UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Tarjetas Gráficas y Aceleradores
CUDA – MxM con Tensor Cores

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

Tensor Cores

- Aparecen por primera vez con Volta.
- Los Tensor Cores son hardware específico para:

$$D = \underset{\text{FP16 or FP32}}{\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}} \underset{\text{FP16}}{\begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}} + \underset{\text{FP16 or FP32}}{\begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}}$$

- En Volta, tenemos 8 Tensor Cores por SM.
- En las siguientes familias, hay cambios significativos:
  - 4 Tensor Cores por SM
  - Mejoras importantes en el rendimiento
  - Nuevos tipos de datos: FP64, TF32, bfloat16, FP16, FP8, INT8
  - Ya vamos por la 4ª generación de Tensor Cores

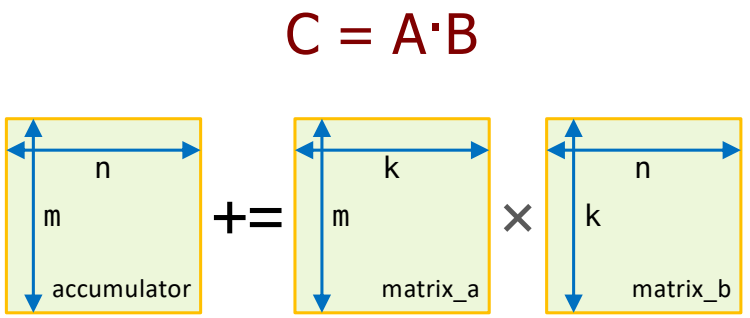
Tensor Cores

- Se pueden utilizar muy fácilmente con librerías:
  - TensorRT, cuDNN, cuBLAS
  - Los rendimientos con estas librerías obtienen rendimientos espectaculares
- También se pueden programar directamente en CUDA
  - No existen instrucciones específicas para un thread.
  - Existen funciones matriciales a nivel de warp a través de `mma.h`
  - **Todos los threads de un warp han de trabajar juntos para operar con los Tensor Cores**
  - **Todos los threads de un warp han de ejecutar la misma instrucción** `nvcuda::wmma`
- La información que hay en los manuales de CUDA es muy escasa
  - La palabra “provisional” aparece muchas veces en la documentación.
  - Nadie garantiza, que las cosas no puedan cambiar en el futuro.
  - La mejor forma de informarse es con ejemplos y cursos (de Nvidia)

Tensor Cores

- Un Tensor Core ejecuta 64 operaciones FMA (a=a+b×c), de coma flotante, por ciclo.
  - En una RTX 3080, tenemos 4 Tensor Cores por SM.
  - Podemos hacer 2×64×4 = 512 ops en coma flotante por ciclo en cada SM con los Tensor Cores.
- Los Tensor Cores soportan una gran variedad de tipos de datos y posibles tamaños de matriz. Algunos ejemplos:

matrix_a	matrix_b	accumulator	size (m × n × k)
_half	_half	float	16×16×16
_half	_half	float	32×8×16
_half	_half	_half	16×16×16
double	double	double	8×8×4
__nv_bfloat16	__nv_bfloat16	float	8×32×16



  - Consultad el apartado "Element Types and Matrix Sizes" del "CUDA C++ Programming Guide".
- Los warps utilizan los Tensor Cores de forma colaborativa para procesar fragmentos de matriz

MxM con Tensor Cores

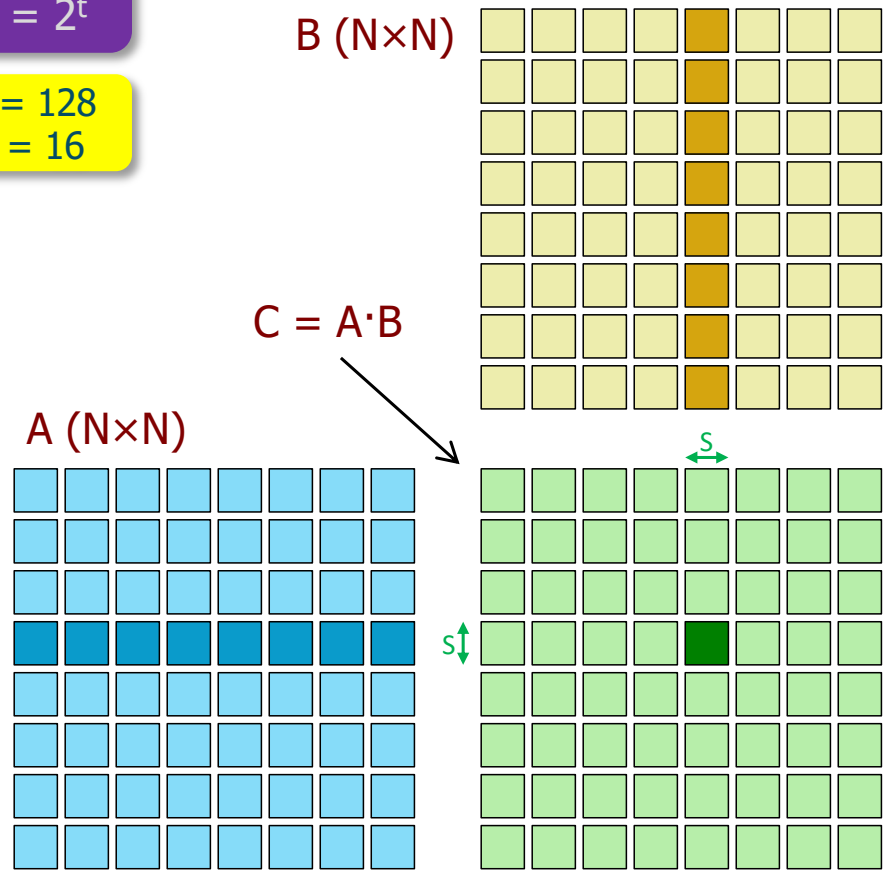
```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      C[i][j] += A[i][k] * B[k][j];
```

```
for (pi=0; i<N; pi+=S)
  for (pj=0; j<N; pj+=S)
    for (pk=0; k<N; pk+=S)
      for (i=pi; i<pi+S; i++)
        for (j=pi; j<pj+S; j++)
          for (k=pi; k<pk+S; k++)
            C[i][j] += A[i][k] * B[k][j];
```

```
for (pi=0; i<N; pi+=S)
  for (pj=0; j<N; pj+=S)
    for (pk=0; k<N; pk+=S)
      MxM<S,S>(A, B, C, pi, pj, pk);
```

$N = 2^m$
$S = 2^t$

N = 128
S = 16



MxM con Tensor Cores

- https://cuda-tutorial.github.io/
- Matrices cuadradas N×N
- $N = 2^m$
- Cada **WARP** calculará una porción de 16×16 elementos de la matriz resultado.

Matrix A	Matrix B	Accumulator	Size (m × n × k)
_half	_half	float	16×16×16

- Usaremos 256 threads

```
unsigned int nThreads = 256;
unsigned int warps_required = (N * N) / (16 * 16);
unsigned int warps_per_block = nThreads / 32;
unsigned int blocks_required = warps_required / warps_per_block;

// Invocación del Kernel
MxMTensor<DIM><<<blocks_required, nThreads>>>(dAptr, dBptr, dCptr);
```

MxM con Tensor Cores

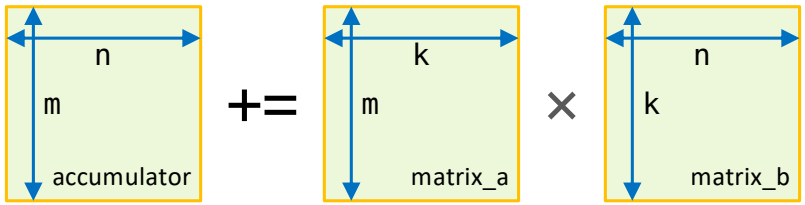
```
template <unsigned int DIM>
__global__ void MxMTensor(half *A, half *B, float *C) {

  using namespace nvcuda;
  // Declaración de los fragmentos de datos con los que vamos a operar
  wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a_frag;
  wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
  wmma::fragment<wmma::accumulator, 16, 16, 16, float> acc_frag;
  // Inicializa a 0.0 el fragmento "acumulador"
  wmma::fill_fragment(acc_frag, 0.0f);

  int warpID = (blockIdx.x * blockDim.x + threadIdx.x) / 32;
  int aRow = 16 * (warpID / (DIM / 16));
  int bCol = 16 * (warpID % (DIM / 16));
  for (int i = 0; i < DIM; i += 16) {
    // Load colaborativo de los fragmentos de A y B a procesar
    wmma::load_matrix_sync(a_frag, A + i + aRow * DIM, DIM);
    wmma::load_matrix_sync(b_frag, B + bCol + i * DIM, DIM);
    // Producto de matrices en los Tensor Cores
    wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
  }
  // Al acabar se guarda el fragmento de matriz resultado que se ha calculado en este bloque
  wmma::store_matrix_sync(C + bCol + aRow * DIM, acc_frag, DIM, wmma::mem_row_major);
}
```

MxM con Tensor Cores

```
...
  using namespace nvcuda;
  // Declaración de los fragmentos de datos con los que vamos a operar
  wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a_frag;
  wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
  wmma::fragment<wmma::accumulator, 16, 16, 16, float> acc_frag;
  // Inicializa a 0.0 el fragmento "acumulador"
  wmma::fill_fragment(acc_frag, 0.0f);
...
```

```
template<typename Use, int m, int n, int k, typename T, typename Layout=void> class fragment;
```

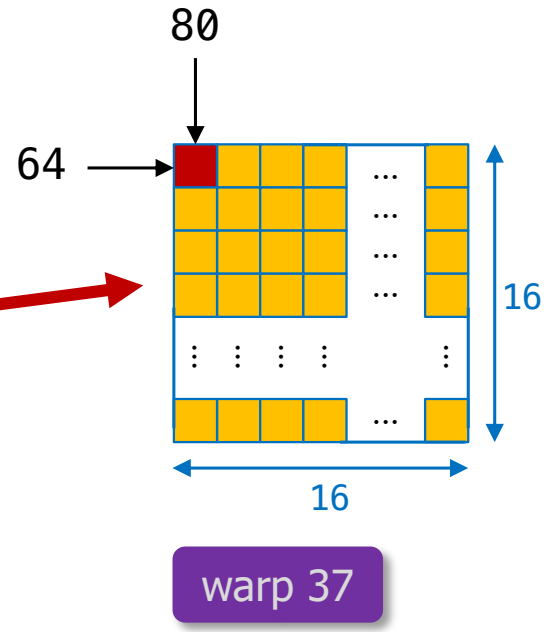
- Use: matrix_a, matrix_b, accumulator
- T: half, float, ...
- Layout: row_major, col_major



MxM con Tensor Cores

```
...
  int warpID = (blockIdx.x * blockDim.x + threadIdx.x) / 32;
  int aRow = 16 * (warpID / (DIM / 16));
  int bCol = 16 * (warpID % (DIM / 16));
...
```

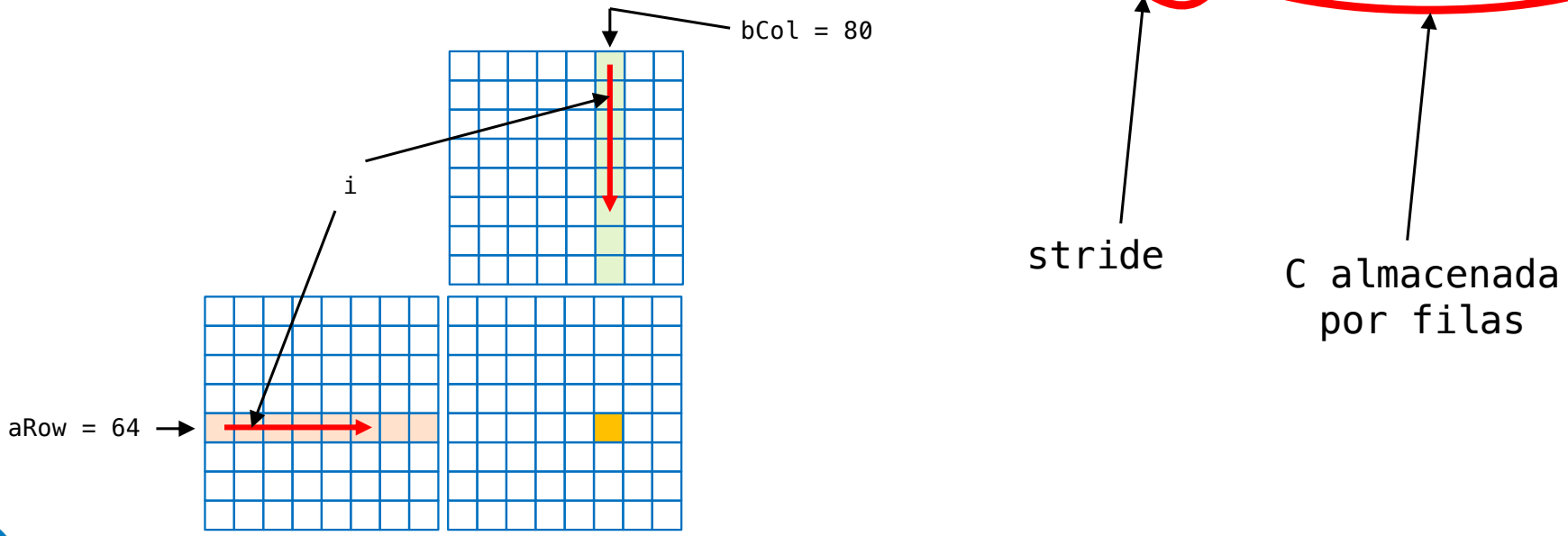
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63



matriz C (128×128)
DIM = 128
DIM/16 = 8
37/8 = 4
37%8 = 5
aRow = 4∗16 = 64
bCol = 5∗16 = 80

MxM con Tensor Cores

```
...
  for (int i = 0; i < DIM; i += 16) {
    // Load colaborativo de los fragmentos de A y B a procesar
    wmma::load_matrix_sync(a_frag, A + i + aRow * DIM, DIM);
    wmma::load_matrix_sync(b_frag, B + bCol + i * DIM, DIM);
    // Producto de matrices en los Tensor Cores
    wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
  }
  wmma::store_matrix_sync(C + bCol + aRow * DIM, acc_frag, DIM, wmma::mem_row_major);
```



MxM con Tensor Cores

```
template <unsigned int DIM>
__global__ void MxMTensor(half *A, half *B, float *C) {

  using namespace nvcuda;
  // Declaración de los fragmentos de datos con los que vamos a operar
  wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a_frag;
  wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
  wmma::fragment<wmma::accumulator, 16, 16, 16, float> acc_frag;
  // Inicializa a 0.0 el fragmento "acumulador"
  wmma::fill_fragment(acc_frag, 0.0f);

  int warpID = (blockIdx.x * blockDim.x + threadIdx.x) / 32;
  int aRow = 16 * (warpID / (DIM / 16));
  int bCol = 16 * (warpID % (DIM / 16));
  for (int i = 0; i < DIM; i += 16) {
    // Load colaborativo de los fragmentos de A y B a procesar
    wmma::load_matrix_sync(a_frag, A + i + aRow * DIM, DIM);
    wmma::load_matrix_sync(b_frag, B + bCol + i * DIM, DIM);
    // Producto de matrices en los Tensor Cores
    wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
  }
  // Al acabar se guarda el fragmento de matriz resultado que se ha calculado en este bloque
  wmma::store_matrix_sync(C + bCol + aRow * DIM, acc_frag, DIM, wmma::mem_row_major);
}
```

Evaluación

- **MxM CUDA**, versión a bloques usando Shared Memory
- **MxM Tensor**
- **cuBLAS Pedantic**, versión robusta de MxM en la librería cuBLAS
- **cuBLAS Tensor**, versión de MxM en la librería cuBLAS usando Tensor Cores

DIM	1.024	2.048	4.096	8.192
MxM CUDA	2,34 TFLOPs	2,40 TFLOPs	2,41 TFLOPs	2,54 TFLOPs
MxM Tensor	**12,26 TFLOPs**	**12,70 TFLOPs**	**13,29 TFLOPs**	**14,97 TFLOPs**
cuBLAS Pedantic	4,01 TFLOPs	12,15 TFLOPs	14,98 TFLOPs	11,91 TFLOPs
cuBLAS Tensor	26,89 TFLOPs	42,91 TFLOPs	53,49 TFLOPs	57,61 TFLOPs

← Buen rendimiento

← Hay mucho margen de mejora

256 threads por bloque

Bibliografía & Documentación

- www.nvidia.com
- Ejemplo extraído de: https://cuda-tutorial.github.io/
  - MUCHOS EJEMPLOS ÚTILES.
- NVIDIA.
  CUDA C++ Programming Guide,
  version 12.2, Nvidia 2023 [LINK]
- cuBLAS.
  version 12.2, Nvidia 2023 [LINK]

- Manuales de Nvidia ACCESIBLES en https://docs.nvidia.com/cuda/
  - Para tener acceso total hay que registrarse.

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Tarjetas Gráficas y Aceleradores
CUDA – MxM con Tensor Cores

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya