CÓMPUTO PARALELO

Francisco Javier Hernández López

fcoj23@cimat.mx

http://www.cimat.mx/~fcoj23

¿Qué es el Cómputo Paralelo (CP)?

- Ejecución de más de un cómputo (cálculo) al mismo tiempo o "en paralelo", utilizando más de un procesador.
- Arquitecturas que hay en la actualidad para el cómputo paralelo:

Computadora con múltiples procesadores (Cores)

Cluster

Tarjetas Graficas (GPUs)

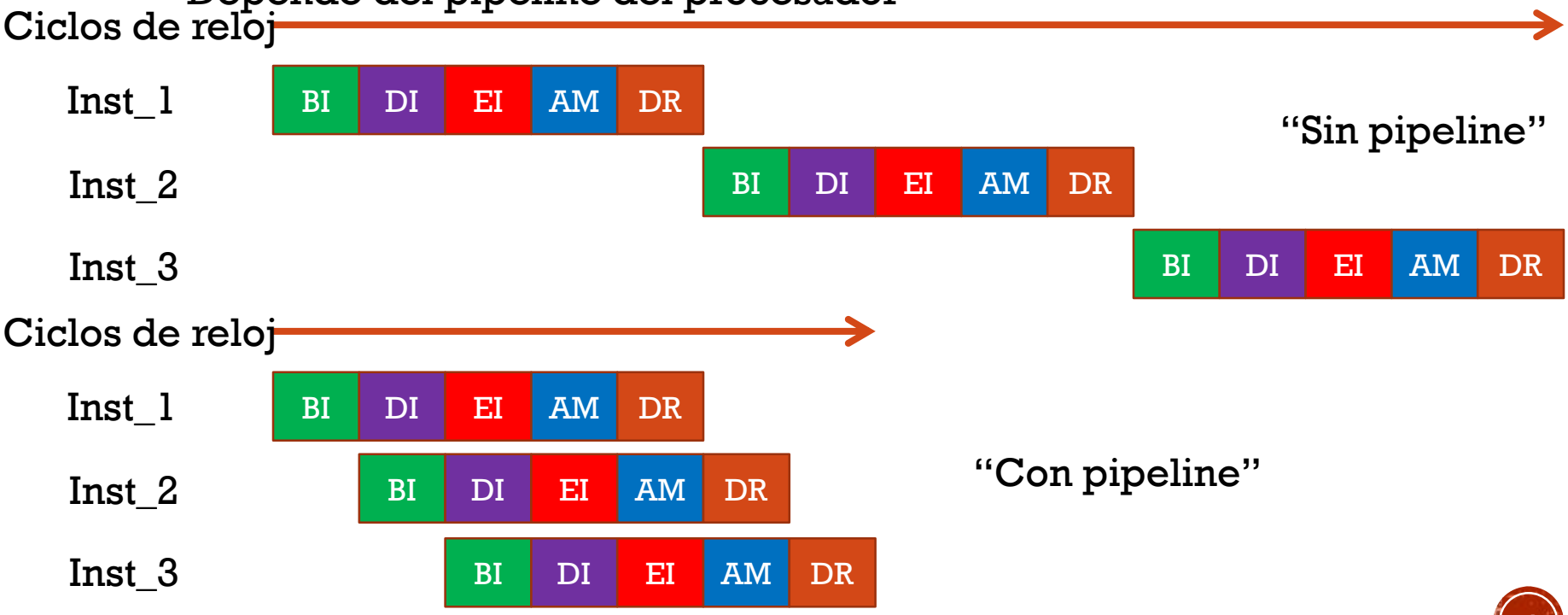
TIPOS DE CP

- **A nivel de bits**
  - Basado en incrementar el numero de bits de una palabra (*word length, word size* o *word width*)
  - Importante característica de los procesadores (8,16, 32 o 64 bits)
  - Ejemplo:
    - Tenemos un procesador de 16 bits y queremos sumar dos enteros (int) de 32 bits
    - Primero el procesador realiza la suma de los primeros 16 bits y después de los últimos 16 bits, completando la suma de los int en dos instrucciones
    - Un procesador mayor o igual a 32 bits realizaría la operación en una sola instrucción.

TIPOS DE CP

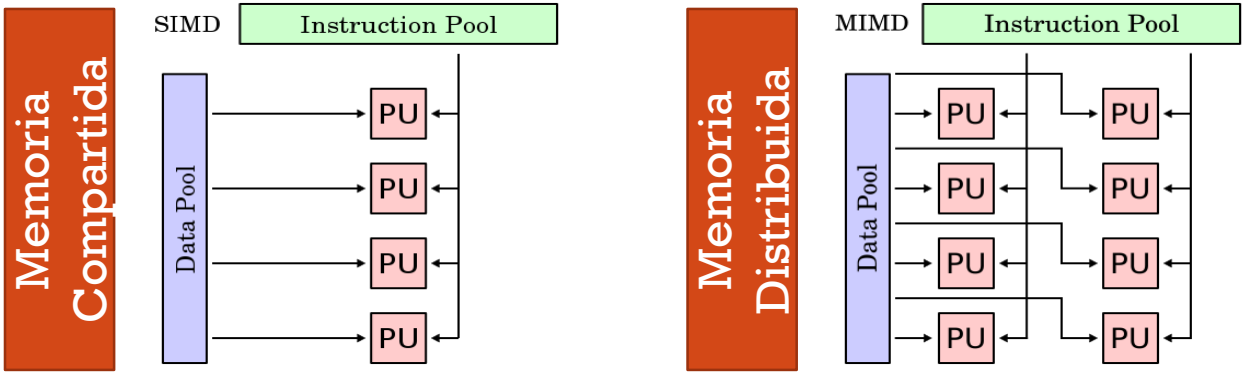
- **A nivel de instrucción**
  - Capacidad de traslapar instrucciones
  - Depende del pipeline del procesador

BI → Buscar Instrucción
DI → Descifrar Instrucción
EI → Ejecutar Instrucción
AM → Acceso a Memoria
DR → Dar Respuesta



TIPOS DE CP

- **A nivel de datos**
  - Cada procesador realiza la misma tarea en diferentes partes de los datos (SIMD → Single Instruction Multiple Data)
- **A nivel de tareas**
  - Diferentes procesadores pueden estar ejecutando diferentes instrucciones en diferentes partes de los datos (MIMD → Multiple Instruction Multiple Data)



Comparación entre SIMD y MIMD, [wikipedia]

LENGUAJES DE PROGRAMACIÓN PARA EL CP

OpenMP (Memoria Compartida)

OpenMPI (Memoria Distribuida)

CUDA (*Compute Unified Device Architecture*) para GPUs.

OpenCL (Open Computing Language) para GPUs y CPUs.

Mezcla de Arquitecturas:

Computadora (o múltiples cores) y un GPU (o un arreglo de GPUs) .

Cluster con computadoras que contienen múltiples cores y a su vez cada maquina tiene un GPU (o un arreglo de GPUs).

Así como mezclamos las arquitecturas también mezclamos los lenguajes:

- CUDA con (OpenMP, OpenMPI).
- OpenCL con (OpenMP, OpenMPI).

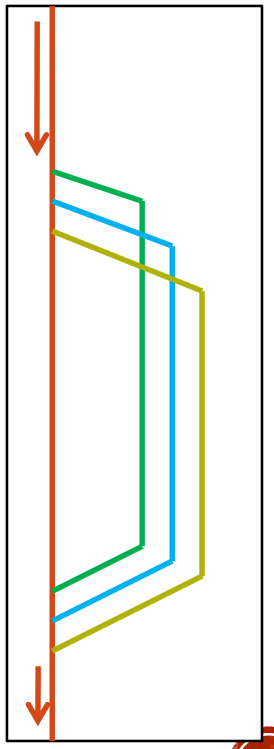
OPENMP

- Estándar para programación en paralelo con memoria compartida
- Corre en sistemas multicore
- Existen también versiones de OpenMP para GPUs y Clusters
- Podemos programar usando OpenMP en Fortran, C/C++, Java, Phyton…
- WebPage: http://openmp.org/wp/

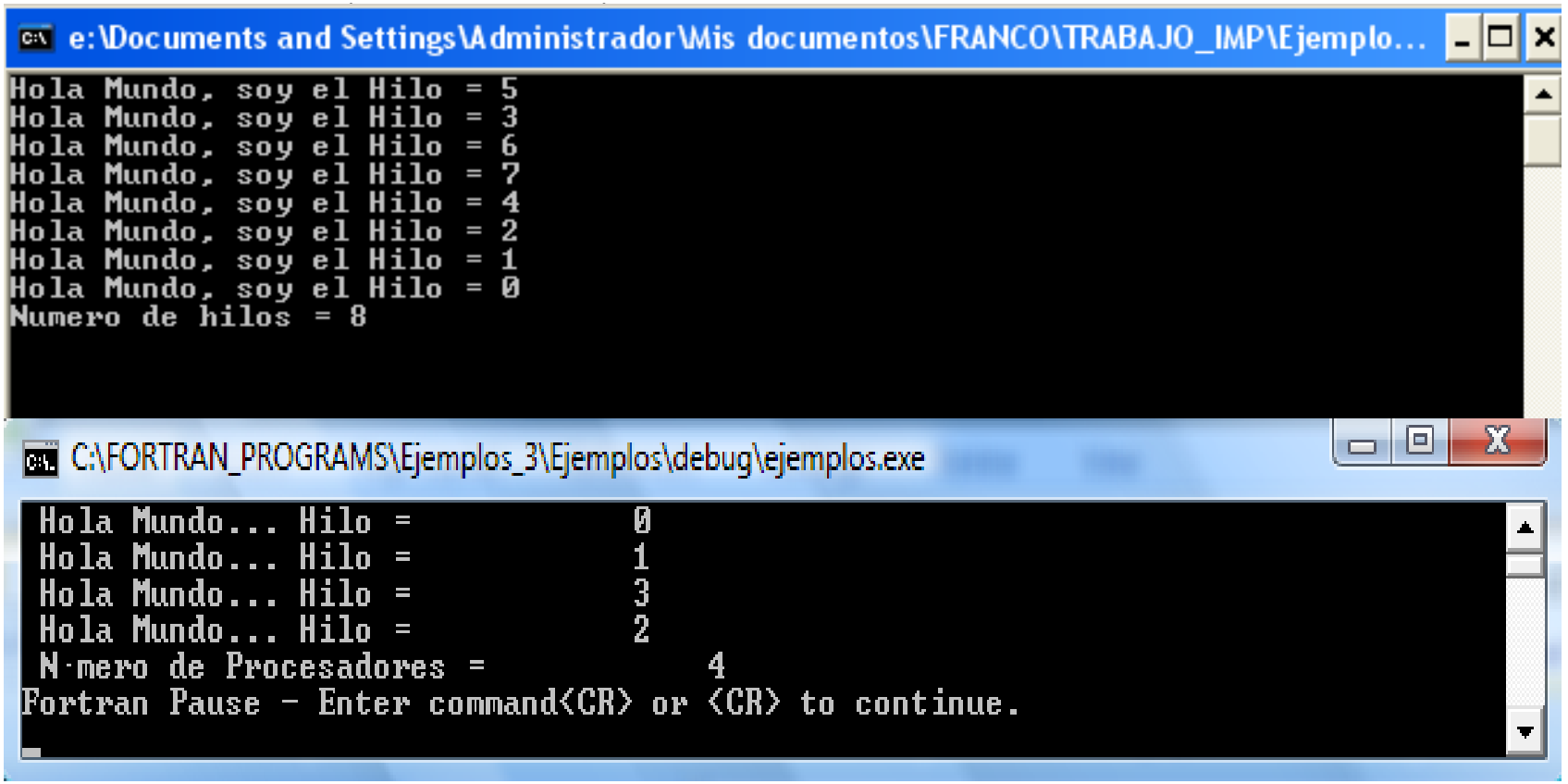
"HELLO WORLD" EN OPENMP

HolaMundo.f

```
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c     Programa: HolaMundo utilizando OpenMP
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

      PROGRAM HolaMundo
          IMPLICIT NONE
          INTEGER NTHREADS,TID,OMP_GET_NUM_THREADS,OMP_GET_THREAD_NUM

!$OMP PARALLEL PRIVATE(TID)
C         Obtenemos e imprimimos el ID del THREAD.
          TID = OMP_GET_THREAD_NUM()
          PRINT *, 'Hola Mundo... Hilo = ', TID

C         Este bloque lo hace solo el THREAD 0
          IF (TID .EQ. 0) THEN
            NTHREADS = OMP_GET_NUM_THREADS()
            PRINT *, 'Número de Procesadores = ', NTHREADS
          END IF
!$OMP END PARALLEL

          pause
      END PROGRAM HolaMundo
```



RESULTADO DEL "HELLO WORLD" EN OPENMP



DIRECTIVAS EN OPENMP PARA EL CP

- **Región Paralela.**
  - **!$**OMP PARALLEL, **!$**OMP END PARALLEL. (Fortran)
  - **#**pragma omp parallel **{ }**. (C/C++)
- **División del Trabajo.**
  - !$OMP **DO**, !$OMP END DO. (Fortran)
  - #pragma omp **for** (C/C++)

    Especifican la ejecución en paralelo de un ciclo de iteraciones

  - !$OMP **SECTIONS**, !$OMP END SECTIONS. (Fortran)
  - #pragma omp **sections** {}. (C/C++)

    Especifica la ejecución en paralelo de algún bloque de código secuencial, cada sección es ejecutada una vez por cada Hilo

  - !$OMP **SINGLE**, !$OMP END SINGLE. (Fortran)
  - #pragma omp **single** (C/C++)

    Define una sección de código donde exactamente un Hilo tiene permitido ejecutar el código, los Hilos que no fueron elegidos, simplemente ignoran dicho código

DIRECTIVAS EN OPENMP PARA EL CP

- **Combinaciones.**
  - !$OMP **PARALLEL DO**, !$OMP END PARALLEL DO. (Fortran)
  - #pragma omp **parallel for** {} (C/C++)
  - !$OMP **PARALLEL SECTIONS**, !$OMP END PARALLEL SECTIONS (Fortran)
  - #pragma omp **parallel sections** {} (C/C++)
- **Sincronización.**
  - **CRITICAL**. Solo un Hilo a la vez tiene permitido ejecutar el código
  - **ORDERED**. Asegura que el código se ejecuta en el orden en que las iteraciones se ejecutan de forma secuencial
  - **ATOMIC**. Asegura que una posición de memoria se modifique sin que múltiples Hilos intenten escribir en ella de forma simultánea
  - **FLUSH**. El valor de las variables se actualiza en todos los hilos (ejemplo: banderas)
  - **BARRIER**. Sincroniza todos los hilos
  - **MASTER**. el código lo ejecuta sólo el hilo maestro (No implica un Barrier)

DIRECTIVAS EN OPENMP PARA MANIPULAR LOS DATOS

- ***private(lista):*** Las variables de la lista son privadas a los hilos, lo que quiere decir que cada hilo tiene una variable privada con ese nombre

- ***firstprivate(lista):*** Las variables son privadas a los hilos, y se inicializan al entrar con el valor que tuviera la variable correspondiente

- ***lastprivate(lista):*** Las variables son privadas a los hilos, y al salir quedan con el valor de la última iteración (si estamos en un bucle do paralelo) o sección

DIRECTIVAS EN OPENMP PARA MANIPULAR LOS DATOS

- ***shared(lista):*** Indica las variables compartidas por todos los hilos

- ***default(shared|none):*** Indica cómo serán las variables por defecto. Si se pone *none* las que se quiera que sean compartidas habrá que indicarlo con la cláusula *shared*

- ***reduction(operador:lista):*** Las variables de la lista se obtienen por la aplicación del operador, que debe ser asociativo

CLAUSULA SCHEDULE DE OPENMP

- Utilizada con la directiva **DO** o **for**
- Especifica un algoritmo de planificación, que determina de qué forma se van a dividir las iteraciones del ciclo entre los hilos disponibles
- Se debe especificar un tipo y, opcionalmente, un tamaño (*chunk*)
- Tipos:
  - **STATIC**.
  - **DYNAMIC**.
  - **GUIDED**.
  - **RUNTIME**.

EJEMPLO: SUMA DE VECTORES EN OPENMP

- Suma de vectores (c_h = a_h + b_h)
  - Creamos los vectores "a_h", "b_h" y "c_h" en el host
  - Inicializamos con cualquier valor los vectores "a_h" y "b_h" (Paralelizar con OpenMP esta parte)
  - Sumamos a_h y b_h; el resultado lo guardamos en c_h

    (Paralelizar con OpenMP esta parte)
  - Desplegamos la suma de los vectores.

EJEMPLO: SUMA DE VECTORES EN OPENMP

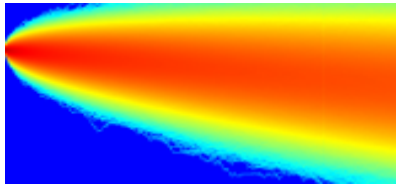
```
// Código principal que se ejecuta en el Host
int main(void){
    float *a_h,*b_h,*c_h; //Punteros a arreglos en el Host
    int N = 100000000;  //Número de elementos en los arreglos //100000000
    int CHUNK=10000000,i;

    size_t size=N * sizeof(float);

    a_h = (float *)malloc(size); // Pedimos memoria en el Host
    b_h = (float *)malloc(size);
    c_h = (float *)malloc(size);//También se puede con cudaMallocHost

    //Inicializamos los arreglos a,b en el Host
#pragma omp parallel shared(a_h,b_h,N,CHUNK) private(i)
    {
#pragma omp for schedule(dynamic,CHUNK) nowait
        for (i=0; i<N; i++){
            a_h[i] = (float)i;
            b_h[i] = (float)(i+1);
        }
    }
    Suma_vectores(c_h,a_h,b_h,N,CHUNK);

    for (i=0; i<20; i++)
        printf("%f ", c_h[i]);

    _getche();
    // Liberamos la memoria del Host
    free(a_h);
    free(b_h);
    free(c_h);
    return(0);
}
```

```
void Suma_vectores(float *c,float *a,float *b, int N,int CHUNK)
{
  int i;
#pragma omp parallel shared(a,b,c,N,CHUNK) private(i)
    {
#pragma omp for schedule(dynamic,CHUNK)nowait
        for(i=0;i<N;i++){
            c[i] = a[i] + b[i];
        }
    }
}
```

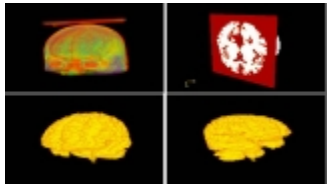
TARJETAS DE VIDEO (GPUS)

GPUS

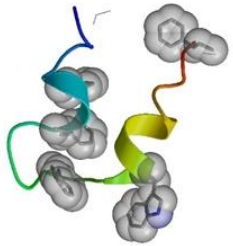
- Procesadores flexibles de procesamiento general
- Se pueden resolver problemas de diversas áreas:
  - Finanzas, Gráficos, Procesamiento de Imágenes y Video, Algebra Lineal, Física, Química, Biología, etc.



Ecs. Diferenciales



Segmentación de Imágenes Medicas

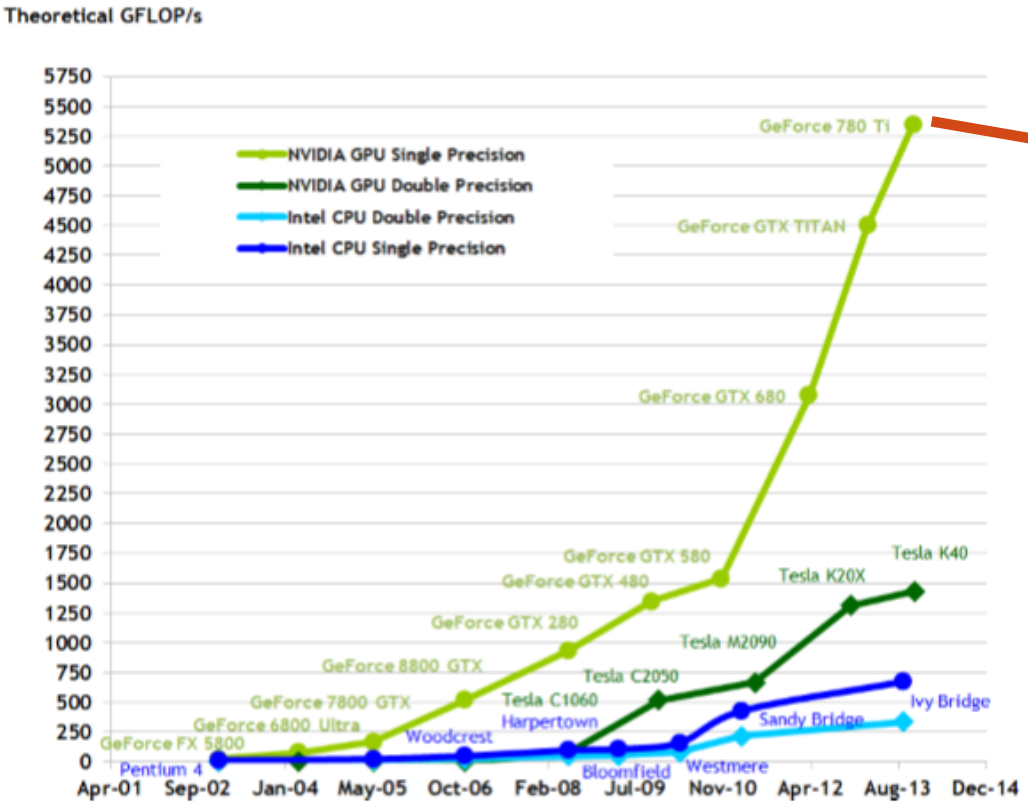


Dinámica Molecular

Detección de Objetos

Visitar CUDA ZONE

GPUS VS CPUS

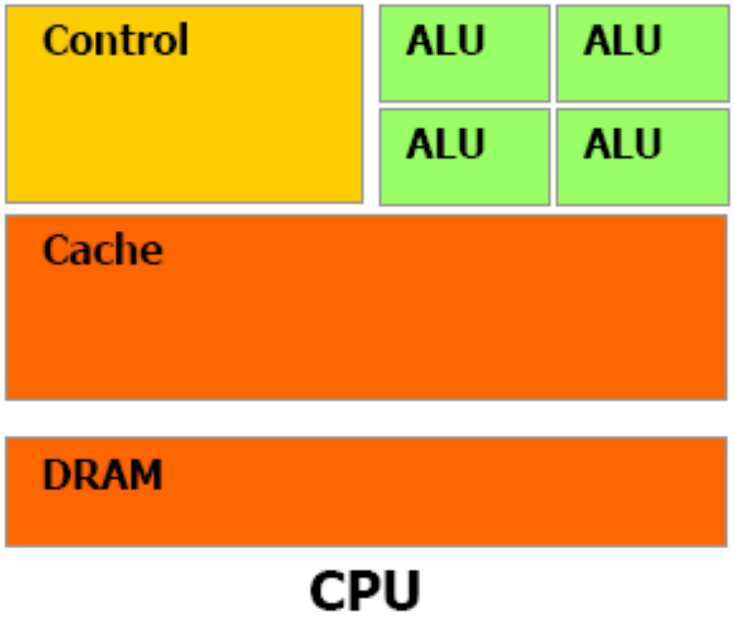


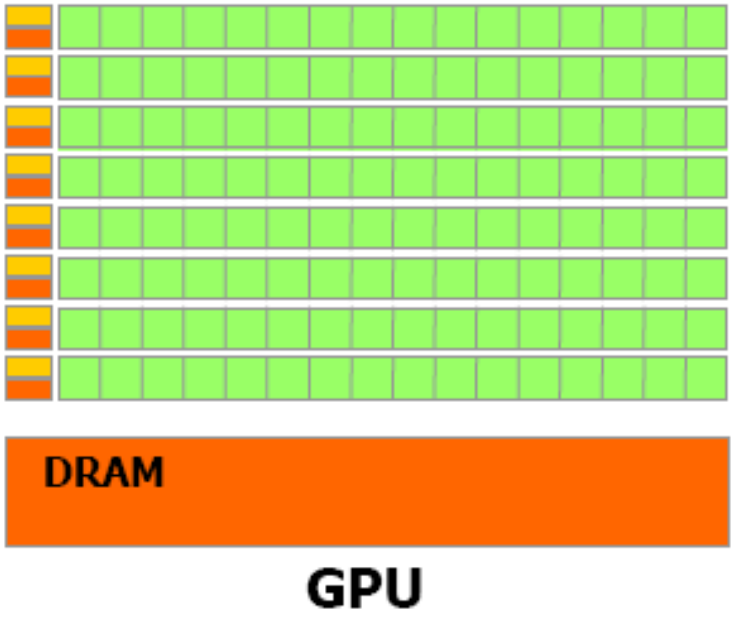
From CUDA_C_Programming_Guide.pdf

GeForce GTX 780Ti

GPUs vs CPUs

- Las GPUs cuentan con mayor número de transistores para procesar los datos





CUDA

- Es una tecnología de propósito general que nos permite ejecutar código en GPUs para hacer Cómputo Paralelo
- Desarrollado por NVIDIA en el 2006

Arquitectura	Capacidad
8-200 series	1.0 - 1.3
FERMI (400 series)	2.0 - 2.1
KEPLER (600 series)	3.0 - 3.5

Nuevas Arq. (2014-2016)	Capacidad
MaxWell	5.0 - 5.2
Volta-Pascal	--

Arquitecturas de las GPUs y sus capacidades

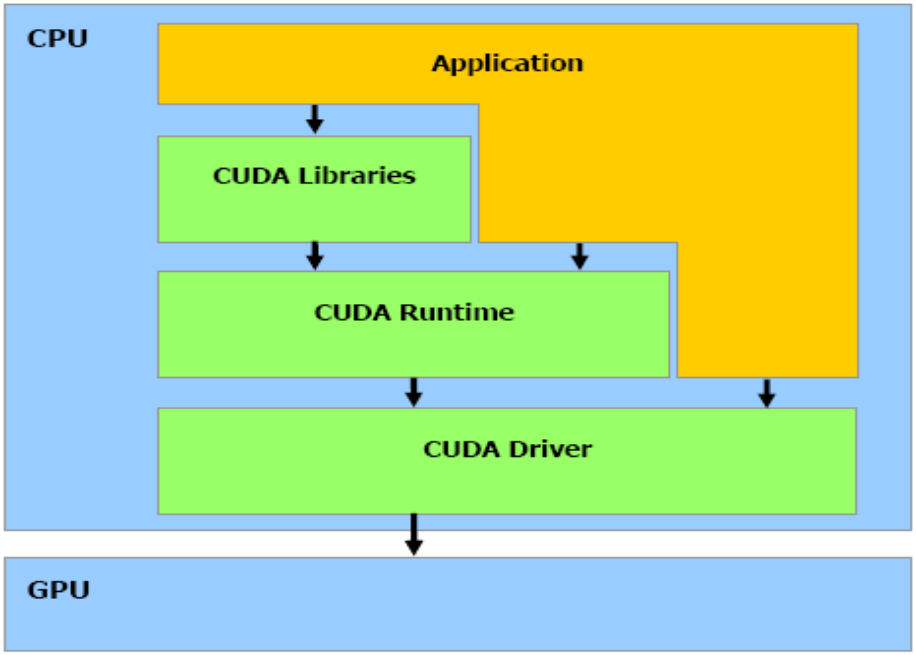
SOFTWARE USANDO CUDA

CARACTERÍSTICAS DE CUDA

- Soporta los lenguajes de programación C/C++, Fortran, Matlab, Python, LabView, etc.
- Soporte de datos en paralelo y manejador de hilos.
- Librerías:
  - FFT (Fast Fourier Transform)
  - BLAS (Basic Linear Algebra Subroutines)
  - CURAND (Generar números aleatorios)
  - CUSPARSE (Subrutinas de algebra lineal para operar matrices ralas)
  - NPP (NVIDIA Performance Primitives)…
- Opera internamente con OpenGL y DirectX.
- Soporta los sistemas operativos:
  - Windows XP 32/64-bit, Windows Vista 32/64-bit, Windows 7 32/64-bit, Linux 32/64-bit y Mac OS.

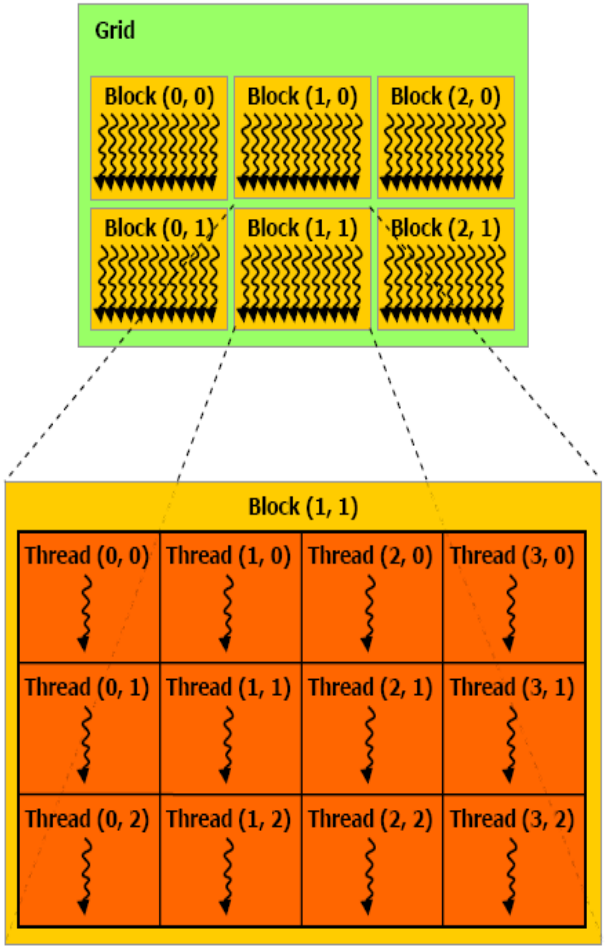
SOFTWARE CUDA

- El software CUDA esta compuesto por:
  - Hardware driver
  - Runtime
  - Libraries



MODELO DE PROGRAMACIÓN EN CUDA

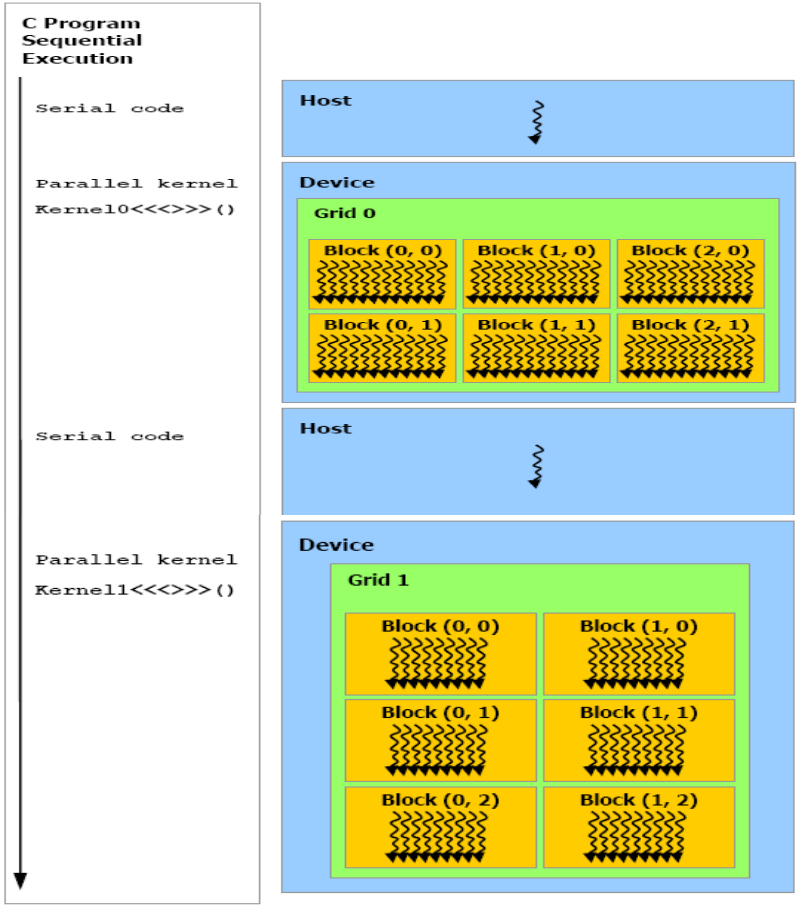
- Un programa que se compila para ejecutarse en una tarjeta gráfica se le llama *Kernel*.
- El conjunto de hilos que ejecuta un *Kernel* están organizados como una cuadricula (grid) de bloques de hilos.
- Un Bloque de hilos es un conjunto de hilos que pueden cooperar juntos:
  - Con rápido acceso a memoria compartida.
  - De forma sincronizada.
  - Con un identificador de hilos ID.
  - Los Bloques pueden ser arreglos de 1, 2 o 3 dimensiones.
- Un Grid de bloques de hilos:
  - Tiene un número limitado de hilos en un bloque.
  - Los bloques se identifican mediante un ID.
  - Pueden ser arreglos de 1 o 2 dimensiones.



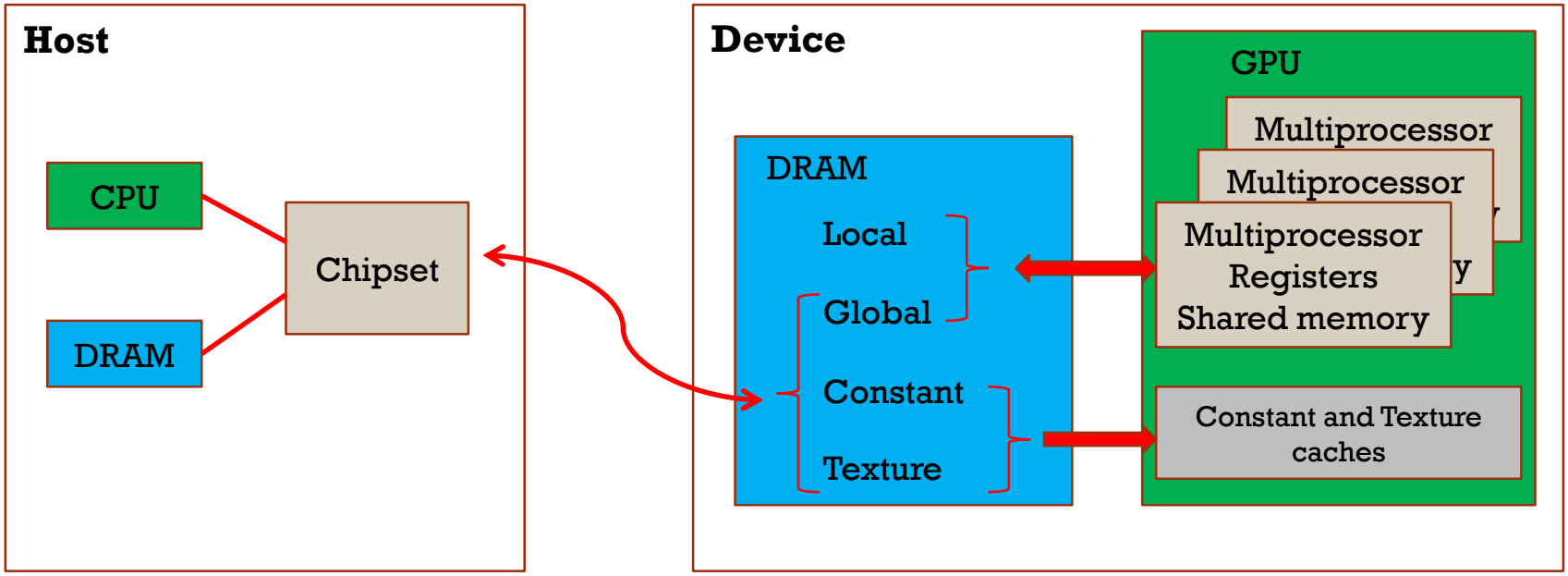
MODELO DE PROGRAMACIÓN EN CUDA

- Ejecución en el Host y Device.

Host = CPU
Device = GPU
Kernel = Conjunto de instrucciones que se ejecutan en el device.



MODELO DE LA MEMORIA EN CUDA



INSTALANDO CUDA

http://developer.nvidia.com/cuda/cuda-downloads

CUDA 6.5 Production Release

NEW CUDA for IBM POWER8 now available.

Read about 10 ways CUDA 6.5 improves performance and productivity in this blog by Mark Harris.

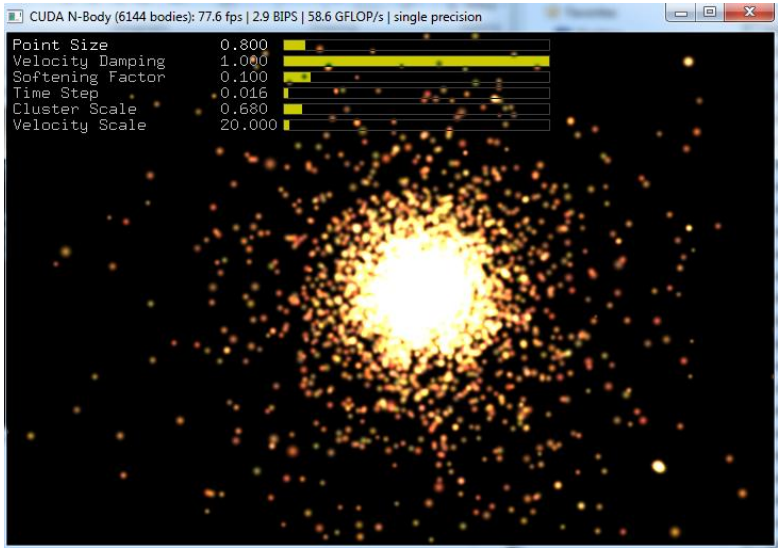
Review the latest CUDA 6.5 performance report to learn how much you could accelerate your code.

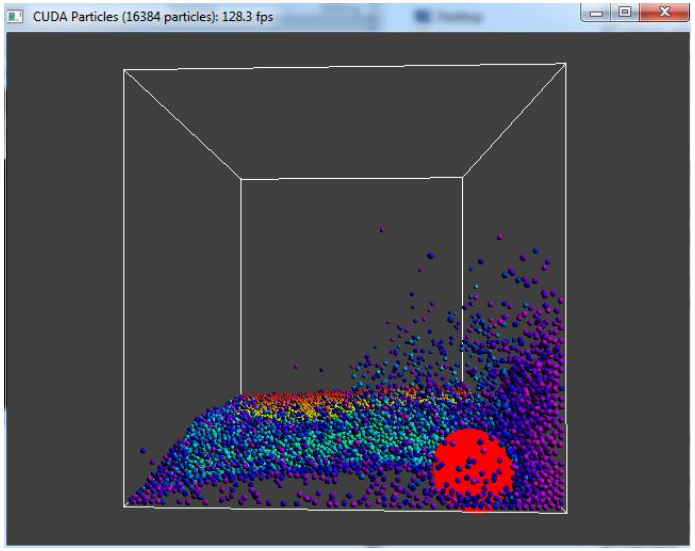
Windows | Linux x86 | Linux ARM | Mac OSX

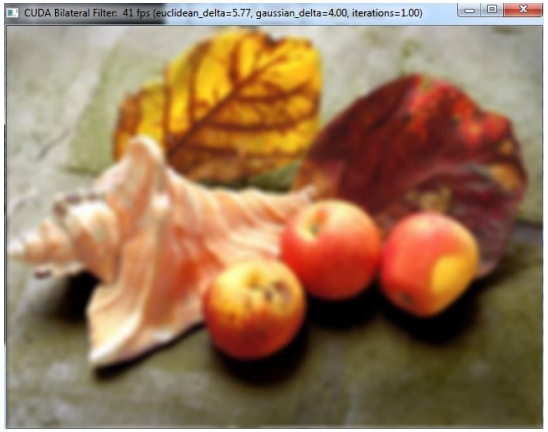
If you are developing with CUDA 6.5 on a GeForce GTX980 or GTX970 download CUDA 6.5 Toolkits with support for your GPU here.

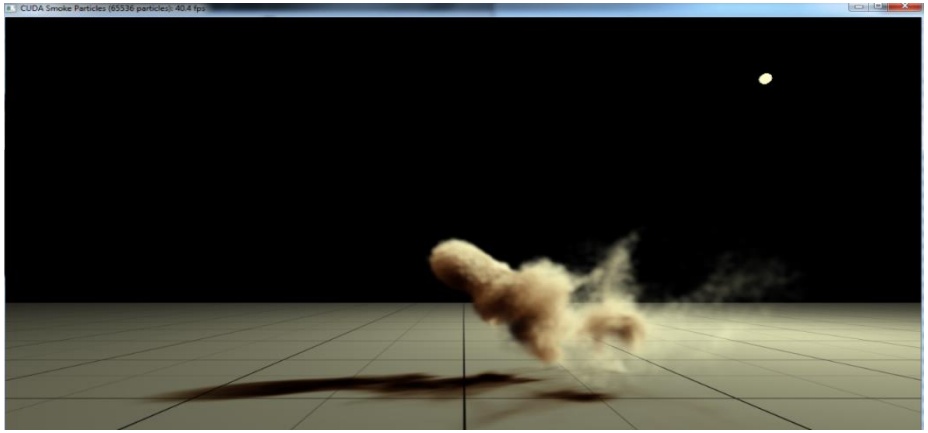
Version		64-bit	32-bit
Windows 8.1 Windows 7	Notebook	EXE	EXE
	Desktop	EXE	EXE

EJEMPLOS DEL SDK









EJEMPLO: SUMA DE VECTORES

- Suma de vectores (c = a + b)
  - Creamos los vectores "a_h", "b_h" y "c_h" en el host.
  - Inicializamos con cualquier valor los vectores "a_h" y "b_h"
  - Creamos los vectores "a_d", "b_d" y "c_d" en el device
  - Copiamos el contenido de los vectores a y b del host al device
  - Sumamos a y b; el resultado lo guardamos en c; en el device
  - Copiamos el resultado del device al host
  - Desplegamos la suma de los vectores a y b

EJEMPLO: SUMA DE VECTORES

```
// Código principal que se ejecuta en el Host
int main(void){
    float *a_h,*b_h,*c_h; //Punteros a arreglos en el Host
    float *a_d,*b_d,*c_d;  //Punteros a arreglos en el Device
    const int N = 24;  //Número de elementos en los arreglos  (probar 1000000)

    size_t size=N * sizeof(float);

    a_h = (float *)malloc(size); // Pedimos memoria en el Host
    b_h = (float *)malloc(size);
    c_h = (float *)malloc(size);//También se puede con cudaMallocHost

    //Inicializamos los arreglos a,b en el Host
    for (int i=0; i<N; i++){
        a_h[i] = (float)i;
        b_h[i] = (float)(i+1);
    }

    printf("\nArreglo a:\n");
    for (int i=0; i<N; i++) printf("%f ", a_h[i]);
    printf("\n\nArreglo b:\n");
    for (int i=0; i<N; i++) printf("%f ", b_h[i]);

    cudaMalloc((void **) &a_d,size);   // Pedimos memoria en el Device
    cudaMalloc((void **) &b_d,size);
    cudaMalloc((void **) &c_d,size);

    //Pasamos los arreglos a y b del Host al Device
    cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);
```

EJEMPLO: SUMA DE VECTORES

```
//Realizamos el cálculo en el Device
int block_size =8;
int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);

Suma_vectores <<< n_blocks, block_size >>> (c_d,a_d,b_d,N);

//Pasamos el resultado del Device al Host
cudaMemcpy(c_h, c_d, size,cudaMemcpyDeviceToHost);

//Resultado
printf("\n\nArreglo c:\n");
for (int i=0; i<N; i++) printf("%f ", c_h[i]);

_getche();

// Liberamos la memoria del Host
free(a_h);
free(b_h);
free(c_h);

// Liberamos la memoria del Device
cudaFree(a_d);
cudaFree(b_d);
cudaFree(c_d);
return(0);
}
```

```
// Función Kernel que se ejecuta en el Device.
__global__ void Suma_vectores(float *c,float *a,float *b, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N){
      c[idx] = a[idx] + b[idx];
  }
}
```

PREGUNTAS

GRACIAS