Sistemas Complejos en Máquinas Paralelas

Introducción a GPGPU

Esteban E. Mocskos (emocskos@dc.uba.ar)

Facultad de Ciencias Exactas y Naturales, UBA

CONICET

29/05/2012

Objetivos:

- Una revisión de temas *conocidos*.
- Tener una idea bastante detallada de la arquitectura de una GPU actual.
- Conocer el modelo de cómputo y su impacto en la programación de aplicaciones.
- Poder hacer un primer programana usando CUDA.

¿Por qué se llega a HPC?

> *"We use HPC to solve problems that could not be solved in a reasonnable amount of time using a single desktop computer."*

- Toman mucho tiempo de cómputo

¿Por qué se llega a HPC?

> *"We use HPC to solve problems that could not be solved in a reasonnable amount of time using a single desktop computer."*

- Toman mucho tiempo de cómputo
- Necesitan una gran cantidad de memoria RAM

¿Por qué se llega a HPC?

> *"We use HPC to solve problems that could not be solved in a reasonnable amount of time using a single desktop computer."*

- Toman mucho tiempo de cómputo
- Necesitan una gran cantidad de memoria RAM
- Se necesita una gran cantidad de corridas (por ejemplo *parameter sweeping*).

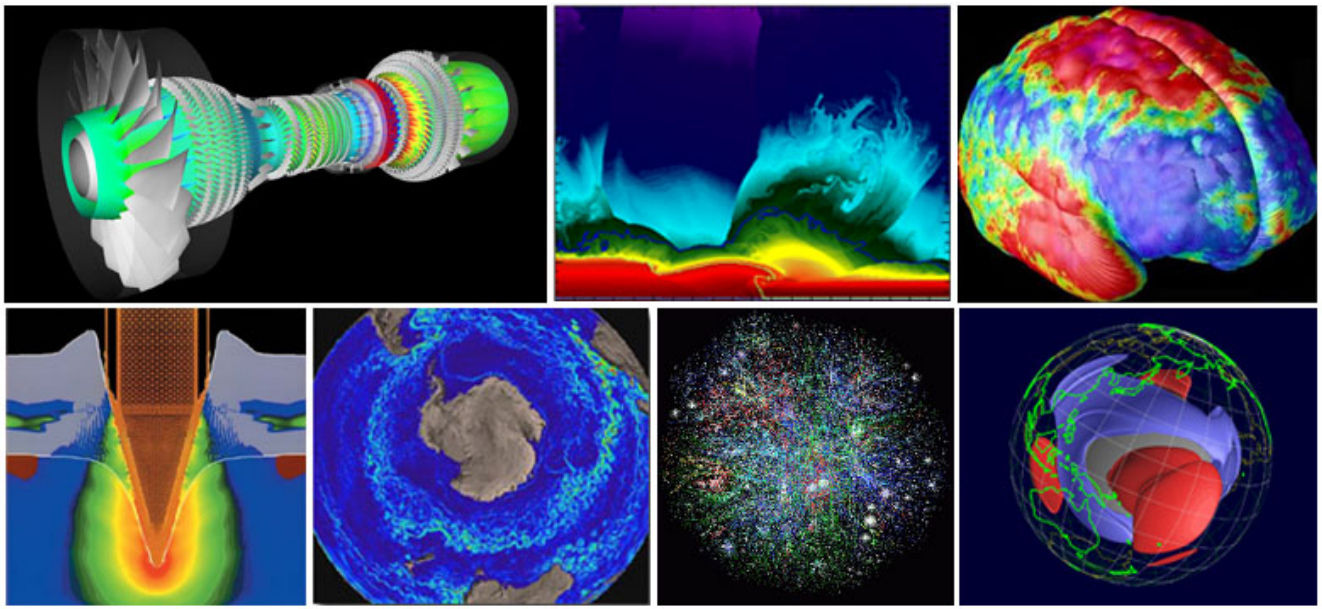
¿Por qué se llega a HPC?

> *"We use HPC to solve problems that could not be solved in a reasonnable amount of time using a single desktop computer."*

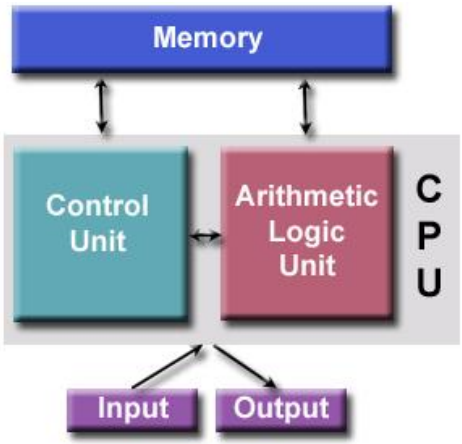
- Toman mucho tiempo de cómputo
- Necesitan una gran cantidad de memoria RAM
- Se necesita una gran cantidad de corridas (por ejemplo *parameter sweeping*).
- Tienen restricciones de tiempo para ser completadas.

¿Por qué se llega a HPC?

> *"We use HPC to solve problems that could not be solved in a reasonnable amount of time using a single desktop computer."*

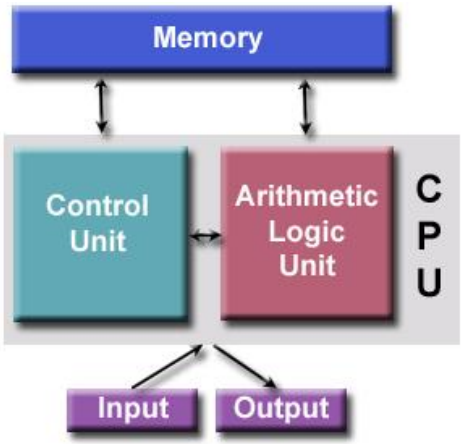


Modelo de von Neumann



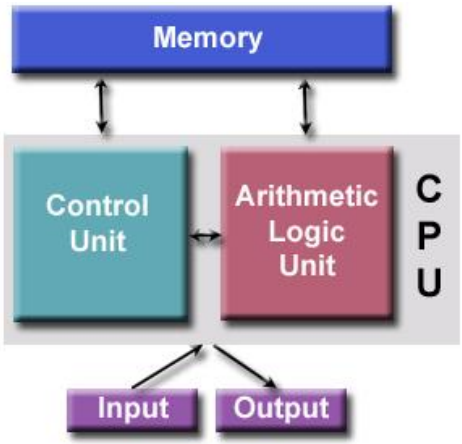
- Se compone de cuatro partes principales:
  1. Memoria
  2. Unidad de Control
  3. Unidad aritmética lógica
  4. Componentes de entrada/salida (input/output)

Modelo de von Neumann



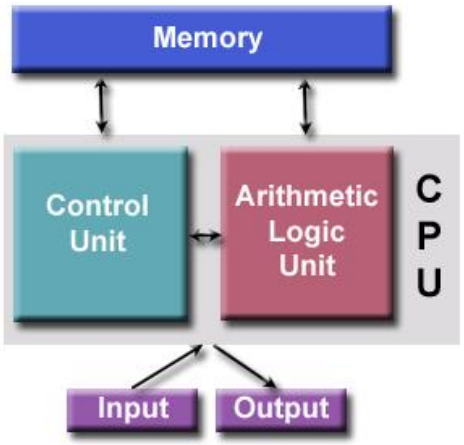
- Se compone de cuatro partes principales:
  1. Memoria
  2. Unidad de Control
  3. Unidad aritmética lógica
  4. Componentes de entrada/salida (input/output)
- Memoria de acceso aleatoria de lectura y escritura para instrucciones **y** datos.

Modelo de von Neumann



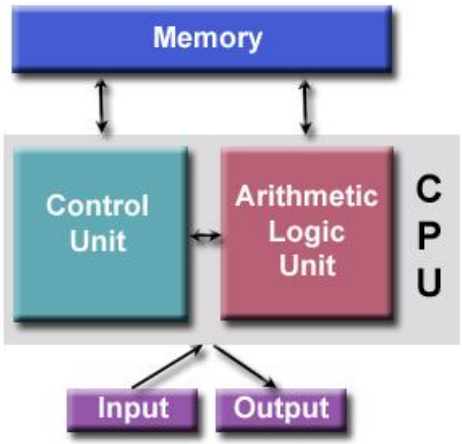
- Se compone de cuatro partes principales:
  1. Memoria
  2. Unidad de Control
  3. Unidad aritmética lógica
  4. Componentes de entrada/salida (input/output)
- Memoria de acceso aleatoria de lectura y escritura para instrucciones **y** datos.
  - Los programas son datos codificados que indican a la computadora qué tiene que hacer
  - Los *datos* son información almacenada que es usada e interpretada por el programa.

Modelo de von Neumann



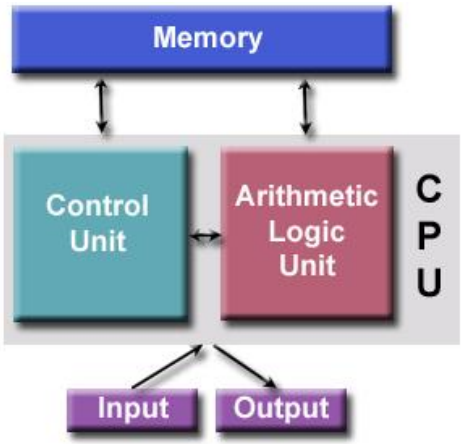
- Se compone de cuatro partes principales:
  1. Memoria
  2. Unidad de Control
  3. Unidad aritmética lógica
  4. Componentes de entrada/salida (input/output)
- Memoria de acceso aleatoria de lectura y escritura para instrucciones **y** datos.
  - Los programas son datos codificados que indican a la computadora qué tiene que hacer
  - Los *datos* son información almacenada que es usada e interpretada por el programa.
- La unidad de control consigue las instrucciones y datos de la memoria, decodifica las instrucciones y luego coordina las distintas tareas para que las operaciones se realicen.

Modelo de von Neumann



- Se compone de cuatro partes principales:
  1. Memoria
  2. Unidad de Control
  3. Unidad aritmética lógica
  4. Componentes de entrada/salida (input/output)
- Memoria de acceso aleatoria de lectura y escritura para instrucciones **y** datos.
  - Los programas son datos codificados que indican a la computadora qué tiene que hacer
  - Los *datos* son información almacenada que es usada e interpretada por el programa.
- La unidad de control consigue las instrucciones y datos de la memoria, decodifica las instrucciones y luego coordina las distintas tareas para que las operaciones se realicen.
- La unidad aritmética se encarga de las operaciones básicas entre números enteros

Modelo de von Neumann

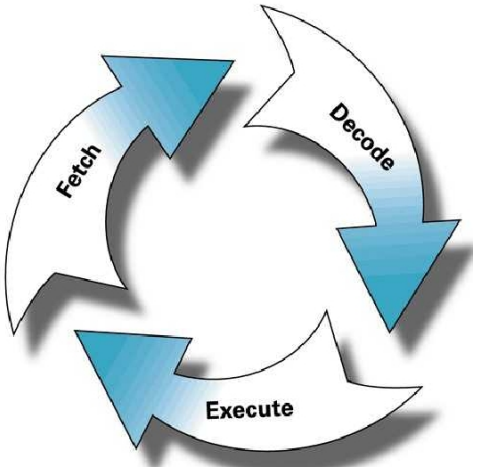


- Se compone de cuatro partes principales:
  1. Memoria
  2. Unidad de Control
  3. Unidad aritmética lógica
  4. Componentes de entrada/salida (input/output)
- Memoria de acceso aleatoria de lectura y escritura para instrucciones **y** datos.
  - Los programas son datos codificados que indican a la computadora qué tiene que hacer
  - Los *datos* son información almacenada que es usada e interpretada por el programa.
- La unidad de control consigue las instrucciones y datos de la memoria, decodifica las instrucciones y luego coordina las distintas tareas para que las operaciones se realicen.
- La unidad aritmética se encarga de las operaciones básicas entre números enteros
- Los componentes de entrada/salida son la comunicación con el universo de los humanos

Ciclo de instrucción

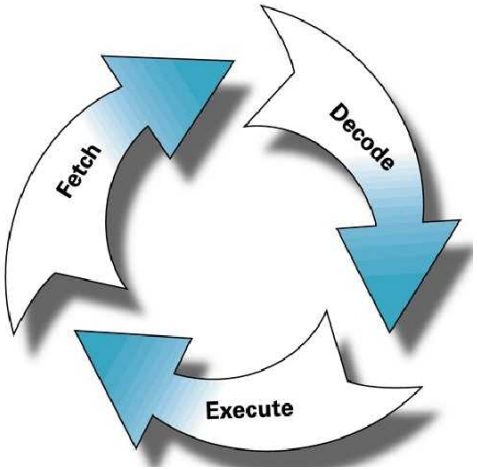
Una computadora *vive* realizando el siguiente ciclo:

1. **Fetch**: conseguir la siguiente instrucción a ejecutar.



Ciclo de instrucción

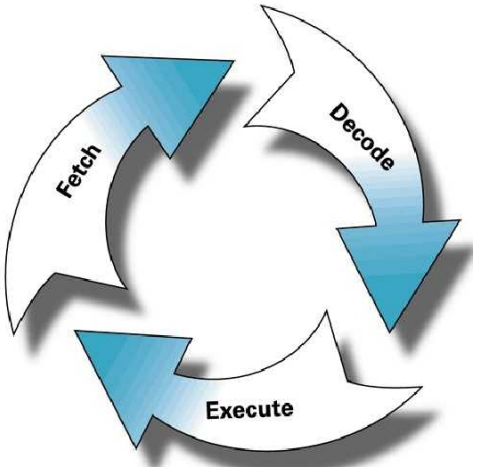
Una computadora *vive* realizando el siguiente ciclo:



1. **Fetch**: conseguir la siguiente instrucción a ejecutar.
2. **Decode**: a partir de un chorizo de bits (`0110 1110 1100 1100`) *adivinar* de qué instrucción se trata.

Ciclo de instrucción

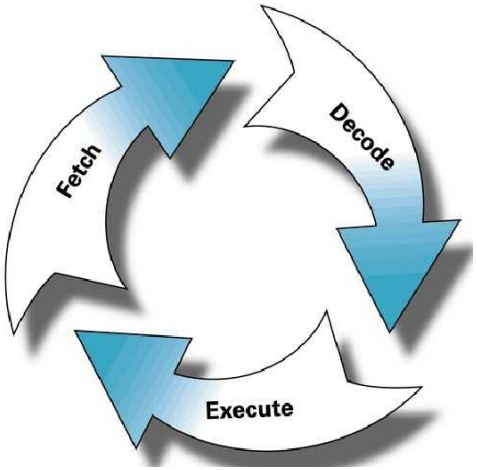
Una computadora *vive* realizando el siguiente ciclo:



1. **Fetch**: conseguir la siguiente instrucción a ejecutar.
2. **Decode**: a partir de un chorizo de bits (`0110 1110 1100 1100`) *adivinar* de qué instrucción se trata.
3. **Memory fetch**: de ser necesario, se deberán conseguir los datos requeridos por la instrucción.

Ciclo de instrucción

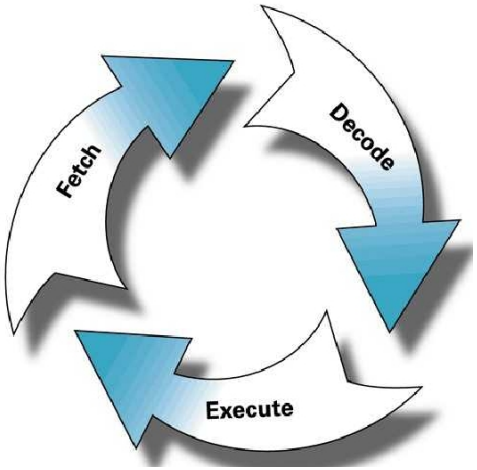
Una computadora *vive* realizando el siguiente ciclo:



1. **Fetch**: conseguir la siguiente instrucción a ejecutar.
2. **Decode**: a partir de un chorizo de bits (`0110 1110 1100 1100`) *adivinar* de qué instrucción se trata.
3. **Memory fetch**: de ser necesario, se deberán conseguir los datos requeridos por la instrucción.
4. **Execute**: se ejecuta la instrucción.

Ciclo de instrucción

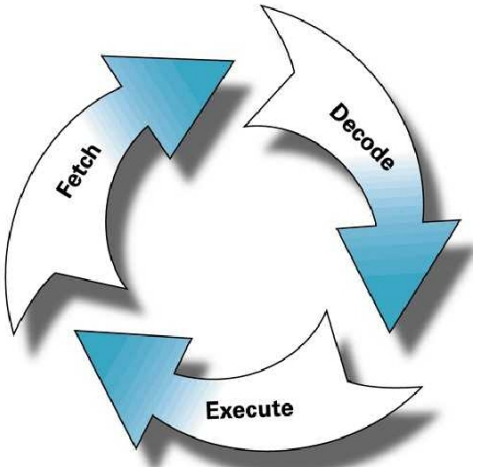
Una computadora *vive* realizando el siguiente ciclo:



1. **Fetch**: conseguir la siguiente instrucción a ejecutar.
2. **Decode**: a partir de un chorizo de bits (`0110 1110 1100 1100`) *adivinar* de qué instrucción se trata.
3. **Memory fetch**: de ser necesario, se deberán conseguir los datos requeridos por la instrucción.
4. **Execute**: se ejecuta la instrucción.
5. **Write back**: si fuera necesario, se deberá almacenar en memoria el resultado de la instrucción.

Ciclo de instrucción

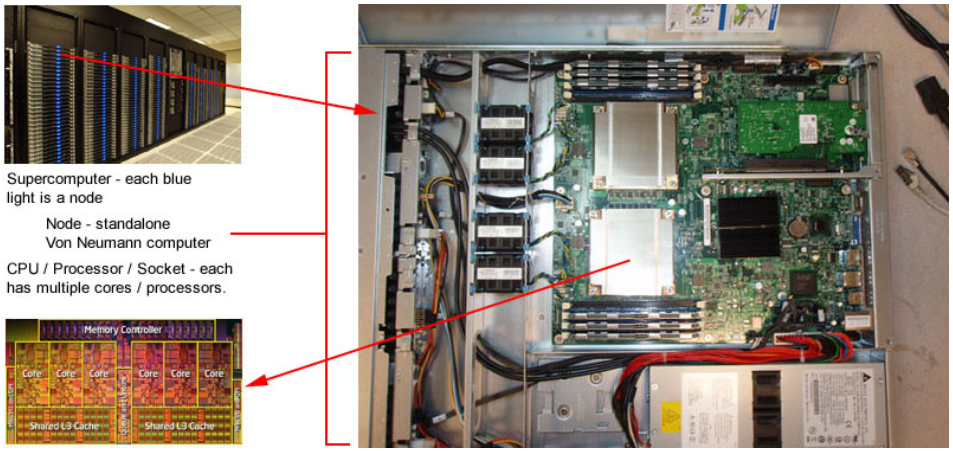
Una computadora *vive* realizando el siguiente ciclo:



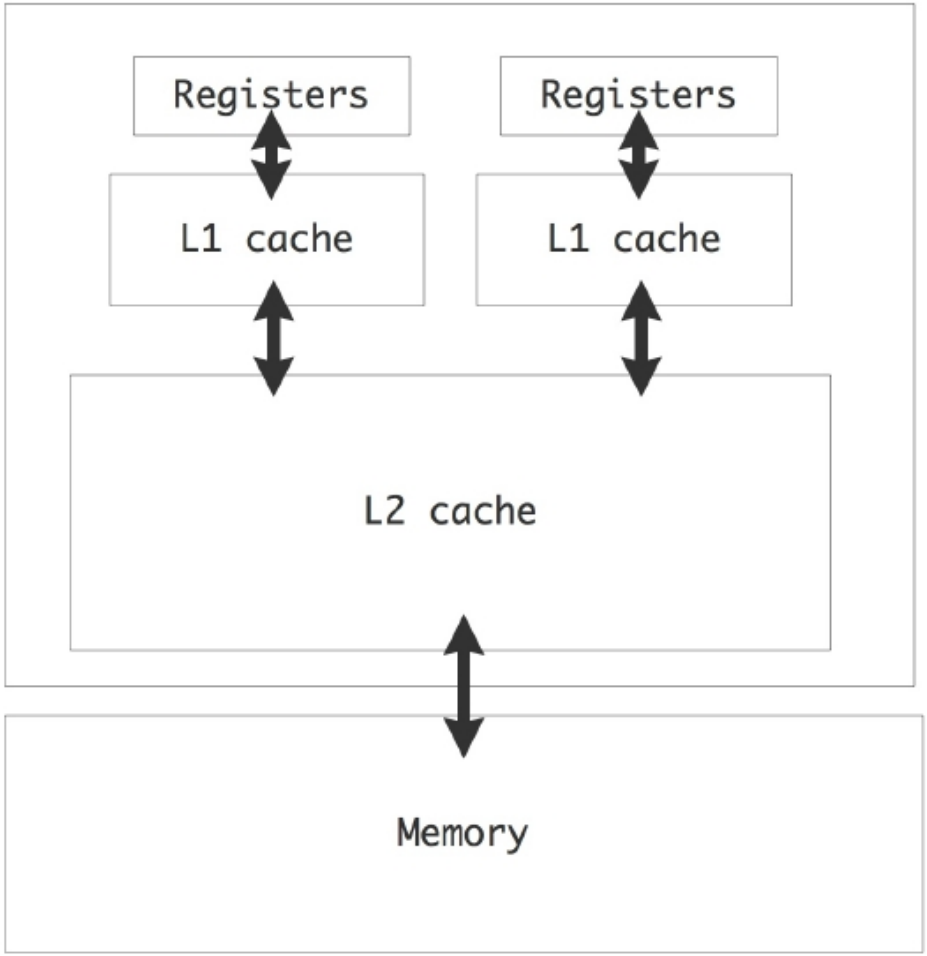
1. **Fetch**: conseguir la siguiente instrucción a ejecutar.
2. **Decode**: a partir de un chorizo de bits (`0110 1110 1100 1100`) *adivinar* de qué instrucción se trata.
3. **Memory fetch**: de ser necesario, se deberán conseguir los datos requeridos por la instrucción.
4. **Execute**: se ejecuta la instrucción.
5. **Write back**: si fuera necesario, se deberá almacenar en memoria el resultado de la instrucción.
6. y se vuelve a empezar...

Accediendo a la memoria

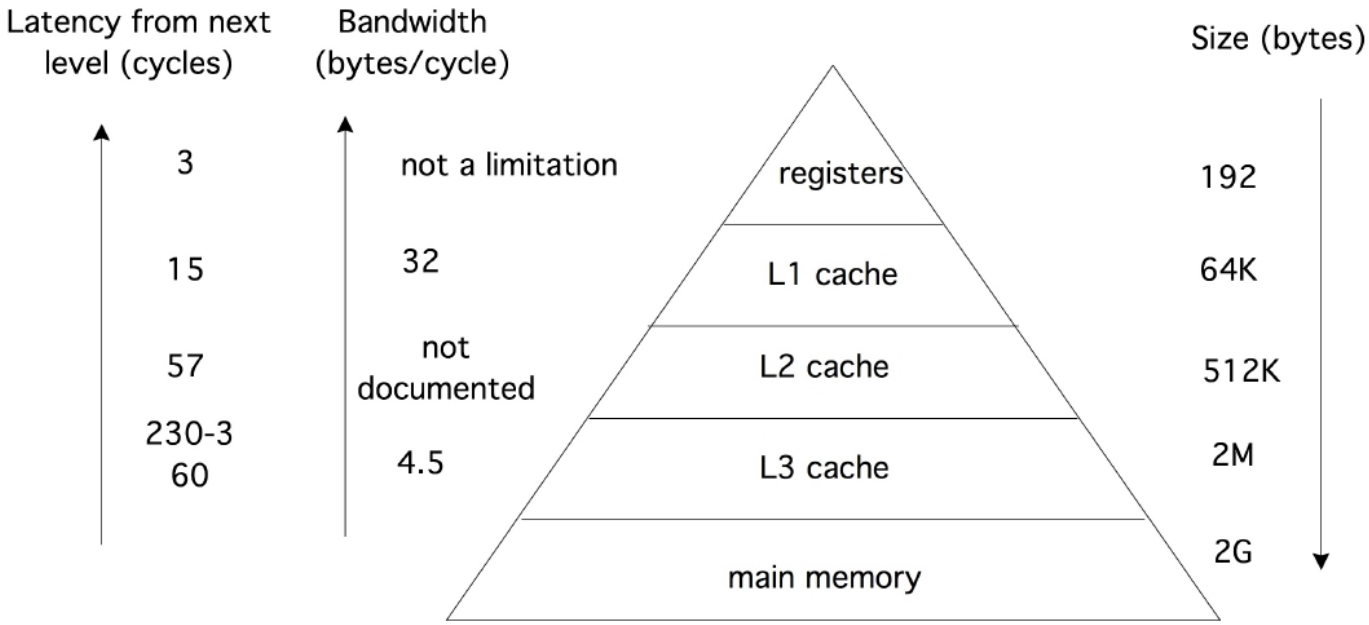
Ir a buscar datos a memoria no es algo tan sencillo...



- **Latencia**: es el tiempo que se tarda desde que se hace un pedido hasta que llega la primer parte a destino.
- **Ancho de banda**: es la velocidad de transferencia una vez que se superó la latencia inicial.

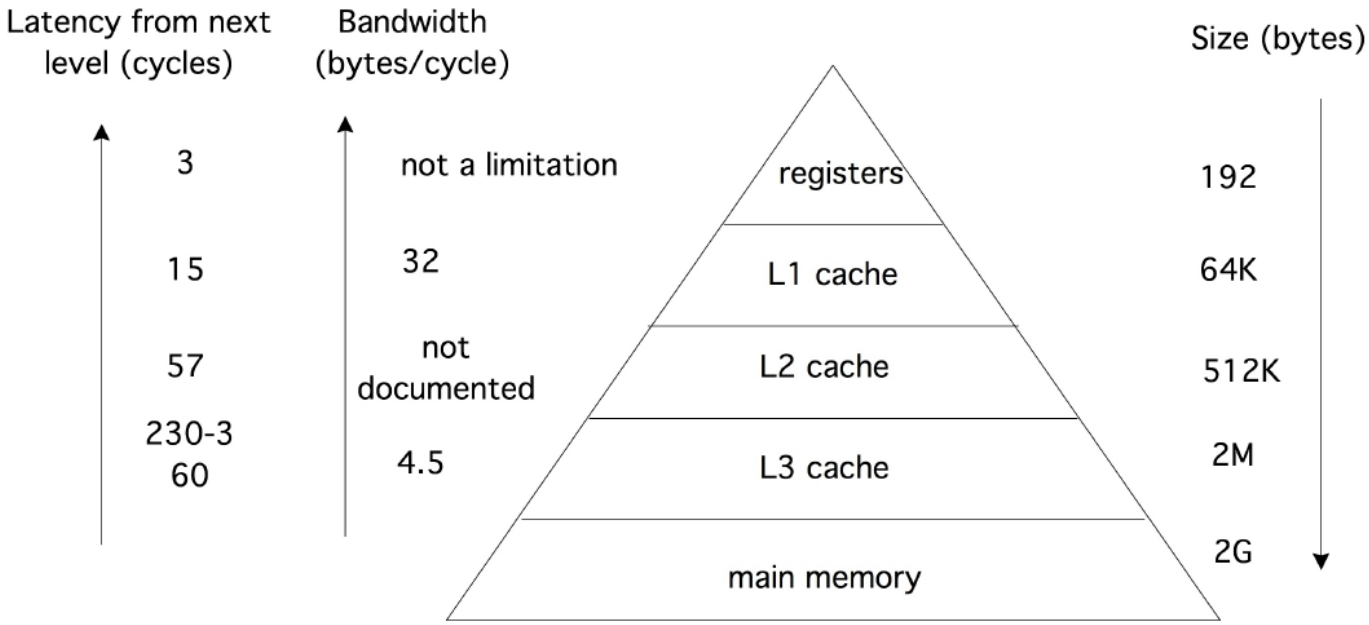


Jerarquía de memoria



Si lo más rápido son los registros... ¿Por qué no es todo registros?

Jerarquía de memoria



Si lo más rápido son los registros... ¿Por qué no es todo registros?

Y la razón principal para esto es...
costo!!!

Programa de ejemplo

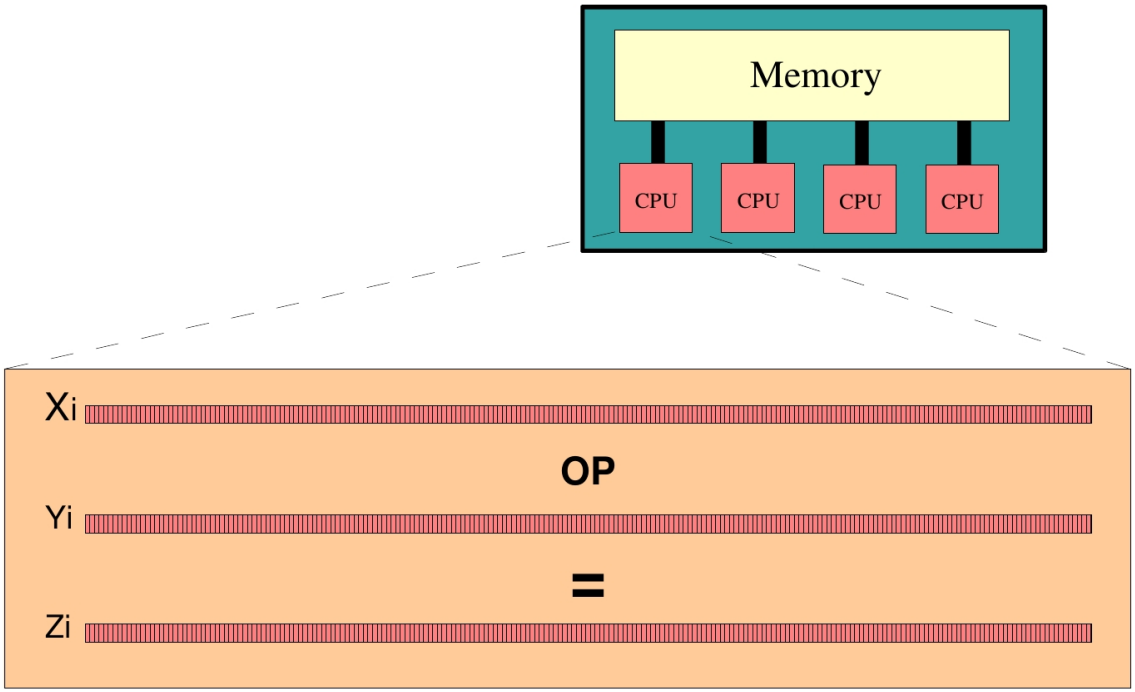
```
 1 | ld   r0, [a]
 2 | ld   r1, [b]
 3 | add  r2, r0, r1   ; r2 = r0 + r1
 4 | ld   r3, [c]
 5 | mul  r4, r2, r3   ; r4 = r2 * r3
 6 | ld   r5, [d]
 7 | push     r5
 8 | push     r4
 9 | op   f   ; f(r4, r5). The result is pushed.
10 | pop  r4
```

`Fetch` and `Decode` no presentan riesgos, se supone que se pueden ejecutar siempre que no estén ocupados. Ejemplo basado en `http://www.moderngpu.com/intro/performance.html`.

Seguimiento de juguete

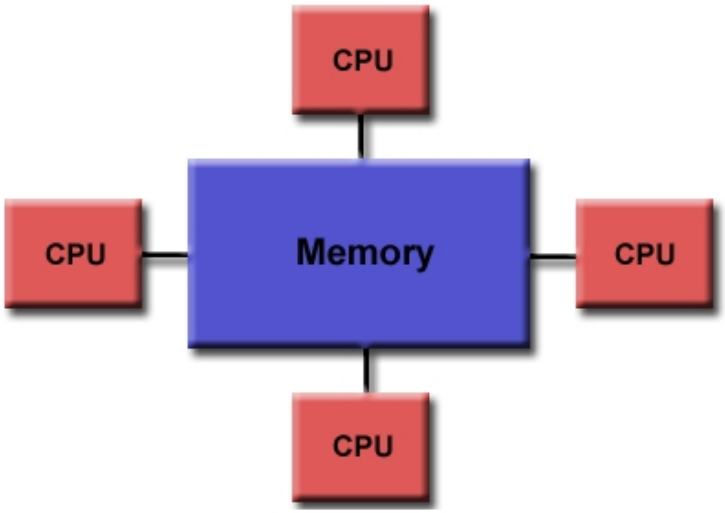
FETCH	DECODE	EXECUTE	OP MEM	WRITE BACK
ld r0, [a]	—	—	—	—
ld r1, [b]	ld r0, [a]	—	—	—
add r2, r0, r1	ld r1, [b]	ld r0, [a]	—	—
ld r3, [c]	add r2, r0, r1	ld r1, [b]	ld r0, [a]	—
mul r4, r2, r3	ld r3, [c]	—STALL—	ld r1, [b]	ld r0, [a]
—STALL—	—STALL—	—STALL—	—STALL—	ld r1, [b]
ld r5, [d]	mul r4, r2, r3	add r2, r0, r1	—STALL—	—STALL—
—STALL—	—STALL—	ld r3, [c]	add r2, r0, r1	—STALL—
—STALL—	ld r5, [d]	—STALL—	ld r3, [c]	add r2, r0, r1
push r5	push r5	—STALL—	—STALL—	ld r3, [c]
push r4	—STALL—	mul r4, r2, r3	—STALL—	—STALL—
—STALL—	—STALL—	ld r5, [d]	mul r4, r2, r3	—STALL—
—STALL—	push r4	—STALL—	ld r5, [d]	mul r4, r2, r3
op f	op f	—STALL—	—STALL—	ld r5, [d]
pop r4	—STALL—	push r5	—STALL—	—STALL—
—STALL—	pop r4	push r4	push r5	—STALL—
—	—	—STALL—	push r4	push r5
—	—	—STALL—	—STALL—	push r4
—	—	op f	—STALL—	—STALL—
—	—	—STALL—	—STALL—	—STALL—
—	—	—STALL—	op f	op f
—	—	pop r4	—STALL—	—STALL—
—	—	—	—STALL—	—STALL—
—	—	—	pop r4	pop r4

Vectorial

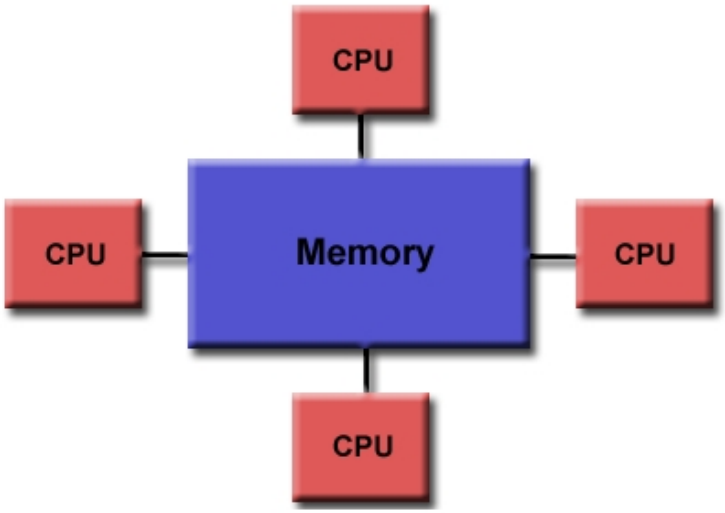


- Se basa en tener un único procesador super-poderoso.
- La idea es aplicar la misma operación a muchos datos en un solo paso.
- El exponente más clásico es la Cray I, dominó el mercado durante los 80s.

Memoria compartida acceso uniforme (UMA)

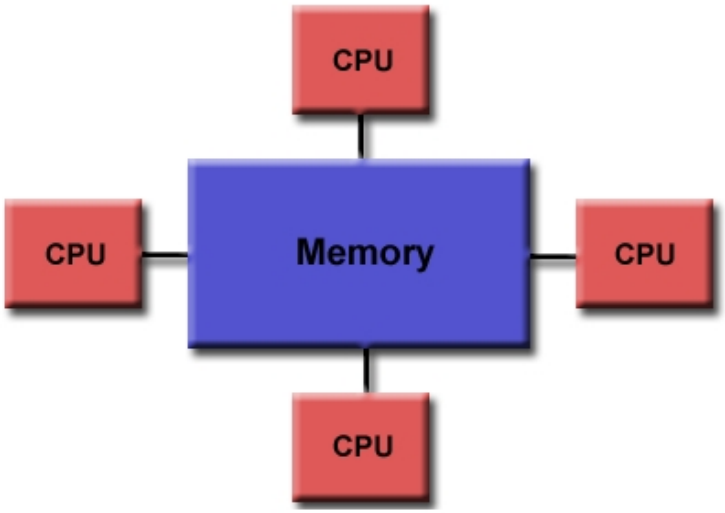


Memoria compartida acceso uniforme (UMA)



- Las más conocidas son las máquinas tipo Symmetric Multiprocessor(SMP)
- Todos los procesadores son idénticos entre sí
- Acceso uniforme a toda la memoria y mismo tiempo de acceso para llegar a cualquier parte del mapa de memoria.

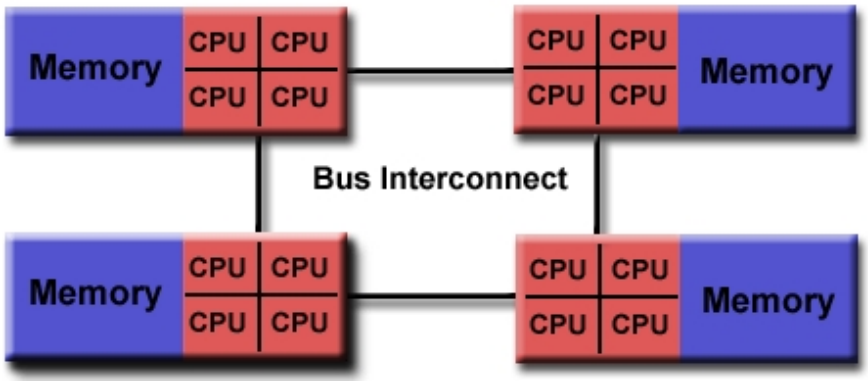
Memoria compartida acceso uniforme (UMA)



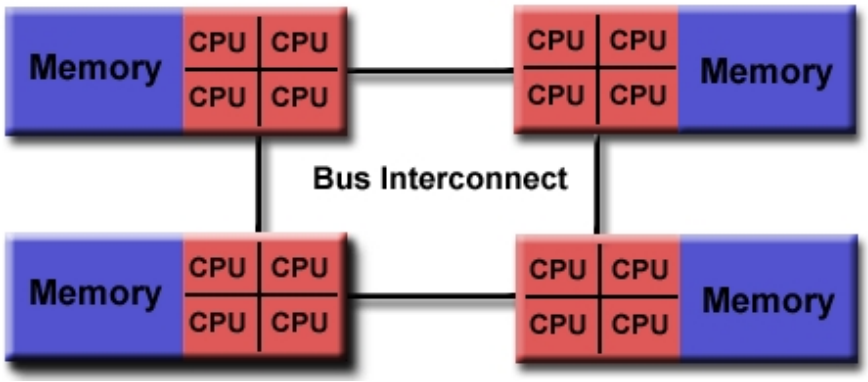
- Las más conocidas son las máquinas tipo Symmetric Multiprocessor(SMP)
- Todos los procesadores son idénticos entre sí
- Acceso uniforme a toda la memoria y mismo tiempo de acceso para llegar a cualquier parte del mapa de memoria.

- A veces se las llama `CC-UMA` (Cache Coherent UMA).
- La coherencia de cache (Cache coherency) significa que si un procesador actualiza el dato almacenado en la memoria, todos los procesadores se enteran del cambio. La coherencia de cache se implementa a nivel hardware.

Memoria compartida acceso no uniforme (NUMA)

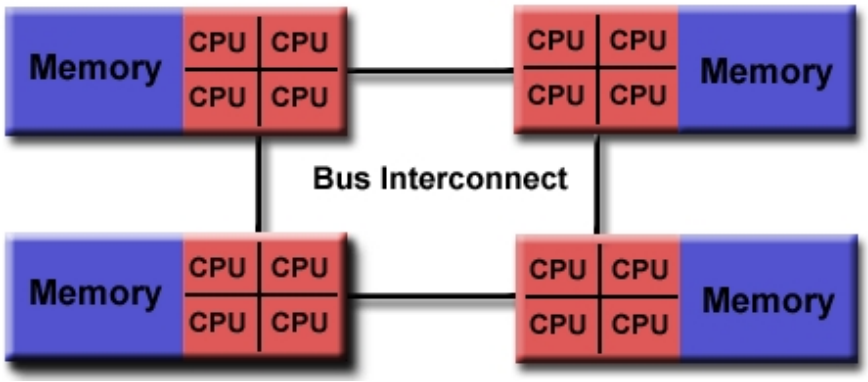


Memoria compartida acceso no uniforme (NUMA)



- A menudo se arman conectando dos o más SMPs
- Una SMP puede acceder directamente a la memoria de otra SMP
- No se tienen un tiempo de acceso uniforme a todas las partes de la memoria

Memoria compartida acceso no uniforme (NUMA)



- A menudo se arman conectando dos o más SMPs
- Una SMP puede acceder directamente a la memoria de otra SMP
- No se tienen un tiempo de acceso uniforme a todas las partes de la memoria

- Acceder a través del link es considerablemente más lento.
- Si hubiera coherencia de cache, se las conoce como `CC-NUMA`(Cache Coherent NUMA)

Memoria compartida

Ventajas:

- El espacio global de memoria le hace la vida *fácil* al programador.
- Compartir datos entre tareas es rápido y directo.

Memoria compartida

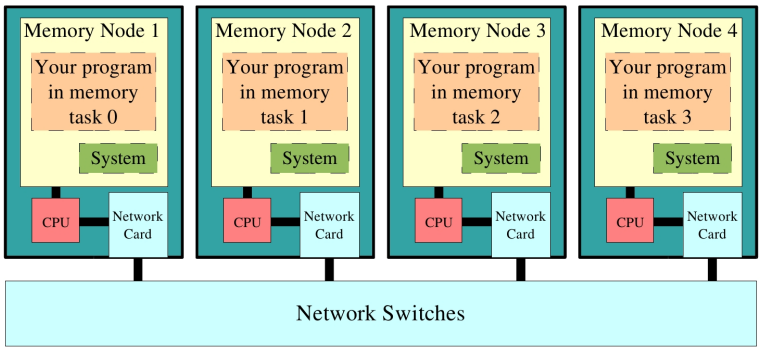
Ventajas:

- El espacio global de memoria le hace la vida *fácil* al programador.
- Compartir datos entre tareas es rápido y directo.

Desventajas:

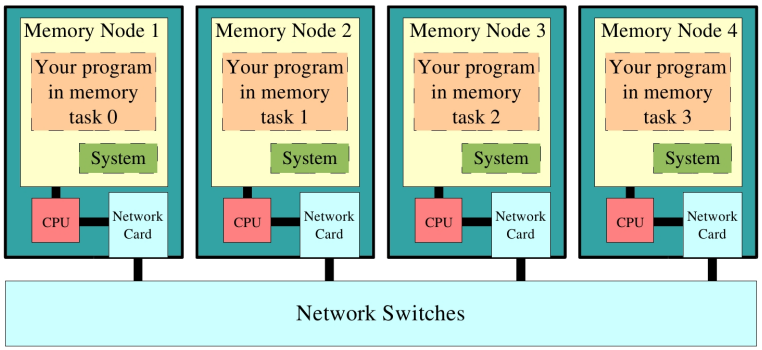
- Falta de escabalibilidad: agregar más procesadores puede incrementar geométricamente el tráfico en el camino memoria-procesador. Para el caso de sistemas con coherencia de cache, la complejidad de mantenerla también se incrementa geométricamente.
- Es responsabilidad del programador implementar la sincronización para asegurarse el acceso correcto a la memoria global.
- Costo: se vuelve cada vez más díficil y caro diseñar y producir máquinas con memoria compartida a medida que se aumenta la cantidad de procesadores.

Memoria distribuida: clusters



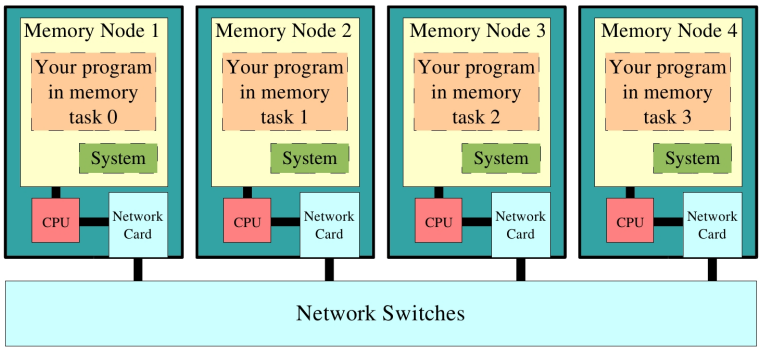
- Requiere una red de comunicación para conectar los nodos.
- Cada procesador tiene su propia memoria local. El espacio de memoria de un procesador no puede accederse desde otro directamente. No hay espacio global de memoria compartido entre todos.

Memoria distribuida: clusters



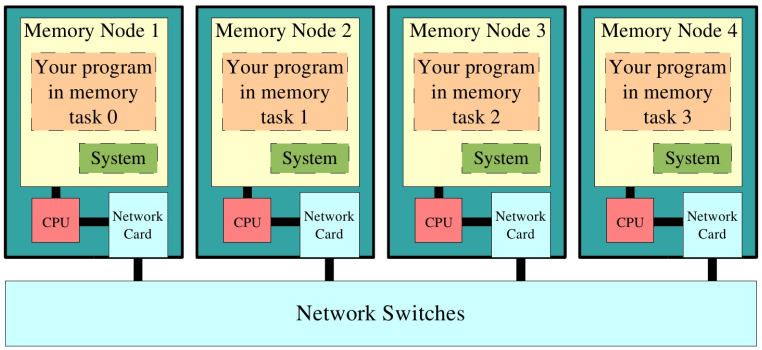
- Requiere una red de comunicación para conectar los nodos.
- Cada procesador tiene su propia memoria local. El espacio de memoria de un procesador no puede accederse desde otro directamente. No hay espacio global de memoria compartido entre todos.
- Cada procesador opera independientemente del resto: los cambios en su espacio de memoria no tienen efecto en el resto de los procesadores. No tiene sentido pensar en coherencia de cache.

Memoria distribuida: clusters



- Requiere una red de comunicación para conectar los nodos.
- Cada procesador tiene su propia memoria local. El espacio de memoria de un procesador no puede accederse desde otro directamente. No hay espacio global de memoria compartido entre todos.
- Cada procesador opera independientemente del resto: los cambios en su espacio de memoria no tienen efecto en el resto de los procesadores. No tiene sentido pensar en coherencia de cache.
- Cuando un procesador necesita acceder a datos que *tiene* otro, es tarea del programador usar explícitamente un mecanismo para transferir ese dato. La sincronización entre las tareas es completa responsabilidad del programador

Memoria distribuida: clusters



- Requiere una red de comunicación para conectar los nodos.
- Cada procesador tiene su propia memoria local. El espacio de memoria de un procesador no puede accederse desde otro directamente. No hay espacio global de memoria compartido entre todos.
- Cada procesador opera independientemente del resto: los cambios en su espacio de memoria no tienen efecto en el resto de los procesadores. No tiene sentido pensar en coherencia de cache.
- Cuando un procesador necesita acceder a datos que *tiene* otro, es tarea del programador usar explícitamente un mecanismo para transferir ese dato. La sincronización entre las tareas es completa responsabilidad del programador
- La infraestructura de red usada para interconectar el cluster varía desde un switch común(Ethernet) hasta equipamiento ultra-específico(Infiniband).

Memoria distribuida

Ventajas:

- La cantidad de memoria escala con el número de procesadores. Es posible incrementar el número de procesadores y el tamaño de la memoria proporcialmente.
- Cada procesador puede acceder *rápidamente* a su propia memoria local sin interferencias y sin el overhead que se incurre al tener que mantener la coherencia de cache.
- Relación costo-beneficio: se puede usar hardware común, conocido como *off-the-shelf* y obtener una performance muy razonable.

Memoria distribuida

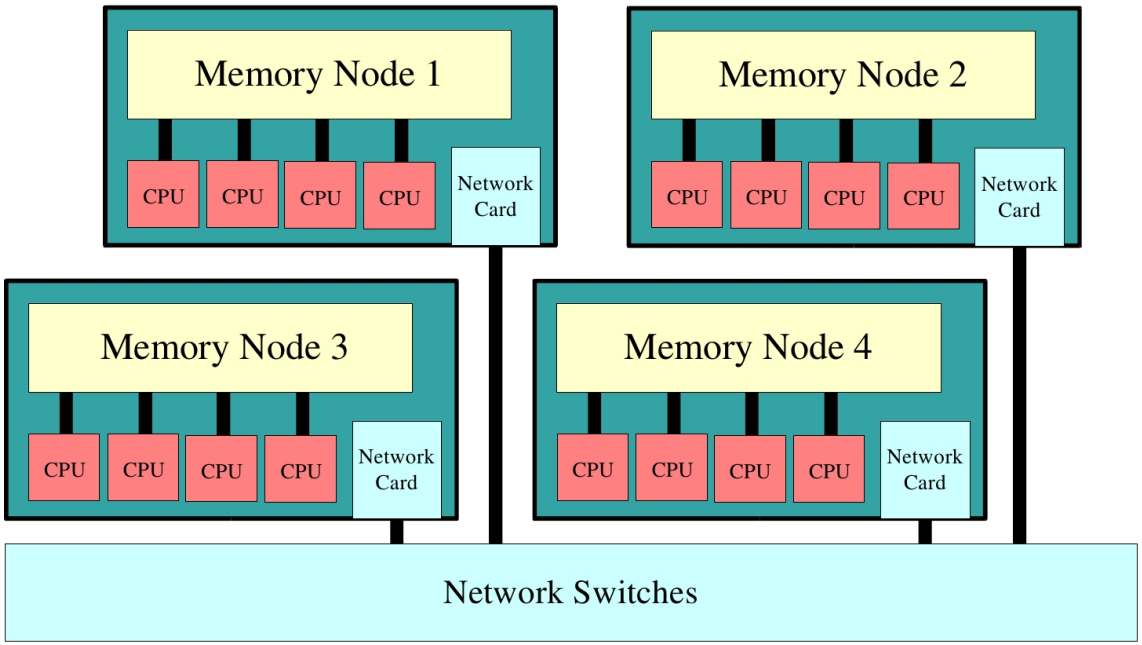
Ventajas:

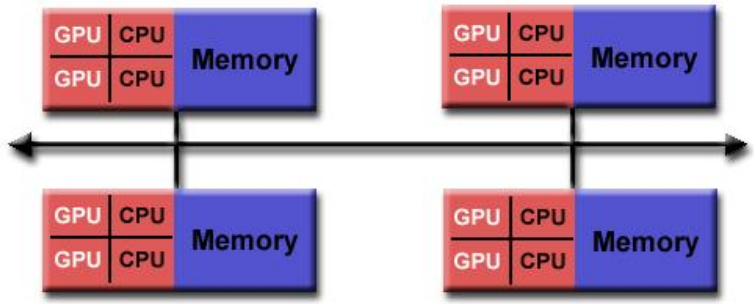
- La cantidad de memoria escala con el número de procesadores. Es posible incrementar el número de procesadores y el tamaño de la memoria proporcialmente.
- Cada procesador puede acceder *rápidamente* a su propia memoria local sin interferencias y sin el overhead que se incurre al tener que mantener la coherencia de cache.
- Relación costo-beneficio: se puede usar hardware común, conocido como *off-the-shelf* y obtener una performance muy razonable.

Desventajas:

- El programador es responsable de muchos detalles (demasiados, dirían algunos) asociados con la comunicación de datos entre procesos.
- A veces resulta **muy** complicado adaptar código existente basado en memoria compartida a este arquitectura.
- El tiempo de acceso a los datos no es uniforme (¡y varía mucho!)

Híbridas


Memory Node 1
CPU
CPU
CPU
CPU
Network Card
Memory Node 2
CPU
CPU
CPU
CPU
Network Card
Memory Node 3
CPU
CPU
CPU
CPU
Network Card
Memory Node 4
CPU
CPU
CPU
CPU
Network Card
Network Switches


GPU
CPU
GPU
CPU
Memory
GPU
CPU
GPU
CPU
Memory
GPU
CPU
GPU
CPU
Memory
GPU
CPU
GPU
CPU
Memory

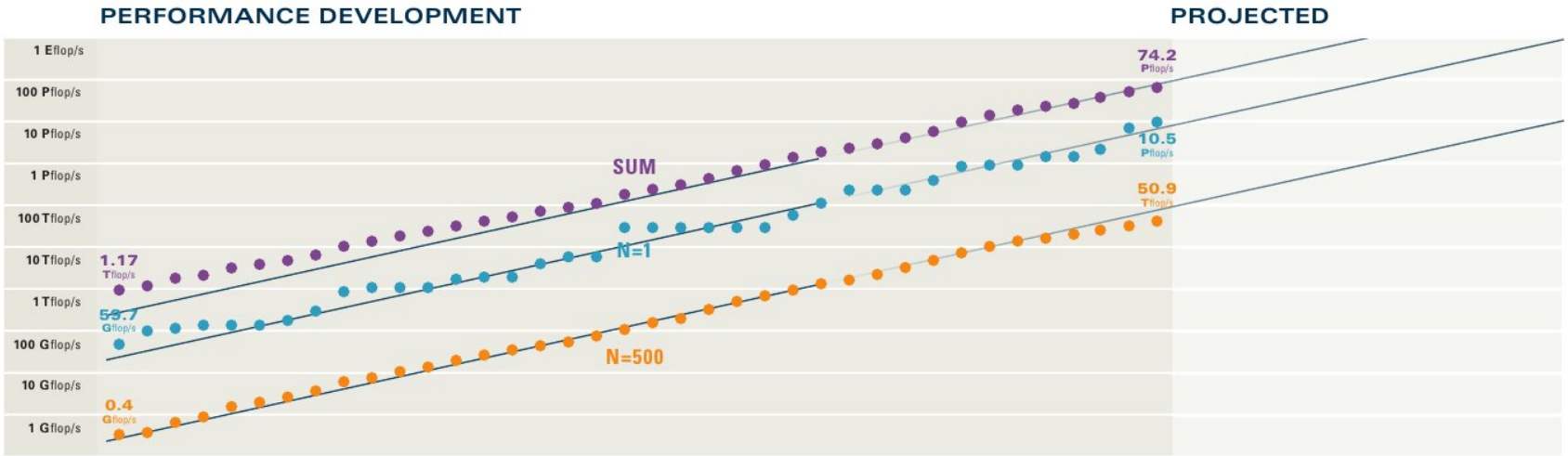
El rendimiento es **MUY** relativo de las aplicaciones

TOP500® NOVEMBER 2011

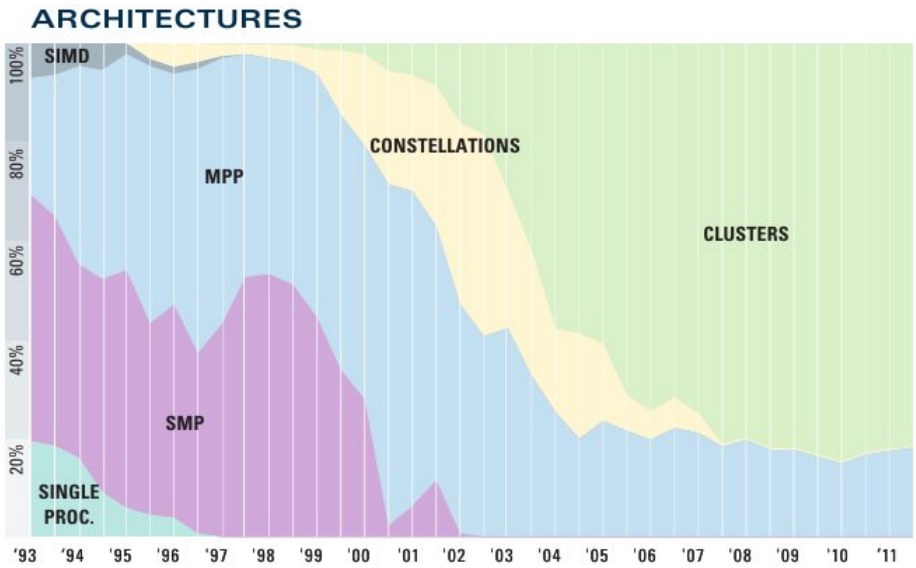
PRESENTED BY UNIVERSITY OF MANNHEIM · ICL UT INNOVATIVE COMPUTING LABORATORY THE UNIVERSITY OF TENNESSEE · BERKELEY LAB Lawrence Berkeley National Laboratory

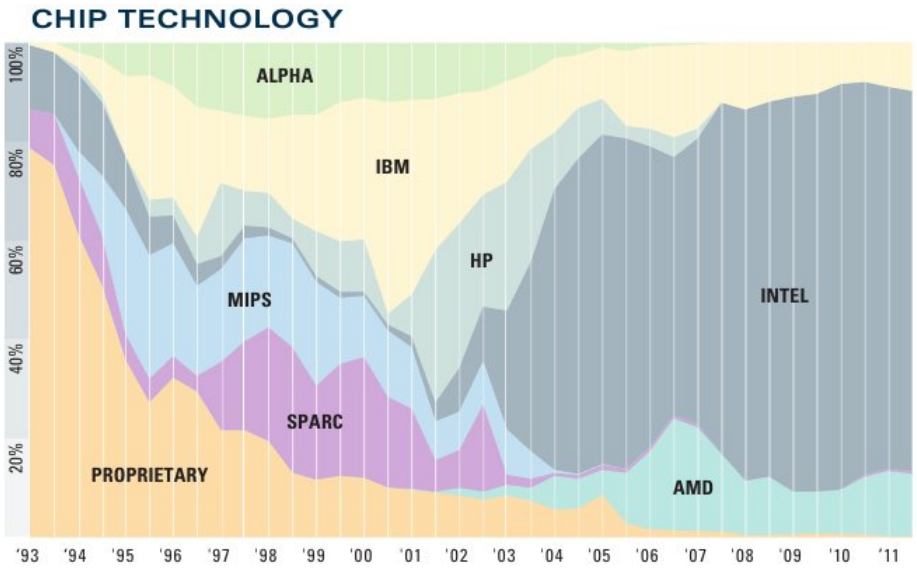
FIND OUT MORE AT www.top500.org

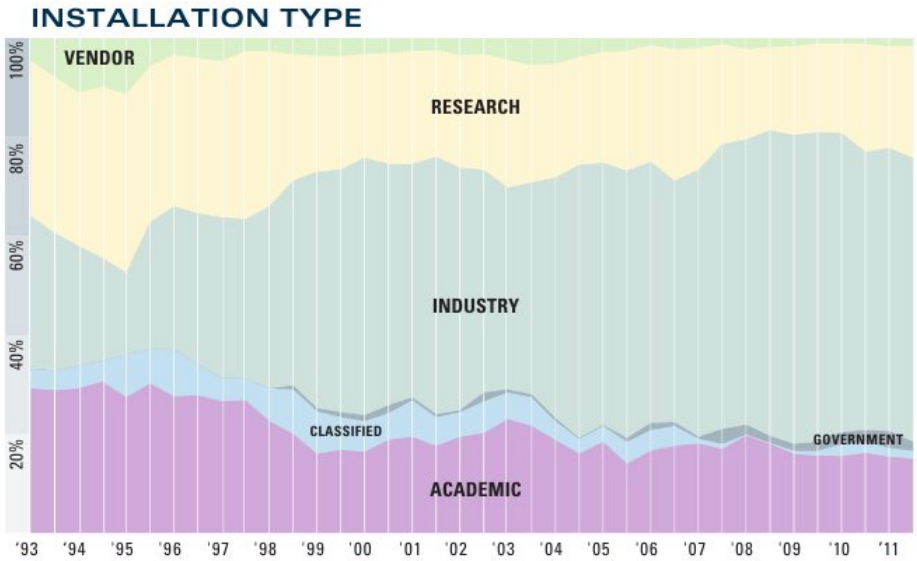
	NAME/MANUFACTURER/COMPUTER	SITE	COUNTRY	CORES	R_{max} Pflop/s
1	**K computer** SPARC64 VIIIfx 2.0GHz, Tofu interconnect	RIKEN	Japan	705,024	10.5
2	**Tianhe-1A** 6-core Intel X5670 2.93 GHz + Nvidia M2050 GPU w/custom interconnect	NUDT/NSCC/Tianjin	China	186,368	2.57
3	**Jaguar** Cray XT-5 6-core AMD 2.6 GHz w/custom interconnect	DOE/OS/ORNL	USA	224,162	1.76
4	**Nebulae** Dawning TC3600 Blade Intel X5650 2.67 GHz, NVidia Tesla C2050 GPU w/ Iband	NSCS	China	120,640	1.27
5	**Tsubame 2.0** HP Proliant SL390s G7 nodes (Xeon X5670 2.93GHz) , NVIDIA Tesla M2050 GPU w/Iband	TiTech	Japan	73,278	1.19



El rendimiento es **MUY** relativo de las aplicaciones







HPLINPACK

A Portable Implementation of the High Performance Linpack Benchmark for Distributed Memory Computers

Algorithm: recursive panel factorizations, multiple lookahead depths, bandwidth reducing swapping

Easy to install, only needs MPI + BLAS or VSIPL

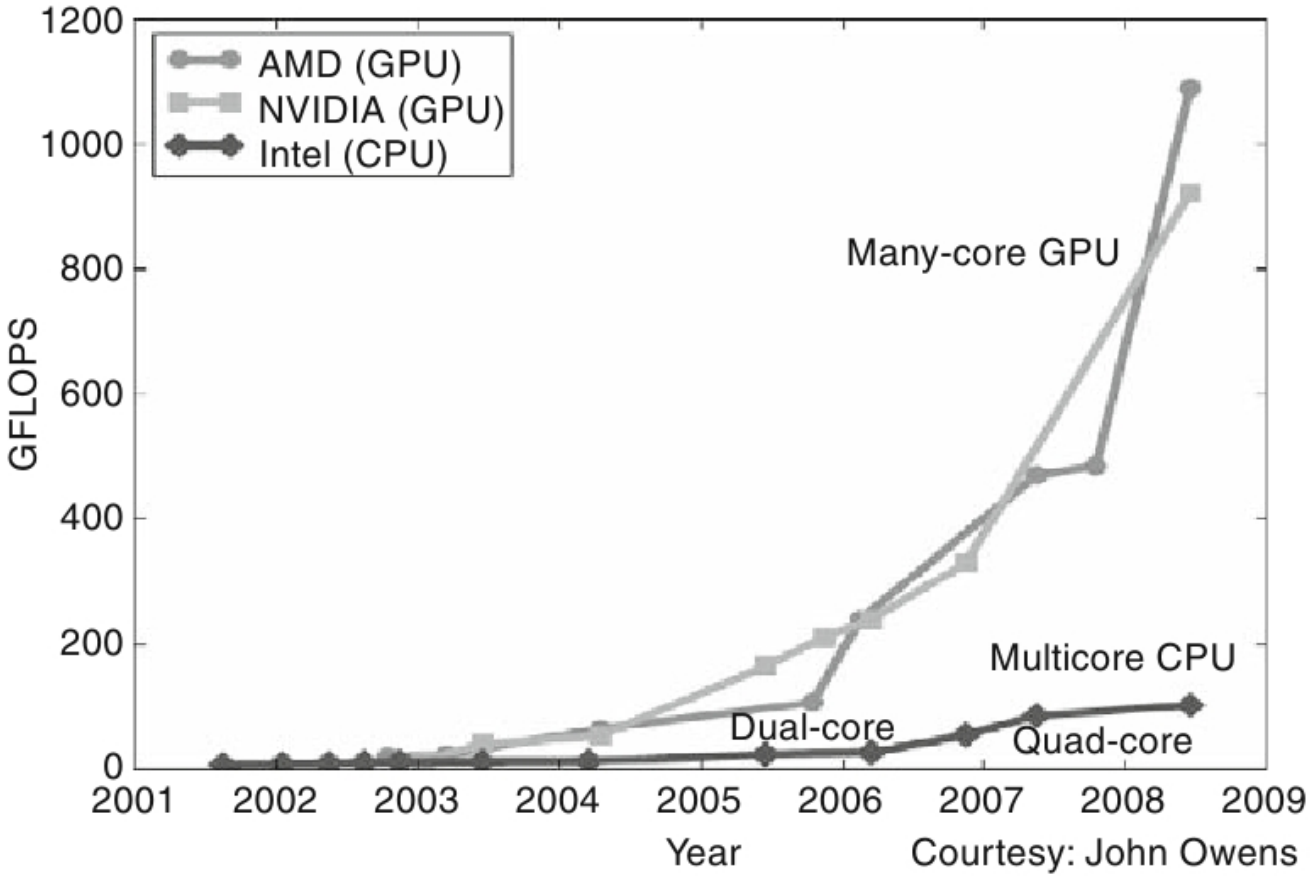
Highly scalable and efficient from the smallest cluster to the largest supercomputers in the world

FIND OUT MORE AT **http://icl.eecs.utk.edu/hpl/**

GPU

- GPU: Graphics Processing Unit
- Se diseñaron para gráficos generados en tiempo real.
- Se hayan presentes en la mayoría de las PCs
- Cada vez más realismo y efectos.

GPGPU



Parece que no hubiera con qué darle...

Prerequisitos

- A CUDA-enabled graphics processor: la primera fue la `GeForce 8800 GTX`, en todas se puede desarrollar y probar.

Prerequisitos

- A CUDA-enabled graphics processor: la primera fue la `GeForce 8800 GTX`, en todas se puede desarrollar y probar.
- Device driver: se consigue de la página de Nvidia: `www.nvidia.com/cuda` y bajar los drivers. Están para Windows y Linux.

Prerequisitos

- A CUDA-enabled graphics processor: la primera fue la `GeForce 8800 GTX`, en todas se puede desarrollar y probar.
- Device driver: se consigue de la página de Nvidia: `www.nvidia.com/cuda` y bajar los drivers. Están para Windows y Linux. Con estas dos cosas se pueden correr aplicaciones ya desarrolladas para CUDA.

Prerequisitos

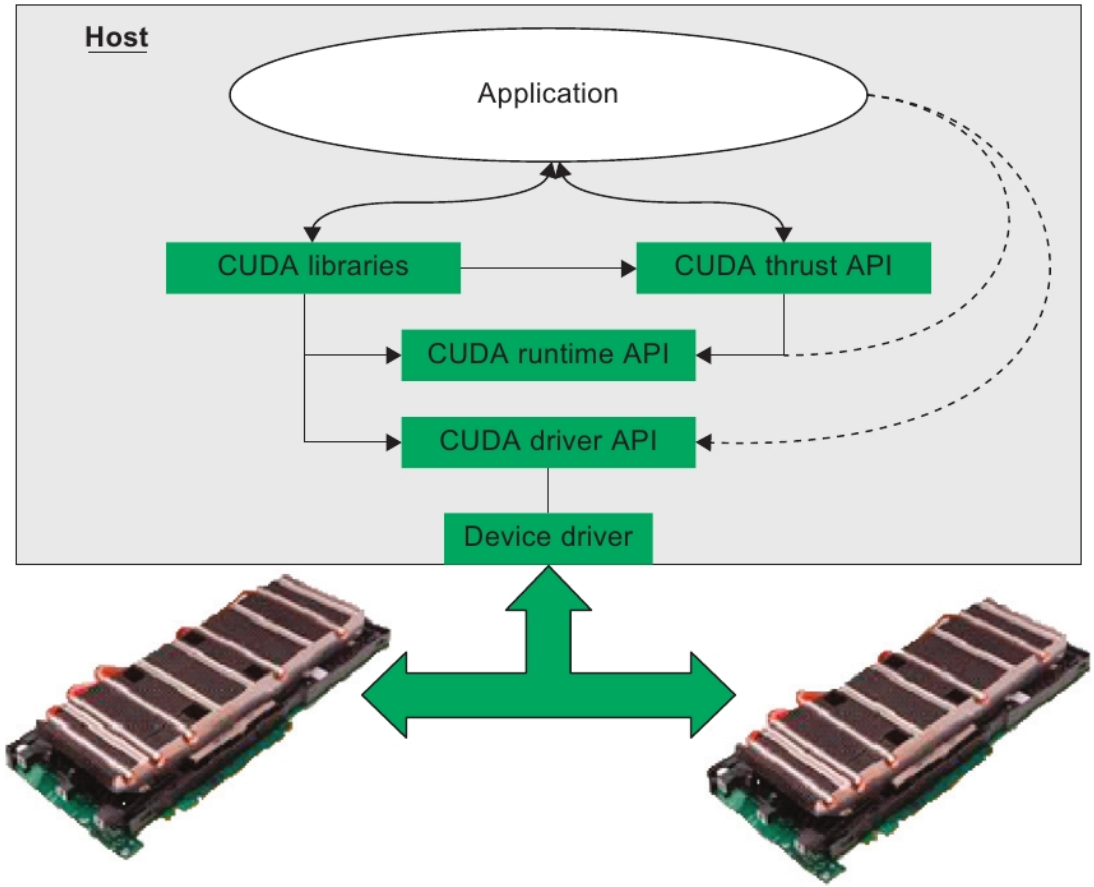
- A CUDA-enabled graphics processor: la primera fue la `GeForce 8800 GTX`, en todas se puede desarrollar y probar.
- Device driver: se consigue de la página de Nvidia: `www.nvidia.com/cuda` y bajar los drivers. Están para Windows y Linux. Con estas dos cosas se pueden correr aplicaciones ya desarrolladas para CUDA.
- A CUDA development toolkit: para desarrollar aplicaciones usando CUDA, es necesario el entorno de desarrollo: `http://developer.nvidia.com/object/gpucomputing.html`. Con esto vienen muchas aplicaciones de ejemplo ya programadas para inspirarse.

Prerequisitos

- A CUDA-enabled graphics processor: la primera fue la `GeForce 8800 GTX`, en todas se puede desarrollar y probar.
- Device driver: se consigue de la página de Nvidia: `www.nvidia.com/cuda` y bajar los drivers. Están para Windows y Linux. Con estas dos cosas se pueden correr aplicaciones ya desarrolladas para CUDA.
- A CUDA development toolkit: para desarrollar aplicaciones usando CUDA, es necesario el entorno de desarrollo: `http://developer.nvidia.com/object/gpucomputing.html`. Con esto vienen muchas aplicaciones de ejemplo ya programadas para inspirarse.
- A standard C compiler: puede ser Windows Visual Studio en plataformas Microsoft o GNU C Compiler(`gcc`).

API's de CUDA

1. The data-parallel *C++ Thrust*.
2. The runtime API: se puede usar tanto desde *C* o *C++*.
3. The driver API: se puede usar tanto desde *C* o *C++*.



Primer ejemplo

```
#include <stdio.h>
int main( void ) {
  printf( "Hello, World!\n" );
  return 0;
}
```

[1] Todos los ejemplos son pensados para Linux

Primer ejemplo

```
#include <stdio.h>
int main( void ) {
  printf( "Hello, World!\n" );
  return 0;
}
```

Para tener el primer programa CUDA se debe hacer[1]:

- Tipear el programa en su editor preferido, guardarlo como `tontin.cu`.

[1] Todos los ejemplos son pensados para Linux

Primer ejemplo

```
#include <stdio.h>
int main( void ) {
  printf( "Hello, World!\n" );
  return 0;
}
```

Para tener el primer programa CUDA se debe hacer[1]:

- Tipear el programa en su editor preferido, guardarlo como `tontin.cu`.
- Generar el ejecutable: `nvcc -o tontin tontin.cu`

[1] Todos los ejemplos son pensados para Linux

Primer ejemplo

```
#include <stdio.h>
int main( void ) {
  printf( "Hello, World!\n" );
  return 0;
}
```

Para tener el primer programa CUDA se debe hacer[1]:

- Tipear el programa en su editor preferido, guardarlo como `tontin.cu`.
- Generar el ejecutable: `nvcc -o tontin tontin.cu`
- Ejecutarlo: `./tontin`

[1] Todos los ejemplos son pensados para Linux

Primer ejemplo

```
#include <stdio.h>
int main( void ) {
  printf( "Hello, World!\n" );
  return 0;
}
```

Para tener el primer programa CUDA se debe hacer[1]:

- Tipear el programa en su editor preferido, guardarlo como `tontin.cu`.
- Generar el ejecutable: `nvcc -o tontin tontin.cu`
- Ejecutarlo: `./tontin`

Este programa sirve para dos cosas fundamentales:

1. Probar que todo este bien instalado (compilador, bibliotecas, etc).

[1] Todos los ejemplos son pensados para Linux

Primer ejemplo

```
#include <stdio.h>
int main( void ) {
  printf( "Hello, World!\n" );
  return 0;
}
```

Para tener el primer programa CUDA se debe hacer[1]:

- Tipear el programa en su editor preferido, guardarlo como `tontin.cu`.
- Generar el ejecutable: `nvcc -o tontin tontin.cu`
- Ejecutarlo: `./tontin`

Este programa sirve para dos cosas fundamentales:

1. Probar que todo este bien instalado (compilador, bibliotecas, etc).
2. Y fundamentalmente para...

[1] Todos los ejemplos son pensados para Linux

Primer ejemplo

```
#include <stdio.h>
int main( void ) {
  printf( "Hello, World!\n" );
  return 0;
}
```

Para tener el primer programa CUDA se debe hacer[1]:

- Tipear el programa en su editor preferido, guardarlo como `tontin.cu`.
- Generar el ejecutable: `nvcc -o tontin tontin.cu`
- Ejecutarlo: `./tontin`

Este programa sirve para dos cosas fundamentales:

1. Probar que todo este bien instalado (compilador, bibliotecas, etc).
2. Y fundamentalmente para... Darse cuenta que el programa es una B8*u*Ez... se podría compilar directamente con `gcc` y debería ser lo mismo.

[1] Todos los ejemplos son pensados para Linux

Primer ejemplo CUDA

```
#include <stdio.h>
#include <cuda.h>
__global__ void kernel(void) {

}
int main() {
   printf("Antes de llamar\n");
   kernel<<<1,1>>>();
   printf("Hello World\n");
   return 0;
}
```

Primer ejemplo CUDA

```
#include <stdio.h>
#include <cuda.h>
__global__ void kernel(void) {

}
int main() {
   printf("Antes de llamar\n");
   kernel<<<1,1>>>();
   printf("Hello World\n");
   return 0;
}
```

Dos cosas nuevitas que aparecen:

- Una función llamada *kernel* (vacía) que tiene el calificador *__global__*.
- Una llamada a la función *kernel* con algo del estilo `<<<1,1>>>`.

Extensiones a C en CUDA

- *__host__HostFunc()* se ejecuta en el *host* y se llama desde el *host*: son las funciones normales y conocidas de *C* que hicimos hasta ahora en nuestra vida.

Extensiones a C en CUDA

- *__host__HostFunc()* se ejecuta en el *host* y se llama desde el *host*: son las funciones normales y conocidas de *C* que hicimos hasta ahora en nuestra vida.
- *__global__Kernelfunc()* se ejecuta en el *device* y se llama desde el *host*: esta corresponde a la función que va a poner a ejecutar la placa de video.

Extensiones a C en CUDA

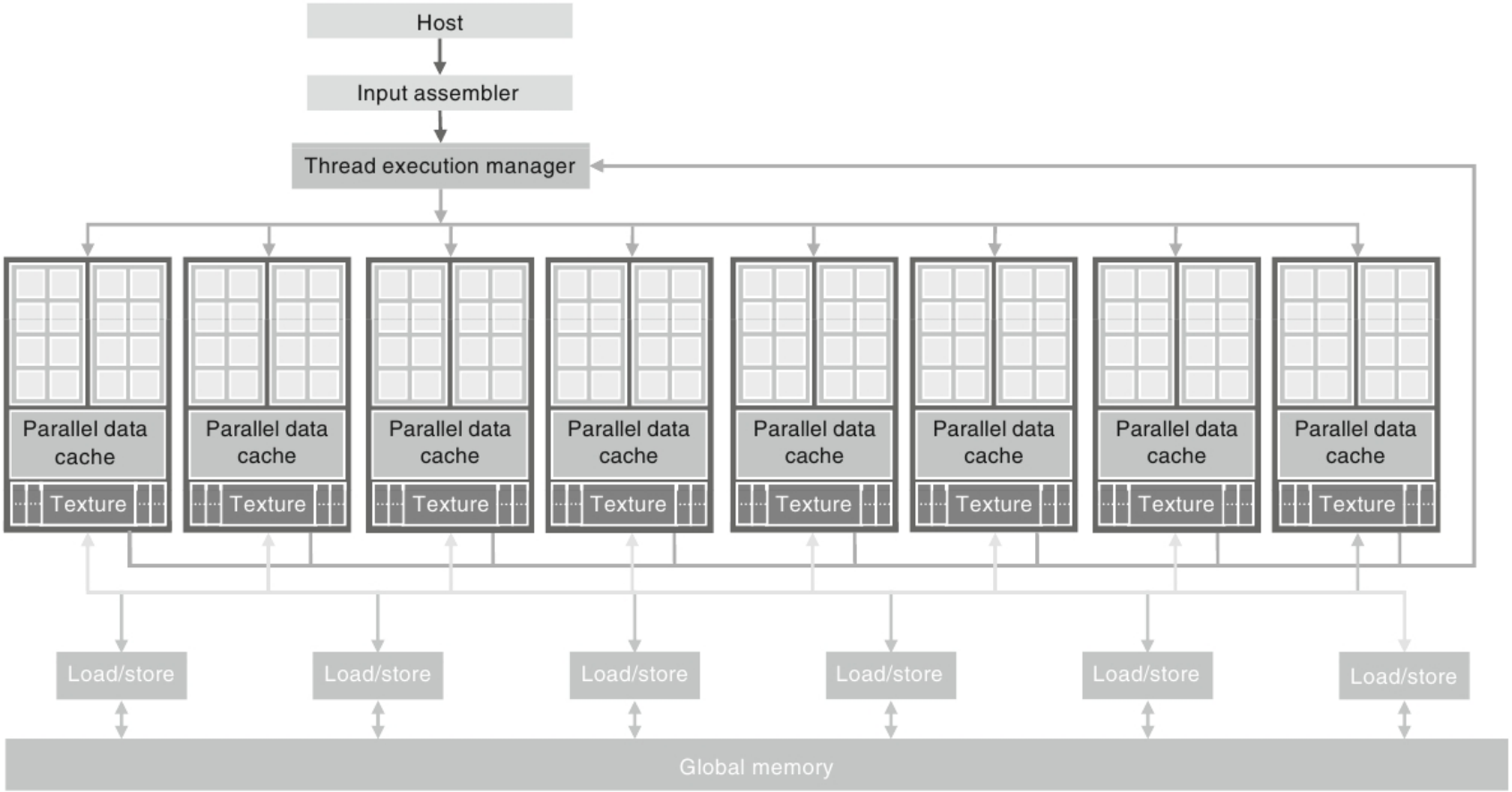
- *__host__HostFunc()* se ejecuta en el *host* y se llama desde el *host*: son las funciones normales y conocidas de *C* que hicimos hasta ahora en nuestra vida.
- *__global__Kernelfunc()* se ejecuta en el *device* y se llama desde el *host*: esta corresponde a la función que va a poner a ejecutar la placa de video.
- *__device__func()* se ejecuta en el *device* y se llama desde el *device*: solo se puede llamar desde un kernel o desde otra función de este tipo. Deben ser funciones *sencillas*(por ejemplo, no pueden ser recursivas).

Extensiones a C en CUDA

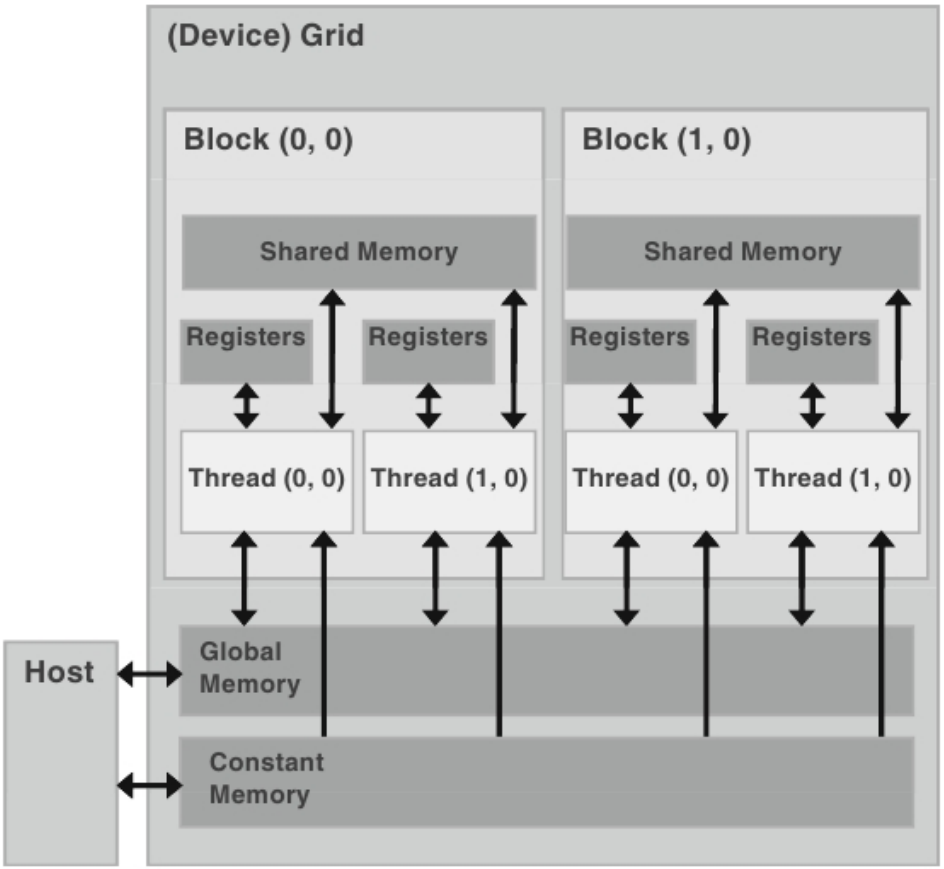
- *__host__HostFunc()* se ejecuta en el *host* y se llama desde el *host*: son las funciones normales y conocidas de *C* que hicimos hasta ahora en nuestra vida.
- *__global__Kernelfunc()* se ejecuta en el *device* y se llama desde el *host*: esta corresponde a la función que va a poner a ejecutar la placa de video.
- *__device__func()* se ejecuta en el *device* y se llama desde el *device*: solo se puede llamar desde un kernel o desde otra función de este tipo. Deben ser funciones *sencillas*(por ejemplo, no pueden ser recursivas).

El estilo de programción en CUDA es que se empieza con una función en el *host*, luego se lanzan uno o más kernels.

Arquitectura de una GPU



Jerarquía de memoria de una GPU



El *device* puede:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Mientras que el *host* puede:

- Transfer data to/from per-grid global and constant memories

¿Qué hay en la máquina?

Se puede utilizar el comando `deviceQuery` o `deviceQueryDrv` (linkeada estáticamente) para averiguar qué dispositivo y qué características tiene, entre otras, te tira:

Device 0: ”GeForce GTX 260”

CUDA Driver Version:	4.2
CUDA Capability Major/Minor version number:	1.3
Total amount of global memory:	895 MBytes (938803200 bytes)
(27) Multiprocessors x (8) CUDA Cores/MP:	216 CUDA Cores
GPU Clock rate:	1.24 GHz
Memory Clock rate:	1000.00 Mhz
Memory Bus Width:	448-bit
Max Texture Dimension Sizes	1D=(8192) 2D=(65536,32768) 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers	1D=(8192) x 512, 2D=(8192,8192) x 512
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	16384 bytes
Total number of registers available per block:	16384
Warp size:	32
Maximum number of threads per block:	512
Maximum sizes of each dimension of a block:	512 x 512 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 1

Hagamos algo con la GPU

```
#include <stdio.h>
#include <cuda.h>
__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}
int main() {
    int c=0;
    int *dev_c;
    cudaMalloc((void**)&dev_c, sizeof(int));
    add<<<1,1>>>( 2, 7, dev_c );
    cudaMemcpy( &c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
    printf( "2 + 7 = %d\n", c );
    cudaFree( dev_c );
    return 0;
}
```

cudaMalloc

```
cudaMalloc((void**)&dev_c, sizeof(int));
```

cudaMalloc

```
cudaMalloc((void**)&dev_c, sizeof(int));
```

La función `cudaMalloc`:

- Tiene aire a la función conocida *malloc*.
- Se puede chequear el resultado para saber si anduvo bien.
- Reserva espacio en la memoria global del *device*.

cudaMalloc

```
cudaMalloc((void**)&dev_c, sizeof(int));
```

La función `cudaMalloc`:

- Tiene aire a la función conocida *malloc*.
- Se puede chequear el resultado para saber si anduvo bien.
- Reserva espacio en la memoria global del *device*.
- Recibe dos parámetros:
  1. Una referencia para ubicar el objeto construido.
  2. El tamaño del objeto a ubicar en cantidad de bytes.

cudaMalloc

```
cudaMalloc((void**)&dev_c, sizeof(int));
```

La función `cudaMalloc`:

- Tiene aire a la función conocida *malloc*.
- Se puede chequear el resultado para saber si anduvo bien.
- Reserva espacio en la memoria global del *device*.
- Recibe dos parámetros:
  1. Una referencia para ubicar el objeto construido.
  2. El tamaño del objeto a ubicar en cantidad de bytes.
- No se pueden usar los punteros de esta función directamente como punteros en *C*, hay que traer y llevar datos.

cudaMalloc

```
cudaMalloc((void**)&dev_c, sizeof(int));
```

La función `cudaMalloc`:

- Tiene aire a la función conocida *malloc*.
- Se puede chequear el resultado para saber si anduvo bien.
- Reserva espacio en la memoria global del *device*.
- Recibe dos parámetros:
  1. Una referencia para ubicar el objeto construido.
  2. El tamaño del objeto a ubicar en cantidad de bytes.
- No se pueden usar los punteros de esta función directamente como punteros en *C*, hay que traer y llevar datos.
- `cudaFree()` libera los recursos, recibe el puntero que entregó *cudaMalloc*.

cudaMemcpy

```
cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
```

cudaMemcpy

```
cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
```

La función `cudaMemcpy`:

- Es *la* función para realizar la transferencia de memoria.
- Desde el *host* se puede acceder de memoria global y constante del dispositivo.

cudaMemcpy

```
cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
```

La función `cudaMemcpy`:

- Es *la* función para realizar la transferencia de memoria.
- Desde el *host* se puede acceder de memoria global y constante del dispositivo.
- Recibe cuatro parámetros:
  1. Puntero al destino.
  2. Puntero al origen.
  3. Cantidad de bytes a transferir.

cudaMemcpy

```
cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
```

La función `cudaMemcpy`:

- Es *la* función para realizar la transferencia de memoria.
- Desde el *host* se puede acceder de memoria global y constante del dispositivo.
- Recibe cuatro parámetros:
  1. Puntero al destino.
  2. Puntero al origen.
  3. Cantidad de bytes a transferir.
  4. Tipo de transferencia:
     - Host a Host: es el standard, no debería hacer falta CUDA para esto.
     - Host to Device: `cudaMemcpyHostToDevice`.
     - Device a Host: `cudaMemcpyDeviceToHost`.
     - Device to Device: `cudaMemcpyDeviceToDevice`.

cudaMemcpy

```
cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
```

La función `cudaMemcpy`:

- Es *la* función para realizar la transferencia de memoria.
- Desde el *host* se puede acceder de memoria global y constante del dispositivo.
- Recibe cuatro parámetros:
  1. Puntero al destino.
  2. Puntero al origen.
  3. Cantidad de bytes a transferir.
  4. Tipo de transferencia:
     - Host a Host: es el standard, no debería hacer falta CUDA para esto.
     - Host to Device: `cudaMemcpyHostToDevice`.
     - Device a Host: `cudaMemcpyDeviceToHost`.
     - Device to Device: `cudaMemcpyDeviceToDevice`.
- La transferencia es asincrónica.

Consultando al dispositivo

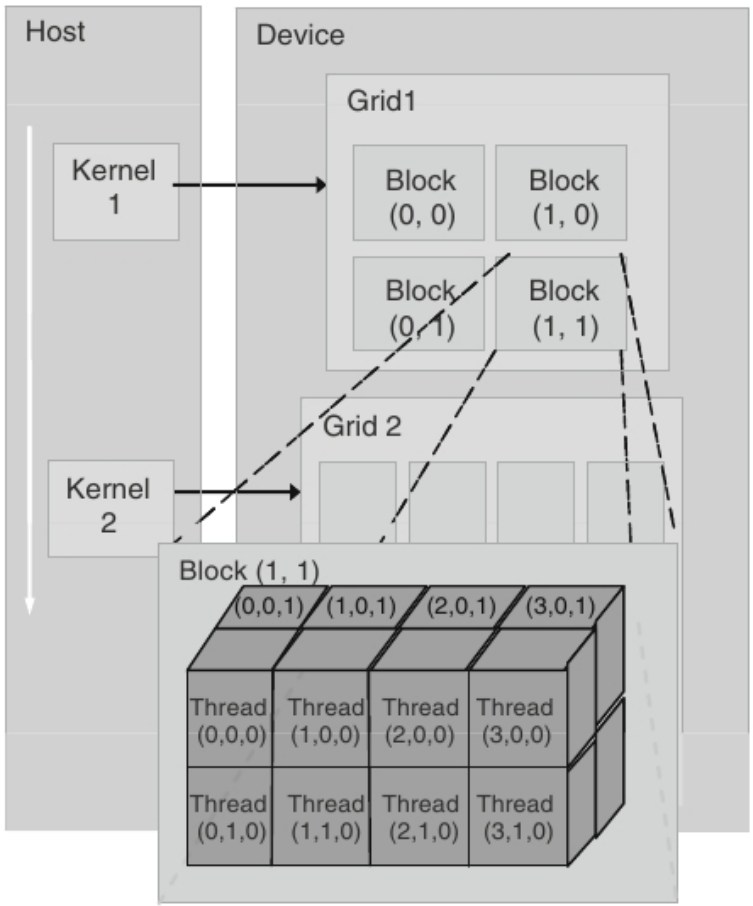
Si se quieren obtener las propiedades del dispositivo desde programa, se puede usar la función `cudaGetDeviceProperties` que permite obtener mucha información en runtime acerca del dispositivo:

```
int main( void ) {
cudaDeviceProp prop;

cudaGetDeviceProperties( &prop, i );
return 0;

fprintf( "Name: %s\n", prop.name );
printf( "Compute capability: %d.%d\n", prop.major, prop.minor );
}
```

Trae mucha más información, googlear para averiguar más.

Paralelismo

```
add<<<1,1>>>( 2, 7, dev_c );
```



- La clave está en los números que acompañan al *kernel*.
- Representan las dimensiones del bloque de threads que lo van a ejecutar.
- Cada thread tiene acceso a las variables `threadIdx.x` y `threadIdx.y`, que representan unívocamente al hilo de sus *hermanos*.
- Cada thread dentro de un bloque puede cooperar con otros dentro del mismo bloque (sincroniza la ejecución, comparten datos eficientemente a través de la *shared memory*).
- Dos threads en bloques distinos NO pueden cooperar.

Dimensionando la ejecución

```
dim3 dimBlock(Width, Width);
dim3 dimGrid(1,1);
MatrixMulKernel<<<dimGrid,dimBlock>>>(Md,Nd,Pd,Width);
```

- Los números que acompañan la invocación del *kernel* se conocen como *execution configuration parameters*.
- No hace falta crear variables para la llamada, se pueden hardcodear directamente.

Importante:

1. Get the data on the GPGPU and keep it there.
2. Give the GPGPU enough work to do.
3. Focus on data reuse within the GPGPU to avoid memory bandwidth limitations.

Ejercicios iniciales:

Realizar los siguientes ejercicios, en todos los casos comprobar que el resultado sea correcto realizando la mismas operaciones en CPU:

1. Obtener el reverso de un arreglo
2. Calcular la inversa de una matriz .
3. Obtener el máximo de un arreglo.
4. Obtener el máximo de una matriz.

Bibliografía:

- *CUDA by Example: An introduction to General-Purpose GPU Programming*, Jason Sanders, Edward Kandrot, Nvidia, 2010.
- *Programming Massivelly Parallel Processors: A Hands-on approach*, David Kirk, Wen-mei W. Hwu, Morgan Kaufmann, 2010.
- *CUDA Application Design And Development*, Ron Farber, Morgan Kaufmann, 2011.
- Online tutorial by Sean Baxter `http://www.moderngpu.com/intro/intro.html` (año 2011).