Optimizaciones

Carlos Bederián, Nicolás Wolovick

FaMAF, Universidad Nacional de Córdoba, Argentina

1 de Agosto de 2012

ECAR12@DC.UBA

Revisión 3742, 2012-08-03

Introducción

Ejemplo: distancias de un vector 4d

Ejemplo: suma de vector de a bloques

Ejemplo: explorando tamaños de bloque

Más cosas

Resumen

Introducción

Vamos a ver algunos ejemplos de optimizaciones típicas

- AoS vs. SoA.
- Atomics sobre globales y sobre shared.
- Exploración de tamaños de bloque.

Quedan muchos temas por ver.
En el práctico y el exámen aumentarán bastante su base de conocimiento.

Introducción

Ejemplo: distancias de un vector 4d

Ejemplo: suma de vector de a bloques

Ejemplo: explorando tamaños de bloque

Más cosas

Resumen

Distancias de un vector

Obtener la norma de una lista de vectores. Es solo un *map*.

Entrada : `float4[N] vectors`.

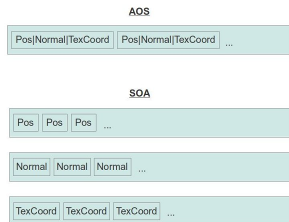
Salida : `float[N] dists`.

Donde $\forall i, dists[i] = |vectors[i]|$.

- Hay una elección importante de estructura de datos.
- ¿Vector de cuatruplas ó cuatrupla de vectores?

AoS vs. SoA

Array of Structures vs. Structure of Arrays.



Fragmento típico de computación con (x, y, z, w), independientemente de su *layout* en memoria:

```
d = x*x + y*y + z*z + w*w;
```

Orden de lectura de un programa

Secuencial : $x, y, z, w, \dots x, y, z, w$

Paralelo SIMD : $x, \dots, x, y, \dots, y, z, \dots, z, w, \dots, w$

Implementaciones

AoS

```
__global__ void dist_aos(float4 *vectors, float *dists) {
        uint tid = threadIdx.x;
        uint bid = blockIdx.y*(gridDim.x) + blockIdx.x;
        uint i = bid*(blockDim.x) + tid;
        float d = sqrtf(vectors[i].x*vectors[i].x +
                        vectors[i].y*vectors[i].y +
                        vectors[i].z*vectors[i].z +
                        vectors[i].w*vectors[i].w);
        dists[i] = d;
}
```

SoA

```
__global__ void dist_soa(float *xs, float *ys, float *zs, float *ws,
                         float *dists) {
        uint tid = threadIdx.x;
        uint bid = blockIdx.y*(gridDim.x) + blockIdx.x;
        uint i = bid*(blockDim.x) + tid;
        float d = sqrtf(xs[i]*xs[i] +
                        ys[i]*ys[i] +
                        zs[i]*zs[i] +
                        ws[i]*ws[i]);
        dists[i] = d;
}
```

Medición

Para $256M$ vectores que consumen el 85 % de los 6GB de la C2070, medimos.

```
$ make
nvcc -O3 -arch=sm_20 -I/opt/cudasdk/C/common/inc -o distances.o -c distances.cu
nvcc -O3 -arch=sm_20 -I/opt/cudasdk/C/common/inc -o distances distances.o
$ ./distances
$ grep dist cuda_profile_0.log
method=[ _Z8dist_aosP6float4Pf ] gputime=[ 59988.449 ] cputime=[ 6.000 ] occupancy=[ 1.000 ]
method=[ _Z8dist_soaPfS_S_S_S_ ] gputime=[ 44686.625 ] cputime=[ 4.000 ] occupancy=[ 1.000 ]
```

- Como era de esperar SoA es mejor que AoS para GPU: 1.34x.
- BW para AoS: $((4\times 4\times 2^{28})/0{,}044686)/2^{30} = 89{,}51 GBps$.

  ¡Bastante bien!

Introducción

Ejemplo: distancias de un vector 4d

Ejemplo: suma de vector de a bloques

Ejemplo: explorando tamaños de bloque

Más cosas

Resumen

Repasando como funciona vectorsum

Entrada: `float a[N]`.
Salida: `float partial_sum[N/BLOCK]`

`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

`float *partial_sum`

10.0	26.0	42.0

Usar la memoria compartida para acumulación atómica rápida.

Dos versiones para comparar velocidad

```
__global__ void vectorsum(float *a, float *partial_sum) {
        uint tid = threadIdx.x;
        uint bid = blockIdx.x;
        uint i = bid*blockDim.x + tid;
        // atomically accumulate into global
        atomicAdd(&partial_sum[bid], a[i]);
}

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

Mediciones

```
$ ./vectorsum && grep vector cuda_profile_0.log
vector_sum maxdiff: 741.610352
vectorsum_shared maxdiff: 0.001038
method=[ _Z9vectorsumPfS_ ] gputime=[ 268549.094 ] cputime=[ 4.000 ] occupancy=[ 0.667 ]
method=[ _Z16vectorsum_sharedPfS_ ] gputime=[ 181925.469 ] cputime=[ 4.000 ] occupancy=[ 0.667 ]
$ ./vectorsum && grep vector cuda_profile_0.log
vector_sum maxdiff: 741.610840
vectorsum_shared maxdiff: 0.000732
method=[ _Z9vectorsumPfS_ ] gputime=[ 268814.625 ] cputime=[ 4.000 ] occupancy=[ 0.667 ]
method=[ _Z16vectorsum_sharedPfS_ ] gputime=[ 181907.844 ] cputime=[ 4.000 ] occupancy=[ 0.667 ]
$ ./vectorsum && grep vector cuda_profile_0.log
vector_sum maxdiff: 741.610718
vectorsum_shared maxdiff: 0.001099
method=[ _Z9vectorsumPfS_ ] gputime=[ 269452.688 ] cputime=[ 5.000 ] occupancy=[ 0.667 ]
method=[ _Z16vectorsum_sharedPfS_ ] gputime=[ 181913.734 ] cputime=[ 4.000 ] occupancy=[ 0.667 ]
```

Notar

- Usar la shared da un 1.47x.
- $2^{25}/(0.268 \times 1e9) = 0.1252 GFLOPS$, puajj!
- $3 \times 4 \times 2^{25}/(0.268 \times 2^{30}) = 1.399 GBps$, puajj!!
- ¡Hay una race condition!
  En realidad es la no-conmutatividad de la suma de `fp32`.
  Solucionarlo sería sumar secuencialmente.

Un detalle sobre la inicialización

```
__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;
}
```

- Elijo uso los primeros `N/BLOCK` hilos para inicializar `partial_sum`,
- Los primeros `N/BLOCK/32` warps trabajan bien con la memoria.
- La otra opción
  `if (tid==0) partial_sum[bid] = 0.0f;`
  desperdicia BW.

Introducción

Ejemplo: distancias de un vector 4d

Ejemplo: suma de vector de a bloques

Ejemplo: explorando tamaños de bloque

Más cosas

Resumen

Explorando tamaños de bloque en sgemm

sgemm simple

- El mismo que mostramos ayer, la versión más sencilla.
- Grilla y bloque bidimensional para $L \times L = 1024 \times 1024$.
- Barremos el rango $[1, 32] \times [1, 32]$ para tamaño de bloque.
- Medimos el tiempo en μs.
- Tesla C2070, driver 295.41, `nvcc-4.2`.

Limitaciones al paralelismo

Limitaciones al paralelismo: registros por hilo

Compilamos con `--ptxas-options=-v`

```
$ nvcc sgemm.cu -O3 -arch=sm_20 --ptxas-options=-v -I/opt/cudasdk/C/common/inc -o sgemm.o
ptxas info    : Compiling entry function '_Z5sgemmjPfS_S_' for 'sm_20'
ptxas info    : Function properties for _Z5sgemmjPfS_S_
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 16 registers, 64 bytes cmem[0], 24 bytes cmem[2]
```

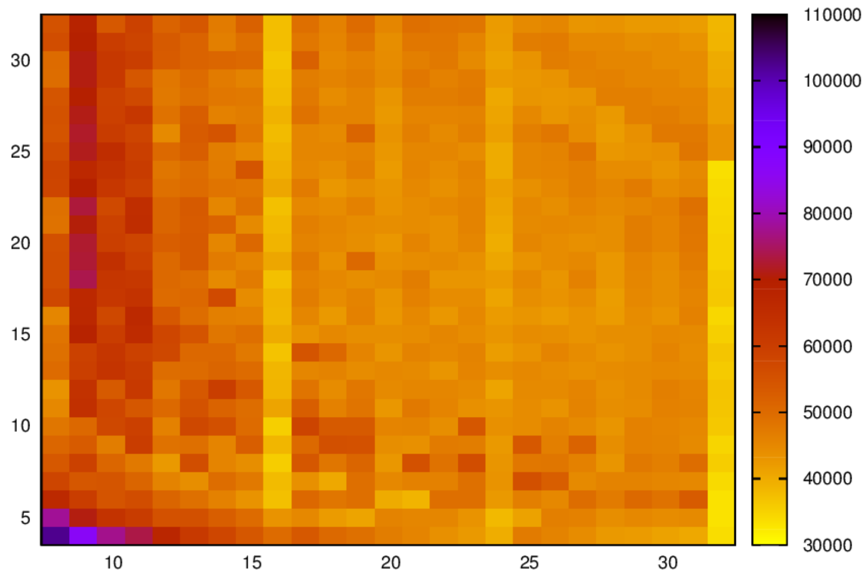
O bien hacemos profiling `regperthread`

```
$ export CUDA_PROFILE=1
$ export CUDA_PROFILE_CONFIG=profile_debug_kernel
$ cat profile_debug_kernel
gridsize
threadblocksize
regperthread
$ ./sgemm 1024 16 16
max_diff: 0.000092
$ cat cuda_profile_0.log
...
method=[ _Z5sgemmjPfS_S_ ] gputime=[ 34407.359 ] cputime=[ 4.000 ]
  gridsize=[ 64, 64 ] threadblocksize=[ 16, 16, 1 ] regperthread=[ 16 ] occupancy=[ 1.000 ]
...
```

No usa `__shared__`, solo 16 registros y $16 \times 1536 \leq 32768$.

No pueden saturar al SM.

Heatmap de velocidad



Notar

- Asimétrico.
- Franjas claras en $bx \in \{16, 24, 32\}$.
- El mejor tiempo está en 32×5, con $33.264ms$.
  Para 2×1024^3 FLOP, implica $60GFLOPS$.
- Muy por debajo de los $\sim 1{,}1 TFLOPS$ `fp32` de la placa.
- Pero el código trivial es casi comparable con una versión SSE+OpenMP en CPU modernas de 8 cores.
- El mínimo en bx=32 no es casualidad:
  $32\times4 \Rightarrow$ warp lee una línea de cache completa.

Occupancy no es performance

Recordemos que para GF100:

warpsPerSM$\leq$48, *blocksPerSM*$\leq$8, granularidad por SM: bloque.

block	thsPerBlock	thsPerSM	blksPerSM	occupancy	gputime
8×8	64	512	8	$\frac{512}{1536}=0.33$	53 ms
16×16	256	1536	6	$\frac{1536}{1536}=1.0$	38 ms
32×32	1024	1024	1	$\frac{1024}{1536}=0.66$	38 ms
8×16	128	1024	8	$\frac{1024}{1536}=0.66$	45 ms
16×8	128	1024	8	$\frac{1024}{1536}=0.66$	35 ms
2×32	64	512	8	$\frac{512}{1536}=0.33$	182 ms
32×2	64	512	8	$\frac{512}{1536}=0.33$	49 ms
32×5	160	1280	8	$\frac{1280}{1536}=0.83$	33 ms

Notar

- `occupancy` no es proporcional a la performance.
- Lo mejor es tener muchos bloques por SM y con buen memory layout.

Introducción

Ejemplo: distancias de un vector 4d

Ejemplo: suma de vector de a bloques

Ejemplo: explorando tamaños de bloque

Más cosas

Resumen

Lo que me hubiera gustado dar

Comentemos un poco.

- Intensidad computacional.
- Branch divergence, predicate computation.
- `#pragma unroll`.
- Deshabilitar las cache.
- Usar templates de C++.
- Las operaciones `fp` e `int` son gratis si el problema es memory-bound.

Introducción

Ejemplo: distancias de un vector 4d

Ejemplo: suma de vector de a bloques

Ejemplo: explorando tamaños de bloque

Más cosas

Resumen

Resumen

- Vimos algunas estrategias de mejor de desempeño.
- Casi siempre, si son acertadas, dan grandes ganancias.
- Es usual ir apilando 2x en cada una. Al final del camino, nuestra performance original era en realidad muy mala.

Lo que sigue

- Mañana y pasado, todo el día de práctico con Carlos.
- Programar mucho, primero `sgemm`, luego `reduce`.