Seminario II: Introducción a la Computación GPU

CONTENIDO

Introducción

Evolución CPUs-Evolución GPUs

Evolución sistemas HPC

Tecnologías GPGPU

Problemática: Programación paralela en clústers heterogéneos

Problemática: Gestión de infraestructura

Conclusiones

REFERENCIAS

Cook, S. CUDA Programming: A Deverlopers Guide to Parallel Computing with GPUs. Morgan Kaufmann-Elsevier, 2013.

La Ley de Moore

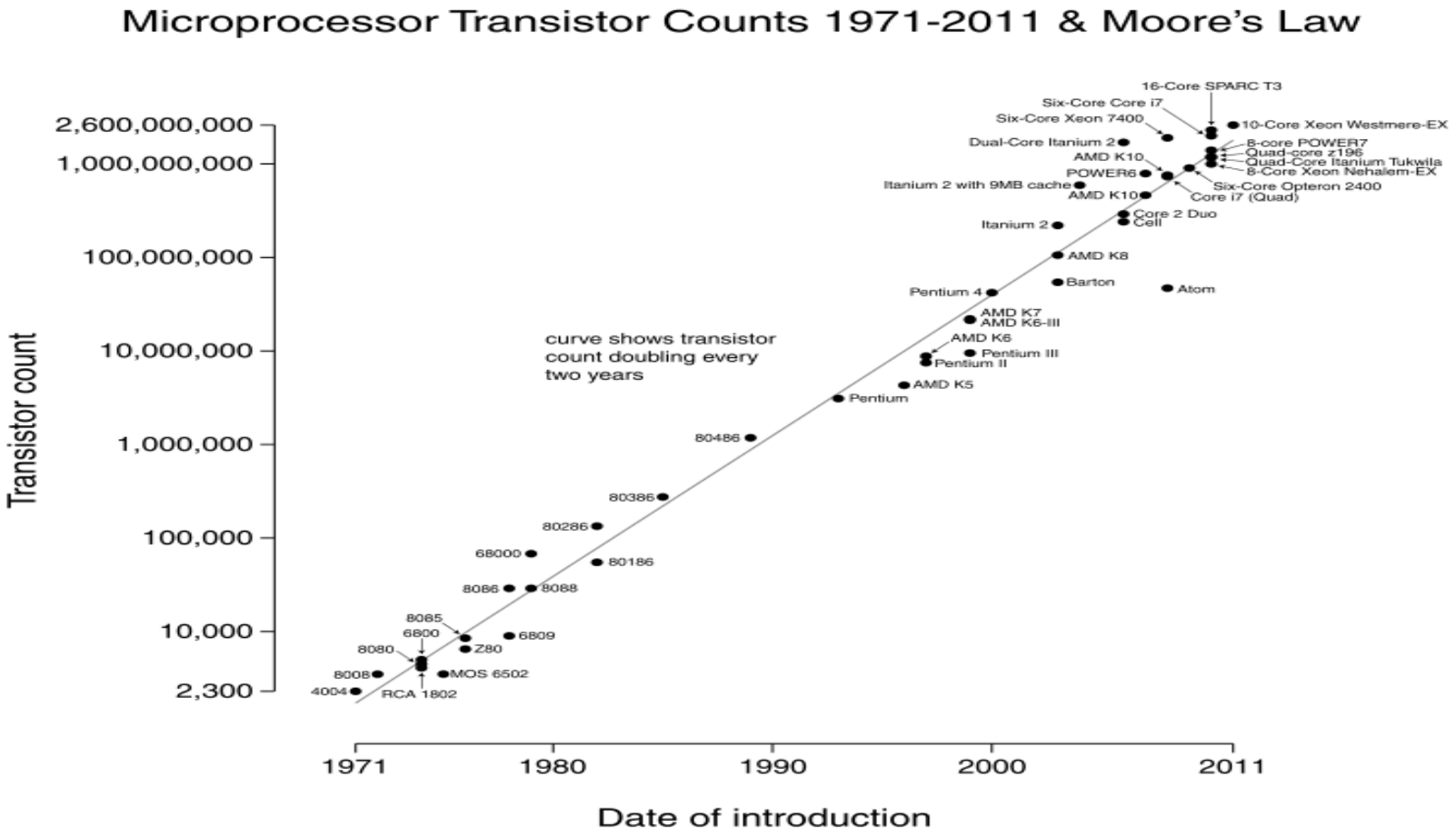


Figura: Crecimiento en Número de Tansistores/Chip

Procesador Mononúcleo: el pasado reciente

- Rendimiento escalaba parejo a Ley de Moore
- Principales motivos:
  - Aumento frecuencia reloj
  - Mejoras/optimizaciones en arquitectura: ILP y nuevas instrucciones: MMX, SSE
  - Más memoria caché
- Mejora *automática* en rendimiento. Todas las aplicaciones se benefician

Procesador Mononúcleo: Fin de una Era

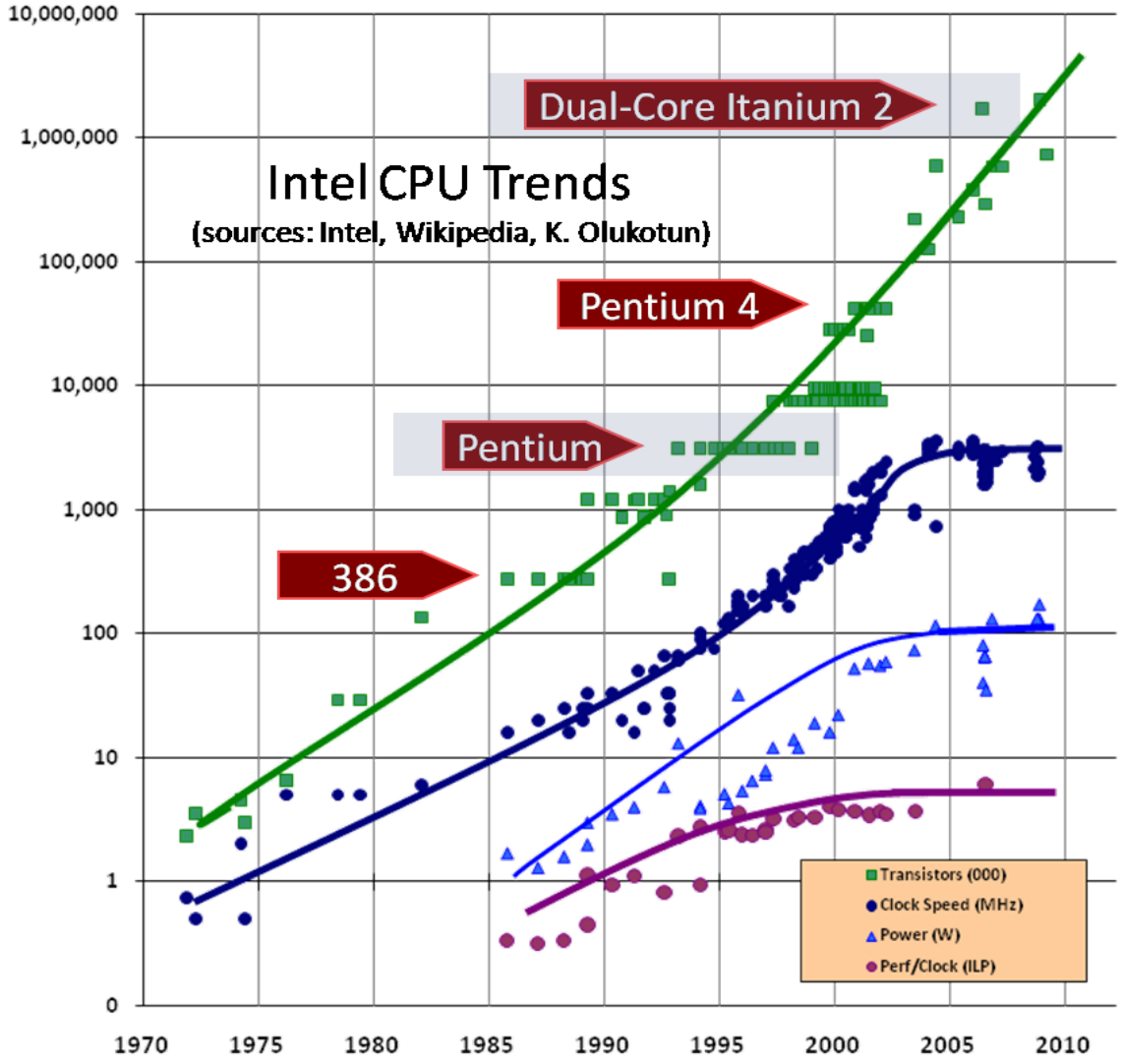
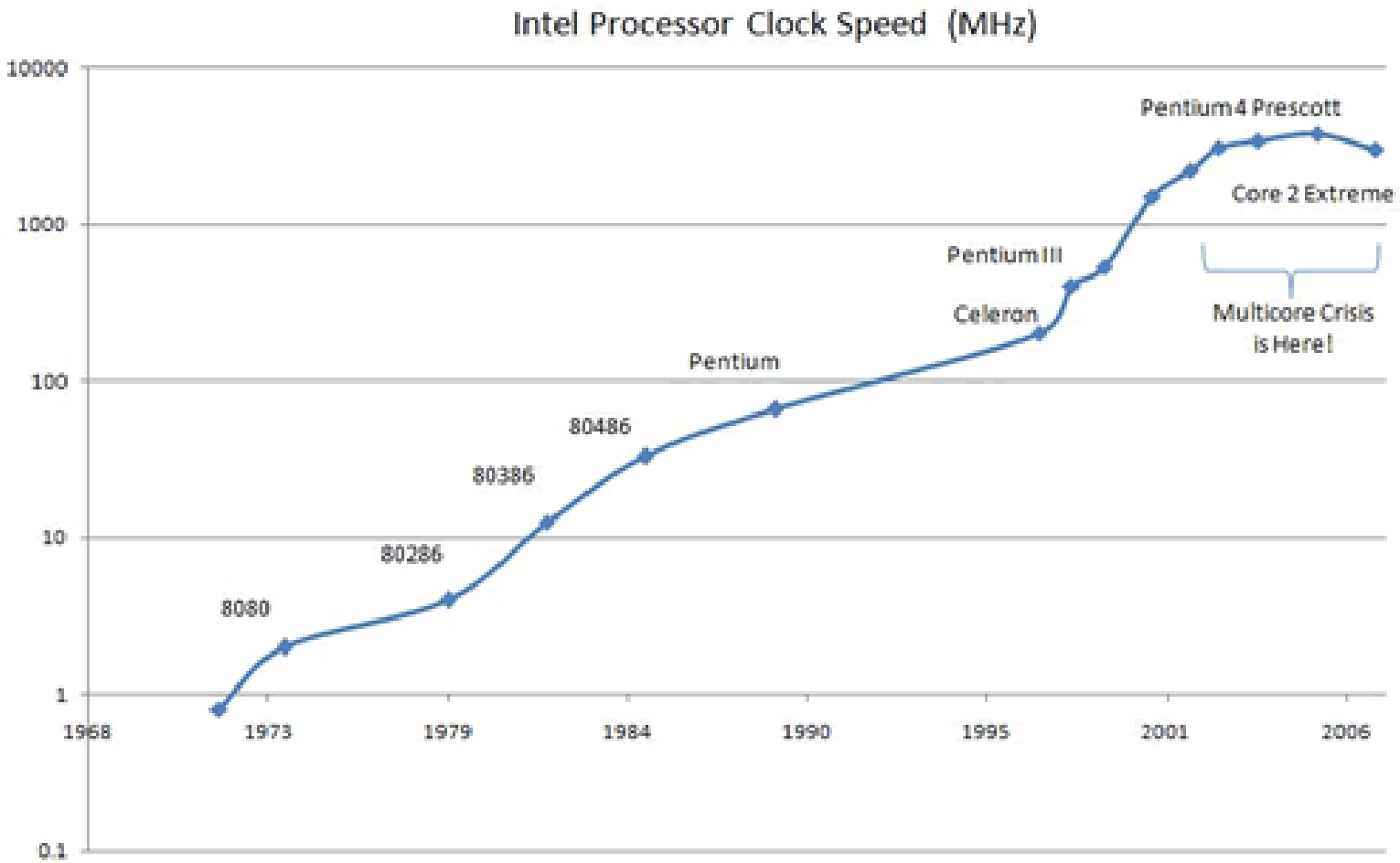


Figura: Rendimiento procesador mononúcleo deja atrás ley de Moore [Herb Sutter]

Procesador Mononúcleo: Multicore Crisis



Procesador Mononúcleo: Multicore Crisis

- ¿Principales razones para el frenazo?
- Dificultades físicas para seguir aumentando frecuencia de reloj en niveles de integración tan elevados:
  - Demasiado calor y difícil de disipar
  - Demasiado consumo de potencia
  - Problemas con corrientes residuales
- Límites en el ILP
  - Procesadores ya muy complejos, optimizaciones arquitectónicas más lentas

Aumento de Rendimiento: Tendencias Actuales

- Mejoras en procesadores:
  - *Simultaneous multithreading* (SMT). Pe. Intel HTT (Hyper-Threading Technology)
  - Multinúcleo (*multicore*)
  - Más cache
- Otras alternativas:
  - Clusters de procesadores homogéneos
  - Clusters de procesadores heterogéneos. Pe. CELL

Aumento de Rendimiento: Tendencias Actuales

- Otras alternativas:
  - Uso de hardware especializado a modo de co-procesador
    - FPGAs (Field-Programmable Gate Array)
    - GPUs (Graphics Processing Unit) ⇒ GPGPU
  - Nuevas arquitecturas innovadoras:
    - ¿Intel Larrabee?
    - AMD Fusion
  - Nuevos retos (punto vista de programador): Aprovechar concurrencia y paralelismo

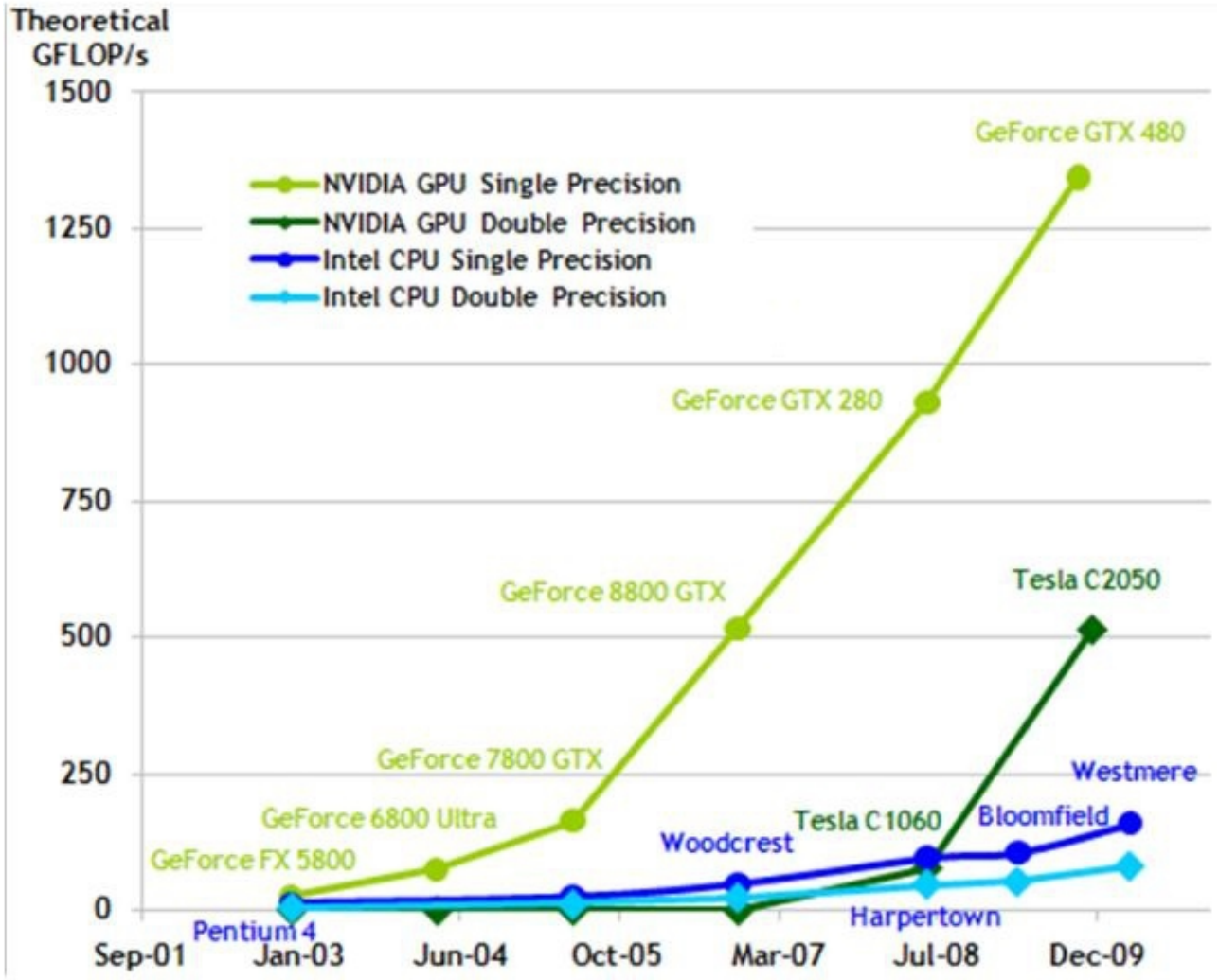
Graphics Processing Unit (GPU)

- Origen: Las tarjetas gráficas de los ordenadores de los años 80
- Propósito: Acelerar por hardware el dibujado de primitivas gráficas en una pantalla, descargando al procesador de estas tareas

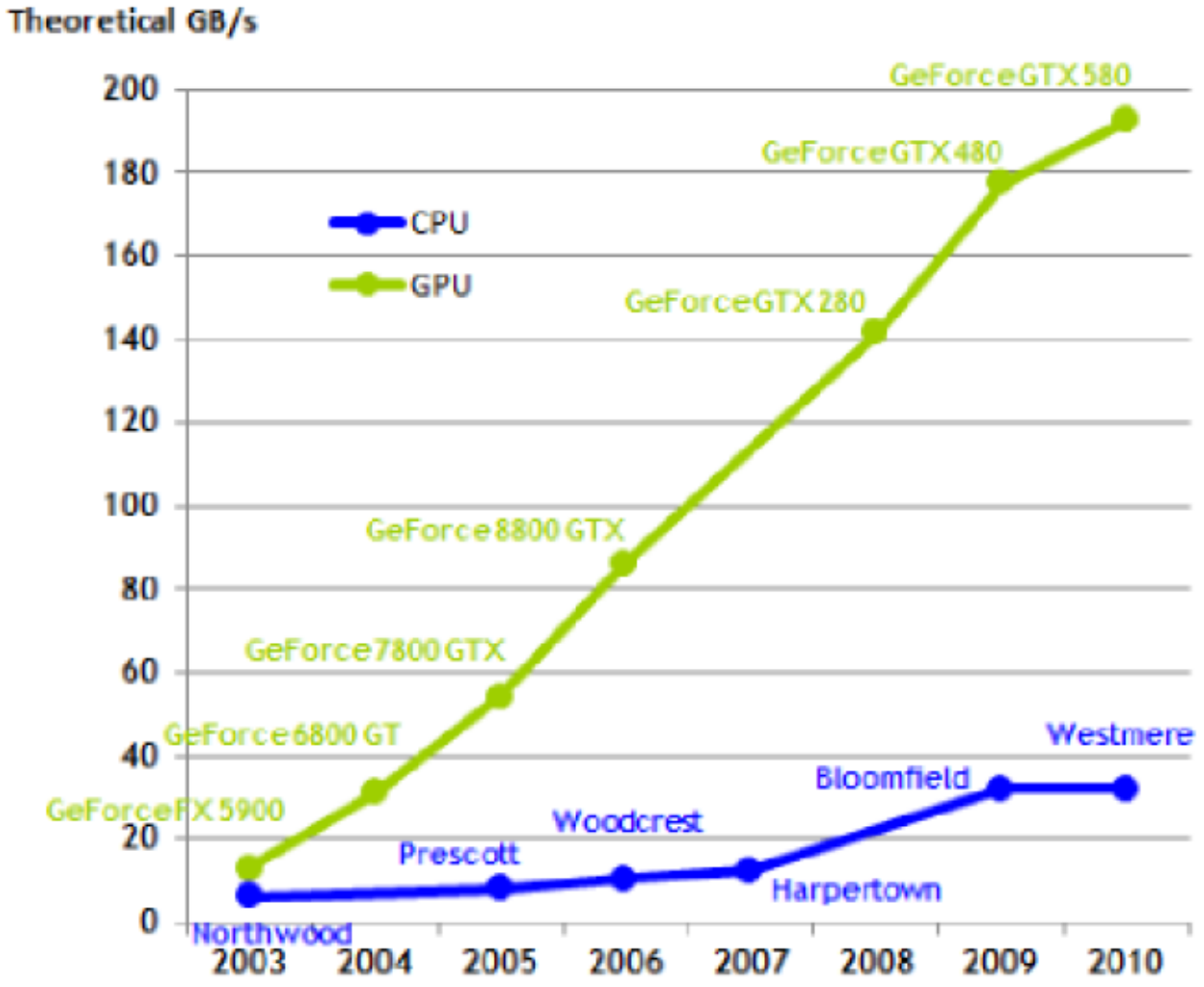
Graphics Processing Unit (GPU)

- Durante los 80s y primeros 90s: aceleración 2D (GUIs)
- En los 90s, las 3D cobran importancia (videojuegos)
  - Aparecen APIs OpenGL y DirectX e idea de *pipeline* gráfico
  - Primeras tarjetas aceleradoras 3D (no programables)
- Cada vez más fases del *pipeline* gráfico en tarj. Gráfica
  - Destacan dos fabricantes: Nvidia y ATI/AMD
  - Se acuña el término de GPU (GeForce 256)
  - Tarjetas gráficas programables: vertex & pixel shaders

CPUs vs. GPUs
Operaciones en punto flotante por segundo



CPUs vs. GPUs
Ancho de banda en transferencias de memoria



CPUs vs. GPUs: Diferencias Arquitectónicas

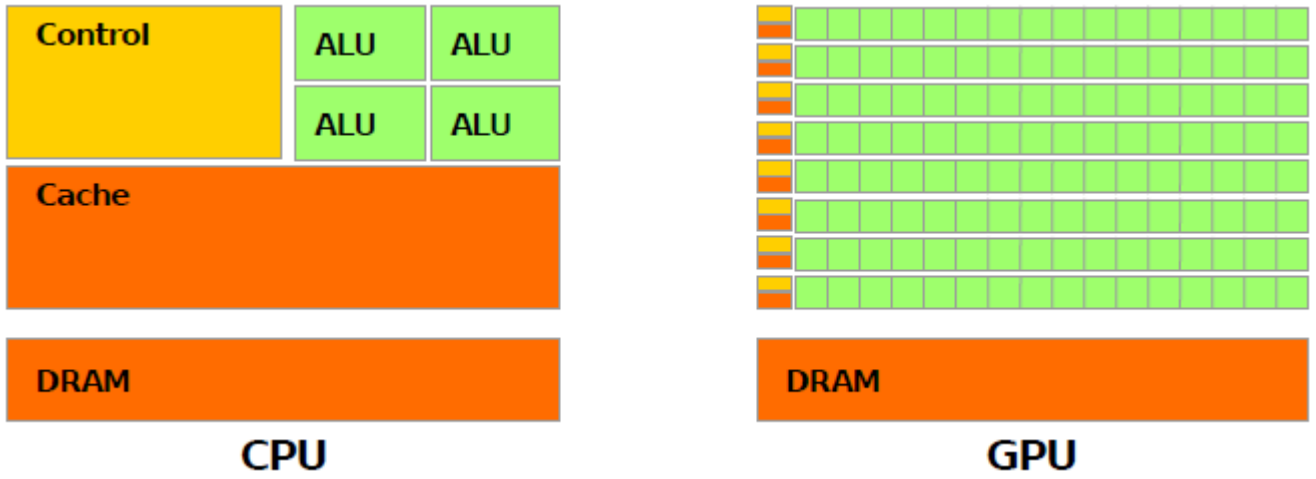


Figura: Diferencias en cantidad de transistores dedicados a procesamiento de datos

CPU Énfasis en
- Más caché
- Control de flujo

GPU Especializada en
- Computación intensiva
- Ejecutar mismas operaciones en paralelo sobre diferentes datos
- Interesa mucho cálculo y poco acceso a memoria

Actualmente: GPU de propósito general

- Uso de GPUs se ha generalizado:
  - Hasta hace poco: Cálculo información gráfica (render): videojuegos, 3D...
  - Ahora, además: Cálculo de propósito general
  - GPUs actuales son dispositivos masivamente paralelos: miles de hilos concurrentes
- GPGPU: General-purpose computing on graphics processing units
- Uso de GPUs como coprocesadores. Explotamos paralelismo:
  - de datos en GPUs
  - de tareas en CPUs

GPGPU: ¿Dónde aplicarla?

- Importante desarrollo y popularización en entornos HPC últimos 3 años
  - sobre todo tecnología CUDA de Nvidia
- Comienza también a cobrar importancia fuera de HPC
- Aplicaciones típicas: las más *data parallel*
  - Informática biomédica: análisis de imágenes. . .
  - Codificación/decodificación de vídeo
  - Simulaciones físicas: fluídos, astrofísica. . .

GPGPU: Límitaciones

- ¿Inconvenientes?
  - Solo buen aprovechamiento paralelismo de datos
  - Cuello de botella: transferencias CPU↔GPU
  - Difíciles de programar para lograr eficiencia elevada
  - Necesarios (de momento) *drivers* cerrados de fabricantes

GPGPU: Modelo de Programación

- Stream processing
  - Paradigma clásico años 80
  - Relacionado con concepto SIMD (Single Instruction, Multiple Data)
- Idea
  - Conjunto de datos: *stream*
  - Operaciones a aplicar: *kernel*
  - Se aplica kernel a cada elemento en stream
  - Concurrencia: explotación paralelismo de datos

GPGPU: Modelo de Trabajo

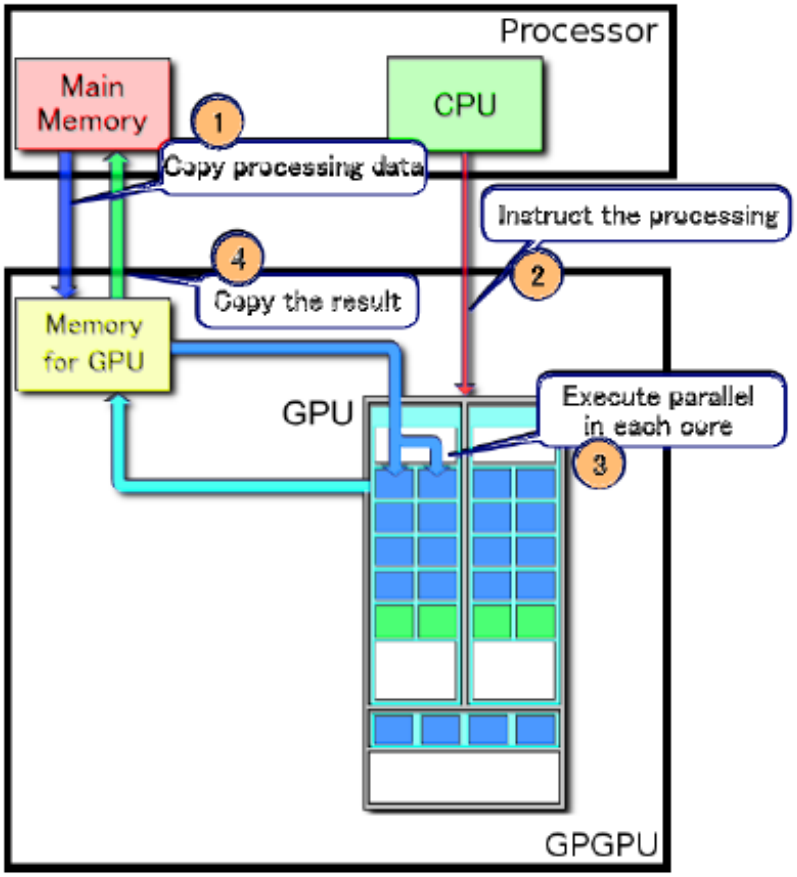


Figura: *Stream processing* se ajusta a modelo GPGPU

Evolución sistemas HPC
Arquitecturas clásicas HPC

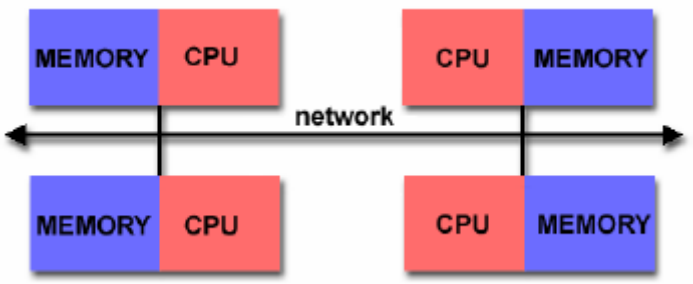


Figura: Sistema de memoria distribuída

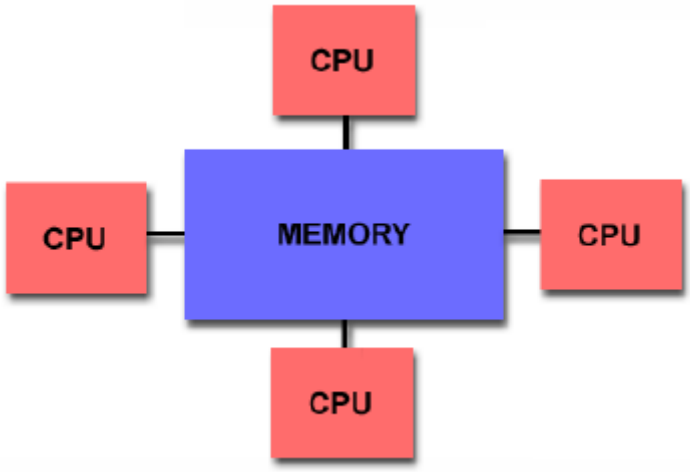


Figura: Sistema de memoria compartida

Sistemas Multiprocesador-Multinúcleo (Última Década)
Clusters homogéneos (Beowulf cluster)

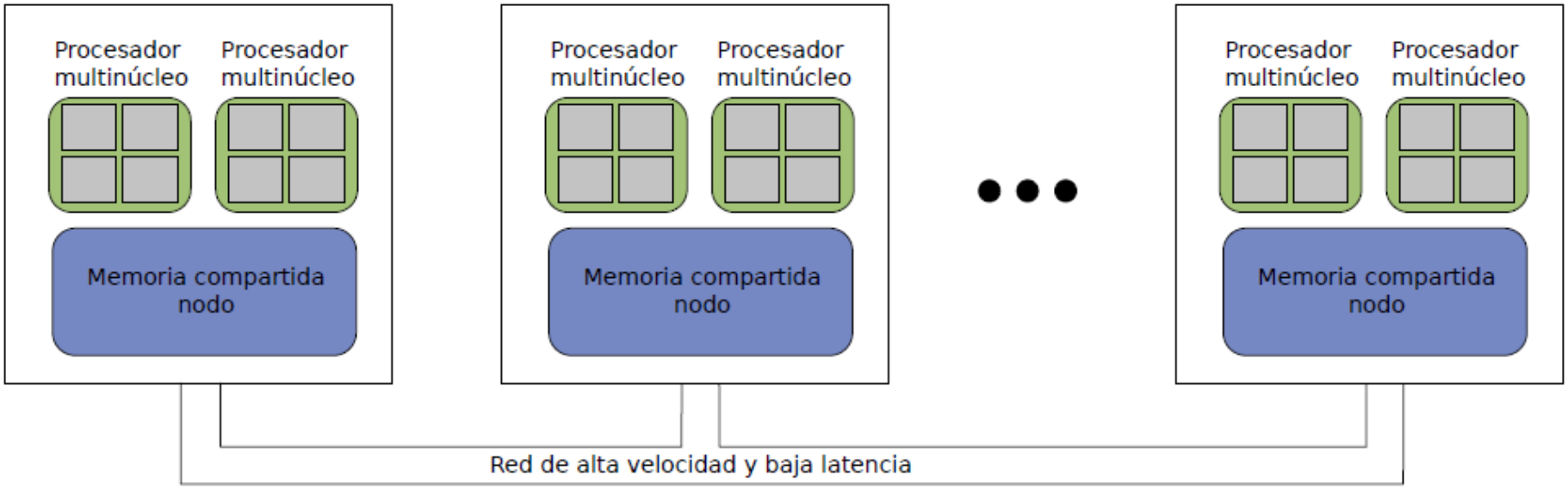


Figura: Cluster homogéneo con nodos multiprocesador y procesadores multinúcleo

Sistemas Heterogéneos CPUs-GPUs para HPC
Escenario Habitual en la Actualidad

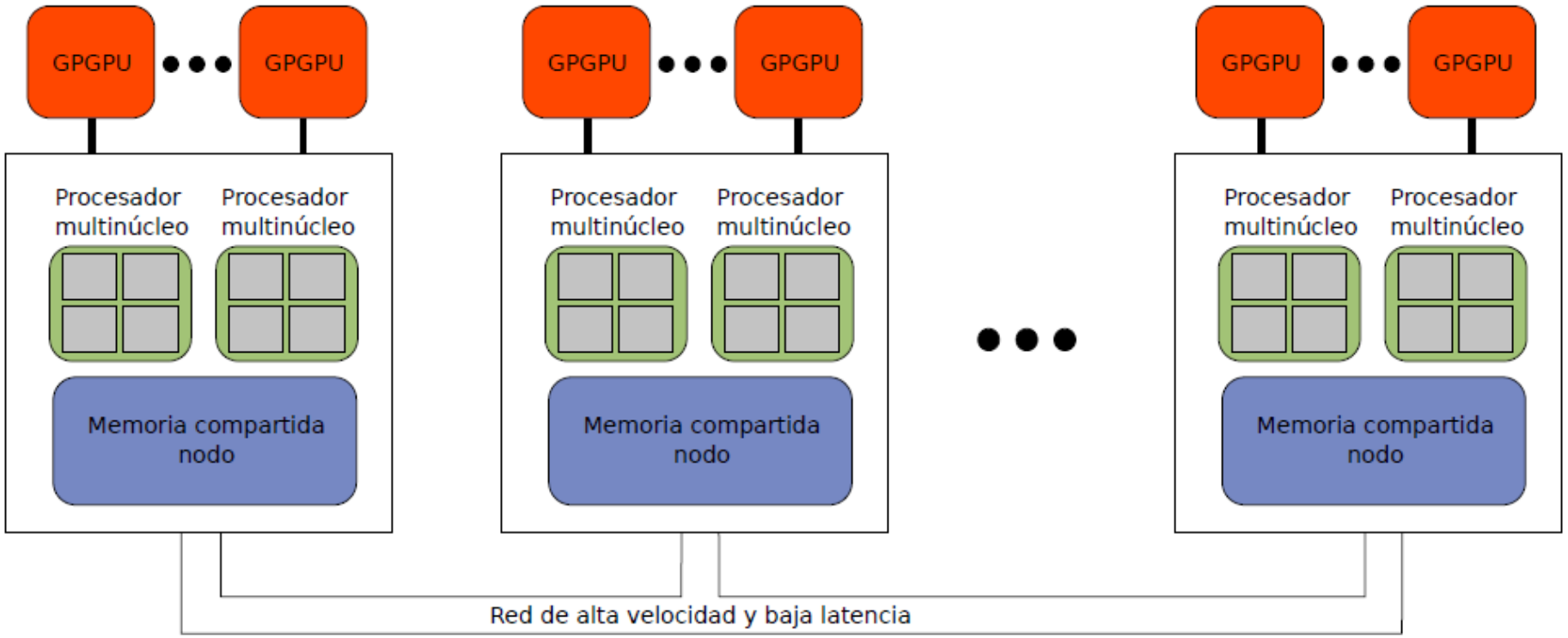


Figura: Cluster híbrido con GPUs y procesadores multinúcleo

Tecnologías GPGPU: Brook, BrookGPU y Brook+

- Brook: Lenguaje de programación, extensión de ANSI C, que incorpora ideas del paradigma *Stream processing*
- BrookGPU
  - Proyecto GPGPU del grupo de gráficos de Univ. Standford basado en leng. Brook (2004, licencia libre: BSD y GPL)
  - Permite explotar GPUs de Nvidia, ATI o Intel como coprocesadores altamente paralelos (data parallel)
  - Múltiples *backends*: CPU, DirectX, OpenGL...
- Brook+
  - Implementación de AMD/ATI basada en BrookGPU
  - Competidor de CUDA en sus inicios, nunca alcanzó el grado de madurez de la tecnología de Nvidia

Tecnologías GPGPU: Brook+

- Mantuvo licencia libre de BrookGPU, buscando ser un estándar abierto
- Incluido en la Stream SDK de AMD, no fue la primera tecnología GPGPU de ATI: Close to Metal (CTM) y AMD FireStream
- Multiplataforma: múltiples backends
  - Predeterminado: CAL, capa de abstracción GPU AMD/ATI
- Buen desarrollo inicial, pero no llegó a ofrecer el 'acabado' y madurez de CUDA
  - Más díficil de programar (todavía)
  - Menos flexibilidad
  - Menos documentación y herramientas de desarrollo
- Última versión: 1.4.1 (2009). Abandonado en favor de OpenCL

CUDA: Compute Unified Device Architecture

- Solución GPGPU de Nvidia
  - Anunciada a finales 2006. 1a versión en 2007
  - Actualmente CUDA 4.0 (junio 2011)
  - Trending topic en HPC últimos 3 años
- *Framework* completo, madurando cada nueva versión:
  - Driver tarjeta gráfica (cerrado: freeware)
  - CUDA Toolkit (cerrado): herramientas para programar tarjetas
  - CUDA SDK (no tan cerrada, pero no libre)
  - Bibliotecas de cálculo (cerradas)
  - Mucha y buena documentación

CUDA: Compute Unified Device Architecture

- Driver tarjeta gráfica (cerrado: freeware)
  - expone arquitectura GPGPU de la GPU
  - pensada para ser explotada con modelo de programación tipo Stream processing
- CUDA Toolkit (cerrado): herramientas para programar tarjetas
  - C for CUDA: C con algunas extensiones
  - runtime library: API gestión GPU desde CPU
  - CUDA driver API: API debajo de runtime. También accesible desde capa aplicación. Ofrece más flexibilidad y control que runtime
  - Herramientas depuración y profiling

CUDA: Compute Unified Device Architecture

- CUDA SDK (no tan cerrada, pero no libre)
  - Ejemplos y pruebas de concepto
- Bibliotecas de cálculo (cerradas)
  - Construidas por encima de runtime library
  - cuBLAS, cuFFT...
- Mucha y buena documentación
  - Guías de programación, de referencia, de buenas prácticas...

CUDA
Compute Unified Device Architecture

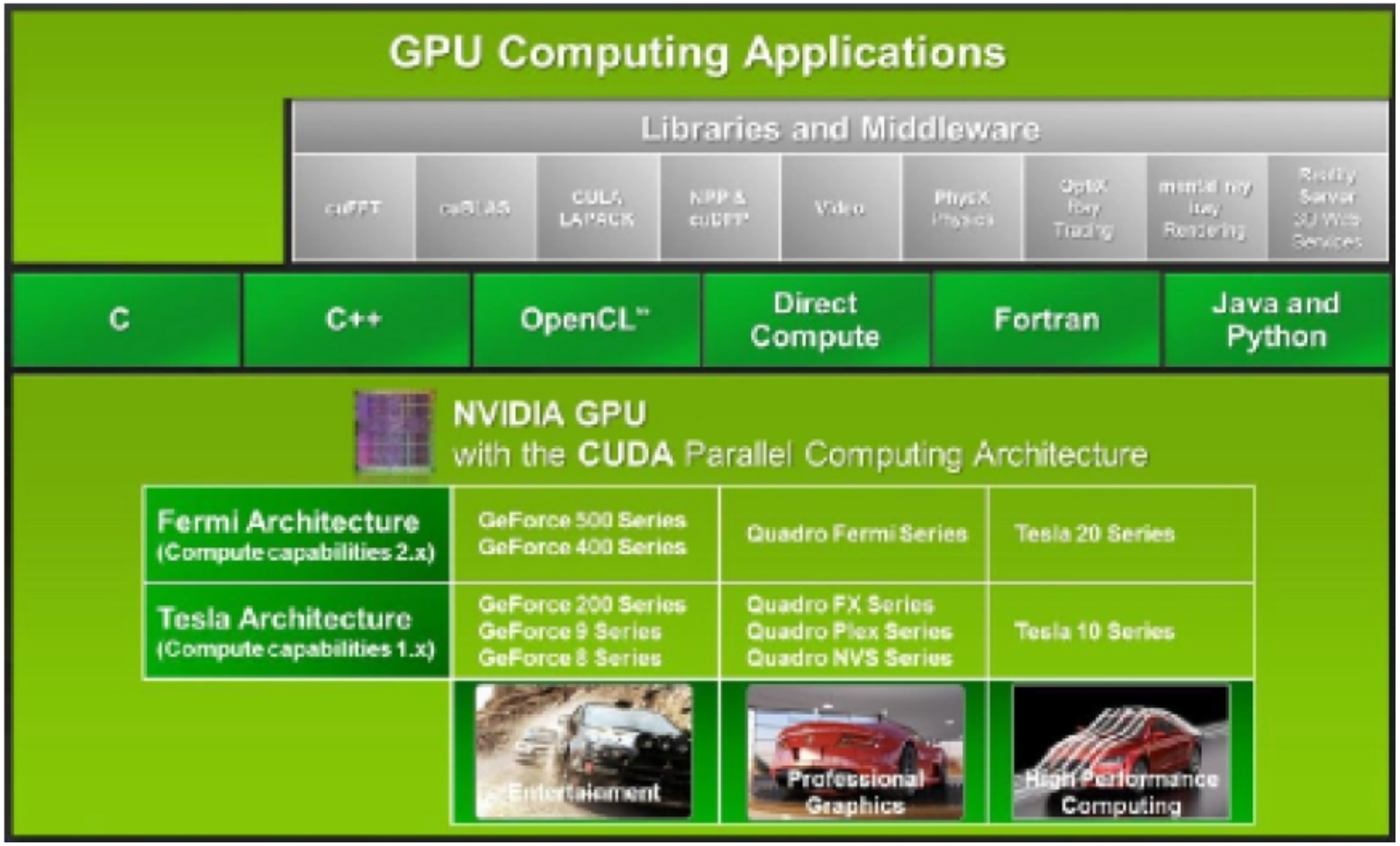


Figura: Arquitectura CUDA

CUDA
Idea clave 1: Muchos hilos de ejecución concurrentes

- Creación y gestión de hilos muy ‘barata’
- Hilos se organizan en bloques (*block*). Dentro de un bloque:
  - Ejecución tipo SIMD: *SIMT*
  - Posibilidad de sincronizar hilos
  - Acceso a una memoria compartida de alta velocidad
  - Hilos de un bloque pueden cooperar entre sí
- Bloques independientes entre sí en cuanto a ejecución
  - Pueden ejecutarse en paralelo y en cualquier orden
  - Garantiza escalabilidad
- Bloques se organizan en una rejilla (*Grid*)
  - Rejilla: total de hilos que ejecutan un kernel en GPU

CUDA
Escalabilidad automática (monoGPU)

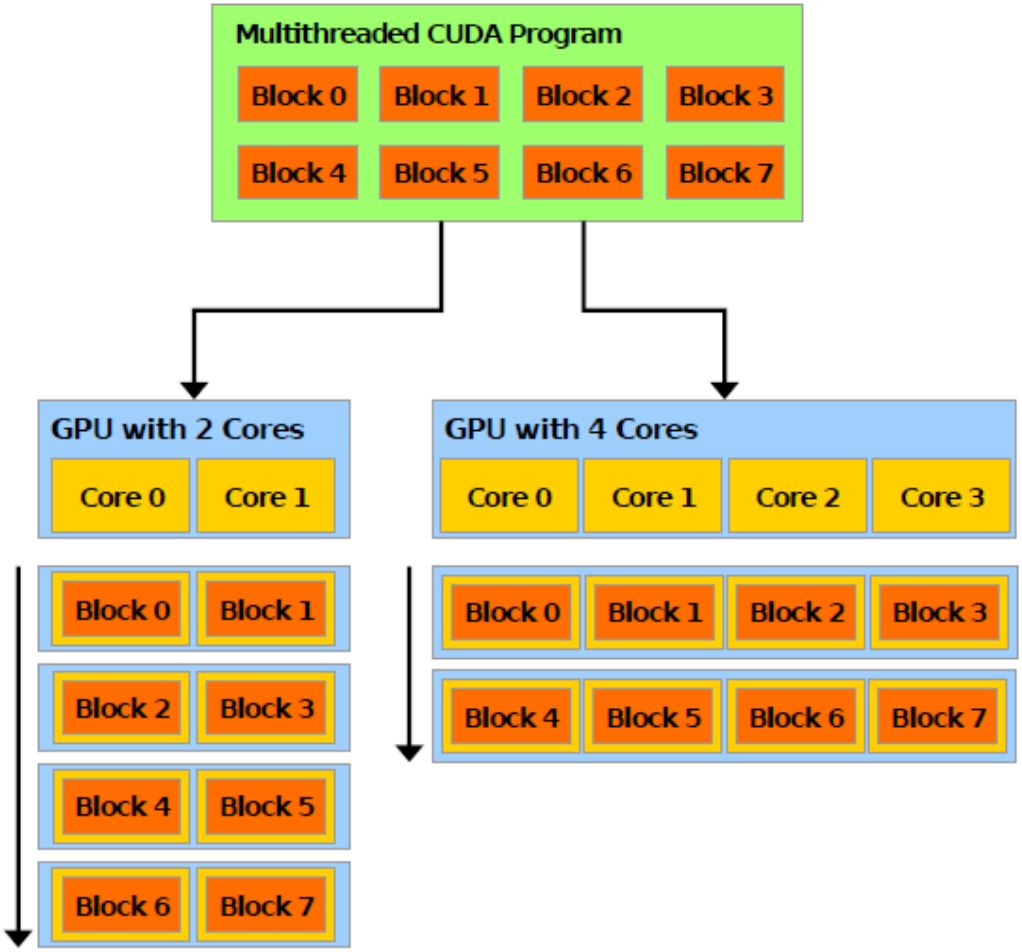


Figura: Ejecución en bloques independientes permite escalabilidad

CUDA
Jerarquía de hilos: bloques y rejilla

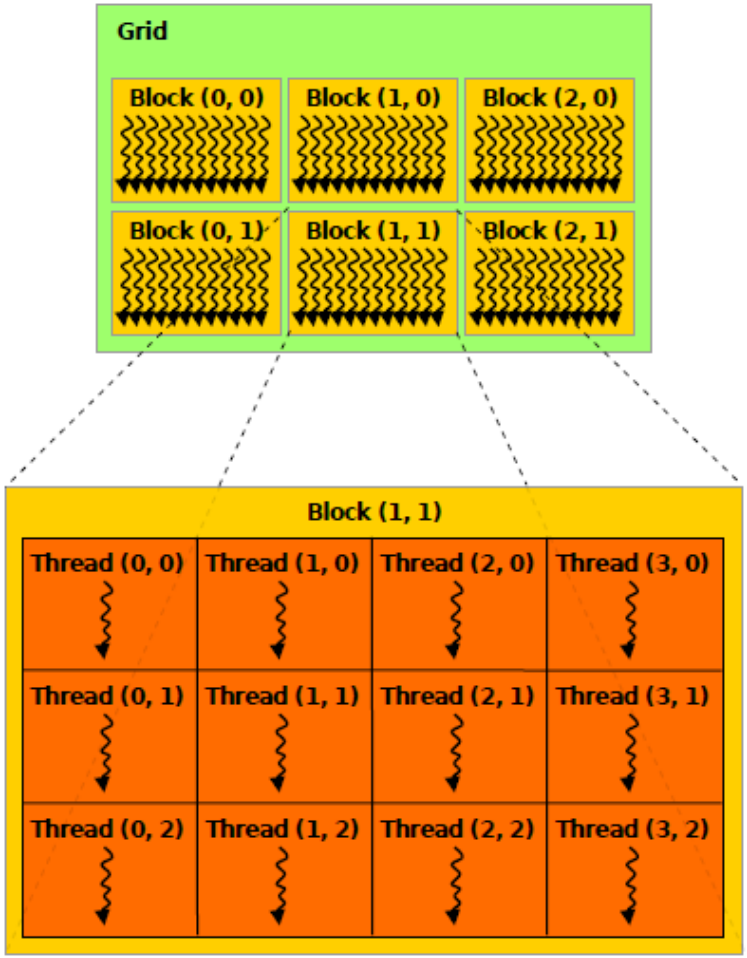


Figura: Rejilla de bloques con hilos que ejecutarán kernel en GPU

CUDA
Idea clave 2: Jerarquía de memoria

- Varios tipos de memoria disponibles para los hilos
  - Registros
  - Memoria local (*local memory*)
  - Memoria compartida (*shared memory*)
  - Memoria global (*global memory*)
- Además, dos tipos de memoria 'de solo lectura'
  - Memoria de constantes (*constant memory*)
  - Memoria de texturas (*texture memory*)
- Uso eficiente de jerarquía de memoria fundamental para buen rendimiento ejecución del kernel

CUDA: Jerarquía de Memoria

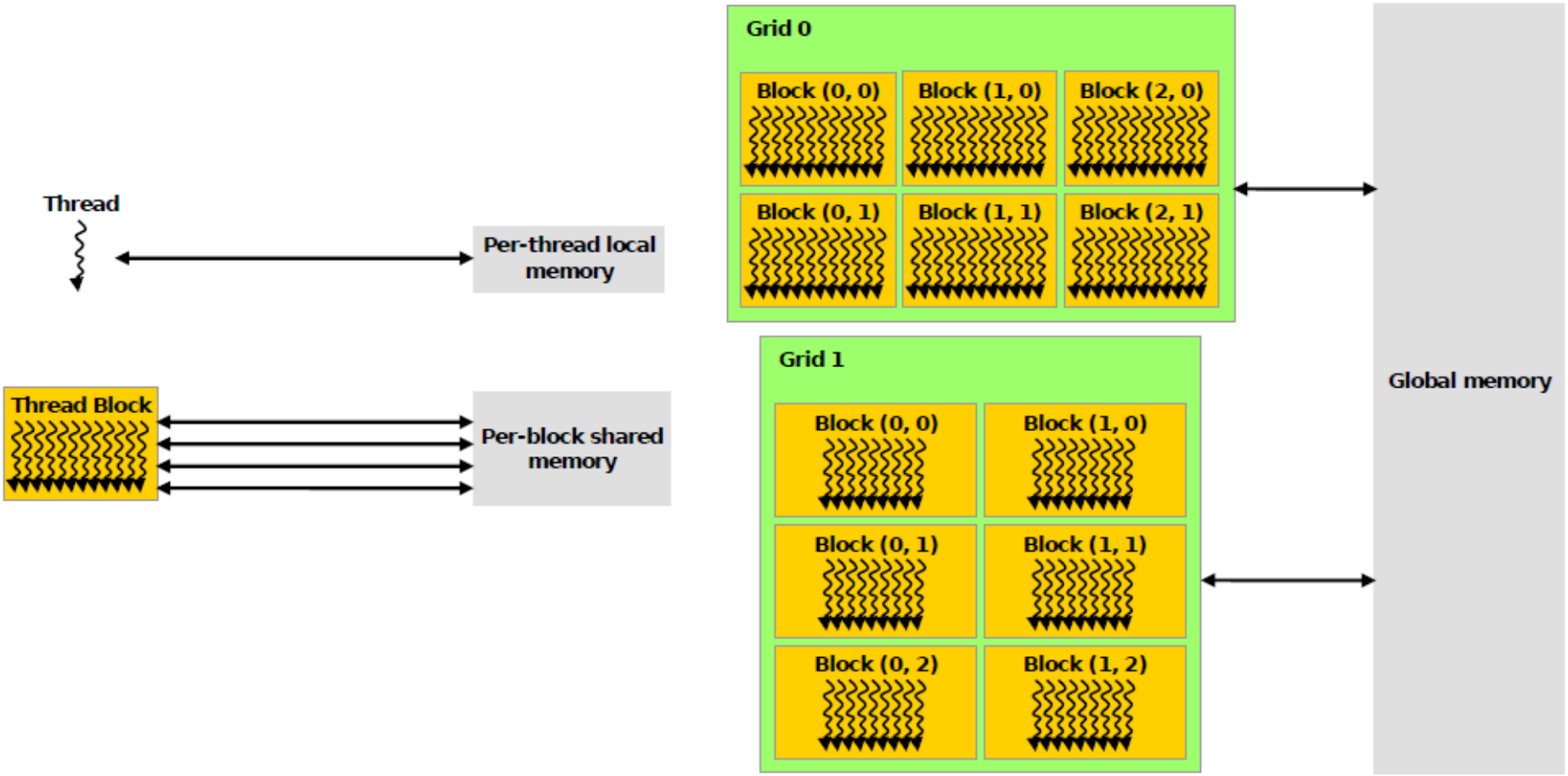


Figura: Principales espacios de memoria disponibles para hilos GPU

CUDA
Idea clave 3: Arquitectura de la GPU que nos expone CUDA

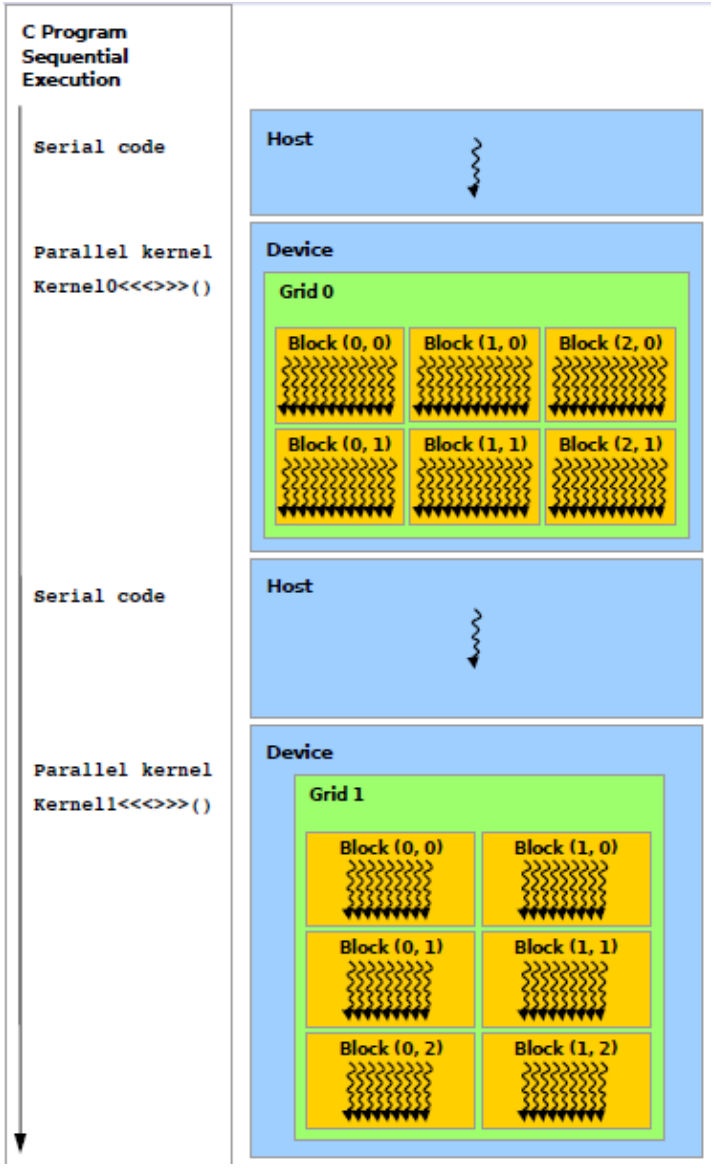
- Una GPU CUDA consta de multiprocesadores (*Stream Multiprocessor* o SM)
- Los bloques de hilos se asignan a los SM para su ejecución
- Los hilos de los bloques asignados a un SM comparten sus recursos
- Los hilos se ejecutan en los núcleos computacionales

Figura: SM Fermi

CUDA: Modelo de Ejecución de CUDA

- GPU actúa como coprocesador de CPU
- CPU se encarga de
  - partes secuenciales
  - partes *task parallel*: POSIX threads (*pthreads*), OpenMP, Intel TBB, MPI
- GPU(s) ejecuta(n) partes data parallel
  - CPU transfiere datos a GPU
  - CPU lanza kernel en GPU
  - CPU recoge resultados de GPU
- Cuello de botella: transferencias
  - Maximizar cálculo - Minimizar transf.
  - Solapar comunicación y computación

CUDA:Modelo de ejecución de CUDA



CUDA: Programación en CUDA

- Paralelizar una aplicación no es fácil (ni en CPU ni GPU)
  - Detectar regiones paralelizables y paradigmas más adecuados
  - Tener en cuenta el hardware objetivo/disponible
- Programar en CUDA un kernel sencillo y ejecutarlo es relativamente sencillo
- Explotar toda la potencia de una o varias GPUs CUDA en el sistema requiere gran conocimiento de la plataforma CUDA y mucho esfuerzo
- Existen soluciones ‘por encima’ de CUDA, como pe. HMPPWorkbench (cerrado)
  - Funciona con pragmas, al estilo OpenMP
  - Nos abstrae de complejidad de CUDA, indicando regiones para ser ejecutadas en GPU

CUDA: Conclusiones

- CUDA ha madurado mucho desde su primera versión:
  - Mejores herramientas de desarrollo
  - Incorporando novedades en arquitectura GPUs de Nvidia
- Principales inconvenientes:
- Solución totalmente cerrada (aunque gratis parte software)
  - Dependencia tecnológica: nuestro código queda ligado al fabricante
- Dependencia de las distintas generaciones de tarjetas Nvidia
  - Algunas características importantes para eficiencia solo disponibles en tarjetas muy caras
  - Para aprovechar novedades en nuevas versiones puede ser necesario bastante trabajo de migración
- No es una solución integral para sistemas heterogéneos
  - Solo aprovechamiento de GPGPU
- Obtener buen rendimiento requiere un gran esfuerzo de programación