



Guía de Buenas prácticas en CUDA

Programación Paralela

José María Cecilia

Grado en Ingeniería informática

- **Assess, Parallelize, Optimize, Deploy (APOD).**
- **Heterogenous Computing.**
 - Differences between Host and Device.
 - ¿Qué podemos ejecutar en una GPU Nvidia?
- **Evaluación de la aplicación (Profiling).**
 - **Profile.**
 - Crear un profile.
 - Identificar puntos críticos.
 - Entender cómo escala el programa
 - Strong Scaling y Ley de Amdahl.
 - Weak Scaling y Ley de Gustafson.
 - Aplicar Weak y Strong Scaling.
- **Comenzando a paralelizar.**
 - Librerías CUDA.
 - Compiladores paralelizantes.
 - Codificando para extraer el paralelismo

- **Assess, Parallelize, Optimize, Deploy (APOD).**
- Heterogenous Computing.
 - Differences between Host and Device.
 - ¿Qué podemos ejecutar en una GPU Nvidia?
- Evaluación de la aplicación (Profiling).
 - Profile.
 - Crear un profile.
 - Identificar puntos críticos.
 - Entender cómo escala el programa
 - Strong Scaling y Ley de Amdahl.
 - Weak Scaling y Ley de Gustafson.
 - Aplicar Weak y Strong Scaling.
- Comenzando a paralelizar.
 - Librerías CUDA.
 - Compiladores paralelizantes.
 - Codificando para extraer el paralelismo

Recomendaciones y buenas prácticas

- Están ordenadas por prioridad.
- Acciones que presentan una mejora substancial para la mayoría de aplicaciones CUDA tienen mayor prioridad.
- Pequeñas optimizaciones que solo afectan a pequeñas situaciones tienen menor prioridad.
- En este caso nos olvidamos de los errores pero en una aplicación CUDA se deben tener en cuenta llamando a `cudaGetLastError()`.

Introducción

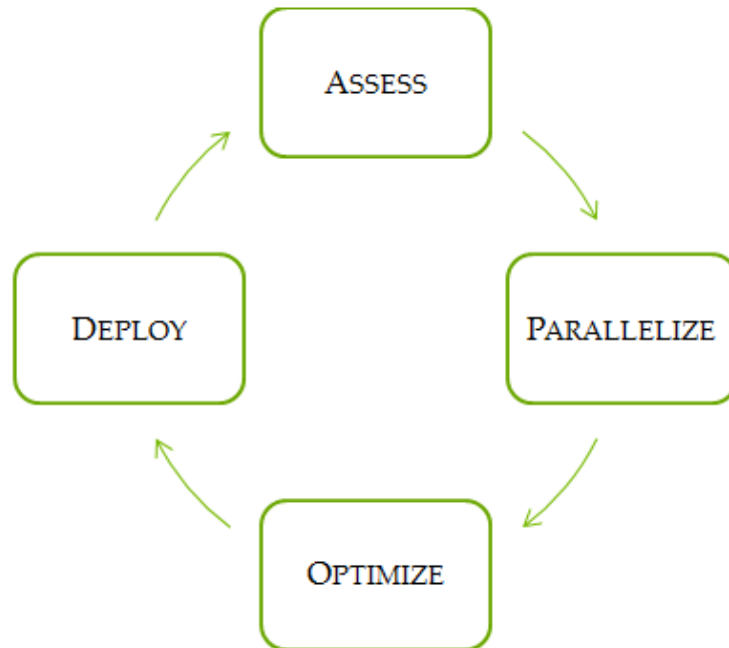
- Las buenas prácticas de CUDA son consejos de programación para obtener el máximo rendimiento de vuestros códigos CUDA.
- Necesitamos una cierta experiencia con C y programación básica CUDA
- Para ello es muy aconsejable leer:
 - CUDA Getting Started Guide
 - CUDA C Programming Guide
 - CUDA Toolkit Reference Manual

APOD: Assess, Parallelize, Optimize, Deploy

- El ciclo de diseño CUDA está basado en 4 grandes procesos: Evaluar, Paralelizar, Optimizar, Implementar
- Objetivos:
 - Identificar qué partes del código son susceptibles de aceleración en la GPU.
 - Conseguir rápidamente conocer los beneficios de usar GPUs.
 - Iniciar el aprovechamiento de las aceleraciones resultantes en vuestros códigos lo antes posible

APOD: Assess, Parallelize, Optimize, Deploy

- APOD es un proceso cíclico.
 - Las aceleraciones se pueden conseguir, testear y desarrollar con poca inversión de tiempo.
 - Una vez el ciclo comienza otra vez podemos evaluar mas oportunidades de aceleraciones y conseguir códigos incluso más rápidos en producción.



Assess (Evaluación)

- El primer paso es la evaluación de la aplicación.
 - Localizar partes del código que tienen mayor peso en el tiempo total de ejecución de la aplicación.
 - Utilizar profilings: Pin, Valgrind, etc...
- Con esta evaluación podemos evaluar los cuellos de botella y empezar la aceleración de la aplicación en la GPU.
- El desarrollador puede establecer una **cota del rendimiento óptimo** de su aplicación mediante:
 - Entender los requisitos de la aplicación y sus restricciones.
 - Ley de Amdahl y ley de Gustafson

Parallelize (Paralelización)

- El segundo paso es la paralelización de la aplicación.
- Depende del código puede ser muy sencillo llamando a librerías como cuBLAS, cuFFT o Thrust o usando OpenAcc
- Pero también puede ser complejo:
 - Repensar la aplicación para buscar el paralelismo con una visión de datos orientada al *throughput*.
 - OJO! No es exclusivo de las GPUs, también en las CPUs

Optimize (Optimizar)

- El tercer paso es la optimización de la aplicación.
- No hay bala de plata. Hay que aprovechar el proceso iterativo de APOD.
 - Identificar una oportunidad de optimización.
 - Aplicar la optimización.
 - Verificar el speedup conseguido
 - Repetir el proceso.
- No buscar todas las optimizaciones a la vez. Realizar el proceso de manera incremental.
- Varios niveles: Desde solapar transferencias de datos con computación hasta tunear las secuencias de operaciones en punto flotante.
- Utilizar herramientas de profiling: Visual Profiler, Occupancy calculator, etc...

Deploy (Implementar)

- Comparar la salida con el tiempo esperado en la etapa de evaluación.
- La etapa inicial de evaluación permite establecer un límite superior de rendimiento a conseguir acelerando ciertas partes del código.
- Crear versiones para ver resultados.
- ¡Mejor evolucionario que revolucionario!

- Assess, Parallelize, Optimize, Deploy (APOD).
- **Heterogenous Computing.**
 - Differences between Host and Device.
 - ¿Qué podemos ejecutar en una GPU Nvidia?
- Evaluación de la aplicación (Profiling).
 - Profile.
 - Crear un profile.
 - Identificar puntos críticos.
 - Entender cómo escala el programa
 - Strong Scaling y Ley de Amdahl.
 - Weak Scaling y Ley de Gustafson.
 - Aplicar Weak y Strong Scaling.
- Comenzando a paralelizar.
 - Librerías CUDA.
 - Compiladores paralelizantes.
 - Codificando para extraer el paralelismo

Computación heterogénea

- Desarrollar una aplicación CUDA supone ejecutar código en dos plataformas a la vez.
 - Host: 1 o mas CPUs
 - Device: 1 o mas Nvidia GPUs.
- Son plataformas diferentes diseñadas para distintos escenarios.
- Es muy importante entender bien las diferencias.

Diferencias entre Host y Device

- Las principales diferencias modelo de hilos y memorias físicas separadas:
 - ***Recursos para ejecución de hilos:***
 - Host CPUs:
 - Los pipeline de ejecución en CPUs solo pueden un **limitado número de threads** concurrentes.
 - Servidores que tienen 4 procesadores de 6 cores cada uno solo pueden correr de 24-48 hilos (si hyperthreading).
 - Device GPUs
 - Unidad mínima de planificación son 32 hilos (**Warp**).
 - Las últimas GPUs pueden tener hasta 1536 hilos por SM ejecutándose.
 - En GPUs con 16 SMs hablamos de **24,000 hilos concurrentes** activos.

Diferencias entre Host y Device

– **Hilos:**

- Host CPUs:

- Son hilos muy **pesados**.
- El SSOO debe planificar los hilos dentro y fuera de los canales de ejecución de la CPU para conseguir la capacidad de multithreading.
- **Cambio de contexto** entre dos hilos lento y caro.

- Device GPUs

- Hilos **muy ligeros**.
- Miles de hilos son encolados para un trabajo (en warps de 32).
- Si la GPU tiene que esperar a un warp, pasa a ejecutar a otro warp -> **Forma de ocultar latencias**.
- Como los registros se reservan estáticamente por hilo, no hay necesidad de cambio de contexto. Reservamos los recursos hasta que los hilos mueren.

Diferencias entre Host y Device

- **RAM:**
 - Host CPUs:
 - Memoria RAM (DDR-3).
 - Device GPUs
 - Memoria Device (GDDR-5)
 - Conectada por el bus PCI-Express

- Assess, Parallelize, Optimize, Deploy (APOD).
- Heterogenous Computing.
 - Differences between Host and Device.
 - **¿Qué podemos ejecutar en una GPU Nvidia?**
- Evaluación de la aplicación (Profiling).
 - Profile.
 - Crear un profile.
 - Identificar puntos críticos.
 - Entender cómo escala el programa
 - Strong Scaling y Ley de Amdahl.
 - Weak Scaling y Ley de Gustafson.
 - Aplicar Weak y Strong Scaling.
- Comenzando a paralelizar.
 - Librerías CUDA.
 - Compiladores paralelizantes.
 - Codificando para extraer el paralelismo

¿Qué ejecutar en una GPU CUDA?

- Aspectos para decidir qué ejecutar en la GPU:
 - Computaciones que se desarrollan sobre muchos datos de manera simultánea (Aritmética sobre matrices. Misma operación sobre muchos datos) -> Tener **millones de hilos muy ligeros**.
 - Para el mejor rendimiento, tener **un patrón coherente de acceso** de memoria para hilos adyacentes.
 - Algunos patrones de acceso a memoria permiten que el hardware **fusiona en una operación lecturas y escrituras a memoria: *Coalescing***
 - Minimizar las transferencias por el PCIe:
 - La complejidad de las operaciones a realizar debe **justificar el coste** de transferencia (Ratio Operations : Data Transferred).
 - Los datos tienen que permanecer en la device memory el máximo tiempo posible

- Assess, Parallelize, Optimize, Deploy (APOD).
- Heterogenous Computing.
 - Differences between Host and Device.
 - ¿Qué podemos ejecutar en una GPU Nvidia?
- **Evaluación de la aplicación (Profiling).**
 - **Profile.**
 - Crear un profile.
 - Identificar puntos críticos.
 - Entender cómo escala el programa
 - Strong Scaling y Ley de Amdahl.
 - Weak Scaling y Ley de Gustafson.
 - Aplicar Weak y Strong Scaling.
- Comenzando a paralelizar.
 - Librerías CUDA.
 - Compiladores paralelizantes.
 - Codificando para extraer el paralelismo

Evaluar tu aplicación (Application Profiling)

- Muchos códigos centran la mayor parte del tiempo de ejecución en **muy pocas líneas** del total de la aplicación.
- **Profilers** **identifican esas líneas** para que los programadores piensen cómo paralelizarlas.
- Hay muchos enfoques para hacer **profiling** pero el objetivo es siempre el mismo:
 - Identificar la **función o funciones críticas** de la aplicación (*Hotspot*).
- **¡Alta Prioridad!:** Para maximizar la productividad del desarrollador, evalúa tu aplicación para encontrar los cuellos de botella y funciones críticas.

- Assess, Parallelize, Optimize, Deploy (APOD).
- Heterogenous Computing.
 - Differences between Host and Device.
 - ¿Qué podemos ejecutar en una GPU Nvidia?
- **Evaluación de la aplicación (Profiling).**
 - **Profile.**
 - **Crear un profile.**
 - Identificar puntos críticos.
 - Entender cómo escala el programa
 - Strong Scaling y Ley de Amdahl.
 - Weak Scaling y Ley de Gustafson.
 - Aplicar Weak y Strong Scaling.
- Comenzando a paralelizar.
 - Librerías CUDA.
 - Compiladores paralelizantes.
 - Codificando para extraer el paralelismo

Crear un profiling

- Es muy importante es hacer el **profiling** con una carga de trabajo realista.
- Hay muchas herramientas para hacer profiling. Una de ellas es gprof de GNU. Pin es muy aconsejable.

```
$ gcc -O2 -g -pg myprog.c
$ gprof ./a.out > profile.txt
Each sample counts as 0.01 seconds.
```

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
33.34	0.02	0.02	7208	0.00	0.00	genTimeStep
16.67	0.03	0.01	240	0.04	0.12	calcStats

Crear un profiling

16.67	0.04	0.01	8	1.25	1.25	
calcSummaryData						
16.67	0.05	0.01	7	1.43	1.43	write
16.67	0.06	0.01				mcount
0.00	0.06	0.00	236	0.00	0.00	tzset
0.00	0.06	0.00	192	0.00	0.00	tolower
0.00	0.06	0.00	47	0.00	0.00	strlen
0.00	0.06	0.00	45	0.00	0.00	strchr
0.00	0.06	0.00	1	0.00	50.00	main
0.00	0.06	0.00	1	0.00	0.00	memcpy
0.00	0.06	0.00	1	0.00	10.11	print
0.00	0.06	0.00	1	0.00	0.00	profil
0.00	0.06	0.00	1	0.00	50.00	report

- genTimeStep() toma 1/3 del tiempo total.
- Es la primera función a paralelizar.
- Entendiendo cómo escala podremos evaluar el beneficio.
- calcStats() También consume bastante tiempo pero lo dejaríamos para siguientes pasadas de APOD

- Assess, Parallelize, Optimize, Deploy (APOD).
- Heterogenous Computing.
 - Differences between Host and Device.
 - ¿Qué podemos ejecutar en una GPU Nvidia?
- **Evaluación de la aplicación (Profiling).**
 - **Profile.**
 - Crear un profile.
 - Identificar puntos críticos.
 - Entender cómo escala el programa
 - Strong Scaling y Ley de Amdahl.
 - Weak Scaling y Ley de Gustafson.
 - Aplicar Weak y Strong Scaling.
- Comenzando a paralelizar.
 - Librerías CUDA.
 - Compiladores paralelizantes.
 - Codificando para extraer el paralelismo

Entender cómo escala

- El beneficio que podemos sacar de CUDA depende de si **es paralelizable** o no la aplicación.
- **¡Alta Prioridad!:** Para obtener el máximo beneficio de CUDA, céntrate primero en encontrar maneras de paralelizar el código secuencial.
- Si entendemos **cómo escala la aplicación** podemos trazar una estrategia de paralelización incremental.
- Cota superior de speed-up a priori:
 - Ley de **Amdhal y escalado fuerte** para un problema de tamaño fijo.
 - Ley de **Gustafson y escalado débil** para un problema incremental.

Strong Scaling y Ley de Amdahl

- **Strong Scaling** es una medida de cómo, para un problema de tamaño fijo, el tiempo de ejecución disminuye cuando se añaden mas procesadores al sistema.
 - **Strong Scaling** lineal (speed-up) tiene un *speedup* igual al número de procesos utilizado.
 - La ecuación de Amdahl formaliza esta idea

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

- Donde P es la fracción de tiempo de ejecución a paralelizar y N número de procesadores utilizados donde corre P
- Cuanto mas grande sea N menor será la fracción de P/N. Si N tiende a infinito la fórmula se convierte a $S=1/(1-P)$
- En el ejemplo anterior tendríamos $S= 1/(1- 3/4) = 4$
- **Idea:** Cuanto mayor sea P mas beneficio tendremos.

Weak Scaling y Ley de Gustafson

- **Weak Scaling** es una medida de cómo el tiempo de la solución cambia cuando mas procesadores son añadidos al sistema para un problema **de tamaño fijo por procesador**.
 - Mas **procesadores tamaño de problema mas grande**.

$$S = N + (1 - P)(1 - N)$$

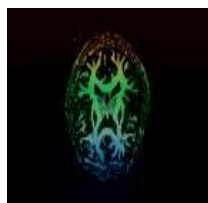
- Donde P es la fracción de tiempo de ejecución a paralelizar y N número de procesadores utilizados donde corre P

Strong Vs Weak Scaling

- Si el tamaño del problema no cambia -> **strong scaling.**
 - Por ejemplo: Simular la interacción de dos moléculas fijas.
- Si el tamaño varía mejor **weak scaling.**
 - Por ejemplo: Montecarlo, MxM, etc...

- Assess, Parallelize, Optimize, Deploy (APOD).
- Heterogenous Computing.
 - Differences between Host and Device.
 - ¿Qué podemos ejecutar en una GPU Nvidia?
- Evaluación de la aplicación (Profiling).
 - Profile.
 - Crear un profile.
 - Identificar puntos críticos.
 - Entender cómo escala el programa
 - Strong Scaling y Ley de Amdahl.
 - Weak Scaling y Ley de Gustafson.
 - Aplicar Weak y Strong Scaling.
- **Comenzando a paralelizar.**
 - **Librerías CUDA.**
 - Compiladores paralelizantes.
 - Codificando para extraer el paralelismo

Desarrollo masivo de aplicaciones desde 2009



146X

**Imágenes
biomédicas**

Univ. Utah

36X

**Dinámica
molecular**

Univ. Illinois,
Urbana

18X

**Transcoding
de vídeo**

Elemental Tech

50X

**Computación
Matlab**

AccelerEyes

100X

Astrofísica

RIKEN



149X

**Simulación
financiera**

Oxford



47X

Algebra lineal

Univ. J aume I



20X

Ultrasonidos 3D

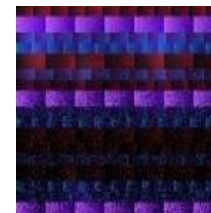
Techniscan



130X

**Química
cuántica**

Univ. Illinois,
Urbana



30X

**Secuenciación
genética**

Univ. Maryland

<http://www.nvidia.com/object/gpu-applications.htm>

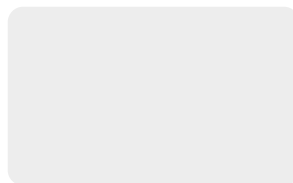
!

Librerías basadas en CUDA

- Muchas librerías desarrolladas dentro y fuera del CUDA toolkit



NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP



Vector Signal
Image Processing



GPU Accelerated
Linear Algebra



Matrix Algebra on
GPU and Multicore



NVIDIA cuFFT



Sparse Linear Algebra

Building-block
Algorithms

C++ Templated
Parallel Algorithms

Matemáticas librerías de CUDA

- Muchas aplicaciones científicas utilizan librerías matemáticas. Muchas de ellas ya están implementadas en CUDA.
 - **cuFFT**: Fast Fourier Transform.
 - **cuBLAS**: BLAS Library (Basic Linear Algebra Subroutines)
 - **cuSPARSE**: Sparse matrix libraries.
 - **cuRAND**: Generating random numbers.
 - **NPP**: image and video processing high performance primitives.
 - **Thrust**: parallel algorithm templated and data structures.
 - **math.h**: floating point C99 library.

Todas ellas incluidas en el cuda toolkit.

Bájalas desde:

<https://developer.nvidia.com/cuda-downloads>

Librerías basadas en CUDA

- **Thrust:** desarrollada por uno de los desarrolladores de CUDA e incluida en la versión 4.
 - Ofrece una abstracción de la programación de los kernel
 - Implementa varios algoritmos paralelos de datos
 - Máximo, suma, Multiplicación de vectores, ...
 - Similar a la librería STL de C++
 - Elige automáticamente el código más rápido en tiempo de compilación
 - Divide el trabajo entre las GPUs y CPUs multi-núcleo
 - Parallel sorting: 5x to 100x faster

Librerías basadas en CUDA

```
int main(void)
{
    // generate random data on the host
    thrust::host_vector<int> h_vec(100);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer to device and compute sum
    thrust::device_vector<int> d_vec = h_vec;
    int x = thrust::reduce(d_vec.begin(), d_vec.end(), 0, thrust::plus<int>());
    return 0;
}
```

Data Structures

- `thrust::device_vector`
- `thrust::host_vector`
- `thrust::device_ptr`
- Etc.

Algorithms

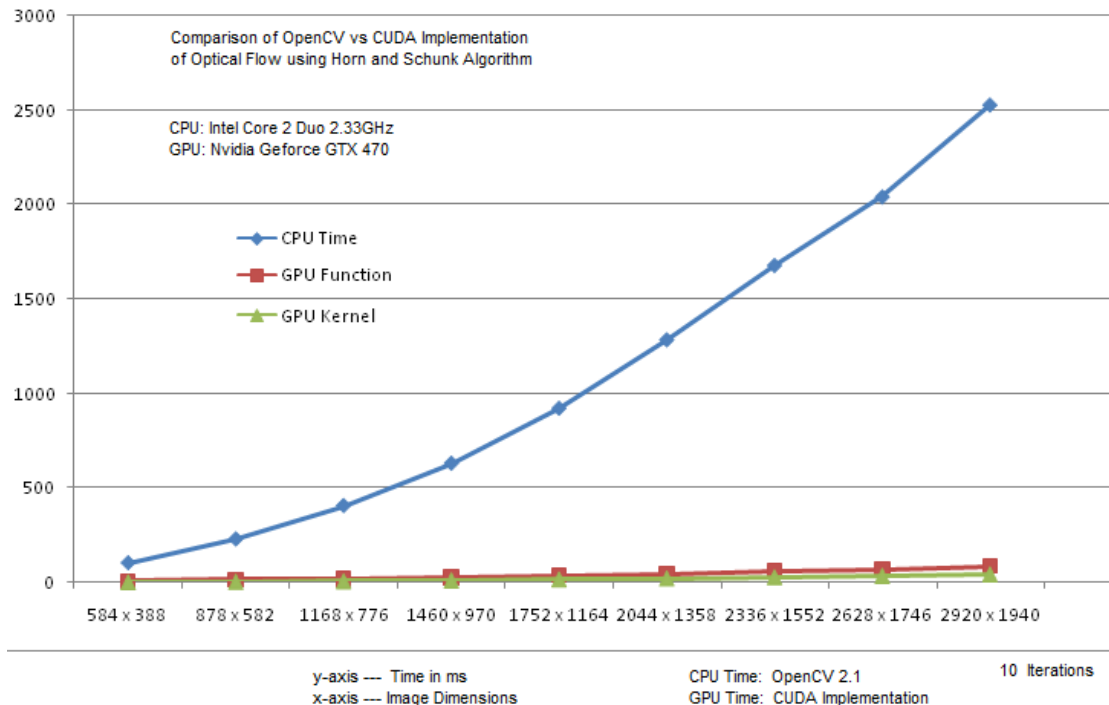
- `thrust::sort`
- `thrust::reduce`
- `thrust::exclusive_scan`
- Etc.



© NVIDIA Corporation 2011

Librerías basadas en CUDA

- **CUVILib**: librería que implementa sobre GPU algoritmos de *visión por computador* y *procesamiento de imágenes*.



Cálculo de flujo de movimiento:

Lucas and Kanade Algorithm


- Assess, Parallelize, Optimize, Deploy (APOD).
- Heterogenous Computing.
 - Differences between Host and Device.
 - ¿Qué podemos ejecutar en una GPU Nvidia?
- Evaluación de la aplicación (Profiling).
 - Profile.
 - Crear un profile.
 - Identificar puntos críticos.
 - Entender cómo escala el programa
 - Strong Scaling y Ley de Amdahl.
 - Weak Scaling y Ley de Gustafson.
 - Aplicar Weak y Strong Scaling.
- **Comenzando a paralelizar.**
 - Librerías CUDA.
 - **Compiladores paralelizantes.**
 - Codificando para extraer el paralelismo

Compiladores paralelizantes

- Para paralelizar códigos secuenciales lo mas sencillo es utilizar **compiladores paralelizantes** -> Usar **enfoques basados en directivas**.
- El programador solo pone unas directivas en el código indicando que esa parte es paralela.
- No hay necesidad de paralelizar el código -> peso en el compilador -> menos rendimiento.
- Ejemplos: OpenACC, OmpSs

OpenACC: A corporative effort for standardization

OpenACC: Open Programming Standard for Parallel Computing




<http://www.openacc-standard.org>

The OpenACC™ API QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.



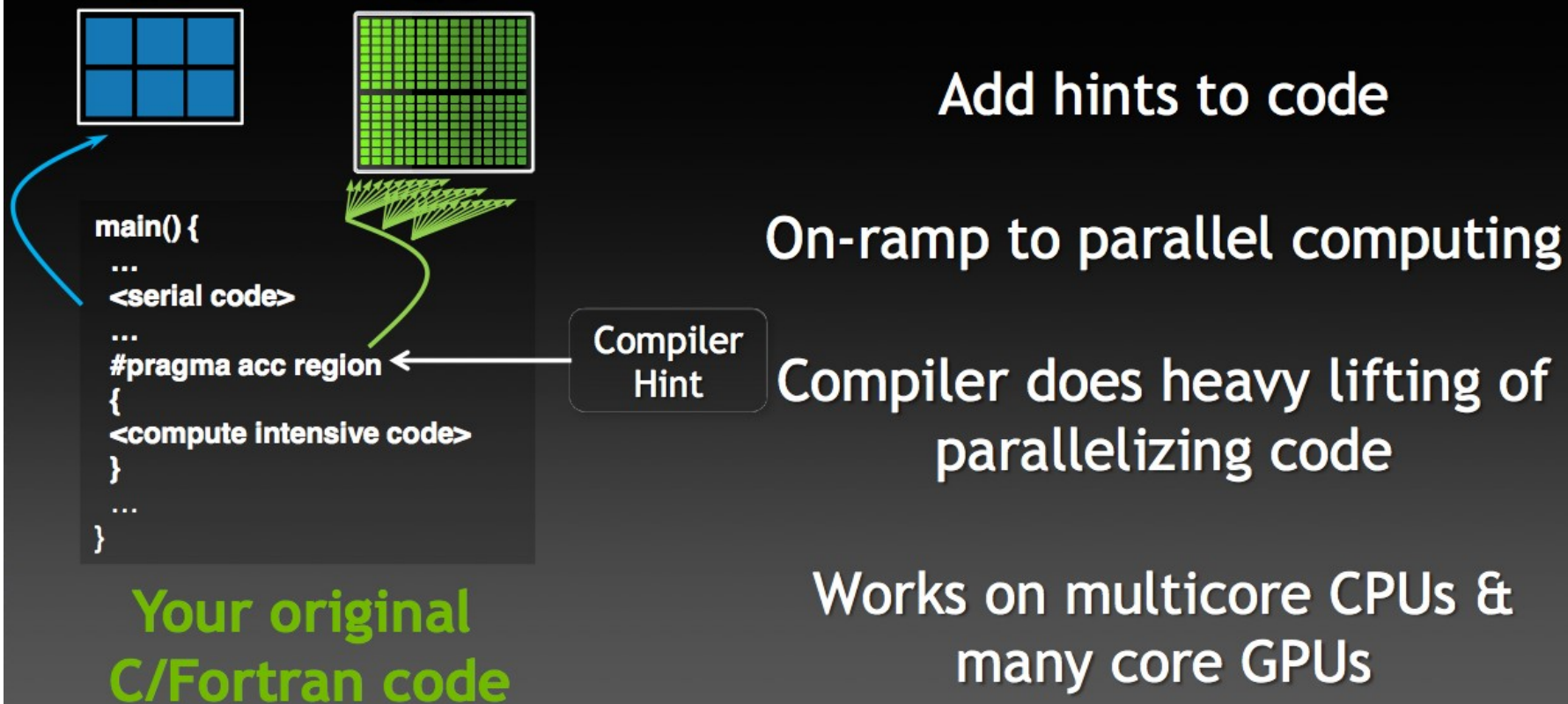
Version 1.0, November 2011

OpenACC: Una alternativa para programadores CUDA de nivel medio

- Las directivas proporcionan una base de código común:
 - Multiplataforma.
 - Multivendedor.
- Esto trae una **manera ideal** de preservar las inversiones en aplicaciones existentes al permitir una **fácil migración** a la computación acelerada.
- Optimización de código con directivas es **muy fácil**, sobre todo en comparación con subprocesos de la CPU o escribir núcleos CUDA.
- Lo más importante es **evitar la reestructuración** del código de salida para aplicaciones de producción.

OpenACC: El modo de utilizar directivas

Directives: Simple Hints for the Compiler



2 pasos sencillo para empezar

- Paso 1: Anotar el código fuente con directivas
 - `!$acc data copy(util1,util2,util3) copyin(ip,scp2,scp2i)`
 - `!$acc parallel loop`
 - `... <source code>`
 - `!$acc end parallel`
 - `!$acc end data`
- Paso 2: Compila y ejecuta
 - `pgf90 -ta=nvidia -Minfo=accel file.f`

Un ejemplo

```
!$acc data copy(A,Anew)
```

Copy arrays into GPU memory
within data region

```
iter=0  
do while ( err > tol .and. iter < iter_max )
```

```
    iter = iter + 1  
    err=0._fp_kind
```

```
!$acc kernels
```

Parallelize code inside region

```
    do j=1,m  
        do i=1,n  
            Anew(i,i) = .25_fp_kind * ( A(i+1,j ) + A(i-1,j ) &  
                                     +A(i ,j-1) + A(i ,j+1))  
            err = max( err, Anew(i,i)-A(i,i))  
        end do  
    end do
```

```
!$acc end kernels
```

Close off parallel region

```
    IF(mod(iter,100)==0 .or. iter == 1)    print *, iter, err  
    A= Anew
```

```
end do
```

```
!$acc end data
```

Close off data region,
copy data back

La pregunta clave es: ¿Cuánto rendimiento perdemos?

- Algunos resultados dicen que solo 5-10% Vs CUDA en algunos casos. Otras fuentes dicen que obtienen 5x de ganancias solo invirtiendo una semana o incluso un día.
- Pero este factor es mas dependiente de la aplicación que por las habilidades del programador

Real-time object detection

Global Manufacturer of Navigation Systems



5x in 1 week

Valuation of stock portfolios using Montecarlo

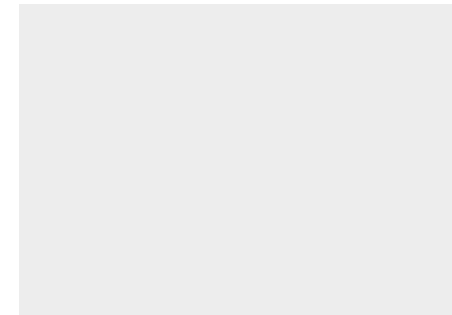
Global Technology Consulting Company



2x in 4 hours

Interaction of solvents and biomolecules

University of Texas at San Antonio



5x in 1 day

¿Por qué no empezamos?

Propuesta de mejora de la última práctica:
Implementa Jacobi 2D con OpenACC y comprará
con tus versiones CUDA.

- Assess, Parallelize, Optimize, Deploy (APOD).
- Heterogenous Computing.
 - Differences between Host and Device.
 - ¿Qué podemos ejecutar en una GPU Nvidia?
- Evaluación de la aplicación (Profiling).
 - Profile.
 - Crear un profile.
 - Identificar puntos críticos.
 - Entender cómo escala el programa
 - Strong Scaling y Ley de Amdahl.
 - Weak Scaling y Ley de Gustafson.
 - Aplicar Weak y Strong Scaling.
- **Comenzando a paralelizar.**
 - Librerías CUDA.
 - Compiladores paralelizantes.
 - **Codificando para extraer el paralelismo**

Expresando paralelismo

- Para aplicaciones que necesitan **funcionalidad adicional o mas rendimiento** que lo que brindan las librerías -> CUDA C/C++
- Una vez identificado el código a acelerar -> Implementar kernel de CUDA e integrar en el resto de la aplicación.
- Ideal para aplicaciones con tiempos de ejecución localizados en una porción de código.
- Mas difícil cuando tiene un perfil plano.

¿Preguntas?

Multi-GPU

“The only thing better than computing on a GPU is computing on two GPUs.”

Conexión de varias GPUs al bus PCI-express

Posibilidad de aprovechar distintos device en un mismo computador

Gestión muy fácil de varios devices desde la aparición de CUDA 4.0

```
cudaGetDeviceCount()
```

```
cudaSetDevice(id context)
```


Multi-GPU

- Utilización de hilos CPU para gestionar las distintas GPUs
 - Válida cualquier implementación:
 - Pthreads
 - CUDThreads
 - OpenMPI
 -
 - Desde CUDA 4.0 es posible intercambiar contextos (devices GPUs)
 - Dos hilos pueden tener el mismo contexto, pero competirán por los recursos.

Multi-GPU

- Para el aprovechamiento de las múltiples GPUs:
 - Particionar el problema (a veces es muy complicado)
 - Resolver problemas distintos en paralelo
 - Cualquier lógica que se nos ocurra que pueda mejorar el rendimiento del algoritmo
- Posibilidad de poner devices en modo de computación exclusiva
 - Estos devices no serán compartidas por otros hilos CPU

Contenido

- Hardware
- Librerías
- Multi-GPU
- **CUDA 4.0**
- Análisis y depuración
- OpenCL
- PGI x86

CUDA 4.0

Cada nueva versión del toolkit de NVIDIA nos sorprende con nuevas funcionalidades y mejoras:

CUDA 4.0 supuso un gran avance en la programación de las GPUs

- Facilidad portabilidad aplicaciones hacia la GPU

- Capacidad compartir GPUs entre distintos hilos

- Un único hilo puede acceder a todas las GPUs

- Nuevas características C/C++

- Primitivas implementadas en thrust y soportadas por NVIDIA

- NPP librería procesamiento de imagen y vídeo

CUDA 4.0

NVIDIA GPUDirect v2.0

- Acceso punto a punto entre GPUs

- Transferencias punto a punto entre GPUs

- Memoria Unificada

Mejoras en las herramientas de desarrollo

- Herramientas de análisis automática de rendimiento

- Depuración C++

- Desensamblador binario GPU

Futuro: CUDA 5.0

Vendrá acompañado de la nueva arquitectura Kepler

Más rendimiento

Los días 16-17 de Mayo se presentarán las nuevas características del nuevo SDK ofrecido por NVIDIA:

CUDA 5 and the Kepler GPU architecture don't just increase application performance; they enable a more powerful parallel programming model that expands the possibilities of GPU computing, and language features that improve programmer productivity.

Análisis y depuración

Existen diversas herramientas para depurar y analizar código CUDA:

Disponibles en el sitio web de NVIDIA

Depuración

cugdb

Windows: podemos utilizar a través de Visual Studio.

Parallel Nsight: plugin depuración visual studio

Permite ver estado memoria GPU en ejecución

CUDA Memory Checking

CUDA Graphics debugging

Visual profiling

Contenido

- Hardware
- Librerías
- Multi-GPU
- CUDA 4.0
- **Análisis y depuración**
- OpenCL
- PGI x86

Contenido

- Hardware
- Librerías
- Multi-GPU
- CUDA 4.0
- Análisis y depuración
- **OpenCL**
- PGI x86

OpenCL

- Plataforma cruzada de computación paralela sobre dispositivos heterogeneos
- Aseguran la correcta ejecución pero no el rendimiento sobre distintos dispositivos
- Existen implementaciones de OpenCL sobre para NVIDIA, AMD GPUs, x86 CPUs, Power CPUs
- Compatibilidad sobre múltiples dispositivos de distintos fabricantes

OpenCL

Relación términos CUDA - OpenCL

OpenCL Parallelism Concept	CUDA Equivalent
kernel	kernel
host program	host program
NDRange (index space)	grid
work item	thread
work group	block

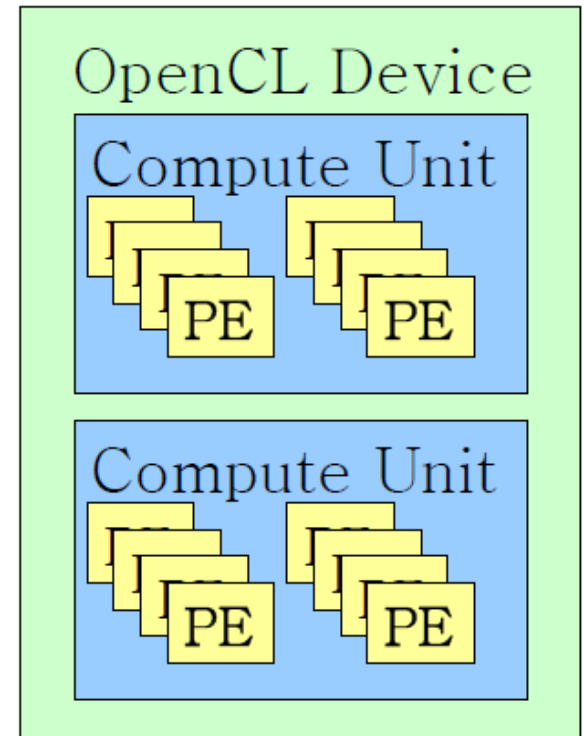
OpenCL

OpenCL Hardware abstraction

Trata CPUs multi-núcleo, GPUs,
y otros aceleradores como devices

Cada device contiene una o más
'Computing Units'. (SMs)

Cada 'Computing Unit' contiene uno
o más "processing elements" SIMD



OpenCL

Relación memorias OpenCL - CUDA

OpenCL Memory Types	CUDA Equivalent
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	registers

OpenCL

El código que se ejecuta en los dispositivos aceleradores es análogo a los kernels de CUDA.

```
__kernel void vadd(__global const float *a,  
    __global const float *b, __global float  
    *result){  
    int id = get_global_id(0);  
    result[id] = a[id] + b[id];  
}
```

Al igual que en CUDA cada unidad de trabajo de Open CL obtiene sus propios índices.

OpenCL

Relación dimensiones e índices OpenCL - CUDA

OpenCL API Call	Explanation	CUDA Equivalent
<code>get_global_id(0);</code>	global index of the work item in the x dimension	<code>blockIdx.x * blockDim.x + threadIdx.x</code>
<code>get_local_id(0)</code>	local index of the work item within the work group in the x dimension	<code>threadIdx.x</code>
<code>get_global_size(0);</code>	size of NDRange in the x dimension	<code>gridDim.x * blockDim.x</code>
<code>get_local_size(0);</code>	Size of each work group in the x dimension	<code>blockDim.x</code>

OpenCL

- Conclusiones OpenCL
 - Intrínsecamente evoluciona más lento que CUDA
 - Aunque OpenCL es correcto en términos de ejecución, el rendimiento de un kernel no está garantizado sobre diferentes dispositivos
 - La programación de aplicaciones en OpenCL requiere introducir más comprobaciones que en CUDA Runtime API
 - En un futuro debería convertirse en el estándar para la programación de GPUs.

Contenido

- Hardware
- Librerías
- Multi-GPU
- CUDA 4.0
- Análisis y depuración
- OpenCL
- **PGI x86**

PGI x86

CUDA para aplicaciones GPU y CPU

Herramienta para construir de forma transparente aplicaciones CUDA en sistemas GPU y no GPU creados por cualquier fabricante

PGI x86 Compiler

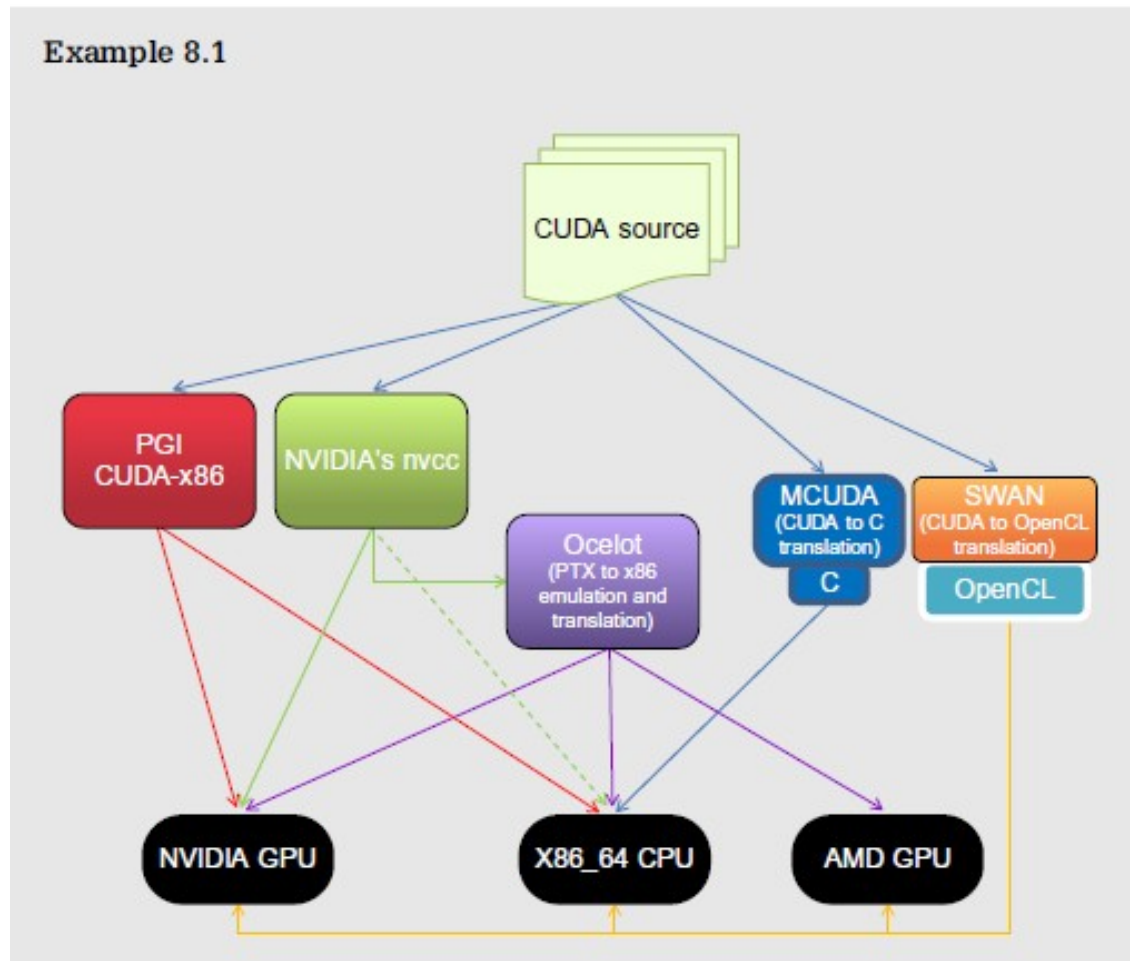
Creación de binarios unificada

Aprovechamiento de los recursos disponibles

Programación mediante anotaciones en el código (estilo OpenMPI)

PGI x86

Example 8.1



Aplicaciones en CUDA

En la pagina de www.nvidia.com podemos encontrar una larga lista de aplicaciones específicas y modificaciones de programas para aprovechar esta tecnología, para diferentes ámbitos de aplicación:

Gobierno y defensa

Dinámica molecular

Bioinformática

Electrodinámica y electromagnetismo

Imágenes medicas

Combustibles y gases

Computación financiera

Extensiones de Matlab

Tratamiento de video y visión por computador

Modelado del tiempo y los océanos

Aplicaciones en CUDA

Actualmente estamos trabajando en:

Visión:

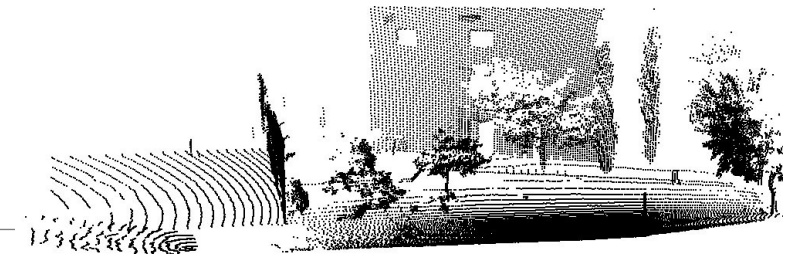
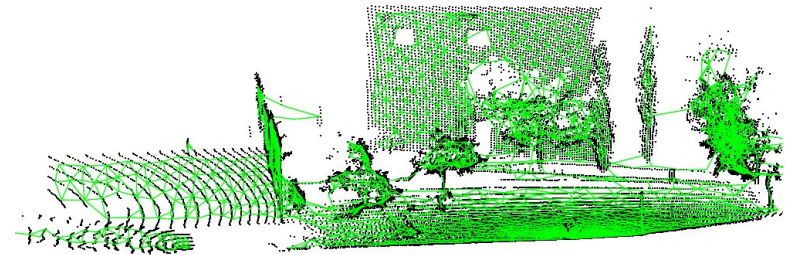
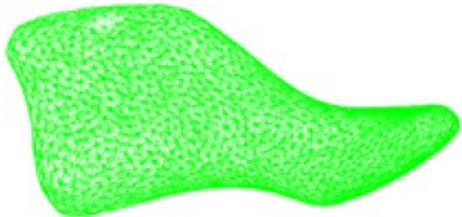
Aceleración algoritmos redes neuronales
auto-organizativas (GNG,NG)

Reconstrucción escenarios 3D

Representación 3D objetos

Algoritmos correspondencia 3D

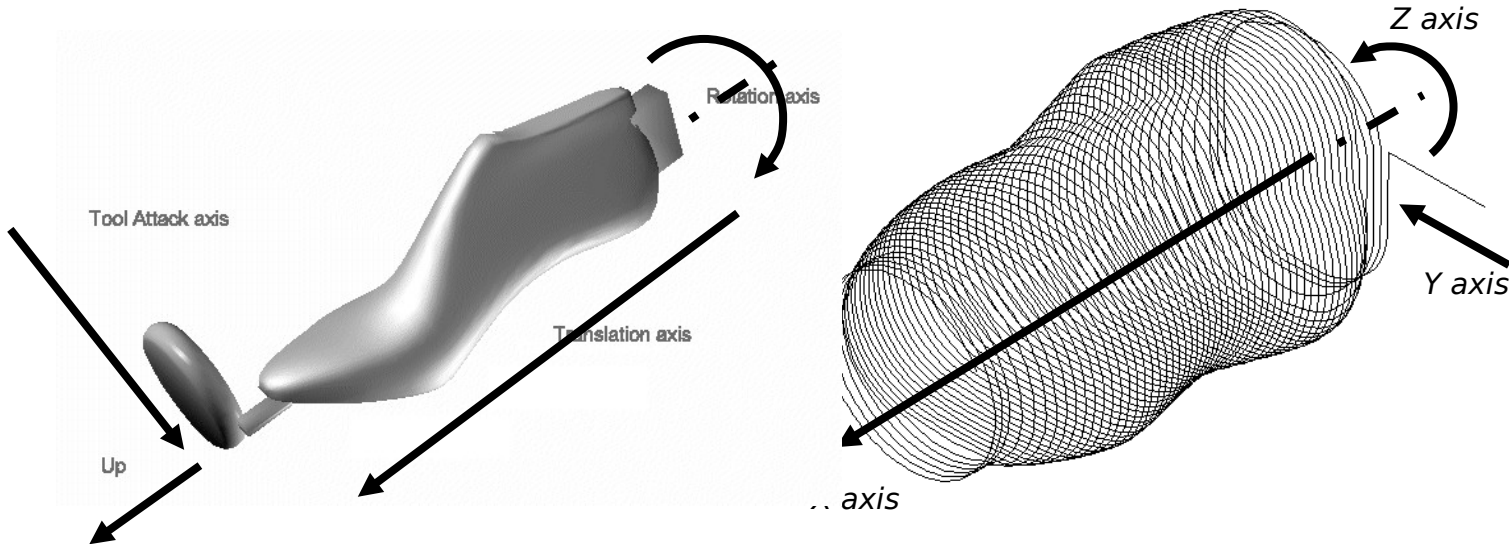
Iterative Closest Point



Aplicaciones en CUDA

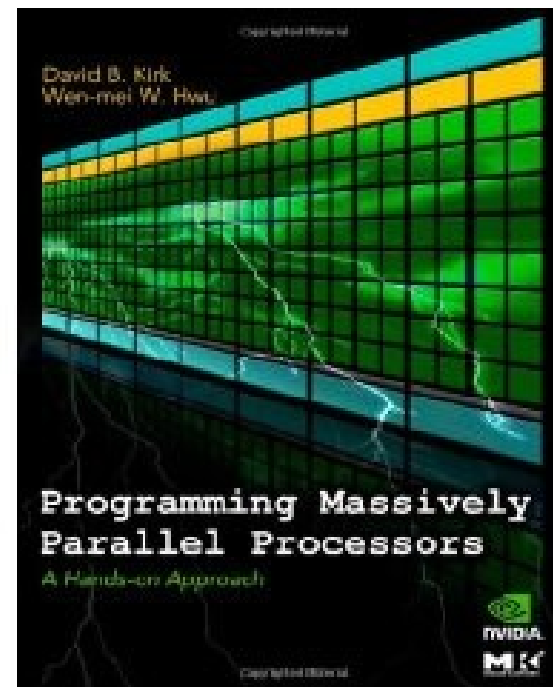
Hemos trabajado en:

CAD/CAM: Calculo de rutas de maquinas-herramienta para recrear modelos digitales



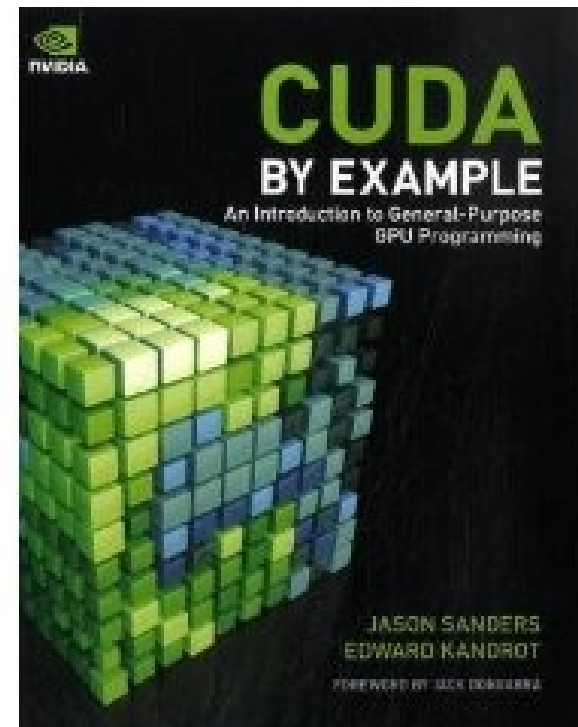
Bibliografía

- Programming Massively Parallel Processors: A Hands-on Approach
 - By David Kirk and Wen-mei Hwu
- Libro indispensable para aprender C
- Escrito por gente de NVIDIA
- Conceptos básicos y avanzados de (



Bibliografía

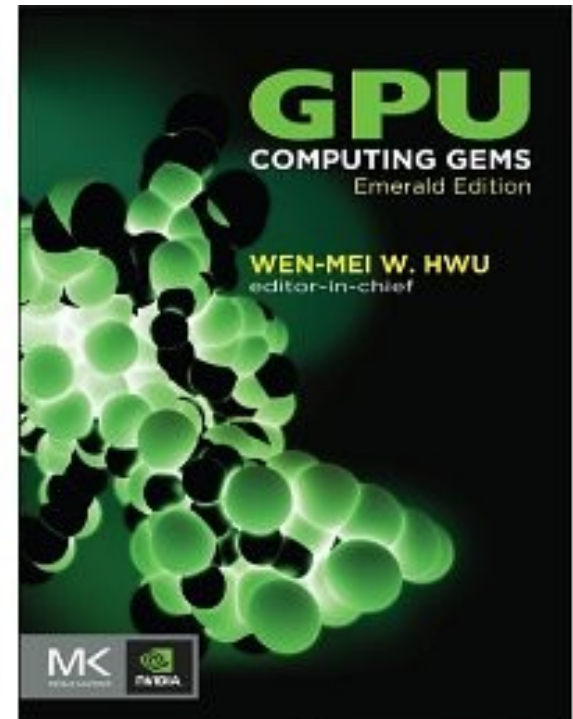
- CUDA by Example: An Introduction to General-Purpose GPU Programming
 - Jason Sanders, Edward Kandrot
- Aprendizaje basado en ejemplo



Bibliografía

□ GPU Computing GEMS

- Wen-mei W. Hwu
- Recopilación de los mejores artículos sobre la tecnología CUDA.
- Aplicaciones sobre distintos campos
 - Física
 - Visión artificial
 - Librerías
 - etcétera



Edición Kindle

Programación Paralela.

¿Preguntas?



Análisis y depuración

Análisis

Compute Visual Profiler

No requiere compilar de una forma especial

Ejecuta el programa repetidas veces para obtener información:

- Tiempo consumido en transferencias device - host

- Tiempo empleado por cada kernel – análisis detallado

- Aprovechamiento de los recursos

- Tasas de fallos/aciertos memorias cache

- Consejos sobre posibles fallos y estrategias a seguir para mejorar el rendimiento de la aplicación

Análisis y depuración

Análisis

Compute Visual Profiler - Ejemplo

Profiler Output					Summary Table	
	Method	#Calls	GPU time (us)	%GPU time	IPC	active warps
1	reduction	11	53651,3	13,4	1,67771	46,9825
2	reduction2	10	27188,9	6,79	1,33395	46,8953
3	reduction3	10	19808,8	4,95	1,40493	46,3148
4	reduction4	3	3111,81	0,77	1,42053	45,8463
5	memcpyHtoD	35	212311	53,06		
6	memcpyDtoH	34	228,064	0,05		

Análisis y depuración

Análisis

Compute Visual Profiler – Ejemplo: cuReduction

Analysis for kernel reduction4 on device GeForce GTX 480

Summary profiling information for the kernel:

Number of calls: 3
Minimum GPU time(us): 1036.77
Maximum GPU time(us): 1038.02
Average GPU time(us): 1037.27
GPU time (%): 0.77
Grid size: [8192 1 1]
Block size: [256 1 1]

Limiting Factor

Achieved Instruction Per Byte Ratio: 29.01 (Balanced Instruction Per Byte Ratio: 3.79)
Achieved Occupancy: 0.96 (Theoretical Occupancy: 1.00)
IPC: 1.42 (Maximum IPC: 2)
Achieved global memory throughput: 16.86 (Peak global memory throughput(GB/s): 177.41)

Limiting Factor
Identification

Memory Throughput
Analysis

Instruction Throughput
Analysis

Occupancy Analysis

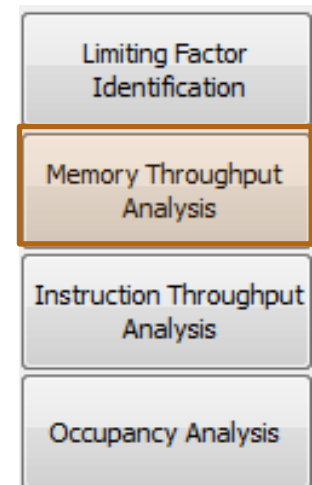
Análisis y depuración

Análisis

Compute Visual Profiler – Ejemplo: cuReduction

Memory Throughput Analysis for kernel reduction4 on device GeForce GTX 480

- Kernel requested global memory read throughput(GB/s): 24.23
- Kernel requested global memory write throughput(GB/s): 16.17
- Kernel requested global memory throughput(GB/s): 40.40
- L1 cache read throughput(GB/s): 16.17
- L1 cache global hit ratio (%): 0.00
- Texture cache memory throughput(GB/s): 0.00
- Texture cache hit rate(%): 0.00
- L2 cache texture memory read throughput(GB/s): 0.00
- L2 cache global memory read throughput(GB/s): 16.17
- L2 cache global memory write throughput(GB/s): 0.25
- L2 cache global memory throughput(GB/s): 16.43
- Local memory bus traffic(%): 0.00
- Global memory excess load(%): -49.80
- Global memory excess store(%): -6300.00
- Achieved global memory read throughput(GB/s): 16.70
- Achieved global memory write throughput(GB/s): 0.17
- Achieved global memory throughput(GB/s): 16.86
- Peak global memory throughput(GB/s): 177.41



Análisis y depuración

Análisis

Compute Visual Profiler – Ejemplo: cuReduction

Instruction Throughput Analysis for kernel reduction4 on device GeForce GTX 480

- IPC: 1.42
- Maximum IPC: 2
- Divergent branches(%): 3.15
- Control flow divergence(%): 5.51
- Replayed Instructions(%): -0.17
 - Global memory replay(%): 0.85
 - Local memory replays(%): 0.00
 - Shared bank conflict replay(%): 0.00
- Shared memory bank conflict per shared memory instruction(%): 0.00

Hint(s)

- **The kernel is compute bound**, to reduce instruction count
 - Understand the instruction mix, as single precision floating point, double precision floating point, in
 - Try using arithmetic intrinsic functions.
 - Try using compiler flags(-ftz=true, -prec-div=false, -prec-sqrt=false etc) to get higher performancRefer to the "Arithmetic Instructions" section in the "Performance Guidelines" chapter of the CUDA C Progr
- **The control flow is divergent**, try
 - Modifying the kernel so that threads within the same warp take the same path;
 - Rearranging the data or pre-processing the data;
 - Rearranging how threads index data (may affect memory performance);Refer to the "Branching and Divergence" section in CUDA C Best Practices Guide for more details.

Limiting Factor
Identification

Memory Throughput
Analysis

Instruction Throughput
Analysis

Occupancy Analysis

Análisis y depuración

Análisis

Compute Visual Profiler – Ejemplo: cuReduction

Occupancy Analysis for kernel reduction4 on device GeForce GTX 480

- Kernel details: Grid size: [8192 1 1], Block size: [256 1 1]
- Register Ratio: 0.375 (12288 / 32768) [7 registers per thread]
- Shared Memory Ratio: 0.125 (6144 / 49152) [1024 bytes per Block]
- Active Blocks per SM: 6 (Maximum Active Blocks per SM: 8)
- Active threads per SM: 1536 (Maximum Active threads per SM: 1536)
- Potential Occupancy: 1 (48 / 48)
- Occupancy limiting factor: None

Note: The potential occupancy is calculated assuming the default cache configuration i.e. 48KB of shared memory.

Limiting Factor
Identification

Memory Throughput
Analysis

Instruction Throughput
Analysis

Occupancy Analysis