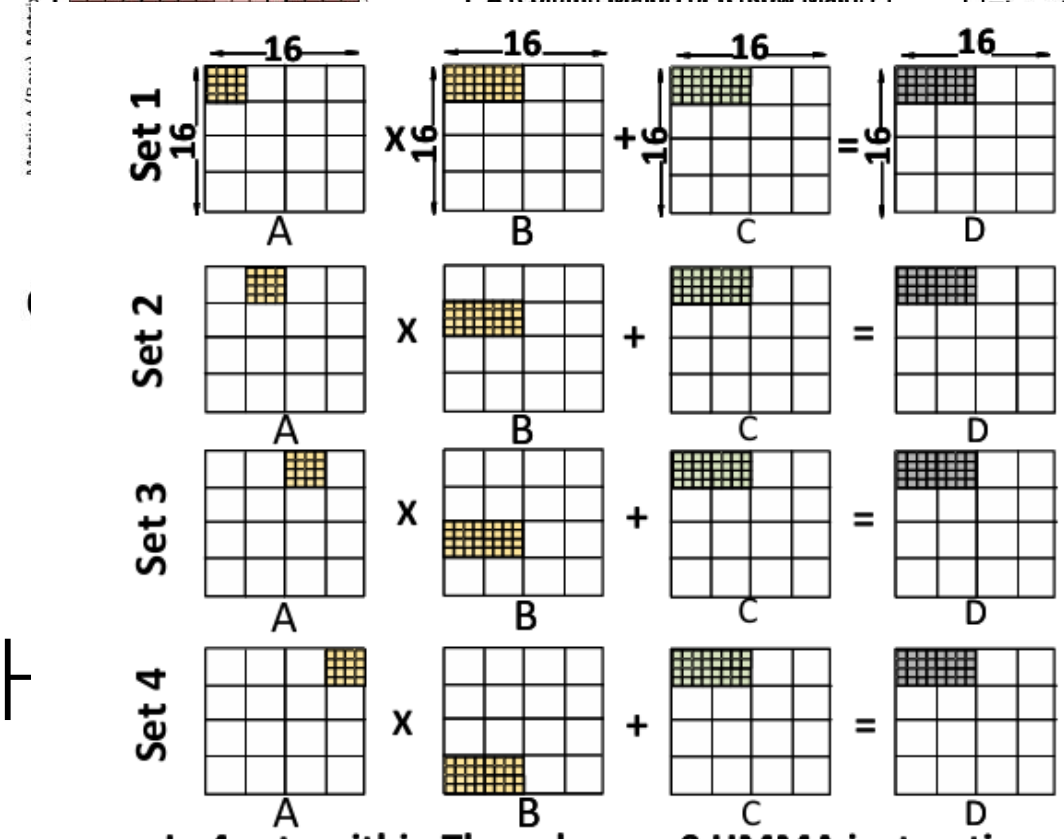
Segments



Distribution within a Warp (FP32 and FP16)

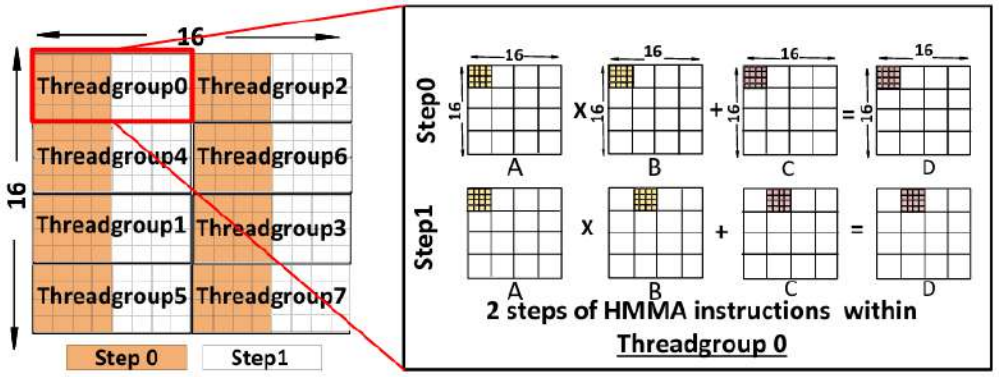
(b) Operand matrix C.

Implemented on Hardware



(a) Elements accessed in each "Set"

Tensor Cores in the Titan V (Volta).



(c) Elements accessed in each "Step" (FP16 mode).

Figure 10: HMMA instruction analysis for Volta (Titan V).

ConvStencil: Transform Stencil Computation to Matrix Multiplication on Tensor Cores

Yuetao Chen*
Microsoft Research
Beijing, China

Kun Li†
Microsoft Research
Beijing, China

Yuhao Wang*
Microsoft Research
University of Science and Technology of China
Hefei, China

PPoPP24 Best Paper

Accelerate **Stencil Computing** in **HPC** using **Tensor Cores**

具体例子：稳态热传导问题

问题描述

假设我们需要解决一个二维稳态热传导问题，物体的温度在平面内分布，并且已知物体边界的温度。我们可以通过稳态热传导方程来描述这个问题：

$$\nabla^2 T = 0$$

其中，T 是温度，∇^2 是拉普拉斯算子，表示热的扩散。这个方程表明，稳态条件下，温度在空间中是平衡的，不随时间变化。

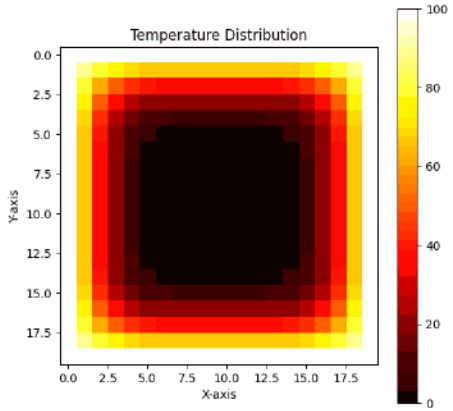
1. 离散化方法：有限差分法（FDM）

为了在计算机上求解这个方程，我们通常将区域离散化成网格，并用有限差分法（FDM）来近似拉普拉斯算子。在二维情况下，假设网格的间距为 h，离散化后的热传导方程为：

$$\frac{T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1} - 4T_{i,j}}{h^2} = 0$$

这里，$T_{i,j}$ 是网格点 (i, j) 处的温度。

```python
# 函数：执行热传导迭代
def solve_heat_conduction(grid, nx, ny, h):
    new_grid = np.zeros_like(grid)

    iter = 0
    while True:
        max_diff = 0.0

        # 迭代计算网格内部的温度
        for i in range(1, nx - 1):
            for j in range(1, ny - 1):
                new_grid[i, j] = 0.25 * (grid[i + 1, j] + grid[i - 1, j] +
                                         grid[i, j + 1] + grid[i, j - 1])
                diff = abs(new_grid[i, j] - grid[i, j])
                max_diff = max(max_diff, diff)

        # 更新网格
        grid[1:-1, 1:-1] = new_grid[1:-1, 1:-1]

        iter += 1
        if max_diff < TOL or iter >= MAX_ITER:
            break

    print(f"Iterations: {iter}")
```



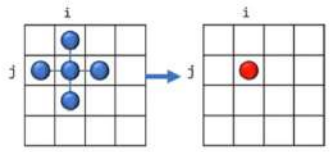

```python
# 主函数
def main():
    nx, ny = 20, 20  # 网格大小
    h = 1.0          # 网格间距

    # 创建温度网格并设置边界条件
    grid = np.zeros((nx, ny))
    grid[:, 0] = 100.0  # 左边界温度
    grid[:, -1] = 100.0  # 右边界温度
    grid[0, :] = 100.0  # 上边界温度
    grid[-1, :] = 100.0  # 下边界温度

    print("Initial grid:")
    print_grid(grid)

    # 进行热传导求解
    solve_heat_conduction(grid, nx, ny, h)

    # 输出最终的温度场
    print("\nFinal grid:")
    print_grid(grid)
```

Observation: Similarity with Convolution

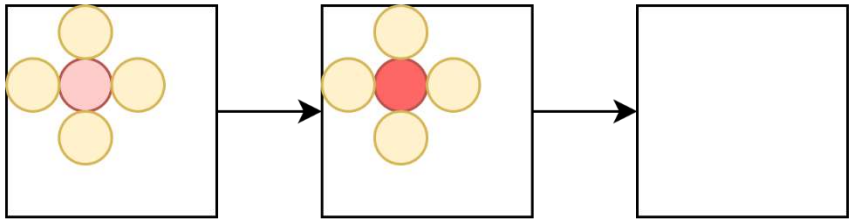
Stencil

- Stencil is one of the most important kernels widely used across a set of scientific and engineering applications, such as:
  - Thermal diffusion, Earth system model, Finite element method
- Stencil is the element-wise computation on a regular grid with a set of neighborhoods.
- At each time step, stencil reads multiple input and writes a single output to update the grid.

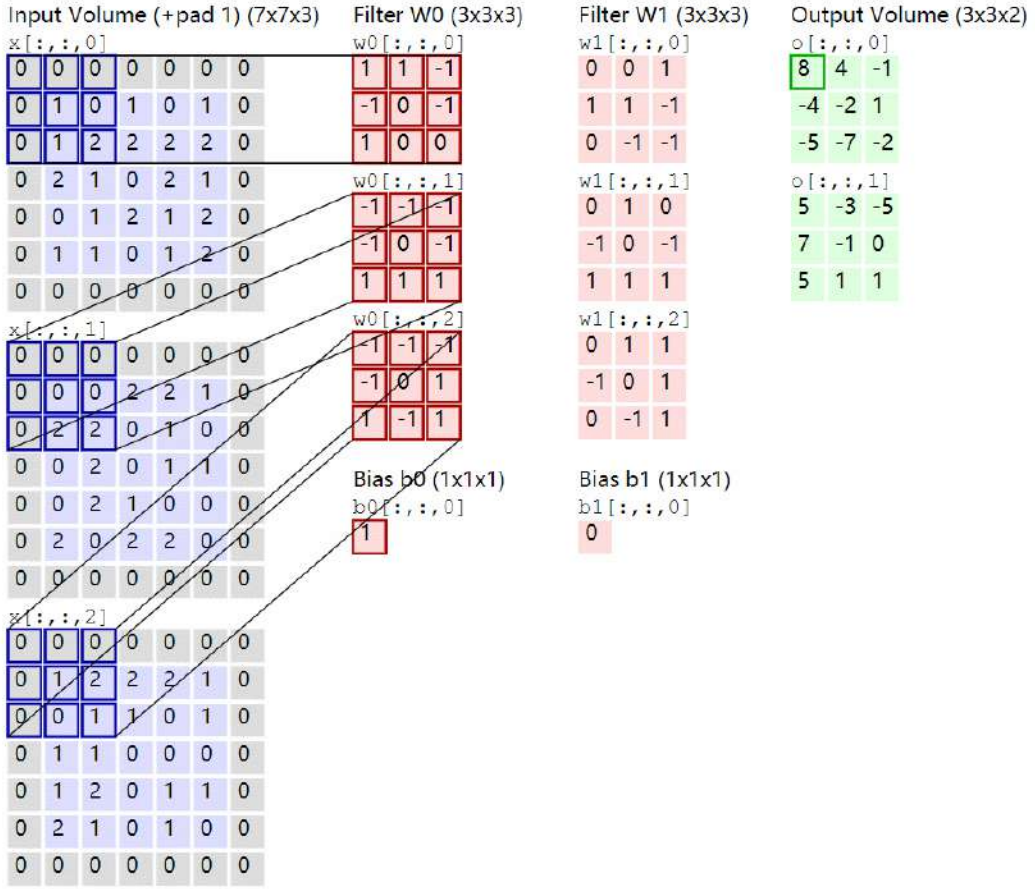
$$out(i,j) = \text{weight\_1} \times in(i,j) + \text{weight}_2 \times (in(i-1,j) + in(i,j-1) + in(i+1,j) + in(i,j+11))$$



Star-2D5P



...



Motivation: Accelerate Stencil like Convolution?

Image2rol->GEMM

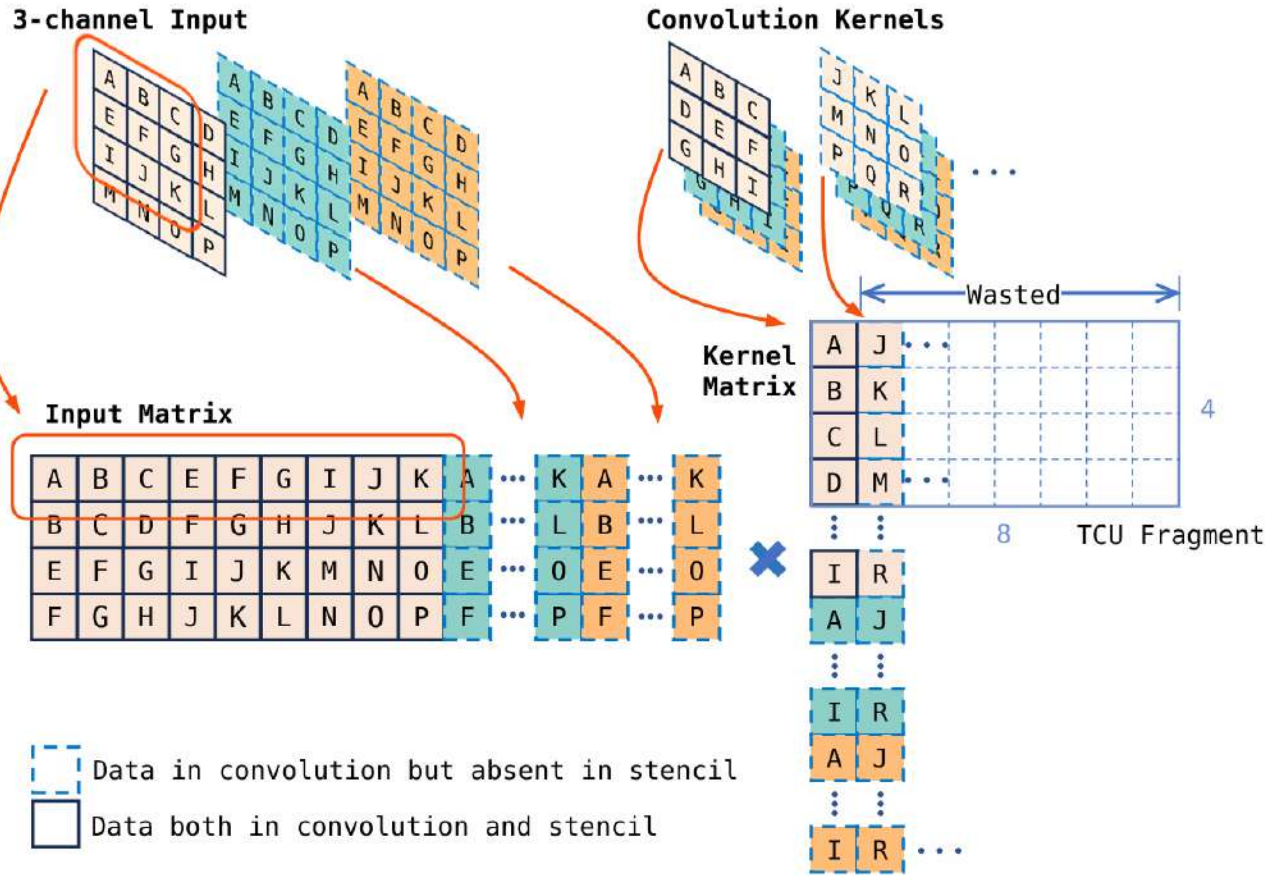


Figure 1. GEMM-based convolution and stencil.

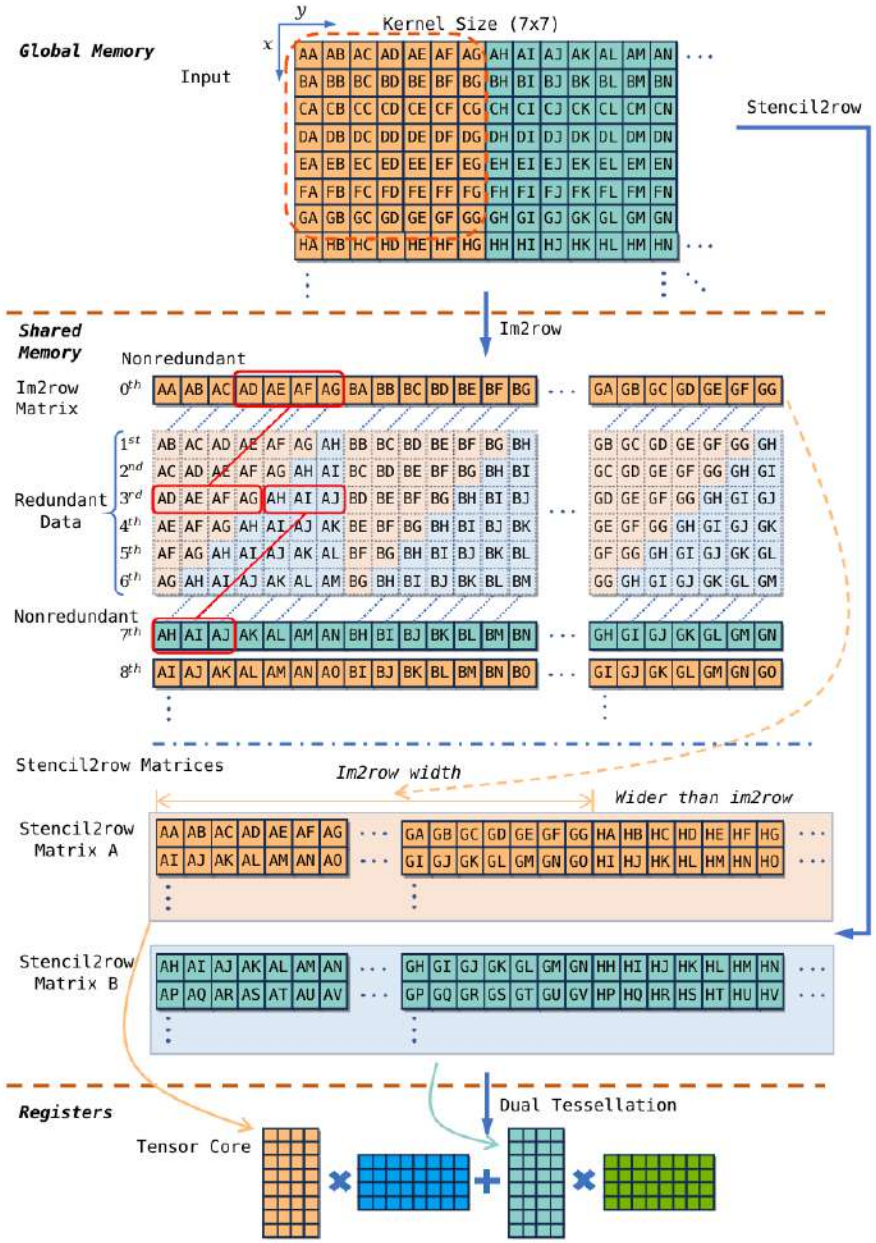
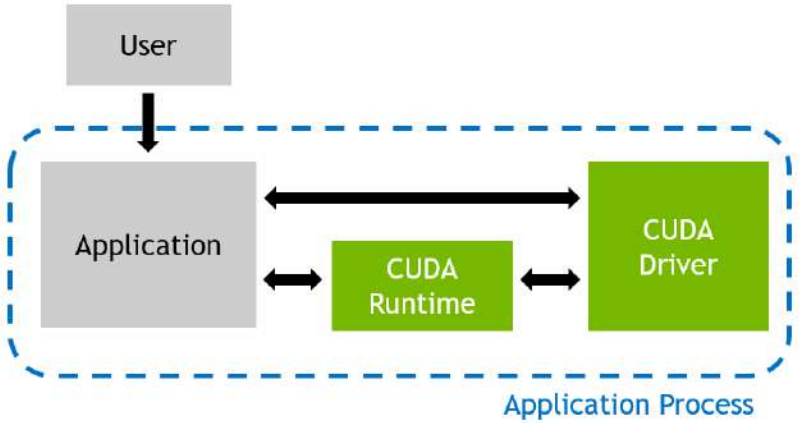


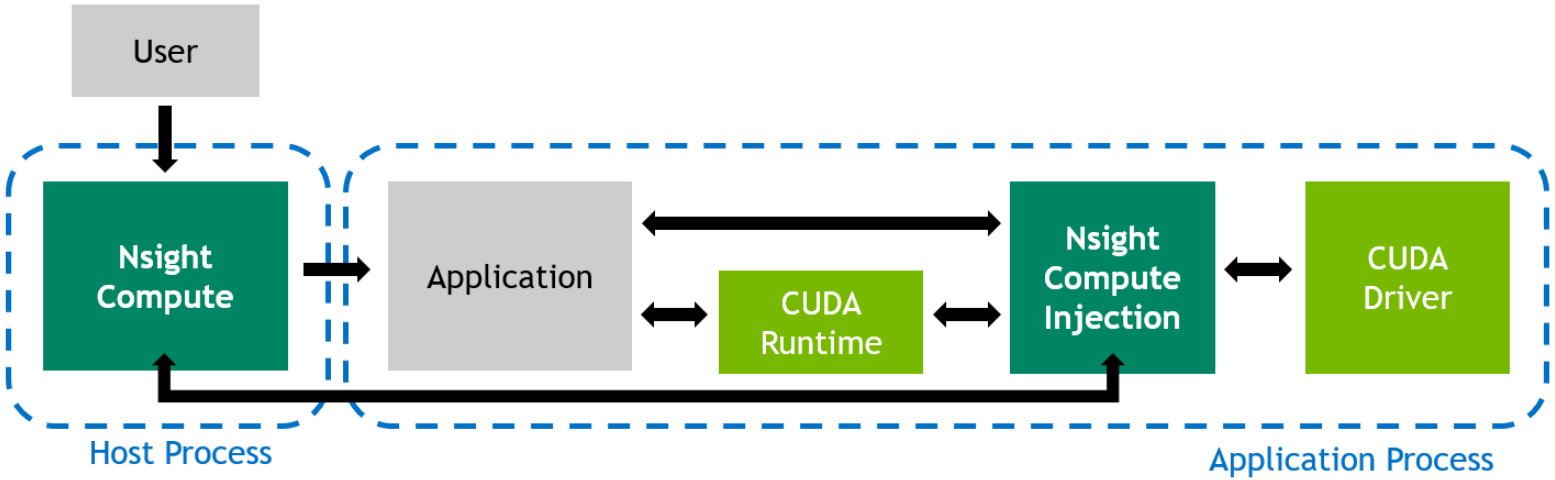
Figure 2. Stencil2row and its comparison with im2row.

Nsight Compute Profiler

Normal CUDA Function

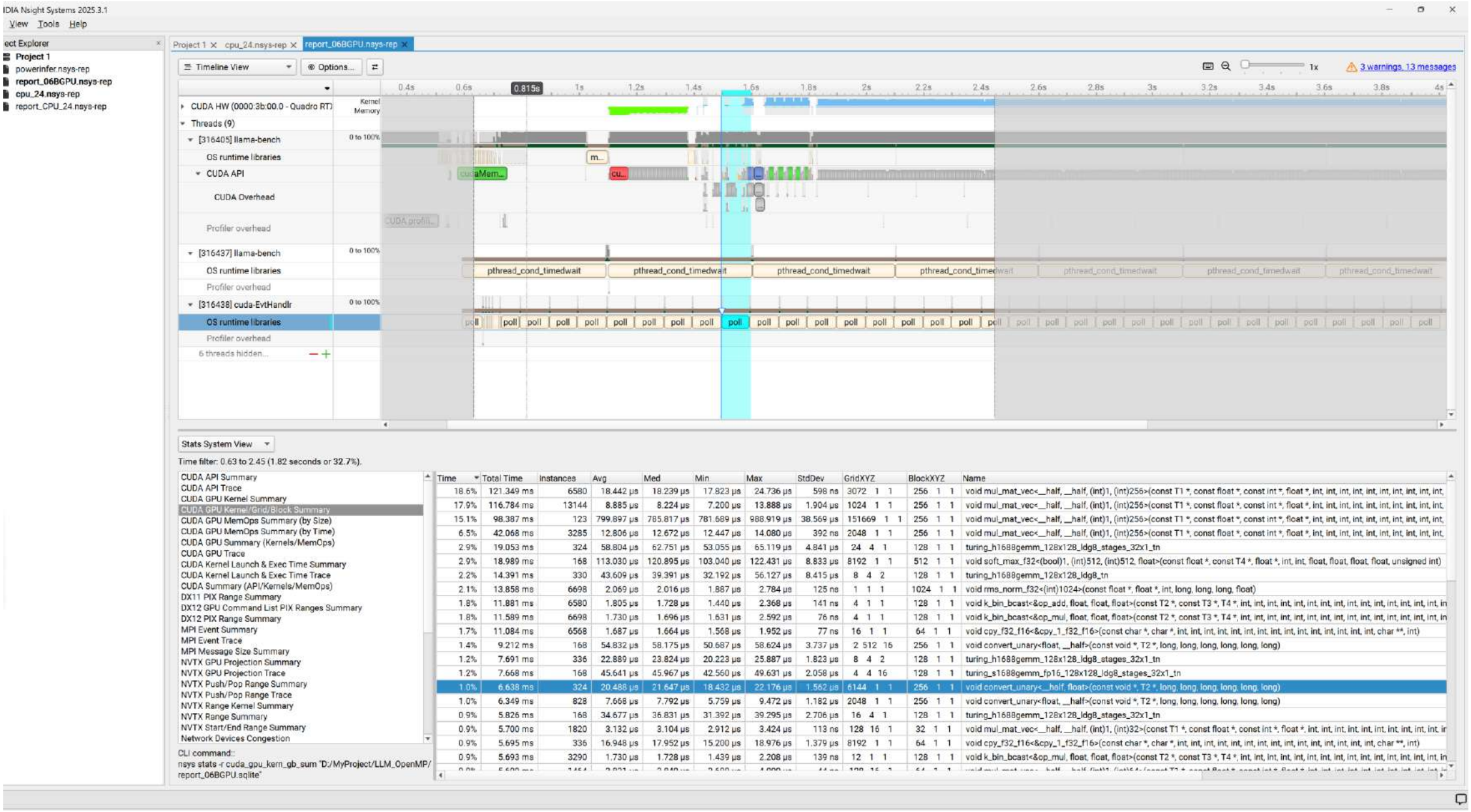


Nsight CUDA Profiling

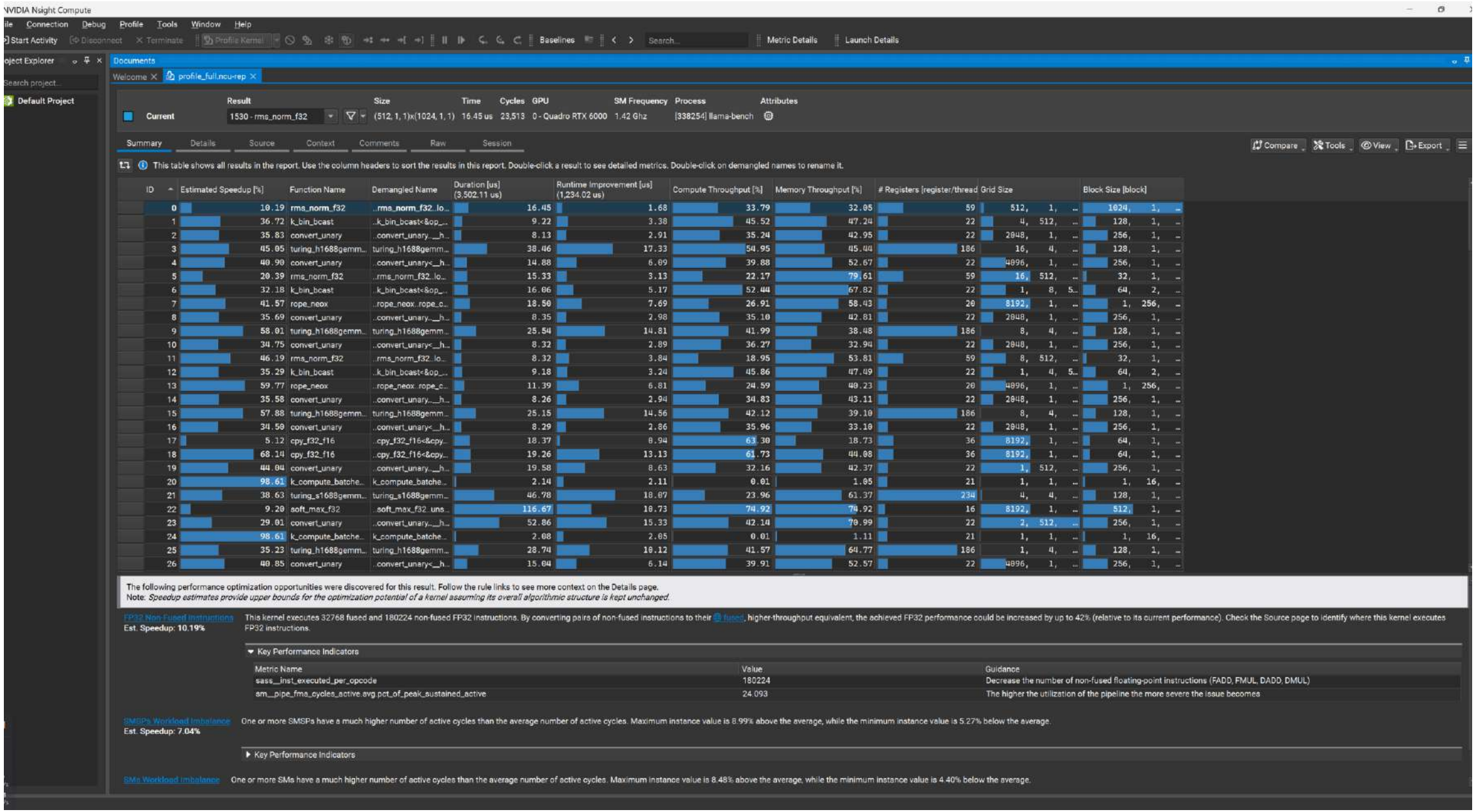


nsys

- a system-wide performance analysis tool designed to visualize an application's algorithms



ncu


NVIDIA Nsight Compute
File Connection Debug Profile Tools Window Help
Start Activity Disconnect Terminate Profile Kernel Baselines Search... Metric Details Launch Details
Project Explorer
Search project...
Default Project
Documents
Welcome
profile_full.ncu-rep
Result Size Time Cycles GPU SM Frequency Process Attributes
Current 1530 - rms_norm_f32 (512, 1, 1)x(1024, 1, 1) 16.45 us 23,513 0 - Quadro RTX 6000 1.42 Ghz [338254] llama-bench
Summary Details Source Context Comments Raw Session
Compare Tools View Export
This table shows all results in the report. Use the column headers to sort the results in this report. Double-click a result to see detailed metrics. Double-click on demangled names to rename it.
ID Estimated Speedup [%] Function Name Demangled Name Duration [us] (3,502.11 us) Runtime Improvement [us] (1,234.02 us) Compute Throughput [%] Memory Throughput [%] # Registers [register/thread] Grid Size Block Size [block]
0 10.19 rms_norm_f32 ..rms_norm_f32..lo... 16.45 1.68 33.79 32.05 59 512, 1, ... 1024, 1, ...
1 36.72 k_bin_bcast ..k_bin_bcast<&op_... 9.22 3.38 45.52 47.24 22 4, 512, ... 128, 1, ...
2 35.83 convert_unary ..convert_unary...__h... 8.13 2.91 35.24 42.95 22 2048, 1, ... 256, 1, ...
3 45.05 turing_h1688gemm... turing_h1688gemm... 38.46 17.33 54.95 45.44 186 16, 4, ... 128, 1, ...
4 40.90 convert_unary ..convert_unary<__h... 14.88 6.09 39.88 52.67 22 4096, 1, ... 256, 1, ...
5 20.39 rms_norm_f32 ..rms_norm_f32..lo... 15.33 3.13 22.17 79.61 59 16, 512, ... 32, 1, ...
6 32.18 k_bin_bcast ..k_bin_bcast<&op_... 16.06 5.17 52.44 67.82 22 1, 8, 5... 64, 2, ...
7 41.57 rope_neox ..rope_neox..rope_c... 18.50 7.69 26.91 58.43 20 8192, 1, ... 1, 256, ...
8 35.69 convert_unary ..convert_unary...__h... 8.35 2.98 35.10 42.81 22 2048, 1, ... 256, 1, ...
9 58.01 turing_h1688gemm... turing_h1688gemm... 25.54 14.81 41.99 38.48 186 8, 4, ... 128, 1, ...
10 34.75 convert_unary ..convert_unary<__h... 8.32 2.89 36.27 32.94 22 2048, 1, ... 256, 1, ...
11 46.19 rms_norm_f32 ..rms_norm_f32..lo... 8.32 3.84 18.95 53.81 59 8, 512, ... 32, 1, ...
12 35.29 k_bin_bcast ..k_bin_bcast<&op_... 9.18 3.24 45.86 47.49 22 1, 4, 5... 64, 2, ...
13 59.77 rope_neox ..rope_neox..rope_c... 11.39 6.81 24.59 40.23 20 4096, 1, ... 1, 256, ...
14 35.58 convert_unary ..convert_unary...__h... 8.26 2.94 34.83 43.11 22 2048, 1, ... 256, 1, ...
15 57.88 turing_h1688gemm... turing_h1688gemm... 25.15 14.56 42.12 39.10 186 8, 4, ... 128, 1, ...
16 34.59 convert_unary ..convert_unary<__h... 8.29 2.86 35.96 33.10 22 2048, 1, ... 256, 1, ...
17 5.12 cpy_f32_f16 ..cpy_f32_f16<&cpy... 18.37 0.94 63.30 18.73 36 8192, 1, ... 64, 1, ...
18 68.14 cpy_f32_f16 ..cpy_f32_f16<&cpy... 19.26 13.13 61.73 44.08 36 8192, 1, ... 64, 1, ...
19 44.04 convert_unary ..convert_unary...__h... 19.58 8.63 32.16 42.37 22 1, 512, ... 256, 1, ...
20 98.61 k_compute_batche... k_compute_batche... 2.14 2.11 0.01 1.05 21 1, 1, ... 1, 16, ...
21 38.63 turing_s1688gemm... turing_s1688gemm... 46.78 18.07 23.96 61.37 234 4, 4, ... 128, 1, ...
22 9.20 soft_max_f32 ..soft_max_f32..uns... 116.67 18.73 74.92 74.92 16 8192, 1, ... 512, 1, ...
23 29.01 convert_unary ..convert_unary...__h... 52.86 15.33 42.14 70.99 22 2, 512, ... 256, 1, ...
24 98.61 k_compute_batche... k_compute_batche... 2.08 2.05 0.01 1.11 21 1, 1, ... 1, 16, ...
25 35.23 turing_h1688gemm... turing_h1688gemm... 28.74 10.12 41.57 64.77 186 1, 4, ... 128, 1, ...
26 40.85 convert_unary ..convert_unary<__h... 15.04 6.14 39.91 52.57 22 4096, 1, ... 256, 1, ...
The following performance optimization opportunities were discovered for this result. Follow the rule links to see more context on the Details page.
Note. Speedup estimates provide upper bounds for the optimization potential of a kernel assuming its overall algorithmic structure is kept unchanged.
FP32 Non-Fused Instructions
Est. Speedup: 10.19%
This kernel executes 32768 fused and 180224 non-fused FP32 instructions. By converting pairs of non-fused instructions to their fused, higher-throughput equivalent, the achieved FP32 performance could be increased by up to 42% (relative to its current performance). Check the Source page to identify where this kernel executes FP32 instructions.
Key Performance Indicators
Metric Name Value Guidance
sass__inst_executed_per_opcode 180224 Decrease the number of non-fused floating-point instructions (FADD, FMUL, DADD, DMUL)
sm__pipe_fma_cycles_active.avg.pct_of_peak_sustained_active 24.093 The higher the utilization of the pipeline the more severe the issue becomes
SMSPs Workload Imbalance
Est. Speedup: 7.04%
One or more SMSPs have a much higher number of active cycles than the average number of active cycles. Maximum instance value is 8.99% above the average, while the minimum instance value is 5.27% below the average.
Key Performance Indicators
SMs Workload Imbalance
One or more SMs have a much higher number of active cycles than the average number of active cycles. Maximum instance value is 8.48% above the average, while the minimum instance value is 4.40% below the average.

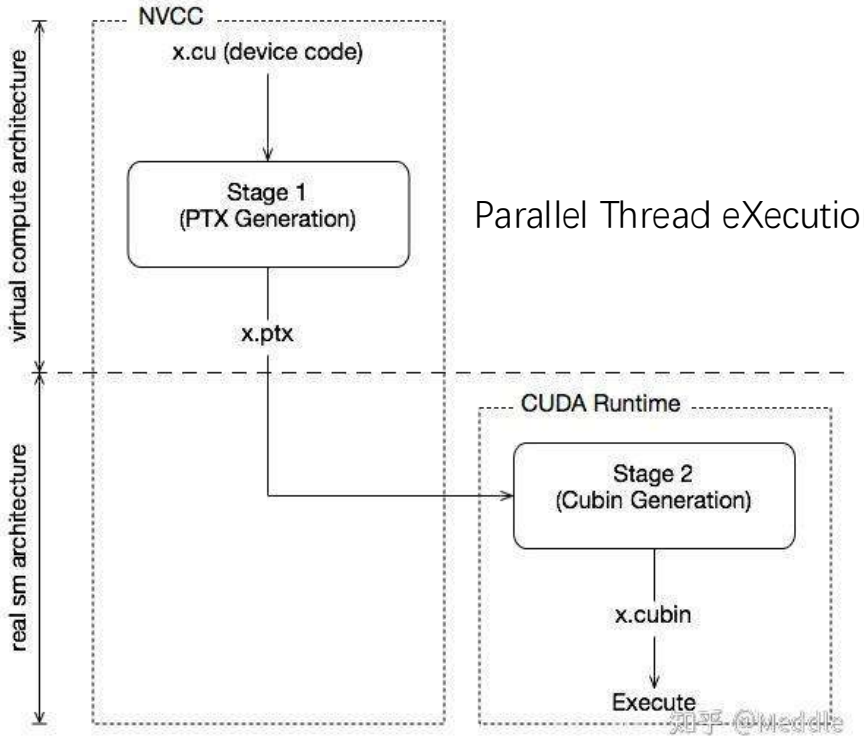
ncu

Lab05: NCU

PTX / SASS

```
.version 6.2
.target sm_35
.entry kernel_func

kernel_func:
  mov.u32 r0, ntid.x ; Thread ID in x-dimension
  mov.u32 r1, ntid.y ; Thread ID in y-dimension
```

```
kernel_func:
    mov r0, ntid.x
    mov r1, ntid.y
```



Parallel Thread eXecution (PTX)

A100
3060

device-specific machine code (SASS)

[deepseek-ai/DeepGEMM: DeepGEMM: clean and efficient FP8 GEMM kernels with fine-grained scaling]

[Parallel Thread Execution ISA]

Compute Intensity

Performance = Bandwidth × Compute Intensity

- **算力** π：也称为计算平台的**性能上限**，指的是一个计算平台倾尽全力每秒钟所能完成的浮点运算数。单位是 `FLOPS` or `FLOP/s` 。

$$\pi : \text{Maximum FLOPs Per Second}$$

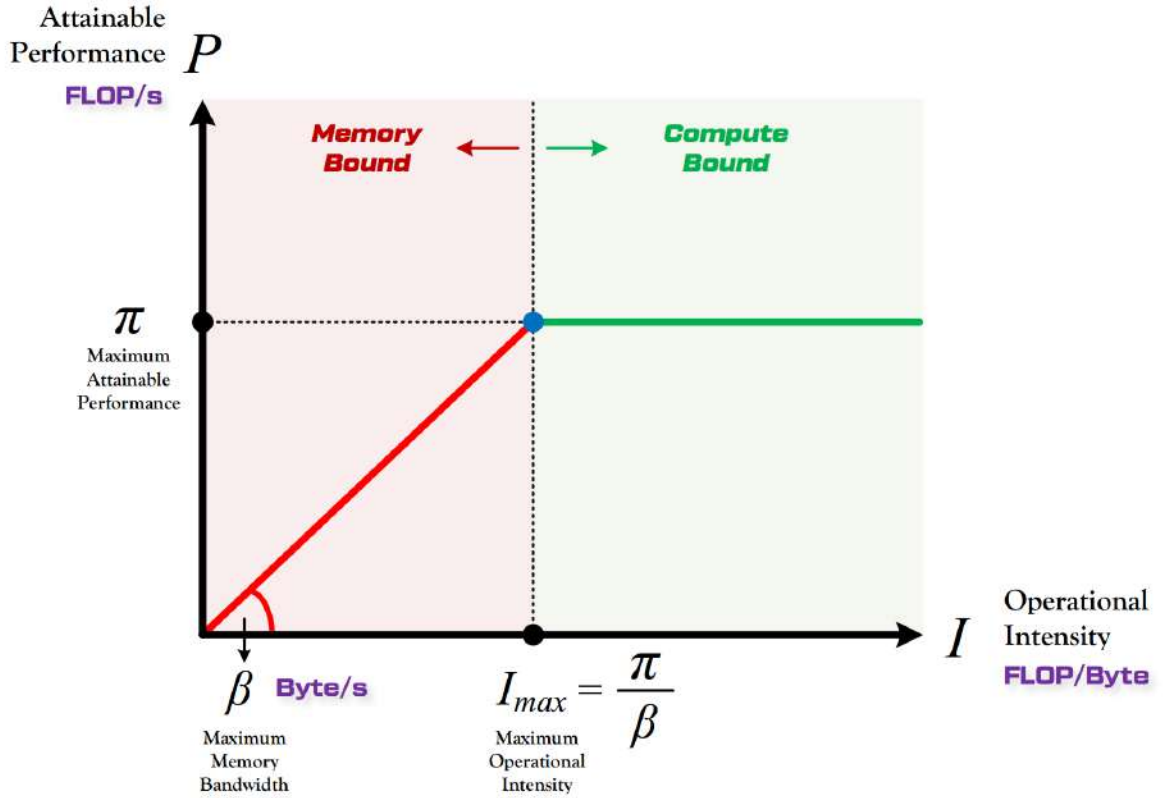
- **带宽** β：也即计算平台的**带宽上限**，指的是一个计算平台倾尽全力每秒所能完成的内存交换量。单位是 `Byte/s` 。

$$\beta : \text{Maximum Memory Access Per Second}$$

- **计算强度上限** I_{max}：两个指标相除即可得到计算平台的**计算强度上限**。它描述的是在这个计算平台上，单位内存交换最多用来进行多少次计算。单位是 `FLOPs/Byte` 。

$$I_{max} = \frac{\pi}{\beta}$$

1. **Pi (Peak Computational Performance)**:
   - Measured in FLOPs per second (e.g., GFLOPs/s or TFLOPs/s).
   - Represents the maximum theoretical computational capability of the processor (e.g., CPU or GPU).
   - Depends on factors like processor frequency, core count, and SIMD capabilities.
2. **Beta (Peak Memory Bandwidth)**:
   - Measured in bytes per second (e.g., GB/s).
   - Represents the maximum rate at which data can be transferred between memory and the processor.
   - Limited by the memory subsystem (e.g., DRAM or cache bandwidth).

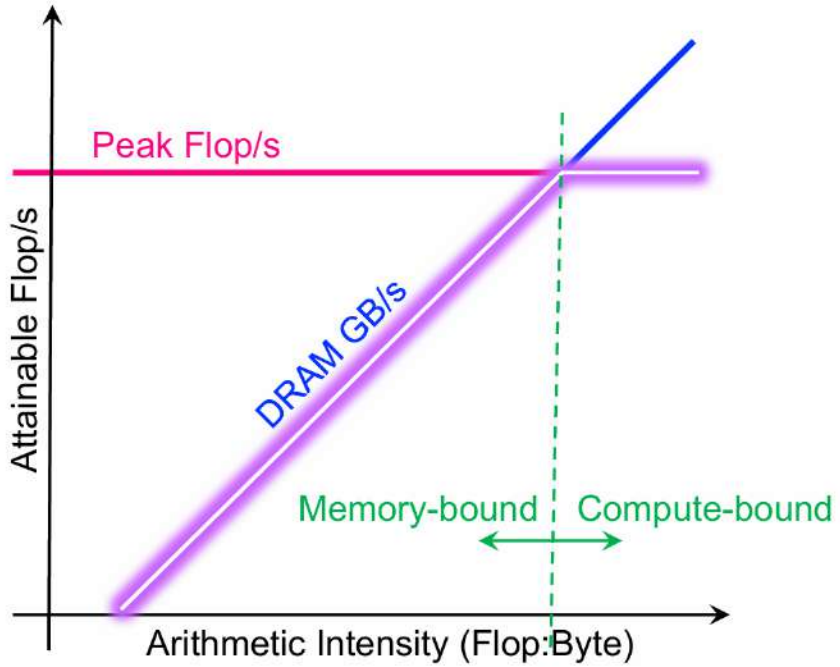


$$P = \begin{cases} \beta \cdot I, & when \ \ I < I_{max} \quad \textbf{Memory Bound} \\ \pi, & when \ \ I \geqslant I_{max} \quad \textbf{Compute Bound} \end{cases}$$

Roofline model

(DRAM) Roofline

- One could hope to always attain peak performance (Flop/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Consider idealized processor/caches
- Plot the performance bound using Arithmetic Intensity (AI) as the x-axis…
  - AI = Flops / Bytes presented to DRAM
  - **Attainable Flop/s = min(peak Flop/s, AI * peak GB/s)**
  - **Log-log scale** makes it easy to doodle, extrapolate performance along Moore's Law, etc…
  - Kernels with AI less than machine balance are ultimately DRAM bound (we'll refine this later…)



Roofline Model

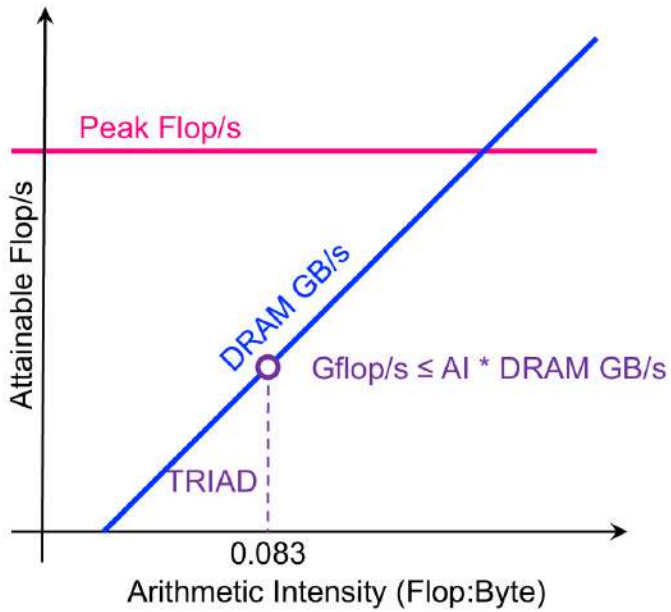
Roofline Example #1

- Typical machine balance is 5-10 flops per byte…
  - 40-80 flops per double to exploit compute capability
  - Artifact of technology and money
  - **Unlikely to improve**

- Consider STREAM Triad…

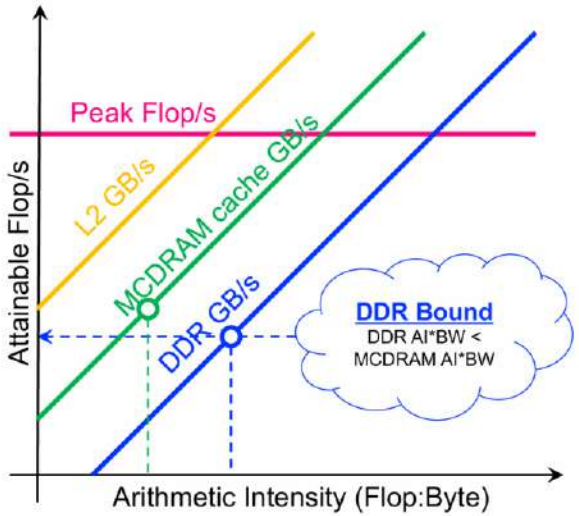
```
#pragma omp parallel for
for(i=0;i<N;i++){
  Z[i] = X[i] + alpha*Y[i];
}
```

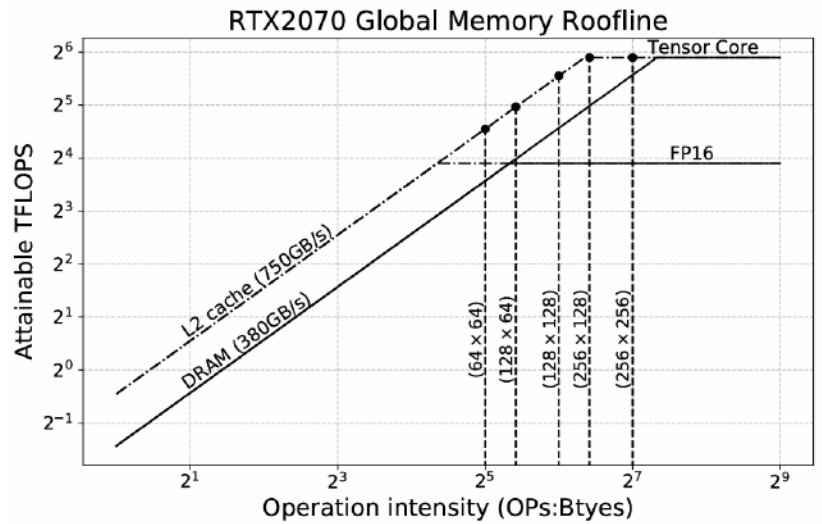
  - 2 flops per iteration
  - Transfer 24 bytes per iteration (read X[i], Y[i], write Z[i])
  - **AI = 0.083 flops per byte == Memory bound**



Hierarchical Roofline

- Construct superposition of Rooflines…
  - Measure a bandwidth
  - Measure AI for each level of memory
  - Although an loop nest may have multiple AI's and multiple bounds (flops, L1, L2, … DRAM)…
  - **… performance is bound by the minimum**





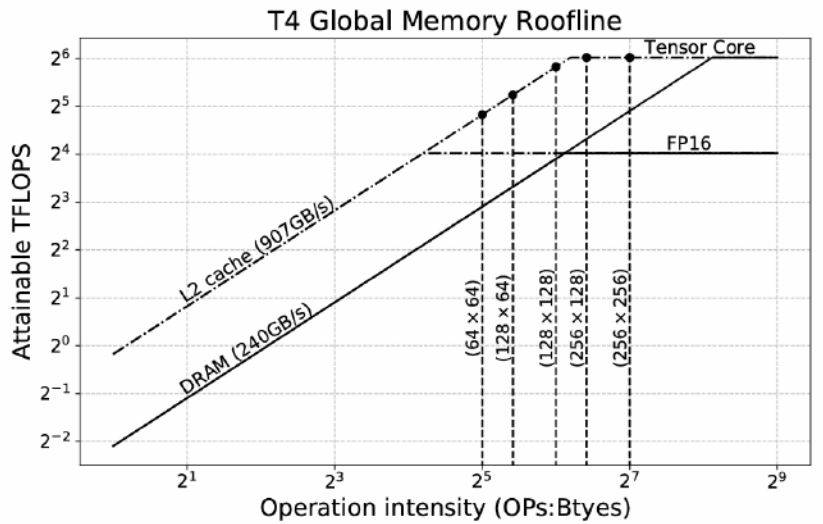
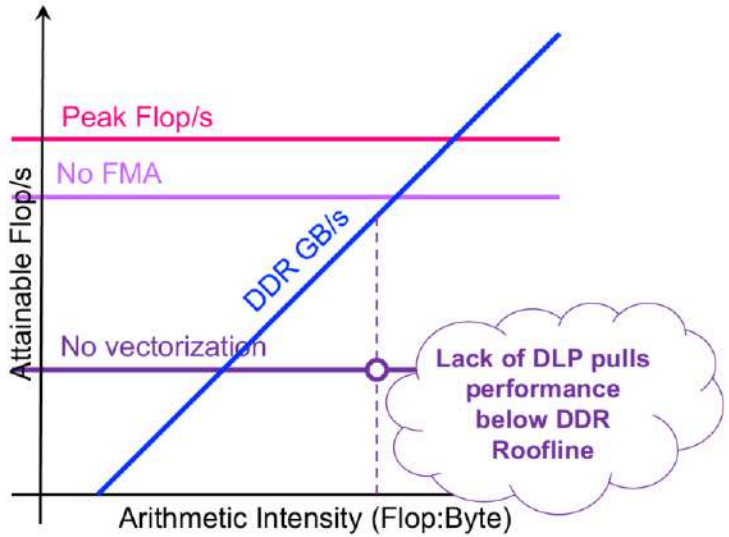


Fig. 3: Global memory roofline model on RTX2070 and T4. Compared with the solution using FP16 units, the high throughput of Tensor Core makes dense GEMM kind of memory-bounded.

Why ≠0?

Data, Instruction, Thread-Level Parallelism…

- We have assumed one can attain peak flops with high locality.
- In reality, this is premised on sufficient…
  - Use special instructions (e.g. fused multiply-add)
  - Vectorization (16 flops per instruction)
  - unrolling, out-of-order execution (hide FPU latency)
  - OpenMP across multiple cores
- Without these, …
  - Peak performance is not attainable
  - Some kernels can transition from memory-bound to compute-bound
  - n.b. in reality, DRAM bandwidth is often tied to DLP and TLP (single core can't saturate BW w/scalar code)



FMA: Fused Multiply-Add

Roofline Models

ASPLOS' 25

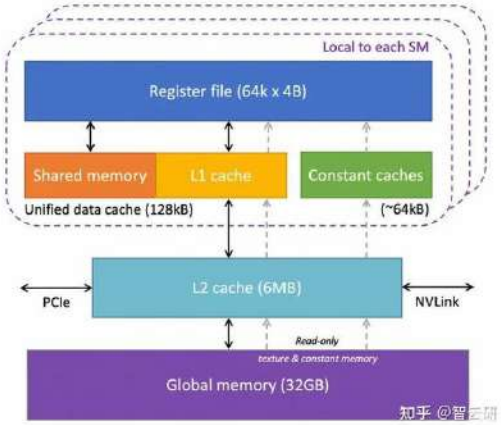
Squeezing Operator Performance Potential for the Ascend Architecture

Yuhang Zhou
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Zhibin Wang*
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Guyue Liu
Peking University
Beijing, China

Shipeng Li
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China



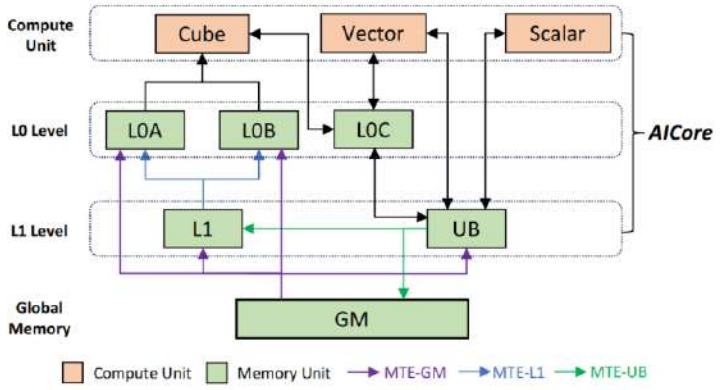
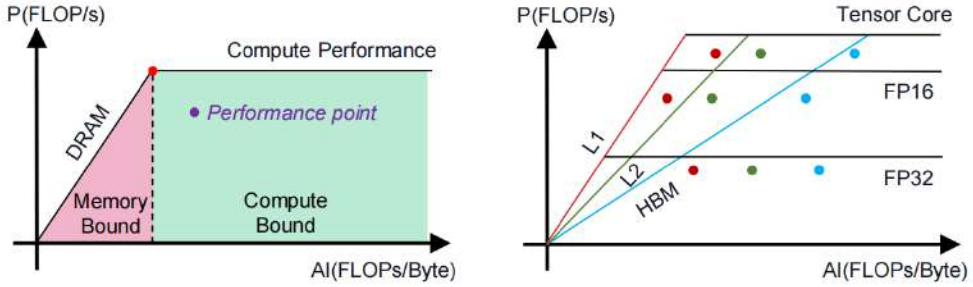


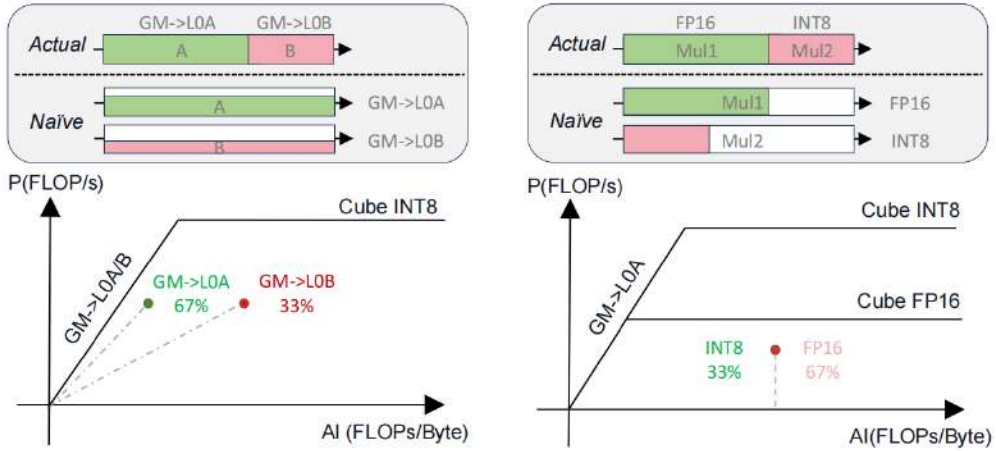
Figure 1. The architecture of AICore in Ascend.



(a) DRAM Roofline. **(b)** Hierarchical Roofline.

Figure 2. Existing roofline models.

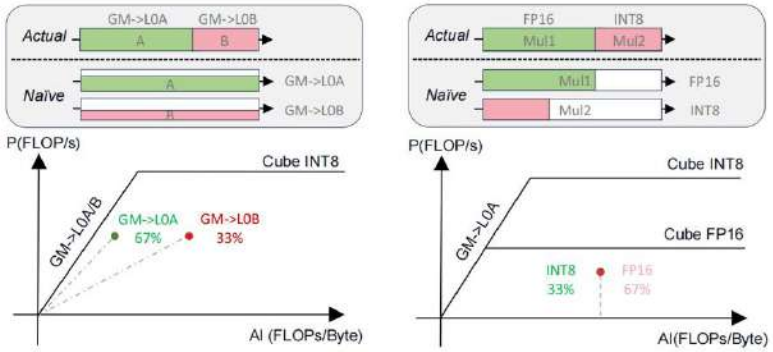
DRAM Roofline [47]. This model was originally conceived to evaluate the performance of CPU kernels accessing Dy-



(a) Bandwidth underutilization. **(b)** Compute underutilization.

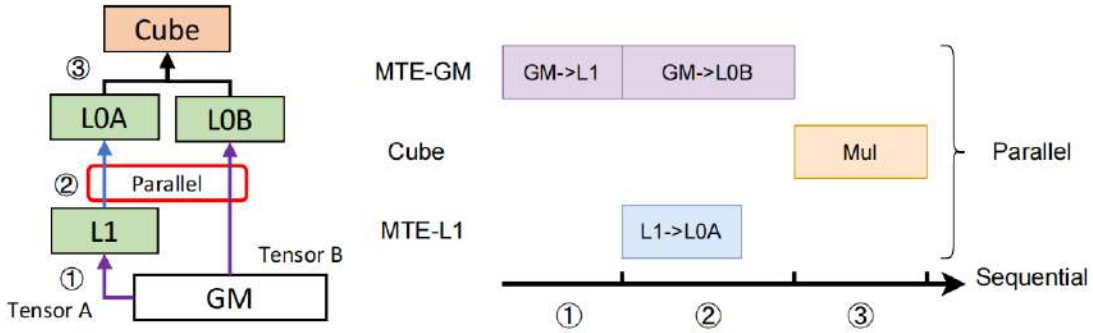
Figure 3. Incorrect analysis cases of the naive roofline.

Idea: Parallel



(a) Bandwidth underutilization. **(b)** Compute underutilization.

Figure 3. Incorrect analysis cases of the naive roofline.



(a) Data flow. **(b)** Execution timeline.

Figure 4. The execution of matrix multiplication $A \times B$.

Solution

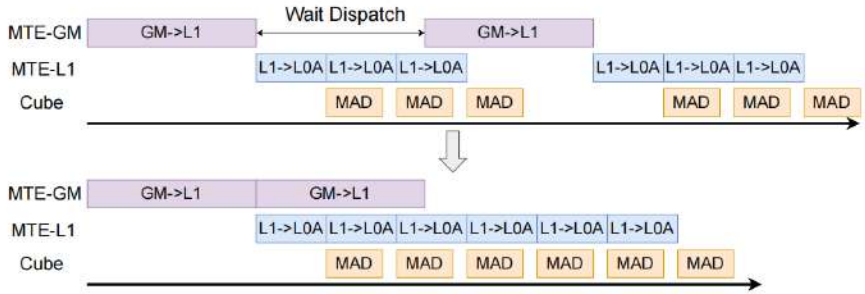


Figure 12. Adjusting instruction sequence.

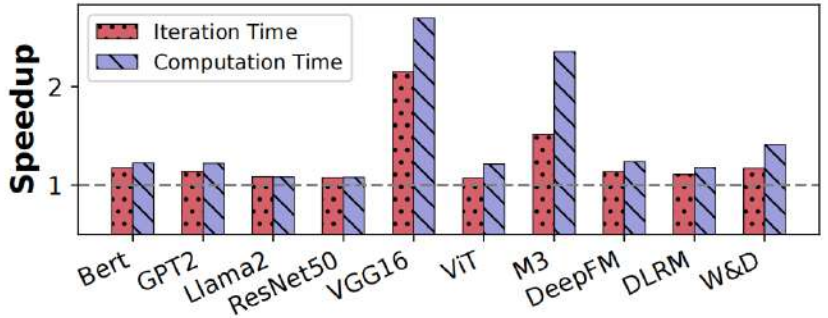
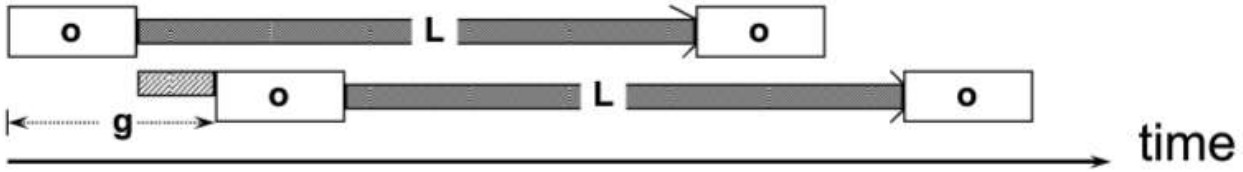


Figure 15. Time speedup with optimization.

LogP



- The LogP model is a simple framework for analyzing parallel computing systems. It abstracts the performance of distributed-memory parallel architectures by focusing on four key parameters:

- - **L (Latency)**: The time it takes to send a small message between two processors across the network.
- - **o (Overhead)**: The time a processor spends managing communication (e.g., initiating or receiving a message).
- - **g (Gap)**: The minimum time interval between consecutive message transmissions or receptions, reflecting network bandwidth limitations.
- - **P (Processors)**: The number of processors in the system.

- The model assumes that processors operate asynchronously and communicate by sending fixed-size messages. It simplifies real-world complexities by focusing on communication costs rather than computation details. LogP is useful for designing and analyzing parallel algorithms, as it helps predict performance bottlenecks related to communication latency, overhead, and bandwidth constraints. It's particularly effective for understanding algorithms on distributed systems like clusters or early parallel computers.

并行计算理论(002)-并行计算模型 - 知乎

High level technique

- Amdahl's Law
- LogP Model
- Roofline Model

- Share Memory
- SASS/PTX
- Warp Divergence
- Bank Conflict
- Double buffer

V100 Data Center GPU | NVIDIA

volta-architecture-whitepaper.pdf

666 赛题

- Greetings, pioneers! | SUSTCSC

- 我真的好懒啊，我不想做了，这些都是google+GPT就能学会的东西。但是一想想这PPT就讲出了我的人生，真是忍不住轻哼起来