Escuela de Computación de Alto Rendimiento 2014

Optimización de aplicaciones GPU

Carlos Bederián
IFEG – CONICET
GPGPU Computing Group – FaMAF – UNC

CONICET

NVIDIA CUDA TEACHING CENTER

NVIDIA CUDA RESEARCH CENTER

UNC Universidad Nacional de Córdoba

Advertencia

- “Premature optimization is the root of all evil”

  -Donald Knuth
- Las siguientes son herramientas a usar
  - No siempre son necesarias
    - Usar el profiler
  - No siempre mejoran la performance

- **¡USAR EL PROFILER!**

Algoritmos paralelos

- Independientes
  - Map
  - Scatter / Gather
    - Aprovechar localidad
- Dependencias entre elementos
  - Reduce
  - Scan
  - Filter
  - Sort
  - ...

Tesla K40

- 2880 cores* a 745-875 MHz
  - 1 FMA por ciclo ~= 2 operaciones
  - 4290-5040 GFLOPS* precisión simple
  - 1430-1680 GFLOPS* precisión doble
- Memoria GDDR5 a 6004 MT/s, bus de 384 bits
  - 288 GBps

Cores* - Ejecución SIMT

- Grupos de 32 cores*
  - 32 hilos (un “warp”) ejecutan la misma instrucción simultáneamente
  - ¡Es una unidad de vectores!
  - Máscara de hilos inactivos
  - Serialización/replay de instrucciones
- Los bloques de hilos se dividen en warps
  - Los warps ejecutan independientemente

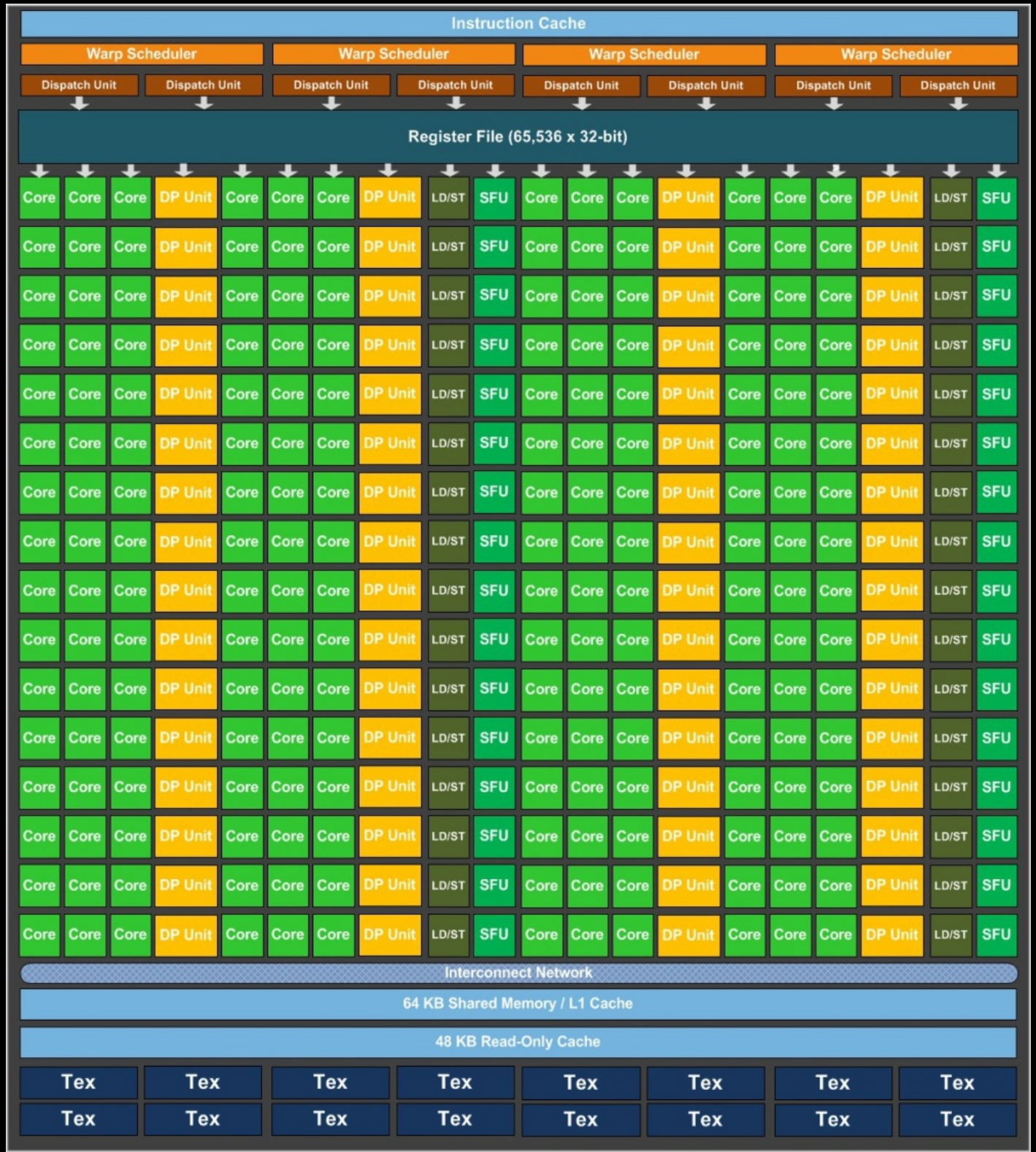
Map - Haciendo números

- Aplicando una función unaria f a los elementos de una matriz: B = map f A
- Si A contiene doubles (64 bits / 8 bytes):
  - Leo+escribo 288GBps /(8B+8B) = 18G doubles/s
  - 1430 GFLOPS / 18G floats/s = ~79 operaciones
- Si A contiene floats (32 bits / 4 bytes):
  - Leo+escribo 288GBps /(4B+4B) = 36G floats/s
  - 4290 GFLOPS / 36G doubles/s = ~119 operaciones
- ¡Memory bound!

Xeon E5-2699 v3

- 18 cores Haswell-E a 1.9-2.3 GHz (clock AVX)
  - Unidades de vectores de 256 bits
  - 2 FMA por ciclo
  - 547-662 GFLOPS* precisión doble
  - 1094-1325 GFLOPS* precisión simple
- Memoria DDR4 a 2133 MT/s, quad channel (256 bits)
  - 68 GBps
- ~129 operaciones por elemento
- 45MB de cache L3

Mapa de una GPU moderna


Instruction Cache
Warp Scheduler
Warp Scheduler
Warp Scheduler
Warp Scheduler
Dispatch Unit
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Cache
Tex

Jerarquía de memoria

	Fermi GF110 CC 2.0	Kepler GK110 CC 3.5	Maxwell GM204 CC 5.2
Registros	128 KB	256 KB	256 KB
Constant cache	8 KB	8 KB	10 KB
Shared memory	64 KB	64 KB	2 x 48 KB
L1 cache			48 KB
Texture cache	12 KB	48 KB	
Read-only cache	-		
L2 cache	768 KB	1536 KB	2048 KB
Total	4160 KB	7176 KB	8198 KB

Cache L2

- Todas las lecturas y escrituras pasan por L2
- No persiste entre llamadas a kernels
- Líneas de 128 bytes
- Para localidad espacial

Pitched memory

- Queremos que cada warp lea líneas completas de cache
- Pedimos memoria con padding por fila óptimo

```
// Ejemplo: Crear matriz de 53x32 float
// 53*4 = 212 bytes por fila
cudaMalloc(&m, 53 * 32 * sizeof(float));

// Que CUDA decida y lo ponga en pitch
size_t pitch;
cudaMallocPitch(&m, &pitch, 53 * sizeof(float), 32);

// pitch probablemente vale 256
size_t stride = pitch / sizeof(float);
```

Usando pitched memory

- Row stride: distancia entre dos elementos de la misma columna en filas consecutivas
  - Cuando hay padding, deja de ser igual al ancho de fila
  - Pitch: Row stride en bytes (`stride * sizeof(tipo)`)
- Funciones `cudaMemcpy2D` y `cudaMemset2D` para escribir a arreglos con stride
-
```
cudaMemcpy2D(destino, pitch_destino,
             origen, pitch_origen,
             ancho_en_bytes,          // sin padding
             alto, cudaMemcpyDefault);
```

Read-only cache

- CC 3.5 en adelante
- Sólo lectura
- No es consistente con lo escrito por el kernel
- Granularidad de 32 bytes
- Dos formas de utilizarlo:
  - Parámetros de kernel `const __restrict__ *`
  - Instrucción `__ldg()`

Local memory

- Espacio en memoria global dedicado a cada hilo
  - ¡Evitar su uso en lo posible!
- Cacheado en L1
- Uso automático
  - Spilling de registros
  - Arreglos con índices generados en runtime

```
__global__ void k(int *idx) {
  int A[10];
  A[idx[threadIdx.x]] = 3;
  …
}
```

```
__global__ void k() {
  int A[10];
  #pragma unroll
  for(int i=0; i<10; ++i)
    A[i] = 3;
  …
}
```

Constant cache

- Sólo lectura dentro de kernels
- Cache sobre 64 KB de memoria global
- Working set de 8 / 10 KB por SM
- Broadcast de 4 bytes a un warp en un ciclo
  - ¡Asegurarse de que todo un warp lea lo mismo!
- Usos implícitos:
  - Parámetros de los kernels
  - Vtables (C++)

Constant cache - Uso

- Declarar símbolos `__constant__` visibles globalmente:

```
__constant__ float alpha;
```

- Copiar datos con `cudaMemcpyToSymbol()`:

```
float a = 2.0f;
cudaMemcpyToSymbol(alpha, &a, sizeof(a));
```

- Usar el nombre del símbolo en kernels

Texturas

- Hardware de gráficos 3D expuesto en CUDA
  - Cache de menor granularidad
    - Optimizado para localidad espacial 2D
  - Generación de coordenadas
  - Interpolación lineal
- Dos interfaces en CUDA
  - Texture references
  - Texture objects (Kepler en adelante)

Texturas - Requisitos

- Datos en algún formato soportado
- `cudaResourceDesc` – Descripción de los datos
  - Puntero a los datos
  - `cudaChannelDesc` - Tamaño por elemento
  - Ancho, alto, row stride (alineado a 2^N)
- `cudaTextureDesc` – Cómo queremos muestrear la textura
  - Normalización de coordenadas
  - Condiciones de contorno
  - Interpolación
  - Normalización de muestras

Channel Descriptor

- Los elementos pueden ser hasta 4-uplas:
  - Cantidad de bits arbitraria por componente
  - Todas del mismo tipo: signed, unsigned o float
- Para tipos conocidos:

```
cudaChannelDesc cd =
                    cudaCreateChannelDesc<float>();
```

- Para definir por componente:
  - `cudaChannelDesc cd = cudaCreateChannelDesc(`
    `32, 0, 0, 0, cudaChannelFormatKindFloat);`

Resource Descriptor

- Descripción del almacenamiento de los elementos

```
cudaMallocPitch(&m,&pitch,53*sizeof(float),32);

cudaResourceDesc rd;
memset(&rd, 0, sizeof(rd));
// Ver un arreglo con pitch como textura 2D
rd.resType = cudaResourceTypePitch2D;
rd.res.pitch2D.desc   = cd;   // channel desc.
rd.res.pitch2D.devPtr = m;    // puntero a datos
rd.res.pitch2D.width  = 53;
rd.res.pitch2D.height = 32;
rd.res.pitch2D.pitchInBytes = pitch;
```

Texture desc. - Coordenadas

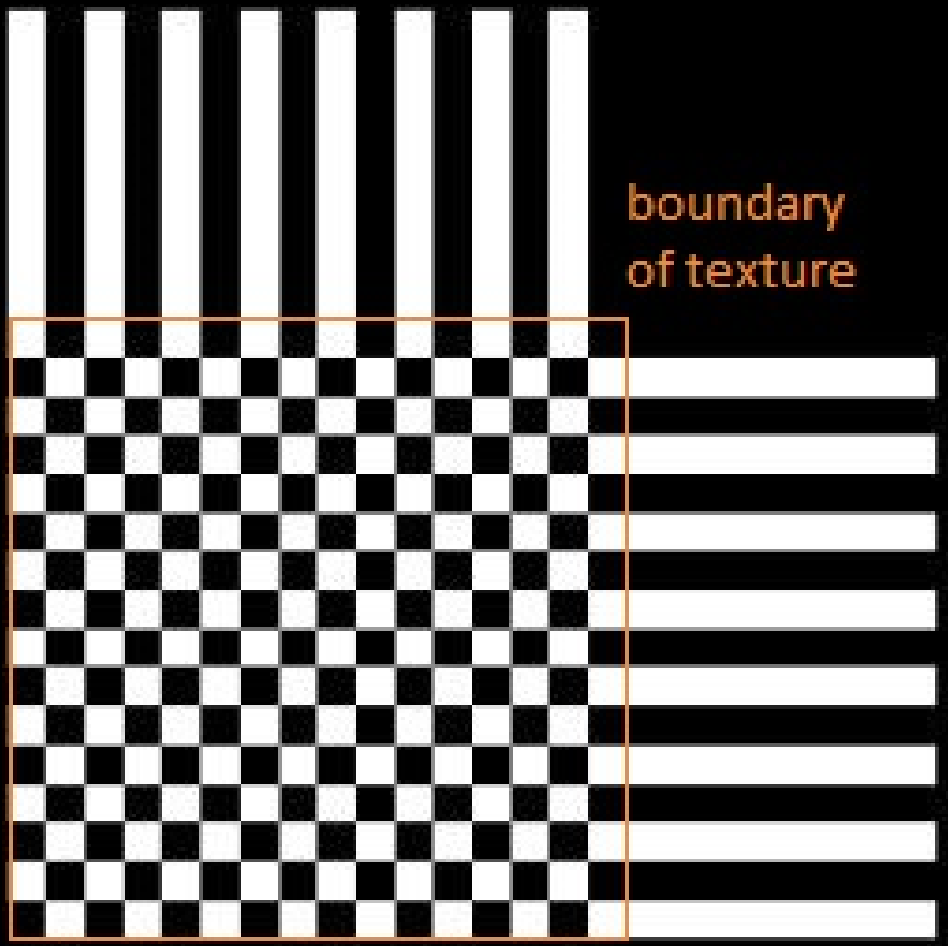
- Dos modos principales:
  - Coordenadas en [0, tam]
    - `td.normalizedCoords = 0;`
    - ¡Notar el elemento extra!
    - Centro de cada elemento: `coordenada + 0.5f`
  - Coordenadas en [0, 1]
    - `td.normalizedCoords = 1;`

Texturas - Bordes

- Campo `addressMode[3]` de `cudaTextureDesc`
  - Modo independiente por coordenada
- ¡Condiciones de contorno gratis!

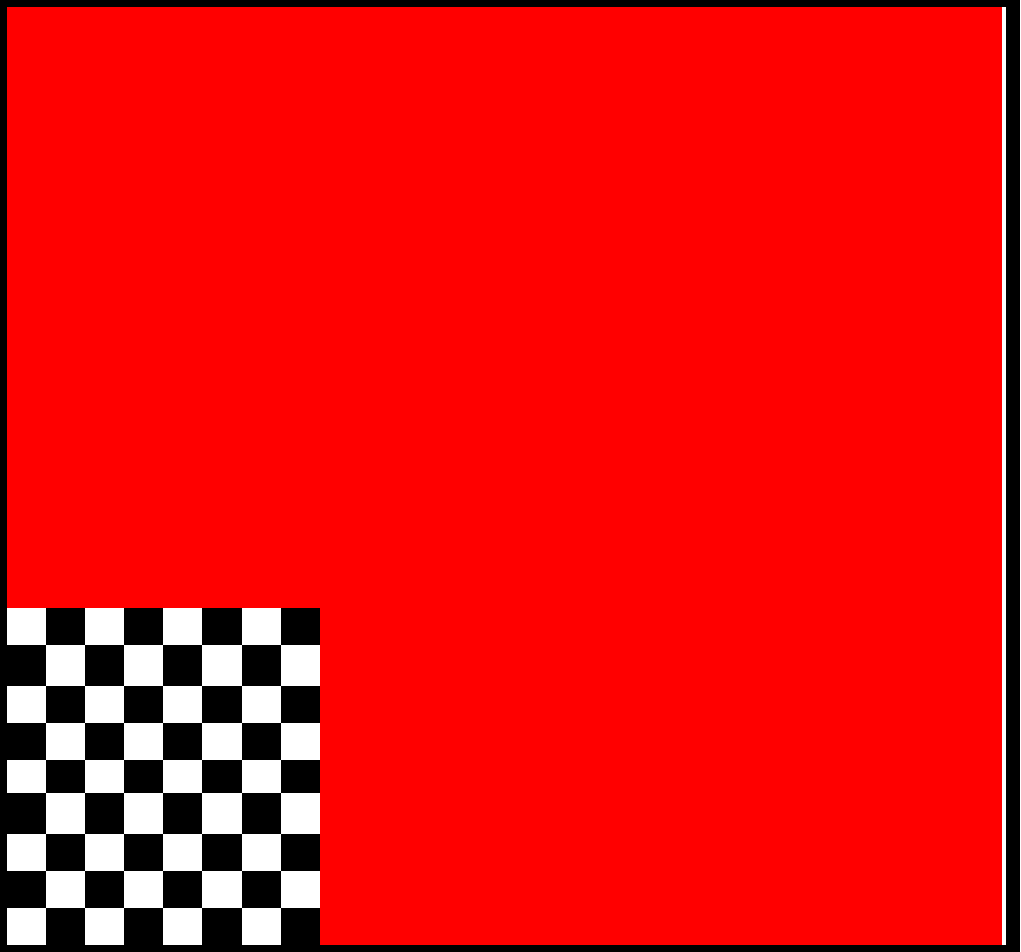
Texturas - Clamp

- Modo `cudaAddressModeClamp`
  - Si se excede lee del borde



Texturas - Border

- Modo `cudaAddressModeBorder`
  - Si se excede lee 0

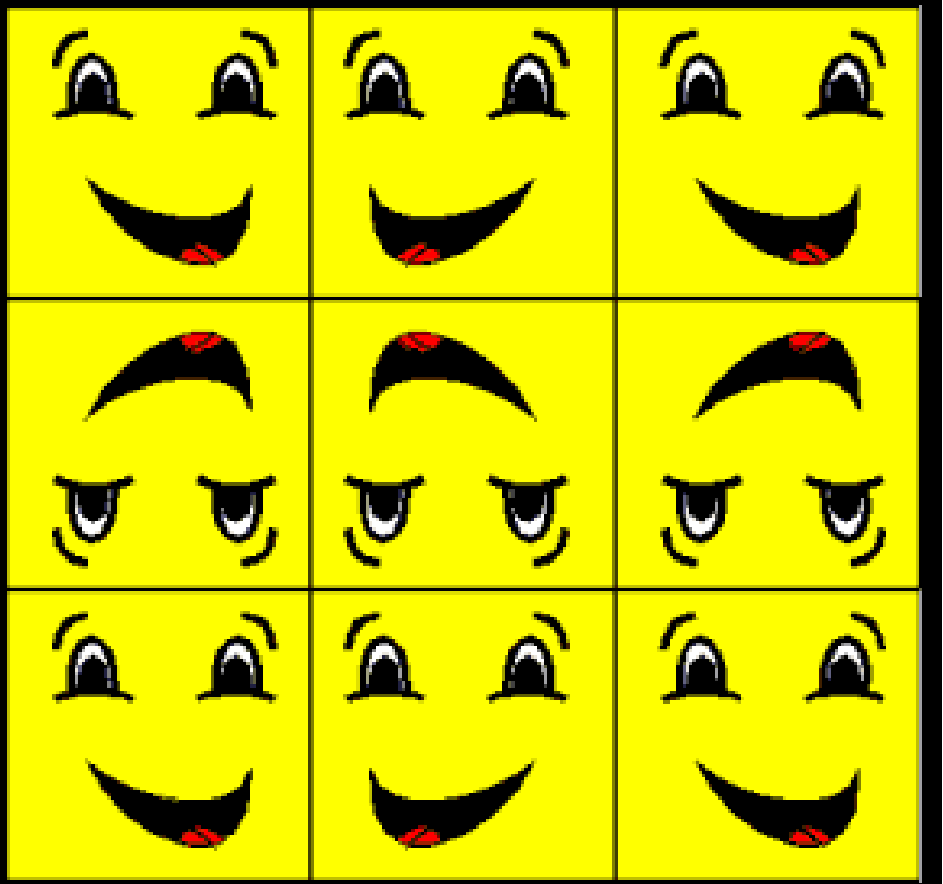


Texturas - Wrap

- Modo `cudaAddressModeWrap`
  - Requiere coordenadas normalizadas
  - Si se excede, empieza de nuevo

Texturas - Mirror

- Modo `cudaAddressModeMirror`
  - Requiere coordenadas normalizadas
  - Si se excede, espeja el original

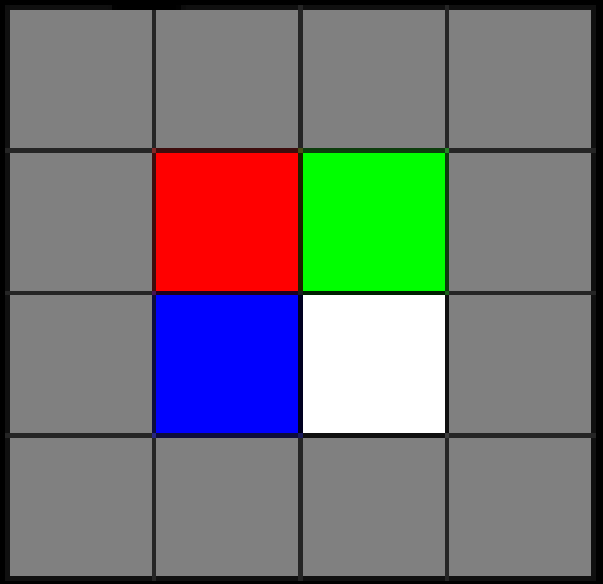


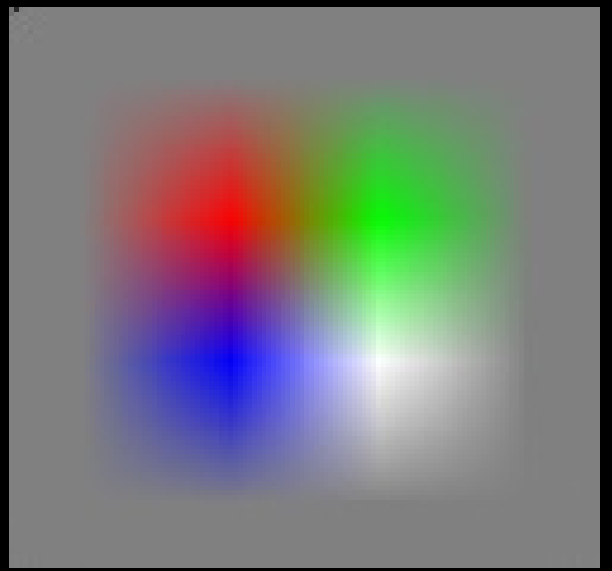
Texturas - Lectura

- `td.readMode` – Modo de lectura
  - `cudaReadModeElementType`: mismo tipo
  - `cudaReadModeNormalizedFloat`: conversión a [0,1]
    - Enteros: sólo de 8 o 16 bits

Texturas - Interpolación

- `td.filterMode` - Interpolación
  - `cudaFilterModePoint`: sin interpolación
  - `cudaFilterModeLinear`: interpolación (bi)lineal
    - El valor está en el centro del pixel, el resto se interpola
    - Sólo con `cudaReadModeNormalizedFloat`





Texturas - Uso

```
// creo cudaResourceDesc y cudaTextureDesc antes

cudaTextureObject_t tex;
cudaCreateTextureObject(&tex, rd, td, 0);
kernel<<<grid, block>>>(tex);


__global__ void kernel(cudaTextureObject_t t)
{
  …
  T sample = tex2D<T>(t, x, y);
  …
}
```

Shared memory

- Sólo accesible dentro de kernels
- Visible a todo un bloque
  - Hasta 48 KB por bloque
  - Método principal de comunicación entre hilos
- Memoria dividida en 32 bancos de 32/64 bits
- ~32 bits por banco por ciclo
  - ¡>1 TBps!
- Dos modos de acceso:
  - Broadcast a un warp
  - Acceso simultaneo a bancos distintos

Banked access (8 bancos)

Banco	0	1	2	3	4	5	6	7
Elementos	0	1	2	3	4	5	6	7
	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23
	24	25	26	27	28	29	30	31

Transpuesta en shared

```
__shared__ int buf[8][8];
buf[threadIdx.y][threadIdx.x] = src[sy][sx];
```

0	1	2	3	4	5	6	7	Warp 0
0	1	2	3	4	5	6	7	
0	1	2	3	4	5	6	7	
0	1	2	3	4	5	6	7	
0	1	2	3	4	5	6	7	Warp 4
0	1	2	3	4	5	6	7	
0	1	2	3	4	5	6	7	
0	1	2	3	4	5	6	7	

Conflicto de bancos

```
dst[dy][dx] = buf[threadIdx.x][threadIdx.y];
```

- ¡Replay 8 veces!

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

Padding

```
__shared__ int T[8][9];
```

0	1	2	3	4	5	6	7	0
1	2	3	4	5	6	7	0	1
2	3	4	5	6	7	0	1	2
3	4	5	6	7	0	1	2	3
4	5	6	7	0	1	2	3	4
5	6	7	0	1	2	3	4	5
6	7	0	1	2	3	4	5	6
7	0	1	2	3	4	5	6	7

Tratando de cooperar

- Pocas propiedades conocidas sobre la ejecución
  - Cantidad de bloques simultáneos
  - Orden de los bloques
  - Orden de los warps
- La memoria global es la única persistente
  - También es la más lenta

Herramientas que tenemos

- Globales
  - Sincronización implícita entre kernels
  - Atomics en memoria global
- Por bloque
  - Shared memory
  - Atomics en shared memory
  - Barreras (`__syncthreads()`)
- Por warp
  - Ejecución simultánea
  - `shfl()`

Reducción

- La más sencilla de las operaciones colectivas
  - Y probablemente la más usada

- reduce(op, s[]) = s[0] op s[1] op s[2] … s[n]
  - Necesitaremos que op sea asociativo

- Mark Harris, 2007: Optimizing Parallel Reduction in CUDA

Reducción – Atomics

```
__global__ void r(float * in, float * out)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  atomicAdd(out, in[idx]);
}
```

- Tiempo (K40, 8M elementos): 150 ms
- Mark Harris, 2007: Optimizing Parallel Reduction in CUDA

¿150ms, es bueno?

- Problema secuencial O(n)
- Problema paralelo O(n log n)
  - Generalmente memory bound

- Calculemos ancho de banda
  - ((8M (input) + 1 (output)) * 4 (sizeof(float)) / 0.150s
  - = 213 MBps
  - K40: 288 GBps!

Reducción - Atomics en shared

```
__global__ void r(float * in, float * out)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  __shared__ float sum;

  if (threadIdx.x == 0)
    sum = 0.0f;
  __syncthreads();

  atomicAdd(&sum, in[idx]);
  __syncthreads();

  if (threadIdx.x == 0)
    atomicAdd(out, sum);
}
```

- Tiempo: 7 ms, ~4,5 GBps

Reducción – Estrategia

- Atomics serializan la ejecución
  - En particular dentro del bloque
    - blockDim instrucciones por bloque (replays)
  - Desde Kepler, un atomic por bloque es ~aceptable (con bloques que hacen mucho trabajo)
- El operador es asociativo
  - Arbol binario de operaciones
    - $\log_2$(blockDim) instrucciones por hilo por bloque
    - O(log(N))

Reducción - Arbol

```
__shared__ float sum[TAM];
sum[threadIdx.x] = in[idx];
__syncthreads();

for (int s = 1; s < blockDim.x; s *= 2) {
  int pos = 2 * s * threadIdx.x;
  if (pos < blockDim.x) {
    sum[pos] += sum[pos + s];
  }
  __syncthreads();
}
```

- Tiempo: 1 ms, ~32 GBps

Reducción - Arbol

```
(...)

for (int s = blockDim.x/2; s > 0; s /= 2) {
  if (threadIdx.x < s) {
    sum[threadIdx.x] += sum[threadIdx.x + s];
  }
  __syncthreads();
}

(...)
```

- Tiempo: 795 µs, ~40 GBps

Reducción – Menos barreras

```
volatile __shared__ sum[TAM];

for (int s = blockDim.x/2; s > warpSize; s /= 2) {
  if (threadIdx.x < s)
    sum[threadIdx.x] += sum[threadIdx.x + s];
  __syncthreads();
}
#pragma unroll
for (int s = warpSize; s > 0; s /= 2) {
  if (threadIdx.x < s)
    sum[threadIdx.x] += sum[threadIdx.x + s];
  // __syncthreads();   // No hace falta
}
```

- Tiempo: 585 µs, ~54 GBps

Quitando más barreras

- Sean Baxter, 2011: Modern GPU
- El costo de sincronizar es muy alto
- Podemos aplicar la reducción sin barreras de los últimos 64 elementos en los niveles anteriores
- Idea:
  1. Warps reducen 64 → 1, ponen resultado en shared
  2. Barrera (¡única!)
  3. Un warp reduce |warps| → 1
- Algoritmos modernos: división grilla→bloque→warp

Situación circa 2012

- Proliferación de shared memory sin sincronización
  - Propenso a errores
  - Engorroso
  - Limita ocupación
  - Memoria volatile, difícil de optimizar por el compilador

CC 3.0: shfl

- shfl: Intercambio de 32 bits dentro de un warp
  - No requiere memoria compartida
  - Más rápido que memoria compartida sin barreras
- `__shfl(T val, uint desde)`
  - Devuelve val de la lane `desde`
- `__shfl_up/down(T val, uint delta)`
  - Devuelve val de (lane -/+ `delta`) si es válido
- `__shfl_xor(T val, uint mask)`
  - Devuelve val de (lane ^ `mask`) si es válido

Reducción en warp con shfl

```
float val = ...
for (int step = warpSize/2;
     step > 0;
     step /= 2)
{
  val += __shfl_down(val, offset);
}
```

Scan / Prefix sum

- Exclusivo: $s[i] = \sum_{j<i} m[j]$
- Inclusivo: $s[i] = \sum_{j \leq i} m[j]$
- Notar que s[N-1] = reduce(+, s)

Scan – Estrategia

- Es recursiva:
  1. Dividir en segmentos de longitud n $d_0[n]...d_m[n]$
  2. $s_0[n]...s_m[n]$ = Scans de los segmentos
  3. e[m] = scan exclusivo de $s_0[n-1]...s_m[n-1]$
  4. Sumar e[i-1] a los elementos del segmento i-ésimo
- ¿Dónde parar la recursión?
  - Donde tenga un scan rápido con otra estrategia
    - Warp scan

Scan - Usos

- Stream compaction / filter
  - Copiar elementos que satisfacen un predicado
  - Estable
- Partición (stream compaction de pred y ¬pred)
- Radix sort (partición bit a bit desde LSB hasta MSB)
- Evaluación de polinomios, recurrencias, quicksort, sparce matrices...

- G. Blelloch, 1990: Prefix Sums And Their Applications

No reinventar la rueda

- CUDA tiene un ecosistema relativamente maduro
- Existen implementaciones optimizadas de estas primitivas
  - Templates de C++ que toman distintos tipos y operaciones
  - Selección de implementación según arquitectura
  - Escritas por gente de NVIDIA

El menú

- Thrust
  - El más amigable
  - Granularidad: Kernel
- CUB
  - Máxima performance
  - Reduce, scan, select, partition, radix sort, histogram
  - Granularidad: Kernel, bloque, warp
- CUDPP
  - Granularidad: Kernel, bloque, warp
  - Compresión (BWT, MTF), suffix arrays, merge/radix sort, list ranking, string sort, solver tridiagonal

Thrust reduce

```
#include <thrust/reduce.hpp>
#include <thrust/device_ptr.hpp>

float sum(float * dev_in, size_t n) {
  thrust::device_ptr<float> t_in(dev_in);
  return thrust::reduce(t_in, t_in + n);
}
```

- Tiempo: 377 µs, ~85 GBps

CUB block reduce

```
#include <cub/cub.cuh>

__global__ void r(float * in, int n, float * out) {
  typedef cub::BlockReduce<float, 128> BlockReduce;
  __shared__ typename BlockReduce::TempStorage temp;

  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  float data = idx < n ? in[idx] : 0.0f;

  float sum = BlockReduce(temp).Sum(data);
  if (threadIdx.x == 0)
    atomicAdd(out, sum);
}
```

- Tiempo: 502 µs, 64 GBps

CUB device reduce

```
#include <cub/cub.cuh>

void sum(float * in, size_t n, float * out) {
  size_t tempsz;
  cub::DeviceReduce::Sum(NULL, tempsz, in, out, n);
  float * temp;
  cudaMalloc(&temp, tempsz);

  cub::DeviceReduce::Sum(temp, tempsz, in, out, n);
  cudaDeviceSynchronize();
  cudaFree(temp);
}
```

- Tiempo: 186 µs, 172 GBps

Eso es todo por hoy

Mañana:

- Comunicación y múltiples GPU