Universidad Autónoma de Madrid

Campus Internacional excelencia UAM+CSIC

Escuela Politécnica Superior

Programación GP-GPU

Procesamiento de Datos a Gran Escala

Máster en Ciencia de Datos

Francisco J. Gómez Arribas

Agenda

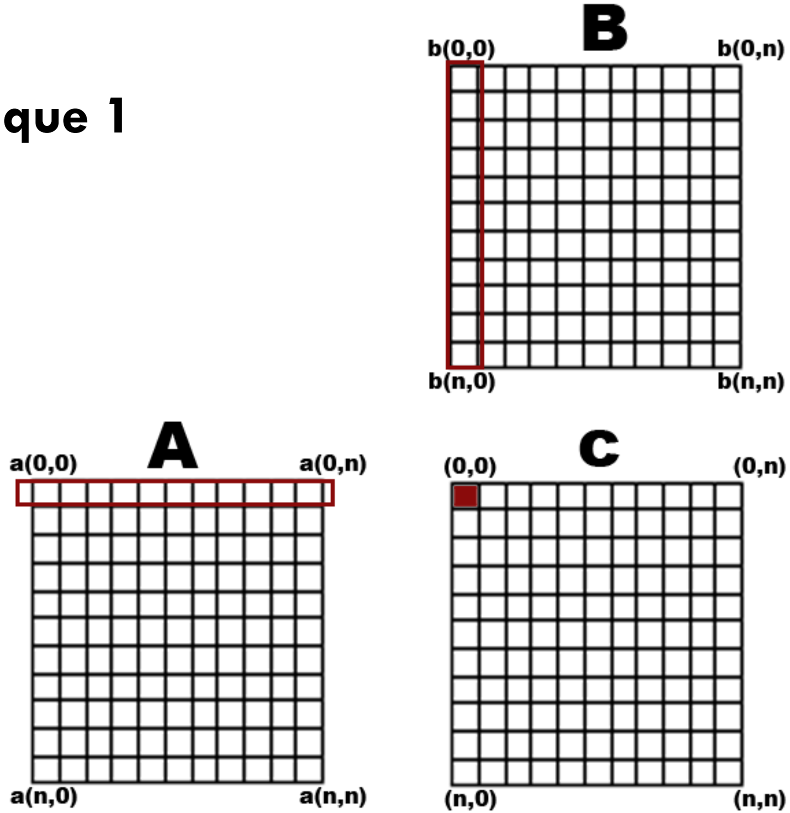
- MAT MUL on GPU
- Reduction on GPU
- CUDA Examples

Matrix-Matrix Operations

- **GPU Matrix Multiplication: Technique 1**

  Express multiplication of two matrices as dot product of vector of matrix row and columns

  Compute matrix C by:
  for each cell of cij take the dot product of row I of matrix A with column j of matrix B



CUDA: Matrix – Matrix Operations

- Mat Mult

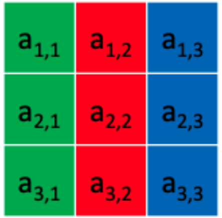
```
for (i = 0; i < n; ++i)
    for (j = 0; j < m; ++j)
        for (k = 0; k < p; ++k)
            a[i+n*j] += b[i+n*k] * c[k+p*j];
```

- Matrices are stored in column-major order

For reference, jki-ordered version runs at 1.7 GFLOPS on 3 GHz Intel Xeon (single core)

CUDA: Matrix – Matrix Operations

- Memory alignment for GPU




```
cudaMalloc(&dev_a, m*n*sizeof(float));
```

Matrix columns are not aligned at 64-bit boundary

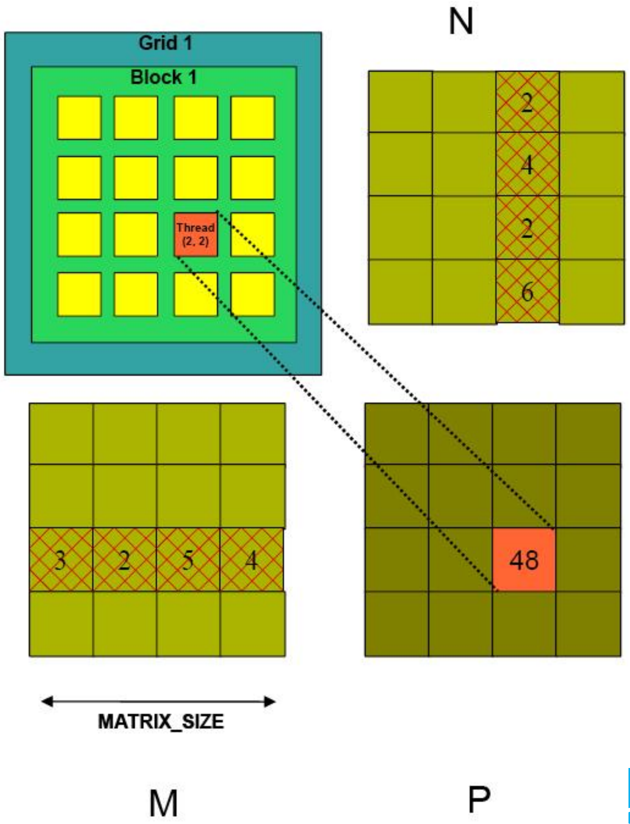
```
cudaMallocPitch(&dev_a, &n, n*sizeof(float), m);
```

Matrix columns are aligned at 64-bit boundary

n is the allocated (aligned) size for the first dimension (the pitch), given the requested sizes of the two dimensions.

CUDA: Matrix – Matrix Operations

- One Block of threads compute matrix P
  - Each thread computes one element of P
- Each thread
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block

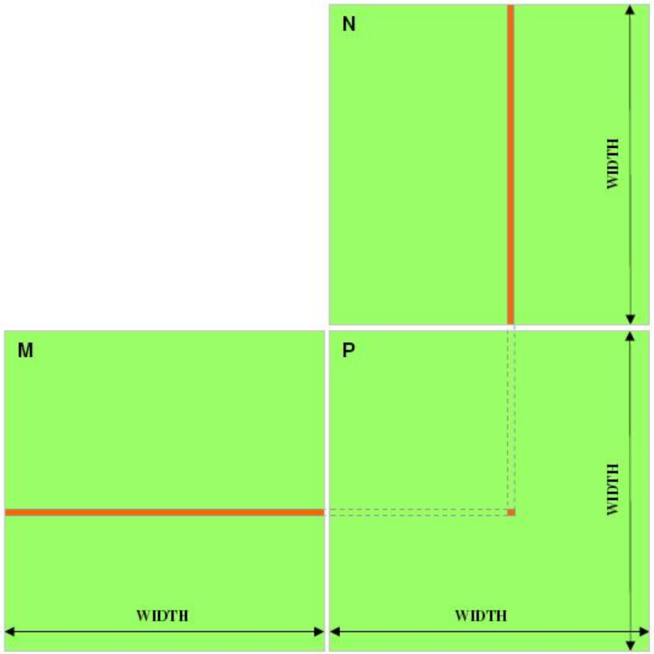


CUDA: Matrix – Matrix Operations

- P=M*N of size

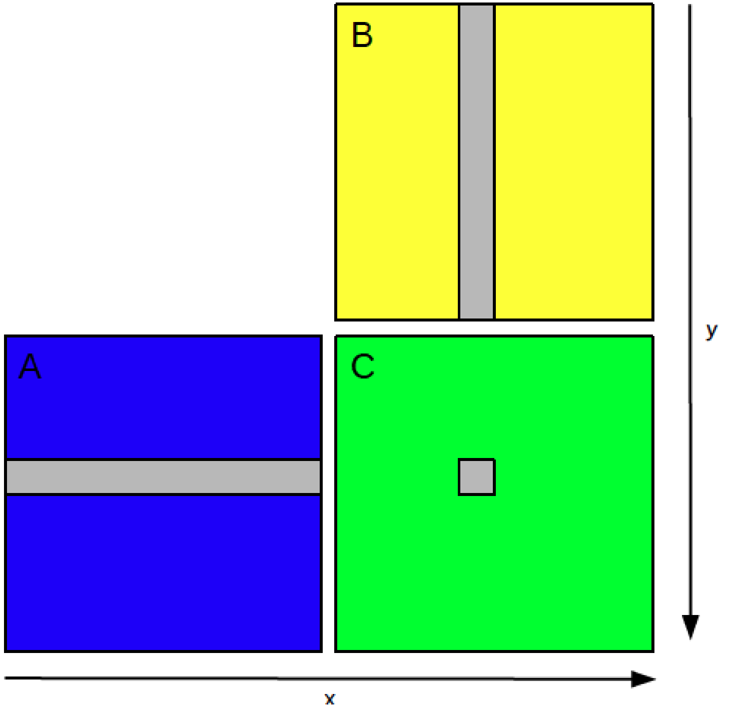
  WIDTHxWIDTH

- Without blocking:
  - One thread handles one element of P
  - M and N are loaded WIDTH times from global memory



CUDA: Matrix – Matrix Operations

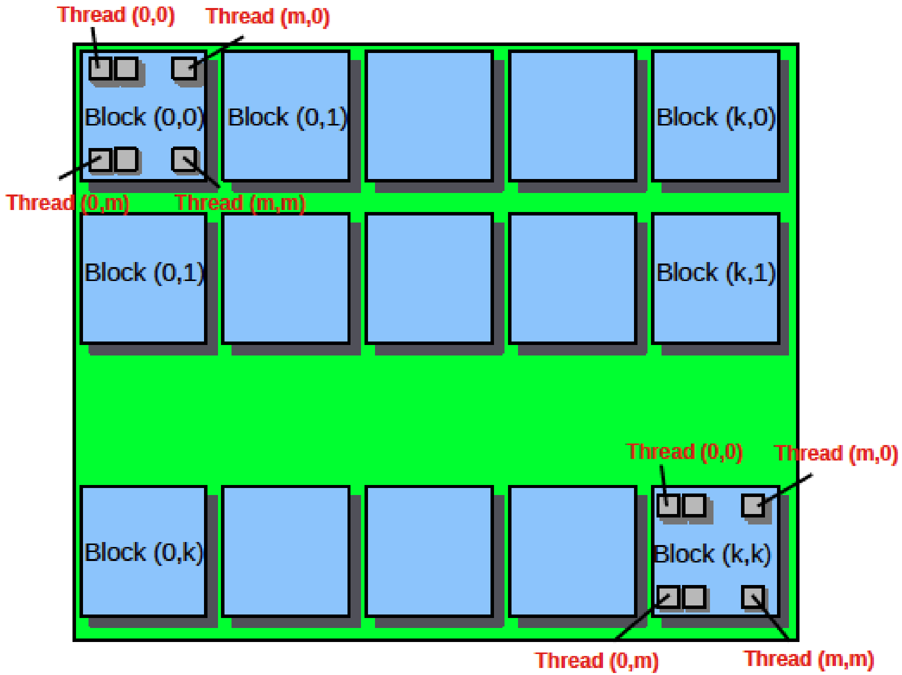
- Work distribution using BLOCKS



- Each thread computes one element of the result matrix C
- n * n threads will be needed
- Indexing of threads corresponds to 2d indexing of the matrices
- Thread(x, y) will calculate element C(x, y) using row y of A and column x of B

CUDA: Matrix – Matrix Operations

Distribution of work



Result matrix C (n * n elements)

- Use 2d execution grid with k * k blocks
- Use 2d thread blocks with fixed block size (m * m)
- k = n / m (n divisible by m)
- k = n / m + 1 (n not divisible by m)

CUDA: Matrix – Matrix Operations

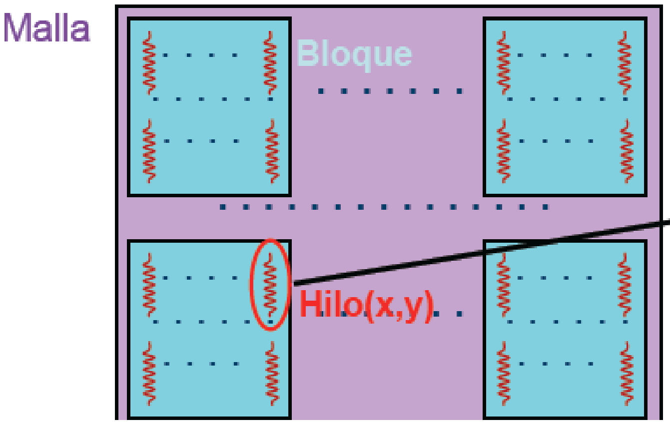
Descripción de la paralelización

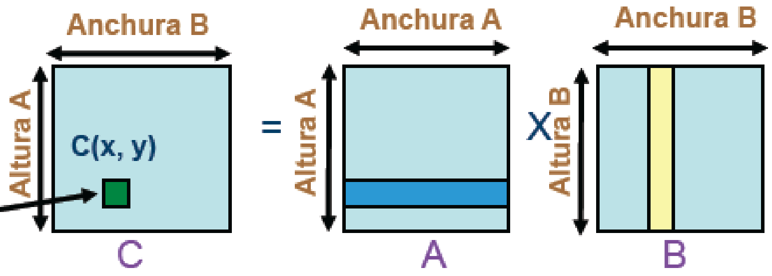
Cada hilo computa un elemento de la matriz resultado C.

- Las matrices A y B se cargan N veces desde memoria de vídeo.

Los bloques acomodan los hilos en grupos de 1024 (limitación interna en arquitecturas Fermi y Kepler).

- Así podemos usar bloques 2D de 32x32 hilos cada uno.






```
dim2 dimBlock(BLOCKSIZE, BLOCKSIZE);
dim2 dimGrid(AnchuraB/BLOCKSIZE, AlturaA/BLOCKSIZE);
...
MxMonGPU <<<dimGrid,dimBlock>>> (A, B, C, N);
```

CUDA: Matrix – Matrix Operations

Kernel (CUDA)

Kernel function

```
__global__ void mm_kernel(float* A, float* B, float* C, int n)
{
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < n && col < n) {
      for (int i = 0; i < n; ++i) {
         C[row * n + col] += A[row * n + i] * B[i * n + col];
      }
    }
}

mm_kernel<<<dimGrid, dimBlock>>> (d_a, d_b, d_c, n);
```

Cada hilo utiliza 10 registros, lo que nos permite alcanzar el mayor grado de paralelismo en Kepler:

- 2 bloques de 1024 hilos (32x32) en cada SMX. [2x1024x10 =20480 registros, que es inferior a la cota de 65536 regs. disponibles].

CUDA: Matrix – Matrix Operations

Limiting Factor

Kernel function

```
void mm_kernel ( float* A, float* B, float* C,int n )
{
    for (int k = 0; k < n; ++k){
        C[i * n + j] += A[i * n + k] * B[k * n + j];
    }
}
```

- One floating point operation per memory access
- One double: 8 bytes
- Global memory bandwidth: ~240 GB/s

Limitaciones:

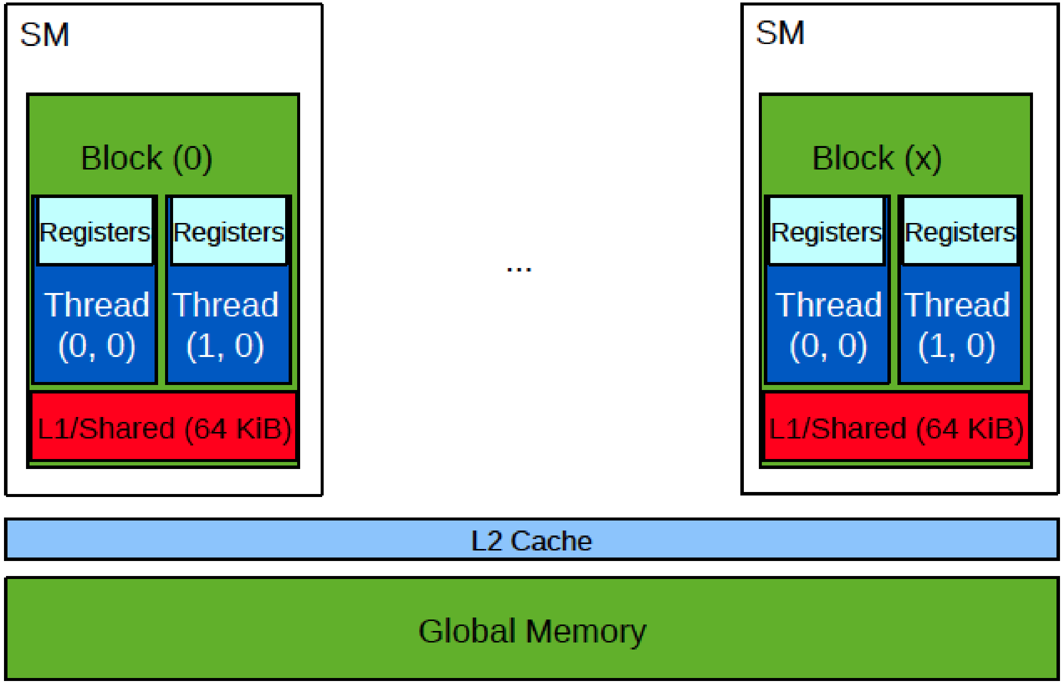
- Baja intensidad aritmética.
- Exigente en el ancho de banda a memoria, que termina siendo el cuello de botella para el rendimiento.

Solución:

- Utilizar la memoria compartida de cada multiprocesador.

CUDA: Matrix – Matrix Operations

Global Memory vs Shared Memory



CUDA: Matrix – Matrix Operations

Using Shared Memory

Allocate shared memory

```
// allocate vector in shared memory
__shared__ float[size];

// can also define multi-dimensional arrays:
// BLOCK_SIZE is length (and width) of a thread block here
__shared__ float Msub[BLOCK_SIZE][BLOCK_SIZE];
```

Copy data to shared memory

```
// fetch data from global to shared memory
Msub[threadIdx.y][threadIdx.x] = M[TidY * width + TidX];
```

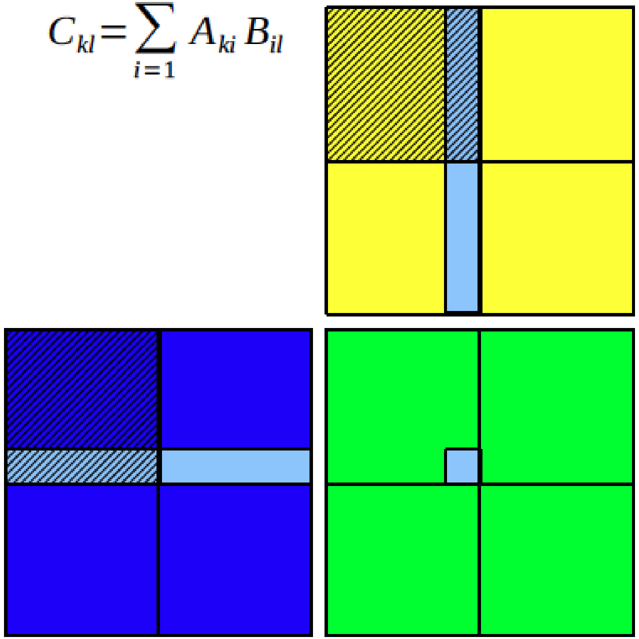
Synchronize threads

```
// ensure that all threads within a block had time to read / write data
__syncthreads();
```

CUDA: Matrix – Matrix Operations

Matrix-matrix multiplication with blocks

$$C_{kl} = \sum_{i=1}^{N} A_{ki} B_{il}$$

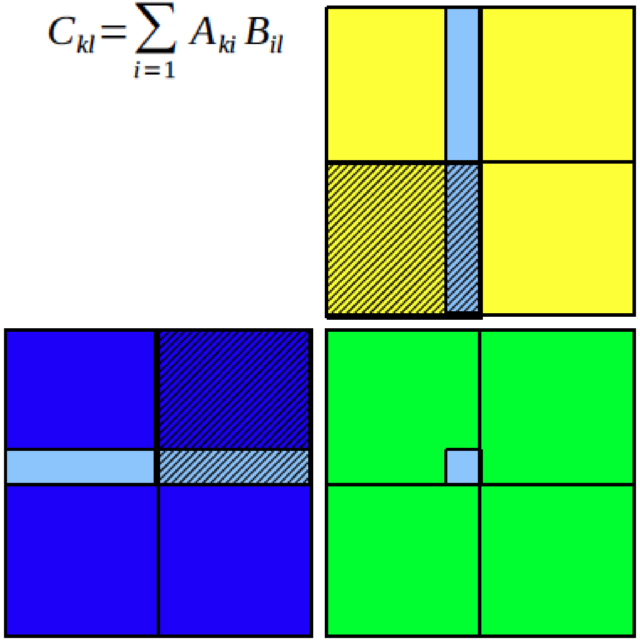


$$C_{kl} = \sum_{i=1}^{N/2} A_{ki} B_{il} + \sum_{i=N/2+1}^{N} A_{ki} B_{il}$$

CUDA: Matrix – Matrix Operations

Matrix-matrix multiplication with blocks

$$C_{kl} = \sum_{i=1}^{N} A_{ki} B_{il}$$



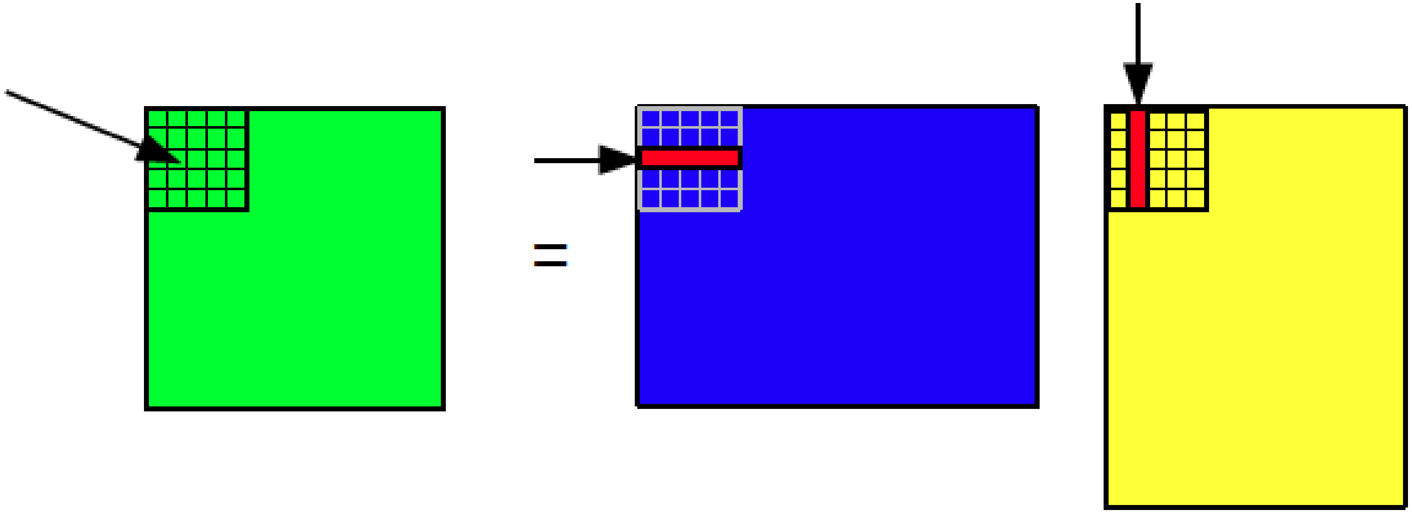
$$C_{kl} = \sum_{i=1}^{N/2} A_{ki} B_{il} + \sum_{i=N/2+1}^{N} A_{ki} B_{il}$$

For each element

- Set result to zero
- For each pair of blocks
  - *Copy data*
  - *Do partial sum*
  - *Add result of partial sum to total*

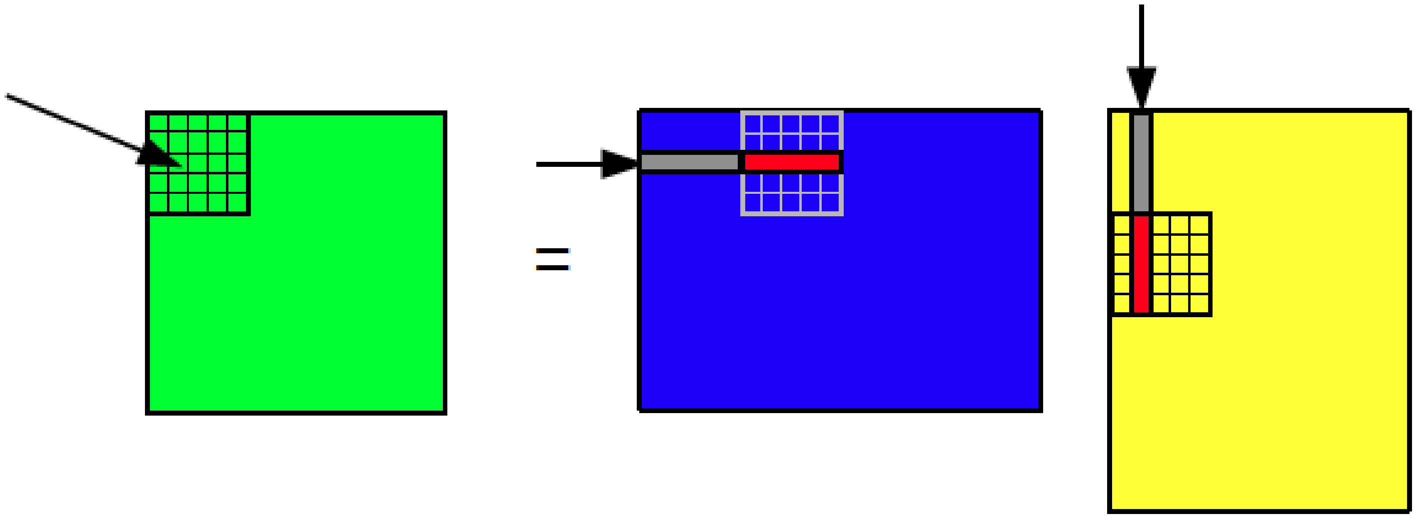
CUDA: Matrix – Matrix Operations

Matrix-matrix multiplication with blocks



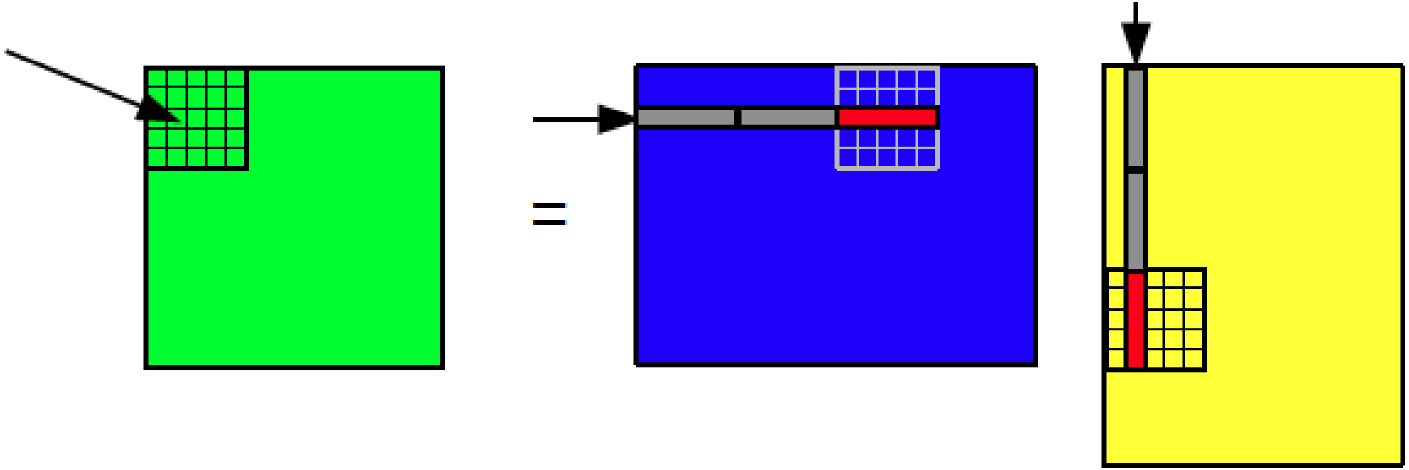
CUDA: Matrix – Matrix Operations

Matrix-matrix multiplication with blocks



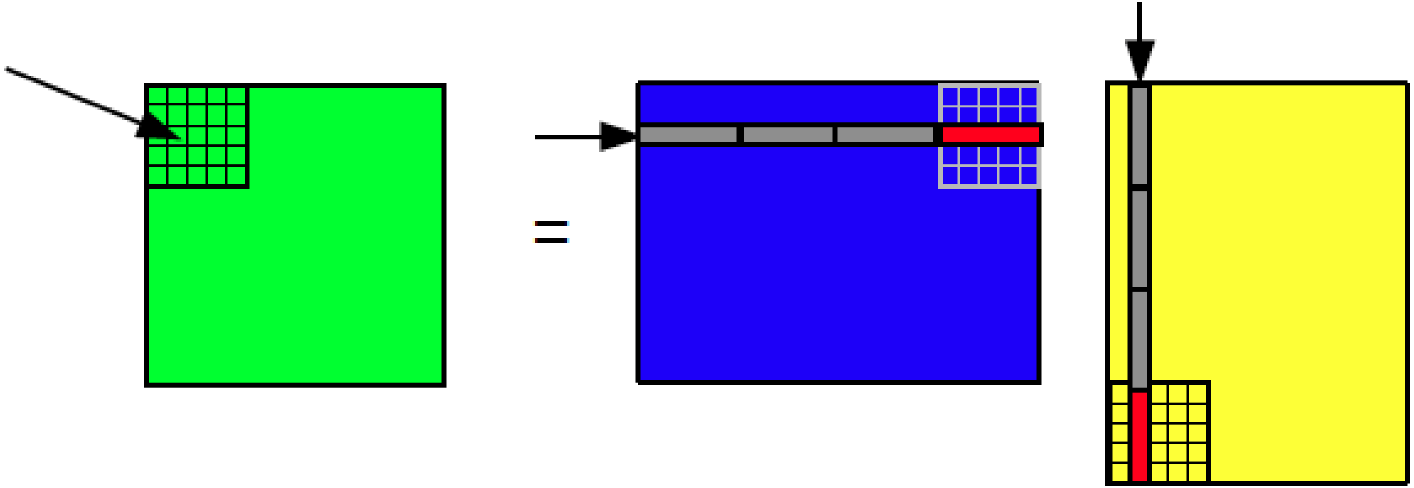
CUDA: Matrix – Matrix Operations

Matrix-matrix multiplication with blocks



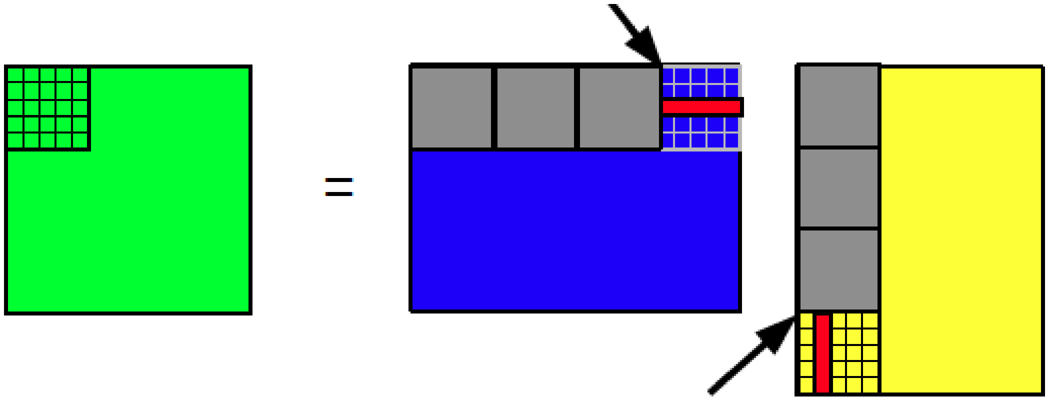
CUDA: Matrix – Matrix Operations

Matrix-matrix multiplication with blocks



CUDA: Matrix – Matrix Operations

Matrix-matrix multiplication with blocks

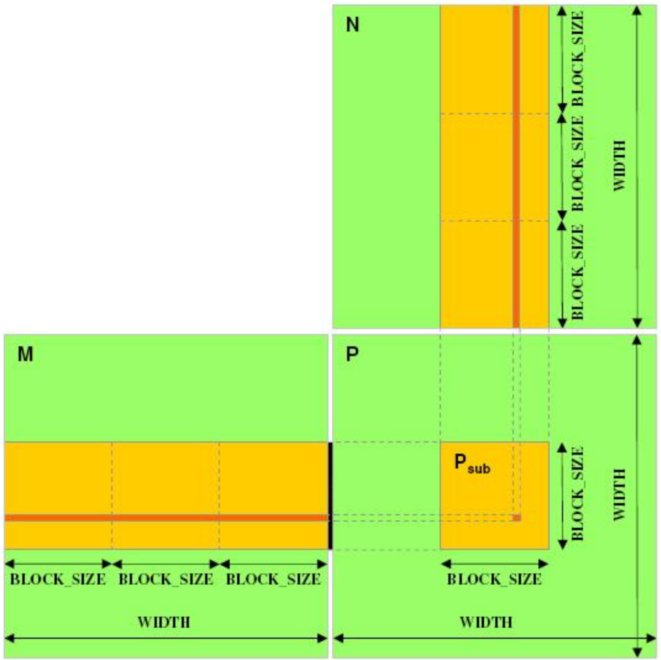


Thread block loops over blocks in blue and yellow matrix:

- Calculate upper left corner
- Load data into shared memory
- Do calculation (one thread is still responsible for an element)
- Add partial sum to resul

CUDA: Matrix – Matrix Operations

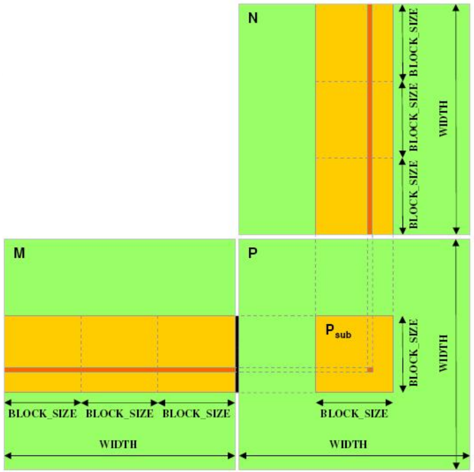
- La submatriz de C de 32x32 datos computada por cada bloque de hilos utiliza mosaicos de 32x32 elementos de Ay B que se alojan de forma reiterativa en memoria compartida.
- A y B se cargan sólo (N/32) veces desde memoria global.
- Logros:
  - Menos exigente en el ancho de banda a memoria.
  - Más intensidad aritmética.



CUDA: Matrix – Matrix Operations

```
__global__ void MxMonGPU(float *A, float *B, float *C, int N)
{
  int sum=0, tx, ty, i, j;
  tx = threadIdx.x;                    ty = threadIdx.y;
  i = blockIdx.x * blockDim.x + tx;    j = blockIdx.y * blockDim.y + ty;
  __shared__ float As[32][32], float Bs[32][32];

  // Recorre los mosaicos de A y B necesarios para computar la submatriz de C
  for (int tile=0; tile<(N/32); tile++)
  {
    // Carga los mosaicos (32x32) de A y B en paralelo (y de forma traspuesta)
    As[ty][tx]= A[(i*N) + (ty+(tile*32))];
    Bs[ty][tx]= B[((tx+(tile*32))*N) + j];
    __syncthreads();
    // Computa los resultados para la submatriz de C
    for (int k=0; k<32; k++) // Los datos también se leerán de forma traspuesta
      sum += As[k][tx] * Bs[ty][k];
    __syncthreads();
  }
  // Escribe en paralelo todos los resultados obtenidos por el bloque
  C[i*N+j] = sum;
}
```



CUDA: Matrix – Matrix Operations

Detalles de la implementación: Hay que gestionar todos los mosaicos de fila y columna que necesita cada bloque de hilos:

- Se cargan los mosaicos de entrada (A y B) desde memoria global a memoria compartida en paralelo (todos los hilos contribuyen). Estos mosaicos reutilizan el espacio de memoria compartida.
- __synchthreads() (para asegurarnos que hemos cargado las matrices completamente antes de comenzar la computación).
- Computar todos los productos y sumas para C utilizando los mosaicos de memoria compartida.
    - Cada hilo puede iterar independientemente sobre los elementos del mosaico.
- __synchthreads() (para asegurarnos que la computación con el mosaico ha acabado antes de cargar, en el mismo espacio de memoria compartida, dos nuevos mosaicos para A y B en la siguiente iteración.

CUDA: Matrix – Matrix Operations

Algunas características de los accesos en CUDA:

- La memoria compartida consta de 16 (pre-Fermi) ó 32 bancos.
- Los hilos de un bloque se enumeran en orden "column major", esto es, hilos consecutivos difieren en la dimensión x (no en la y).
  - Si accedemos de la forma habitual a los vectores en memoria compartida: As[threadIdx.x][threadIdx.y], los hilos de un mismo warp leerán de la misma columna, esto es, del mismo banco en memoria compartida.
  - En cambio, usando As[threadIdx.y][threadIdx.x], leerán de la misma fila, accediendo a un banco diferente.Por tanto, los mosaicos se almacenan y acceden en memoria compartida de forma invertida o traspuesta.

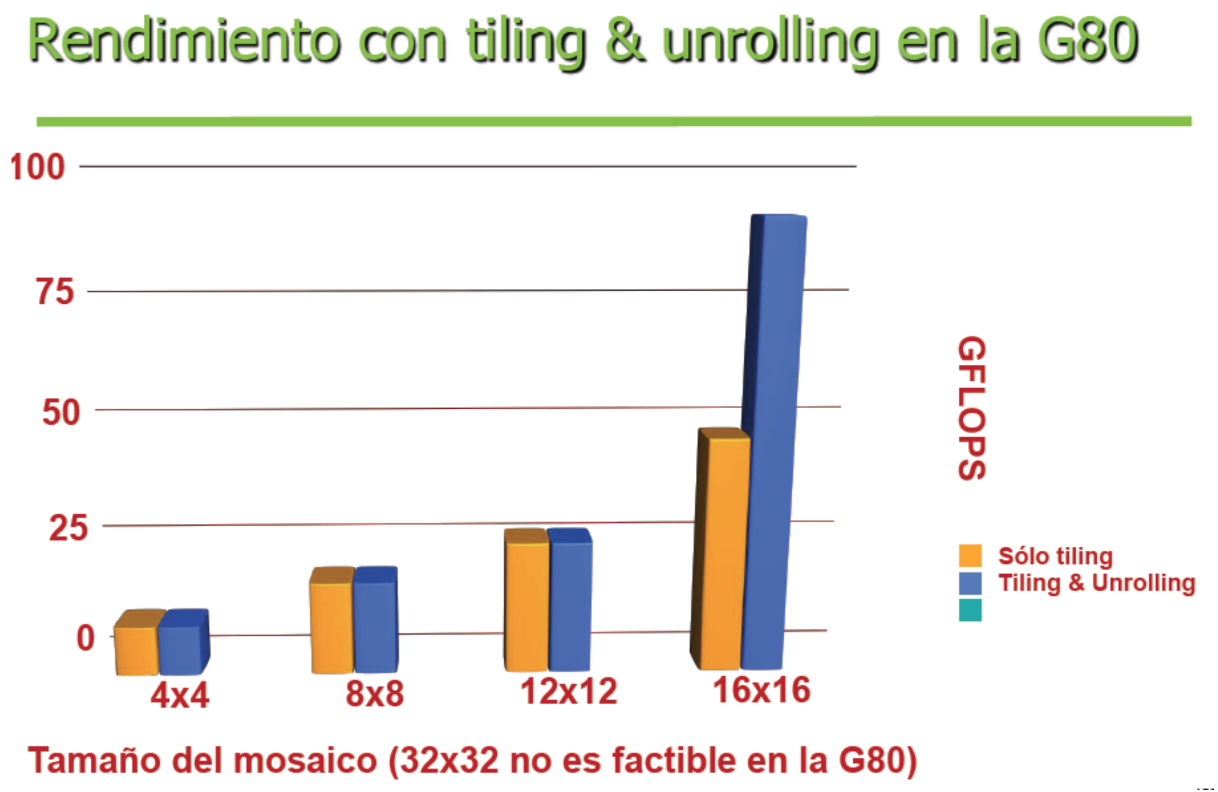
CUDA: Matrix – Matrix Operations

Optimización del compilador:
Desenrollado de bucles (loop unrolling)

Sin desenrollar el bucle	Desenrollando el bucle

`...`
`__syncthreads();`
 	
`// Computar la parte de ese mosaico`
`for (k=0; k<32; k++)`
`sum += As[tx][k]*Bs[k][ty];`
 	
`__syncthreads();`
`}`
`C[indexC] = sum;`
	`__syncthreads();`
	
	`// Computar la parte de ese mosaico`
	`sum += As[tx][0]*Bs[0][ty];`
	`sum += As[tx][1]*Bs[1][ty];`
	`sum += As[tx][2]*Bs[2][ty];`
	`sum += As[tx][3]*Bs[3][ty];`
	`sum += As[tx][4]*Bs[4][ty];`
	`sum += As[tx][5]*Bs[5][ty];`
	`sum += As[tx][6]*Bs[6][ty];`
	`sum += As[tx][7]*Bs[7][ty];`
	`sum += As[tx][8]*Bs[8][ty];`
	`...`
	`sum += As[tx][31]*Bs[31][ty];`
	`__syncthreads();`
	`}`
	`C[indexC] = sum;`

CUDA: Matrix – Matrix Operations

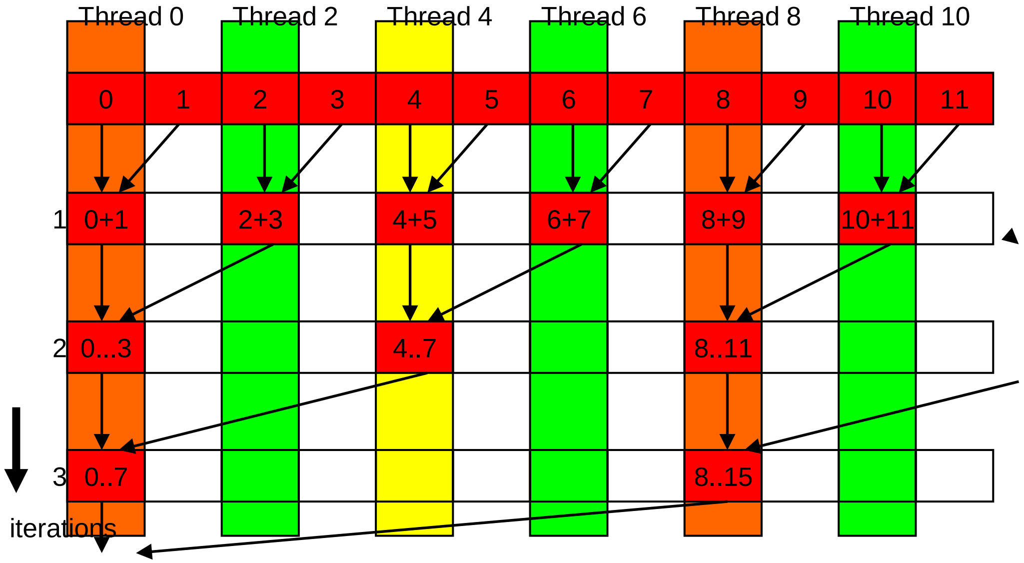


Agenda

- **MAT MUL on GPU**
- **Reduction on GPU**
- **CUDA Examples**

Vector Reduction

- Naïve mapping




```
__shared__ float partialSum[]
unsigned int t = threadIdx.x;

for (int stride = 1; stride < blockDim.x; stride *= 2) {

  __syncthreads();
  if (t % (2*stride) == 0)
    partialSum[t] += partialSum[t + stride];
}
```

Vector Reduction

- Divergence-free mapping