HPC y GPUs

GPGPU y software libre

Emilio J. Padrón González

GRUPO ARQUITECTURA DE COMPUTADORES
UNIVERSIDADE DA CORUÑA
GAC.UDC.ES

Workshop HPC y Software Libre
Ourense, Octubre 2011

Contenidos

La ley de Moore

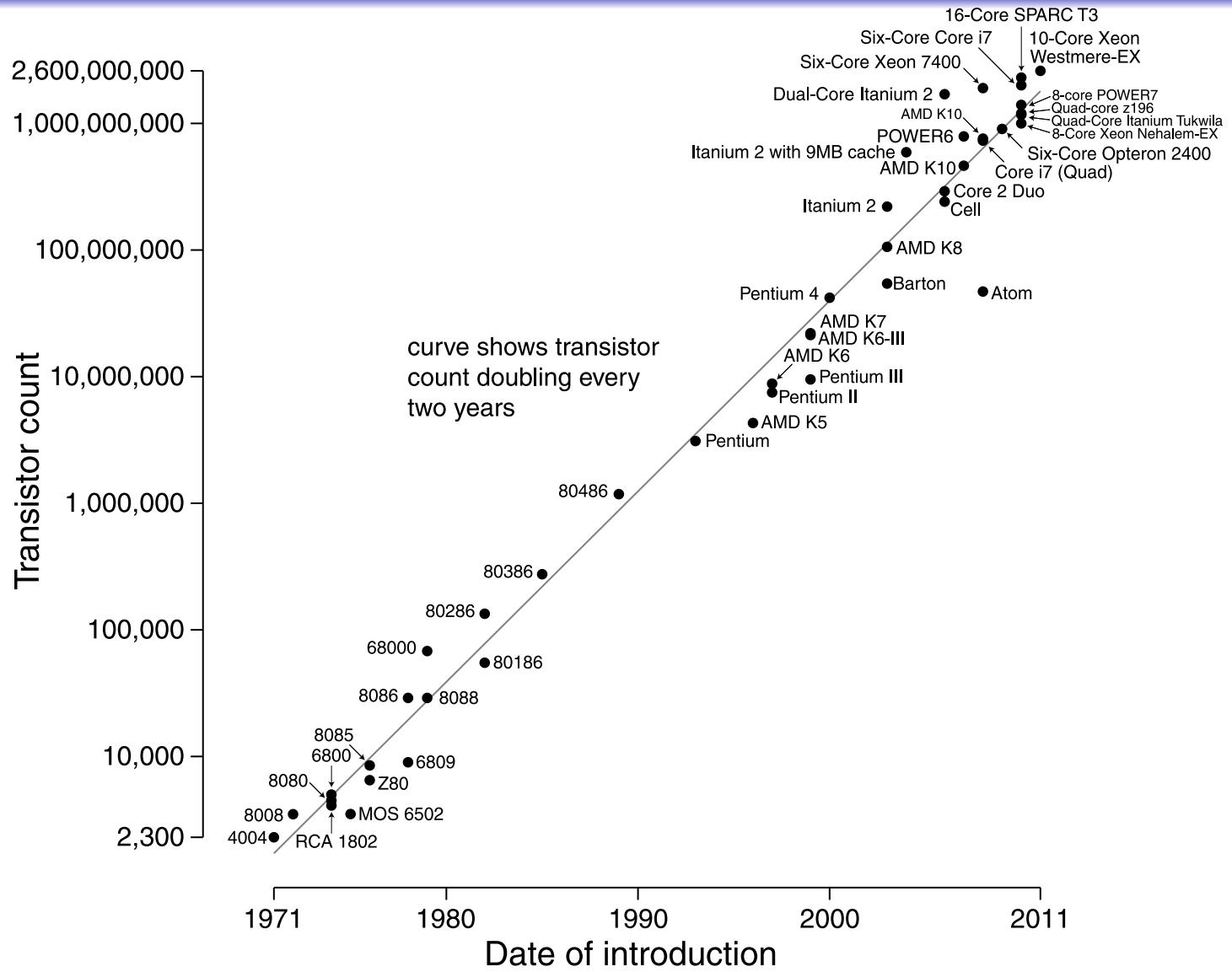


Figura: Crecimiento en número de transistores por chip [Wikipedia]

Procesador mononúcleo
Hasta hace pocos años...

- Rendimiento escalaba parejo a Ley de Moore
- Principales motivos:
  - Aumento frecuencia reloj
  - Mejoras/optimizaciones en arquitectura:
    - **ILP** ▸ Formas de Paralelismo
    - Nuevas instrucciones: MMX, SSE
  - Más memoria cache

Mejora *automática* en rendimiento

Todas las aplicaciones se benefician

Procesador mononúcleo
Tendencia se detiene alrededor de 2003

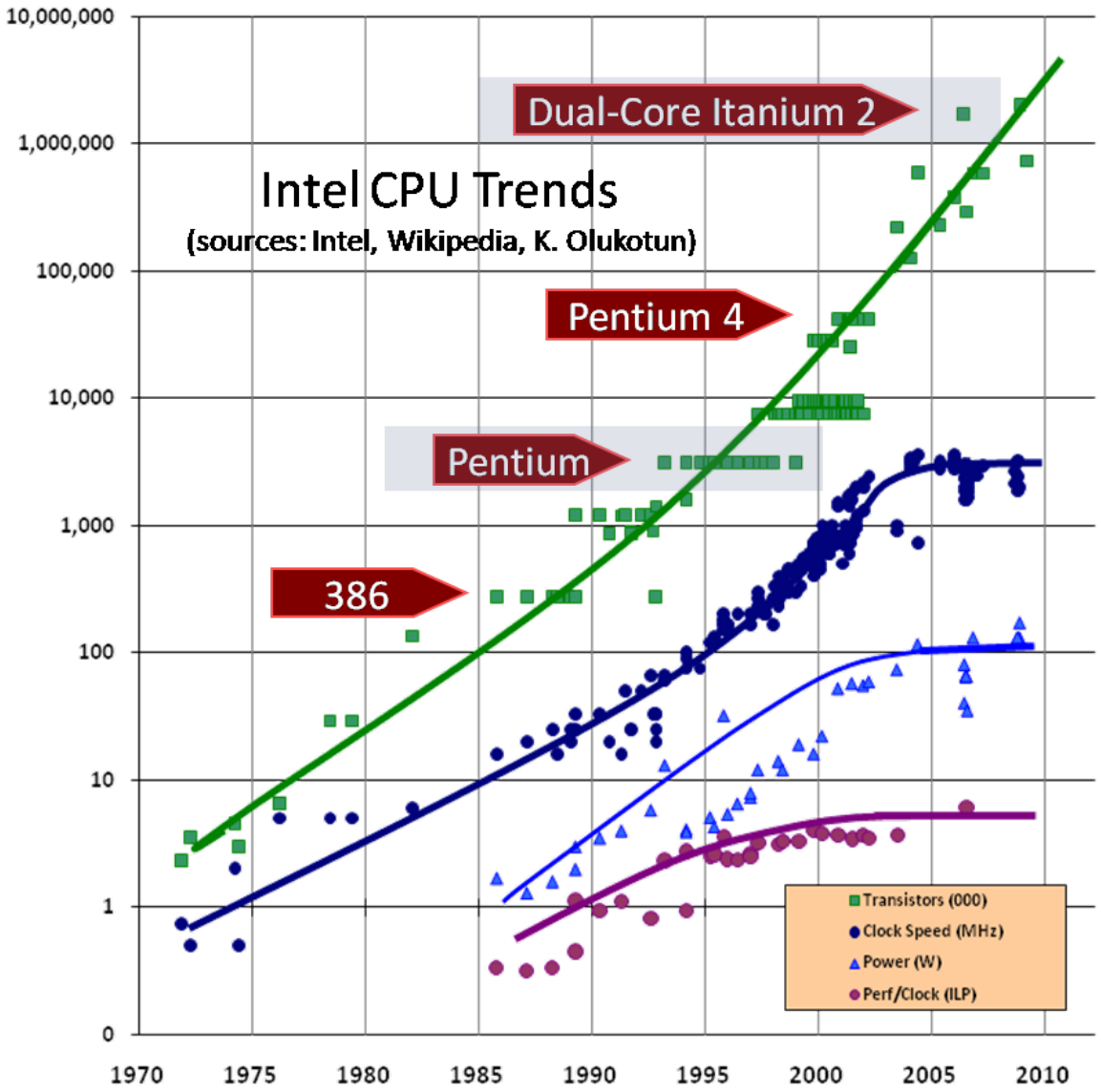
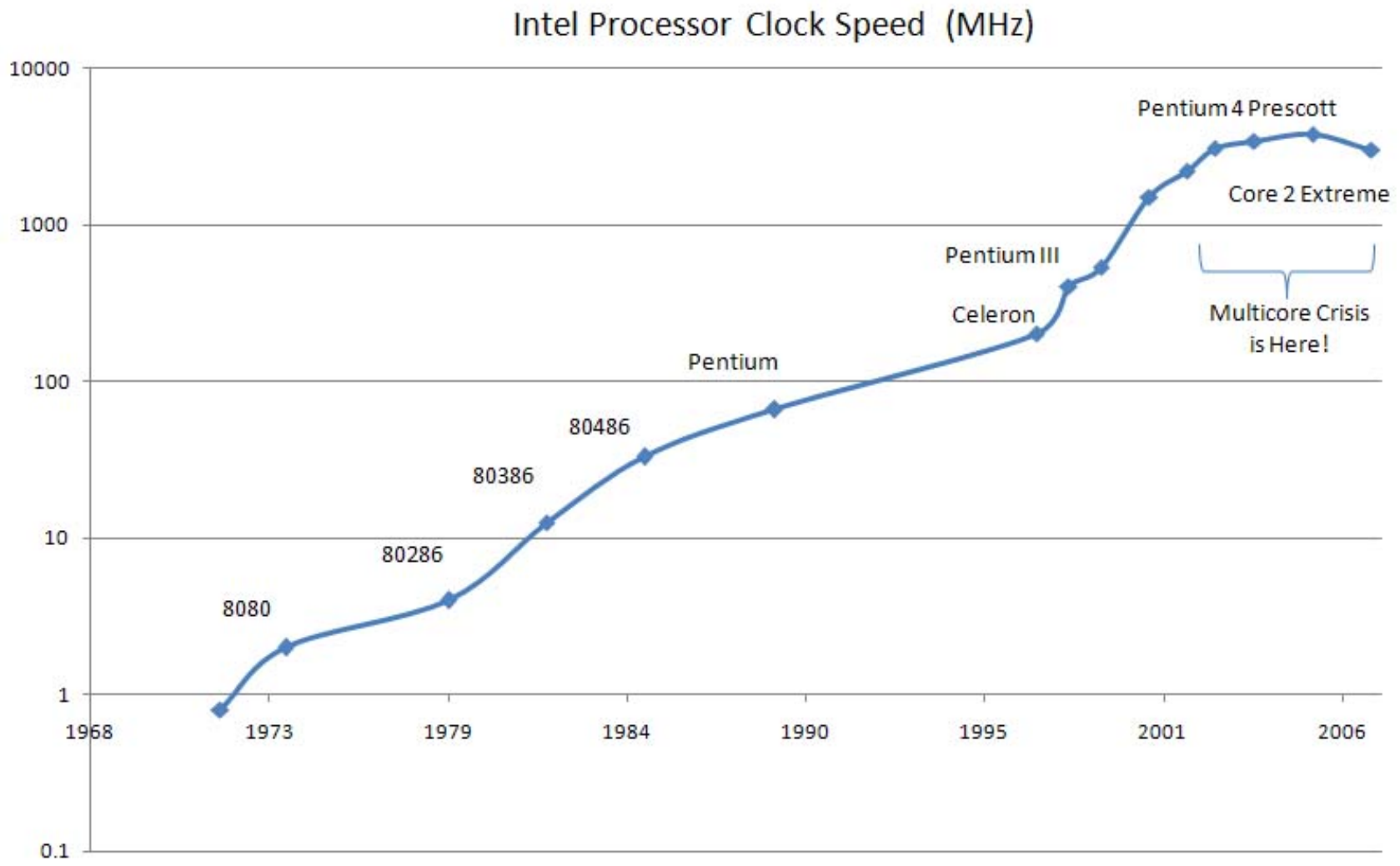


Figura: Rendimiento procesador mononúcleo deja atrás ley de Moore [Herb Sutter]

Procesador mononúcleo

Resultado: la 'crisis del hardware' (*multicore crisis*)



Procesador mononúcleo
La crisis del hardware (*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

Alternativas para aumentar rendimiento
Y así seguir aprovechando Ley de Moore

- Mejoras en procesadores:
  - *Simultaneous multithreading* (SMT). Pe. Intel HTT (Hyper-Threading Technology)
  - Multinúcleo (*multicore*)
  - Más cache

Alternativas para aumentar rendimiento
Y así seguir aprovechando Ley de Moore

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

Alternativas para aumentar rendimiento
Y así seguir aprovechando Ley de Moore

- 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

Alternativas para aumentar rendimiento
Y así seguir aprovechando Ley de Moore

- 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
  - Uso de hardware especializado a modo de co-procesador
    - FPGAs (*Field-Programmable Gate Array*)
    - GPUs (*Graphics Processing Unit*) $\Rightarrow$ **GPGPU**

Alternativas para aumentar rendimiento
Y así seguir aprovechando Ley de Moore

- 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
  - Uso de hardware especializado a modo de co-procesador
    - FPGAs (*Field-Programmable Gate Array*)
    - GPUs (*Graphics Processing Unit*) $\Rightarrow$ **GPGPU**
  - Nuevas arquitecturas innovadoras:
    - ¿Intel Larrabee?
    - AMD Fusion

Alternativas para aumentar rendimiento
Y así seguir aprovechando Ley de Moore

- 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
  - Uso de hardware especializado a modo de co-procesador
    - FPGAs (*Field-Programmable Gate Array*)
    - GPUs (*Graphics Processing Unit*) $\Rightarrow$ **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)

Origen

Las **tarjetas gráficas** de los ordenadores de los años 80

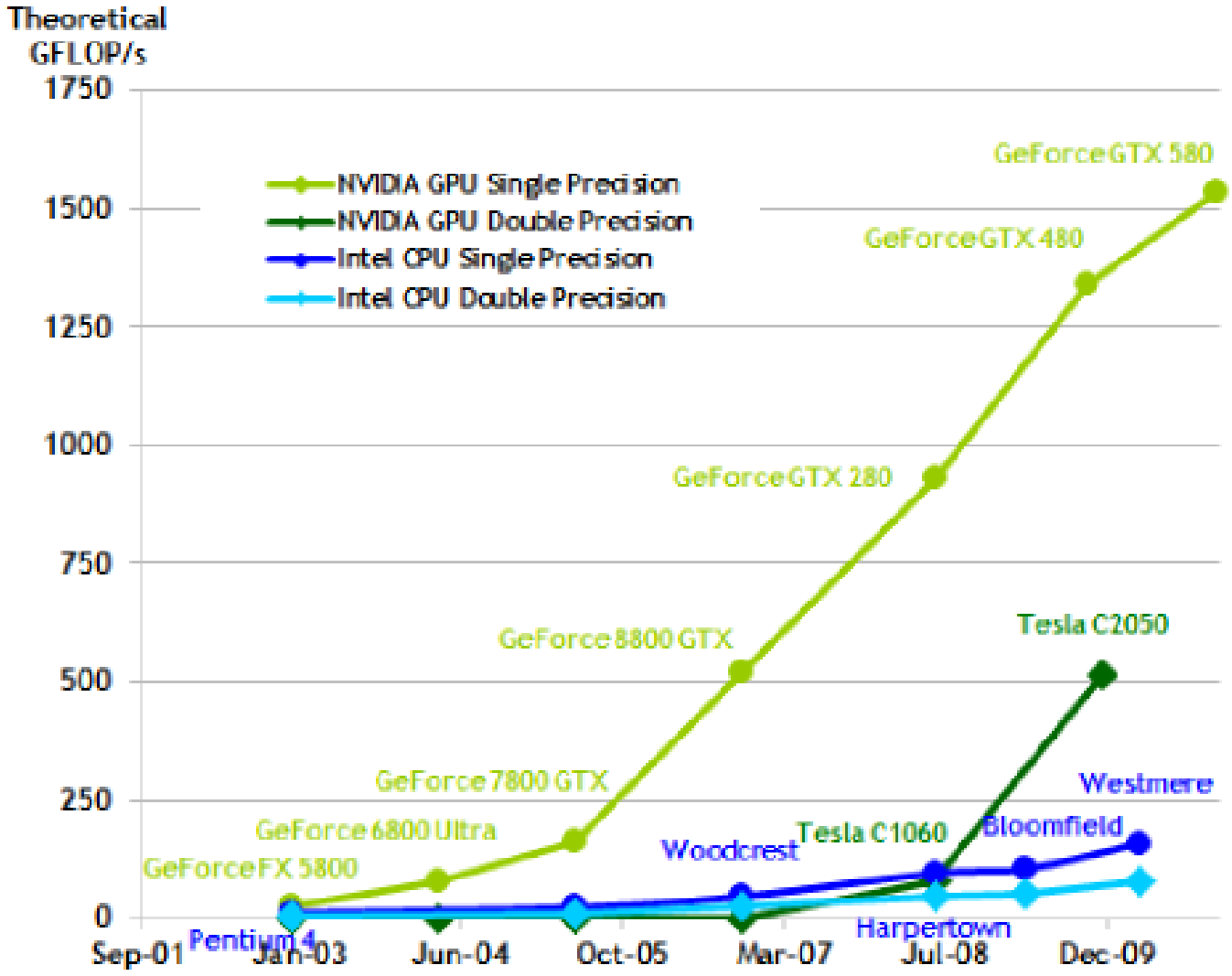
Propósito

Acelerar por hardware el dibujado de primitivas gráficas en una pantalla, descargando al procesador de estas tareas

- 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 ▸ graphics rendering pipeline
  - 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*

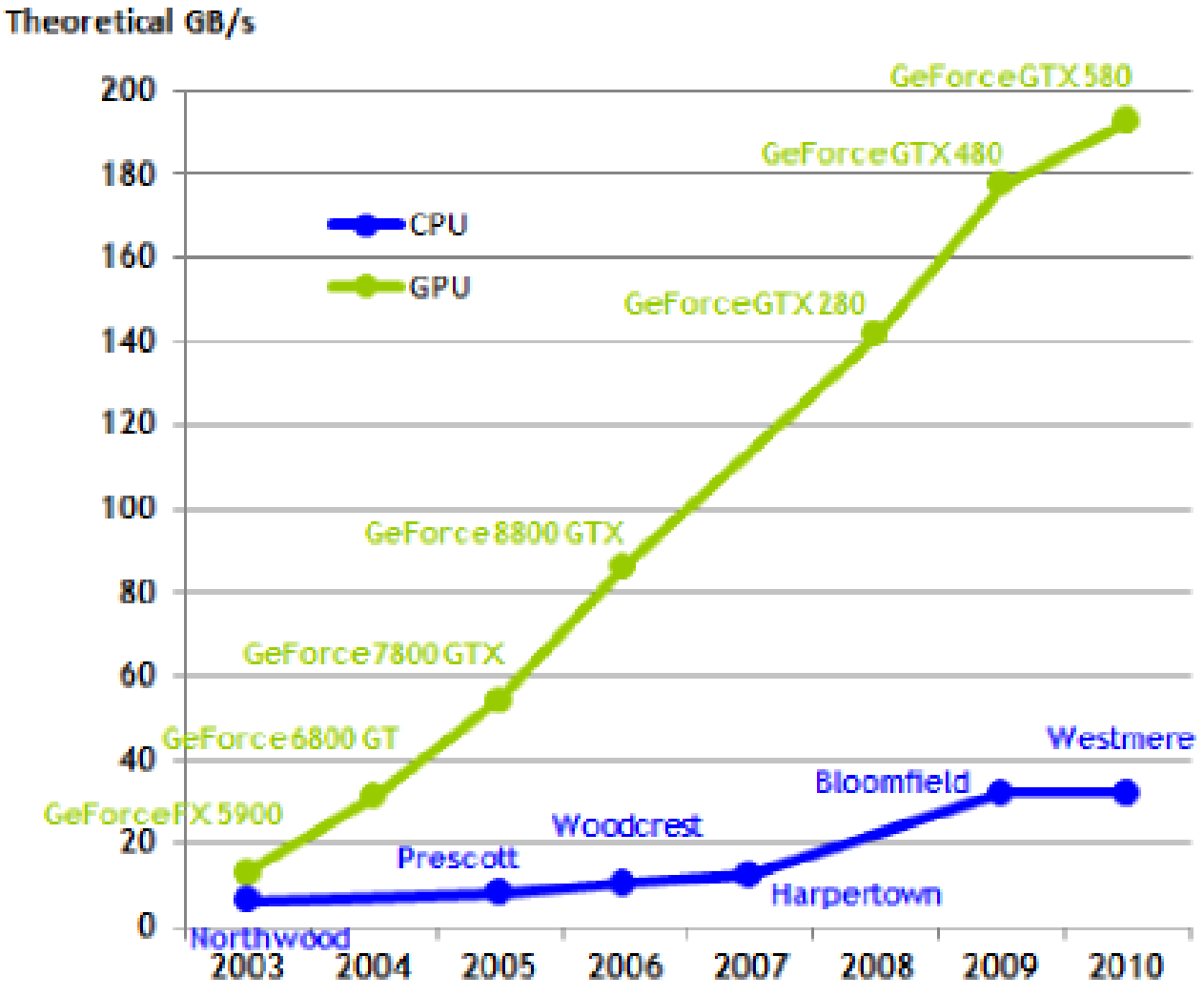
CPUs vs. GPUs

Operaciones en punto flotante por segundo



CPUs vs. GPUs

Ancho de banda en transferencias de memoria



CPUs vs. GPUs
Principales diferencias de diseño

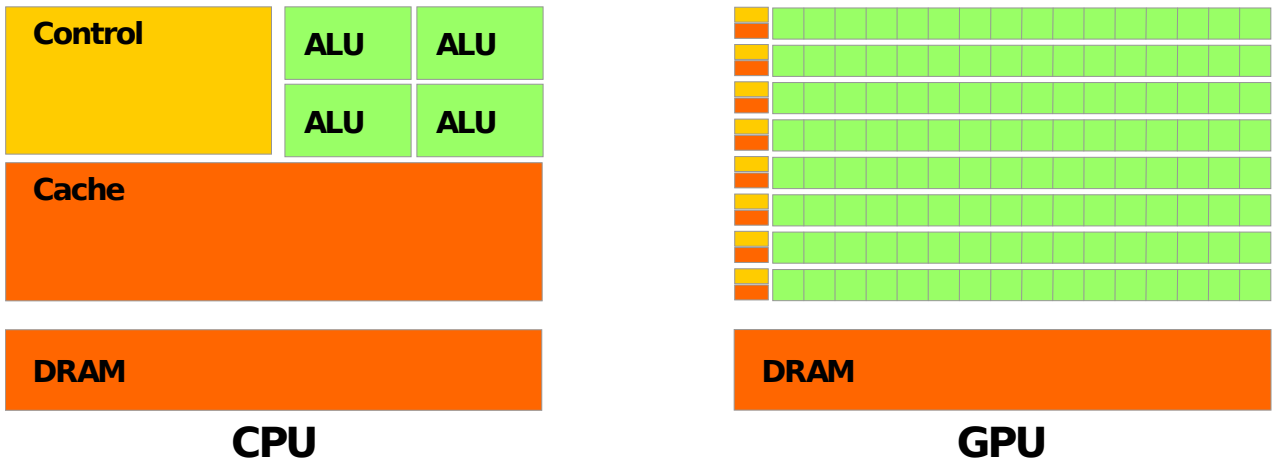


Figura: Diferencias en cantidad de transistores dedicados a procesamiento de datos

CPUs vs. GPUs
Principales diferencias de diseño

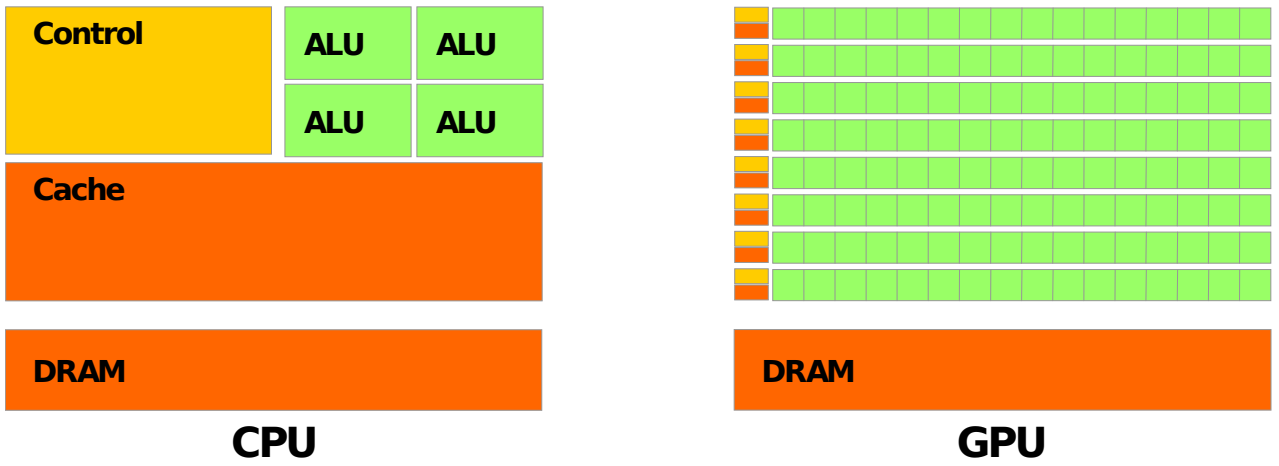


Figura: Diferencias en cantidad de transistores dedicados a procesamiento de datos

CPU Énfasis en

- Más cache
- Control de flujo

GPU Especializada en

- Computación intensiva
- Ejecutar mismas operaciones en paralelo sobre diferentes datos

$\Rightarrow$ Interesa mucho cálculo y poco acceso a memoria

GPUs para cálculo 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

GPUs para cálculo 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

- 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

- 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...

¿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*

$\Rightarrow$ Se aplica kernel a cada elemento en stream

$\star$ Concurrencia: explotación paralelismo de datos

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*

$\Rightarrow$ Se aplica kernel a cada elemento en stream

$\star$ Concurrencia: explotación paralelismo de datos

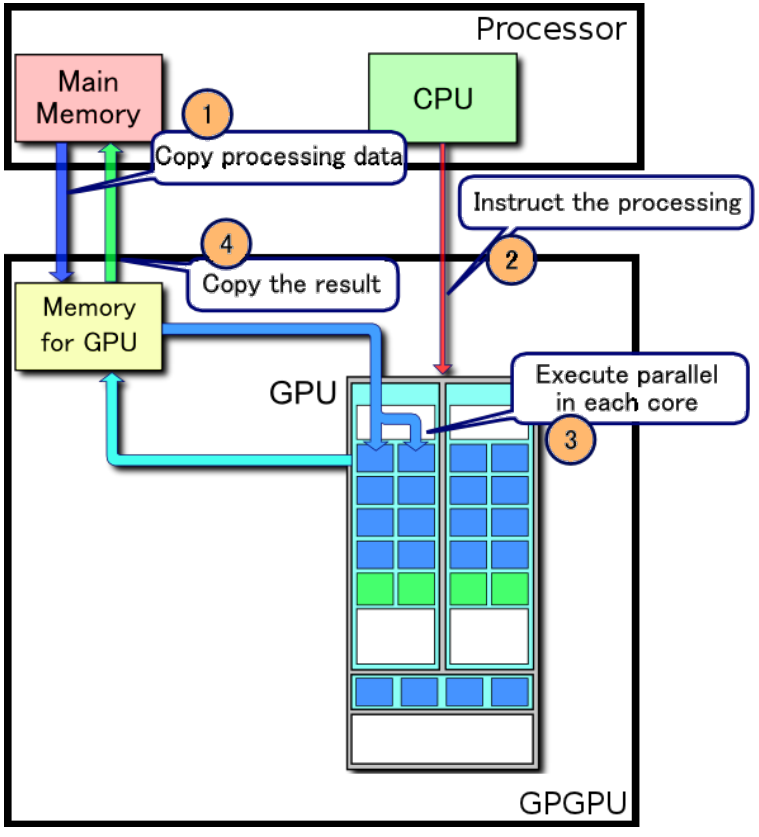


Figura: *Stream processing* se ajusta a modelo GPGPU

Arquitecturas clásicas HPC

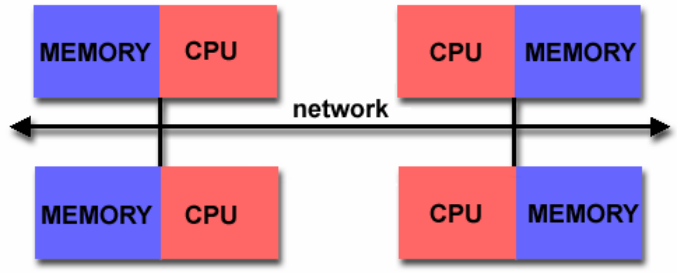


Figura: Sist. memoria distribuida

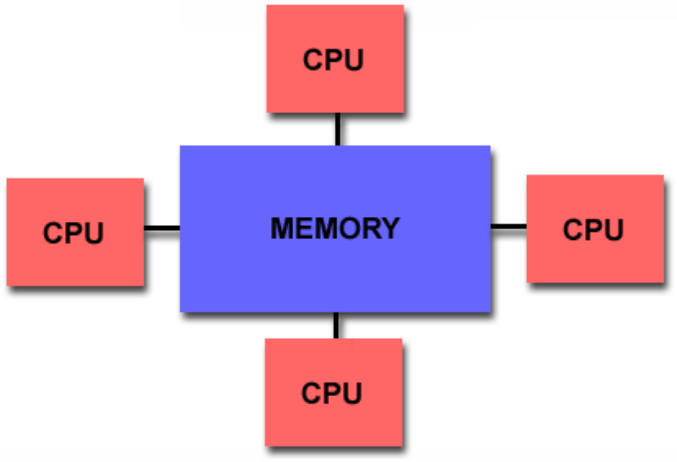


Figura: Sist. memoria compartida

Sistemas multiprocesador-multinúcleo última década
Clusters homogéneos (*Beowulf cluster*)

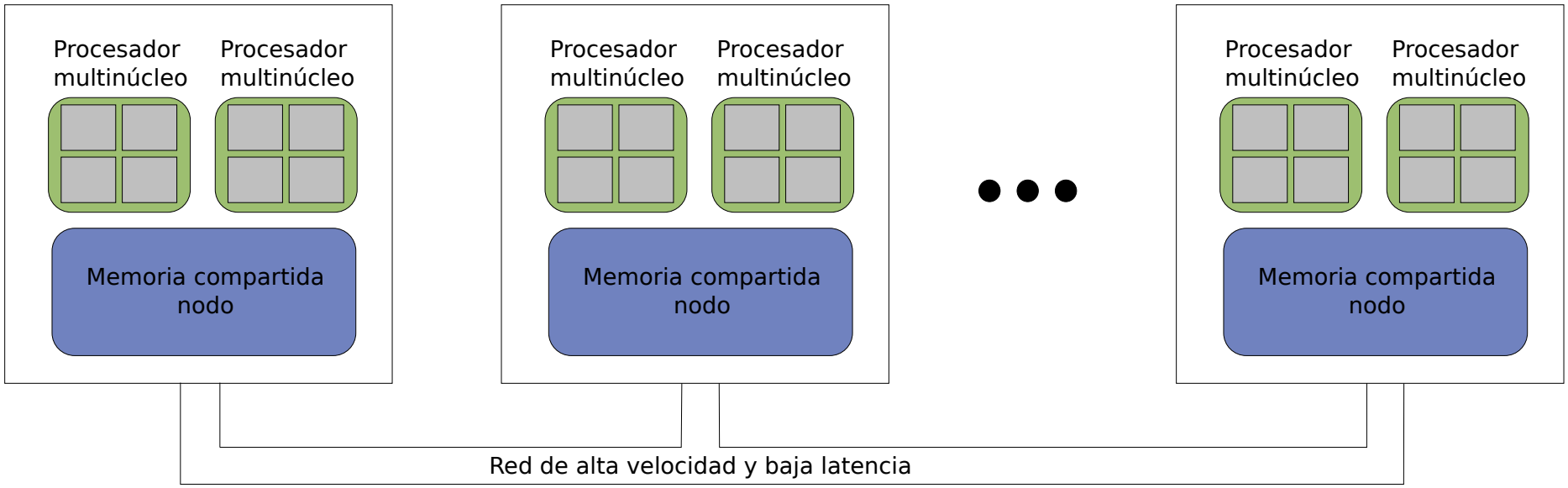


Figura: Cluster homogéneo con nodos multiprocesador y procesadores multinúcleo

Sistemas heterogéneos CPUs-GPUs para HPC
Escenario habitual hoy

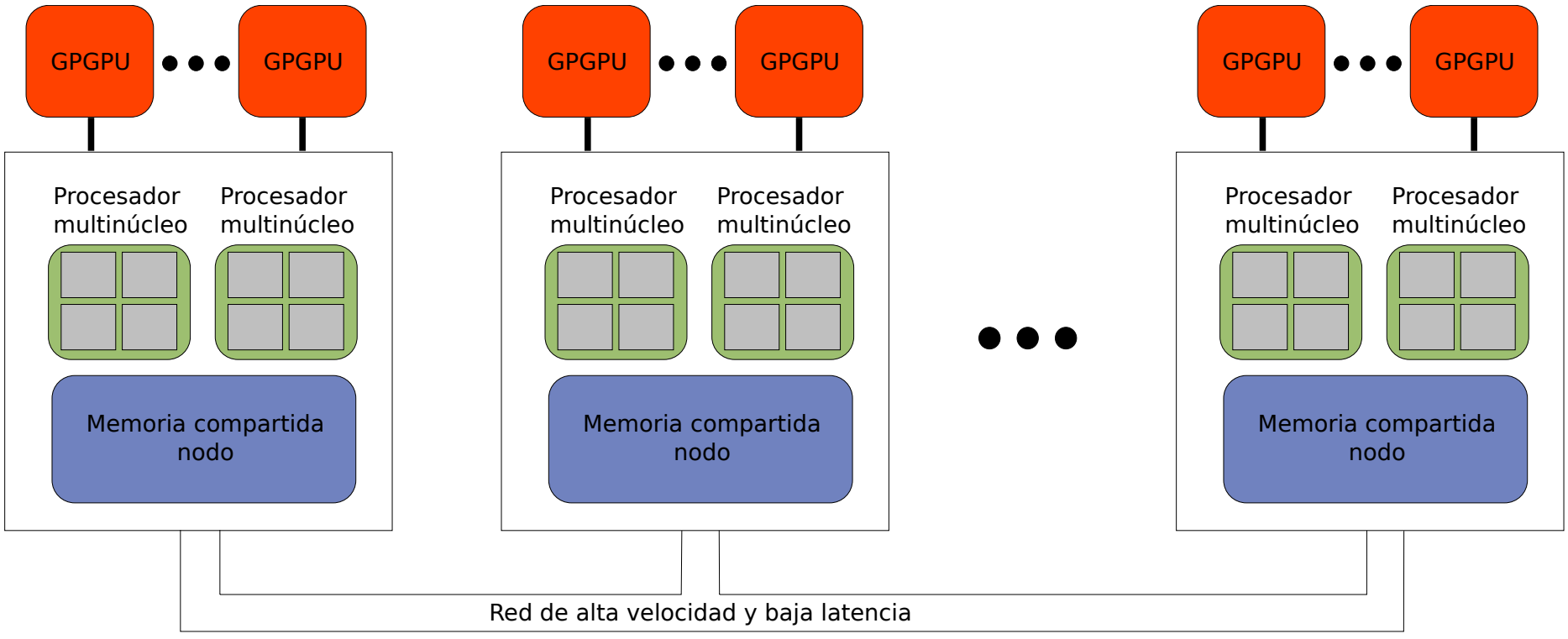


Figura: Cluster híbrido con GPUs y procesadores multinúcleo

Brook, BrookGPU y Brook+

Brook

Lenguaje de programación, extensión de ANSI C, que incorpora ideas del paradigma *Stream processing*

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, 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

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* tarj. gráfica** (cerrado: *freeware*)

CUDA Toolkit (tb cerrado): herram. para programar tarjeta

CUDA SDK (no tan cerrada, pero no libre)

Bibliotecas de cálculo (cerradas)

Mucha y buena documentación

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* tarj. gráfica** (cerrado: *freeware*)

- expone arquitectura GPGPU de la GPU
- pensada para ser explotada con modelo de programación tipo *Stream processing*

CUDA Toolkit (tb cerrado): herram. para programar tarjeta

CUDA SDK (no tan cerrada, pero no libre)

Bibliotecas de cálculo (cerradas)

Mucha y buena documentación

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* tarj. gráfica** (cerrado: *freeware*)

CUDA Toolkit (tb cerrado): herram. para programar tarjeta

- *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

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* tarj. gráfica** (cerrado: *freeware*)

CUDA Toolkit (tb cerrado): herram. para programar tarjeta

CUDA SDK (no tan cerrada, pero no libre)

- Ejemplos y pruebas de concepto

Bibliotecas de cálculo (cerradas)

Mucha y buena documentación

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* tarj. gráfica** (cerrado: *freeware*)

CUDA Toolkit (tb cerrado): herram. para programar tarjeta

CUDA SDK (no tan cerrada, pero no libre)

Bibliotecas de cálculo (cerradas)

- Construidas por encima de *runtime library*
- cuBLAS, cuFFT...

Mucha y buena documentación

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* tarj. gráfica** (cerrado: *freeware*)

CUDA Toolkit (tb cerrado): herram. para programar tarjeta

CUDA SDK (no tan cerrada, pero no libre)

Bibliotecas de cálculo (cerradas)

Mucha y buena documentación

- Guías de programación, de referencia, de buenas prácticas...

CUDA

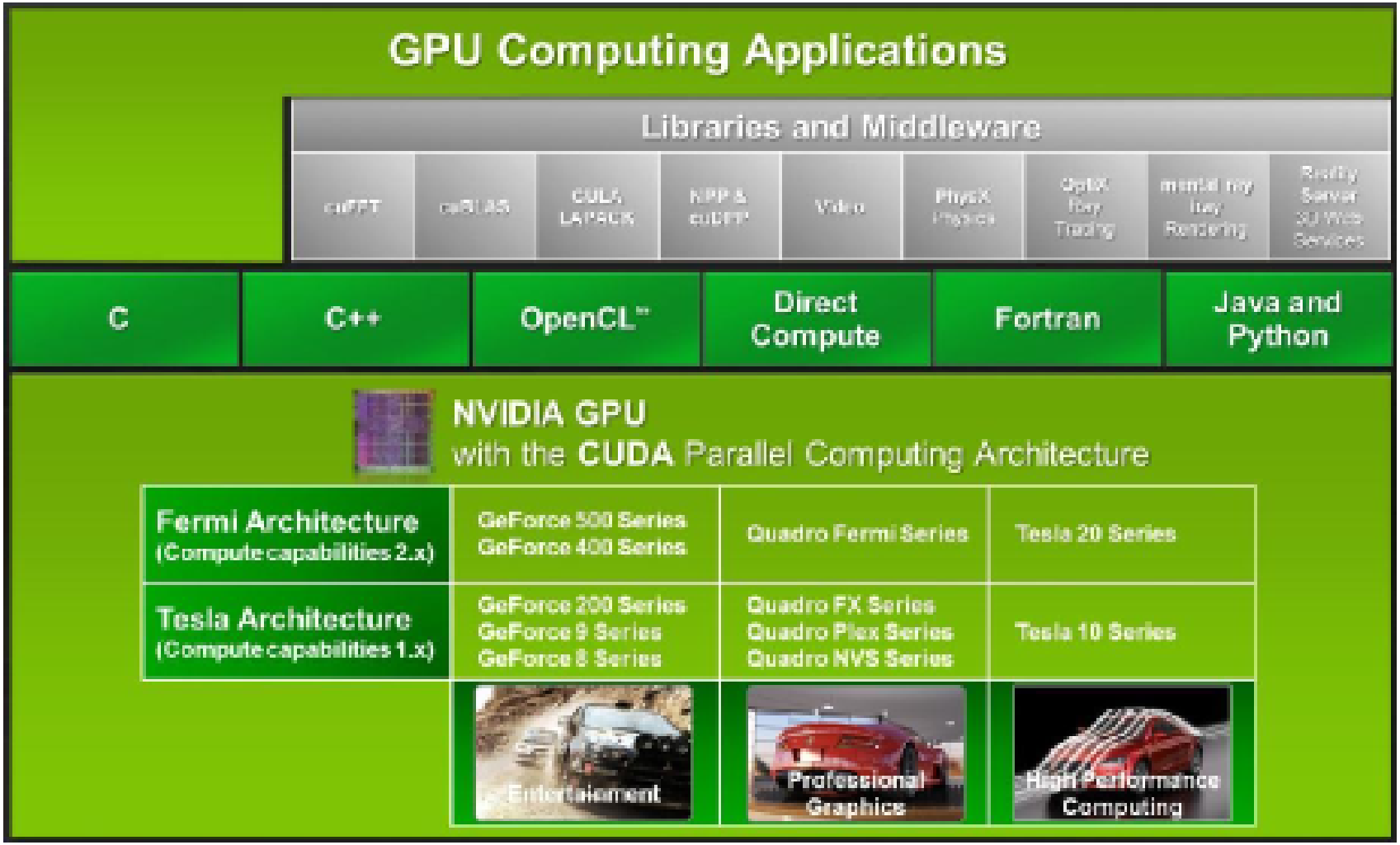


Figura: Arquitectura CUDA

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 mem. compartida de alta velocidad

  $\Rightarrow$ 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)

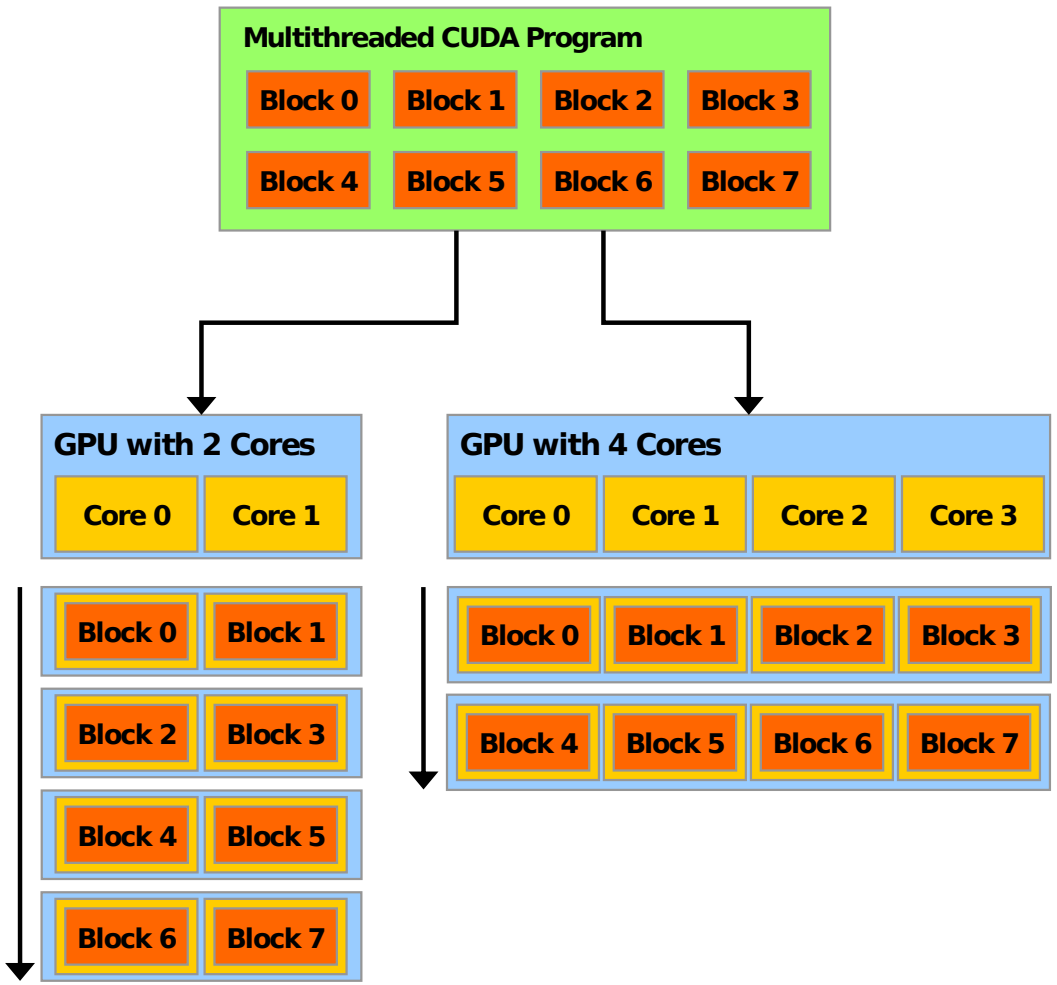


Figura: Ejecución en bloques independientes permite escalabilidad

CUDA
Jerarquía de hilos: bloques y rejilla

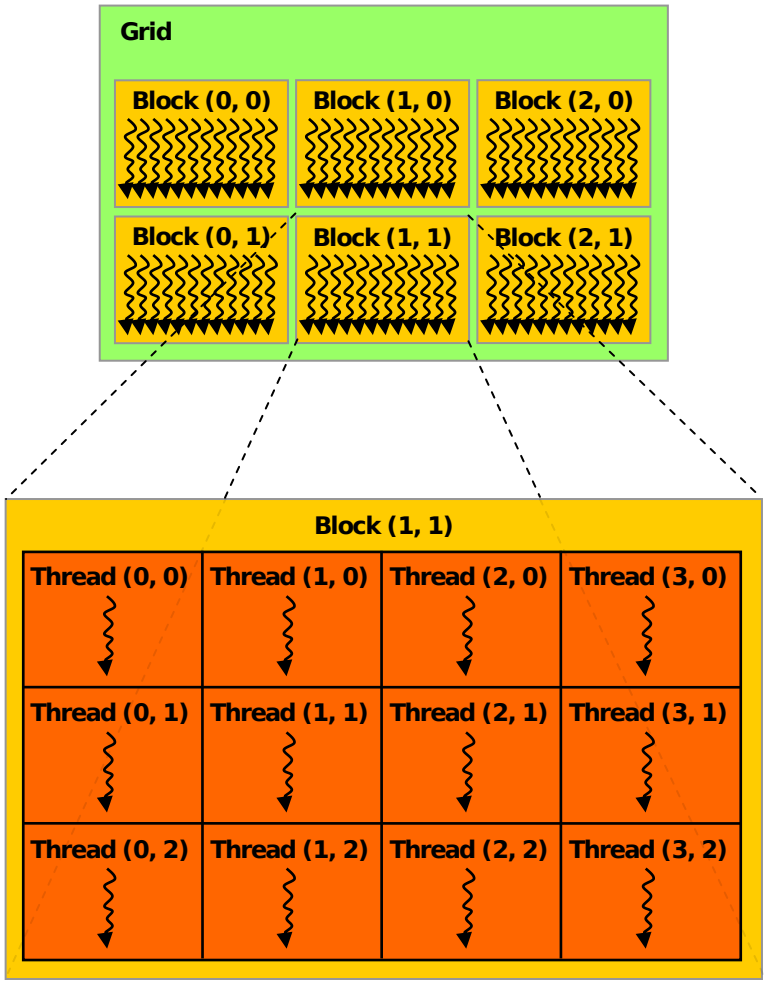


Figura: Rejilla de bloques con hilos que ejecutarán kernel en GPU

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

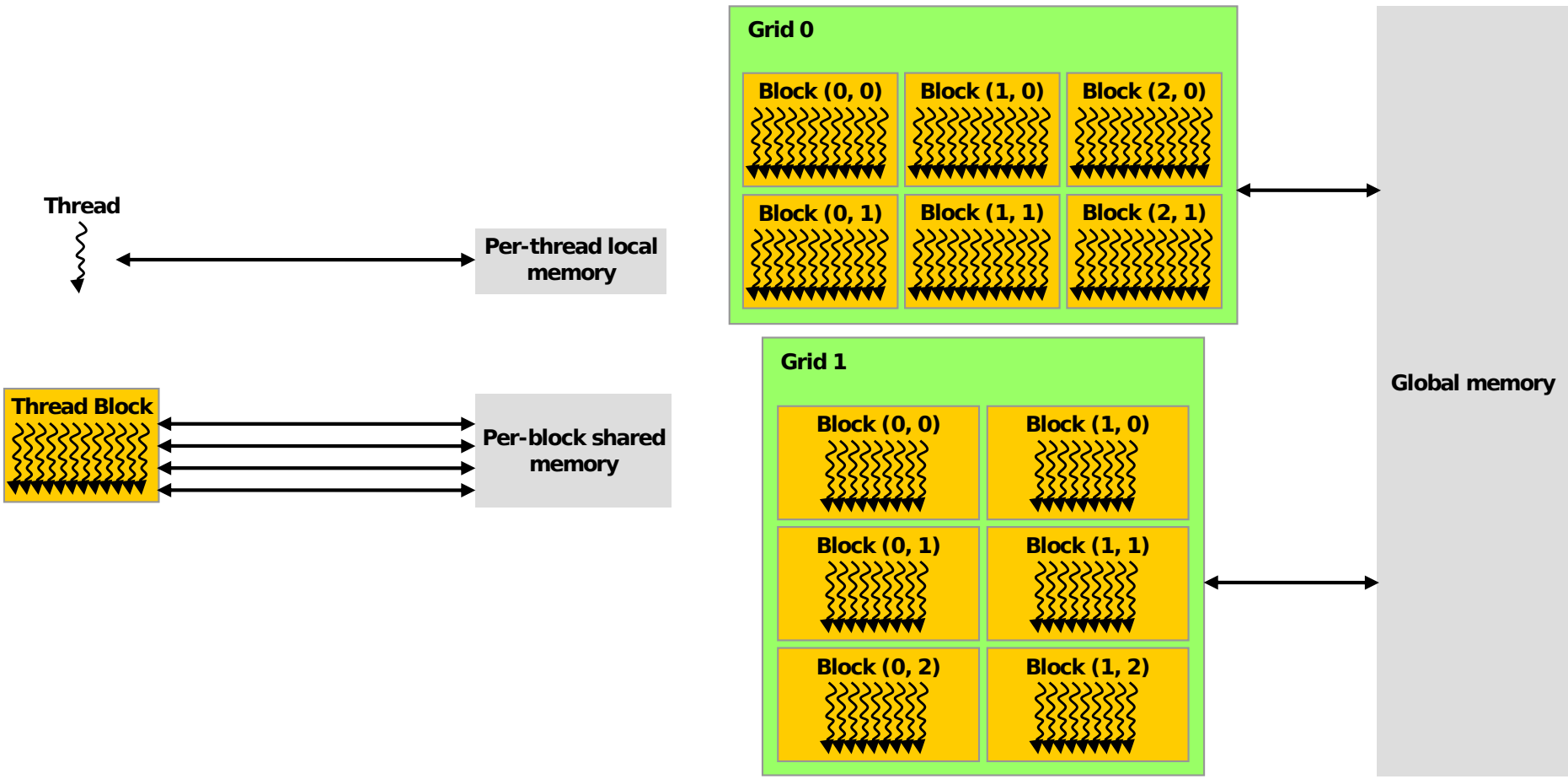


Figura: Principales espacios de memoria disponibles para hilos GPU

CUDA

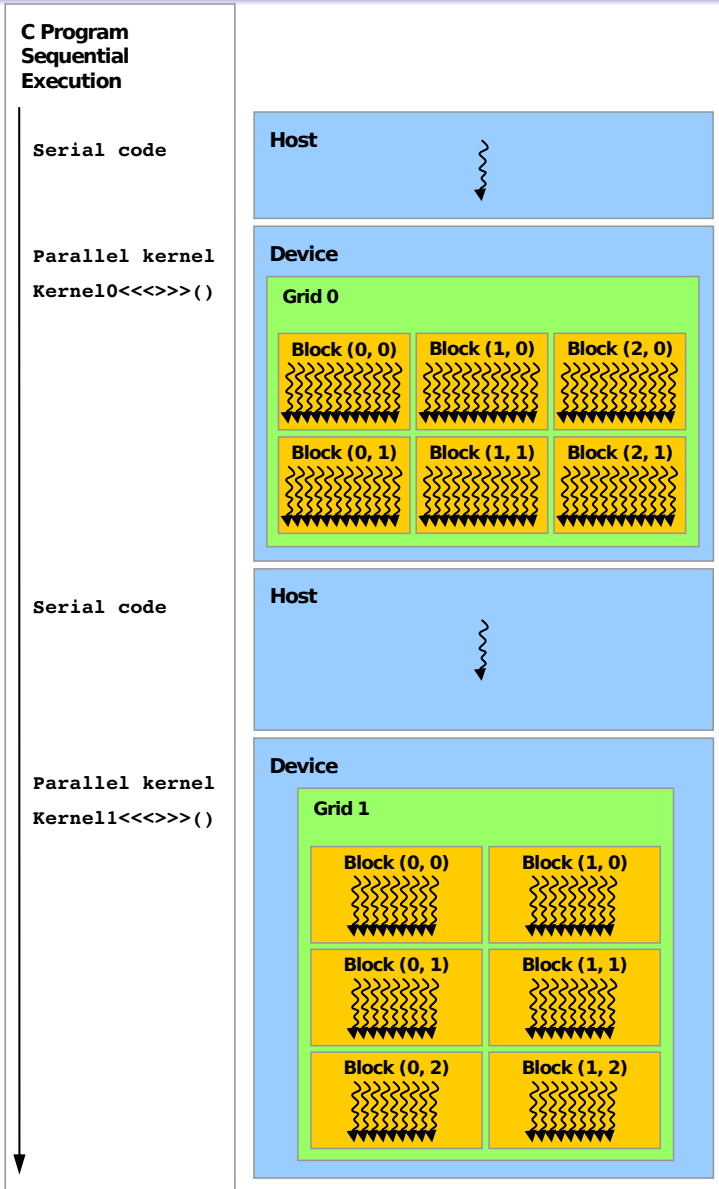
Idea clave 3: Arquitectura de la GPU que nos expone CUDA

- 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
Programación en CUDA

- Paralelizar una aplicación no es fácil (ni en CPU ni GPU)
  - Detectar regiones paralelizables y paradigmas más adecuado
  - 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)
    - Dependendia 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 sist. heterogéneos
    - Solo aprovechamiento de GPGPU
  - Obtener buen rendimiento requiere un gran esfuerzo de programación

OpenCL
Historia

- Desarrollo iniciado por Apple
- Se unen AMD, IBM, Intel y Nvidia
- Enviado como propuesta al Khronos Group
  - Khronos OpenCL Working Group se forma en junio 2008 (forman parte 38 empresas e instituciones)
- OpenCL 1.0 en noviembre 2008
  - Última revisión octubre 2009
- OpenCL 1.1 en junio 2010

OpenCL
Motivación

- Carencia de un estándar de programación para sistemas heterogéneos
- Imposibilidad de tener código multiplataforma para explotar esos sistemas
- Interesa la posibilidad de poder ejecutar código desarrollado para un acelerador (pe. una GPU) en procesadores normales

¿Solución?

Definición de un nuevo estándar

OpenCL
Open Computing Language

- Estándar para programación de propósito general multiplataforma
- Acceso portable y eficiente a plataformas heterogéneas
  - Procesadores 'normales': CPUs
  - Aceleradores: GPUs, CELL...
- Componentes:
  - APIs para una librería de control de plataformas
  - Runtime
  - Lenguaje de programación

OpenCL
Mecanismos de adaptabilidad

- Definición de funcionalidades **básicas** y opcionales
  - Mecanismo de extensiones
- Funciones para averigüar
  - Dispositivos disponibles en sistema
  - Características hardware
  - Extensiones soportadas
- Énfasis en compilación en tiempo de ejecución

OpenCL
Disponibilidad

- Implementaciones de AMD, Nvidia, Intel, IBM, Apple...
- Interfaces
  - Estándar definido para C y C++
  - Integración en Java, Python, Ruby, .NET (C#)...

OpenCL
Modelo de plataforma

- Anfitrión (*Host*)
  - Procesador principal del sistema
  - Ejecuta programa principal
  - Conectado a uno o varios aceleradores
- Dispositivo computacional (*Compute device*)
  - Realiza computaciones solicitadas por anfitrión
  - Consta de una o varias Unidades de computación independientes
- Unidad de computación (*Compute unit*)
  - Formadas por Elementos de procesamiento (*Processing Elements*)
  - Elementos de una misma unidad de computación pueden sincronizarse y operar en SIMD o SPMD

OpenCL
Modelo de ejecución

- Un programa OpenCL consta de
  - Aplicación principal, que corren en el anfitrión
  - Kernels, que se ejecutan en los dispositivos
    - Escritos en lenguaje OpenCL (C) o nativos

Programa principal en anfitrión...

Gestiona ejecución de los kernels

OpenCL
Diccionario CUDA-OpenCL

- Jerarquía de hilos y de memoria muy similar a CUDA

Terminología CUDA	Terminología OpenCL
Thread	Work-item
Block	Work-group
Global memory	Global memory
Shared memory	Local memory
Local memory	Private memory

OpenCL vs. CUDA

- Programabilidad y conceptos similares
  - No suele ser difícil portar los kernels
  - Migrar la parte del host es más costoso
- Rendimiento GPGPU en tarjetas Nvidia
  - En general mejor CUDA, que no parece esforzarse demasiado en su soporte OpenCL
- OpenCL todavía más ‘verde’: observar evolución
- Ventajas claras OpenCL
  - Estándar abierto independiente de plataforma
  - Más general: no solo GPGPU

OpenCL vs. CUDA
Rendimiento

- Comparación rendimiento
  - K. Karimi, N. G. Dickson, F. Hamze, A Performance Comparison of CUDA and OpenCL. *Journal CoRR*, volume abs/1005.2581. 2010 http://arxiv.org/abs/1005.2581
- GPU: NVIDIA GeForce GTX-260 con CUDA/OpenCL 2.3
- Aplicación: Adiabatic QUantum Algorithms (simulación Monte Carlo)

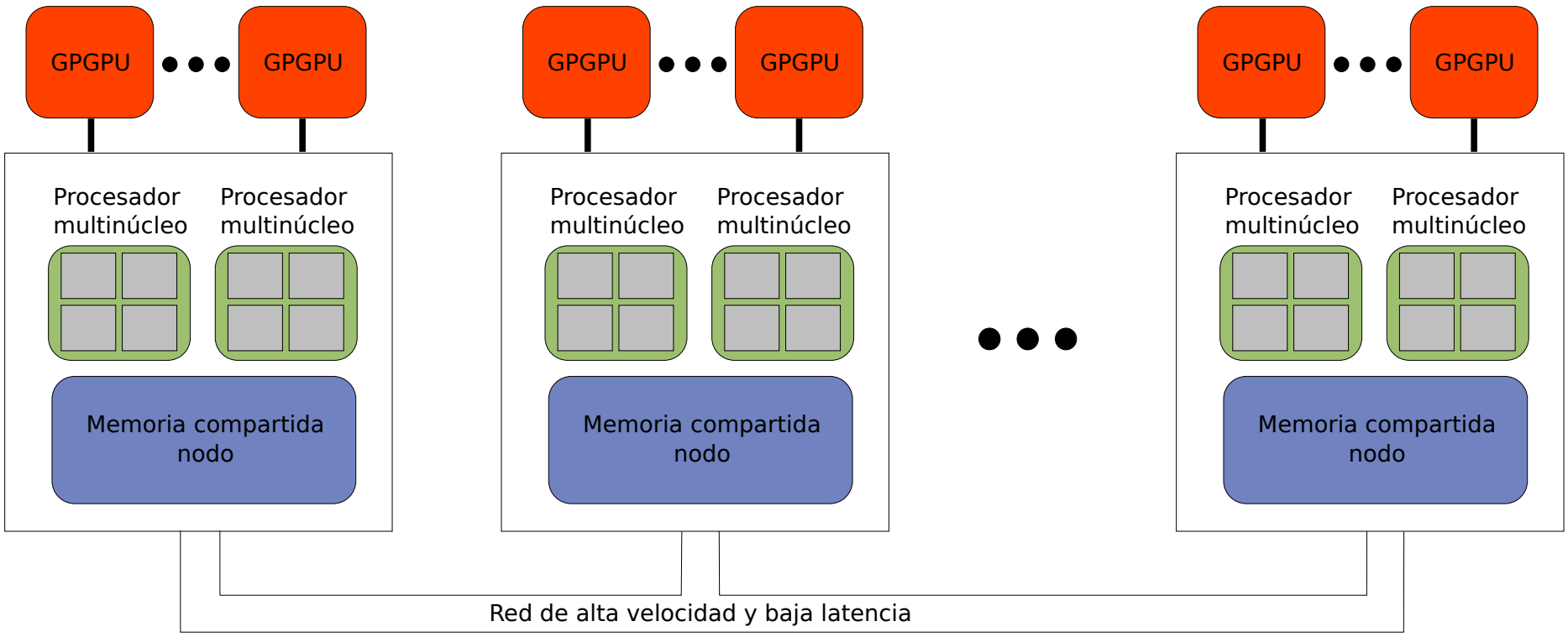
Qubits	GPU Operations Time				End-To-End Running Time			
	CUDA		OpenCL		CUDA		OpenCL	
	avg	stdev	avg	stdev	avg	stdev	avg	stdev
8	1.97	0.030	2.24	0.006	2.94	0.007	4.28	0.164
16	3.87	0.006	4.75	0.012	5.39	0.008	7.45	0.023
32	7.71	0.007	9.05	0.012	10.16	0.009	12.84	0.006
48	13.75	0.015	19.89	0.010	17.75	0.013	26.69	0.016
72	26.04	0.034	42.32	0.085	32.77	0.025	54.85	0.103
96	61.32	0.065	72.29	0.062	76.24	0.033	92.97	0.064
128	101.07	0.523	113.95	0.758	123.54	1.091	142.92	1.080

OpenCL vs. CUDA
Rendimiento

- Comparación rendimiento
  - K. Karimi, N. G. Dickson, F. Hamze, A Performance Comparison of CUDA and OpenCL. *Journal CoRR*, volume abs/1005.2581. 2010 http://arxiv.org/abs/1005.2581
- GPU: NVIDIA GeForce GTX-260 con CUDA/OpenCL 2.3
- Aplicación: Adiabatic QUantum Algorithms (simulación Monte Carlo)

Qubits	Kernel Running Time				Data Transfer Time			
	CUDA		OpenCL		CUDA		OpenCL	
	avg	stdev	avg	stdev	avg	stdev	avg	stdev
8	1.96	0.027	2.23	0.004	0.009	0.007	0.011	0.007
16	3.85	0.006	4.73	0.013	0.015	0.001	0.023	0.008
32	7.65	0.007	9.01	0.012	0.025	0.010	0.039	0.010
48	13.68	0.015	19.80	0.007	0.061	0.010	0.086	0.008
72	25.94	0.036	42.17	0.085	0.106	0.006	0.146	0.010
96	61.10	0.065	71.99	0.055	0.215	0.009	0.294	0.011
128	100.76	0.527	113.54	0.761	0.306	0.010	0.417	0.007

Programación paralela en clusters híbridos CPUs-GPUs



- Sistemas difíciles de programar hoy (de forma eficiente)
  - MPI + OpenMP/pthreads + CUDA
  - MPI + OpenCL

Gestión básica de cluster multi-gpu con CUDA
Salvo CUDA, todo SwL

- Gestión de varias GPUs para uso transparente multiusuario:
  - **cuda_wrapper**
- Sistemas de colas y manejo de recursos HPC que tengan en cuenta GPUs:
  - Slurm
- GPGPU en remoto de forma transparente
  - rCUDA (Proyecto español, UPV)

Conclusiones

- GPGPU con software libre todavía no es posible a 100 %
  - Drivers no libres necesarios
  - Hay trabajo al respecto para solucionar esto:
    - Nvidia: Pathscale http://github.com/pathscale/pscnv
    - ATI: Soporte OpenCL para Gallium http://cgit.freedesktop.org/mesa/clover
- Apostar por OpenCL: Especificación abierta no ligada a un fabricante

Paralelismo en computadores
Tipos o clases de paralelismo

a nivel de bit *Bit-level parallelism*

- Tamaño palabra mayor $\Rightarrow$ más paralelismo
- 8, 16, 32, 64, 128 bits

Paralelismo en computadores
Tipos o clases de paralelismo

a nivel de bit *Bit-level parallelism*

- Tamaño palabra mayor $\Rightarrow$ más paralelismo
- 8, 16, 32, 64, 128 bits

a nivel de instrucción *Instruction-level parallelism* (ILP)

- Objetivo: Hacer más trabajo por ciclo de reloj
- Varias instruc. en ejecución al mismo tiempo
  - *Pipeline* o cauce de ejecución
  - Múltiples unidades ejecución y ALUs
  - Reordenamiento y ejecución fuera de orden

Paralelismo en computadores
Tipos o clases de paralelismo

a nivel de bit *Bit-level parallelism*

- Tamaño palabra mayor $\Rightarrow$ más paralelismo
- 8, 16, 32, 64, 128 bits

a nivel de instrucción *Instruction-level parallelism* (ILP)

- Objetivo: Hacer más trabajo por ciclo de reloj
- Varias instruc. en ejecución al mismo tiempo
  - *Pipeline* o cauce de ejecución
  - Múltiples unidades ejecución y ALUs
  - Reordenamiento y ejecución fuera de orden

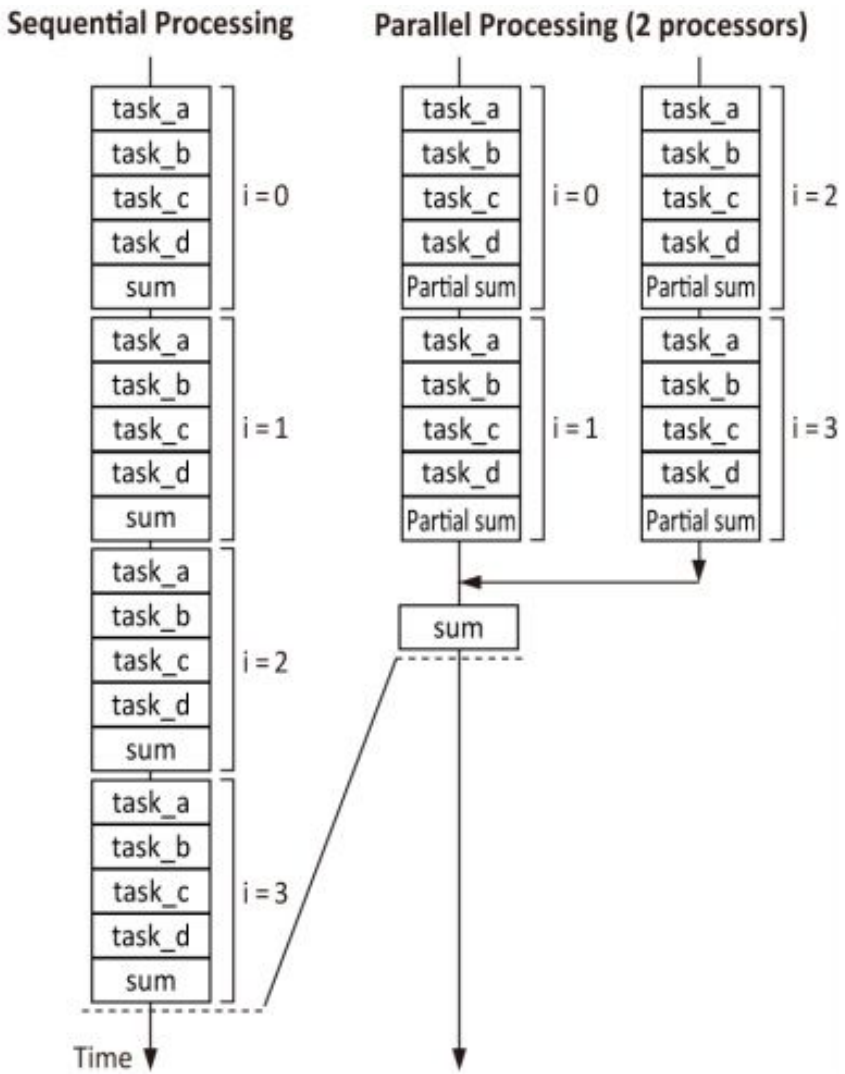
de datos *Data parallelism*

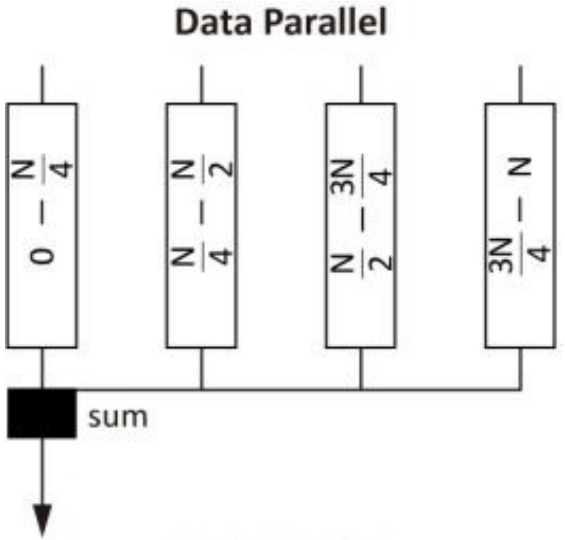
- Poca o nula dependencia entre datos
- Concurrencia: misma operación(es) sobre distintos datos

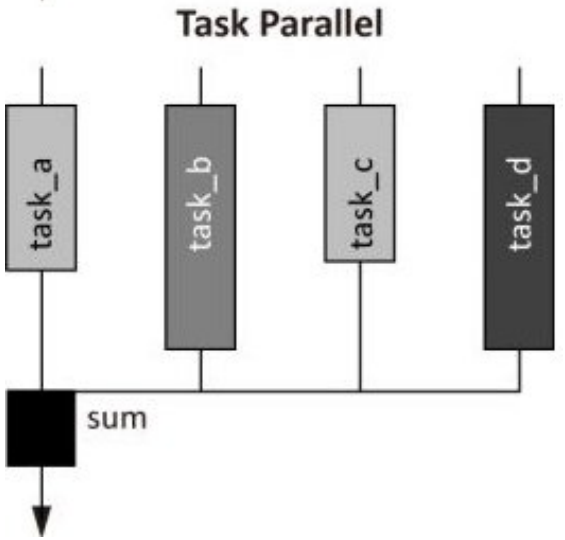
de tareas *Task parallelism*

- Concurrencia: varias tareas simultáneas en distintas unid. ejecución

Paralelismo en computadores
Paralelismo de datos y paralelismo de tareas







◂ Volver

El pipeline gráfico
graphics rendering pipeline

Pipeline (cauce)

Cadena con varios pasos independientes

- Idea: acelerar producción (*n* veces con *n* pasos)
- Eslabón más lento determina velocidad máxima

- *Pipeline* gráfico: 3 pasos o fases conceptuales:

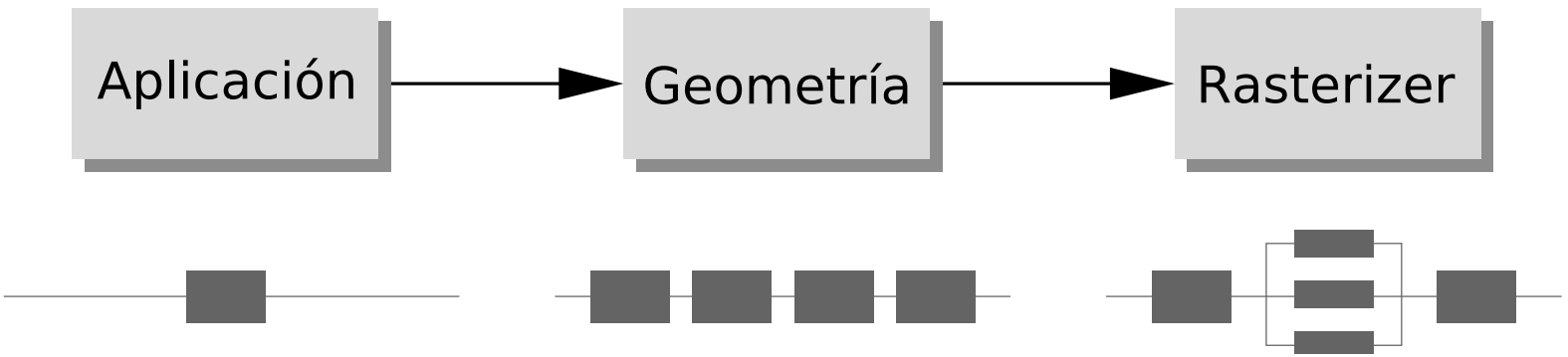


Figura: *Pipeline* básico para *rendering*

- Velocidad de *render* (**fps**, *frames per second*)

Pipeline gráfico

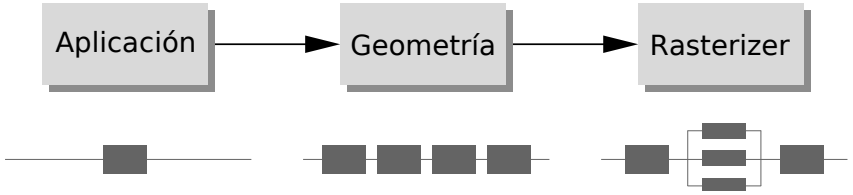


Figura: *Pipeline* básico para *rendering*

- Fase de **Aplicación**: única puramente *software*
  - se genera geometría para las siguientes fases
  - detección de colisiones, . . .

Pipeline gráfico

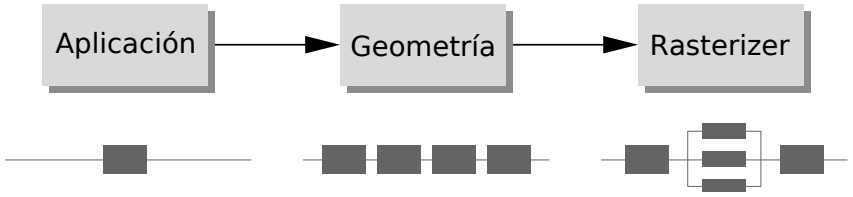
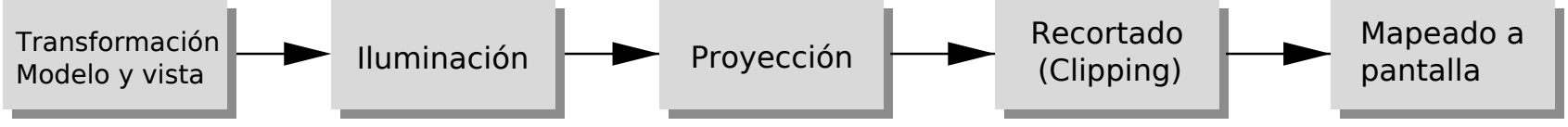


Figura: *Pipeline* básico para *rendering*

- Fase de **Aplicación**: única puramente *software*
  - se genera geometría para las siguientes fases
  - detección de colisiones, . . .
- Fase de **Geometría**
  - mayoría de operaciones sobre **polígonos** y **vértices**



Pipeline gráfico

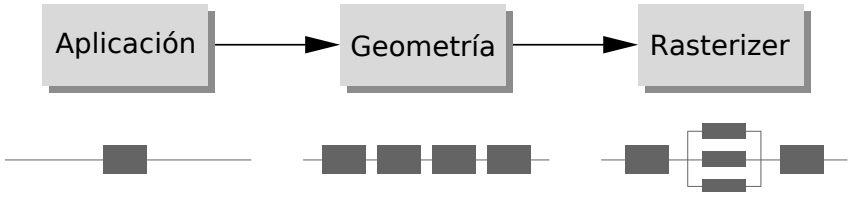
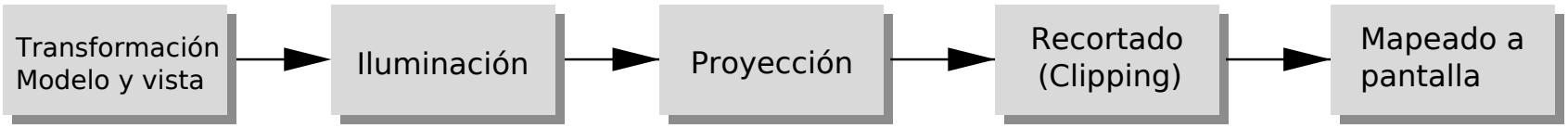


Figura: *Pipeline* básico para *rendering*

- Fase de **Aplicación**: única puramente *software*
  - se genera geometría para las siguientes fases
  - detección de colisiones, . . .
- Fase de **Geometría**
  - mayoría de operaciones sobre **polígonos** y **vértices**



- Fase de ***Raster***

Volver