Los 10 mandamientos del buen cudista

Curso de Deep Learning y CUDA

Titulaciones Propias. Universidad de Málaga

Manuel Ujaldón

Catedrático de Universidad
Departamento de Arquitectura de Computadores
Universidad de Málaga

NVIDIA
DEEP LEARNING INSTITUTE
UNIVERSITY AMBASSADOR

Primera:
El volumen de datos

- El umbral para aspirar a una mínima aceleración en GPU se sitúa en los **10 millones de datos**. Si tu kernel candidato está por debajo de esa cifra, va a ir más rápido en CPU.
  - No quieras arrancar el avión para un viaje de pocos kilómetros.

Segunda:
Aprovecha la ubicuidad de CUDA

- Busca una librería (preferiblemente de Nvidia) o un código similar en repositorios como Github que me ayude durante la implementación. Merece la pena ***googlear*** **unas horas** para conocer el estado del arte de mi algoritmo en GPU, pues cada vez es más frecuente encontrar hallazgos que resuelven mi problema incluso mejor de lo que había previsto.
  - *"Si he visto más lejos, es porque estoy sentado sobre hombros de gigantes"* (Isaac Newton, en una carta a Robert Hooke, aludiendo a la suerte de haber aprovechado el legado de Copérnico, Galileo y Kepler en sus estudios).

Tercera:
Detectar las dependencias de datos

- Pueden arruinar el paralelismo en los bucles de un kernel candidato. El principal enemigo es una dependencia RAW (Read-After-Write) que provoca *carried-loop dependencies* en un vector particionado entre los hilos. Ejemplo:

```
for (i=1; i<N; i++)
{
  a[i] = b[i] + 7;         // El warp 1 lee el valor a[31], que debe
  x[i] = a[i-1] * 1000;    // haber sido escrito antes por el warp 0
}
```

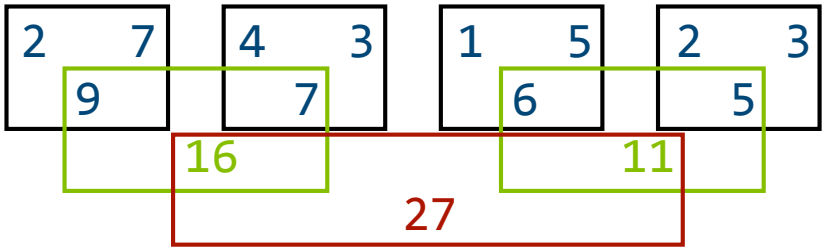
- Solución parcial (sólo funciona dentro de cada bloque):

```
idx = blockIdx.x * blockDim.x + threadIdx.x;
a[idx] = b[idx] + 7;
synchthreads();   // Barrera de sincron. para los warps del bloque
x[idx] = a[idx-1] * 1000;
```

Otros métodos para resolver dependencias de datos

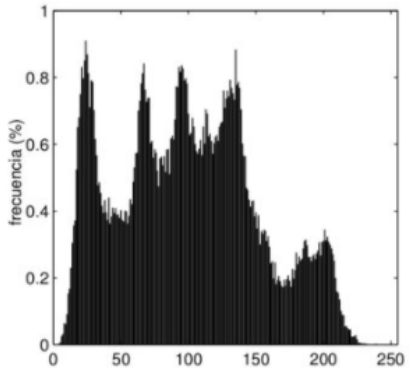
- En **operadores de reducción** que sean asociativos, podemos desplegar los hilos en forma de árbol binario sobre los datos (log(N) pasos). Ejemplo:

```
for (i=0; i<N; i++)
  sum += a[i];
```



- En **histogramas** o códigos de escritura masiva sobre un vector pequeño, podemos usar operaciones atómicas que resuelve internamente la GPU. Ejemplo:

```
for (i=0; i<Nbins; i++)
  histo[i] = 0;
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    atomicInc(histo[image[i][j]]);
```



Cuarta:
Aplicar el paralelismo adecuado

- Tipos de paralelismo:
  - De grano fino en la GPU (vs. grano grueso en la CPU).
  - De datos en la GPU (vs. tareas en la CPU).
- Lanzar tantos hilos como datos tenga el vector a paralelizar.
- Dado que el máximo nº de hilos por kernel es 2^{41}-1, el techo aquí es 2 Teradatos (8 Terabytes para `int` y `float`).
  - Ese umbral se puede flanquear aplicando *tiling* al lanzar los kernels desde CPU (así es como aprovechamos también la memoria compartida en GPU).
  - Alternativa: Asigna varias iteraciones de un bucle a cada hilo. Ej:

```
int index = blockIdx.x * blockDim.x + threadIdx.x;
int stride = blockDim.x * gridDim.x;
for (int i=index; i<N; i+=stride)
  if (i<N)
    result[i] = a[i] + b[i];
```

Quinta: En ausencia de dependencias (forall), paralelizar aplicando *owner's computer rule*

- Cada iteración de un bucle ***forall*** debe computarla el hilo propietario del dato en el vector resultado. Ejemplo: nBody.

```
void bodyForce(Body *p, float dt, int n) {
  for (int i = 0; i < n; ++i) {
    float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

    for (int j = 0; j < n; j++) {
      float dx = p[j].x - p[i].x;
      float dy = p[j].y - p[i].y;
      float dz = p[j].z - p[i].z;
      float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
      float invDist = rsqrtf(distSqr);
      float invDist3 = invDist * invDist * invDist;
      Fx += dx * invDist3;   Fy += dy * invDist3;   Fz += dz * invDist3;
    }
    p[i].vx += dt*Fx;
    p[i].vy += dt*Fy;
    p[i].vz += dt*Fz;
  }
}
```

Sexta: Delimitar las transferencias y sincronizaciones

- Qué datos deben transferirse entre CPU y GPU, y cuándo.
- Qué sincronización CPU-GPU es necesaria al usar memoria unificada (cudaDeviceSynchronize). Ejemplos:

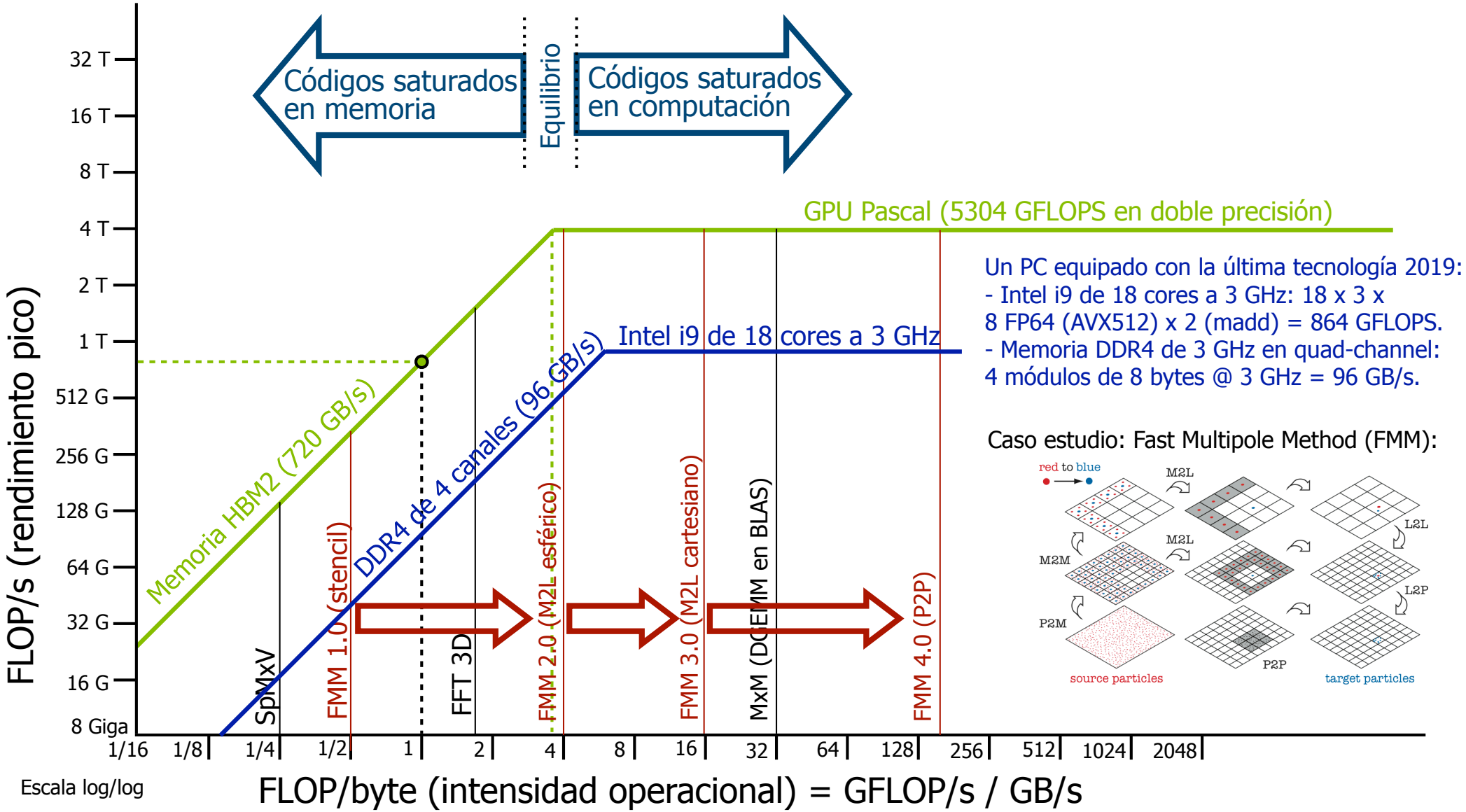
```
__global__ void incr (float *a, float b, int N) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < N)
    a[idx] = a[idx] + b;
}
void main() {
  unsigned int numBytes = N*sizeof(float);
  float* h_A = (float* ) malloc(numBytes);
  float* d_A; cudaMalloc(&d_A, numBytes);
 cudaMemcpy(d_A,h_A,numBytes,cudaMemcpyHostToDevice);
 incr<<<N/blocksize,blocksize>>>(d_A, b, N);
 cudaMemcpy(h_A,d_A,numBytes,cudaMemcpyDeviceToHost);
 cudaFree(d_A);
 free(h_A);
}
```

```
/* USING UNIFIED MEMORY ON THIS SIDE */

__global__ void incr (float *a, float b, int N) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < N)
    a[idx] = a[idx] + b;
}

void main() {
  float* m_A; cudaMallocManaged(&m_A, numBytes);

  incr<<<N/blocksize,blocksize>>>(m_A, b, N);
  cudaDeviceSynchronize();
  cudaFree(m_A);
}
```

Séptima: Analizar si el kernel candidato es *memory-bound* o *compute-bound*

- Ayúdate del modelo *roofline* para analizar la reutilización de los datos que llegan a la GPU y la complejidad de las operaciones que ésta realiza sobre ellos.
  - Más del 70% de los códigos científicos que forman parte de los benchmarks más populares son *memory-bound*.
  - Un kernel es *memory-bound* si efectúa menos de 4-8 ops./byte.
- Mejoras:
  - Si es *memory-bound*, usa más memoria compartida y registros en cada bloque de hilos CUDA.
  - Si es *compute-bound*, aprovecha más el hardware dedicado: *tensor cores, shuffling instructions,* …

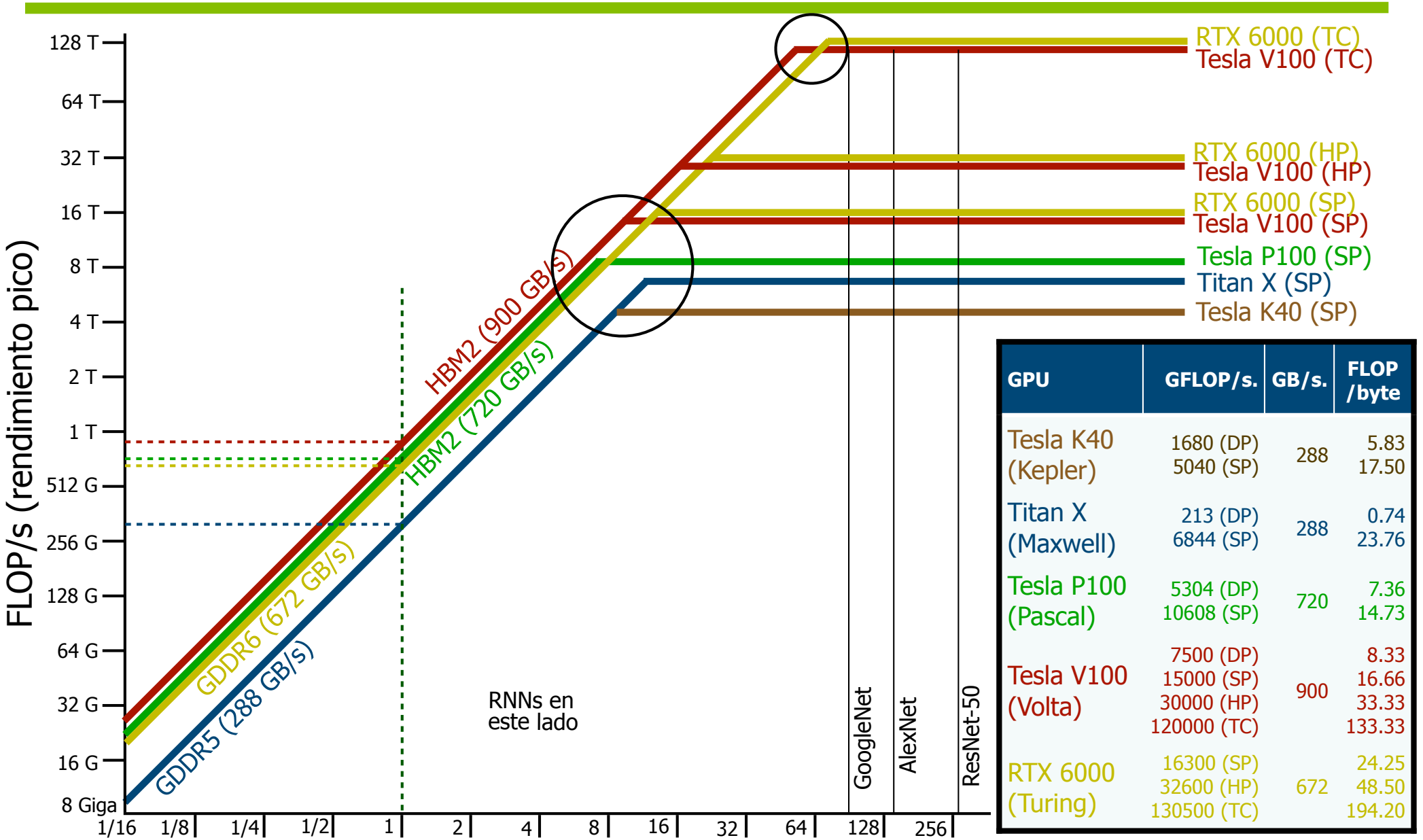
El modelo roofline: Rendimiento según las propiedades del código



Un PC equipado con la última tecnología 2019:
- Intel i9 de 18 cores a 3 GHz: 18 x 3 x 8 FP64 (AVX512) x 2 (madd) = 864 GFLOPS.
- Memoria DDR4 de 3 GHz en quad-channel: 4 módulos de 8 bytes @ 3 GHz = 96 GB/s.

Caso estudio: Fast Multipole Method (FMM):

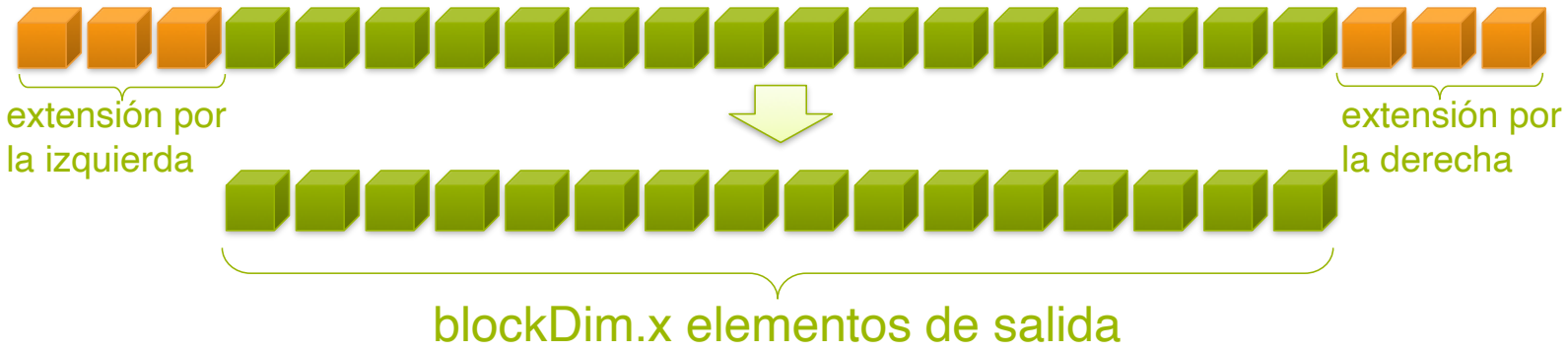
Rendimiento para Deep Learning en las últimas 5 generaciones de GPUs



GPU	GFLOP/s.	GB/s.	FLOP /byte
Tesla K40 (Kepler)	1680 (DP) 5040 (SP)	288	5.83 17.50
Titan X (Maxwell)	213 (DP) 6844 (SP)	288	0.74 23.76
Tesla P100 (Pascal)	5304 (DP) 10608 (SP)	720	7.36 14.73
Tesla V100 (Volta)	7500 (DP) 15000 (SP) 30000 (HP) 120000 (TC)	900	8.33 16.66 33.33 133.33
RTX 6000 (Turing)	16300 (SP) 32600 (HP) 130500 (TC)	672	24.25 48.50 194.20

Octava: La fuerza bruta es caballo ganador

- Busca la implementación más regular frente a otra más inteligente que seguramente sea más difícil de expresar.
- Evita las sentencias IF, que provocan divergencias en los warps. Ejemplo: *Padding* (o relleno de un vector).
- Apóyate en un *toy example.* Lo verás todo más claro y te permitirá después beneficiar a la regla frente a la excepción. Ejemplo: Stencils.



Novena:
Solapar computación y comunicación

- Aprovecha `cudaMemcpyAsync()` y el Visual Profiler, sobre todo si se trata de un *kernel memory-bound.*
- Aparte de las optimizaciones ya vistas, hay un *workshop* en el DLI dedicado a este tipo de mejoras:
  - *"Accelerating CUDA C Applications with Multiple GPUs"*
- Complementa el solapamiento anterior con el uso de *streams* entre *kernels* para expresar paralelismo de tareas que permita al *run-time* simultanear la ejecución de kernels.
- Hay otro workshop en el DLI dedicado a esta optimización:
  - *"Scaling Workloads Across Multiple GPUs with CUDA"*

Décima: Elegir la mejor configuración para la ejecución de un kernel

- Selecciona la mejor combinación de bloques/malla e hilos/bloque a partir de N (tamaño del problema a resolver). Un punto de partida sería: `mykernel<<<N/256,256>>>();`
  - Y luego sube a 512, 1024, y baja a 128, 64 hilos/bloque.
  - De forma más genérica: `mykernel<<<((N-1)/256)+1,256>>>();`
- Herramienta: CUDA Occupancy Calculator, que puedes usar además para conocer la ocupación de los SMs.
- Cuidado: La herramienta procede siempre bajo las características de un modelo de GPU concreto.
  - Toma las decisiones más escalables, al margen del modelo de GPU con el que trabajes en ese momento.

Corolario final

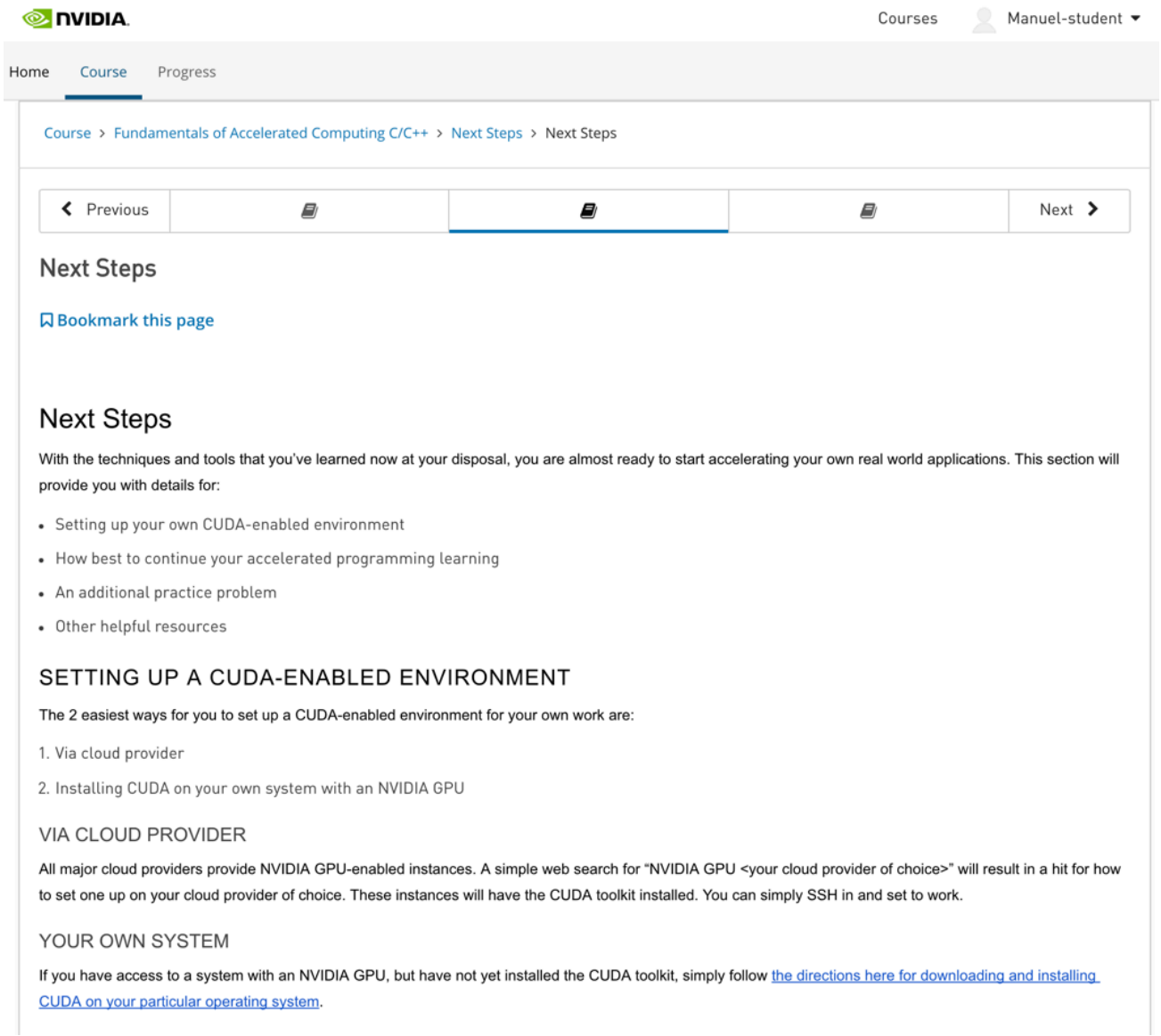
- Muchas optimizaciones CUDA se logran con la experiencia, al estar basadas más en:
  - Intuición frente a ciencia.
  - Creatividad frente a reglas preestablecidas.
  - Heurísticos frente a teoremas.
- En eso se parece mucho al Deep Learning…

Epílogo:
Para crecer en el futuro con CUDA

Cómo completar tu entrenamiento una vez concluida la parte presencial del curso

Actividad	Método
Mejora tu proyecto de certificación	Mejora y transforma tu programa CUDA de diferentes maneras para conseguir rendimiento adicional.
Configura tu propio entorno de desarrollo CUDA	Las dos fórmulas más sencillas son: 1. El alquiler, a través de un proveedor de la nube como AWS. 2. La compra, instalando CUDA en tu propio sistema con GPU Nvidia.
Rellena la encuesta de valoración del curso.	Evalúa el material del curso, la labor de tu instructor y la experiencia global.
Acelera tus propias aplicaciones	1. Toma algunas métricas de una aplicación en la que trabajes. 2. Piensa dónde podrías acelerarla. 3. Realiza los cambios oportunos para paralelizarla en GPU. 4. Utiliza el Visual Profiler y repite el ciclo de desarrollo.
Resuelve proyectos adicionales propuestos	Acepta el reto de acelerar un simulador de Mandelbrot siguiendo el proceso interactivo que guía el *profiler*. Se proporciona un simulador C++ para que puedas ver visualmente el impacto de tu aceleración.
Lecturas recomendadas	*CUDA C Best Practices Guide*. Bibliografía recomendada en nvidiaDLI.uma.es.
Comienzo ágil del entorno	Iníciate en la nube con NVIDIA AMI, *nvidia-docker*, y *CUDA development image*.

Contenidos adicionales (tras la certificación)



Contenidos adicionales (2)

CONTINUING YOUR ACCELERATED COMPUTING DEVELOPMENT

After setting up your own accelerated system, there is one very best thing you can do to further your development as an accelerated computing programmer, work to accelerate your own applications. In addition,

WORK TO ACCELERATE YOUR OWN APPLICATIONS

You have learned how to approach accelerated computing iteratively, and in a profile-driven manner, so:

- Take some baseline measurements of a compute-intensive application you work on
- Make some hypotheses about where you might accelerate it
- Make some naive changes
- Profile and repeat

READ AND APPLY THE *CUDA C BEST PRACTICES GUIDE*

Even though you are ready to accelerate CPU-only applications in ways that will meaningfully improve their performance, you can take the study of accelerated computing much further, and in time, you should.
The *CUDA C Best Practices Guide* is an essential resource for effective CUDA programming. After accelerating your own application in the ways you already know, start a study of this document, applying the techniques it describes to further improve your application's performance.

GPU-ACCELERATING PRACTICE APPLICATION

By far the best practice is to accelerate your own applications, but for those of you who might not yet have a real-world use case, try your hand at accelerating the following Mandelbrot Set simulator. Per usual, take an iterative and profile-driven approach.

- Mandelbrot Set Simulator: this C++ simulation includes a link to a detailed explanation of the application, and will allow you to visually see the impact of GPU-acceleration

HELPFUL RESOURCES

A lot of highly talented programmers have used CUDA to create highly optimized libraries to be used for accelerated computing. There are many scenarios in your own applications where you will need to write your own CUDA code, but as usual in programming, there are also many scenarios where someone else has already written the code for you.
Peruse *GPU-Accelerated Libraries for Computing* to learn where you can use highly optimized CUDA libraries for tasks like basic linear algebra solvers (BLAS), graph analytics, fast fourier transforms (FFT), random number generation (RNG), and image and signal processing, to name a few.

Contenidos adicionales (3)