# Tema Seminario

## Programación manycore: resumen y ejercicios

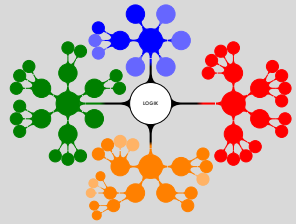
### Sistemas Concurrentes y Distribuidos (SCD)- Doble Grado Ingeniería Informática Matemáticas

Asignatura *de GIIM (3er curso)*

Fecha 28 septiembre 2023

Manuel I. Capel
manuelcapel@ugr.es
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada



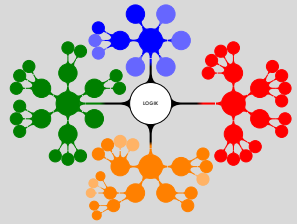


# Programación de aplicaciones paralelas en CUDA

1. El programador escribe un kernel `<<< ... >>>` y organiza su ejecución como un *grid* de `threads`:
   - Dividir el trabajo que ha de realizar el algoritmo en `threads`
   - Agrupar los `threads` en `thread-blocks`
   - Asignar los `thread-blocks` en *grids*



Manuel I. Capel
manuelcapel@ugr.es

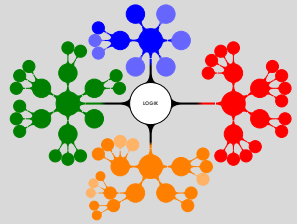


# Ejecución de los thread-blocks

1. Una vez que un bloque de `threads` se asigna a 1 SM, se le proporcionan recursos: Warps y memoria compartida
2. Una vez asignado, el Warp no puede migrar a otro SM
3. Una vez asignado un Warp se convierte en un *Warp* activo
4. Cada SM posee 2 planificadores de *Warp activos*
5. Cada Warp Scheduler es capaz de enviar dos instrucciones de Warp por ciclo de reloj a las unidades de ejecución
6. No es posible hacer una correspondencia entre `threads` y *cores*: la ejecución real de un `thread` es realizada por los cores de CUDA contenidos en el SM, pero no se puede decir cuál de ellos ejecuta cada `thread`



Manuel I. Capel
manuelcapel@ugr.es

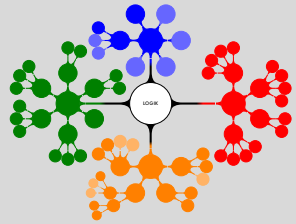


# Comportamiento de los SMs

- Cada SM divide sus propios bloques (hasta un máximo de 32 por SM) en Warps (cada Warp tiene un tamaño máximo de 32 `threads`)
- `WarpsPerBlock` = (`ThreadsPerBlock` + `WarpSize` − 1)/`WarpSize`
- No es necesario que los *Warp Schedulers* del SM seleccionen 2 Warps del mismo bloque de `threads`
- Los Warps pueden bloquearse si se programan sincronizaciones de barrera en los programas, en operaciones de memoria, dependencias de datos, etc.
- Un Warp bloqueado no es elegible para ser seleccionado por el Warp Scheduler del SM.
- Es, por tanto, aconsejable tener al menos 2 Warps elegibles para ejecución por ciclo



Manuel I. Capel
manuelcapel@ugr.es



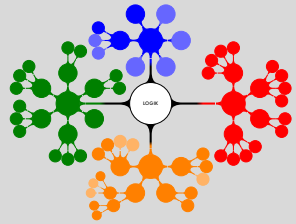
# Escenario 1

**Supongamos que creamos 8 bloques de 64 `threads` cada uno y cada SM tendrá 1 bloque para ejecutar, por lo que en la microarquitectura Pascal cada SM tendrá 64 cores CUDA que podrían asignarse para ejecutar los 64 hilos de cada bloque en paralelo.**

- ¿con el lanzamiento del kernel `<<< 8, 64 >>>` siempre se conseguirá que los 64 cores de cada SM se ejecuten?
- ¿En qué se traduce exactamente (bloques, Warps) la orden `<<< 8, 64 >>>` ?
- ¿Podría resultar que sólo se use un máximo de 32 cores en la ejecución del kernel anterior?



Manuel I. Capel
manuelcapel@ugr.es



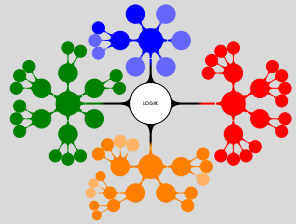
# Escenario 2

**¿Hay alguna diferencia si se lanza un kernel de 64 bloques de 8 `threads` en lugar de otro kernel con 8 bloques de 64 hilos (suponiendo que se distribuirán uniformemente entre los SM)?**

- Calcular cuántas instrucciones se ejecutan en cada caso suponiendo que todos los `threads` ejecutan exactamente 10 instrucciones (El hardware no juntará `theads` de diferentes warps)



Manuel I. Capel
manuelcapel@ugr.es



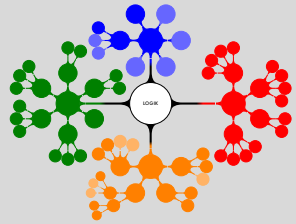
# Escenario 3

**Si intentamos ocupar completamente la GPU con trabajo planificado lanzando un kernel: `<<< 1024,1024 >>>` (creando 1024 bloques de 1024 `threads` cada uno), ¿es razonable suponer que todos los cores CUDA de la GPU se utilizarán en cierto momento y realizarán los mismos cálculos (suponiendo que los `threads` nunca se atasquen durante toda la ejecución)?**

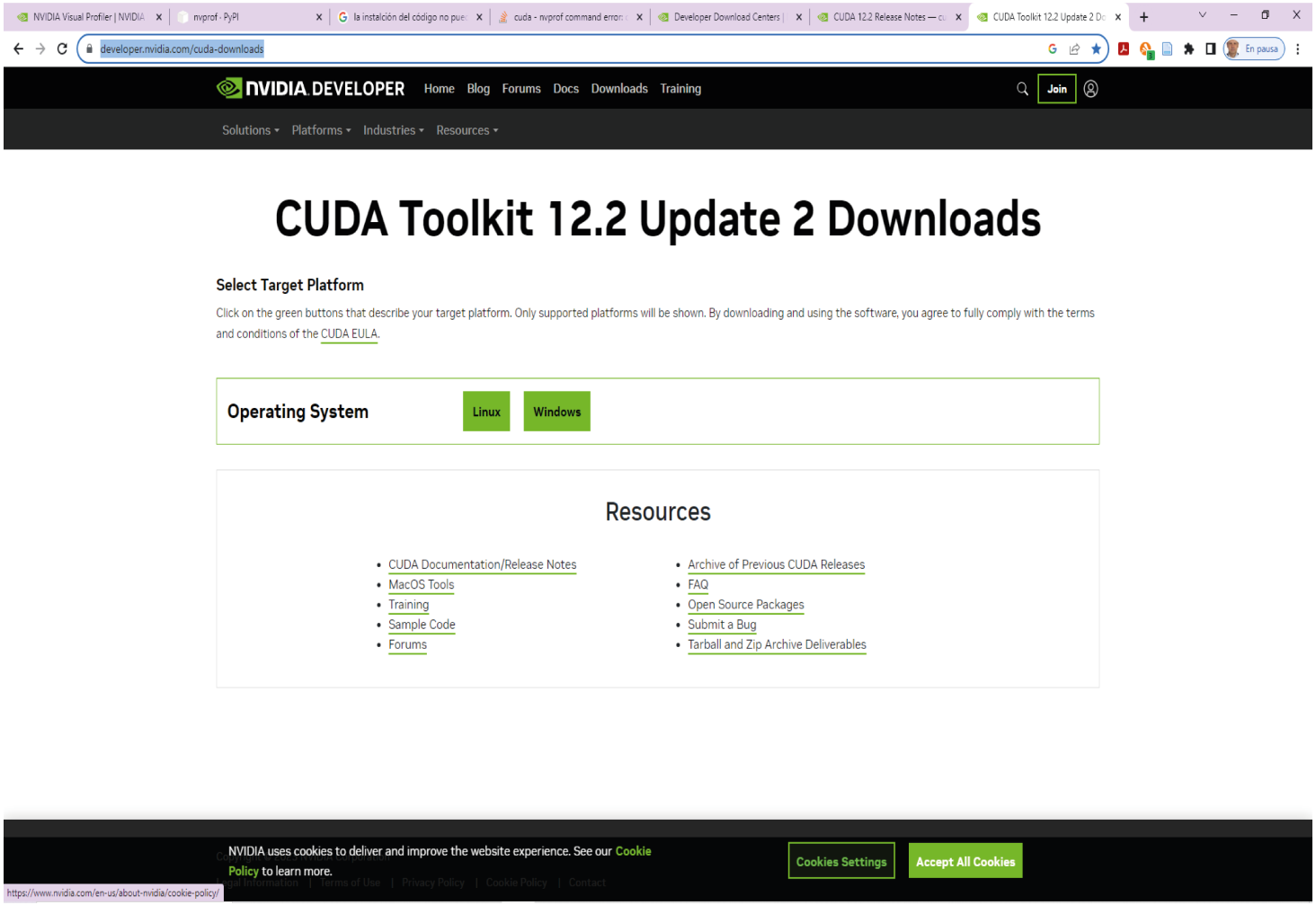
- Nota: El número máximo de warps activos es 64 (=máximo número de Warps que pueden ser planificados para ejecutarse en el siguiente ciclo) por SM, en GPUs nVIDIA con capacidad de cálculo 6.0, y este número corresponde a: 2048 = 64 * 32 `threads` en paralelo.



Manuel I. Capel
manuelcapel@ugr.es

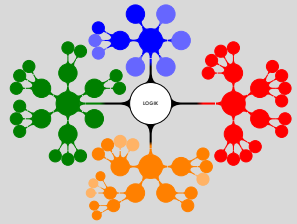


# Ejecución de los ejemplos






Manuel I. Capel
manuelcapel@ugr.es

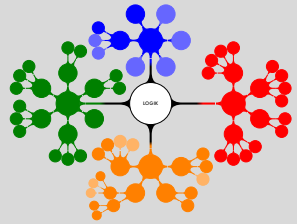


# Ejemplo1: 1 bloque, 1 thread

```
...
int main()
{
  int N = 1 << 20;
  float *x, *y;
        //Asigna Memoria Unificada
  cudaMallocManaged(&x, N * sizeof(float));
  cudaMallocManaged(&y, N * sizeof(float));
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
  add << <1, 1 >> > (N, x, y);
  cudaDeviceSynchronize();

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i] - 3.0f));
  std::cout << "Max error: " << maxError << std::endl;
  cudaFree(x);
  cudaFree(y);
  return 0;
}
```



Manuel I. Capel
manuelcapel@ugr.es

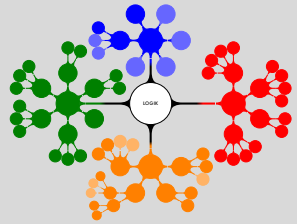


# Ejemplo2: 4096 bloques, 256 threads

```
...
int main(){
  int N = 1 << 20; //1048576 elementos
  float *x, *y;
  int blockSize= 256;
  int numBlocks = (N + blockSize - 1) / blockSize;

  cudaMallocManaged(&x, N * sizeof(float));
  cudaMallocManaged(&y, N * sizeof(float));

  for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
  }

  add << <numBlocks, blockSize >> > (N, x, y);

  cudaDeviceSynchronize();
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
  maxError = fmax(maxError, fabs(y[i] - 3.0f));
  std::cout << "Max error: " << maxError << std::endl;
  cudaFree(x);
  cudaFree(y);
  return 0;
}
```



Manuel I. Capel
manuelcapel@ugr.es



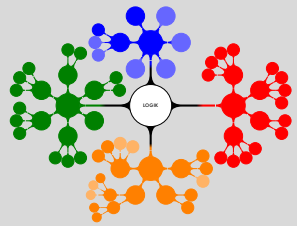
# Ejecución y resultadops en Windows 11 /64x

```
nvcc vectors1.cu -ccbin "C:\Program Files (x86)\Microsoft
    Visual Studio\2019\BuildTools\VC\Tools\MSVC
    \14.29.30133\bin\Hostx64\x64" --machine 64 -o
    add_vectors1.exe
```

```
nvprof add_vectors1.exe

Type  Time(%)      Time     Calls       Avg       Min
          Max  Name
GPU activities:  100.00%  190.05ms      1  190.05ms  190.05
    ms  190.05ms  add(int, float*, float*)
API calls:   56.86%  190.14ms      1  190.14ms  190.14ms
    190.14ms  cudaDeviceSynchronize
                 35.84%  119.86ms      2  59.931ms
                     1.9278ms  117.94ms
                     cudaMallocManaged
```



Manuel I. Capel
manuelcapel@ugr.es



# Ejecución y resultadops en Windows 11 /64x

```
nvcc vectors3.cu -ccbin "C:\Program Files (x86)\Microsoft
    Visual Studio\2019\BuildTools\VC\Tools\MSVC
    \14.29.30133\bin\Hostx64\x64" --machine 64 -o
    add_vectors3.exe
```

```
nvprof add_vectors3.exe

Type  Time(%)      Time     Calls       Avg       Min
          Max  Name
GPU activities:  100.00%  130.12us         1  130.12us
    130.12us  130.12us  add(int, float*, float*)
API calls:   73.11%  125.53ms         2  62.766ms  2.5588
    ms  122.97ms  cudaMallocManaged
                 13.50%  23.174ms         1  23.174ms
                     23.174ms  23.174ms
                     cuLibraryLoadData
```



Manuel I. Capel
manuelcapel@ugr.es