Vocabulario

Tema 3: Arquitecturas paralelas

Parte 1

- **Ley de Moore:** El número de transistores que se pueden integrar en un dispositivo con un coste determinado se duplica cada 18 meses. La reducción de tamaño posibilita un aumento de la frecuencia del reloj.
- **Paralelismo:** Hace referencia al procesamiento paralelo que es el tipo de computación que permite realizar varios cálculos simultáneamente.
- **Programa:** Conjunto ordenado de instrucciones (programador). Fichero ejecutable almacenado en memoria (SO)
- **Proceso:** Realización de un trabajo, asociándole los recursos que necesita: espacio de memoria I+D y tiempo de procesador (SO)
- **Hebra (thread):** Secuencia de instrucciones que se crea asociada a un proceso concreto. Todas las hebras de un proceso comparten recursos (memoria)
- **Ejecución concurrente:** Varios procesos a ejectuar se turnan en el uso de los recursos (tiempo compartido)
- **Paralelismo a nivel de instrucción (ILP):** Estrategias y técnicas orientadas a la ejecución en paralelo de instrucciones maquina próximas dentro de un programa concreto
- **Segmentación (pipeline):** No es una estrategia pura de paralelizacion, sino más bien una especialización. El incremento del rendimiento está limitado por el número de etapas del procesador.
- **Procesadores superescalares:** Paralelismo real multiplicando el caude de ejecución completo. Implementa técnicas que permiten romper el flujo secuencias de instrucciones.

Parte 2

- **Procesadores supersegmentados:** Aplican el concepto de segmentación a dos niveles, a nivel de diseño global (segmentación convencional, etapas) y a nivel interno, dentro de sus unidades funcionales. Consiste en subdividir las etapas en varias subetapas permitiendo así que varias instrucciones ocupen una unidad funcional sin tener que replicarla.
- **Limitaciones en el ILP:** La ejecución paralela de instrucciones no se puede hacer de forma indiscriminada, dado que existen restricciones: Dependencia de datos verdadera, dependencia relativa al procedimiento, conflictos en los recursos, dependencias de saida, antidependencia.
- **Paralelismo SIMD (Single Instruction, Multiple Data):** Aplicar una misma instrucción sobre un conjunto de dato simultáneamente, en lugar de aplicarlo a un solo dato.

Parte 3

- **VLIW (Very Long Instruction Word):** Esquema arquitectónico orientado a soportar ILP de forma explícita.
- **Filosofia VLIW:** Se basa en descargar al procesador de la toma de decisiones sobre como paralelizar un programa.
- **Transmeta Crusoe:** Compatible con x86, ejecuta una máquina virtual CMS que traduce las instrucciones a moléculas VLIW.
- **Intel Itanium:** Longitud de 128 bits, pudiéndose ejecutar 2 mazos de instrucciones en paralelo. Posee 128 registros GPR 64 bits y 128 FP de 82 bits. Posee también 30 UF divididas en 11 grupos.
- **Arquitectura MIMD:** Basadas en organizar n procesadores, n flujos de instrucciones y n flujos de datos; cada procesador puede trabajar de forma asíncrona bajo un flujo de instrucciones procedente de su propia unidad de control.
- **Procesadores fuertemente acoplados:** Los procesadores comparten un único espacio de direcciones. Se comunican a raves de variables compartidas en memoria, cualquier procesador tiene acceso a cualquier posición de memoria, a través de operaciones carga/almacenamiento.
    - **Modelos de acceso a memoria compartida**
        - **UMA:** Memoria física compartida por todos los procesadores, los periféricos se comparten de la misma forma. Son más sencillos pero difícilmente escalables.
        - **NUMA:** El tiempo de acceso a memoria depende de la ubicación de la palabra dentro del espacio de almacenamiento, estando la memoria compartida distribuida por los procesadores (locales). Mas complejos pero fácilmente escalables
        - **COMA:** Caso especial de NUMA, convirtiéndose la memoria principal de los procesadores en cache (no local). No existe jerarquía de memoria todas las caches forman un espacio de direccionamiento global.
        - **CcNUMA:** Controla la coherencia de la cache de NUMA con protocolos específicos: MSIF, SCI, QPI etc.
        - **Problema coherencia caches**
            - **Solución estática:** Clasificar las variables compartidas en varios tipos: solo lectura, lectura múltiple, lectura/escritura única, lectura/escritura múltiple
            - **Solución dinámica protocolos:** De actualización en escritura, de invalidación en escritura, protocolo snoopy y protocolos basados en directorios.

- **Procesadores débilmente acoplados**
    - **Cluster:** Conjunto de computadores interconectados ediante una red de alta velocidad, de modo que operan de forma coordinada creando la abstracción de un único computador de alto rendimiento.
    - **Middleware:** Capa de abstracción entre el sistema operativo y las aplicaciones, con el fin de gestionar un cluster proporcionando diversos servicios.
    - **Paso de mensajes:** Sistema de comunicación entre sistemas débilmente acoplados que no suelen compartir un mismo espacio de memoria.

Tema 4: CUDA

Parte 1: Evolución, rendimiento y programación de la GPU

- **Hardware gráfico, usos:** Visualizacion 2D en PCs de sobremesa. Ejecucion de video (descompresión y/o visualización). Graficos 3D, particularmente en la industria de los video-juegos.
- **Transistores, uso:** Incrementar la resolución en pantalla. Incrementar el frame-rate. Incrementar el nivel de detalle de las escenas.
- **Aceleración especifica (envió kernel a GPU) depende de:** Si el kernel de la GPU admite un numero suficiente de hilos concurrentes. Si el flujo de datos y el flujo de control se adaptan bien a la forma de computar. Si el código resulta bastante hostil a la CPU y además se realizan optimizaciones del código.
- **Aceleración global (aplicación) depende:** Factor logrado por el kernel y del peso que tenga dicho kernel en el tiempo de ejecución global.
- **Limitación de rendimiento (Amdahl):** El factor de mejora que podemos lograr la aplicación depende del porcentaje de tiempo que consuma el kernel más que del número de procesadores que disponga nuestra GPU.
- **Falsas creencias**
    - **El consumo es gratis, los transistores son caros**
    - **Apostar por el paralelismo a nivel de instrucción**
    - **Las operaciones aritméticas son lentas, el acceso a memoria es rápido**
    - **Se cumple la leye de monr 2x/18 meses**
- **Procesadores multi-core:** Un chip compuesto de varios procesadores que ofrece más rendimiento a baja frecuencia y además proporciona una mayor eficiencia energética.
- **Green computing:** Híbridos entre GPU y CPU.

Parte 2: Introducción a las arquitecturas masivamente paralelas (GPUS)

- **SIMD:** Todos los nucleos ejecutan la misma instrucción al mismo tiempo, solo se necesita decodificar la instrucción una única vez para todos los nucleos.
- **Procesadores masivamente paralelos:** GPUs
- **Primera generación (chips):** Compuesta de SM "Streaming Multiprocessor": 8 núcleos cada uno de ellos con 2048 registro, hasta 128 hilos por nucleo, 16KB de memoria compartida y 8KB de cache para constantes.
- **Generación Fermi (chips):** Cada SM "Streaming Multiprocessor": 32 nucleos, cada uno con 1024 registros, 48 hilos por nucleo, 64KB de memoria compartida / Cache L1, Cache L2 común a los SM, 8JB de cache para constantes.
- **Núcleos dentro de un SM:** Todos los nucleos ejecutan la misma instrucción simultáneamente pero con distintos datos. Similar a la computación CRAY. Con un minimo de 32 hilos realizando la misma tarea al mismo tiempo.
- **Múltiples hilos:** Gran cantidad de hilos es la clave del alto rendimiento (no penalización cambios de contexto)
- **CUDA (Compute Unified Device Architecture):** Hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creada por NVIDIA.
- **1º Parte programa cuda:** Código Host en la CPU que hace interfaz con la GPU
- **2º Parte programa cuda:** Código Kernel que se ejecuta sobre la GPU
- **1ºAPI Device:** Runtime simplificada, mas sencilla de usar.
- **2º API Device:** Driver mas flexible, mas compleja de usar.
- **Kernel:** es una función la cual al ejecutarse lo hara en N distintos hilos en lugar de en secuencial.
- **Grid:** forma de estructurar los bloques en el kernel (bloques 1D, 2D) (Hilos/Bloques: 1D, 2D o 3D)
- **Bloque:** Agrupación de hilos. Cada bloque se ejecuta sobre un solo SM. Un SM puede tener asignados varios bloques.
- **Invocacion Kernel:** kernel_routime<<<gridDIm, blockDim>>>(args)
    - **GridDim:** número de bloques. Tamaño del grid
    - **blockDim:** número de hilos que se ejecutan dentro de un bloque
    - **Args:** número limitado de argumentos, normalmente punteros a memoria de la GPU
- **Informacion hilos (ejeutado kernel):** Variables pasadas por argumento, punteros a memoria de la GPU, constantes globales en memoria GPU.
    - **Variabless especiales**
        - **gridDim:** tamaño o dimensión de la malla de bloques
        - **blockIdx:** índice del bloque (propia de cada bloque)
        - **blockDim:** tamaño o dimensión de cada bloque
        - **thereadIdx:** índice del hilo (propio de cada hilo)
- **blockDim.x, blockDim.y:** Acceder índices de conjunto de bloques
- **threadIdx.x, threadIdx.y:** Acceder índices de conjunto de hilos

- **Dim3:** Es un tipo especial de CUDA con 3 componentes x,y,z por defecto se inicializa a 1.

Parte 3: Modelo de hilos de procesamiento CUDA

- **Escalabilidad transparente:** Hardware asigna libremente los bloques en los procesadores
- **Planificación Hilos:** Cada bloque se divide en warpss de 32 hilos. Los hilos de un warp se ejecutan físicamente en paralelo (warp unidad de planificación en los SMs)
- **Múltiples bloques (8x8):** tenemos 64 hilos por bloque. Cada SM puede albergar hasta 1535 hilos, por lo que tenemos 24 bloques. Sin embargo cada SM solo puede ejecutar hasta bloques, por lo que solo 512 hilos se ejecutarán sobre 1 SM.
- **Múltiples bloques (16x16):** tenemos 256 hilos por bloque. Cada SM puede albergar hasta 1535 hilos, por lo que tenemos 6 bloques. Con estos 6 bloques podemos aprovechar la capacidad máxima de ejecución de 1536 hilos por SM.
- **Múltiples bloques (32x32):** tendremos 1024 hilos por bloque. Solo podremos ejecutar 1 bloque dentro de un SM en la arquitectura Fermi. Usando solo 2/3 de la capacidad de ejecución de hilos máxima (1536 hilos).
- **Sincronización hilos:** Para sincronizar los hilos de forma interna al kernel (dentro de un mismo SM) se utiliza→ _syncthreads(). Si se utiliza el código condicional hay que asegurarse que todos los hilos alcancen la instrucción de sincronización.
- **Control de flujo:** Todos los hilos de un mismo warp ejecutan al mismo tiempo la misma instrucción. Los kernels con varios caminos posibles se preparan para que todos ejecuten la misma instrucción pero cada hilo ejecute su respectiva funcionalidad
- **Divergencia:** Problema de rendimiento
- **API:** Extension del lenguaje de programación C
    - **Extensiones de lenguaje:** Para etiquetar aquellas partes del código a ejecutar en el device
    - **Librería:** el tiempo de ejcucion dividida en:
        - **Common component:** Provee tipo de datos y un subconunto de la librería C para ejecutarse en el device y en el host
        - **Host component:** Funciones para controlar y acceder a uno o más devices desde el host.
        - **Device component:** Provee funciones especificas para el manejo del device
- **Dim3 gridDim:** Dimensiones del grid en bloques
- **Dim3 blockDim:** Dimensiones del bloque en hilos
- **Dim3 blockIdx:** Índice del bloque dentro del grid
- **Dim3 threadIdx:** Índice del hilo dentro del bloque

- **Int warpSize:** Tamaño del warp
- **Calificadores de tipo funcion**
    - **__global__:** Se ejecuta en la GPU y se llama siempre desde el host. No permite un número variable de argumentos, recursión, puntero a función y variables estáticas. Tiene que ser tipo void, y tiene como límite de parámetros 256 bytes. Calificador para kernels CUDA
    - **__device__:** Se ejecuta en la GPU y se llama siempre desde la GPU. No permite un numero variables de argumentos, recursión y variables estaticas
    - **__host__:** Se eejcuta en el host, se llama siempre desde el host
    - **__host__ __device__:** Funcion para el host y la GPU
- **Calificadores de tipo variable**
    - **__device__:** Reside en la GPU. Reside en memoria global, su tiempo de vida es el de la aplicación y es accesible por todas las tareas en el grid y desde el host.
    - **__constant__:** puede combinarse con __device__ , la variable reside en la memoria constante, su tiempo de vida es el de la aplicación y es accesible por todas las tareas del grid y desde el host.
    - **__shared__:** puede combinarse con __device__ , la variable reside la memoria shared de un bloque, su tiempo de vida es el del bloque y es accesible solo por las tareas del bloque.
- **Reserva memoria:** cudaMalloc(), cudaFree(), cudaMemcpy(), cudaMemcpyToSymbol()
- **Invocacion kernel:** <<<Dg, Db, Shm, Stream>>>(..)
    - **Dg:** (tipo dim3) dimensión y tamaño del grid
    - **Db:** (tipo dim3) dimensión y tamaño de bloque
    - **Shm:** (tipo size_t): cantidad de memoria compartida reservada por bloque para esta llamada (en bytes, 0 por defecto)
    - **S:** (tipo cudaStream): especifica stream asociado (0 por defecto)
- **Funciones matemáticas:** pow, squrt, sinh, cosh etc. Cuando son ejecutadas en el host se utiliza la implementación de C si esta disponible, funciones soportadas para tipos escalares.
- **Función sincronización:** void_syncthreads(). Una vez que todos los hilos han alcanzan este punto la ejecución se reanuda normalmente. Es utilizado para evitar RAW/WAR/WAW problemas cuando se accede a la memoria compartida global (permitidos en construcciones condicionales, si condicional es uniforme a través de todo el bloque de hilos)

Parte 4: Uso y manejo de memorias en CUDA

- **Localidad temporal:** Un dato usado recientemente es probable que se use otra vez a corto plazo
- **Localidad espacial:** Localidad espacial: es probable usar los dato adyacentes a los usados, por ello se usan caches para guardar varios datos en una línea del tamaño del bus.
- **Memorial global:** Declarada y manejada desde la parte del host, ubicación para los datos de la aplicación, normalmente vectores y las variables globales del sistema. Es la memoria compartida por todos los SM y no es cacheable (acceso “coalesced”)
- **Memoria constante:** Valores no pueden ser cambiados, optimizada para lecturas, su uso aumenta notablemente la velocidad del sistema. Su velocidad es similar a los registros sin necesidad de ocuparlos (propia memoria cache)
- **Registros:** Ubicación para las variables usadas en los kernels. Cada SM dispone de un número de registros a repartir entre el número de hilos. Útil para grandes aplicaciones, su manejo es esencial.
- **Memoria local:** Destinada a la información local de cada hilo, para variables o vectores locales que no caben en sus respectivos registros. Tiene bastante capacidad y su velocidad es como la memoria global (mejoradas mediante caches L1/L2)
- **Memoria compartida:** Dedicada a cada multiprocesador, compartida por todos los hilos de un mismo bloque, útil para compartir información. Puede necesitar instrucciones de control de acceso y flujo para controlar los accesos. Normalmente se copian los datos desde la memoria global en la memoria compartida
- **Variables globales:** puede ser modificada y leída por cualquier kernel, su tiempo de vida es el tiempo de vida de la aplicación. Pueden ser leídas y modificadas por código host utilizando rutinas especiales (cudaMemcpyToSymbol, cudaMemcpyFromSymbol)
- **Variables constantes:** Similares en las globales excepto que no pueden ser modificadas por los kernels. Muy utilizadas en las aplicaciones para almacenar valores constantes.
- **Arrays locales:**
- **Transferencia bloqueante:** cudaMemcpy
- **Transferencia asíncrona:** Necesita memoria pinned.

Parte 5: OPENCL, otros lenguajes y librerías externas

- **OpenCL:** Plataforma cruzada de computación paralela sobre dispositivos heterogéneos. Asegura la correcta ejecución pero no el rendimiento sobre los distintos dispositivos.
    - **OpenCL Hardware abstraction:** Trata CPUs multi-nucleo, GPUs y otros aceleradores como devices. Cada devices contiene una o mas 'Computing Units' cada uno de estos contiene uno o mas 'Processing elementes' SIMD
- **Librerias basadas en CUDA:** Diferentes librerías para varios ámbitos
    - **CUBLAS:** operaciones sobre algebra lineal
    - **CUFFT:** transformadas rápidas de Fourier (1D, 2D, 3D)
    - **CUDPP:** operaciones paralelas primitivas
- **Librerias Thrust:** Abstracción de la programación de los kernel. Implementa varios algoritmos paralelos de datos (máximo, sima, multiplicación de vectores). Similar a la librería STL de C++, elige automáticamente el código más rápido en tiempo de compilación.
    - **Tipos de datos**
        - **Host:** thrust::host_vector<int> H(4);
        - **Device:** thrust::device_vector<int> D(5)
    - **Algoritmos**
        - **Transformations**
        - **Reductions**
        - **Prefix-Sums**
        - **Reordering**
        - **Sorting**
    - **Iteradores:** estructuras útiles para operar sobre los datos y los algoritmos
        - **Constant_iterator**
        - **Counting_iterator**
        - **Tranform_iterator**
        - **Permutation_iterator**
        - **Zip_iterator**
- **CUVILib:** librería que implementa sobre GPU algoritmos de visión por computador y procesamiento de imágenes.
- **MultiGPU:** Conexión de varias GPUs al bus PCI-express. Posibilidad de aprovechar distintos device en un mismo computador. Gestión muy fácil de varios devices desde aparición CUDA 4.0
- **PGI x86:** Herramienta para construir de forma transparente aplicaciones CUDA en sistemas GPU y no GPU creados por cualquier fabricante
    - **PGI x86 Compiler:** Creación de binarios unificada, aprovechamiento de los recursos disponibles. Programación mediante anotaciones en el código.