CUDA

Carlos Bederián, Nicolás Wolovick

FaMAF, Universidad Nacional de Córdoba, Argentina

31 de Julio de 2012

ECAR12@DC.UBA

Revisión 3739, 2012-08-02

Introducción

Manejo de Memoria

Configuración de ejecución

Paralelismo de datos

Comunicación y sincronización

Desempeño y Debugging

Resumen

Modelo de programación

CPU tradicional + Placa aceleradora.

Dos arquitecturas, dos espacios de memoria,
dos filosofías distintas de programación.

Modelo de programación

CPU tradicional + Placa aceleradora.
host device

Dos arquitecturas, dos espacios de memoria, dos filosofías distintas de programación.

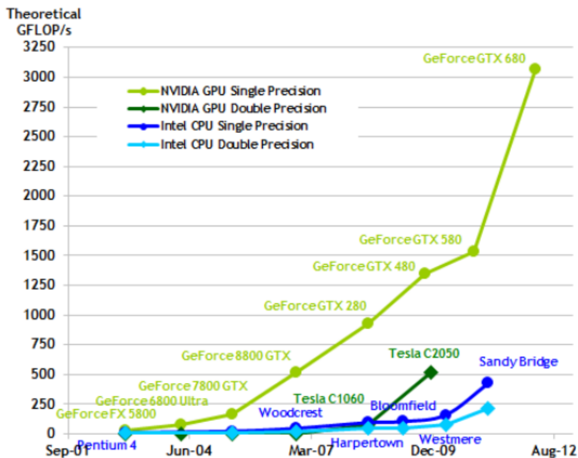
Modelo de programación

CPU tradicional + Placa aceleradora.
host device

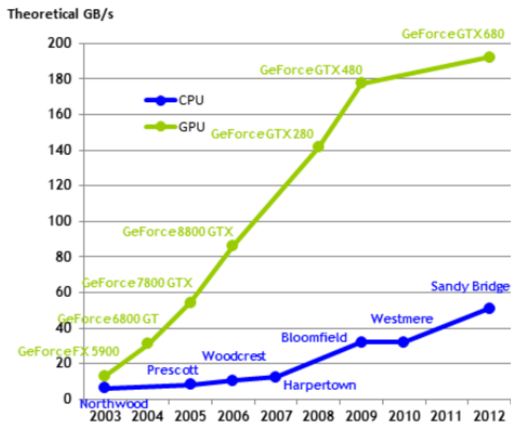
Dos arquitecturas, dos espacios de memoria,
dos filosofías distintas de programación.

¿Porqué?

Potencia de Cálculo



Ancho de Banda de Memoria

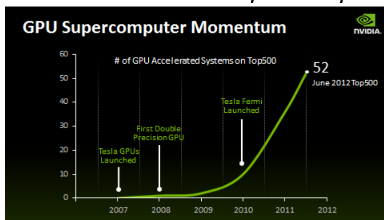


No hay que ser ingenuos

- ¿Cuánto de estos GFLOPS y GBps soy capaz de sacarle yo a mi aplicación?
- ¿Mi aplicación es FLOPS-bound o GBps-bound?
- ¿Podré hacer una corrida de 132 días sin problemas?
- ¿Entra mi problema en la RAM?
- Ahora puedo calcular mucho más ¿Puedo hacer crecer mi problema?
- ¿Cuál es el costo total de pasarse a GPU?
- ¿Es eficiente energéticamente?
- ¿Evacúa mucho calor? ¿Necesito refrigeración con nitrógeno?
- ¿Qué opina la Iglesia?

Indicadores positivos

- *"New Top500 list: 4x more GPU supercomputers"*.



- Intel Xeon Phi.

Contraejemplo: el Top500-nov11 fue liderado por los SPARC64 VIIIfx, y el Top500-jun12 fue liderado por los Power7.

C para CUDA

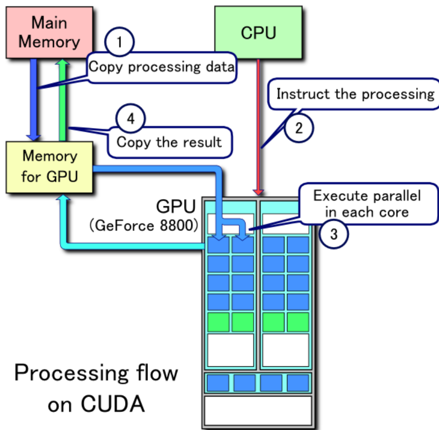
Mezcla de

- Extensiones y restricciones de `C/C++`.
- Tipos de datos especiales.
- Bibliotecas.

Compilado por `nvcc`:

- Genera código CPU: x86_64 (lo hace `gcc`).
- Genera código GPU: PTX.
- Suma código runtime. (Ej: interoperar con el driver)

Flujo de procesamiento



- Copiar datos host $\rightarrow$ device.
- Configurar la ejecución.
- Lanzar kernels.
- Copiar datos host $\leftarrow$ device.

Introducción

Manejo de Memoria

Configuración de ejecución

Paralelismo de datos

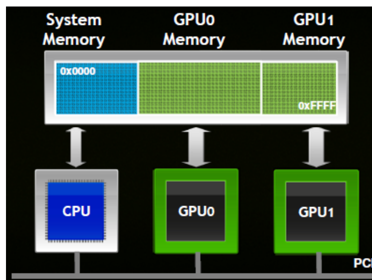
Comunicación y sincronización

Desempeño y Debugging

Resumen

Memoria CPU-GPU

Tenemos Unified Virtual Addressing–UVA (CUDA$\geq$4.0)



- Punteros adornados para identificar CPU, GPU0, GPU1, ...
- Elimina el paso por la CPU: *direct access*, *direct transfer*.

Memoria: pedido

```
#include <stdio.h> // printf
#include <assert.h>
#include <cuda.h> // CUDA library
#include <cutil_inline.h> // CUDA error checking

const unsigned int SIZE=1;

int main(void)
{
    float *h_heat = NULL; // pointer to a float in host memory
    float *d_heat = NULL; // pointer to a float in device memory

    h_heat = (float *) calloc(SIZE, sizeof(float));
    assert(h_heat);
    cutilSafeCall(cudaMalloc(&d_heat, SIZE*sizeof(float)));
    ...
```

- `cutilSafeCall()` revisa errores en llamados a CUDA Lib.
- `assert(p1)` revisa que el puntero en el host no sea nulo.

Memoria: copias y devolución

```
    ...
    *h_heat = 42.0f;
    cutilSafeCall(cudaMemcpy(d_heat, h_heat, SIZE*sizeof(float),
                   cudaMemcpyDefault));
    *h_heat = 0.0f;
    cutilSafeCall(cudaMemcpy(h_heat, d_heat, SIZE*sizeof(float),
                   cudaMemcpyDefault));
    printf("Heat: %f\n", *h_heat);

    cutilSafeCall(cudaFree(d_heat));
    free(h_heat);

    return 0;
}
```

Esto imprime:

Memoria: copias y devolución

```
    ...
    *h_heat = 42.0f;
    cutilSafeCall(cudaMemcpy(d_heat, h_heat, SIZE*sizeof(float),
                  cudaMemcpyDefault));
    *h_heat = 0.0f;
    cutilSafeCall(cudaMemcpy(h_heat, d_heat, SIZE*sizeof(float),
                  cudaMemcpyDefault));
    printf("Heat: %f\n", *h_heat);

    cutilSafeCall(cudaFree(d_heat));
    free(h_heat);

    return 0;
}
```

Esto imprime: 42.

Configuración de ejecución

Queremos lanzar hilos incrementando `d_heat`.

```
__global__ void inc(volatile float *heat) {
    *heat = *heat+1.0f;
}

int main(void)
{
    ...
    inc<<<32768,512>>>(d_heat);
    cutilCheckMsg("inc kernel failed");
    cutilSafeCall(cudaDeviceSynchronize());
    ...
    return 0;
}
```

Son 32768×512 hilos: paralelismo masivo.

Corrección

Resultado correcto

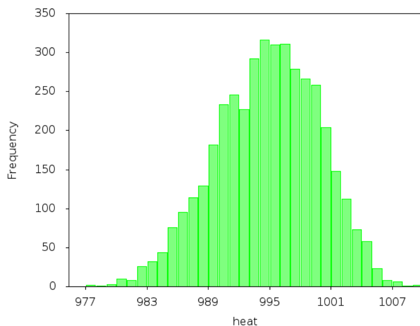
$$\text{heat} = 42 + 32768 \times 512 = 16777258$$

Corrección

Resultado correcto

$$\text{heat} = 42 + 32768 \times 512 = 16777258$$

Histograma, 4096 ejecuciones, Tesla C2070

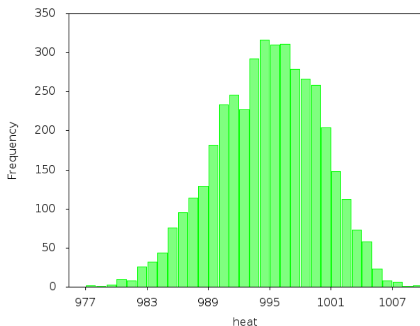


Corrección

Resultado correcto

$$\text{heat} = 42 + 32768 \times 512 = 16777258$$

Histograma, 4096 ejecuciones, Tesla C2070



¿Corrección?

Nodeterminismo

Race conditions, *lost updates*, *memory reorders*, etc.

Los problemas de la concurrencia son muy notables.
(debería dar ≈ 16777258, pero retorna ≈ 1000.)

Lo que ejecuta: Fermi ISA y traducción

```
/*0000*/     /*0x00005de428004404*/     MOV R1, c [0x1] [0x100];
/*0008*/     /*0x80009de428004000*/     MOV R2, c [0x0] [0x20];
/*0010*/     /*0x9000dde428004000*/     MOV R3, c [0x0] [0x24];
/*0018*/     /*0x00201f8584000000*/     LD.E.CV R0, [R2];
/*0020*/     /*0x00001c005000cfe0*/     FADD R0, R0, 0x3f800;
/*0028*/     /*0x00201f8594000000*/     ST.E.WT [R2], R0;
/*0030*/     /*0x00001de780000000*/     EXIT;
```

Nodeterminismo

Race conditions, *lost updates*, *memory reorders*, etc.

Los problemas de la concurrencia son muy notables.
(debería dar ≈ 16777258, pero retorna ≈ 1000.)

Lo que ejecuta: Fermi ISA y traducción

$a_1 := heat$	$a_2 := heat$	$a_3 := heat$	
$a_1 := a_1 + 1.0$	$a_2 := a_2 + 1.0$	$a_3 := a_3 + 1.0$	$\cdots \approx 16M$más
$heat := a_1$	$heat := a_2$	$heat := a_3$	

Nodeterminismo

Race conditions, *lost updates*, *memory reorders*, etc.

Los problemas de la concurrencia son muy notables.
(debería dar ≈ 16777258, pero retorna ≈ 1000.)

Lo que ejecuta: Fermi ISA y traducción

$$
\begin{array}{llll}
a_1 := heat & a_2 := heat & a_3 := heat & \\
a_1 := a_1 + 1.0 & a_2 := a_2 + 1.0 & a_3 := a_3 + 1.0 & \cdots \approx 16M \text{más} \\
heat := a_1 & heat := a_2 & heat := a_3 &
\end{array}
$$

Es una ejecución muy síncrona.

Solución: incrementos atómicos

```
__global__ void atomicinc(float *heat) {
    atomicAdd(heat, 1.0f);
}

int main(void)
{
    ...
    atomicinc<<<32768,512>>>(d_heat);
    ...
}
```

Ahora si

```
$ ./inc
16777258
```

Introducción

Manejo de Memoria

Configuración de ejecución

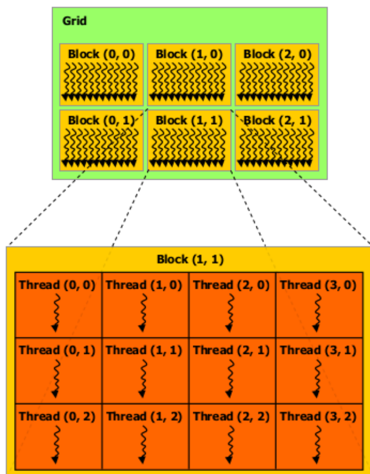
Paralelismo de datos

Comunicación y sincronización

Desempeño y Debugging

Resumen

Configuración de ejecución



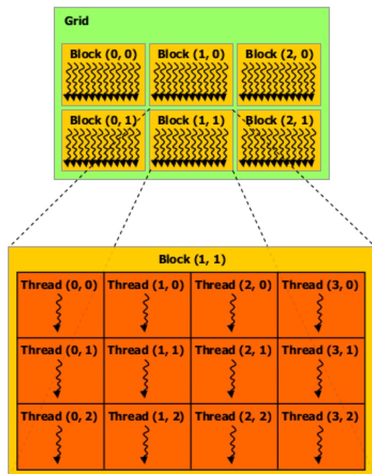
Los hilos se agrupan en bloques.
Los bloques se agrupan en grillas.

Esquema bidimensional para definición de hilos:

$$(threadIdx, blockIdx)$$

¿Porqué?

Configuración de ejecución



Los hilos se agrupan en bloques.
Los bloques se agrupan en grillas.

Esquema bidimensional para definición de hilos:

$$(threadIdx, blockIdx)$$

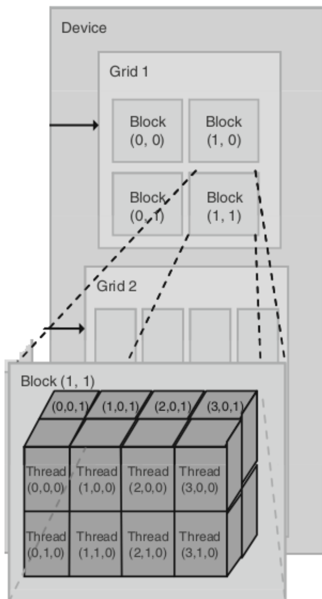
¿Porqué?

Limitar escalabilidad en la comunicación y sincronización.

Restricción

$$threadIdx \leq 1024$$

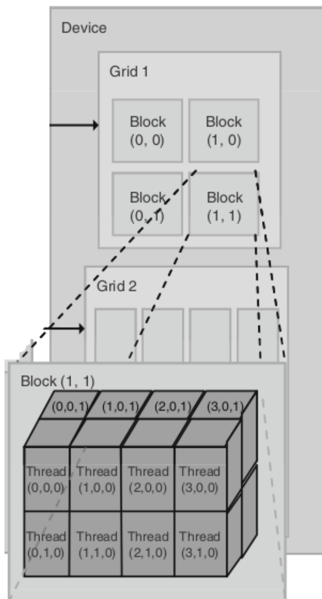
Dimensionalidad de bloques y grillas



Bloques y grillas pueden ser uni, bi o tridimensionales.

¿Para qué?

Dimensionalidad de bloques y grillas



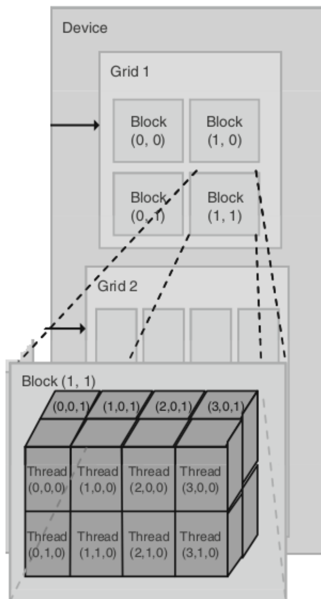
Bloques y grillas pueden ser uni, bi o tridimensionales.

¿Para qué?

Definir fácil el mapa hilo↔dato.

Restricciones

Dimensionalidad de bloques y grillas



Bloques y grillas pueden ser uni, bi o tridimensionales.

¿Para qué?

Definir fácil el mapa hilo↔dato.

Restricciones

$$threadIdx.\{x, y\} \leq 1024$$
$$threadIdx.z \leq 64$$
$$blockIdx.\{x, y, z\} \leq 65536$$

Con estos límites no puedo tener más de 65535 × 1024 *threads unidimensionales.*

Configuración de bloques y grillas

Definir dimensiones de grillas y bloques.

```
dim3 grid_dim(2, 2);
dim3 block_dim(4, 2, 2);
my_kernel<<<grid_dim, block_dim>>>(d_array);
```

`dim3` tripla con *default values* a 1.

`<<<·,·>>>` sintáxis para configuración de ejecución.

Configuración de bloques y grillas

Definir dimensiones de grillas y bloques.

```
dim3 grid_dim(2, 2);
dim3 block_dim(4, 2, 2);
my_kernel<<<grid_dim, block_dim>>>(d_array);
```

`dim3` tripla con *default values* a 1.

<<<·,·>>> sintáxis para configuración de ejecución.

Comparación

MPI Lo usual es unidimensional, opción línea de comandos: `mpirun -np 16`. También hay opciones de mapeos cartesianos: `MPI_CART_CREATE`, `MPI_CART_GET`, `MPI_CART_RANK`.

OMP unidimensional, variable de entorno `OMP_NUM_THREADS=16`, ó `#pragma omp parallel num_threads(16)`.

Identidad de los hilos

- Cada hilo ejecuta el mismo programa: kernel.
- Cada hilo se le asigna un identificador único.
- Tupla $(threadIdx, blockIdx)$ de triplas (x, y, z).

Identidad de los hilos

- Cada hilo ejecuta el mismo programa: kernel.
- Cada hilo se le asigna un identificador único.
- Tupla $(threadIdx, blockIdx)$ de triplas (x, y, z).

Variables especiales

Identificadores `threadIdx.{x,y,z}`, `blockIdx.{x,y,z}`.
Dentro de las dimensiones.

Identidad de los hilos

- Cada hilo ejecuta el mismo programa: kernel.
- Cada hilo se le asigna un identificador único.
- Tupla (*threadIdx*, *blockIdx*) de triplas (x, y, z).

Variables especiales

Identificadores `threadIdx.{x,y,z}`, `blockIdx.{x,y,z}`.
Dentro de las dimensiones.

Dimensiones `blockDim.{x,y,z}`, `gridDim.{x,y,z}`.
Definidas en `kernel<<<grid,block>>>`.

Identidad de los hilos

- Cada hilo ejecuta el mismo programa: kernel.
- Cada hilo se le asigna un identificador único.
- Tupla (*threadIdx*, *blockIdx*) de triplas (x, y, z).

Variables especiales

Identificadores `threadIdx.{x,y,z}`, `blockIdx.{x,y,z}`.
Dentro de las dimensiones.

Dimensiones `blockDim.{x,y,z}`, `gridDim.{x,y,z}`.
Definidas en `kernel<<<grid,block>>>`.

Comparación

MPI `MPI_Comm_size()`, `MPI_Comm_rank()`.

OMP `omp_get_num_threads()`, `omp_get_thread_num()`.

Ejemplo: identificador de thread lineal

Unir la división bloque-grilla para que los 32768×512 hilos accedan a un índice lineal.

```
    ...
    set_vector<<<32768, 512>>>(d_values, d_sum);
    ...
```

Biyección 2d$\leftrightarrow$1d, usando la unicidad de la división de enteros.

```
__global__ void set_vector(float *d_vector) {
    uint i = blockIdx.x*(blockDim.x) + threadIdx.x;
    d_vector[i] = sqrtf(2.0f);
}
```

Ejemplo: vector largo

- Sumar un vector de 128M `floats`.
- Mapeo 1-a-1 de hilos a datos.
- Grilla unidimensional no alcanza: $65535 \times 1024 < 128 \times 2^{20}$.

Ejemplo: vector largo

- Sumar un vector de 128M `floats`.
- Mapeo 1-a-1 de hilos a datos.
- Grilla unidimensional no alcanza: $65535 \times 1024 < 128 \times 2^{20}$.

```
const unsigned int N = 128<<20; // 128M floats
const unsigned int BLOCK_SIZE = 1024; // maximum
const unsigned int BLOCKS = N/BLOCK_SIZE; // >=65535
dim3 block_dim(BLOCK_SIZE); // unidimensional
assert(BLOCKS%1024==0);
dim3 grid_dim(1024, BLOCKS/1024); // bidimensional
add_values<<<grid_dim, block_dim>>>(d_values, d_sum);
```

Ejemplo: vector largo

- Sumar un vector de 128M `floats`.
- Mapeo 1-a-1 de hilos a datos.
- Grilla unidimensional no alcanza: $65535 \times 1024 < 128 \times 2^{20}$.

```
    const unsigned int N = 128<<20; // 128M floats
    const unsigned int BLOCK_SIZE = 1024; // maximum
    const unsigned int BLOCKS = N/BLOCK_SIZE; // >=65535
    dim3 block_dim(BLOCK_SIZE); // unidimensional
    assert(BLOCKS%1024==0);
    dim3 grid_dim(1024, BLOCKS/1024); // bidimensional
    add_values<<<grid_dim, block_dim>>>(d_values, d_sum);
```

En el kernel, lo hacemos lineal

```
__global__ void add_values(float *d_values, float *d_sum) {
    uint tid = threadIdx.x;
    // 2d <-> 1d to overcome 16-bit limit on grid size.
    uint bid = blockIdx.y*(gridDim.x) + blockIdx.x;
    uint i = bid*(blockDim.x) + tid;
    atomicAdd(d_sum, d_values[i]);
    ...
```

Ejemplo: máxima cantidad de hilos

- Mapeo 1-a-1 de hilos a datos.
- Grilla tridimensional:
  $65535^3 \times 1024 = 288217182213504000 \approx 262132 \times 2^{40}$.

Ejemplo: máxima cantidad de hilos

- Mapeo 1-a-1 de hilos a datos.
- Grilla tridimensional:
  $65535^3 \times 1024 = 288217182213504000 \approx 262132 \times 2^{40}$.

```
    ...
    dim3 block_dim(1024); // max threads per block
    dim3 grid_dim(65535, 65535, 65535); // max 3d
    kernel<<<grid_dim, block_dim>>>();
    ...

__global__ void kernel(void) {
    uint tid = threadIdx.x;
    ulong bid = blockIdx.z*(gridDim.x*gridDim.y) +
                blockIdx.y*(gridDim.x) +
                blockIdx.x;
    ulong i = bid*(blockDim.x) + tid;
    ...
```

Notar el uso de `ulong` para tener índices de 64 bits.

Problemas de divisibilidad

La cantidad de hilos siempre es de la forma $grid \times block$.
Si quiero $N = grid \times block + r$, con $0 < r$,
tengo que usar $\lceil \frac{N}{block} \rceil \times block$ y recortar en N.

```
// integer ceiling division
#define DIV_CEIL(a,b) (((a)+(b)-1)/(b))

int main(void)
{
    const unsigned int BLOCK = 512;
    const unsigned int N = 1597; // prime number
    ...
    // 4*512 = 2048 threads
    kernel<<<DIV_CEIL(N,BLOCK), BLOCK>>>(N, d_heat);
    ...
}

__global__ void kernel(const uint N, float *d_values) {
    uint i = blockIdx.x*(blockDim.x) + threadIdx.x;
    if (i<N) // skip from 1597 to 2047
        d_values[i] = 0.0f;
}
```

Para matrices de $2^k \times 2^k$ es directo

```
int main(void)
{
    const unsigned int B = 32;
    const unsigned int L = 4096;
    ...
    assert(L%B==0);
    dim3 grid_dim(L/B, L/B);
    dim3 block_dim(B,B); // 32*32=1024
    reset<<<grid_dim, block_dim>>>(L, d_matrix);
    ...
}

__global__ void reset(uint L, float *d_matrix) {
    uint i = blockIdx.y*(blockDim.y) + threadIdx.y;
    uint j = blockIdx.x*(blockDim.x) + threadIdx.x;
    d_matrix[i*L + j] = 0.0f;
}
```

En estos ejemplos se ve para que sirven este tipo de mapeos.

Introducción

Manejo de Memoria

Configuración de ejecución

Paralelismo de datos

Comunicación y sincronización

Desempeño y Debugging

Resumen

Concurrencia de grano fino

- Podemos (queremos!) lanzar muchísimos hilos ($\sim 1e6$).
- La posibilidad (sensata) es que cada hilo maneje unos pocos datos.
- Paralelismo de datos de grano fino.

Comparación

MPI depende del dinero, supongamos $\sim 1e2$ procesadores.

OMP aproximadamente $\sim 1e1$ procesadores.

Aunque tengamos $\sim 1e6$ procesadores, el overhead hace muy difícil lograr una relación 1-a-1 de hilos a datos.

Ejemplo: FMA4. Device code.

Fused-mutiply and add, 4 operands: $d \leftarrow a \times b + c$

```
#include <stdio.h>
#include <assert.h>
#include <cutil_inline.h>
#include <cuda.h>

__global__ void set(float *a, float *b, float *c, float *d) {
        uint tid = threadIdx.x;
        uint bid = blockIdx.x;
        uint i = bid*blockDim.x + tid;
        a[i] = sinf((float)bid*tid);
        b[i] = cosf((float)bid+tid);
        c[i] = sqrtf((float)bid+tid);
        d[i] = 0.0f;
}

__global__ void fma4(float *a, float *b, float *c, float *d) {
        uint i = blockIdx.x*(blockDim.x) + threadIdx.x;
        d[i] = a[i]*b[i]+c[i];
}
```

Ejemplo: FMA4. Host code

```
int main(void)
{
        const unsigned int N = 1<<25;
        const unsigned int BLOCK = 1024;
        float *da = NULL, *db = NULL, *dc = NULL, *dd = NULL;
        float *ha = NULL, *hb = NULL, *hc = NULL, *hd = NULL;
        // allocate memory
        ha = (float *) calloc(N, sizeof(float));
        hb = (float *) calloc(N, sizeof(float));
        hc = (float *) calloc(N, sizeof(float));
        hd = (float *) calloc(N, sizeof(float));
        assert(ha && hb && hc && hd);
        cutilSafeCall(cudaMalloc(&da, N*sizeof(float)));
        cutilSafeCall(cudaMalloc(&db, N*sizeof(float)));
        cutilSafeCall(cudaMalloc(&dc, N*sizeof(float)));
        cutilSafeCall(cudaMalloc(&dd, N*sizeof(float)));
        // set memory & compute fma4
        assert(N%BLOCK==0);
        set<<<N/BLOCK, BLOCK>>>(da, db, dc, dd);
        cutilCheckMsg("set kernel failed");
        fma4<<<N/BLOCK, BLOCK>>>(da, db, dc, dd);
        cutilCheckMsg("fma4 kernel failed");
        cutilSafeCall(cudaDeviceSynchronize());
```

Ejemplo: FMA4. Host code (cont'd)

```
        // bring back results
        cutilSafeCall(cudaMemcpy(ha, da, N*sizeof(float), cudaMemcpyDefault));
        cutilSafeCall(cudaMemcpy(hb, db, N*sizeof(float), cudaMemcpyDefault));
        cutilSafeCall(cudaMemcpy(hc, dc, N*sizeof(float), cudaMemcpyDefault));
        cutilSafeCall(cudaMemcpy(hd, dd, N*sizeof(float), cudaMemcpyDefault));
        // check against a CPU implementation
        for (unsigned int i=0; i<N; ++i)
                if (hd[i] != ha[i]*hb[i]+hc[i]) {
                        printf("%d, %f!=%f*%f+%f\n", i, hd[i],ha[i],hb[i],hc[i
                        break;
                }

        return 0;
}
```

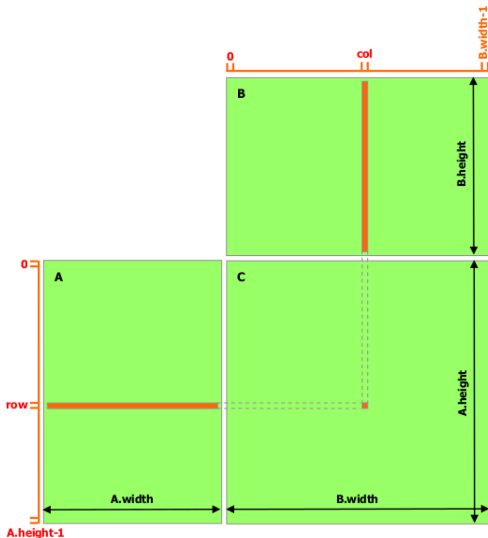
Ejecución

```
$ ./fma4
1045, 3.853793!=0.836656*-0.999961+4.690416
```

Ouch! Las implementaciones de punto flotante no son iguales.

Ejemplo: `sgemm`

Single precision general matrix multiplication.



- Un hilo por celda de la matriz c.
- Cada hilo hace mucho más que unos pocos FLOP.
- No hay concurrencia en la escritura.

Ejemplo: sgemm. Device code.

```
#include <stdio.h> // printf
#include <stdlib.h> // calloc
#include <assert.h> // assert
#include <cuda.h> // CUDA library
#include <cutil_inline.h> // CUDA error checking

// 2D to 1D bijection
#define IX(i,j) ((i)*(N)+(j))
// integer ceiling division
#define DIV_CEIL(a,b) (((a)+(b)-1)/(b))
// MAX is defined in cutil_inline.h


__global__ void setmm(const uint N, float *a, float *b, float *c) {
        const uint i = blockIdx.y*(blockDim.y) + threadIdx.y;
        const uint j = blockIdx.x*(blockDim.x) + threadIdx.x;
        if (i<N && j<N) {
                a[IX(i,j)] = sinf((float)i*j);
                b[IX(i,j)] = cosf((float)i+j);
                c[IX(i,j)] = 0.0f;
        }
}

__global__ void sgemm(const uint N, float *a, float *b, float *c) {
        const uint i = blockIdx.y*(blockDim.y) + threadIdx.y;
        const uint j = blockIdx.x*(blockDim.x) + threadIdx.x;
        if (i<N && j<N)
                for (uint k=0; k<N; ++k)
                        c[IX(i,j)] += a[IX(i,k)] * b[IX(k,j)];
}
```

Ejemplo: sgemm. Host code.

```
int main(int argc, char **argv)
{
        float *d_a = NULL, *h_a = NULL; // pointers to a, b and c in Host and Device
        float *d_b = NULL, *h_b = NULL;
        float *d_c = NULL, *h_c = NULL;
        assert(argc==4);
        const unsigned int N = atoi(argv[1]); // mtx side size
        const unsigned int BX = atoi(argv[2]); // block columns
        const unsigned int BY = atoi(argv[3]); // block rows
        const unsigned int SIZE = N*N; // linear size

        h_a = (float *) calloc(SIZE, sizeof(float));
        h_b = (float *) calloc(SIZE, sizeof(float));
        h_c = (float *) calloc(SIZE, sizeof(float));
        assert(h_a && h_b && h_c);
        cutilSafeCall(cudaMalloc(&d_a, SIZE * sizeof(float)));
        cutilSafeCall(cudaMalloc(&d_b, SIZE * sizeof(float)));
        cutilSafeCall(cudaMalloc(&d_c, SIZE * sizeof(float)));

        dim3 grid_dim(DIV_CEIL(N,BX), DIV_CEIL(N,BY));
        dim3 block_dim(BX,BY);
        setmm<<<grid_dim, block_dim>>>(N, d_a, d_b, d_c);
        cutilCheckMsg("setmm kernel failed");
        sgemm<<<grid_dim, block_dim>>>(N, d_a, d_b, d_c);
        cutilCheckMsg("sgemm kernel failed");
        cutilSafeCall(cudaDeviceSynchronize());
```

Ejemplo: sgemm. Host code (cont'd)

```
        cutilSafeCall(cudaMemcpy(h_a, d_a, SIZE*sizeof(float), cudaMemcpyDefault));
        cutilSafeCall(cudaMemcpy(h_b, d_b, SIZE*sizeof(float), cudaMemcpyDefault));
        cutilSafeCall(cudaMemcpy(h_c, d_c, SIZE*sizeof(float), cudaMemcpyDefault));
        double max_diff = 0.0;
        for (unsigned int i=0; i<N; ++i) {
                for (unsigned int j=0; j<N; ++j) {
                        float cij = 0.0f;
                        for (unsigned int k=0; k<N; ++k)
                                cij += h_a[IX(i,k)] * h_b[IX(k,j)];
                        max_diff = MAX(max_diff, abs(cij-h_c[IX(i,j)]));
                }
        }
        printf("max_diff: %f\n", max_diff);

        cutilSafeCall(cudaFree(d_c));
        cutilSafeCall(cudaFree(d_b));
        cutilSafeCall(cudaFree(d_a));
        free(h_c); free(h_b); free(h_a);

        return 0;
}
```

Ejecución

```
$ ./sgemm 2048 32 32
max_diff: 0.000183
```

Introducción

Manejo de Memoria

Configuración de ejecución

Paralelismo de datos

Comunicación y sincronización

Desempeño y Debugging

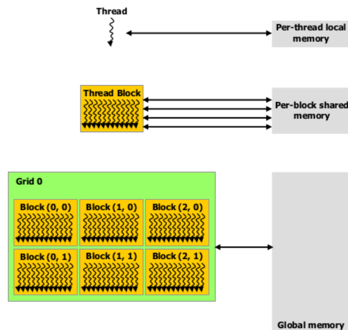
Resumen

Tipos de memoria utilizada

Hasta ahora utilizamos memoria ...

Global

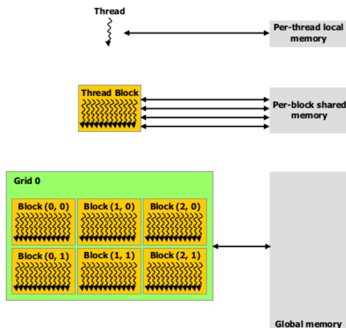
- Cachos de memoria grandes.
- Una para todos los hilos.
- Tiempo de vida: aplicación.
- Ej: `cudaMalloc(&d_a, SIZE * sizeof(float))`.



Tipos de memoria utilizada

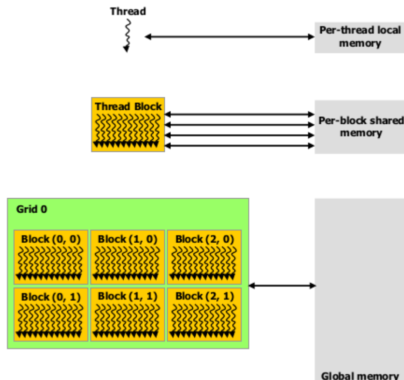
Privada de hilo

- Ahi se almacenan las variables automáticas.
- Una para cada hilo.
- Tiempo de vida: hilo.
- Ej: `uint tid = blockIdx.x`.



Memoria compartida

- Almacenamiento intermedio (rápido).
- Una para un conjunto de hilos: bloque.
- Tiempo de vida: hilos que componen el bloque.
- Ej: `__shared__ float sb[B][B]`.



Permite comunicación intra-bloque.

Sincronización

Los hilos de un bloque se pueden sincronizar por barrera.

```
__syncthreads();
```

threadIdx$\leq$1024, elimina problemas de escalabilidad.

Comparación

MPI `MPI_Barrier(comm)`. Todos los procesos del `comm`.

OMP `#pragma omp barrier`. Todos los hilos del *team*.

Sincronización global

No hay un `global_syncthreads()`.

Cuando veamos el modelo de ejecución esto será obvio.

Se logra con llamadas sucesivas de kernel.

```
set<<<BLOCKS, BLOCK>>>(da, dpartial_sum);
cutilCheckMsg("set kernel failed");
vectorsum<<<BLOCKS, BLOCK>>>(da, dpartial_sum);
cutilCheckMsg("vectorsum kernel failed");
cutilSafeCall(cudaDeviceSynchronize());
```

La semántica de `kernel<<<g,b>>>(parms)` es
`barrier(); kernel<<<g,b>>>(parms)`.
La salida es asíncrona, por eso está `cudaDeviceSynchronize()`.

Comparación

OMP Secuencia de constructores `#pragma omp parallel`.
Donde el último tiene la opción `nowait`.

Uso típico de la memoria compartida

- Los hilos de un bloque cargan cooperativamente la memoria compartida.
- Se opera sobre la memoria compartida.
- El(los) resultado(s) se pasa(n) a la memoria global.

Entre cada una de las etapas, hay que sincronizar por barrera.

Ejemplo: suma de un vector

Sumar un arreglo de a bloques.
Los vectores `float a[N]` y `float partial_sum[N/BLOCK]` están en memoria global.

Entrada

`float *a`

1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0

Salida

`float *partial_sum`

10.0	26.0	42.0

Usar la memoria compartida para que cada bloque acumule allí.

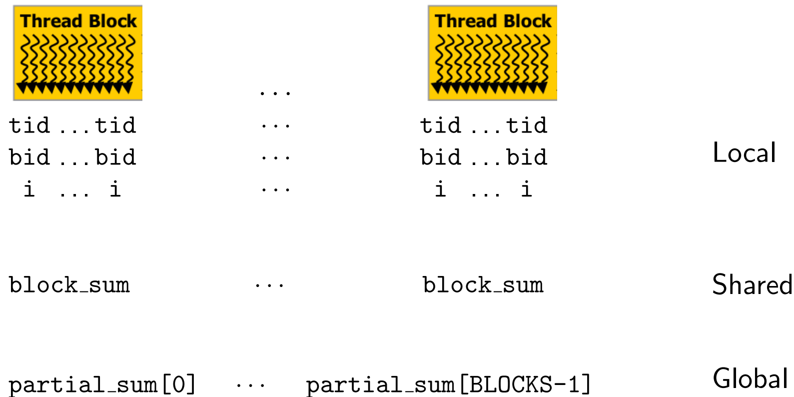
Ejemplo: suma de un vector. Device code.

```
__global__ void vectorsum(float *a, float *partial_sum) {
        uint tid = threadIdx.x;
        uint bid = blockIdx.x;
        uint i = bid*blockDim.x + tid;
        __shared__ float block_sum; // << HERE we use block-shared memory
        // first thread in a block resets shared
        if (tid==0)
                block_sum = 0.0f;
        __syncthreads();
        // atomically accumulate into shared
        atomicAdd(&block_sum, a[i]);
        __syncthreads();
        // copy to partial sum
        if (tid==0)
                partial_sum[bid] = block_sum;
}
```

Ejemplo: suma de un vector. Device code (cont'd).

```
__global__ void set(float *a, float *partial_sum) {
        uint tid = threadIdx.x;
        uint bid = blockIdx.x;
        uint i = bid*blockDim.x + tid;
        a[i] = sinf((float)bid*tid);
        // first blockDim threads initialize
        if (i<blockDim.x)
                partial_sum[i] = 0.0f;
}
```

Ejemplo: suma de un vector. Variables en memoria.



Introducción

Manejo de Memoria

Configuración de ejecución

Paralelismo de datos

Comunicación y sincronización

Desempeño y Debugging

Resumen

Mediciones

CUDA incluye mecanismos de profiling.

- Tiempos de ejecución de kernel y movimientos de memoria.
- Configuración de ejecución: `gridsize`, `threadblocksize`.
- Parámetros de compilación: `dynsmemperblock`, `stasmemperblock`, `regperthread`, `occupancy`.
- Hardware profile counters: `instructions`, `cta_launched`, `gld_inst_128bit`, `branch`, etc.

Más información: NVIDIA, *Compute Command Line Profiler*, *CUDA Tools SDK CUPTI User's Guide*, 2011.

Ejemplo: incremento no-atómico

```
nicolasw@mini:~/ECAR12/Clase1/Inc$ make
nvcc -O3 -arch=sm_20 -I/opt/cudasdk/C/common/inc -o inc.o -c inc.cu
nvcc -O3 -arch=sm_20 -I/opt/cudasdk/C/common/inc -o inc inc.o
nicolasw@mini:~/ECAR12/Clase1/Inc$ export CUDA_PROFILE=1
nicolasw@mini:~/ECAR12/Clase1/Inc$ ./inc
Heat: 995.000000
nicolasw@mini:~/ECAR12/Clase1/Inc$ cat cuda_profile_0.log
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla C2070
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff6c283d7b340
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1.472 ] cputime=[ 6.000 ]
method=[ _Z3incPVf ] gputime=[ 13881.696 ] cputime=[ 6.000 ] occupancy=[ 1.000
method=[ memcpyDtoH ] gputime=[ 2.240 ] cputime=[ 74.000 ]
```

- Está mostrando que el programa realmente funciona.
- Tiempos expresados en μs.
- La medida de occupancy está en el intervalo $[0, 1]$.

Ejemplo: incremento atómico

Ahora si cambiamos el incremento por un incremento atómico.

```
nicolasw@mini:~/ECAR12/Clase1/Inc$ export CUDA_PROFILE=1
nicolasw@mini:~/ECAR12/Clase1/Inc$ ./inc
Heat: 16777216.000000
nicolasw@mini:~/ECAR12/Clase1/Inc$ cat cuda_profile_0.log
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1.408 ] cputime=[ 6.000 ]
method=[ _Z9atomicincPf ] gputime=[ 331406.906 ] cputime=[ 6.000 ] occupancy=[
method=[ memcpyDtoH ] gputime=[ 2.176 ] cputime=[ 73.000 ]
```

Jua! 25 veces más lento.
"*La corrección tiene su precio*".

Buenas Prácticas

Algunas veces la programación en GPUs es parecida a los *embedded devices*. Falla sileciosamente. Quiero que cuando algo falle, explote!.

Programar de manera defensiva

- Utilizar `assert` para comprobar hipótesis:
  - Elecciones de tamaños: `assert(N%BLOCK==0)`.
  - Comprobar que se respetan los límites del hardware: `assert(blockIdx.x<=1024)`.
- Revisar errores: `cutilCheckMsg`, `cutilSafeCall`.
- En lo posible comparar contra una versión "de especificación" (versión confiable secuencial).

  Recordar que en punto flotante las operaciones no son asociativas y las implementaciones de Intel vs. NVIDIA son ligeramente distintas: Intel 80-bit "double extended precision".
- Desconfiar de todo: mirar que la placa trabaje, estudiar profiling, revisar resultados.

Debugging

Hay herramientas maduras:

- Aunque rudimentario, acepta `printf` en los kernels.
- También programación defensiva, `assert` en los kernels.
- Debugger completo: `cuda-gdb`.
- Checkers: `cuda-memcheck`.

Introducción

Manejo de Memoria

Configuración de ejecución

Paralelismo de datos

Comunicación y sincronización

Desempeño y Debugging

Resumen

Resumen

- Presentamos el modelo de programación básico.
- Mostramos algunos problemas de la concurrencia masiva.
- Hicimos hincapié en como definir la división de bloques y grillas y mapeo de datos.
- Vimos como, dentro de un bloque, tenemos más opciones de sincronización.
- Dimos ejemplos simples, pero significativos.
- Aconsejamos como comenter menos errores, y cuando eso sucede, que aparezcan claramente.

Lo que sigue

- Descanso.
- Empecemos a programar.