Gpu como poder de cálculo

Nancy Hitschfeld Kahler
(nancy@dcc.uchile.cl)

Departamento de Ciencias de la Computación
Universidad de Chile

Outline

- Introduction
- Conceptos básicos
- Tipo de problemas: *data parallel or/and task parallel*
- *Data parallelism* y GPU
- GPU *mapping*
- Primeros pasos: programando en Cuda
- Ejemplos

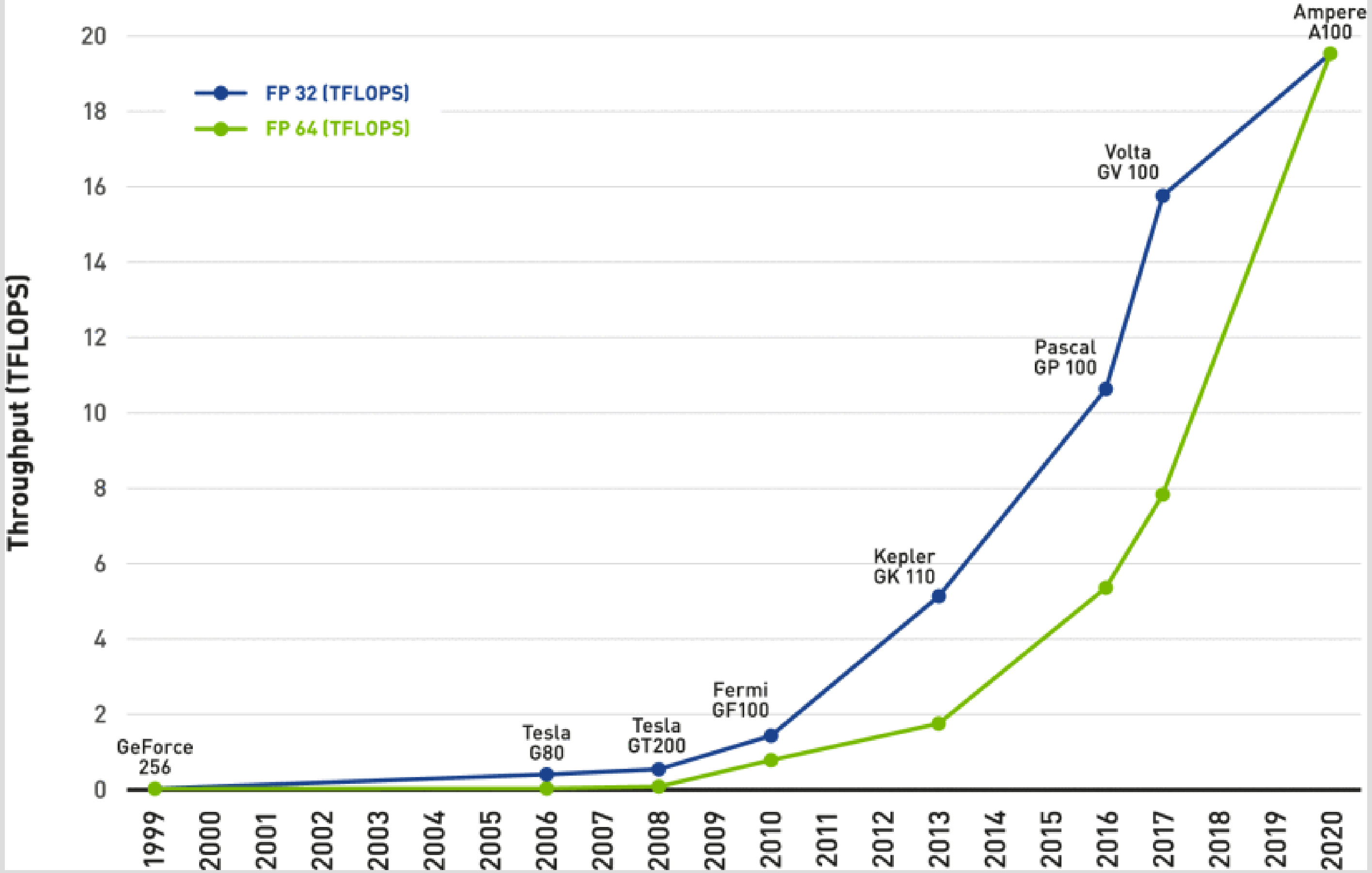
CA Navarro, N Hitschfeld-Kahler, L Mateu
A survey on parallel computing and its applications in data-parallel problems using GPU architectures
Communications in Computational Physics 15 (02), 285-329. 2014

Introducción

- The scientific community became interested in the power of GPUs (GPGPU)
  - Its low cost compared to other solutions (clusters, super-computers)
    - In 2002, McCool et al. published a paper detailing a meta-programming GPGPU language, named Sh
    - In 2004, Buck et al. proposed Brook for GPUs
    - In 2006, Nvidia proposed CUDA (Compute Unified Device Architecture)
    - In 2008, an open standard was released with the name of OpenCL (Open Computing Language), allowing the creation of multi-platform, massively parallel code

Introducción

- Evolution of the Graphics Processing Unit (GPU) William J. Dally et al. IEEE Micro >Volume: 41 Issue: 6



¿Qué tipo de problemas puede ser resuelto eficientemente con la GPU?

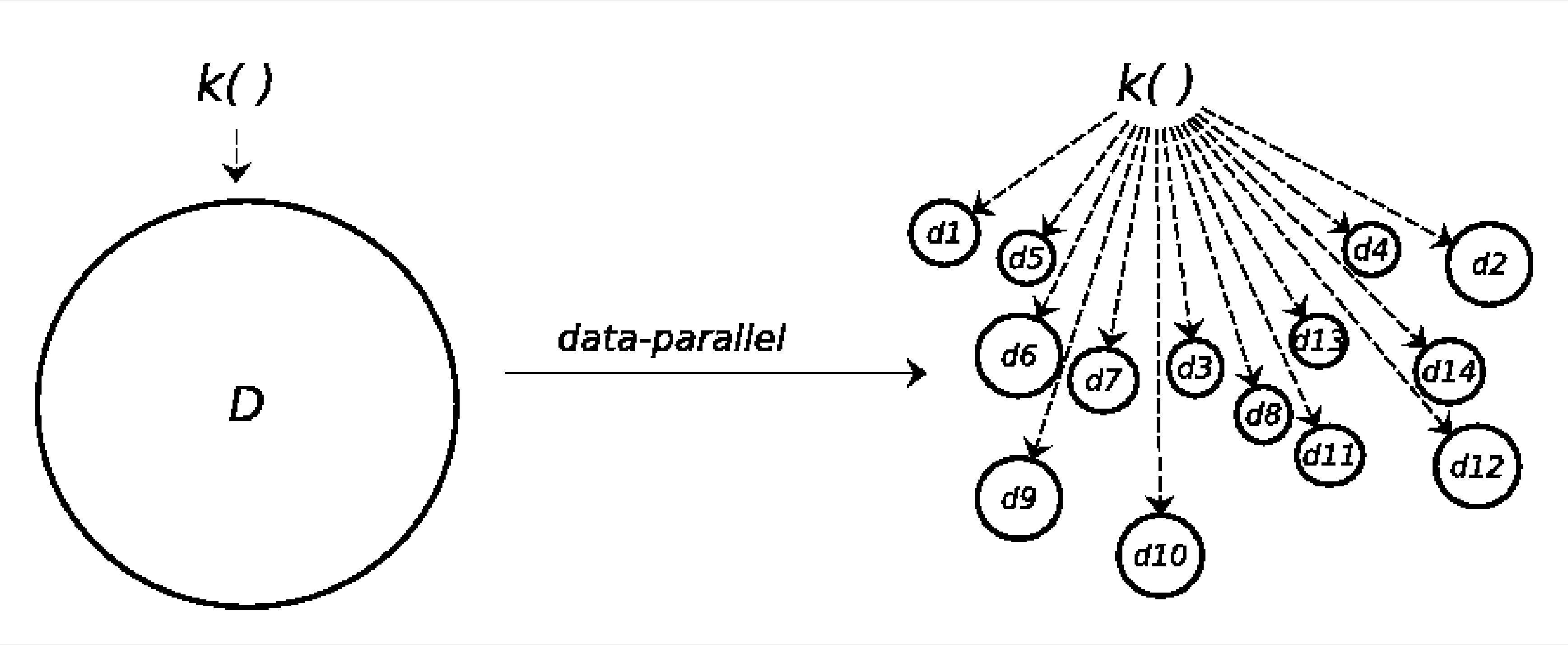
Conceptos básicos

- Parallel computing: the act of solving a problem of size n by dividing its domain into $l \geq 2$ (with $l \in N$) parts and solving them with p physical processors
- The identification of the type of problem is essential in the formulation of a parallel algorithm
- Let P_D be a problem with domain D.
  - Is P_D data-parallel?
  - Is P_D task parallel?

Basic Concepts

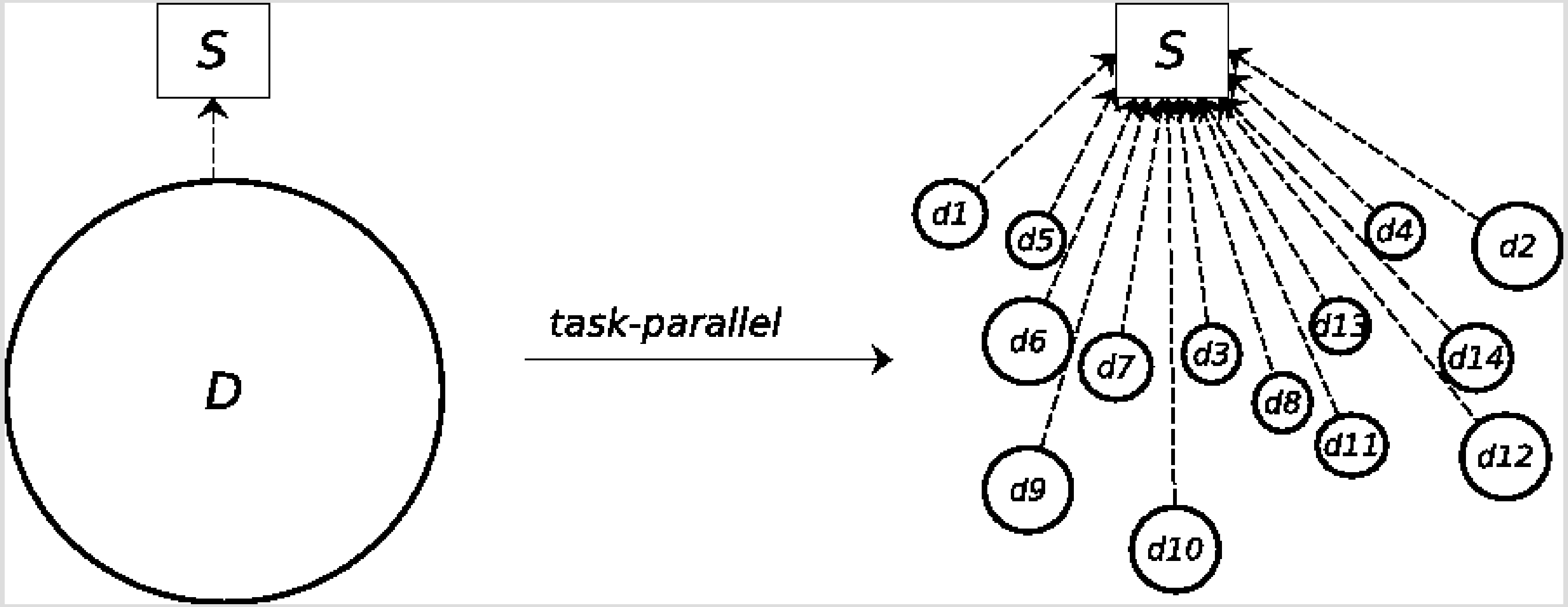
- P_D is data-parallel:

$$k(D)=k(d_1)+k(d_2)+\ldots+k(d_l)=\sum_{i=1}^{l} k(d_i)$$

- 

Basic Concepts

- P_D is task-parallel $D(S) = d_1(S) + d_2(S) + \ldots + d_k(S) = \sum_{i=1}^{k} d_i(S)$



Ejemplos de problemas paralelizables

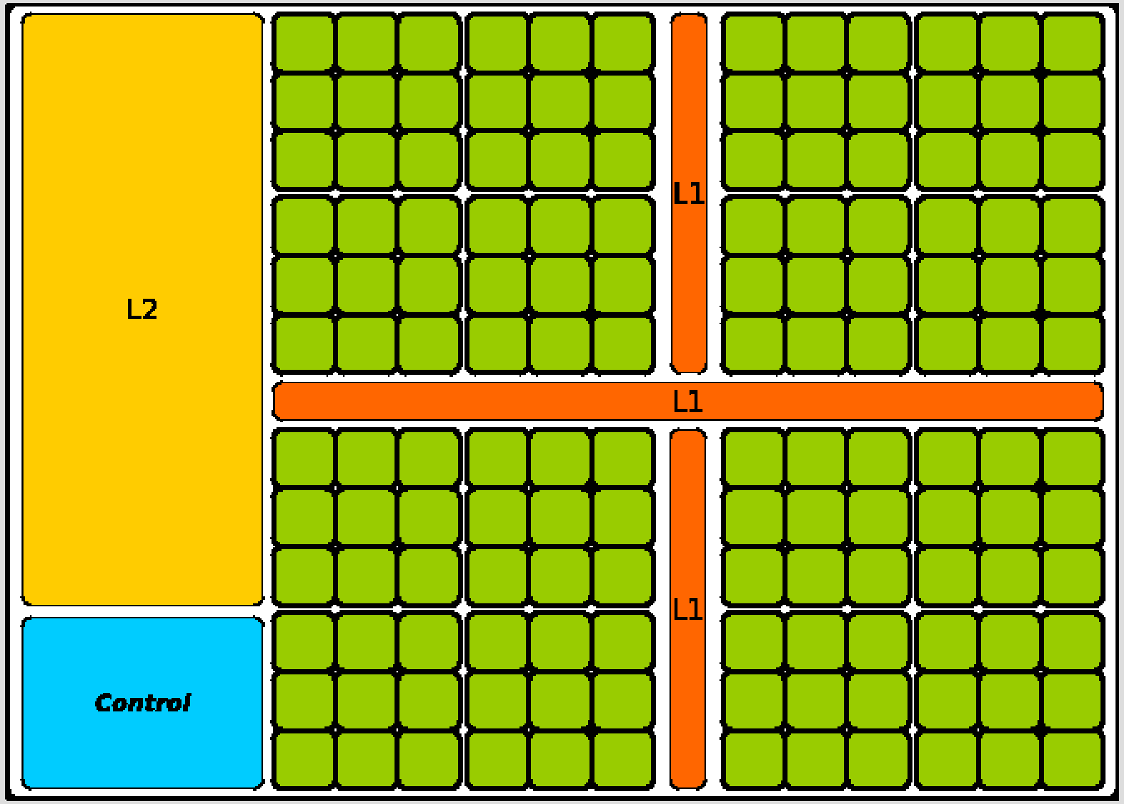
- Problems typically fall in between the two definitions:
  - Closer to Data-parallel
    - Vector addition
    - Matrix multiplication
    - N-body problem
    - ...
  - Closer to Task-parallel
    - Operating system processes
    - Videogame engines
    - ...

Data parallelism y GPU

- Well suited for GPUs; thousands of cores
- Each core can handle one sub-problem
- GPU works as a massively parallel processor
- Physical parallelism is limited by the number of physical cores
- ...

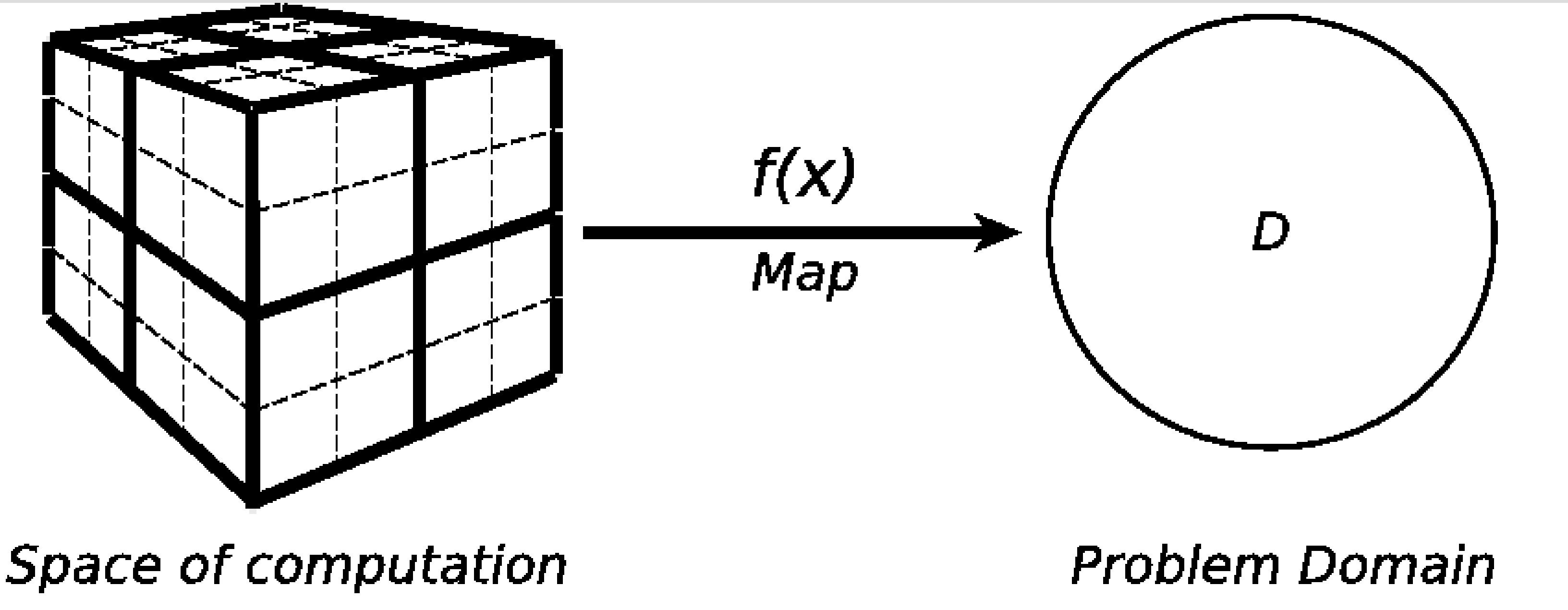
GPU Architecture

- Typically, n > p → work scheduling problem solved internally



GPU mapping

- Space of computation structure

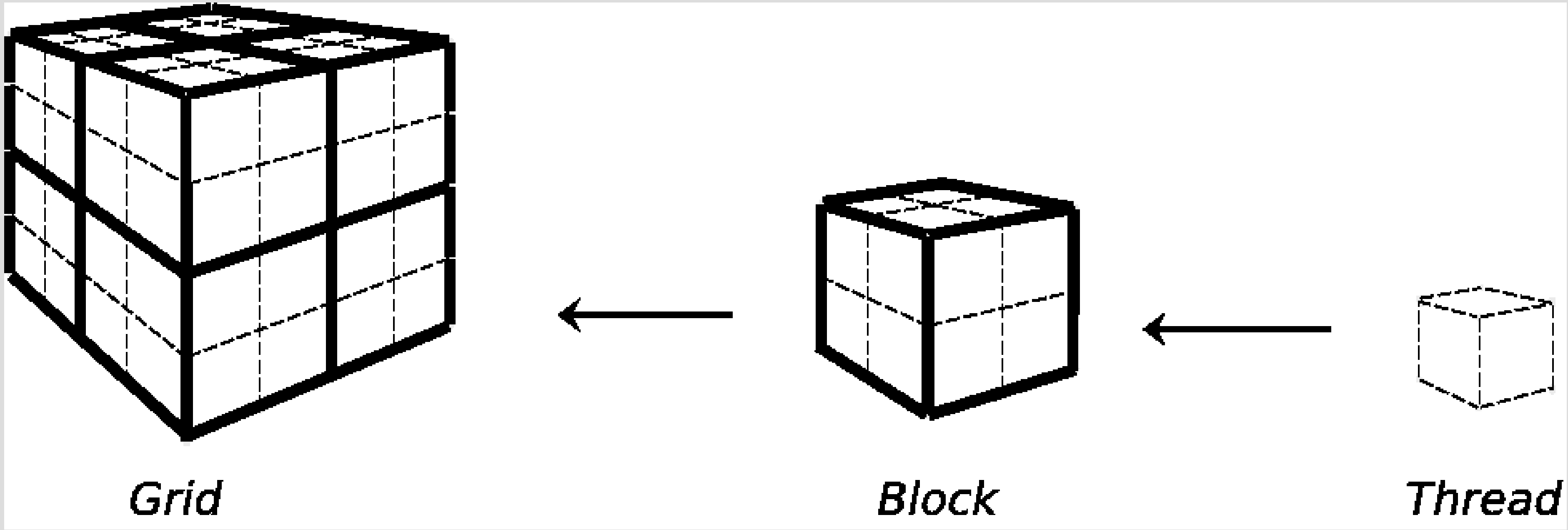


The GPU programming model

- CUDA or OpenCL → we accept variable p
- GPU programming model → abstraction layer for p
- We can use (almost) as many threads as we want, concurrently
- *Space of computation:*
  - A discrete space where a massive amount of threads are organized
  - In CUDA, the space of computation is composed of a grid, blocks and threads
  - In OpenCL, it is work-space, work-group and work-item, respectively.

GPU mapping

- Space of computation structure



Programando en Cuda

- Code Adds two vectors A and B of size N and stores the result into vector C

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}



int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Nota: 1 Bloque no es eficiente

https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Programando en Cuda

- Thread hierarchy: threadIdx is a 3-component vector
- A thread block is a one-dimensional, two-dimensional, or three-dimensional block of threads
- The index of a thread and its thread ID relate to each other in a straightforward way
  - For a one-dimensional block, they are the same
  - For a two-dimensional block of size (Dx, Dy), the thread ID of a thread of index (x, y) is (x + y Dx)
  - For a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is (x + y Dx + z Dx Dy)

https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Programming in Cuda

- There is a limit to the number of threads per block !!
  - All threads of a block are expected to reside on one streaming multiprocessor (SM)
  - They must share the limited memory resources of that SM
  - On current GPUs, a block may contain up to 1024 threads
- But a kernel can be executed by multiple equally-shaped thread blocks
  - The total number of threads is equal to the number of threads per block times the number of blocks
  - The number of thread blocks in a grid is usually dictated by the size of the data being processed (recommended: NthreadsPerBlocks%32 == 0)

Revisar Material: https://blitzman.gitbooks.io/cuda/content/
https://blitzman.gitbooks.io/cuda/content/problemas-3-hilos-en-cuda.html

Programming in Cuda

- Each CUDA block is executed by one streaming multiprocessor (SM) and cannot be migrated to other SMs in GPU (except during preemption, debugging, or CUDA dynamic parallelism).
- One SM can run several concurrent CUDA blocks depending on the resources needed by CUDA blocks.
- Figure 3 shows the kernel execution and mapping on hardware resources available in GPU.

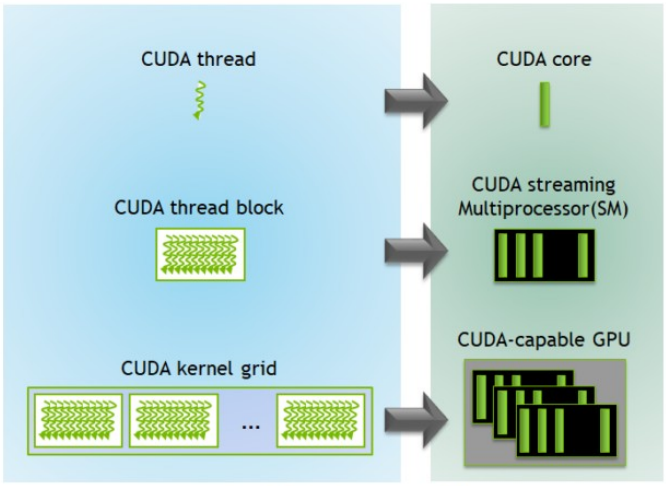


Figure 3. Kernel execution on GPU.

Programando en Cuda

- Code adds two matrices A and B of size NxN and stores the result into matrix C

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Nota: 1 Bloque no es eficiente

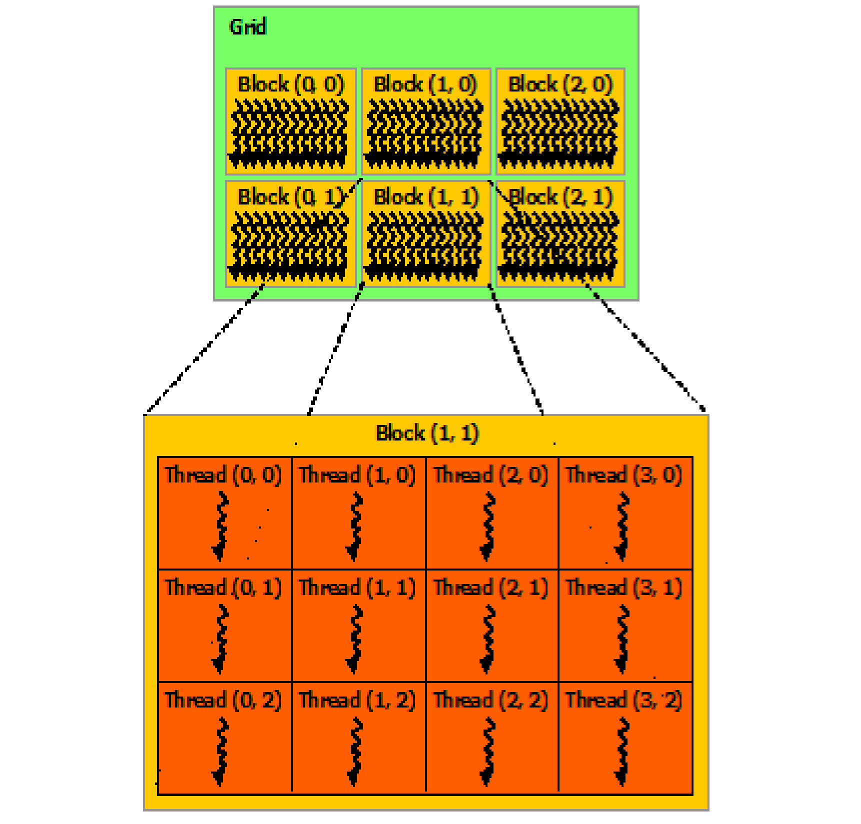
https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Programming in Cuda

- Code adds two matrices A and B of size NxN and stores the result into matrix C

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
   // Kernel invocation
   dim3 threadsPerBlock(16, 16);
   dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```



https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Programming in Cuda

- A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice
  - Thread blocks are required to execute independently
  - It must be possible to execute them in any order, in parallel or in series
  - This allows thread blocks to be scheduled in any order across the cores
- Threads within a block can cooperate by sharing data through some shared memory
  - Each thread has private local memory.
  - Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block.
  - All threads have access to the same global memory.

Comentarios

- Experimentar para encontrar la mejor configuración grid/bloques/threads
- Debugging es difícil
- ….